Programming Floppy Disk Controllers

Overview

The PC usually uses the NEC ÂµPD765 floppy disk controller. The AT can also incorporate an Intel 82072A controller, while the PS/2 uses an Intel 82077A. The ÂµPD765 and the ROM code in the controller form a microcontroller, and this handles the majority of the work of the controller. All PC compatibles have FDCs which are compatible with the controllers described in this document.

This document describes the registers used to interface with the controller, and the commands which it recognises.

There are a lot of delays involved in communicating with the controller. These delays are for a variety of reasons, including the time needed to spin up the drive motor, and the time taken to move the head to a new position and wait for it to settle in place.

When the drive motor is started up or the a seek is requested, there will be a delay until the drive is ready for the next command. An interrupt is issued by the hardware when it is ready for the next command, and this will tell you that the drive is ready for your command.

In a single tasked environment (such as DOS), the only option is to have your driver constantly wait for an interrupt, and then respond to it. However, in a multi-tasked or multi-threaded environment, it is perfectly acceptable to write a driver which allows other tasks to be executed while waiting for the interrupt.

When performing a read or write operation, data may be transferred a byte at a time by reading from or writing to the appropriate port, or a sector/track at a time through the use of DMA channel 2. Programming the DMA controller is beyond the scope of this document, but will be described in a future document to be added to this site.

Configuration of an FDC on a PC

The Floppy Controller on a PC uses a standard configuration. On the XT there are 3 ports available for control and data access registers. On the AT, there are 4, and on the PS/2 there are 6.

The base port address used for the controller is dependant on whether the controller is configured as the primary or secondary controller. This base address controls the port addresses used for each of the registers on the controller. It can additionally be noted that all floppy controllers on a PC use DMA channel 2 for data transfer during a read or write, and they all issue a hardware interrupt via IRQ6 to be serviced by INT 0eh by default.

	Primary Address	**Secondary Address**	**Write (W) / Read (R)**
base address	3f0h	370h	
status register A (PS/2)	3f0h	370h	R
status register B (PS/2)	3f1h	371h	R
digital output register DOR	3f2h	372h	W
main status register	3f4h	374h	R
data rate select register (DSR)(PS/2)	3f4h	374h	W
data register	3f5h	375h	R/W
digital input register DIR (AT)	3f7h	377h	R
configuration control register (AT)	3f7h	377h	W
DMA channel	2	2	
IRQ	6	6	
INTR	0eh	0eh	

Note that the controller can be configured differently from the defaults for handling interrupts.

FDC Registers

This section gives more detailed information on the use of the registers listed in the above table. Additionally, the first registers to be described are those common to each system described, and these will be followed by descriptions of AT specific registers and PS/2 specific registers.

Common Registers:

1. Digital Output Register (DOR)
2. Main Status Register (MSR)
3. Data Register (DR)

AT Specific Registers:

1. Digital Input Register (DIR)
2. Configuration Control Register (CCR)

PS/2 Specific Registers:

1. Status Register A
2. Status Register A
3. Data Rate Select Register

Digital Output Register

7	6	5	4	3	2	1	0
MOTD	**MOTC**	**MOTB**	**MOTA**	**DMA**	**$\overline{\text{REST}}$**	**DR1**	**DR2**

MOTD, MOTC, MOTB, MOTA:	**Motor control for floppy drive D, C, B, A**	
	1 = Start Motor	0 = Stop Motor
DMA:	**DMA and IRQ channel**	
	1 = Enabled	0 = Disabled
$\overline{\text{REST}}$:	**Controller reset**	
	1 = Controller enabled	0 = Execute controller reset
DR1,DR0:	**Drive select**	
	00 = drive 0 (A) 01 = drive 1 (B) 10 = drive 2 (C) 11 = drive 3 (D)	

This register is write only, and controls the drive motors, as well as selecting a drive and the DMA/IRQ mode, and resetting the controller.

If the REST bit is set, the controller is enabled, in order to accept and execute commands. If it is equal to 0, the controller ignores all commands and carries out an internal reset of all internal registers (except the DOR).

Note that a drive cannot be selected unless its motor is on, although setting the bits at the same time is acceptable.

Note also that drives 2 and 3 (floppy drives C and D) are not supported in all systems.

Example: To start the motor

To start the motor on drive A and select it, ready for another operation, you would use the following:

MOTA = 1 (start motor for drive A)
DMA = 1 (assuming you want to use DMA and interrupts)
REST = 1 (Controller enabled, otherwise no other commands will be executed)
DR1,DR0 = 00 (Select drive A)

All other bits are set to 0

Thus 16 + 8 + 4 + 0 = 28, or 01Ch, is sent to port 3f2h (Drive A is usually on the primary controller).

In assembly language, this would be written as:

```
mov     al,01ch
mov     dx,03f2h
out     dx,al
```

Example: To reset the controller

To reset the controller, send 0 to port 3f2h. This turns off all motors, selects no drives (because drive A's motor is not active, it cannot be selected), disables the DMA and IRQ line, and resets the controller.

The code for this is as follows:

```
        mov     al,0
        mov     dx,03f2h
        out     dx,al
```

Main Status Register

7	6	5	4	3	2	1	0
MRQ	**DIO**	**NDMA**	**BUSY**	**ACTD**	**ACTC**	**ACTB**	**ACTA**

MRQ:	**Main Request**	
	1 = Data Register ready	0 = Data Register not ready
DIO:	**Data input/output**	
	1 = Controller ? CPU	0 = CPU ? Controller
NDMA:	**non-DMA mode**	
	1 = Controller not in DMA mode	0 = Controller in DMA mode
BUSY:	**Instruction (device busy)**	
	1 = active	0 = not active
ACTD, ACTC, ACTB, ACTA:	**Drive D, C, B, A in positioning mode**	
	1 = active	0 = not active

The MSR is read-only, and contains the controller's status information. This register can be read whatever else the controller is doing.

It should be noted that this register is different from status registers ST0-ST3, which contain data concerning the last command executed. These registers are accessed via the data registers.

Bit 7 (MRQ) indicates whether the controller is ready to receive or send data or commands via the data register.

DIO is used to provide an indication of whether the controller is expecting to receive data from the CPU, or if it wants to output data to the CPU.

If the controller is set up to use DMA channel 2 to transfer data to or from main memory, the NDMA bit is not set. If this bit is set, data transfer is carried out exclusively by means of read or write commands to the data register. In this case, the controller issues a hardware interrupt every time that it either expects to receive or wants to supply a data byte.

Bit 4 indicates whether the controller is busy or not. If the bit is set, the controller is currently executing a command.

Bits 0-3 indicate which (if any) drive is currently in the process of positioning it's read/write heads, or being recalibrated.

Note that the delay waiting for the controller to be ready for a read or write can be as much as 175Âµs on an Intel controller, and longer on older controllers.

Example: To test whether the controller is ready to receive commands and data, it is necessary to test MRQ and DIO. This involves reading the port, masking the bits, and doing a comparison on the result (to test the values of the bits).

```
mrqloop:
        mov dx,03f4h
        in  al,dx
        and al,0c0h
        cmp al,080h
        jne mrqloop
```

This code will keep looping until the controller says that it is ready to receive data. Note that if the controller is expecting to output data to the CPU, this code will not spot that. You would need to add another couple of lines of code if you need to check for both possibilities (In general, you would know from previous commands whether the controller should be expecting or offering data, and so only need to check for one condition).

Data Register

The data register is an 8 bit register, like each of the other registers, which provides indirect access to a stack of registers. A command can be one to nine bytes in length, and the first byte tells the controller how many more bytes to expect. The controller sends the command bytes to the correct registers in it's stack, saving the programmer from the need to use a separate index register, as is the case in some other devices (e.g. some VGA registers).

Some controllers, such as the i82077A, have a buffer, with a programmable threshold, allowing the data to be transferred several bytes at a time. This helps to speed up the transfer of data and commands, as well as reducing the response time seen on the ÂµPD765.

Status Register ST0

7	6	5	4	3	2	1	0
IC$_1$	**IC$_0$**	**SE**	**UC**	**NR**	**HD**	**US$_1$**	**US$_0$**

IC$_1$, IC$_0$:	**Interrupt Code**
	00 = normal termination; command terminated without any errors 01 = abnormal termination; the controller started execution of the command, but couldn't terminate it correctly 10 = invalid command; the controller could not start command execution 11 = abnormal termination by polling; drive became not ready
SE:	**Seek End**
	The controller has completed a seek or a calibration command, or has correctly executed a read or write command which has an implicit seek
UC:	**Unit Check**
	Set if the drive faults or if a recalibrate cannot find track 0 after 79 pulses.
NR:	**drive not ready**
HD:	**head currently active**
	1 = head 1 0 = head 0
US$_1$, US$_0$:	**currently selected drive (unit select)**
	00 = drive 0 (A:) 01 = drive 1 (B:) 10 = drive 2 (C:)

11 =
drive 3
(D:)

Back