Floppy Disk Controller

From OSDev Wiki

Contents

Overview and Documentation

The Floppy Disk Controller (FDC) is a (legacy) device that controls **internal** 3.5/5.25 inch floppy disk drive devices on desktop x86 systems. There are a range of chips that have been produced for this function which include: 8272A, 82078, 82077SL & 82077AA. The 82077AA is the most advanced, and has been produced since 1991. For more recent systems, a model of that chip has been embedded in the motherboard chipset. (So pay close attention to that datasheet, below.)

Accessing Floppies in Real Mode

For bootloaders or OSes that run with the CPU remaining in Real Mode, use BIOS Function INT13h AH=2 (read) or AH=3 (write) to access floppy drives. You need to know the "drive number" (typically 0 or 1), and put that value in DL. More detailed info can be found in the ATA in x86 RealMode (BIOS) article, because accessing a floppy is identical to accessing a hard disk (using CHS) in Real Mode. The rest of this article deals with creating Protected Mode drivers for the floppy subsystem.

Note: the Extended BIOS Int13h functions do not work with floppies.

Note2: the BIOS IRQ0 handler remembers a timeout for turning the motor off, from the last BIOS floppy access. The last access may have been attempting to load your bootloader. So, in the distant future, if the BIOS ever receives 36 more IRQ0 ticks (if you ever return to Real Mode) it may turn off your floppy motors for you, once.

Accessing USB Floppy Drives

All USB devices, including USB floppy drives, are accessed indirectly (using SCSI-style commands encoded in USB datapackets) over the USB bus. USB floppy drives do not use any of the IO ports or FDC commands described in this article.

The Floppy Subsystem is Ugly

Many of the devices that an OS controls in an x86 system have had functional patches added to them over the years. This makes them all rather unpleasant when coding drivers. The floppy subsystem is probably the worst. As the functionality evolved, some of the bit definitions were actually **reversed** in meaning, not merely made obsolete. Several commands require duplicating the same information in two different locations -- and if the duplicated info doesn't match, the command fails. There were never definitive flags added to identify which "mode" the controller is operating in. Also, the floppy subsystem is the primary remaining one that accesses the obsolete and terrible ISA DMA system for its data transfers.

How Many Controllers? One!

Each ribbon cable for floppy drives can support 2 drives. One floppy controller chip can control 2 floppy cables, for a total of 4 drives. It is theoretically possible for a system to have more than one controller chip, but you will not find any existing systems with more than one. If a system were to have more than one floppy controller, the second controller would be found at a base IO port address of 0x370.

How Many Drives?

It is probably wisest to always get the drive count and types from CMOS, register 0x10.

The Actual Drive

In the olden days, there used to be 5.25 inch low-density, high-density, and single-sided drives. There used to be media for each of these drive types. There also used to be 3.5 inch low density media. None of this really exists anymore. The only actual hardware you will typically run into now is 3.5 inch, 1.44MB drives.

A 1.44MB floppy disk is usually formatted with 80 cylinders, 2 sides, and 18 sectors per track. The drive records the two sides with two "heads" that are bolted together. They cannot seek independently. The "track" on one side of a disk is always exactly opposite the track on the other side of the disk. (There is a misconception about this in many floppy driver code examples.) You only need to seek one head to find a particular cylinder, so that both heads may read/write that cylinder.

CHS

Floppy drives use CHS addressing exclusively. There are always 2 heads (sides), but the driver (and controller) must also know how many cylinders and sectors per track the media expects. Typically, as said above, it is 80 cylinders and 18 sectors per track. In usual CHS fashion, the cylinders and heads are counted starting with a 0 base, but the sector numbers start counting from 1. That is, legal cylinder numbers are typically 0 to 79 inclusive, heads are 0 or 1, and sector numbers are 1 to 18 inclusive. Asking for sector number 0 is always highly illegal and this is a major source of errors in prototype driver code.

DMA Data Transfers

The floppy typically uses ISA DMA (which is **not** the same thing as PCI BusMastering DMA) to do data transfers. The floppy is hardwired to DMA channel 2. The only other way of doing data transfers is called "PIO Mode" (see below).

To do DMA data transfers: Set up DMA channel 2 (as in the DMA article) -- by setting the total transfer bytecount - 1, the target buffer physical address, and the transfer direction.

init/reset the controller if needed (see below), select the drive if needed (see below), set up the floppy controller for DMA using the "specify" command, seek to the correct cylinder, issue a sense interrupt command, then issue the standard read/write commands. The data transfer happens invisibly. The controller will send an IRQ6 when the transfer is complete. Then read the "result" bytes to see if there were any errors. See below for more detail.

PIO Data Transfers

PIO data transfers can either be done using polling or interrupts. Using PIO mode can transfer data 10 percent faster than a DMA transfer, but there is an immense cost in CPU cycle requirements. However, if your OS or application is single-tasking, then there is nothing else that the CPU cycles can be used for anyway, so you may as well use PIO mode.

In general, the controller has a 16 byte buffer, and wants to send an IRQ6 whenever the buffer count reaches a "threshold" value that is set by your driver. If you are using PIO mode floppy transfers in a multitasking environment (bad idea), then the IRQ6 events should be used to fill or drain the floppy controller's buffer via some system buffer, in the interrupt handler code.

If you are using PIO mode in a singletasking environment then the IRQ6s just waste CPU cycles, and you should be using polling instead. You can usually shut the IRQs off by setting bit 3 (value = 8) of the Digital Output Register (see below). If you want, you can even toggle that bit on and off for specific commands, so that you receive some types of interrupts but not others.

To do PIO data transfers: init/reset the controller if needed (see below), select the drive if needed (see below), seek to the correct cylinder, issue a sense interrupt command, then issue the standard read/write commands. After you send the command, either poll the RQM bit in the Main Status Register to determine when the controller wants to have data moved in/out of the FIFO buffer -- or wait for an IRQ6 to do the same thing. When the transfer is complete, read the "result" bytes to see if there were any errors. See below for more detail.

Bochs Can't Handle Polling PIO

If you try to turn off the IRQs in Bochs (to use pure polling PIO mode), Bochs will panic. It may also not be able to handle non-DMA data transfers properly, even if IRQs are turned on. It is currently mainly designed to model the floppy using DMA.

There are 3 "Modes"

There were several generations of floppy controller chips on several generations of 80286 (and prior) computers, before the 82077AA chip existed. The 82077AA was built to emulate all of them, by setting various pins on the chip to 5 volts. The three modes are: PC-AT mode, PS/2 mode, and Model 30 mode. The most likely mode you will see on any hardware that still runs is Model 30 mode. You may find some pre-1996 Pentium machines using PS/2 mode. Sadly, most emulator programs run in PS/2 mode! In the documentation, you can ignore PC-AT mode. Or you can try to handle all three, by only using registers and commands that are identical in all 3 modes.

Most Commands Run "Silently"

There are only a few commands/conditions that produce interrupts: a reset (in polling mode only), Seek, Recalibrate, or one of the read/write/verify/format commands. Several commands do not produce any result bytes, either. If you want to verify that a silent command actually worked, just about the only

thing you can do is use the Dumpreg command to check the current state of the controller.

Timing Issues

On real hardware, there are definite timing issues. Seek delays and motor delays are just what any programmer would expect. It would be nice if the drive would send an IRQ when the motor was up to speed, but it does not. Two things that you may not expect are that quite new hardware probably still needs artificial delays between outputting "command/parameter" bytes, and that you probably also need artificial delays between inputting "result" bytes. There is a bit in the MSR to test in order to know when the next byte can be sent/retrieved. It is not a good idea to simply hardcode specific delays between output/input bytes. Looping 20 times, and testing the value of the RQM bit in the MSR each time, should always be a sufficient "timeout".

However, using IO Port reads to generate delays (or polling MSR) leads to poor performance in a multitasking environment, and you may want to put the driver to sleep for the shortest possible period (microsleep), instead.

Reliability

In real hardware, floppy drives are extremely unreliable. On emulators, it doesn't matter -- but for code that is intended for real hardware, make sure to retry every command at least twice more after any error.

Registers

The floppy controller is programmed through 9 registers, which can be accessed through IO ports 0x3F0 through 0x3F7 (excluding 0x3F6). As usual on the PC architecture, some of those IO ports access different controller registers depending on whether you read from or write to them. Note that code snippets and datasheets name these registers based on their trigrams (e.g. SRA, MSR, DIR, CCR, etc.). Use standard "outb" and "inb" commands to access the registers.

The basic set of floppy registers can be found in the following enumeration:

```
enum FloppyRegisters
{
   STATUS_REGISTER_A                = 0x3F0, // read-only
   STATUS_REGISTER_B                = 0x3F1, // read-only
   DIGITAL_OUTPUT_REGISTER          = 0x3F2,
   TAPE_DRIVE_REGISTER              = 0x3F3,
   MAIN_STATUS_REGISTER             = 0x3F4, // read-only
   DATARATE_SELECT_REGISTER         = 0x3F4, // write-only
   DATA_FIFO                        = 0x3F5,
   DIGITAL_INPUT_REGISTER           = 0x3F7, // read-only
   CONFIGURATION_CONTROL_REGISTER   = 0x3F7  // write-only
};
```

All commands, parameter information, result codes, and disk data transfers go through the FIFO port. MSR contains the "busy" bitflags, that must be checked before reading/writing each byte through the FIFO. DOR controls the floppy drive motors, floppy drive "selection", and resets. The other registers contain very little information, and are typically accessed very little, if at all.

Note: IO port 0x3F6 is the ATA (hard disk) Alternate Status register, and is not used by any floppy controller.

Note2: some people prefer to give the registers values based on their offset from the base address, and then add the FDC's base address (0x3F0 or 0x370). So STATUS_REGISTER_A would have value 0, STATUS_REGISTER_B value 1, etc., and to access STATUS_REGISTER_B on FDC 1, you would use FDC1_BASE_ADDRESS + STATUS_REGISTER_B = 0x370 + 1 = 0x371.

As said above, the most common controller chip has 3 modes, and many bitflags in the registers are different (or opposite!) depending on the mode. However, all of the important registers and bitflags remain the same between modes. They are the following:

FIFO and Tape Drive Registers

FIFO: The FIFO register may not have a 16byte buffer in all modes, but this is a minor difference that does not really affect its operation.

TDR: The Tape Drive Register is identical in all modes, but it is useless (you will never find functional equipment that requires it).

DOR bitflag definitions

Mnemonic	bit number	value	meaning/usage
MOTD	7	0x80	Set to turn drive 3's motor ON
MOTC	6	0x40	Set to turn drive 2's motor ON
MOTB	5	0x20	Set to turn drive 1's motor ON
MOTA	4	0x10	Set to turn drive 0's motor ON
IRQ	3	8	Set to enable IRQs and DMA
RESET	2	4	Clear = enter reset mode, Set = normal operation
DSEL1 and 0	0, 1	3	"Select" drive number for next access

Note: the IRQ/DMA enable bit (bit 3, value = 8) cannot be cleared in "PS/2 mode", so for PIO transfers you should make sure to have a stubbed IRQ6 handler in place, just in case the IRQs happen anyway. The bit **must** be set for DMA to function.

Note2: if you want to execute a command that accesses a disk (see the command list below), then that respective disk must have its motor spinning (and up to speed), **and** its "select" bits must be set in the DOR, **first**.

Note3: toggling DOR reset state requires a 4 microsecond delay. It may be smarter to use DSR reset mode, because the hardware "untoggles" reset mode automatically after the proper delay.

MSR bitflag definitions

Mnemonic	bit number	value	meaning/usage
RQM	7	0x80	Set if it's OK (or mandatory) to exchange bytes with the FIFO IO port

DIO	6	0x40	Set if FIFO IO port expects an IN opcode
NDMA	5	0x20	Set in Execution phase of PIO mode read/write commands only.
CB	4	0x10	Command Busy: set when command byte received, cleared at end of Result phase
ACTD	3	8	Drive 3 is seeking
ACTC	2	4	Drive 2 is seeking
ACTB	1	2	Drive 1 is seeking
ACTA	0	1	Drive 0 is seeking

The two important bits are RQM and DIO. NDMA and BUSY are also useful in polling PIO mode. Most important is RQM, which is set when it is OK (or necessary!) to read/write data from/to the FIFO port. NDMA signals the end of the "execution phase" of a command, and the beginning of "result phase". DIO and BUSY should be checked to verify proper command termination (the end of "result phase" and beginning of "command phase").

CCR and DSR

The bottom 2 bits of DSR match CCR, and setting one of them sets the other. The upper 6 bits on both DSR and CCR default to 0, and can always be set to zero safely. So, even though they have different bit definitions, you always set them with identical values, and one or the other of DSR and CCR can be ignored in any modern system. Alternately, you can always set both of them, for maximum compatibility with ancient chipsets.

The bottom 2 bits specify the data transfer rate to/from the drive. You want both bits set to zero for a 1.44MB or 1.2MB floppy drive. So generally, you want to set CCR to zero just once, after bootup (because the BIOS may not have done it, unless it booted a floppy disk).

Note: a reset procedure does not affect this register. However, if you have drives of different types on the bus which use different datarates, then you need to switch the datarate when you select the other drive. It also seems to be possible to modify this register while the FDC is in "reset state".

Note2: some tutorials seem to claim that changing/setting the datarate causes an IRQ6. This is false.

Datarates used for setting either DSR or CCR:

```
Datarate   value   Drive Type
1Mbps        3       2.88M
500Kbps      0       1.44M, 1.2M
```

Note: There is also a 300Kbps (value = 1), and a 250Kbps setting (value = 2) but they are only for utterly obsolete drives/media.

DIR register, Disk Change bit

This bit (bit 7, value = 0x80) is fairly useful. It gets set if the floppy door was opened/closed. Sadly, almost all the emulator programs set and clear this bit completely inappropriately (especially after a reset). Do not trust your handling of this bit until you have tested the functionality on real hardware.

Note: The datasheet is very confusing about the value of the bit, because Model 30 mode shows the bit as being inverted. But in Model 30 mode, the **signal** is also inverted, so it comes out the same. "False" always means the bit is cleared, and "true" always means the bit is set.

Note2: You must turn on the drive motor bit before you access the DIR register for a selected drive (you do not have to wait for the motor to spin up, though). It may also be necessary to read the register five times (discard the first 4 values) when **changing** the selected drive -- because "selecting" sometimes takes a little time.

Basically, you want to keep a flag for whether there is media in each drive. If Disk Change is set and there was media, the OS should get a signal that the previous media was ejected.

Once the Disk Change bit signals "true" (and you have processed that "event"), you need to try to clear the bit. The main way to clear the bit is with a **successful** Seek/Recalibrate to a **new** cylinder on the media. (A reset does not work. If the controller thinks the heads are already on the correct cylinder, it will eat a Seek command without clearing the Disk Change bit. If the heads are already on cylinder 0, a Recalibrate is also a no-op.) If the seek **fails**, you can be fairly certain that there is no media in the drive anymore. It is important to note that this means you should check the value of Disk Change just prior to every Seek command that you do, because otherwise you will lose any Disk Change information. This is also true for implied seeks, and relative seeks.

Apparently a small number of floppy drives also support one additional way to clear the bit -- something that Linux calls a "twaddle". Simply toggle the drive motor bit on, off, and then on again. When your driver tries to clear the Disk Change bit the first time, it can try a twaddle, and see if it works, and keep a flag if it does.

Programming Details

Overall, the controller needs to be initialized and reset once (see below for the steps involved). Then drives can be accessed. To access drives:

1. Turn on the drive's motor and select the drive, using an "outb" command to the DOR IO port.
2. Wait for awhile for the motor to get up to speed, using some waiting scheme.
3. Issue your command byte plus some parameter bytes (the "command phase") to the FIFO IO port.
4. Exchange data with the drive / "seek" the drive heads (the "execution phase"), on the FIFO IO port.
5. Get an IRQ6 at the end of the execution phase, but **only if the command HAS an execution phase**.
6. Read any "result bytes" produced by the command (the "result phase"), on the FIFO IO port.
7. The commands "Recalibrate", "Seek", and "Seek Relative" do not have a result phase, and require an additional "Sense Interrupt" command to be sent.

And then you are ready for the next command. See below for more detail on how to issue a command.

Motor Delays

Note: the Linux floppy driver sourcecode has a comment that claims that turning on the MOTC or MOTD bits in the DOR can completely lock up some systems. This claim seems unlikely?

When you turn a floppy drive motor on, it takes quite a few milliseconds to "spin up" -- to reach the (stabilized) speed needed for data transfer. The controller has electronics to handle a large variation in rotation speed, but it has its limits. If you start reading/writing immediately after the motor is turned on, "the PLL will fail to lock on to the data signal" and you will get an error.

After you are done reading (or writing), you should typically wait an additional two seconds to turn the motor off. (It may also be smart to seek the heads back to cylinder 0 just before turning the motor off.) You could leave the motor on longer, but it might give the user the idea that the floppy is still transferring data and that it's taking an awfully long time. The reason to leave the motor on is that your driver may not know if there is a queue of sector reads or writes that are going to be executed next. If there are going to be more drive accesses immediately, they won't need to wait for the motor to spin up.

The suggested delays when turning the motor on are:

- 300 milliseconds (for a 3.5" floppy).
- 500 milliseconds (for a 5.25" floppy).

These values should be more than enough for any floppy drive to spin up correctly. 50ms should be sufficient, in fact.

Another functional method is not delaying or waiting at all, but just enter a loop and keep retrying any command until it works. A 3.5 inch disk rotates once every 200ms, so each retry is effectively a delay.

Gap Lengths

There are two different gap lengths that are controlled by software for specifying the amount of blank space between sectors. The gap lengths are used by the floppy hardware to help find the "start of sector" markers, and to avoid problems caused by speed variations in different floppy drives (for e.g. writing a sector on a slower drive would cause the sector to take up more physical space on the disk, potentially overwriting the next sector). The first gap length (typically called "GPL1") is used when reading or writing data, and sets the length of the gap between sectors that the floppy should expect (but doesn't change this gap). The second (typically called "GPL2") is the gap length for the "format track" command, which specifies the amount of space between sectors to use. The actual gap lengths depend on many factors, but GPL1 is always a bit less than GPL2 so that the floppy hardware starts expecting the next sector near the end of the blank space.

Standard values for these gap lengths can be found in the floppy controller datasheets. However, it is possible to squeeze more sectors on each track by reducing the gap length. For example, by using a gap length of 0x1C when formatting (and 0x0C when reading/writing) instead of 0x54 (and 0x1B when reading/writing) it's possible to format a normal 1440 KB floppy disk with 21 sectors per track to create a 1680Kb floppy disk. These disks may be unreliable on very old computers (where accuracy and speed variation may be worse), but were considered reliable enough for Microsoft to distribute a version of Windows on 1680Kb floppies (Windows 95 on a set of 12 floppies). It is also possible to format 3 extra cylinders on each disk, for a total of 83.

Hardware Control of Bad Cylinders

The FDC subsystem has a built-in method for handling unreliable media. However, it is probably not a good idea to use it. It involves finding a bad sector on the media, and then marking the entire track or cylinder as being bad, during the formatting process.

If you try to do this, then you cannot simply seek to a cylinder. All of the cylinders get "remapped" with new "TrackID"s. Whenever you seek to a cylinder, then you need to use the ReadID command to verify that the cylinder you seeked to contains the data that you actually want. So this eliminates any possibility of using implied seeks, and adds an extra step to most read/write operations.

Procedures

Reinitialization

The BIOS probably leaves the controller in its default state. "Drive polling mode" on, FIFO off, threshold = 1, implied seek off, lock off. This is a lousy state for the controller to be in, and your OS will need to fix it. The BIOS probably also does not have any better guess as to the proper values for the "Specify" command than your OS does (the values are specific to the particular drive). There is certainly no reason why you should trust the BIOS to have done any of it **correctly.**

So, when your OS is initializing:

1. Send a Version command to the controller.
2. Verify that the result byte is 0x90 -- if it is not, it might be a good idea to abort and not support the floppy subsystem. Almost all of the code based on this article will work, even on the oldest chipsets -- but there are a few commands that will not.
3. If you don't want to bother having to send another Configure command after every Reset procedure, then:
    1. Send a better Configure command to the controller. A suggestion would be: drive polling mode off, FIFO on, threshold = 8, implied seek on, precompensation 0.
    2. send a Lock command.
4. Do a Controller Reset procedure.
5. Send a Recalibrate command to each of the drives.

Controller Reset

The second most common failure mode for any floppy command is for the floppy controller to lock up forever. This condition is detected with a timeout, and needs to be fixed with a Reset. You also need one Reset during initialization. Hopefully, it is the only one you will ever need to do.

1. Either use:
    1. Bit 2 (value = 4) in the DOR: Save the current/"original" value of the DOR, write a 0 to the DOR, wait 4 microseconds, then write the original value (bit 2 is always set) back to the DOR.
    2. **or** Bit 7 (value = 0x80) in the DSR: Precalculate a good value for the DSR (generally 0), and OR it with 0x80. Write that value to the DSR.
2. Wait for the resulting IRQ6 (unless you have IRQs turned off in the DOR)
3. If (and only if) drive polling mode is turned on, send 4 Sense Interrupt commands (required).
4. If your OS/driver never sent a Lock command, then you probably need to send a new Configure command (the fifo settings were lost in the reset).
5. Do a Drive Select procedure for the next drive to be accessed.

Note: A reset clears all the Specify information, so the next Select procedure must send a new Specify command (use some sort of flag to tell the driver to do this).

Note2: Emulators will often set the Disk Change flag to "true" after a reset, **but this does not happen on real hardware** -- it is a shared bug in all the emulators that do it. Another shared bug is that most emulators do not fire an IRQ6 if disk polling mode is off.

Note3: A reset does not change the drive polling mode or implied seek settings.

Drive Selection

Each floppy drive on the system may be a different type. When switching between accessing different types of drives, you need to fix the Specify and Datarate settings in the controller. The controller does **not** remember the settings on a per-drive basis. The only per-drive number that it remembers is the current cylinder.

1. Send the correct Datarate setting to CCR. Usually this is a 0 (1.44MB floppy drive).
2. If the newly selected drive is a different type than the previously selected drive (or changing from PIO to DMA mode), send a new Specify command.
3. Set the "drive select" bits (and the other bitflags) in DOR properly, including possibly turning on the Motor bit for the drive (if it will be accessed soon).

Detecting Media

The user can swap media out of a floppy drive at any moment. If your driver sends a command to the drive, and the command fails -- this may be the reason why.

1. Turn the drive motor bit on.
2. Read DIR. If the "Disk Change" bitflag is set to "true", then the floppy drive door was opened, so the OS needs to test if a new disk is in the drive.

Waiting

Waiting for the drive can be done in many ways, and it is an OS-specific design decision. You can poll the value of the PIT, waiting for it to count down to a certain value. You can poll a memory location that contains "the current time" in some format, waiting for it to reach a certain value. Your OS can implement a realtime callback, where a particular function in the driver will be called at a particular time. Or you can implement some form of multitasking "blocking", where the driver process is put in a "sleep" state, and is not assigned any timeslices for a certain length of time.

The Proper Way to issue a command

1. Read MSR (port 0x3F4).
2. Verify that RQM = 1 and DIO = 0 ((Value & 0xc0) == 0x80) -- if not, reset the controller and start all over.
3. Send your chosen command byte to the FIFO port (port 0x3F5).
4. In a loop: loop on reading MSR until RQM = 1. Verify DIO = 0, then send the next parameter byte for the command to the FIFO port.
5. Either Execution or Result Phase begins when all parameter bytes have been sent, depending on whether you are in PIO mode, and the command has an Execution phase. If using DMA, or the command does not perform read/write/head movement operations, skip to the Result Phase.
6. (In PIO Mode Execution phase) read MSR, verify NDMA = 1 ((Value & 0x20) == 0x20) -- if it's not set, the command has no Execution phase, so skip to Result phase.
7. begin a loop:
8. Either poll MSR until RQM = 1, or wait for an IRQ6, using some waiting method.
9. In an inner loop: transfer a byte in or out of the FIFO port via a system buffer, then read MSR. Repeat while RQM = 1 and NDMA = 1 ((Value & 0xa0) == 0xa0).
10. if NDMA = 1, loop back to the beginning of the outer loop, unless your data buffer ran out (detect underflow/overflow).

11. Result Phase begins. If the command does not have a Result phase, it silently exits to waiting for the next command.
12. If using DMA on a read/write command, wait for a terminal IRQ6.
    1. Loop on reading MSR until RQM = 1, verify that DIO = 1.
    2. In a loop: read the next result byte from the FIFO, loop on reading MSR until RQM = 1, verify CMD BSY = 1 and DIO = 1 ((Value & 0x50) == 0x50).
13. After reading all the expected result bytes: check them for error conditions, verify that RQM = 1, CMD BSY = 0, and DIO = 0. **If not** retry the entire command again, several times, starting from step 2!

Note: implementing a failure timeout for each loop and the IRQ is pretty much required -- it is the only way to detect many command errors.

Commands

Each command is a single byte with a value less than 32, which is written to the DATA_FIFO port. There are three "option bits" that can be OR'ed onto some command bytes, typically called MF, MT, and SK. Each command must be followed by a specific set of "parameter bytes", and returns a specific set of "result bytes". See the discussion of each command for its list of parameter bytes, and result bytes.

A command byte may only be sent to the FIFO port if the RQM bit is 1 and the DIO bit is 0, in the MSR. If these bits are not correct, then the previous command encountered a fatal error, and you must issue a reset.

The following is an enumeration of the values of the command bytes. The ones that you actually will **use** are marked with a * and a comment.

```
enum FloppyCommands
{
   READ_TRACK =                 2,      // generates IRQ6
   SPECIFY =                    3,      // * set drive parameters
   SENSE_DRIVE_STATUS =         4,
   WRITE_DATA =                 5,      // * write to the disk
   READ_DATA =                  6,      // * read from the disk
   RECALIBRATE =                7,      // * seek to cylinder 0
   SENSE_INTERRUPT =            8,      // * ack IRQ6, get status of last
   WRITE_DELETED_DATA =         9,
   READ_ID =                    10,     // generates IRQ6
   READ_DELETED_DATA =          12,
   FORMAT_TRACK =               13,     // *
   SEEK =                       15,     // * seek both heads to cylinder
   VERSION =                    16,     // * used during initialization,
   SCAN_EQUAL =                 17,
   PERPENDICULAR_MODE =         18,     // * used during initialization,
   CONFIGURE =                  19,     // * set controller parameters
   LOCK =                       20,     // * protect controller params fr
   VERIFY =                     22,
   SCAN_LOW_OR_EQUAL =          25,
   SCAN_HIGH_OR_EQUAL =         29
};
```

Option bits

OR these bits onto the above read/write/format/verify commands.

Bit MT

Value = 0x80. Multitrack mode. The controller will switch automatically from Head 0 to Head 1 at the end of the track. This allows you to read/write twice as much data with a single command.

Bit MF

Value = 0x40. "MFM" magnetic encoding mode. Always set it for read/write/format/verify operations.

Bit SK

Value = 0x20. Skip mode. Ignore this bit and leave it cleared, unless you have a really good reason not to.

Status Registers

There are 3 registers that hold information about the last error encountered. The st0 register information is passed back with the result bytes of most commands. The st1 and st2 information is returned in the result bytes of read/write commands. They can also be retrieved with a Dumpreg command.

st0

The top 2 bits (value = 0xC0) are set after a reset procedure, with polling on. Either bit being set at any other time is an error indication. Bit 5 (value = 0x20) is set after every Recalibrate, Seek, or an implied seek. The other bits are not useful.

st1

The st1 register provides more detail about errors during read/write operations.

Bit 7 (value = 0x80) is set if the floppy had too few sectors on it to complete a read/write. This is often caused by **not subtracting 1** when setting the DMA byte count. Bit 4 (value = 0x10) is set if your driver is too slow to get bytes in or out of the FIFO port in time. Bit 1 (value = 2) is set if the media is write protected. The rest of the bits are for various types of data errors; indicating bad media, or a bad drive.

st2

The st2 register provides more (useless) detail about errors during read/write operations.

The bits all indicate various types of data errors for either bad media, or a bad drive.

Configure

This command initializes controller-specific values: the data buffer "threshold" value, implied seek enable, FIFO disable, polling enable. (And "Write Precompensation".) A good setting is: implied seek on, FIFO on, drive polling mode off, threshold = 8, precompensation 0.

If you enable implied seeks, then you don't have to send Seek commands (or Sense Interrupt commands for the Seek commands).

If you leave the FIFO disabled, then it cannot buffer data -- you will get an IRQ6 or DMA request for every single byte (which needs to be serviced quickly, because the very next byte will automatically cause an overflow/underflow and error out your transfer!).

Drive polling mode is just an annoyance that is there for backwards software compatibility. It makes you need Sense Interrupts after every reset. Always turn it off, if you send a Configure command.

A big "threshold" such as 15 will wait 15 bytes between interrupts. So there won't be many interrupts, but you only have one byte left in the FIFO before it overflows/underflows and kills the r/w operation. thresh_val = threshold - 1.

Write precompensation is a technical thing having to do with drive head magnetics. A precomp_val of 0 tells the controller/drive to use the manufacturer default value. Use that unless you have a very good reason for setting another value.

- Configure command = 0x13
- First parameter byte = 0
- Second parameter byte = (implied seek ENable << 6) | (fifo **DIS**able << 5) | (drive polling mode **DIS**able << 4) | thresh_val (= threshold - 1)
- Third parameter byte = precomp_val = 0
- No result bytes.
- No interrupt.

Version

Returns one byte. If the value is 0x90, the floppy controller is a 82077AA.

- Version command = 0x10
- No parameter bytes.
- No interrupt.
- First result byte = 0x90

Lock

Under default circumstances, every Controller Reset will disable the fifo, and set the fifo threshold to 1 (thresh_val = 0). If you change these settings with the Configure command and don't want to have to fix them after every Controller Reset, then you can send a Lock command with the lock bit turned on. You can "unset" the lock, by sending another Lock command with the lock bit turned off. Use the MT option bit as the lock bit.

- Lock command = 0x94
- **or** Unlock command = 0x14
- No parameter bytes.
- No interrupt.
- First result byte = lock bit << 4

Specify

This command puts information in the controller about the next disk drive to be accessed.

Important Note: The controller only has one set of registers for this information. It does not **store** the information between multiple drives. So if you are switching control between two different types of drives (with different specify values) then you need to send a new Specify command every single time you select the other drive.

If your driver will be using DMA to transfer data, set the NDMA bit to 0. If using PIO mode instead, set it to 1.

- Specify command = 0x3
- First parameter byte = SRT_value << 4 | HUT_value
- Second parameter byte = HLT_value << 1 | NDMA
- No result bytes.
- No interrupt.

SRT, HLT and HUT

These parameters sent with the Specify command to the controller are meant to optimize drive performance, and head lifetime.

The values are specific to the exact model, condition, and age of floppy drive installed on the system. The values sent by the driver to the controller were always meant to be **adaptive**. That is, your driver is theoretically supposed to keep statistics of how often Seek commands fail with the current setting of SRT. If they always work, and your driver wants to optimize performance, then it can send a new Specify command, with the SRT value reduced by 1. Then begin keeping new statistics. Similarly for HLT regarding Read/Write operations. As drives age and collect dirt, the driver would automatically compensate by seeing higher statistical error rates, and increase the values of SRT and HLT.

Keeping statistics in that way only works when the drive in question is used often. Now that internal floppy drives are nearly obsolete, it is worthless. So the current recommendation is just to use very safe values, and forget about performance.

If you look up spec sheets for individual floppy drives, they usually show a worst-case "track to track seek time" = SRT, but not the other two.

- SRT = "Step Rate Time" = time the controller should wait for the head assembly to move between successive cylinders. A reasonable amount of time

to allow for this is 3ms for modern 3.5 inch floppy drives. A very safe amount would be 6 to 8ms. To calculate the value for the SRT setting from the given time, use "SRT_value = 16 - (milliseconds * data_rate / 500000)". For a 1.44 MB floppy and 8ms delay this gives "SRT_value = 16 - (8 * 500000 / 500000)" or a parameter value of 8.

- HLT = "Head Load Time" = time the controller should wait between activating a head and actually performing a read/write.

A reasonable value for this is around 10ms. A very safe amount would be 30ms. To calculate the value for the HLT setting from the given time, use "HLT_value = milliseconds * data_rate / 1000000". For a 1.44 MB floppy and a 10ms delay this gives "HLT_value = 10 * 500000 / 1000000" or 5.

- HUT = "Head Unload Time" = time the controller should wait before deactivating the head. To calculate the value for the HUT setting

from a given time, use "HUT_value = milliseconds * data_rate / 8000000". For a 1.44 MB floppy and a 240 mS delay this gives "HUT_value = 24 * 500000 / 8000000" or 15. However, it seems likely that the smartest thing to do is just to set the value to 0 (which is the maximum in any mode).

Sense Interrupt

This command's main function is to return any error code from a Seek or Recalibrate command to your driver. It also clears an internal bitflag in the controller. It is required in three circumstances that produce interrupts.

1. After doing a Controller Reset procedure with drive polling mode turned on.
2. After the completion of a Seek command (or Relative Seek).
3. After the completion of a Recalibrate command.

These are the only times when you should send a Sense Interrupt. You should still send them even if you have IRQs turned off in the DOR and you are using PIO polling instead. If you send Sense Interrupt commands at other times: the command will complete, return a 0x80, and then lock up the controller until you do a Reset.

- Sense Interrupt command = 0x8
- No parameter bytes.
- No interrupt.
- First result byte = st0
- Second result byte = controller's idea of the current cylinder

Note: if you try to read the result bytes without waiting for RQM to set, then you are likely to always get an incorrect result value of 0. This is also likely to get your driver out of sync with the FDC for input/output. The correct value of st0 after a reset should be 0xC0 | drive number (drive number = 0 to 3). After a Recalibrate/Seek it should be 0x20 | drive number.

Recalibrate

Note: There is an error in the FDC datasheet regarding this command. Some statements say the command will try a maximum of 80 head assembly steps. In other places it says 79 steps. The value 79 is correct.

The motor needs to be on, and the drive needs to be selected. For this particular command, you do not have to wait for the command to complete before selecting a different drive, and sending another Recalibrate command to it (but the Step Rates have to match, for this to work).

It is possible for a normal 1.44M floppy to be formatted with 83 cylinders. So, theoretically, it may take two (or more) Recalibrates to move the head back to cylinder 0. It is a good idea to test bit 5 (value = 0x20) in st0 after the Sense Interrupt, and retry the Recalibrate command if that bit is clear.

- Recalibrate command = 0x7
- First parameter byte = drive number = 0 to 3.
- No result bytes.
- The interrupt may take up to 3 seconds to arrive, so use a long timeout.

It is possible to poll the "disk active" bits in the MSR to find out when the head movement is finished. A Sense Interrupt command is required after this command completes, to clear it from being BUSY. (Multiple Sense Interrupts, if you ran multiple simultaneous Recalibrates.)

Seek

The motor needs to be on, and the drive needs to be selected. For this particular command, you do not have to wait for the command to complete before selecting a different drive, and sending another Seek command to it. Maximum cylinder number is 255; if the disk has more, you must use the Relative Seek command, instead. There is really no reason to ever use head 1 when seeking.

- Seek command = 0xf
- First parameter byte = (head number << 2) | drive number (the drive number must match the currently selected drive!)
- Second parameter byte = requested cylinder number
- No result bytes.
- The interrupt may take up to 3 seconds to arrive, so use a long timeout.

It is possible to poll the "disk active" bits in the MSR to find out when the head movement is finished.

A Sense Interrupt command is required after this command completes, to clear it from being BUSY. (Multiple Sense Interrupts, if you ran multiple Seeks.)

Note: the controller tries to remember what cylinder each drive's heads are currently on. If you try to seek to that same cylinder, then the controller will silently ignore the command (and return a "success" value). One of the things this means is that you can get a "success" return value on a seek **even if there is no media in the drive**, if you happen to seek to the wrong cylinder number.

Relative seek

The normal Seek command allows you to select an absolute cylinder number from 0 to 255. It is also possible to use a "relative" seek command in all situations, and especially for drives that have more than 255 cylinders (there are none, currently).

To use a relative seek, set the MT bit to 1. To seek to higher cylinder numbers set the MFM bit -- clear MFM to seek backwards to lower cylinder numbers. If you seek past cylinder 255, there are a lot of extra complications. Otherwise, the command behaves identically to regular Seek.

Read/Write

Note: Remember that this is in CHS format, so the sector number starts at 1.

- Read command = MT bit | MFM bit | 0x6
- or Write command = MT bit | MFM bit | 0x5
- First parameter byte = (head number << 2) | drive number (the drive number must match the currently selected drive!)
- Second parameter byte = cylinder number
- Third parameter byte = head number (yes, this is a repeat of the above value)
- Fourth parameter byte = starting sector number
- Fifth parameter byte = 2 (all floppy drives use 512bytes per sector)
- Sixth parameter byte = sector count to transfer
- Seventh parameter byte = 0x1b (GAP1 default size)

- Eighth parameter byte = 0xff (all floppy drives use 512bytes per sector)

- First result byte = st0 status register
- Second result byte = st1 status register
- Third result byte = st2 status register
- Fourth result byte = cylinder number
- Fifth result byte = ending head number
- Sixth result byte = ending sector number
- Seventh result byte = 2

Note: if you try to read the result bytes without waiting for RQM to set, then you are likely to always get an incorrect result value of 0. This is also likely to get your driver out of sync with the FDC for input/output.

Note2: Floppy media and electronics are well known for being unreliable. Any read or write command that fails should be retried at least twice, unless it was a write and failed on "write protect".

Perpendicular Mode and 2.88M floppies

If you are using an emulator and you need a floppy disk image that is bigger than 1440Kb, there is a 2880Kb image available. In order to access it in Pmode, you need to modify your driver to handle Perpendicular Mode. Basically, it is an extra configuration command where you enable any of the four drives for perpendicular mode.

Note: If the parameter byte is 0 (except for the "perpendicular enable" bits), then a reset will not affect the settings.

- Perpendicular Mode command = 0x12
- First parameter byte = (Drive 3 enable << 5) | (Drive 2 enable << 4) | (Drive 1 enable << 3) | (Drive 0 enable << 2)
- No result bytes.
- No interrupt.

You also need to set CCR/DSR for the 1M datarate (value = 3) to access a 2.88M drive.

Additional Programming Notes

If you are doing a transfer between 2 floppy drives (so that both motors are on), and you are toggling "selection" between the two, there may be a short delay required.

Code Examples

A common coding error example

The following code intentionally contains a common bug that causes an infinite loop (waiting for IRQ6) on most emulators.

```
volatile byte ReceivedIRQ = false;

// This function gets called when an IRQ6 is generated.
void FloppyHandler()
```

```
{
   ReceivedIRQ = true;
}

// Waits for an IRQ to be issued.
void WaitForIRQ()
{
   ReceivedIRQ = false;
   while(!ReceivedIRQ) ;
}

// buggy example Controller Reset function
void ResetFloppy()
{
    DisableController();
    EnableController();

    WaitForIRQ();
}
```

Sure this code *looks* OK, but some emulators or floppy drives might manage to be faster than your code. What if you've just returned from *EnableController()* and the floppy already issued the IRQ6? Then *ReceivedIRQ* will be set to true, your driver will enter *WaitForIRQ()*, set it to false again and then infinitely loop, waiting for an IRQ that has already been received. It's usually better to do something like:

```
volatile byte ReceivedIRQ = false;

// This function gets called when an IRQ6 is generated.
void FloppyHandler()
{
   ReceivedIRQ = true;
}

// pretty good Controller Reset function (it should do more checking of M
void ResetFloppy()
{
    ReceivedIRQ = false;        // This will prevent the FDC from being j

    // Enter, then exit reset mode.
    outb(Controller.DOR,0x00);
    outb(Controller.DOR,0x0C);

    while(!ReceivedIRQ) ;       // Wait for the IRQ handler to run

    // sense interrupt -- 4 of them typically required after a reset
    for (i = 4 ; i > 0 ; --i);
    {
       send_command(SENSE_INTERRUPT);
       read_data_byte();
```

```
        read_data_byte();
    }

    outb(controller.CCR,0x00);  // 500Kbps -- for 1.44M floppy

    // configure the drive
    send_command(SPECIFY);
    outb(Controller.FIFO, steprate_headunload);
    outb(Controller.FIFO, headload_ndma);
}
```

Related Links

Articles

- DMA

Reference Documents

- http://www.osdever.net/documents/82077AA_FloppyControllerDatasheet.pdf?the_id=41
- http://bos.asmhackers.net/docs/floppy/
- http://bos.asmhackers.net/docs/floppy/docs/floppy_tutorial.txt
- Intel 82078 CHMOS SINGLE-CHIP FLOPPY DISK CONTROLLER datasheet (useless) (http://www.intel.com/design/archives/periphrl/docs/29046803.htm)
- http://www.brokenthorn.com/Resources/OSDev20.html

Forum Posts

- TUTORIAL, with DMA, by Mystran (highly recommended, but has a few tiny errors)
- Floppy in pmode
- Floppy Disk Driver
- Floppy programming tutorial (floppy_tutorial.txt) companion thread
- Multitask Floppy Driver
- PIO mode information

Implementations

- freedos (http://koders.com/c/fid051291340B94EC7F5D1A38EF6843466C0B07627B.aspx?s=fdc) (C,GPL)
- GazOS (http://bos.asmhackers.net/docs/floppy/snippet_9/fdc.c) (C,GPL)
- RDOS (http://bos.asmhackers.net/docs/floppy/snippet_5/FLOPPY.ASM) (ASM, GPL)
- Linux (http://www.gelato.unsw.edu.au/lxr/source/drivers/block/floppy.c) (C,GPL)

Categories: Storage | Common Devices
