Rs9194 EL648HW4 - Homework 4

Real Embedded system (New York University)

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 first line of code:
while(!(UCSR0A & 0x20));
This line continuously checks bit 5 (UDRE0) of the UCSR0A register. Bit 5, the “USART Data Register Empty” flag, indicates whether the transmit buffer is empty and ready to accept new data. The loop runs until this bit is set to 1, signaling that the buffer is ready for the next data transmission.

The second line fo code:
UDR0 = x;
This line assigns the value of x (a uint8_t) to the UDR0 register, which represents the transmit buffer. Once the value is placed in this buffer, it will be transmitted over the serial interface via the UART peripheral.

(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 when the code runs, the while condition will evaluate to false immediately, consuming only a few processor cycles (1–3 cycles) to check the flag and write the byte to the buffer.
However, if the transmit buffer is full, the processor will repeatedly evaluate the while loop condition until the buffer becomes empty. At a baud rate of 57,600, it takes approximately 138.9 µs for the buffer to become empty. During this time, the processor executes the while loop, consuming roughly:
Processor cycles = ime waiting / time per cycle = 138.9 / 1/ 8,000, 000
Thus, the code snippet consumes about 1,112 processor cycles in the worst case.

(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: If readByte() is called when there is no incoming byte over the serial interface, the function will enter an infinite loop.

(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:

Yes, a call to readByte() is blocking because the calling program halts execution and waits until a byte is received. This can be problematic because:
1. If no byte arrives, the program will remain stuck indefinitely.
2. The program wastes CPU cycles waiting, preventing it from performing other tasks concurrently.

Here's a non-blocking implementation of readByte():

```
uint8_t readByte(uint8_t *status) {
    // Check if a byte is ready in the receive buffer
    if (!(UCSR0A & 0x80)) {
        *status = 0;  // No byte received
        return 0;     // Return a default value
    }
    // If a byte is available, return it
    *status = 1;
    return UDR0;
}
```

- The function immediately checks if the RXC0 bit in UCSR0A is set.
- If not, it sets *status to 0 and returns, signaling no data is ready.
- If a byte is ready, it retrieves the byte from UDR0, sets *status to 1, and returns the received byte.

The calling program can handle the status to decide whether to retry receiving data or perform other tasks, avoiding indefinite blocking or wasted CPU cycles.

(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:

To approximate 57600 baud:
UBRR = 8 * 10^6 / 16 * 57600 - 1 = 7.68
Bach = 8 * 10^6 / 16(8 + 1) = 55555

The calculated UBRR value is approximately 7.68. Since only integer values can be used, it is rounded to the nearest whole number, which is 8.
Using this
UBRR value, the resulting baud rate is approximately 55555 bits per second.
The percent error, indicating the difference between the desired and achieved baud rates, is calculated as approximately -3.55%, showing the achieved rate is slightly slower than the desired one.

(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:

UBRR = 2 * 10^6 / 16 * 57600 - 1 = 1.17
Bach = 62500

The percent error from the desired rate of 57600 bps is approximately 8.5%, indicating that the achieved rate is faster than intended.

Comparing this to the baud rate of the other device (55555 bps), the percent error is approximately 12.5%, showing a significant mismatch between the two communication partners.

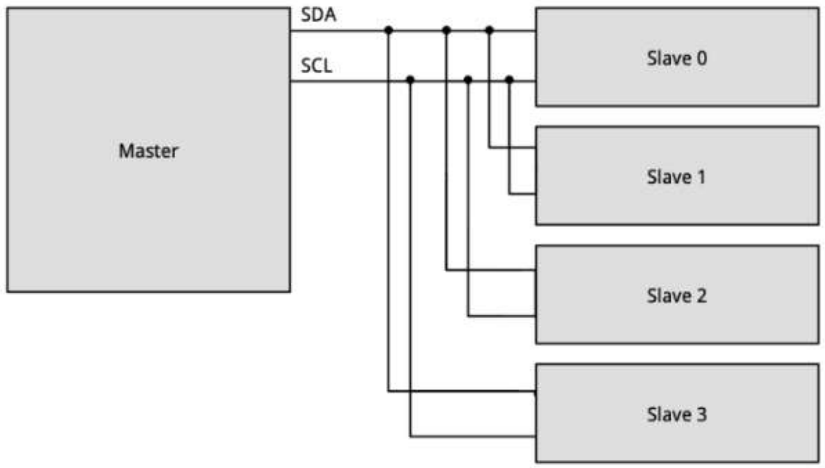
The total error exceeds the maximum allowable error specified in Table 75 of the ATmega128 reference manual for any data/parity/stop bit configuration.

Conclusion: The total error is too high to meet the recommended specifications.

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?

Total number of wires: 2



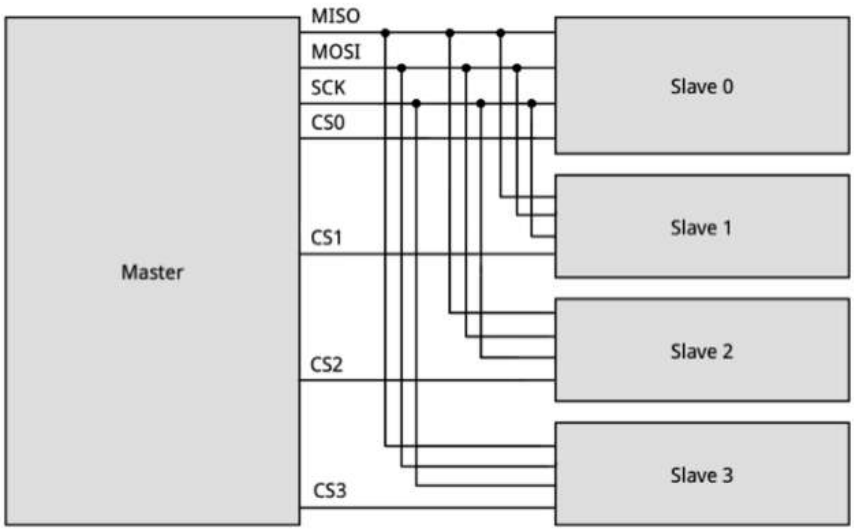
   (b) Suppose the microcontroller reads one data byte from each of the four devices sequentially. (This is similar to the single-byte read shown on page 33 of the lecture slides, but instead of a stop at the end, there is a repeated start condition followed by a different slave address.) What is the data 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 specific timing requirements (e.g., how long a start condition must be held or the necessary idle time on the bus) depend on the details provided in the datasheet. However, we can estimate throughput as follows:
Each single-byte read operation, as outlined in the lecture slides (page 33), constitutes an I2C read transaction. This transaction typically includes the following: a start condition, a slave address with a write bit and acknowledgment (9 bits), a register address with acknowledgment (9 bits), a repeated start condition, a slave address with a read bit and acknowledgment (9 bits), a data byte with acknowledgment (9 bits), and a stop or repeated start condition. Altogether, this amounts to approximately 40 bits transmitted per byte read.
At a clock rate of 400 kHz, each bit takes 1/400,000 seconds, or 2.5 µs. Therefore, a single transaction takes approximately:
100 µs
If four devices are polled sequentially, each device can transmit one byte every four transactions, which is:
400 µs
Data rate: 8 / 400 = 20 kb/s

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

Solution:

Total number of wires: 7



The specific timing details (such as the duration for holding a start condition or the idle time required on the bus) depend on the datasheet. However, we can estimate throughput as follows:

An SPI read transaction generally includes sending a register address with a read bit to the slave (8 bits), followed by the slave responding with a data byte (8 bits). This totals approximately 16 bits transmitted for each byte read.
At a clock rate of 400 kHz, each bit takes 2.5 µs.
One transaction requires 16 * 2.5 = 40 µs
When polling four devices sequentially, each device sends one byte every four transactions: 160 µs

Therefore the data rate is around 50 kb/s