EL-GY 6483
Communications

- This homework assignment is not graded.
- You are encouraged to work in groups, and to use the Internet or any other tools available to learn the material in order to answer these questions.

---

1. The ATmega128 microcontroller includes a UART that can be used to provide a serial interface. The following code snippet is often seen in programs that use the UART interface:

```
while(!(UCSR0A & 0x20));
  UDR0 = x;
```

where `x` is a previously declared and initalized `uint8_t`; `UCSR0A` and `UDR0` are defined in header files to refer to memory locations corresponding to the USART Control and Status Register A and USART Data Register, respectively; and the UART interface has already been configured, i.e. is ready for use.

(a) Refer to page 188 of the manual for the ATmega128, (available online). What does each line of the code snippet above do, with respect to the peripheral registers? What does the code snippet as a whole do?

> **Solution:** The line
>
> ```
> while(!(UCSR0A & 0x20));
> ```
>
> waits until the USART Data Register Empty register (bit 5 in the `UCSR0A` register) is set to one, indicating that the transmit buffer is empty and ready to be written to.
> The line
>
> ```
>   UDR0 = x;
> ```
>
> places the byte in `x` in the transmit buffer, to be sent over the serial interface.
> As a whole, the code snippet sends a byte of data over the serial interface.

(b) Suppose that the serial port operates at 57600 baud and the processor operates at 8 MHz. Approximately how many processor cycles are consumed by the code snippet above?

Solution: If the transmit buffer is empty already when we run this code, the condition in the `while` loop will be false the first time it is checked (because the transmit buffer is empty already). In this scenario, it will just take a few CPU cycles to check the flag status (once) and place the byte in the transmit buffer.

But if there is a byte already in the transmit buffer immediately before we run this code, then the transmit buffer will not become empty for about 138.9 μs. During this time, the processor will keep re-evaluating the condition in the while loop over and over again, consuming about $\frac{8/57600}{1/8000000}$ (about 1112) processor cycles, and doing no *useful* work.

(c) To receive a byte over the serial port, a programmer might use the following code snippet to implement a `readByte()` function:

```
uint8_t readByte() {
  while(!(UCSR0A & 0x80));
    return UDR0;
}
```

What will happen if `readByte()` is called and there is no incoming byte over the serial interface?

Solution: The calling program will wait forever for `readByte()` to return.

(d) We say that a call to an I/O function is *blocking* if it blocks the calling program from continuing until the communication has finished. (Look up "Asynchronous I/O" on Wikipedia for more details.) Is a call to `readByte()` blocking? Why might this be problematic in some cases? Can you implement a non-blocking version of `readByte()`?

Solution:

A call to `readByte()` *is* blocking. The calling program cannot continue until a byte has been received. If no byte is received, the calling program will be blocked indefinitely.

One way to implement a non-blocking receive would be to pass an additional argument by reference, and set its value to indicate whether a byte was received. The function would return immediately, whether a byte is ready to be received or not.

```
uint8_t readByte(uint8_t *status) {
    // return immediately if there's no byte waiting in buffer
    if (!(UCSR0A & 0x80)) {
        *status = 0;
        return 0;
    }
    // otherwise, return the value in the buffer
    *status = 1;
    return UDR0;
}
```

The calling program can check the value in `status` to know whether or not a byte was received. If it wasn't, the calling program can decide whether to try again immediately, or do some other work. It isn't forced to wait and consume useless CPU cycles (and it won't wait forever, in the event that the other partner never sends a byte).

(On a microcontroller, this would not be very practical, but when programming a general-purpose system in a language that has exceptions, a non-blocking `send` is often implemented by raising an exception if the operation would block.)

After spring break, we will learn about interrupts. In interrupt-driven I/O (as opposed to polling-based I/O), a signal is generated on certain I/O events, so that the processor does not need to poll (keep checking the status of the event).

(e) On this microcontroller, the baud rate is set by writing the value $UBRR = \frac{f_{osc}}{16 B_{des}} - 1$ to a `UBRR` register, where f_{osc} is the oscillator frequency in Hz and B_{des} is the desired baud rate in bits per second. The achieved baud rate is then $B_{ach} = \frac{f_{osc}}{16(UBRR+1)}$ (See page 172-173 of the ATmega128 reference manual for more details.) Because we can only write integer values to the register, not all baud rates can be achieved exactly.

- What is the closest we can get to 57600 baud (i.e., what is B_{ach}) if f_{osc} is 8 MHz? (Assume `U2X` is 0.)
- What value should be written to the `UBRR` register to achieve this baud rate?
- What is the percent error in this case, calculated as $\left(\frac{B_{ach}}{B_{des}} - 1\right) \times 100\%$?

Solution:
Here, we calculate $UBRR = \frac{8\times10^6}{16\times57600} - 1 \approx 7.68$. Since we can only write an integer value to the `UBRR` register, we'll round to the nearest integer, which is 8.
Then, to calculate the achieved baud rate, we will use $B_{ach} = \frac{8\times10^6}{16(8+1)} \approx 55555$.
Finally, to calculate the percent error, we'll use $\left(\frac{55555}{57600} - 1\right) \times 100\% = -3.55\%$
So we have:

- The closest achievable baud rate is ≈ 55555.
- Write 8 to `UBRR` register to achieve this rate.
- The percent error is -3.55 %. (A negative percent error indicates that the baud rate is too slow.)

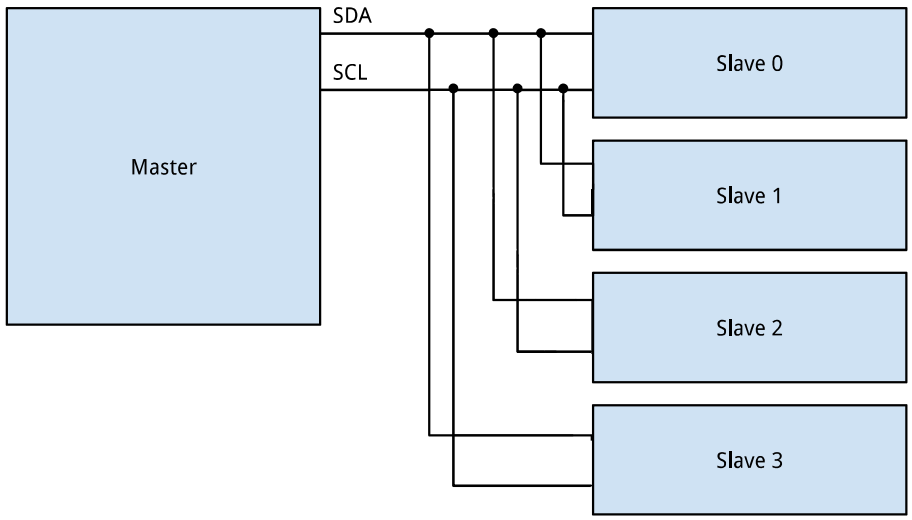
(f) Suppose the other communication partner is an ATmega128 using $f_{osc} = 2$MHz. (Assume `U2X` is 0.) What will its B_{ach} be if it tries to operate at 57600 baud? What will be the total error between the pair, and is it less than the maximum error recommended in Table 75 of the ATmega128 reference manual (page 186)?

Solution:
Here, we calculate $UBRR = \frac{2\times10^6}{16\times57600} - 1 \approx 1.17$. Since we can only write an integer value to the `UBRR` register, we'll round to the nearest integer, which is 1.
Then, to calculate the achieved baud rate, we will use $B_{ach} = \frac{2\times10^6}{16(1+1)} = 62500$.
The percent error from 57600 $\left(\frac{62500}{57600} - 1\right) \times 100\% \approx 8.5\%$ (A positive percent error indicates that the baud rate is too fast.)
The percent error from 55555 (the rate the other communication partner is operating at) is $\left(\frac{62500}{55555} - 1\right) \times 100\% \approx 12.5\%$
This device is operating 8.5% faster than it is supposed to, and 12.5% faster than the other communication partner. The total error is definitely bigger than the maximum error recommended in Table 75 for *any* configuration of data/parity/stop bits.

2. Assume you have four (slave) devices connected to a (master) microcontroller over a shared I2C bus that uses standard (7-bit) addressing and is running at 400 kHz (most I2C devices can communicate at 100 kHz or 400 kHz).

   (a) Draw a connection diagram for this configuration. What is the total number of wires?

Solution: 2 wires:



   (b) Suppose the microcontroller reads one data byte from each of the four devices sequentially. (This is similar to the single-byte read shown on page 33 of the lecture slides, but instead of a stop at the end, there is a repeated start condition followed by a different slave address.) What is the data transfer rate in (*data*) bits per second from *each device*? (In other words, over some long interval of time T, how many bytes can slave S_1 transmit?)

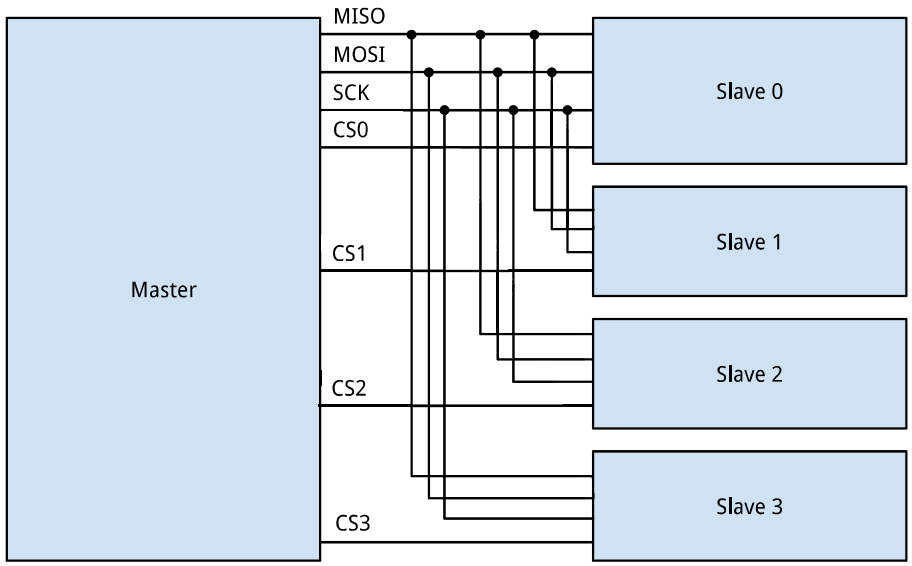
Solution:

The exact details (e.g., how long a start condition must be held, required idle time on the bus) will depend on the timing information given in the datasheet, but we can approximate throughput as follows.

We'll call each single-byte read (as in page 33 of the lecture slides) an I2C read *transaction*. Each transaction involves (approximately) the following: a start signal, a slave address + write bit + acknowledgement (9 bits), a register address + acknowledgement (9 bits), a repeated start signal, a slave address + read bit + acknowledgement (9 bits), a data byte + acknowledgement (9 bits), and a stop or repeated start bit, for a total of about 40 bits on the wire for each byte read from the slave. That's about 40 bits with a duration of $\frac{1}{400000}$ s each, for a total of 100 μs per transaction. Since we are polling four devices in sequence, each device will get to send one byte every four transactions (400 μs) for a data rate of about 20 kb/s.

(c) Repeat parts (a) and (b) for the SPI equivalent of the same setup.

Solution: Now there are 7 wires:



Again, the exact timing details (e.g., how long a start condition must be held, required idle time on the bus) will depend on the timing information given in the datasheet, but we can approximate throughput as follows.

An SPI read transaction involves (approximately) the following: a register address + read bit is sent to the slave (8 bits), and the slave returns a data byte (8 bits), for a total of about 16 bits on the wire for each byte read from the slave. That's about 16 bits with a duration of $\frac{1}{400000}$ s each, for a total of 40 μs per transaction. Since we are polling four devices in sequence, each device will get to send one byte every four transactions (160 μs) for a data rate of about 50 kb/s.

(Note that in practice, an SPI bus can support a much higher clock rate than an I2C bus, which increases the data rate even more.)