Nano-Kernel : A Bare Metal OS

## Part 8 - The PC Keyboard Controller

The original Intel PC included an i8042, an early programmable chip that is an ancestor of the i8048 and i8051 microcontrollers. The original i8042 worked as a programmable interface between the CPU and the serial keyboard and mice. Backward compatibility is still maintained, and even when using a modern USB keyboard, modern computers provide “legacy mode” support for the i8042. As usual, there is more detailed documentation available on [OS Dev](https://wiki.osdev.org/%228042%22_PS/2_Controller).

Just like the i8259, there are a few 8-bit I/O ports that are accessible using in and out instructions:

|  |  |  |
| --- | --- | --- |
| I/O port address | Function | Read / Write |
| 0x60 | Data Register | Read and Write |
| 0x64 | Status Register | Read |
| 0x64 | Command Register | Write |

Reads from the status register (inb 0x64) show the state of the controller chip. This is especially important because the controller chip does not generate interrupts on its own. Thus, the only way to detect a change in the status of the chip, for example, waiting a command to finish processing and make its result available in the data register, is to poll the status register until the output buffer status becomes ready.

**8042 Status Register Input Port 0x64**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| A0 | 7 | 6 | 5 …. 4 | 3 | 2 | 1 | 0 |
| 0x64 | PE | TO | RESERVED | CORD | SF | IBS | OBS |

**PE** – Parity Error

* 1 – A parity error was detected on the serial port
* 0 – No parity error detected

**PE** – Time-Out Error

* 1 – A time-out occurred waiting for PS/2 device
* 0 – No time-out error detected

**CORD** – Command or Data

* 1 – Data written to input buffer (0x60) is command for controller
* 0 – Data written to input buffer (0x60) is data for PS/2 device

**SF** – System Flag

* 1 – Set by firmware after system has passed power-on self-test (POST)
* 0 – Cleared on reset

**IBS** – Input Buffer Status

* 1 – Input (from CPU to 8042) is full, do not write more data to 0x60
* 0 – Input buffer is empty, may write data to 0x60

**OBS** – Output Buffer Status

* 1 – Output (from 8042 to CPU) is ready, may read from 0x60
* 0 – Output buffer is empty, do not read from 0x60

### Controller Configuration

The i8042 controller has a number of configuration registers and they are fully documented elsewhere.[[1]](#footnote-1) There are a handful of registers that we will need to use and will be documented here. Before we do that, we need to determine how to read and write those registers.

|  |  |  |
| --- | --- | --- |
| Command Byte | Description | Response |
| 0x20 | Read Controller Configuration Byte | CCB in data port |
| 0xA7 | Disable second PS/2 port | None |
| 0xA8 | Enable second PS/2 port | None |
| 0xAA | Test PS/2 controller | 0x55 – pass, 0xFC - fail |
| 0xAD | Disable first PS/2 port | None |
| 0xAE | Enable first PS/2 port | None |

Sending a Command w/ No Response

1. *Poll* the *Status Port* (0x64) until the IBS bit is clear.
2. *Output* the i8042 register number to be read to the *Command Port (0x64).*

Requesting a Register Read

1. *Poll* the *Status Port* (0x64) until the IBS bit is clear.
2. *Output* the i8042 register number to be read to the *Command Port (0x64).*
3. *Poll* the *Status Port* (0x64) until the OBS bit is set.
4. *Input* the value reported by the i8042 by reading the *Data Port* (0x60)

Requesting a Register Write

1. Poll the Status Port (0x64) until the IBS bit is clear.
2. Output the i8042 register number to be read to the Command Port (0x64).
3. Poll the Status Port (0x64) until the IBS bit is clear.
4. Output the new register value by writing it to the Data Port (0x60)

There seems to be an opportunity for race conditions that can be cleared by adding some delay between the output of the register number and polling the status bit. Its at least theoretically possible that the status bit will remain clear before the new data is actually available. When I added a “busy-wait” loop of a few thousand iterations, I actually had much better and consistent results.

#### Controller Configuration Byte

One of the most important registers for our purposes is the “controller configuration byte,” which can be read from controller register 0x20, and written to register 0x60. Using different register numbers for a read and write was actually a pretty common strategy in older electronics, so its not much of a surprise to see it employed here. The fields of the configuration byte are shown below:

**PS/2 Controller Configuration Byte**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | XLAT1 | CLK2 | CLK1 | 0 | SF | INT2 | INT1 |

**XLAT1** – First PS/2 Port Translation

* 1 – Translation is enabled
* 0 – Translation is disabled

**CLK2** – Second PS/2 Port Clock

* 1 – Clock is disabled
* 0 – Clock is enabled

**CLK1** – First PS/2 Port Clock

* 1 – Clock is disabled
* 0 – Clock is enabled

**SF** – System Flag

* 1 – System
* 0 – Data written to input buffer (0x60) is data for PS/2 device

**SF** – System Flag

* 1 – Set by firmware after system has passed power-on self-test (POST)
* 0 – Cleared on reset

**INT2** – Second PS/2 Port Interrupt Generation

* 1 – Second PS/2 port will raise interrupts to i8259
* 0 – Interrupt disabled

**INT1** – First PS/2 Port Interrupt Generation

* 1 – First PS/2 port will raise interrupts to i8259
* 0 – Interrupt disabled

The most important features of this register are the control of the interrupts. Disabling interrupts effectively suppresses the normal behavior of this chip and will allow us to configure it without creating spurious interrupts to the rest of the keyboard driver system.

### PS/2 Controller Initialization

The minimum viable controller initialization will logically look like:

1. Disable system interrupts
2. Disable PS2 ports (send i8042 command 0xAD and 0xA7)
3. Flush any data on the PS2 controller by reading from input 0x60 until the status register is clear.
4. Disable interrupts and translation in the controller configuration byte
5. Perform a self-test to make sure there is a keyboard (0xAA)
6. Register an interrupt handler for interrupt #33 / IRQ 1
7. Enable PIC interrupt IRQ1
8. Enable the PS/2 port (send command 0xAE)
9. Enable interrupt generation in controller configuration byte
10. Enable system interrupts

### PS/2 Driver

1. Create a “ps2.c” and “ps2.h” file
2. Add a “nop” intrinsic to your i386.h file. Include this in your ps2.c file.
3. Implement a delay function that will delay for a few thousand no-ops (in a for loop).
4. Implement a “write command” function that will take a uint8\_t command byte and output it to 0x64 (again, using your intrinsics)
5. Implement a “read status” function that will input the value from 0x64 (using your intrinsics) and return that value.
6. Implement a “write data” function that will write a data byte to port 0x60, using your intrinsics.
7. Implement a “read data” function that will read a data byte from port 0x60 and return it.
8. Implement a “write register” function that will take a command byte (uint8\_t), and perform the steps described previously in “Sending a Command w/ no response.” You should add a call to your delay function after the status bit has become clear.
9. Implement a “write register2” function that will write two command bytes – it should call your write register with the first command byte, and then call your write data function with the second command byte, but adding appropriate status checks and delays.
10. Implement a “read register” function that will read a byte from a register following the steps described previously in “Requesting a Register Read”
11. Implement a “read data” function that will wait for the data port to become ready (polling the status byte) and then reading that data and returning the value.
12. Implement a function “disable ports” that will disable all of the PS2 ports (by sending the commands to the proper registers and reusing the previous functions).
13. Implement a “flush” function that will keep reading data from the data register until the status register indicates that its clear.
14. Implement a “self test” function that will write send the self test command and then read the result, and returning an error if the self test failed.
15. Create an interrupt handler, “void handler\_name(int intr\_num)” that will registered with the PIC interrupt handler to be called whenever the keyboard interrupt is fired. For now, read the data from the i8042 data register and print it with your kprintf.
16. Finally, create an “initialization” function that will:
    1. disable system interrupts
    2. disable the PS2 ports (see your previous function)
    3. flush the data port
    4. turn off interrupts in the command configuration byte – you don’t want to touch the other bits, so you’ll need to first read the command configuration byte, turn off the interrupt bits and just those bits and write the changed value back out.
    5. Perform the self-test (using your previous function) and reporting the error if it was detected
    6. Register your interrupt handler by calling “register\_handler(33, your\_handler)” (note, you’ll need to include that header file)
    7. Enable the PS2 ports (using your previous function)
    8. Enable IRQ1 by calling “pic\_enable\_interrupt(IRQ1)”, but you’ll need to include that header file too.
    9. Enable the system interrupts
17. Edit your ps2.h file and expose only the ps2\_init function, this is the only function that the rest of your OS should call directly. That means that each of the other functions in your .c file should be marked as static so that they cannot be called by other parts of your OS.

Before compiling the OS, I found that to test the keyboard that I needed to modify my kernel main:

1. Add the call to the ps2 initialization function that you just wrote
2. Prevent kernel main from returning which halts the processor by adding an infinite loop at the end: while(1) { nop(); } , which will use the nop intrinsic that you just wrote!
3. You **should** be able to type keys and see the interrupt fire and display the data from the registers.

#### Scan Codes vs. Characters

Now, you are ready to test your kernel – build and run it. You should be able to type keys in the QEMU window and see that the interrupt is reporting that its been triggered. While testing your PS/2 driver, you should quickly realize that the numbers that are displayed by the interrupt handler having nothing in common with what you are typing. For example, pressing letter “a” on the keyboard should be either 0x61 or 0x41 (a or A), but that’s not what you see. Instead, you should see:

* 0x1C when you press the key
* 0xF0, 0x1C when you release the key

So, pressing and releasing a key generates three interrupts! This is not an error. What you are seeing is the keyboard sending “scan codes” which represent a change in the keyboard state. For example, if you press the num-lock key, instead of toggling the numeric keys on the keypad, the keyboard simply sends scan code 0x77, then when you release it, you get 0xf0, 0x77. It doesn’t even change the numeric keypad, and in fact, the num lock light doesn’t even turn on! The same thing is true with caps lock. Pressing caps lock doesn’t actually make your “a” key send a different scan code. Your operating system driver has to make all of these things happen.

### Keyboard Driver

The keyboard driver will work in conjunction with the PS2 driver to receive scan codes and convert them to a stream of characters that can be read by the rest of your system (e.g. using getchar()). To do this, we need to map the scan codes to the corresponding ASCII value.

There are several different ways that this can be done, but the way one common way is to create look-up-tables (LUT) for lower-case mode, upper-case mode, and control-mode. I chose to ignore “alt-key” combos:

static const char charmap2\_lower[] = {

// 0/8 1/9 2/a 3/b 4/c 5/d 6/e 7/f

0, 0, 0, 0, 0, 0, 0, 0, // 0 - 07

0, 0, 0, 0, 0, '\t', '`', 0, // 8 - F

0, 0, 0, 0, 0, 'q', '1', 0, // 10-17

0, 0, 'z', 's', 'a', 'w', '2', 0, // 18-1F

….

To find the key for a given scan code, simply return the value in that location. For example, the scan code 0x1F corresponds to pressing the ‘2’ key on the row of keys above the letters (the keypad 2 is a different scan code), and code like:

char ascii = charmap2\_lower[ scan\_code ];

See <https://wiki.osdev.org/PS/2_Keyboard> for a complete list of the scan codes. QEMU seems to use “code page #2.” Also, if the scan code doesn’t map to an ascii table I just return 0. A more advanced keyboard driver could perform more elaborate translations, but we’re not going to do that.

#### Keyboard Driver

1. Create the three character maps: lower, upper, and control characters, each one should have entries for 0x00 through 0x7f, inclusively.
2. Create a keyboard driver that can track the “state” of the keyboard. The state includes:

* Alt-mode – the alt key was pressed but not released
* Ctrl-mode – the ctrl key was pressed but not released
* Shift-mode – the shift key was pressed but not released
* Caps-Lock mode – the caps-lock key was toggled on
* Num-Lock mode – the num-lock key was toggled on

1. The keyboard driver also needs a FIFO (see your fifo.c/fifo.h) to hold the translated keys. Create a FIFO that holds up to 32 ASCII values.
2. Build a function to lookup a code based on the current “state” of the keyboard driver. Its input arguments are the keyboard driver (holding the different modes) and the scan code. This function will lookup the scan code in the appropriate table and return its ASCII value. Pay careful attention to what happens when CapsLock is on and you press shift – you get lower-case values, so you should XOR the caps-lock mode and shift-mode to determine whether to lookup in the lower- or upper-case LUT.
3. Build a function that can be called when a new scan-code is detected. Its input arguments are the keyboard driver and the scan code, and it will add a converted character to the FIFO. One complication is that the driver needs a state machine to keep track of state of the keyboard:
4. Reading an 0xF0 will move from state 0 to state 1, to track that the next scan is the released key
5. Reading an 0xE0 will move from state 0 to state 2, to track that sometimes the 0xE0 scans will read three characters
6. Some scan codes change the state of the driver (e.g. pressing or releasing shift) while others actually should be sent for lookup to the table, based on the state of the state machine

Finally, your driver will, if the lookup character is non-zero, write it to the FIFO.

1. Write a function that returns true if the FIFO is not empty
2. Write a function that reads the next character from the FIFO and returns it
3. Write a function to initialize your driver and set everything to a sane state.

### Connecting the Keyboard and PS/2 Drivers

The two drivers are coupled together in such a way that the PS/2 driver’s interrupt function will need to inject the scanned character into the keyboard driver’s scan-code state machine function (from previous step). The most straightforward way to do this is to create and initialize the keyboard driver structure (from the previous step) and then modify the PS/2 initialization function to store that it its memory (i.e. as a static file variable ).

Finally, we are ready to test the operation of this whole system: modify the kernel main to initialize both keyboard and PS2 drivers, and then instead of the NOP loop, check if the keyboard has a key (in its FIFO) and read that key (from the FIFO) using the code you’ve already developed in the previous step. Then, print the read ASCII key to the keyboard.

At this point, it may be helpful to turn off extraneous output from the interrupt services.

# Deliverables and Demos

Arrange a time for us to meet, and show be prepared to show me the following:

1. Demonstrate that you are able to properly detect the keyboard (if present)
2. Show me how you are reading in scan codes and processing them
3. Show me how you are converting the scan codes into “ascii characters”
4. Show me how you are managing your FIFO

Points: \_\_\_\_\_\_ / 50

1. https://wiki.osdev.org/%228042%22\_PS/2\_Controller#PS.2F2\_Controller\_IO\_Ports [↑](#footnote-ref-1)