Chapter 4

Output

This chapter will present how to display text on the console as well as writing data to the serial port. Furthermore, we will create our first *driver*, that is, code that acts as a layer between the kernel and the hardware, providing a higher abstraction than communicating directly with the hardware. The first part of this chapter is about creating a driver for the *framebuffer* [26] to be able to display text on the console. The second part shows how to create a driver for the serial port. Bochs can store output from the serial port in a file, effectively creating a logging mechanism for the operating system.

Interacting with the Hardware

There are usually two different ways to interact with the hardware, *memory-mapped I/O* and *I/O ports*.

If the hardware uses memory-mapped I/O then you can write to a specific memory address and the hardware will be updated with the new data. One example of this is the framebuffer, which will be discussed in more detail later. For example, if you write the value `0x410F` to address `0x000B8000`, you will see the letter A in white color on a black background (see the section on the framebuffer for more details).

If the hardware uses I/O ports then the assembly code instructions `out` and `in` must be used to communicate with the hardware. The instruction `out` takes two parameters: the address of the I/O port and the data to send. The instruction `in` takes a single parameter, the address of the I/O port, and returns data from the hardware. One can think of I/O ports as communicating with hardware the same way as you communicate with a server using sockets. The cursor (the blinking rectangle) of the framebuffer is one example of hardware controlled via I/O ports on a PC.

The Framebuffer

The framebuffer is a hardware device that is capable of displaying a buffer of memory on the screen [26]. The framebuffer has 80 columns and 25 rows, and the row and column indices start at 0 (so rows are labelled 0 - 24).

Writing Text

Writing text to the console via the framebuffer is done with memory-mapped I/O. The starting address of the memory-mapped I/O for the framebuffer is `0x000B8000` [27]. The memory is divided into 16 bit cells, where the 16 bits determine both the character, the foreground color and the background color. The highest eight bits is the ASCII [28] value of the character, bit 7 - 4 the background and bit 3 - 0 the foreground, as can be seen in the following figure:

```
Bit:     | 15 14 13 12 11 10 9 8 | 7 6 5 4 | 3 2 1 0 |
Content: | ASCII                 | FG      | BG      |
```

The available colors are shown in the following table:

Color	Value	Color	Value	Color	Value	Color	Value
Black	0	Red	4	Dark grey	8	Light red	12
Blue	1	Magenta	5	Light blue	9	Light magenta	13
Green	2	Brown	6	Light green	10	Light brown	14
Cyan	3	Light grey	7	Light cyan	11	White	15

The first cell corresponds to row zero, column zero on the console. Using an ASCII table, one can see that A corresponds to 65 or `0x41`. Therefore, to write the character A with a green foreground (2) and dark grey background (8) at place (0,0), the following assembly code instruction is used:

```
    mov [0x000B8000], 0x4128
```

The second cell then corresponds to row zero, column one and its address is therefore:

```
    0x000B8000 + 16 = 0x000B8010
```

Writing to the framebuffer can also be done in C by treating the address `0x000B8000` as a char pointer, `char *fb = (char *) 0x000B8000`. Then, writing A at place (0,0) with green foreground and dark grey background becomes:

```
    fb[0] = 'A';
    fb[1] = 0x28;
```

The following code shows how this can be wrapped into a function:

```
    /** fb_write_cell:
     *  Writes a character with the given foreground and background to position i
     *  in the framebuffer.
     *
     *  @param i  The location in the framebuffer
     *  @param c  The character
     *  @param fg The foreground color
     *  @param bg The background color
     */
    void fb_write_cell(unsigned int i, char c, unsigned char fg, unsigned char bg)
    {
        fb[i] = c;
        fb[i + 1] = ((fg & 0x0F) << 4) | (bg & 0x0F)
    }
```

The function can then be used as follows:

```
#define FB_GREEN     2
#define FB_DARK_GREY 8

fb_write_cell(0, 'A', FB_GREEN, FB_DARK_GREY);
```

Moving the Cursor

Moving the cursor of the framebuffer is done via two different I/O ports. The cursor's position is determined with a 16 bits integer: 0 means row zero, column zero; 1 means row zero, column one; 80 means row one, column zero and so on. Since the position is 16 bits large, and the `out` assembly code instruction argument is 8 bits, the position must be sent in two turns, first 8 bits then the next 8 bits. The framebuffer has two I/O ports, one for accepting the data, and one for describing the data being received. Port `0x3D4` [29] is the port that describes the data and port `0x3D5` [29] is for the data itself.

To set the cursor at row one, column zero (position `80 = 0x0050`), one would use the following assembly code instructions:

```
out 0x3D4, 14      ; 14 tells the framebuffer to expect the highest 8 bits of the position
out 0x3D5, 0x00    ; sending the highest 8 bits of 0x0050
out 0x3D4, 15      ; 15 tells the framebuffer to expect the lowest 8 bits of the position
out 0x3D5, 0x50    ; sending the lowest 8 bits of 0x0050
```

The `out` assembly code instruction can't be executed directly in C. Therefore it is a good idea to wrap `out` in a function in assembly code which can be accessed from C via the cdecl calling standard [25]:

```
global outb             ; make the label outb visible outside this file

; outb - send a byte to an I/O port
; stack: [esp + 8] the data byte
;        [esp + 4] the I/O port
;        [esp    ] return address
outb:
    mov al, [esp + 8]    ; move the data to be sent into the al register
    mov dx, [esp + 4]    ; move the address of the I/O port into the dx register
    out dx, al           ; send the data to the I/O port
    ret                  ; return to the calling function
```

By storing this function in a file called `io.s` and also creating a header `io.h`, the `out` assembly code instruction can be conveniently accessed from C:

```
#ifndef INCLUDE_IO_H
#define INCLUDE_IO_H

/** outb:
 *  Sends the given data to the given I/O port. Defined in io.s
 *
 *  @param port The I/O port to send the data to
 *  @param data The data to send to the I/O port
 */
void outb(unsigned short port, unsigned char data);

#endif /* INCLUDE_IO_H */
```

Moving the cursor can now be wrapped in a C function:

```
#include "io.h"

/* The I/O ports */
#define FB_COMMAND_PORT         0x3D4
#define FB_DATA_PORT            0x3D5

/* The I/O port commands */
#define FB_HIGH_BYTE_COMMAND    14
#define FB_LOW_BYTE_COMMAND     15

/** fb_move_cursor:
 *  Moves the cursor of the framebuffer to the given position
 *
 *  @param pos The new position of the cursor
 */
void fb_move_cursor(unsigned short pos)
{
    outb(FB_COMMAND_PORT, FB_HIGH_BYTE_COMMAND);
    outb(FB_DATA_PORT,    ((pos >> 8) & 0x00FF));
    outb(FB_COMMAND_PORT, FB_LOW_BYTE_COMMAND);
    outb(FB_DATA_PORT,    pos & 0x00FF);
}
```

The Driver

The driver should provide an interface that the rest of the code in the OS will use for interacting with the framebuffer. There is no right or wrong in what functionality the interface should provide, but a suggestion is to have a `write` function with the following declaration:

```
int write(char *buf, unsigned int len);
```

The `write` function writes the contents of the buffer `buf` of length `len` to the screen. The `write` function should automatically advance the cursor after a character has been written and scroll the screen if necessary.

The Serial Ports

The serial port [30] is an interface for communicating between hardware devices and although it is available on almost all motherboards, it is seldom exposed to the user in the form of a DE-9 connector nowadays. The serial port is easy to use, and, more importantly, it can be used as a logging utility in Bochs. If a computer has support for a serial port, then it usually has support for multiple serial ports, but we will only make use of one of the ports. This is because we will only use the serial ports for logging. Furthermore, we will only use the serial ports for output, not input. The serial ports are completely controlled via I/O ports.

Configuring the Serial Port

The first data that need to be sent to the serial port is configuration data. In order for two hardware devices to be able to talk to each other they must agree upon a couple of things. These things include:

- The speed used for sending data (bit or baud rate)
- If any error checking should be used for the data (parity bit, stop bits)
- The number of bits that represent a unit of data (data bits)

Configuring the Line

Configuring the line means to configure how data is being sent over the line. The serial port has an I/O port, the *line command port*, that is used for configuration.

First the speed for sending data will be set. The serial port has an internal clock that runs at 115200 Hz. Setting the speed means sending a divisor to the serial port, for example sending 2 results in a speed of `115200 / 2 = 57600` Hz.

The divisor is a 16 bit number but we can only send 8 bits at a time. We must therefore send an instruction telling the serial port to first expect the highest 8 bits, then the lowest 8 bits. This is done by sending `0x80` to the line command port. An example is shown below:

```
#include "io.h" /* io.h is implement in the section "Moving the cursor" */

/* The I/O ports */

/* All the I/O ports are calculated relative to the data port. This is because
 * all serial ports (COM1, COM2, COM3, COM4) have their ports in the same
 * order, but they start at different values.
 */

#define SERIAL_COM1_BASE                0x3F8      /* COM1 base port */

#define SERIAL_DATA_PORT(base)          (base)
#define SERIAL_FIFO_COMMAND_PORT(base)  (base + 2)
#define SERIAL_LINE_COMMAND_PORT(base)  (base + 3)
#define SERIAL_MODEM_COMMAND_PORT(base) (base + 4)
#define SERIAL_LINE_STATUS_PORT(base)   (base + 5)

/* The I/O port commands */

/* SERIAL_LINE_ENABLE_DLAB:
 * Tells the serial port to expect first the highest 8 bits on the data port,
 * then the lowest 8 bits will follow
 */
#define SERIAL_LINE_ENABLE_DLAB         0x80

/** serial_configure_baud_rate:
 *  Sets the speed of the data being sent. The default speed of a serial
 *  port is 115200 bits/s. The argument is a divisor of that number, hence
 *  the resulting speed becomes (115200 / divisor) bits/s.
```

```
     *
     *  @param com      The COM port to configure
     *  @param divisor  The divisor
     */
    void serial_configure_baud_rate(unsigned short com, unsigned short divisor)
    {
        outb(SERIAL_LINE_COMMAND_PORT(com),
             SERIAL_LINE_ENABLE_DLAB);
        outb(SERIAL_DATA_PORT(com),
             (divisor >> 8) & 0x00FF);
        outb(SERIAL_DATA_PORT(com),
             divisor & 0x00FF);
    }
```

The way that data should be sent must be configured. This is also done via the line command port by sending a byte. The layout of the 8 bits looks like the following:

```
Bit:     | 7 | 6 | 5 4 3 | 2 | 1 0 |
Content: | d | b | prty  | s | dl  |
```

A description for each name can be found in the table below (and in [31]):

Name	Description
d	Enables (`d = 1`) or disables (`d = 0`) DLAB
b	If break control is enabled (`b = 1`) or disabled (`b = 0`)
prty	The number of parity bits to use
s	The number of stop bits to use (`s = 0` equals 1, `s = 1` equals 1.5 or 2)
dl	Describes the length of the data

We will use the mostly standard value `0x03` [31], meaning a length of 8 bits, no parity bit, one stop bit and break control disabled. This is sent to the line command port, as seen in the following example:

```
    /** serial_configure_line:
     *  Configures the line of the given serial port. The port is set to have a
     *  data length of 8 bits, no parity bits, one stop bit and break control
     *  disabled.
     *
     *  @param com  The serial port to configure
     */
    void serial_configure_line(unsigned short com)
    {
        /* Bit:     | 7 | 6 | 5 4 3 | 2 | 1 0 |
         * Content: | d | b | prty  | s | dl  |
         * Value:   | 0 | 0 | 0 0 0 | 0 | 1 1 | = 0x03
         */
        outb(SERIAL_LINE_COMMAND_PORT(com), 0x03);
    }
```

The article on OSDev [31] has a more in-depth explanation of the values.

Configuring the Buffers

When data is transmitted via the serial port it is placed in buffers, both when receiving and sending data. This way, if you send data to the serial port faster than it can send it over the wire, it will be buffered. However, if you send too much data too fast the buffer will be full and data will be lost. In other words, the buffers are FIFO queues. The FIFO queue configuration byte looks like the following figure:

```
Bit:     | 7 6 | 5  | 4 | 3   | 2   | 1   | 0 |
Content: | lvl | bs | r | dma | clt | clr | e |
```

A description for each name can be found in the table below:

Name	Description
lvl	How many bytes should be stored in the FIFO buffers
bs	If the buffers should be 16 or 64 bytes large
r	Reserved for future use
dma	How the serial port data should be accessed
clt	Clear the transmission FIFO buffer
clr	Clear the receiver FIFO buffer
e	If the FIFO buffer should be enabled or not

We use the value `0xC7 = 11000111` that:

- Enables FIFO
- Clear both receiver and transmission FIFO queues
- Use 14 bytes as size of queue

The WikiBook on serial programming [32] explains the values in more depth.

Configuring the Modem

The modem control register is used for very simple hardware flow control via the Ready To Transmit (RTS) and Data Terminal Ready (DTR) pins. When configuring the serial port we want RTS and DTR to be 1, which means that we are ready to send data.

The modem configuration byte is shown in the following figure:

```
Bit:     | 7 | 6 | 5  | 4  | 3   | 2   | 1   | 0   |
Content: | r | r | af | lb | ao2 | ao1 | rts | dtr |
```

A description for each name can be found in the table below:

Name	Description
r	Reserved
af	Autoflow control enabled
lb	Loopback mode (used for debugging serial ports)
ao2	Auxiliary output 2, used for receiving interrupts
ao1	Auxiliary output 1
rts	Ready To Transmit
dtr	Data Terminal Ready

We don't need to enable interrupts, because we won't handle any received data. Therefore we use the configuration value `0x03 = 00000011` (RTS = 1 and DTS = 1).

Writing Data to the Serial Port

Writing data to the serial port is done via the data I/O port. However, before writing, the transmit FIFO queue has to be empty (all previous writes must have finished). The transmit FIFO queue is empty if bit 5 of the line status I/O port is equal to one.

Reading the contents of an I/O port is done via the `in` assembly code instruction. There is no way to use the `in` assembly code instruction from C, therefore it has to be wrapped (the same way as the `out` assembly code instruction):

```
global inb

; inb - returns a byte from the given I/O port
; stack: [esp + 4] The address of the I/O port
;        [esp    ] The return address
inb:
    mov dx, [esp + 4]       ; move the address of the I/O port to the dx register
    in  al, dx              ; read a byte from the I/O port and store it in the al register
    ret                     ; return the read byte
```

```
/* in file io.h */

/** inb:
 *  Read a byte from an I/O port.
 *
 *  @param  port The address of the I/O port
 *  @return      The read byte
 */
unsigned char inb(unsigned short port);
```

Checking if the transmit FIFO is empty can then be done from C:

```
#include "io.h"
```

```
/** serial_is_transmit_fifo_empty:
 *  Checks whether the transmit FIFO queue is empty or not for the given COM
 *  port.
 *
 *  @param  com The COM port
 *  @return 0 if the transmit FIFO queue is not empty
 *          1 if the transmit FIFO queue is empty
 */
int serial_is_transmit_fifo_empty(unsigned int com)
{
    /* 0x20 = 0010 0000 */
    return inb(SERIAL_LINE_STATUS_PORT(com)) & 0x20;
}
```

Writing to a serial port means spinning as long as the transmit FIFO queue isn't empty, and then writing the data to the data I/O port.

Configuring Bochs

To save the output from the first serial serial port the Bochs configuration file `bochsrc.txt` must be updated. The `com1` configuration instructs Bochs how to handle first serial port:

```
com1: enabled=1, mode=file, dev=com1.out
```

The output from serial port one will now be stored in the file `com1.out`.

The Driver

We recommend that you implement a `write` function for the serial port similar to the `write` function in the driver for the framebuffer. To avoid name clashes with the `write` function for the framebuffer it is a good idea to name the functions `fb_write` and `serial_write` to distinguish them.

We further recommend that you try to write a `printf`-like function, see section 7.3 in [8]. The `printf` function could take an additional argument to decide to which device to write the output (framebuffer or serial).

A final recommendation is that you create some way of distinguishing the severeness of the log messages, for example by prepending the messages with `DEBUG`, `INFO` or `ERROR`.

Further Reading

- The book "Serial programming" (available on WikiBooks) has a great section on programming the serial port, http://en.wikibooks.org/wiki/Serial_Programming/8250_UART_Programming#UART_Registers
- The OSDev wiki has a page with a lot of information about the serial ports, http://wiki.osdev.org/Serial_ports