Chapter 5 Polling and Interrupt-Driven I/O

In this chapter we work with serial-port read and write operations in Assembly. We become aware of the various ways to communicate with the peripherals. Understand their advantages and disadvantages. The related files are in the following directory:

/home/TDDI11/lab/skel/lab3

Please copy them as usual to your own account. Your modifications go to "serial.asm".

1.1 Assignment

You use a program that reads characters typed on the keyboard and sends the read characters on the serial interface. Another computer will be connected to this serial communication line through its serial port. This second computer reads the port and prints the received characters on the screen. The second computer can read the keyboard and send the read characters serially to the first computer. At the same time, both computers display characters read from the serial interface. Both computers, display in a separate window, the typed character read from their own keyboards. In principle the programs for the first and second computer are identical.

You will develop assembly code to write the serial port by polling. With polling you continuously test a bit of a register. If the bit is not set, the serial interface is busy sending another character, and therefore you should not write the new character to the serial interface. If the bit is set, the serial interface is accepting new characters to be sent (transmitted).

Write the assembly code to read characters from the serial interface by interrupts. Upon receiving an interrupt, the serial interface controller notifies the CPU by means of an interrupt. The serial port share interrupt signal with other devices, so you must still check that a byte really arrived at the serial port before reading it.

Edit the file "serial.asm" to fill-in the missing details of the polled waiting-loop output function (SerialPut) and the serial input ISR (SerialISR).

In order to be able to test the program, you need to emulate two computers connected by a serial line. You may check the "makeNrun.sh" to see how this is done. You will only need to implement the methods "SerialPut" and the "SerialISR" in "serial.asm" file.

1.2 Deliverables

The file is "serial.asm". The code written in "SerialPut" and "SerialISR" sections. Demo for the lab assistant.

Fill and send in the feedback questionnaire.

1.3 Background

We discuss two ways of communicating with peripherals: polling and interrupts.

Polling means continuously checking the peripheral to detect if it has changed state. For example, if the keyboard is expected to place the ASCII code of a pressed key in a certain register of the keyboard controller, polling would imply continuously reading that register to see if the value changes. We can see immediately the disadvantage: instead of using the processor to do something useful, we waste resources checking something that happens with a low rate.

An alternative to polling is interrupt-driven communication. Instead of the processor checking the peripheral each time, the peripheral notifies the processor if its state has changed. The peripheral sends a signal that interrupts the execution of the processor. The execution jumps to a predefined address where the Interrupt Service Routine (ISR) must reside. The ISR implements the response to the interrupt. In this way, the processor does not need to frequently check the peripherals.

Nevertheless, polling may have advantages too. For example, there might be intervals when we want to ignore the peripheral and to run uninterrupted. If we used interrupt-driven communication the processor would be interrupted even if it decides not to service the interrupt. Temporarily disabling the interrupts may not be a good idea if we want not to be interrupted, because we could be interested in other interrupts.

Polling can also have advantage when communicating with a device that operate almost as fast as the processor. If we use interrupts many cycles will go to waste in the overhead of doing the interrupt, while polling may be able to read one data every iteration in the poll loop.

In such cases DMA is another solution, in which a large block of data is transferred to memory without CPU intervention, and an interrupt signal delivered when the transfer is done, and the CPU need to provide a new empty buffer. While the new buffer is filled, the CPU can process the previous buffer content in parallel. This approach requires extra hardware and must be supported in the platform. Many embedded platforms might not support this. The main part of the extra hardware is sometimes called a DMA controller.

1.4 The IBM-PC Serial port

The IBM-PC typically supports two (or more) serial ports, called COM1 and COM2. These devices are implemented using an I/O chip called a Universal Asynchronous Receiver-Transmitter (UART). UARTs convert the 8-bit parallel format used inside the PC to a serial formation in which information is transmitted one bit at a time, and vice versa. This is sometimes called serial to parallel and parallel to serial conversion.

Each serial port is assigned a block of eight I/O port addresses, starting with a "base" address of 0x2f8, 0x3F8, 0x2E8, or 0x3E8. These addresses typically correspond to COM1, COM2, COM3, and COM4 respectively. These port addresses are used with I/O instructions to write data or commands into the internal registers of the UART, or to read data or status information. The three most important registers are:

1. LSR (Line Status Register) I/O Port

Base+5 (Read-Only)

7	6	5	4	3	2	1	0
FIFO Error	Transmitter Empty	**THRE**	Break Detected	Framing Error	Parity Error	Overrun Error	**RBF**

For this lab, only two status bits are important:

Bit **5**, THRE: (Transmitter Holding Register Empty): This bit is 1 when the UART is ready to transmit another byte of data out on the serial line. Wait for this bit to become 1 before writing to THR.

Bit **0**, RBF: (Receiver Buffer Full): If this bit is 1, input data is available in RBR.

2. THR (Transmitter Holding Register) I/O Port

Base+0 (Write-Only) Data to be transmitted serially to a remote computer may be written to this port when bit THRE in the LSR is 1.

3. RBR (Receiver Buffer Register) I/O Port

Base+0 (Read-Only) Data received serially from a remote computer may be read from this port when bit RBF in the LSR is 1.

These two registers (THR and RBR), albeit being physically distinct, have the same address. System can tell them apart since THR is write-only and RBR is read-only. When writing to this address, data is copied to THR and when reading from this address, data is copied from RBR.

To write a byte to the serial port you must first make sure the device is ready. This is done by reading the Line Status Register (LSR) and extracting bit 5 which is called Transmitter Holding Register Empty (THRE). If it is "1" it means the previous value in Transmitter Holding Register (THR) has been transmitted, and you can safely proceed by writing a new value to the THR. If bit 5 is not set you must first wait for it to become set. Check/Poll it until the previous data is transmitted.

To read a byte from the serial port by using interrupts, the address to an interrupt handler must be places in the processors interrupt service vector. (In the lab this is already done for you.) In the interrupt handler you must then determine what caused the interrupt. Besides data arrival (getting a character on the serial line), other events such as "break detected", some errors, and so on may cause interrupts. To see if a new character is available on the serial port read the Line Status Register (LSR) and extract bit 0 which is called Receive Buffer Full (RBF). If it is set, new data has arrived, and you can safely proceed by fetching it from the base port. If it is not set it means the interrupt was for something else and you should do nothing (as we only care about the serial port). In both cases you have to send the End Of Interrupt (EOI) to the interrupt controller. This is done by writing 0x20 to port 0x20.

1.5 The x86 I/O Instructions

The 80x86 supports two I/O instructions: “in” and “out”. They are like very limited versions of the “mov” instruction, with the major difference that they do not access the normal memory bus (or the usual register bank), but instead a special I/O bus, with 65536 I/O ports (a port is an address on the I/O bus, but named port to distinguish it from a memory bus address). The I/O instructions take the forms:

```
in eax/ax/al, port

in eax/ax/al, dx

out port, eax/ax/al

out dx, eax/ax/al
```

The “in” instruction must always have (part of) eax as destination, and can read the port address either as a directly specified 8-bit port number (0 - 0xFF) or use the full 16-bit port number in edx. In most cases you must thus load the port number to edx first. The out instruction is exactly the same, but reverses the source and destination. Neither instruction modifies any flags in the flags register.

1.6 Resources

To better understand I/O with polling and interrupts the structure of the x86 system you can read the relevant chapters in:

http://flint.cs.yale.edu/cs422/doc/art-of-asm/pdf/

Relevant chapters are CH03-6, CH17-3, and CH22-1.