ATA PIO Mode

According to the ATA specs, PIO mode must always be supported by all ATA-compliant drives as the default data transfer mechanism.

PIO mode uses a tremendous amount of CPU resources, because every byte of data transferred between the disk and the CPU must be sent through the CPU's IO port bus (not the memory). On some CPUs, PIO mode can still achieve actual transfer speeds of 16MB per sec, but no other processes on the machine will get any CPU time.

However, when a computer is just beginning to boot there are no other processes. So PIO mode is an excellent and simple interface to utilize during bootup, until the system goes into multitasking mode.

Hardware

The ATA disk specification is built around an older specification called ST506. With ST506, each disk drive was connected to a controller board by two cables -- a data cable, and a command cable. The controller board was plugged into a motherboard bus. The CPU communicated with the controller board through the CPU's IO ports, which were directly connected to the motherboard bus.

What the original IDE specification did was to detach the disk controller boards from the motherboard, and stick one controller onto each disk drive, permanently. When the CPU accessed a disk IO port, there was a chip that shorted the CPU's IO bus pins directly onto the IDE cable -- so the CPU could directly access the drive's controller board. The data transfer mechanism between the CPU and the controller board remained the same, and is now called PIO mode. (Nowadays, the disk controller chips just copy the electrical signals between the IO port bus and the IDE cable, until the drive goes into some other mode than PIO.)

Master/Slave Drives

There is only one wire dedicated to selecting which drive on each bus is active. It is either electrically "high" or "low", which means that there can never be more than two devices operational on any ATA bus. They are called the master and the slave devices, for no particular reason. Their functionality is almost completely identical. There is a special IO port bit that allows a driver to select either drive as the target drive for each command byte.

Primary/Secondary Bus

Current disk controller chips almost always support two ATA buses per chip. There is a standardized set of IO ports to control the disks on the buses. The first two buses are called the Primary and Secondary ATA bus, and are almost always controlled by IO ports 0x1F0 through 0x1F7, and 0x170 through 0x177, respectively (unless you change it). The associated Device Control Registers/Alternate Status ports are IO ports 0x3F6, and 0x376, respectively. The standard IRQ for the Primary bus is IRQ14, and IRQ15 for the Secondary bus.

If the next two buses exist, they are normally controlled by IO ports 0x1E8 through 0x1EF, and 0x168 through 0x16F, respectively. The associated Device Control Registers/Alternate Status ports are IO ports 0x3E6, and 0x366.

The actual control registers and IRQs for each bus can often be determined by enumerating the PCI bus, finding all the disk controllers, and reading the information from each controller's PCI Configuration Space. So, technically, PCI enumeration should be done before ATA device detection. However, this method is not exactly reliable.

When the system boots, according to the specs, the PCI disk controller is supposed to be in "Legacy/Compatibility" mode. This means it is supposed to use the standardized IO port settings. You may have no real choice but to rely on that fact.

400ns delays

The method suggested in the ATA specs for sending ATA commands tells you to check the BSY and DRQ bits before trying to send a command. This means that you need to read a Status Register (Alternate Status is a good choice) *for the proper drive* before sending the next command. Which means that you need to select the correct device *first*, before you can read that status (and then send all the other values to the other IO ports). Which means that a drive select may always happen *just before* a status read. This is bad. Many drives require a little time to respond to a "select", and push their status onto the bus. The suggestion is to read the Status register **FIVE TIMES**, and only pay attention to the value returned by the last one -- after selecting a new master or slave device. The point being that you can assume an IO port read takes approximately 100ns, so doing the first four creates a 400ns delay -- which allows the drive time to push the correct voltages onto the bus.

Reading IO ports to create delays wastes a lot of CPU cycles. So, it is actually smarter to have your driver remember the last value sent to each Drive Select IO port, to avoid doing unneeded drive selections, if the value did not change. If you do not send a drive select, then you only have to read the Status Register **once**.

Alternately, you never want to send new commands to a drive that is already servicing a previous command, anyway. Your driver **always** needs to block if the current device is actively modifying BSY/DRQ/ERR, and your device driver always already knows that the device is in that condition (because the driver just sent the command to the device, and it hasn't been marked "complete" yet). Once a drive has actually completed a command, it will always clear BSY and DRQ. You can simply verify this, *before* your next Device Select command -- that the previously selected device cleared BSY and DRQ properly at command completion. Then you will never have to check if they are clear **after** a Device Select -- so you will not have to read the Status Register after the Device Select at all.

There is a similar problem after writing the Command Register, with the ERR/DF bits. They are two slightly different kinds of errors that can terminate a command. BSY and DRQ will be cleared, but ERR or DF remain set *until just after you write a new command to the Command Register*. If you are using polling (see below), you should account for the fact that your first four reads of the Status Register, after sending your command byte, may have the ERR or DF bits still set accidentally. (If you are using IRQs, the Status will always be correct by the time the IRQ is serviced.)

Cache Flush

On some drives it is necessary to "manually" flush the hardware write cache after every write command. This is done by sending the 0xE7 command to the Command Register (then waiting for BSY to clear). If a driver does not do this, then subsequent write commands can fail invisibly, or "temporary bad sectors" can be created on your disk.

Bad Sectors

For practical purposes, there are three different types of bad sectors on an ATA disk.

- Sectors that can't be written (permanent)
- Sectors that can't be read (permanent)
- Sectors that can't be read (temporary)

Some disk manufacturers have a feature that allows a small supply of "spare" sectors on the disk to be remapped onto permanent bad sectors. However, that feature is non-standard and completely manufacturer-specific. In general, an OS/filesystem will need to keep a "bad sector list" for each partition of each drive, and work around the bad sectors.

As said above, there are also "temporary bad sectors". When you read them you will get a hardware error, just like for a permanently bad sector. If you write to that sector, however, the write will work perfectly and the sector will turn back into a good sector. Temporary bad sectors can happen as a result of unflushed write caches, power spikes, or power failures.

Detection and Initialization

Floating Bus

The disk that was selected last (by the BIOS, during boot) is supposed to maintain control of the electrical values on each IDE bus. If there is no disk connected to the bus at all, then the electrical values on the bus will all go "high" (to +5 volts). A computer will read this as an 0xFF byte -- this is a condition called a "floating" bus. This is an excellent way to find out if there are no drives on a bus. **Before sending any data to the IO ports,** read the Regular Status byte. The value 0xFF is an illegal status value, and indicates that the bus has no drives. The reason to read the port before writing anything is that the act of writing can easily cause the voltages of the wires to go screwy for a millisecond (since there may be nothing attached to the wires to control the voltages!), and mess up any attempt to measure "float".

Measuring "float" is a shortcut for detecting that drives do not exist. Reading a non-0xFF value is not completely definitive. The definitive test for detecting drives is the #IDENTIFY command.

Detecting Controller IO Ports

Detecting controller IO ports is probably a waste of time. During boot, the IO ports assigned to the ATA bus are supposed to be located at standardized addresses. If they are not there, your only chance of finding them is to enumerate the disk controllers on the PCI bus. Alternately, if the ports *are* at the standard addresses, then "detecting" them gains you nothing. However, you will see controller detection code around, and it is a good thing to be able to recognize it, and know what it is for. If you suspect you know the location of a set of ATA controller IO ports, and there is at least one drive attached to that bus, then you can "detect" those IO ports. ATA controller IO ports are mostly read/write ports. This means that if you write a value to (for example) the "SectorCount" IO port, then you are supposed to be able to read the same value back again, to see what it is set to. This read/write function is performed by the master drive on the bus, unless the slave drive both exists and is selected. One caveat is that if you write a value to a "non-existent" IO port, you may be able to read that value back off the bus, if you read it immediately.

So, generally, the way ATA IO port detection software works is to write a byte to one suspected ATA IO port, write a different byte to a different ATA IO port, then read back and verify the two values written to the two ports. If both bytes verify, then the IO ports are read/write ports, and they can be presumed to be ATA controller IO ports. On the Primary bus, ports 0x1F2 through 0x1F5 should all be read/write.

Standard and Non-standard Detection

All current BIOSes have standardized the use of the IDENTIFY command to detect the existence of all types of ATA bus devices ... PATA, PATAPI, SATAPI, SATA.

There are two other nonstandard techniques that are not recommended. The first is to select a device (then do a 400ns delay) then read the device's Status Register. For ATA devices that are not "sleeping", the RDY bit will always be set. This should be detectable, just so long as you have already tested for float (where *all* the bits are always set). If there is no device, then the Status value will be 0. This method does not work for detecting ATAPI devices -- their RDY bit is always **clear** (until they get their first PACKET command).

The other method is to use the Execute Device Diagnostics command (0x90). It supposedly sets bits in the Error Register (0x1F1 on the Primary bus) to show the existence of master and slave devices on the bus.

IDENTIFY command

To use the IDENTIFY command, select a target drive by sending 0xA0 for the master drive, or 0xB0 for the slave, to the "drive select" IO port. On the Primary bus, this would be port 0x1F6. Then set the Sectorcount, LBAlo, LBAmid, and LBAhi IO ports to 0 (port 0x1F2 to 0x1F5). Then send the IDENTIFY command (0xEC) to the Command IO port (0x1F7). Then read the Status port (0x1F7) again. If the value read is 0, the drive does not exist. For any other value: poll the Status port (0x1F7) until bit 7 (BSY, value = 0x80) clears. Because of some ATAPI drives that do not follow spec, at this point you need to check the LBAmid and LBAhi ports (0x1F4 and 0x1F5) to see if they are non-zero. If so, the drive is not ATA, and you should stop polling. Otherwise, continue polling one of the Status ports until bit 3 (DRQ, value = 8) sets, or until bit 0 (ERR, value = 1) sets.

At that point, if ERR is clear, the data is ready to read from the Data port (0x1F0). Read 256 words, and store them.

"Command Aborted"

ATAPI or SATA devices are supposed to respond to an ATA IDENTIFY command by immediately reporting an error in the Status Register, rather than setting BSY, then DRQ, then sending 256 words of PIO data. These devices will also write specific values to the IO ports, that can be read. Seeing ATAPI specific values on those ports after an IDENTIFY is definitive proof that the device is ATAPI -- on the Primary bus, IO port 0x1F4 will read as 0x14, and IO port 0x1F5 will read as 0xEB. If a normal ATA drive should ever happen to abort an IDENTIFY command, the values in those two ports will be 0. A SATA device will report 0x3c, and 0xc3 instead. See below for a code example.

However, at least a few real ATAPI drives do not set the ERR flag after aborting an ATA IDENTIFY command. So do not depend completely on the ERR flag after an IDENTIFY.

Interesting information returned by IDENTIFY

- Word 0: is useful if the device is not a hard disk.
- Word 83: Bit 10 is set if the drive supports LBA48 mode.
- Word 88: The bits in the low byte tell you the supported UDMA modes, the upper byte tells you which UDMA mode is active. If the active mode is not the highest supported mode, you may want to figure out why.
- Word 93 from a master drive on the bus: Bit 12 is supposed to be set if the drive detects an 80 pin cable.
- Words 60 & 61 taken as a DWORD contain the total number of 28 bit LBA addressable sectors on the drive. (If non-zero, the drive supports LBA28.)
- Words 100 through 103 taken as a QWORD contain the total number of 48 bit addressable sectors on the drive. (Probably also proof that LBA48 is supported.)

Addressing Modes

Currently there are three addressing modes to select particular sectors to read or write on a disk. They are 28 bit LBA, 48 bit LBA, and CHS. CHS mode is obsolete, but is discussed quickly below. The number of bits in the LBA modes refer to the number of significant bits in the sector "address", called an LBA. In 28 bit mode, LBAs from 0 to 0x0FFFFFFF are legal. This gives a total of 256M sectors, or 128GB of addressible space. So 28 bit LBA mode is also obsolete for many current drives. However, 28 bit PIO mode is faster than 48 bit addressing, so it may be a better choice for drives or partitions that do not violate the maximum LBA value limitation.

Absolute/Relative LBA

All the ATA commands that use LBA addressing require "absolute" LBAs (ie. the sector offset from the very beginning of the disk -- completely ignoring partition boundaries). At first glance, it might seem most efficient to store the LBA values in this same format in your OS. However, this is not the case. It is always necessary to validate the LBAs that are passed into your driver, as truly belonging to the partition that is being accessed. It ends up being smartest to use partition-relative LBA addressing in your code, because you then never need to test if the LBA being accessed is "off the front" of your current partition. So you only need to do half as many tests. This makes up for the fact that you need to add the absolute LBA of the beginning of the current partition to every "relative" LBA value passed to the driver. At the same time, doing this can give you access to one additional LBA address bit. (See the "33 bit LBA" driver code below.)

Registers

An ATA bus typically has 9 I/O ports that control its behavior. For the primary bus, these I/O ports are 0x1F0 through 0x1F7, and 0x3F6 (see the directions below for usage details). The values in this table are relative to the so-called I/O port base address. So a port value of 1 actually means 0x1F0 + 1 = 0x1F1. This is done because the base address may vary depending on the hardware.

Port Offset	Function	Description
0	Data Port	Read/Write PIO **data** bytes on this port.
1	Features / Error Information	Usually used for ATAPI devices.
2	Sector Count	Number of sectors to read/write (0 is a special value).
3	Sector Number / LBAlo	This is CHS / LBA28 / LBA48 specific.
4	Cylinder Low / LBAmid	Partial Disk Sector address.
5	Cylinder High / LBAhi	Partial Disk Sector address.
6	Drive / Head Port	Used to select a drive and/or head. May supports extra address/flag bits.
7	Command port / Regular Status port	Used to send commands or read the current status.

Status Byte

In the following table you will find the layout of the so-called Status Byte.

Bit	Abbreviation	Function
0	ERR	Indicates an error occurred. Send a new command to clear it (or nuke it with a Software Reset).
3	DRQ	Set when the drive has PIO data to transfer, or is ready to accept PIO data.
4	SRV	Overlapped Mode Service Request.
5	DF	Drive Fault Error (**does not set ERR**).
6	RDY	Bit is clear when drive is spun down, or after an error. Set otherwise.
7	BSY	Indicates the drive is preparing to send/receive data (wait for it to clear). In case of 'hang' (it never clears), do a software reset.

Technically, when BSY is set, the other bits in the Status byte are meaningless. It is also generally a Bad Idea to test the "Seek Complete" (DSC) bit, because it has been deprecated and reused for another purpose.

Device Control Register / Alternate Status

There is an additional IO port that changes the behavior of each ATA bus, called the Device Control Register (on the Primary bus, port 0x3F6). Each ATA bus has its own Control Register. You cannot read the Control Register. Reading the port gets you the value of the Alternate Status Register, instead. The value of Alternate Status is always the same as the Regular Status port (0x1F7 on the Primary bus), but reading the Alternate Status port does not affect interrupts. (See Preempting IRQs, below).

Control Register bit definitions:

Bit	Abbreviation	Function
1	nIEN	Set this to stop the current device from sending interrupts.
2	SRST	Set this to do a "Software Reset" on all ATA drives on a bus, if one is misbehaving.
7	HOB	Set this to read back the High Order Byte of the last LBA48 value sent to an IO port.

All other bits are reserved and should always be clear. In general, you will want to leave HOB, SRST, and nIEN cleared. Set each Device Control Register to 0 once, during boot.

Resetting a drive / Software Reset

For non-ATAPI drives, the only method a driver has of resetting a drive after a major error is to do a "software reset" on the bus. Set bit 2 (SRST, value = 4) in the proper Control Register for the bus. This will reset **both** ATA devices on the bus. Then, you have to clear that bit again, yourself. The master drive on the bus is automatically selected. ATAPI drives set values on their LBA_LOW and LBA_HIGH IO ports, but are not supposed to reset or even terminate their current command.

IRQs

Note: When a command terminates with an error, it does **not** generate an IRQ. It is smart to check the Alternate Status Register a few times per second to see if the ERR bit has set. Otherwise, you will not know until your command times out.

Handling an IRQ

In the early days, the only intent of an IRQ was to inform the IRQ handler that the drive was ready to send or accept data. The expectation was that the IRQ handler itself would perform a PIO based data transfer of the next data block, immediately. Now things are not so simple. One or both of the drives on the bus may be in DMA mode, or have data block sizes other than 256 words. Also, there is more emphasis now on returning as quickly as possible out of the IRQ handler routine. So the question is: what is the minimal set of operations that an IRQ handler needs to do?

If you are using IRQ sharing, you will need to check the PCI Busmaster Status byte, to verify that the IRQ came from the disk. If it did, it is necessary to read the Regular Status Register once, to make the disk clear its interrupt flag. If the ERR bit in the Status Register is set (bit 0, value = 1), you may want to read and save the "error details" value from the Error IO port (0x1F1 on the Primary bus).

If the transfer was a READ DMA operation, you *must* read the value from the Busmaster Status Register. Since the IRQ handler probably doesn't know whether the operation was a DMA operation or not, you will probably end up checking the Busmaster Status byte after all IRQs (if the bus is controlled by a PCI controller at all -- which it almost certainly is). If that byte has its ERR bit set (bit 1, value = 2), you may want to save the current values in the disk's LBA IO ports -- they can tell you which sector on the drive generated the error. You will also need to clear the error bit, by writing a 2 to it.

You will also need to send EOI (0x20) to both PICs, to clear their interrupt flags. Then you need to set a flag to "unblock" the driver, and let it know that another IRQ has occurred -- so the driver can do any necessary data transfer.

Note: if you are still in singletasking mode, and polling the Regular Status Register in PIO mode only, then the only thing the IRQ handler needs to do is send EOI to the PICs. You may even want to set the Control Register's nIEN bit, to try to shut off disk IRQs completely.

Polling the Status vs. IRQs

When a driver issues a PIO read or write command, it needs to wait until the drive is ready before transferring data. There are two ways to know when the drive is ready for the data. The drive will send an IRQ when it is good and ready. Or, a driver can poll one of the Status ports (either the Regular or Alternate Status).

There are two advantages to polling, and one gigantic disadvantage. Advantages: Polling responds more quickly than an IRQ. The logic of polling is much simpler than waiting on an IRQ.

The disadvantage: In a multitasking environment, polling will eat up all your CPU time. However, in singletasking mode this is not an issue (the CPU has nothing better to do) -- so polling is a good thing, then.

How to poll (waiting for the drive to be ready to transfer data): Read the Regular Status port until bit 7 (BSY, value = 0x80) clears, and bit 3 (DRQ, value = 8) sets -- or until bit 0 (ERR, value = 1) or bit 5 (DF, value = 0x20) sets. If neither error bit is set, the device is ready right then.

Preempting/Preventing IRQs from firing:

If a driver ever reads the Regular Status port after sending a command to a drive, the "response" IRQ may never happen. If you *want* to receive IRQs, then always read the Alternate Status port, instead of the Regular Status port. But sometimes IRQs are just wasteful, and it is a good idea to make them go away.

A much more complete way to prevent ATA IRQs from happening is to set the nIEN bit in the Control Register of a particular *selected drive*. This should prevent the drive on the bus from sending any IRQs at all, until you clear the bit again. **However, it may not always work!** Several programmers have reported problems making nIEN work. Drives only respond to *newly written values* of nIEN when they are the selected drive on the bus. That is, if a drive is selected, and you set nIEN, then select the other drive with the Drive Select Register, then clear nIEN -- then the first drive should "remember" forever that it was told not to send IRQs -- until you select it again, and write a 0 to the nIEN bit in the Control Register.

Read/Write Multiple

One way of trying to reduce the number of IRQs in **multitasking** PIO mode is to use the READ MULTIPLE (0xC4), and WRITE MULTIPLE (0xC5) commands. These commands make the drive buffer "blocks" of sectors, and only send one IRQ per block, rather than one IRQ per sector. See Words 47 and 59 of the IDENTIFY command, to determine the number of sectors in a block. You can also try to use the SET MULTIPLE MODE (0xC6) command, to change the sectors per block.

NOTE: Overall, PIO mode is a slow transfer method. Under real working conditions, almost any drive should be controlled by a DMA driver, and should not be using PIO. Trying to speed up PIO mode by preempting IRQs (or any other method) is mostly a waste of time and effort. ATA drives that are 400MB or smaller may not support Multiword DMA mode 0, however. If you want to support drives that size, then perhaps a *little* effort spent on PIO mode drivers is worthwhile.

x86 Directions

28 bit PIO

Assume you have a sectorcount byte and a 28 bit LBA value. A sectorcount of 0 means 256 sectors = 128K.

Notes: When you send a command byte and the RDY bit of the Status Registers is clear, you may have to wait (technically up to 30 seconds) for the drive to spin up, before DRQ sets. You may also need to ignore ERR and DF the first four times that you read the Status, if you are polling.

An example of a 28 bit LBA PIO mode read on the Primary bus:

1. Send 0xE0 for the "master" or 0xF0 for the "slave", ORed with the highest 4 bits of the LBA to port 0x1F6: outb(0x1F6, 0xE0 **|** (slavebit << 4) **|** ((LBA >> 24) & 0x0F))
2. Send a NULL byte to port 0x1F1, if you like (it is ignored and wastes lots of CPU time): outb(0x1F1, 0x00)
3. Send the sectorcount to port 0x1F2: outb(0x1F2, (unsigned char) count)
4. Send the low 8 bits of the LBA to port 0x1F3: outb(0x1F3, (unsigned char) LBA))
5. Send the next 8 bits of the LBA to port 0x1F4: outb(0x1F4, (unsigned char)(LBA >> 8))
6. Send the next 8 bits of the LBA to port 0x1F5: outb(0x1F5, (unsigned char)(LBA >> 16))
7. Send the "READ SECTORS" command (0x20) to port 0x1F7: outb(0x1F7, 0x20)
8. Wait for an IRQ or poll.
9. Transfer 256 words, a word at a time, into your buffer from I/O port 0x1F0. (In assembler, REP INSW works well for this.)
10. Then loop back to waiting for the next IRQ (or poll again -- see next note) for *each successive sector*.

Note for polling PIO drivers: After transferring the last word of a PIO data block to the data IO port, give the drive a 400ns delay to reset its DRQ bit (and possibly set BSY again, while emptying/filling its buffer to/from the drive).

Note on the "magic bits" sent to port 0x1f6: Bit 6 (value = 0x40) is the LBA bit. This must be set for either LBA28 or LBA48 transfers. It must be clear for CHS transfers. Bits 7 and 5 are obsolete for **current** ATA drives, but must be set for backwards compatibility with very old (ATA1) drives.

Writing 28 bit LBA

To write sectors in 28 bit PIO mode, send command "WRITE SECTORS" (0x30) to the Command port. Do **not** use **REP** OUTSW to transfer data. There must be a tiny delay between each OUTSW output word. A jmp $+2 size of delay. Make sure to do a Cache Flush (ATA command 0xE7) after each write command completes.

48 bit PIO

Reading sectors using 48 bit PIO is very similar to the 28 bit method:

(Notes: A sector count of 0 means 65536 sectors = 32MB. Try not to send bytes to the same IO port twice in a row. Doing so is **much** slower than doing two outb() commands to**different** IO ports. The important thing is that the high byte of the sector count, and LBA bytes 4, 5, & 6 go to their respective ports **before the low bytes**.)

Assume you have a sectorcount word and a 6 byte LBA value. Mentally number the LBA bytes as 1 to 6, from low to high. Send the 2 byte sector count to port 0x1F2 (high byte first), and the six LBA byte pairs to ports 0x1F3 through 0x1F5 in some appropriate order.

An example:

1. Send 0x40 for the "master" or 0x50 for the "slave" to port 0x1F6: outb(0x1F6, 0x40 **|** (slavebit << 4))
2. outb (0x1F2, sectorcount high byte)
3. outb (0x1F3, LBA4)
4. outb (0x1F4, LBA5)
5. outb (0x1F5, LBA6)
6. outb (0x1F2, sectorcount low byte)
7. outb (0x1F3, LBA1)
8. outb (0x1F4, LBA2)
9. outb (0x1F5, LBA3)
10. Send the "READ SECTORS EXT" command (0x24) to port 0x1F7: outb(0x1F7, 0x24)

Note on the "magic bits" sent to port 0x1f6: Bit 6 (value = 0x40) is the LBA bit. This must be set for either LBA28 or LBA48 transfers. It must be clear for CHS transfers. Any drive that can support LBA48 will ignore all other bits on this port for an LBA48 command. You can set them if it will make your code cleaner (to use the same magic bits as LBA28).

To write sectors in 48 bit PIO mode, send command "WRITE SECTORS EXT" (0x34), instead. (As before, do not use REP OUTSW when writing.) And remember to do a Cache Flush after each write command completes.

After the command byte is sent, transfer each sector of data in exactly the same way as for a 28 bit PIO Read/Write command.

CHS mode

Cylinder, Head, Sector mode is completely obsolete, but there are a few things to know about it, for legacy reasons.

The oldest drives had many glass "platters", and two read/write "heads" per platter. The heads are always lined up, vertically. One head of one of the platters was usually used for "timing". As all the platters rotated, each head traced out a circle, and all the heads together traced out a "cylinder." Each circle traced out by each head was subdivided into some number of "sectors." Each sector could be used for storing 512 bytes of data. Selecting the Cylinder, Head, and Sector became an addressing mode.

Changing the cylinder meant moving the whole head assembly, which was to be avoided if possible.

But the important point is that none of this information has been true for the last 20 years -- except that computers kept accessing data via artificial CHS addressing.

In CHS mode, every drive has a "geometry" -- the legal ranges for the CHS values. The typical maximum legal values are Cyl = 0 to 1023, Head = 0 to 15, Sector = 1 to 63.

Please note that Sector = 0 is always illegal! That is a common cause of errors. (It is also possible that some hardware / drives will accept Cylinder values as high as 65537.)

Converting CHS addressing to LBA is straightforward: (Cylinder * TotalHeads + SelectedHead) * SectorsPerTrack + (SectorNum - 1). Sometimes programs will ask for a CHS address, and you will need to do that calculation by hand.

Accessing sectors in CHS mode is basically identical to doing 28 bit LBA reads and writes, except that you leave the LBA bit (value = 0x40) turned off when writing the Bit Flags port, and you send various CHS bytes instead of LBA bytes to the IO ports.

An example:

1. Send 0xA0 for the "master" or 0xB0 for the "slave", ORed with the Head Number to port 0x1F6: outb(0x1F6, 0xA0 **|** (slavebit << 4) **|** Head Number)
2. outb (0x1F2, bytecount/512 = sectorcount)
3. outb (0x1F3, Sector Number -- the S in CHS)
4. outb (0x1F4, Cylinder Low Byte)
5. outb (0x1F5, Cylinder High Byte)
6. Send the "READ SECTORS" command (0x20) to port 0x1F7: outb(0x1F7, 0x20)

To write, send command "WRITE SECTORS" (0x30).

Note on the "magic bits" sent to port 0x1f6: Bit 6 (value = 0x40) is the LBA bit. This must be clear for CHS transfers, as said above. Bits 7 and 5 are obsolete for **current** ATA drives, but must be set for backwards compatibility with very old (ATA1) drives.

x86 Code Examples

Detecting device types

(Using a Software Reset -- adapted from PypeClicker)

```
/*on Primary bus: ctrl->base=0x1F0, ctrl->dev_ctl=0x3F6. REG_CYL_LO=4, REG_CYL_HI=5, REG_DEVSEL=6 */
int detect_devtype (int slavebit, struct DEVICE *ctrl)
{
        ata_soft_reset(ctrl->dev_ctl);    /* waits until master drive is ready again */
        outb(ctrl->base + REG_DEVSEL, 0xA0 | slavebit<<4);
        inb(ctrl->dev_ctl);               /* wait 400ns for drive select to work */
        inb(ctrl->dev_ctl);
        inb(ctrl->dev_ctl);
        inb(ctrl->dev_ctl);
        unsigned cl=inb(ctrl->base + REG_CYL_LO);  /* get the "signature bytes" */
        unsigned ch=inb(ctrl->base + REG_CYL_HI);

        /* differentiate ATA, ATAPI, SATA and SATAPI */
        if (cl==0x14 && ch==0xEB) return ATADEV_PATAPI;
        if (cl==0x69 && ch==0x96) return ATADEV_SATAPI;
        if (cl==0 && ch == 0) return ATADEV_PATA;
        if (cl==0x3c && ch==0xc3) return ATADEV_SATA;
        return ATADEV_UNKNOWN;
}
```

ATA Driver

A complete singletasking (polling) PIO mode driver for reading a hard disk (Note: the following routines should all include some form of OS-specific timeout.)

```
; do a singletasking PIO ATA read
; inputs: ebx = # of sectors to read, edi -> dest buffer, esi -> driverdata struct, ebp = 4b LBA
; Note: ebp is a "relative" LBA -- the offset from the beginning of the partition
; outputs: ebp, edi incremented past read; ebx = 0
; flags: zero flag set on success, carry set on failure (redundant)
read_ata_st:
          push edx
          push ecx
          push eax
          test ebx, ebx               ; # of sectors < 0 is a "reset" request from software
          js short .reset
          cmp ebx, 0x3fffff           ; read will be bigger than 2GB? (error)
          stc
          jg short .r_don
          mov edx, [esi + dd_prtlen]  ; get the total partition length (sectors)
          dec edx                               ; (to avoid handling "equality" case)
          cmp edx, ebp                          ; verify ebp is legal (within partition limit)
          jb short .r_don                       ; (carry is set automatically on an error)
          cmp edx, ebx                ; verify ebx is legal (forget about the ebx = edx case)
          jb short .r_don
          sub edx, ebx                          ; verify ebp + ebx - 1 is legal
          inc edx
          cmp edx, ebp                          ; (the test actually checks ebp <= edx - ebx + 1)
          jb short .r_don
          mov dx, [esi + dd_dcr]                ; dx = alt status/DCR
          in al, dx                   ; get the current status
          test al, 0x88               ; check the BSY and DRQ bits -- both must be clear
          je short .stat_ok
.reset:
          call srst_ata_st
          test ebx, ebx                         ; bypass any read on a "reset" request
          jns short .stat_ok
          xor ebx, ebx                          ; force zero flag on, carry clear
          jmp short .r_don
.stat_ok:
; preferentially use the 28bit routine, because it's a little faster
; if ebp > 28bit or esi.stLBA > 28bit or stLBA+ebp > 28bit or stLBA+ebp+ebx > 28bit, use 48 bit
          cmp ebp, 0xfffffff
          jg short .setreg
          mov eax, [esi + dd_stLBA]
          cmp eax, 0xfffffff
          jg short .setreg
          add eax, ebp
          cmp eax, 0xfffffff
          jg short .setreg
          add eax, ebx
          cmp eax, 0xfffffff
.setreg:
          mov dx, [esi + dd_tf]                 ; dx = IO port base ("task file")
          jle short .read28           ; test the flags from the eax cmp's above
.read48:
          test ebx, ebx               ; no more sectors to read?
          je short .r_don
          call pio48_read             ; read up to 256 more sectors, updating registers
          je short .read48   ; if successful, is there more to read?
          jmp short .r_don
.read28:
          test ebx, ebx               ; no more sectors to read?
          je short .r_don
          call pio28_read             ; read up to 256 more sectors, updating registers
          je short .read28   ; if successful, is there more to read?
.r_don:
          pop eax
          pop ecx
          pop edx
          ret
```

```
;ATA PI0 28bit singletasking disk read function (up to 256 sectors)
; inputs: ESI -> driverdata info, EDI -> destination buffer
; BL = sectors to read, DX = base bus I/O port (0x1F0, 0x170, ...), EBP = 28bit "relative" LBA
; BSY and DRQ ATA status bits must already be known to be clear on both slave and master
; outputs: data stored in EDI; EDI and EBP advanced, EBX decremented
; flags: on success Zero flag set, Carry clear
pio28_read:
          add ebp, [esi + dd_stLBA]    ; convert relative LBA to absolute LBA
          mov ecx, ebp                          ; save a working copy
          mov al, bl                   ; set al= sector count (0 means 256 sectors)
          or dl, 2           ; dx = sectorcount port -- usually port 1f2
          out dx, al
          mov al, cl                   ; ecx currently holds LBA
          inc edx                      ; port 1f3 -- LBAlow
          out dx, al
          mov al, ch
          inc edx                      ; port 1f4 -- LBAmid
          out dx, al
          bswap ecx
          mov al, ch                   ; bits 16 to 23 of LBA
          inc edx                      ; port 1f5 -- LBAhigh
          out dx, al
          mov al, cl                            ; bits 24 to 28 of LBA
          or al, byte [esi + dd_sbits] ; master/slave flag | 0xe0
          inc edx                               ; port 1f6 -- drive select
          out dx, al

          inc edx                      ; port 1f7 -- command/status
          mov al, 0x20                 ; send "read" command to drive
          out dx, al

; ignore the error bit for the first 4 status reads -- ie. implement 400ns delay on ERR only
; wait for BSY clear and DRQ set
          mov ecx, 4
.lp1:
          in al, dx          ; grab a status byte
          test al, 0x80                ; BSY flag set?
          jne short .retry
          test al, 8                   ; DRQ set?
          jne short .data_rdy
.retry:
          dec ecx
          jg short .lp1
; need to wait some more -- loop until BSY clears or ERR sets (error exit if ERR sets)

.pior_l:
          in al, dx          ; grab a status byte
          test al, 0x80                ; BSY flag set?
          jne short .pior_l  ; (all other flags are meaningless if BSY is set)
          test al, 0x21                ; ERR or DF set?
          jne short .fail
.data_rdy:
; if BSY and ERR are clear then DRQ must be set -- go and read the data
          sub dl, 7          ; read from data port (ie. 0x1f0)
          mov cx, 256
          rep insw           ; gulp one 512b sector into edi
          or dl, 7           ; "point" dx back at the status register
          in al, dx          ; delay 400ns to allow drive to set new values of BSY and DRQ
          in al, dx
          in al, dx
          in al, dx

; After each DRQ data block it is mandatory to either:
; receive and ack the IRQ -- or poll the status port all over again

          inc ebp                      ; increment the current absolute LBA
          dec ebx                      ; decrement the "sectors to read" count
          test bl, bl                  ; check if the low byte just turned 0 (more sectors to read?)
          jne short .pior_l

          sub dx, 7          ; "point" dx back at the base IO port, so it's unchanged
          sub ebp, [esi + dd_stLBA]    ; convert absolute lba back to relative
; "test" sets the zero flag for a "success" return -- also clears the carry flag
          test al, 0x21                ; test the last status ERR bits
          je short .done
.fail:
          stc
.done:
          ret
```

```
;ATA PIO 33bit singletasking disk read function (up to 64K sectors, using 48bit mode)
; inputs: bx = sectors to read (0 means 64K sectors), edi -> destination buffer
; esi -> driverdata info, dx = base bus I/O port (0x1F0, 0x170, ...), ebp = 32bit "relative" LBA
; BSY and DRQ ATA status bits must already be known to be clear on both slave and master
; outputs: data stored in edi; edi and ebp advanced, ebx decremented
; flags: on success Zero flag set, Carry clear
pio48_read:
          xor eax, eax
          add ebp, [esi + dd_stLBA]     ; convert relative LBA to absolute LBA
; special case: did the addition overflow 32 bits (carry set)?
          adc ah, 0                     ; if so, ah = LBA byte #5 = 1
          mov ecx, ebp                            ; save a working copy of 32 bit absolute LBA

; for speed purposes, never OUT to the same port twice in a row -- avoiding it is messy but best
;outb (0x1F2, sectorcount high)
;outb (0x1F3, LBA4)
;outb (0x1F4, LBA5)                     -- value = 0 or 1 only
;outb (0x1F5, LBA6)                     -- value = 0 always
;outb (0x1F2, sectorcount low)
;outb (0x1F3, LBA1)
;outb (0x1F4, LBA2)
;outb (0x1F5, LBA3)
          bswap ecx           ; make LBA4 and LBA3 easy to access (cl, ch)
          or dl, 2            ; dx = sectorcount port -- usually port 1f2
          mov al, bh                    ; sectorcount -- high byte
          out dx, al
          mov al, cl
          inc edx
          out dx, al                    ; LBA4 = LBAlow, high byte (1f3)
          inc edx
          mov al, ah                    ; LBA5 was calculated above
          out dx, al                    ; LBA5 = LBAmid, high byte (1f4)
          inc edx
          mov al, 0           ; LBA6 is always 0 in 32 bit mode
          out dx, al                    ; LBA6 = LBAhigh, high byte (1f5)

          sub dl, 3
          mov al, bl                    ; sectorcount -- low byte (1f2)
          out dx, al
          mov ax, bp                    ; get LBA1 and LBA2 into ax
          inc edx
          out dx, al                    ; LBA1 = LBAlow, low byte (1f3)
          mov al, ah                    ; LBA2
          inc edx
          out dx, al                    ; LBA2 = LBAmid, low byte (1f4)
          mov al, ch                    ; LBA3
          inc edx
          out dx, al                    ; LBA3 = LBAhigh, low byte (1f5)

          mov al, byte [esi + dd_sbits]           ; master/slave flag | 0xe0
          inc edx
          and al, 0x50                  ; get rid of extraneous LBA28 bits in drive selector
          out dx, al                    ; drive select (1f6)

          inc edx
          mov al, 0x24                  ; send "read ext" command to drive
          out dx, al                    ; command (1f7)

; ignore the error bit for the first 4 status reads -- ie. implement 400ns delay on ERR only
; wait for BSY clear and DRQ set
          mov ecx, 4
.lp1:
          in al, dx           ; grab a status byte
          test al, 0x80                 ; BSY flag set?
          jne short .retry
          test al, 8                    ; DRQ set?
          jne short .data_rdy
.retry:
          dec ecx
          jg short .lp1
; need to wait some more -- loop until BSY clears or ERR sets (error exit if ERR sets)

.pior_l:
          in al, dx           ; grab a status byte
          test al, 0x80                 ; BSY flag set?
          jne short .pior_l   ; (all other flags are meaningless if BSY is set)
          test al, 0x21                 ; ERR or DF set?
          jne short .fail
```

```
.data_rdy:
; if BSY and ERR are clear then DRQ must be set -- go and read the data
          sub dl, 7          ; read from data port (ie. 0x1f0)
          mov cx, 256
          rep insw           ; gulp one 512b sector into edi
          or dl, 7           ; "point" dx back at the status register
          in al, dx          ; delay 400ns to allow drive to set new values of BSY and DRQ
          in al, dx
          in al, dx
          in al, dx

; After each DRQ data block it is mandatory to either:
; receive and ack the IRQ -- or poll the status port all over again

          inc ebp                      ; increment the current absolute LBA (overflowing is OK!)
          dec ebx                      ; decrement the "sectors to read" count
          test bx, bx                  ; check if "sectorcount" just decremented to 0
          jne short .pior_l

          sub dx, 7          ; "point" dx back at the base IO port, so it's unchanged
          sub ebp, [esi + dd_stLBA]    ; convert absolute lba back to relative
; this sub handles the >32bit overflow cases correcty, too
; "test" sets the zero flag for a "success" return -- also clears the carry flag
          test al, 0x21                ; test the last status ERR bits
          je short .done
.fail:
          stc
.done:
          ret


; do a singletasking PIO ata "software reset" with DCR in dx
srst_ata_st:
          push eax
          mov al, 4
          out dx, al                             ; do a "software reset" on the bus
          xor eax, eax
          out dx, al                             ; reset the bus to normal operation
          in al, dx                    ; it might take 4 tries for status bits to reset
          in al, dx                    ; ie. do a 400ns delay
          in al, dx
          in al, dx
.rdylp:
          in al, dx
          and al, 0xc0                           ; check BSY and RDY
          cmp al, 0x40                           ; want BSY clear and RDY set
          jne short .rdylp
          pop eax
          ret
```