"8042" PS/2 Controller

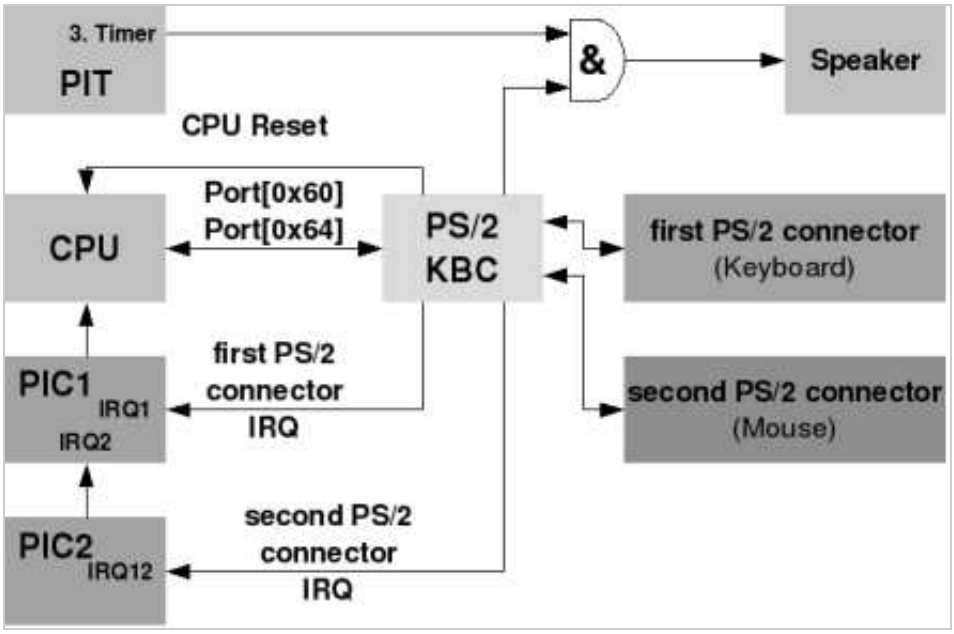
From OSDev Wiki

Contents

Overview

The PS/2 Controller (often called a "Keyboard controller") is located on the mainboard. In the early days the controller was a single chip (8042). As of today it is part of the Advanced Integrated Peripheral.

The name is misleading because the controller does more than controlling communication with PS/2 devices.

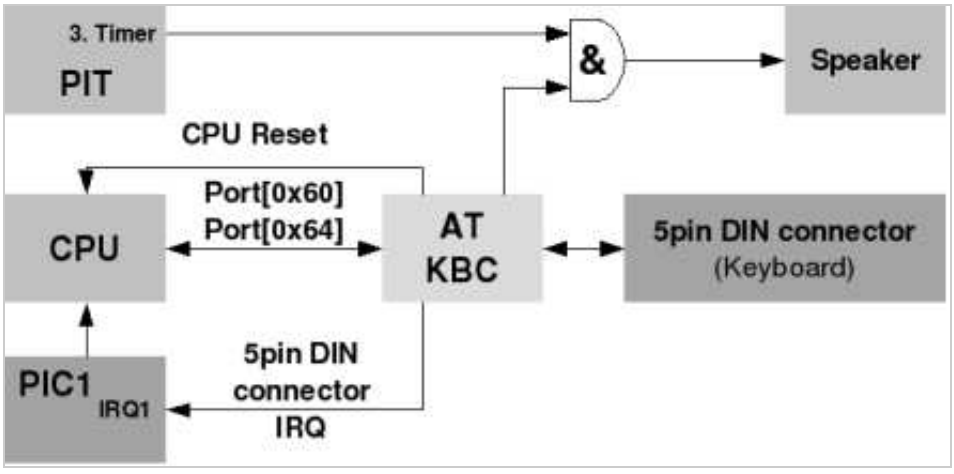


Overview of the PS/2-Controller

History

The original uni-directional, single channel IBM PC and PC-XT keyboard interface was controlled by a multi purpose PPI (Intel 8048, Programmable peripheral interface; also used to control other functions, like sound and parity error). The XT controller is 100% obsolete and won't be discussed further in this page.

With the PC-AT, IBM introduced new keyboards (with a new bi-directional protocol) and a new keyboard controller (Intel 8042). The old PPI was not part of the mother board any more.



Overview of the AT-Controller

The 8042 was a powerful microcontroller. To reduce costs, some of the general purpose input/output capabilities of the AT controller was used to control various functions unrelated to the keyboard, including :

- System Reset
- The A20-Gate

With the introduction of the PS/2 series, the main change to the keyboard controller subsystem was its expansion to control both a keyboard and a mouse. Previously PC and compatible mice were connected to different physical interfaces, including Serial Ports. The AT keyboard controller and its "clones" were not capable of interfacing the new PS/2 mouse. Eventually (around the late 80486 and early Pentium time frame) PS/2 style mice became popular, and "PC-compatible" controllers have supported dual channels from then on (nominally one keyboard and one mouse).

For the keyboard functions proper, the PS2 and AT controllers are very similar. The adjunction of a second channel (for the mouse) has forced however to redefine a few status and control bits.

Translation

The original IBM-PC keyboards (using the old XT interface) used "scan code set 1". The new AT keyboards generated different scan codes, or "scan code set 2". This change would have created compatibility problems for software that was expecting different scan codes from the keyboard. To avoid the compatibility problem, the keyboard controller supports a translation mode. If translation is enabled the controller will translate "scan code set 2" into "scan code set 1".

Whenever this translation is enabled (and by default, it is) there is no way to reverse it in software. For example, if you receive the byte 0xB5 from the controller, then you can't know if the original data (sent to the controller by the device) was the byte 0xB5; or if it was the two bytes 0xF0, 0x33; or if it was the two bytes 0xF0, 0xB3.

For software to actually use "scan code set 2" (or the even newer, rarely used, "scan code set 3"), or to allow different types of devices to be used in the keyboard port, you need to disable this translation to avoid having the data from the device mangled.

USB Legacy Support

By modern standards you will find many PCs bundled with USB input devices. Some PCs may not even have PS/2 connectors at all. To remain compatible with old software, the mainboard emulates USB Keyboards and Mice as PS/2 devices. This is called USB Legacy Support.

Because the implementation differ by manufacturer and mainboard there are flaws and sometimes even bugs:

- Some emulation layers also handle the communication with the real PS/2 connectors regardless of any connected USB device. So maybe not all capabilities of the PS/2 connectors and devices can be used. For example extended mouse modes needed to use the scroll wheel won't work or the keyboard only works on the first PS/2 connector and the mouse only on the second connector.
- The SMM BIOS that's providing the PS/2 USB Legacy Support may not support extended memory techniques or Long Mode and may cause system crashes.

This USB Legacy Support should be disabled by the OS as soon as the OS initialises the USB Controller, and this should be done before the OS attempts to initialise the real PS/2 controller. Otherwise the OS would only be initialising the emulated PS/2 controller and there's a large risk of problems caused by deficiencies in the firmware's emulation.

Buffer Naming Perspective

The PS/2 controller has two (one byte) buffers for data - one buffer for data received from devices that is waiting to be read by your OS, and one for data written by your OS that is waiting to be sent to a PS/2 device. Most datasheets for PS/2 controllers are written from the perspective of the PS/2 device and not from the perspective of software running on the host. Because of this, the names given to these buffers are the opposite of what you expect: the output buffer contains a device's output data (data waiting to be read by software), and the input buffer contains a device's input (data that was sent by software).

PS/2 Controller IO Ports

The PS/2 Controller itself uses 2 IO ports (IO ports 0x60 and 0x64). Like many IO ports, reads and writes may access completely different internal registers.

Historical note : The PC-XT PPI had used port 0x61 to reset the keyboard interrupt request signal (among other unrelated functions). Port 0x61 has no keyboard related functions on AT and PS/2 compatibles.

IO Port	Access Type	Purpose
0x60	Read/Write	Data Port
0x64	Read	Status Register
0x64	Write	Command Register

Data Port

The Data Port (IO Port 0x60) is used for reading data that was received from a PS/2 device or from the PS/2 controller itself, and writing data to a PS/2 device or to the PS/2 controller itself.

Status Register

The Status Register contains various flags that indicate the state of the PS/2 controller. The meanings for each bit are:

Bit	Meaning
0	Output buffer status (0 = empty, 1 = full) (must be set before attempting to read data from IO port 0x60)

1	Input buffer status (0 = empty, 1 = full) (must be clear before attempting to write data to IO port 0x60 or IO port 0x64)
2	System Flag Meant to be cleared on reset and set by firmware (via. PS/2 Controller Configuration Byte) if the system passes self tests (POST)
3	Command/data (0 = data written to input buffer is data for PS/2 device, 1 = data written to input buffer is data for PS/2 controller command)
4	Unknown (chipset specific) May be "keyboard lock" (more likely unused on modern systems)
5	Unknown (chipset specific) May be "receive time-out" or "second PS/2 port output buffer full"
6	Time-out error (0 = no error, 1 = time-out error)
7	Parity error (0 = no error, 1 = parity error)

Command Register

The Command Port (IO Port 0x64) is used for sending commands to the PS/2 Controller (not to PS/2 devices).

PS/2 Controller Commands

The PS/2 Controller accepts some commands and performs them. These commands should not be confused with bytes sent to a PS/2 device (e.g. keyboard, mouse).

To send a command to the controller, simply write the command byte to IO port 0x64. If there is a "next byte" (the command is 2 bytes) then the next byte needs to be written to IO Port 0x60 after making sure that the controller is ready for it (by making sure bit 1 of the Status Register is clear). If there is a response byte, then the response byte needs to be read from IO Port 0x60 after making sure that it has arrived (by making sure bit 0 of the Status Register is set).

Command Byte	Meaning	Response Byte
0x20	Read "byte 0" from internal RAM	Controller Configuration Byte (see below)
0x21 to	Read "byte N" from internal RAM (where 'N' is	Unknown (only the first byte of internal RAM has a standard

0x3F	the command byte & 0x1F)	purpose)
0x60	Write next byte to "byte 0" of internal RAM (Controller Configuration Byte, see below)	None
0x61 to 0x7F	Write next byte to "byte N" of internal RAM (where 'N' is the command byte & 0x1F)	None
0xA7	Disable second PS/2 port (only if 2 PS/2 ports supported)	None
0xA8	Enable second PS/2 port (only if 2 PS/2 ports supported)	None
0xA9	Test second PS/2 port (only if 2 PS/2 ports supported)	0x00 test passed 0x01 clock line stuck low 0x02 clock line stuck high 0x03 data line stuck low 0x04 data line stuck high
0xAA	Test PS/2 Controller	0x55 test passed 0xFC test failed
0xAB	Test first PS/2 port	0x00 test passed 0x01 clock line stuck low 0x02 clock line stuck high 0x03 data line stuck low 0x04 data line stuck high
0xAC	Diagnostic dump (real all bytes of internal RAM)	Unknown
0xAD	Disable first PS/2 port	None
0xAE	Enable first PS/2 port	None
0xC0	Read controller input port	Unknown (none of these bits have a standard/defined purpose)
0xC1	Copy bits 0 to 3 of input port to status bits 4 to 7	None
0xC2	Copy bits 4 to 7 of input port to status bits 4 to 7	None
0xD0	Read Controller Output Port	Controller Output Port (see below)
0xD1	Write next byte to Controller Output Port (see below) Note: Check if output buffer is empty first	None
0xD2	Write next byte to first PS/2 port output buffer (only if 2 PS/2 ports supported) (makes it look like the byte written was received from the first PS/2 port)	None

0xD3	Write next byte to second PS/2 port output buffer (only if 2 PS/2 ports supported) (makes it look like the byte written was received from the second PS/2 port)	None
0xD4	Write next byte to second PS/2 port input buffer (only if 2 PS/2 ports supported) (sends next byte to the second PS/2 port)	None
0xF0 to 0xFF	Pulse output line low for 6 ms. Bits 0 to 3 are used as a mask (0 = pulse line, 1 = don't pulse line) and correspond to 4 different output lines. Note: Bit 0 corresponds to the "reset" line. The other output lines don't have a standard/defined purpose.	None

Note: Command bytes not listed in the table above should be treated as either "chipset specific" or "unknown" and shouldn't be issued. Commands bytes that are marked as "only if 2 PS/2 ports supported" should also be treated as either "chipset specific" or "unknown" if the controller only supports one PS/2 port.

PS/2 Controller Configuration Byte

Commands 0x20 and 0x60 let you read and write the PS/2 Controller Configuration Byte. This configuration byte has the following format:

Bit	Meaning
0	First PS/2 port interrupt (1 = enabled, 0 = disabled)
1	Second PS/2 port interrupt (1 = enabled, 0 = disabled, only if 2 PS/2 ports supported)
2	System Flag (1 = system passed POST, 0 = your OS shouldn't be running)
3	Should be zero
4	First PS/2 port clock (1 = disabled, 0 = enabled)
5	Second PS/2 port clock (1 = disabled, 0 = enabled, only if 2 PS/2 ports supported)
6	First PS/2 port translation (1 = enabled, 0 = disabled)
7	Must be zero

Note: Bits listed in the table above as "unknown" should be treated as either "chipset specific" or "unknown". Bits that are marked as "only if 2 PS/2 ports supported" should also be treated as either "chipset specific" or "unknown" if the controller only supports one PS/2 port.

PS/2 Controller Output Port

Commands 0xD0 and 0xD1 let you read and write the PS/2 Controller Output Port. This output port has the following format:

Bit	Meaning
0	System reset (output) **WARNING** always set to '1'. You need to pulse the reset line (e.g. using command 0xFE), and setting this bit to '0' can lock the computer up ("reset forever").
1	A20 gate (output)
2	Second PS/2 port clock (output, only if 2 PS/2 ports supported)
3	Second PS/2 port data (output, only if 2 PS/2 ports supported)
4	Output buffer full with byte from first PS/2 port (connected to IRQ1)
5	Output buffer full with byte from second PS/2 port (connected to IRQ12, only if 2 PS/2 ports supported)
6	First PS/2 port clock (output)
7	First PS/2 port data (output)

Note: Bits that are marked in the table above as "only if 2 PS/2 ports supported" should be treated as either "chipset specific" or "unknown" if the controller only supports one PS/2 port.

Initialising the PS/2 Controller

Some people assume the PS/2 controller exists and was configured correctly by firmware. This approach can work, but isn't very robust and doesn't correctly support "less simple" scenarios. Examples of why this approach may not work well include:

- Something (e.g. a Boot Manager) left the PS/2 Controller in a dodgy state
- The PS/2 Controller has hardware faults and your OS didn't do any testing
- There's a USB keyboard and a PS/2 mouse, and the BIOS didn't bother initialising the PS/2 controller because it was using USB Legacy Support and not using the mouse
- You want to reliably send data to the second PS/2 device on older hardware and have to know the second PS/2 port exists (see the warning for "Sending Bytes To The Second PS/2 Port" below).

The following steps are for "comprehensive PS/2 Controller initialisation". It may be excessive for your purposes, and a more limited version of it may be more suitable. However, it's easy enough to (selectively) remove steps from the following description.

Step 1: Initialise USB Controllers

This doesn't have anything to do with the PS/2 Controller or PS/2 Devices, however if the system is using (typically limited/dodgy) USB Legacy Support it will interfere with PS/2 Controller initialisation. Therefore you need to initialise USB controllers and disable USB Legacy Support beforehand.

Step 2: Determine if the PS/2 Controller Exists

Before you touch the PS/2 controller at all, you should determine if it actually exists. On some systems (e.g. 80x86 Apple machines) it doesn't exist and any attempt to touch it can result in a system crash. The correct way to do this is is with ACPI. More specifically, check bit 1 (value = 2, the "8042" flag) in the "IA PC Boot Architecture Flags" field at offset 109 in the Fixed ACPI Description Table (FADT). If this bit is clear then there is no PS/2 Controller to configure. Otherwise, if the bit is set or the system doesn't support ACPI (no ACPI tables and no FADT) then there is a PS/2 Controller.

Step 3: Disable Devices

So that any PS/2 devices can't send data at the wrong time and mess up your initialisation; start by sending a command 0xAD and command 0xA7 to the PS/2 controller. If the controller is a "single channel" device, it will ignore the "command 0xA7".

Step 4: Flush The Output Buffer

Sometimes (e.g. due to interrupt controller initialisation causing a lost IRQ) data can be stuck in the PS/2 controller's output buffer. To guard against this, now that the devices are disabled (and can't send more data to the output buffer) it can be a good idea to flush the controller's output buffer. There's 2 ways to do this - poll bit 0 of the Status Register (while reading from IO Port 0x60 if/when bit 0 becomes set), or read from IO Port 0x60 without testing bit 0. Either way should work (as you're discarding the data and don't care what it was).

Step 5: Set the Controller Configuration Byte

Because some bits of the Controller Configuration Byte are "unknown", this means reading the old value (command 0x20), changing some bits, then writing the modified value back (command 0x60). You want to disable all IRQs and disable translation (clear bits 0, 1 and 6).

While you've got the Configuration Byte, test if bit 5 was set. If it was clear then you know it can't be a "dual channel" PS/2 controller (because the second PS/2 port should be disabled).

Step 6: Perform Controller Self Test

To test the PS/2 controller, send command 0xAA to it. Then wait for its response and check that it replied with 0x55.

Step 7: Determine If There Are 2 Channels

If you know it's a single channel controller (from Step 5) then skip this step. Otherwise, send a command 0xA8 to enable the second PS/2 port and read the Controller Configuration Byte again. Now bit 5 of the Controller Configuration Byte should be clear - if it's set then you know it can't be a "dual channel" PS/2 controller (because the second PS/2 port should be enabled). If it is a dual channel device, send a command 0xA7 to disable the second PS/2 port again.

Step 8: Perform Interface Tests

This step tests the PS/2 ports. Use command 0xAB to test the first PS/2 port, then check the result. Then (if it's a "dual channel" controller) use command 0xA9 to test the second PS/2 port, then check the result.

At this stage, check to see how many PS/2 ports are left. If there aren't any that work you can just give up (display some errors and terminate the PS/2 Controller driver). *Note: If one of the PS/2 ports on a dual PS/2 controller fails, then you can still keep going and use/support the other PS/2 port.*

Step 9: Enable Devices

Enable any PS/2 port that exists and works using command 0xAE (for the first port) and command 0xA8 (for the second port). If you're using IRQs (recommended), also enable interrupts for any (usable) PS/2 ports in the Controller Configuration Byte (set bit 0 for the first PS/2 port, and/or bit 1 for the second PS/2 port, then set it with command 0x60).

Step 10: Reset Devices

All PS/2 devices should support the "reset" command (which is a command for the device, and not a command for the PS/2 Controller). To send the reset, just send the byte 0xFF to each (usable) device. The device/s will respond with 0xFA (success) or 0xFC (failure), or won't respond at all (no device present). If your code supports "hot-plug PS/2 devices" (see later), then you can assume each device is "not present" and let the hot-plug code figure out that the device is present if/when 0xFA or 0xFC is received on a PS/2 port.

Detecting PS/2 Device Types

All PS/2 devices should support the "identify" command and the "disable scanning" command (which are commands for the device, and not commands for the PS/2 Controller). The device should respond to the "identify" command by sending a sequence of none, one or two identification bytes. However, if you just send the "identify" command you can't prevent the response from the "identify" command from being mixed up with keyboard/mouse data. To fix this problem, you need to send the "disable scanning" command first. Disabling scanning means that the device ignores the user (e.g. keyboards ignore keypresses, mice ignore mouse movement and button presses, etc) and won't send data to mess your device identification code up.

The full sequence is:

- Send the "disable scanning" command 0xF5 to the device
- Wait for device to send "ACK" back (0xFA)
- Send the "identify" command 0xF2 to the device
- Wait for device to send "ACK" back (0xFA)
- Wait for device to send up to 2 bytes of reply, with a time-out to determine when it's finished (e.g. in case it only sends 1 byte)

A partial list of responses includes:

Byte/s	Device Type
None	Ancient AT keyboard with translation enabled in the PS/Controller (not possible for the second PS/2 port)

0x00	Standard PS/2 mouse
0x03	Mouse with scroll wheel
0x04	5-button mouse
0xAB, 0x41 or 0xAB, 0xC1	MF2 keyboard with translation enabled in the PS/Controller (not possible for the second PS/2 port)
0xAB, 0x83	MF2 keyboard

Note: If anyone sees any other responses please add to the list above!

Once your PS/2 Controller driver knows what types of PS/2 devices are present, it can start suitable device drivers for those devices. Don't forget that we've left devices in a "scanning disabled" state.

Hot Plug PS/2 Devices

WARNING: PS/2 was never intentionally designed to support hot-plug. Usually it is fine as most PS/2 controllers have reasonably robust IO lines, however some PS/2 controllers (mostly those in old chipsets) may potentially be damaged.

Despite the warning, most OSs (Windows, Linux, etc) do support hot-plug PS/2. It is also relied on by old "mechanical switch" KVMs (which allow the same PS/2 devices to be shared by multiple computers by effectively disconnecting the device from one computer and connecting it to the next).

When a PS/2 device is removed the PS/2 controller won't know. To work around this, when no data has been received from the device for some length of time (e.g. 2 seconds), an OS can periodically test for the presence of the device by sending an "echo" command to the device and checking for a reply. If the device doesn't respond, then assume the device has been unplugged.

When a PS/2 device is first powered up (e.g. when it is plugged in to a PS/2 port), the device should perform its Basic Assurance Test and then attempt to send a "BAT completion code". This means that software (e.g. an OS) can automatically detect when a PS/2 device has been inserted. *Note: If a device is removed and then another device (or the same device) is plugged in quickly enough, the software may not have had time to detect the removal.*

When software detects that a device was plugged in it can determine the type of device (see above). If the device was the same type as before software can re-configure it so that the device is in the same state as it was before removal. This means that (for example) someone using an old "mechanical switch" KVMs doesn't lose state (things like keyboard LEDs, typematic rate, etc) when switching between computers. If the device is not the same as before or there was no previously connected device, then software may need to start a new device driver (and terminate the old device driver, if any).

Sending Bytes To Device/s

Unfortunately, the PS/2 Controller does not support interrupt driven transmission (e.g. you can't have a queue of bytes waiting to be sent and then send each byte from inside a "transmitter empty" IRQ handler). Fortunately very little data needs to be sent to typical PS/2 devices and polling suffices.

First PS/2 Port

To send data to the first PS/2 Port:

- Set up some sort of timer or counter to use as a time-out
- Poll bit 1 of the Status Register ("Input buffer empty/full") until it becomes clear, or until your time-out expires
- If the time-out expired, return an error
- Otherwise, write the data to the Data Port (IO port 0x60)

Second PS/2 Port

Sending data to the second PS/2 port is a little more complicated, as you need to send a command to the PS/2 controller to tell it that you want to talk to the second PS/2 port instead of the first PS/2 port. To send data to the second PS/2 Port:

- Write the command 0xD4 to IO Port 0x64
- Set up some sort of timer or counter to use as a time-out
- Poll bit 1 of the Status Register ("Input buffer empty/full") until it becomes clear, or until your time-out expires
- If the time-out expired, return an error
- Otherwise, write the data to the Data Port (IO port 0x60)

WARNING: If the PS/2 controller is an (older) "single PS/2 device only" controller, if you attempt to send a byte to the second PS/2 port the controller will ignore the command 0xD4 you send to IO Port 0x64, and therefore the byte you send will actually be sent to the first PS/2 device. This means that (if you support older hardware) to reliably send data to the second device you have to know that the PS/2 Controller actually does have a second PS/2 port.

Recieving Bytes From Device/s

There are 2 ways to receive bytes from device/s: polling, and using IRQ.

Polling

To poll, wait until bit 0 of the Status Register becomes set, then read the received byte of data from IO Port 0x60.

There are 2 major problems with polling. The first problem is that (like all polling) it wastes a lot of CPU time for nothing. The second problem is that if the PS/2 controller supports two PS/2 devices there's no way to reliably determine which device sent the byte that you've received, unless one of them is disabled and unable to send data.

Note: if the PS/2 controller uses bit 5 of the Status Register as a "second PS/2 port output buffer full" flag, you'd still have problems trying to determine which device sent a byte of data you've received without race conditions. For example, there may be data from the second PS/2 device waiting for you

when you check the flag, but before you read from IO Port 0x60 data from the first PS/2 device might arrive and you might read data from the first PS/2 device when you think you're reading data from the second PS/2 device. Of course there's also no easy way to know if the PS/2 controller does use bit 5 of the Status Register as a "second PS/2 port output buffer full" flag.

Interrupts

Using interrupts is actually easy. When IRQ1 occurs you just read from IO Port 0x60 (there is no need to check bit 0 in the Status Register first), send the EOI to the interrupt controller and return from the interrupt handler. You know that the data came from the first PS/2 device because you received an IRQ1.

When IRQ12 occurs you just read from IO Port 0x60 (there is no need to check bit 0 in the Status Register first), send the EOI to the interrupt controller/s and return from the interrupt handler. You know that the data came from the second PS/2 device because you received an IRQ12.

Unfortunately there is one problem to worry about. If you send a command to the PS/2 controller that involves a response, the PS/2 controller will put a "response byte" into the buffer and won't generate any IRQ (because the byte didn't come from any PS/2 device). In this case you have to poll, and if you have to poll you can't determine where the byte came from unless all PS/2 devices are disabled. Fortunately you should never need to send a command to the PS/2 controller itself after initialisation (and you can disable both PS/2 devices where necessary during initialisation).

CPU Reset

To trigger a CPU Reset, regardless of what state the CPU is currently in, write the value 0xFE to the Output port.

```
;Wait for a empty Input Buffer
wait1:
in   al, 0x64
test al, 00000010b
jne  wait1

;Send 0xFE to the keyboard controller.
mov  al, 0xFE
out  0x64, al
```

See Also

- PS/2
- PL050 PS/2 Controller (ARM)
- PS/2 Keyboard
- PS/2 Mouse

Threads

- PS/2 controller initialisation

External Links

- SMS "8042" Datasheet (http://www.diakom.ru/el/elfirms/datashts/Smsc/42w11.pdf)

Categories: X86 | Common Devices
