CAB202 Topic 8 - Serial Communication

Authors:

- Luis Mejias, Lawrence Buckingham, QUT (2020)

Contents

Roadmap

Previously:

7. AVR ATMega328P Introduction to Microcontrollers; Digital Input/Output.

This week:

8. **Serial Communication - communicating with another computer/microcontroller**

Still to come:

9. Debouncing, Timers and Interrupts. Asynchronous programming.
10. Analogue to Digital Conversion; Pulse Width Modulation (PWM); Assignment 2 Q&A.
11. LCD Display, sending digital signals to a device.

References

Recommended reading:

- Blackboard→Learning Resources→Microcontrollers→atmega328P datasheet.pdf.

Serial Communication Introduction

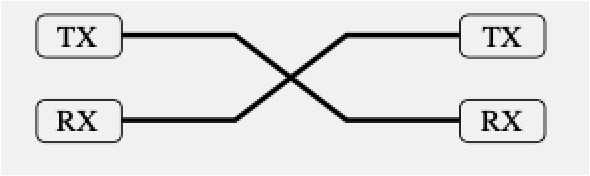
- Serial Communication is a way for two or more electronic devices to exchange data.
- Electrical connections between devices are made by connecting a pin on one device to a pin on the other device.
  - If the components are mounted on the same PCB, the pins are connected via a conductive path.
  - If the pins are on physically separate devices, wire is typically used.
- To transmit data, a voltage is applied to the connection(s) at one end and detected at the other end.
- Large payloads are sent as a sequence of bytes.
- A byte is transmitted as a packet or frame, which may include:
  - A bit pattern indicating the start of the byte.
  - The content of the byte, as 8 (or in some applications, 7) bits.
  - A bit pattern indicating the end of the byte.
  - The packet may include information to help detect transmission errors.

- The packet is sent one bit at a time.
  - The sender must extract each bit from the packet to be transmitted, and send the bit.
  - The bit is sent by holding the voltage steady (high or low) on the line for an agreed period of time.
  - During the agreed period, the receiver reads the state of the line, and interprets the voltage as a 0 or 1.
  - The receiver then builds up a model of the packet which matches the packet being sent.
- As well as a data link, communicating devices need to agree on timing to allow the receiver to decide when each bit starts and stops.
  - Without some kind of shared time-frame, there is no way to tell when one bit begins and the next ends.
  - Sometimes there will be transitions when a 1 is followed by a 0, or a 0 is followed by a 1. This is ambiguous. For example, consider a slow signal (`1010`) compared with a fast signal (`11001100`). The content is very different, but they both look the same on the data line.
  - A range of strategies are adopted (each protocol has its own way) to enable the receiver and transmitter to establish a common time frame.

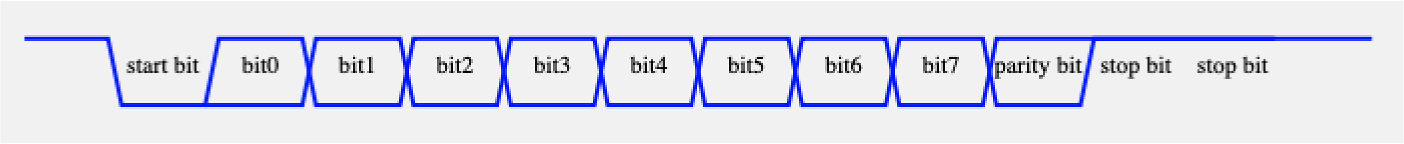
UART (Universal Asynchronous Receiver-Transmitter)

Introduction

- Reference: Data sheet, Chapter 20 (Pages 179–204).
- The `U`niversal `A`synchronous `R`eceiver-`T`ransmitter (`UART`) is a dedicated circuit integrated into the microcontroller.
  - `UART` is can be used for direct communication with another device (e.g. another microncontroller).
  - Bidirectional data transfer is possible because each device has a `TX` (transmit) and `RX` (receive) pin.
  - Connect `TX` on one device to `RX` on the other, and vice-versa.



- A `UART` data frame looks like this (the lines cross over to indicate possible transitions between low and high states):



- To transmit a byte using UART:
  - Bits are transmitted holding the line high (1) or low (0) for a fixed duration of δt seconds.
  - The line is initially held high, indicating that it is idle.
  - At the start of the transmission, the line transitions from high to low, where it stays for δt seconds. This is called the *start bit*.
  - Then each bit is signalled in turn, usually least significant bit first.
  - A parity bit may be sent after the data bits. If used, the sum of the bits in the frame plus the parity bit should always be even. Otherwise, the data is corrupt.
  - Finally, the line reverts to high for at least δt seconds. This signals the end of the byte. It is called the *stop bit*.
- `UART` does not use a clock signal. Instead, both devices must be set to use a common timeframe.
  - This is done by deciding on a fixed speed which will be used for transmission.
  - Both devices must be set to use this speed setting.

- The transmission speed is called the *baud rate*. It measures the number of bits per second that will be transmitted.
- Normal baud rates are: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, or 115200.
- If one device is set to a different baud rate than the other the signal will not be received intact.

`UART` register usage

- `UART` register usage is as follows.
- `UDR0` – I/O Data Register (8 bits)
  - Transmit data buffer and Receive data buffer map to a common address in RAM.
  - Reading the register returns the contents of the Receive Data buffer.
  - Writing to this address places data in the Transmit Data buffer.
    - The `UDRE0` bit must be set to enable data transmission (see `UCSR0A` below).
- `UCSR0A` – USART Control and Status Register 0 A

Bit	7	6	5	4	3	2	1	0	
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn	UCSRnA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

Pin	Name	Interpretation
7	`RXC0`	USART Receive Complete flag – can generate an Receive Complete interrupt (see `RXCIE0` bit).
6	`TCX0`	USART Transmit Complete flag – can generate Transmit Complete interrupt.
5	`UDRE0`	USART Data Register Empty – transmit buffer is ready to receive new data. Can generate a Data Register Empty interrupt (see `UDRIE0` bit).
4	`FE0`	Frame Error – always clear this bit when writing to `UCSR0A` (this register).
3	`DOR0`	Data Overrun – always clear this bit when writing to `UCSR0A` (this register).

2	**`UPE0`**	USART Parity Error – always clear this bit when writing to **`UCSR0A`** (this register).
1	**`U2X0`**	Double transmission speed: 0 → Normal speed; 1 → Double speed.
0	**`MPCM0`**	Multi-processor Communication Mode.

- **`UCSR0B`** – USART Control and Status Register 0 B

Bit	7	6	5	4	3	2	1	0	
	RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXB8n	TXB8n	UCSRnB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Pin	Name	Interpretation
7	**`RXCIE0`**	RX Complete Interrupt Enable.
6	**`TXCIE0`**	TX Complete Interrupt Enable.
5	**`UDRIE0`**	USART Data Register Empty Interrupt Enable.
4	**`RXEN0`**	Receiver enable.
3	**`TXEN0`**	Transmitter enable.
2	**`UCSZ02`**	Character Size bit 2 (combined with **`UCSZ01`** and **`UCSZ00`**).
1	**`RXB80`**	Receive data bit 8 – the ninth bit of a 9-bit character received (when operating with 9-bit characters).
0	**`TXB80`**	Transmit data bit 8 – the ninth bit of a 9-bit character (when operating with 9-bit characters).

- **`UCSR0C`** – control and Status Register 0 C

Bit	7	6	5	4	3	2	1	0	
	UMSELn1	UMSELn0	UPMn1	UPMn0	USBSn	UCSZn1	UCSZn0	UCPOLn	UCSRnC
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	1	1	0	

Pin	Name	Interpretation

<table>
<tr><td>7</td><td><code>UMSEL01</code></td><td>USART mode select bit 1.</td></tr>
<tr><td>6</td><td><code>UMSEL00</code></td><td>USART mode select bit 0: combine with <code>UMSEL01</code>.
<table>
<tr><th><code>UMSEL01</code></th><th><code>UMSEL00</code></th><th>Mode</th></tr>
<tr><td><code>0</code></td><td><code>0</code></td><td>Asynchronous</td></tr>
<tr><td><code>0</code></td><td><code>1</code></td><td>Synchronous</td></tr>
<tr><td><code>1</code></td><td><code>1</code></td><td>Master SPI</td></tr>
</table></td></tr>
<tr><td>5</td><td><code>UPM01</code></td><td>Parity mode, bit 1.</td></tr>
<tr><td>4</td><td><code>UPM00</code></td><td>Parity mode, bit 0: combine with <code>UPM01</code>.
<table>
<tr><th><code>UPM01</code></th><th><code>UPM00</code></th><th>Mode</th></tr>
<tr><td><code>0</code></td><td><code>0</code></td><td>Disabled</td></tr>
<tr><td><code>1</code></td><td><code>0</code></td><td>Even parity</td></tr>
<tr><td><code>1</code></td><td><code>1</code></td><td>Odd parity</td></tr>
</table></td></tr>
<tr><td>3</td><td><code>USBS0</code></td><td>Stop bits: 0 → 1 stop bit; 1 → 2 stop bits.</td></tr>
<tr><td>2</td><td><code>UCSZ01</code></td><td>Character size bit 1.</td></tr>
<tr><td>1</td><td><code>UCSZ00</code></td><td>Character size bit 0. Combine with <code>UCSZ01</code> and <code>UCSZ02</code>.
<table>
<tr><th><code>UCSZ02</code></th><th><code>UCSZ01</code></th><th><code>UCSZ00</code></th><th>Character size</th></tr>
<tr><td><code>0</code></td><td><code>0</code></td><td><code>0</code></td><td>5 bits</td></tr>
<tr><td><code>0</code></td><td><code>0</code></td><td><code>1</code></td><td>6 bits</td></tr>
<tr><td><code>0</code></td><td><code>1</code></td><td><code>0</code></td><td>7 bits</td></tr>
<tr><td><code>0</code></td><td><code>1</code></td><td><code>1</code></td><td>8 bits</td></tr>
<tr><td><code>1</code></td><td><code>1</code></td><td><code>1</code></td><td>9 bits</td></tr>
</table></td></tr>
<tr><td>0</td><td><code>UCPOL0</code></td><td>Clock polarity.
<table>
<tr><th><code>UCPOL0</code></th><th>Output to TxD1 pin</th><th>Input sampled on RxD1 pin</th></tr>
</table></td></tr>
</table>

		0	Rising edge	Falling edge
		1	Falling edge	Rising edge

- `UBRR0L and UBRR0H` – USART Baud Rate Registers.

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	UBRRn[11:8]				UBRRnH
	UBRRn[7:0]								UBRRnL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

  - A two-byte (16 bit) register which is used to define the baud rate.
  - 12 bits are used.
  - Optimal values of `UBRR0` for a CPU running at 16MHz are listed in the table 20-1on page 182 of the datasheet. Fairly accurate approximations are obtained with the formulae:

Mode	Equation
Async normal	`UBRR0 = (F_CPU/16/BAUD) - 1`
Async double speed	`UBRR0 = (F_CPU/8/BAUD) - 1`
Sync master mode	`UBRR0 = (F_CPU/2/BAUD)` - 1

Here, F_CPU is 16MHz (16000000UL) and BAUD any of the following 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, or 115200.

`UART` hardware programming

- Once the **`UART`** control registers have been configured, the programming model is extremely simple.
- For general purpose usage, an Init, Transmit and Receive functions are anough for most applications.
- The sample code below is an example how to implement each of these functions.
- To initialise **`UART`**:

```
void uart_init(unsigned int ubrr){

    UBRR0H = (unsigned char)(ubrr>>8);
    UBRR0L = (unsigned char)(ubrr);
        UCSR0B = (1 << RXEN0) | (1 << TXEN0);
        UCSR0C = (3 << UCSZ00);

}
```

What this does:

1. Set **`UBRR0`** using the formula above. ubrr is the argument that result from the formula (table above).
2. Enable receive, transmit, and the receive-complete interrupt.
   - A receive-complete interrupt handler is also implemented (not shown here).
3. Set character size to 8 bits.

- To send characters to a connected device:

```
void uart_putchar(unsigned char data){


   while (!( UCSR0A & (1<<UDRE0))); /* Wait for empty transmit buffer*/

          UDR0 = data;            /* Put data into buffer, sends the data */


}
```

What this does:

1. Waits until there is room in the transmit buffer for another character.
2. Copy characters to the I/O **UDR0** data register

- To receive one character from a connected device:

```
unsigned char uart_getchar(void){

        while ( !(UCSR0A & (1<<RXC0)));

 return UDR0;

}
```

What this does:

1. Waits for character to become available in receive buffer.
2. Returns the data regiter

Case Study – Bidirectional communication between a microcontroller and serial console

- A sample program, **`uart_example1.c`**, is listed below.

```
/* File: uart_example1.c
 * Description: C program for the ATMEL AVR microcontroller (ATmega328 chip)
 * Send characters via serial and receives characters from serial console
 *
 * Includes (pretty much compulsory for using the Teensy this semester)
 *      - avr/io.h: port and pin definitions (i.e. DDRB, PORTB, PB1, etc)
 *
 */

#define F_CPU 16000000UL
// AVR header file for all registers/pins
#include <avr/io.h>
```

```
*  Setting data directions in a data direction register (DDR)
 *
 *
 *  Setting, clearing, and reading bits in registers.
 *      reg is the name of a register; pin is the index (0..7)
 *  of the bit to set, clear or read.
 *  (WRITE_BIT is a combination of CLEAR_BIT & SET_BIT)
 */

#define SET_BIT(reg, pin)                        (reg) |= (1 << (pin))
#define CLEAR_BIT(reg, pin)                      (reg) &= ~(1 << (pin))
#define WRITE_BIT(reg, pin, value)       (reg) = (((reg) & ~(1 << (pin))) | ((value
#define BIT_VALUE(reg, pin)                      (((reg) >> (pin)) & 1)
#define BIT_IS_SET(reg, pin)             (BIT_VALUE((reg),(pin))==1)

//Functions declaration
void setup(void);
void process(void);
void uart_init(unsigned int ubrr);
unsigned char uart_getchar(void);
void uart_putchar(unsigned char data);


//UART definitions
//define baud rate
#define BAUD 9600
#define MYUBRR F_CPU/16/BAUD-1

//receiving buffer
unsigned char rx_buf;

void setup(void) {

    // initialise uart
    uart_init(MYUBRR);

    // Enable B5 as output, led on B5
        SET_BIT(DDRB, 5);
}


void process(void) {

   //define a character to sent
```

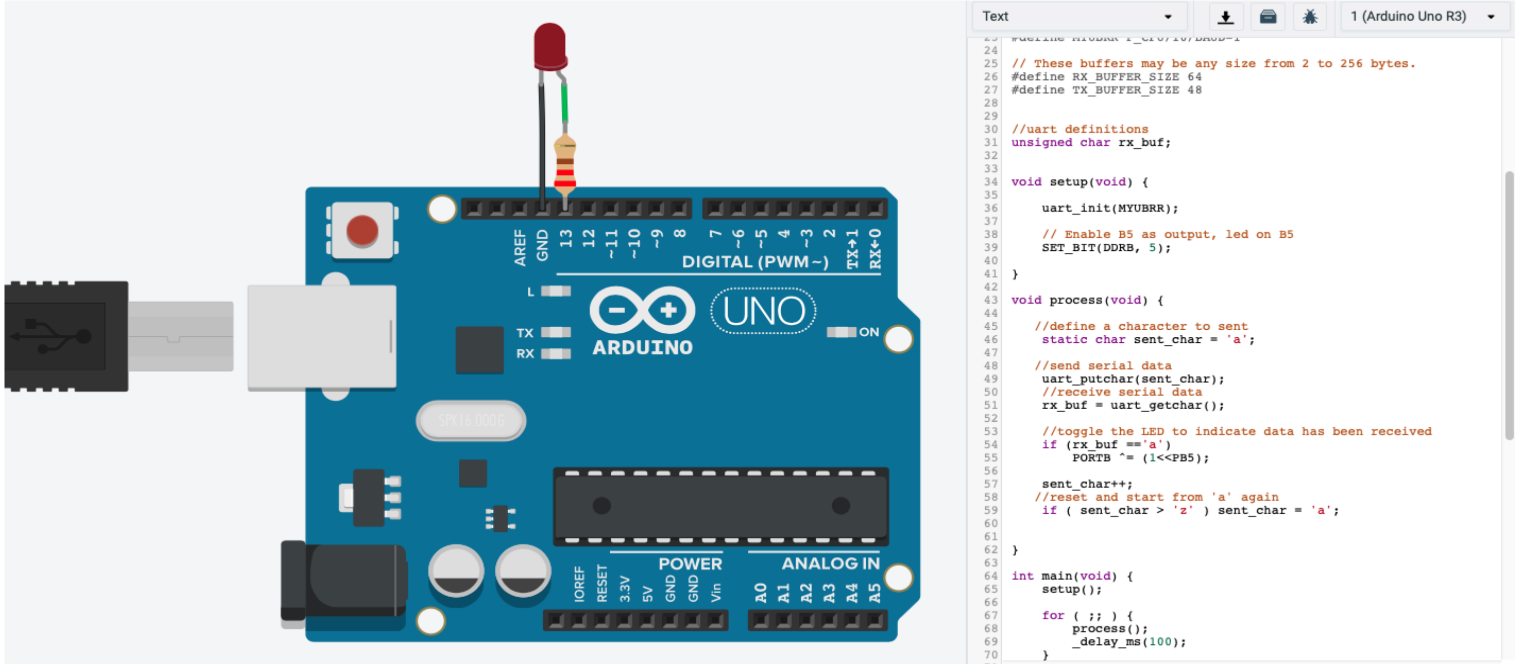
```
    static char sent_char = 'a';
   //send serial data
        uart_putchar(sent_char);
    //receive serial data
    rx_buf = uart_getchar();

        //toggle the LED to indicate data has been received
        if (rx_buf =='a')
        PORTB ^= (1<<PB5);

    sent_char++;
   //reset and start from 'a' again
        if ( sent_char > 'z' ) sent_char = 'a';


}

int main(void) {

    setup();
      for ( ;; ) {
                process();
                _delay_ms(100);
        }
}

/*  ****** serial definitions ************ */
// Initialize the UART
void uart_init(unsigned int ubrr){

    UBRR0H = (unsigned char)(ubrr>>8);
    UBRR0L = (unsigned char)(ubrr);
    UCSR0B = (1 << RXEN0) | (1 << TXEN0);
    UCSR0C =(3 << UCSZ00);



    }

//transmit data
void uart_putchar(unsigned char data){

   while (!( UCSR0A & (1<<UDRE0))); /* Wait for empty transmit buffer*/
```

```
    UDR0 = data;             /* Put data into buffer, sends the data */


}

//receive data
unsigned char uart_getchar(void){

  /* Wait for data to be received */
    while ( !(UCSR0A & (1<<RXC0)) );


  return UDR0;

}
```

- In this program, a microcontroller devices talk to a serial console over the **`UART`** connection.
    - In **`setup`**:
        - **`UART`** is initialised, running at 9600 bits per second.
    - Each time **`process`** is called:
        - An incrementing character is sent via **`uart_putchar`**.
        - Characters are received via **`uart_getchar`**.
        - The received character is used to turn an LED on.
        - Note this code is blocking in the sense that process will stop at get_char() until a character is received.

[TinkerCad](https://www.tinkercad.com/things/5MhUMk8uAzR)version of this program:

https://www.tinkercad.com/things/5MhUMk8uAzR



Case Study – Bidirectional communication between two Microcontrollers

A sample program, **`uart_example2a.c`** and **`uart_example2b.c`**, are listed below.

```
/* File: uart_example2a.c
 * Description: C program for the ATMEL AVR microcontroller (ATmega328 chip)
 * Communication between two microcontrollers
 * Send characters via serial and receives characters from serial
 *
 * Includes (pretty much compulsory for using the Teensy this semester)
 *      - avr/io.h: port and pin definitions (i.e. DDRB, PORTB, PB1, etc)
 *
 */
```

```
#define F_CPU 16000000UL
// AVR header file for all registers/pins

#include <avr/io.h>

/*  useful macros for Setting data directions in a data direction register (DDR)
 *
 *
 *  Setting, clearing, and reading bits in registers.
 *      reg is the name of a register; pin is the index (0..7)
 *  of the bit to set, clear or read.
 *  (WRITE_BIT is a combination of CLEAR_BIT & SET_BIT)
 */

#define SET_BIT(reg, pin)                       (reg) |= (1 << (pin))
#define CLEAR_BIT(reg, pin)                     (reg) &= ~(1 << (pin))
#define WRITE_BIT(reg, pin, value)      (reg) = (((reg) & ~(1 << (pin))) | ((value) << (pi
#define BIT_VALUE(reg, pin)                     (((reg) >> (pin)) & 1)
#define BIT_IS_SET(reg, pin)            (BIT_VALUE((reg),(pin))==1)


//Functions declaration
void setup(void);
void process(void);
void uart_init(unsigned int ubrr);
unsigned char uart_getchar(void);
void uart_putchar(unsigned char data);


//UART definitions
//define baud rate
#define BAUD 9600
#define MYUBRR F_CPU/16/BAUD-1


void setup(void) {

  // initialise uart
    uart_init(MYUBRR);

  // Enable B5 as output, led on B5
        SET_BIT(DDRB, 5);
    SET_BIT(DDRB, 4);
```

```
   // Enable D6 and D7 as inputs
    CLEAR_BIT(DDRD,6);
    CLEAR_BIT(DDRD,7);
}


void process(void) {

   //receiving buffer
        char rx_buf;

    //define a character to sent
    static char sent_char_a = 'a';
    static char sent_char_b = 'b';


  //send character for heartbeat
    uart_putchar('\n');

  //detect pressed switch on D7
  if (BIT_IS_SET(PIND,7)){

   //send serial data
        uart_putchar(sent_char_a);

  }

    //detect presed switch on D6
  if (BIT_IS_SET(PIND,6)){

   //send serial data
        uart_putchar(sent_char_b);

  }


    //receive serial data
    rx_buf = uart_getchar();

        //toggle the LED to indicate data has been received
  if (rx_buf =='c'){SET_BIT(PORTB,5); CLEAR_BIT(PORTB,4);}
  else if (rx_buf =='d'){SET_BIT(PORTB,4); CLEAR_BIT(PORTB,5);}
```

```
}

int main(void) {

    setup();
      for ( ;; ) {
                process();
                _delay_ms(100);
        }
}

/*  ****** serial definitions ************ */

// Initialize the UART
void uart_init(unsigned int ubrr){

    UBRR0H = (unsigned char)(ubrr>>8);
    UBRR0L = (unsigned char)(ubrr);
        UCSR0B = (1 << RXEN0) | (1 << TXEN0) | (1 << RXCIE0);
        UCSR0C = (3 << UCSZ00);



}

//transmit data
void uart_putchar(unsigned char data){

    while (!( UCSR0A & (1<<UDRE0))); /* Wait for empty transmit buffer*/

        UDR0 = data;            /* Put data into buffer, sends the data */


}

//receive data
unsigned char uart_getchar(void){


   /* Wait for data to be received */
    while ( !(UCSR0A & (1<<RXC0)) );

    return UDR0;
```

```
}
```

```
/* File: uart_example2b.c
 * Description: C program for the ATMEL AVR microcontroller (ATmega328 chip)
 * Communication between two microcontrollers
 * Send characters via serial and receives characters from serial
 *
 * Includes (pretty much compulsory for using the Teensy this semester)
 *      - avr/io.h: port and pin definitions (i.e. DDRB, PORTB, PB1, etc)
 *
 */

#define F_CPU 16000000UL
// AVR header file for all registers/pins

#include <avr/io.h>

/*  Setting data directions in a data direction register (DDR)
 *
 *
 *  Setting, clearing, and reading bits in registers.
 *      reg is the name of a register; pin is the index (0..7)
 *  of the bit to set, clear or read.
 *  (WRITE_BIT is a combination of CLEAR_BIT & SET_BIT)
 */

#define SET_BIT(reg, pin)                       (reg) |= (1 << (pin))
#define CLEAR_BIT(reg, pin)                     (reg) &= ~(1 << (pin))
#define WRITE_BIT(reg, pin, value)      (reg) = (((reg) & ~(1 << (pin))) | ((value) << (pi
#define BIT_VALUE(reg, pin)                     (((reg) >> (pin)) & 1)
#define BIT_IS_SET(reg, pin)            (BIT_VALUE((reg),(pin))==1)


//Functions declaration
void setup(void);
void process(void);
void uart_init(unsigned int ubrr);
unsigned char uart_getchar(void);
void uart_putchar(unsigned char data);
```

```
//UART definitions
//define baud rate
#define BAUD 9600
#define MYUBRR F_CPU/16/BAUD-1


void setup(void) {

    // initialise uart
    uart_init(MYUBRR);

     // Enable B5 as output, led on B5
        SET_BIT(DDRB, 5);
    SET_BIT(DDRB, 4);

   // Enable D6 and D7 as inputs
    CLEAR_BIT(DDRD,6);
    CLEAR_BIT(DDRD,7);

}


void process(void) {

    //receiving buffer
        char rx_buf;

   //define a character to sent
    static char sent_char_c = 'c';
    static char sent_char_d = 'd';

    //send character for heartbeat
    uart_putchar('\n');


  //detect pressed switch on D7
  if (BIT_IS_SET(PIND,7)){

   //send serial data
        uart_putchar(sent_char_c);

  }

    //detect presed switch on D6
```

```
  if (BIT_IS_SET(PIND,6)){

   //send serial data
        uart_putchar(sent_char_d);

  }


    //receive serial data
    rx_buf = uart_getchar();

        //toggle the LED to indicate data has been received
  if (rx_buf =='a'){SET_BIT(PORTB,5); CLEAR_BIT(PORTB,4);}
  else if (rx_buf =='b'){SET_BIT(PORTB,4); CLEAR_BIT(PORTB,5);}


}

int main(void) {

    setup();
      for ( ;; ) {
                process();
                _delay_ms(100);
        }
}

/*  ****** serial definitions ************ */
// Initialize the UART
void uart_init(unsigned int ubrr){

   UBRR0H = (unsigned char)(ubrr>>8);
   UBRR0L = (unsigned char)(ubrr);
        UCSR0B = (1 << RXEN0) | (1 << TXEN0) | (1 << RXCIE0);
        UCSR0C = (3 << UCSZ00);



}

//transmit data
void uart_putchar(unsigned char data){

    while (!( UCSR0A & (1<<UDRE0))); /* Wait for empty transmit buffer*/
```

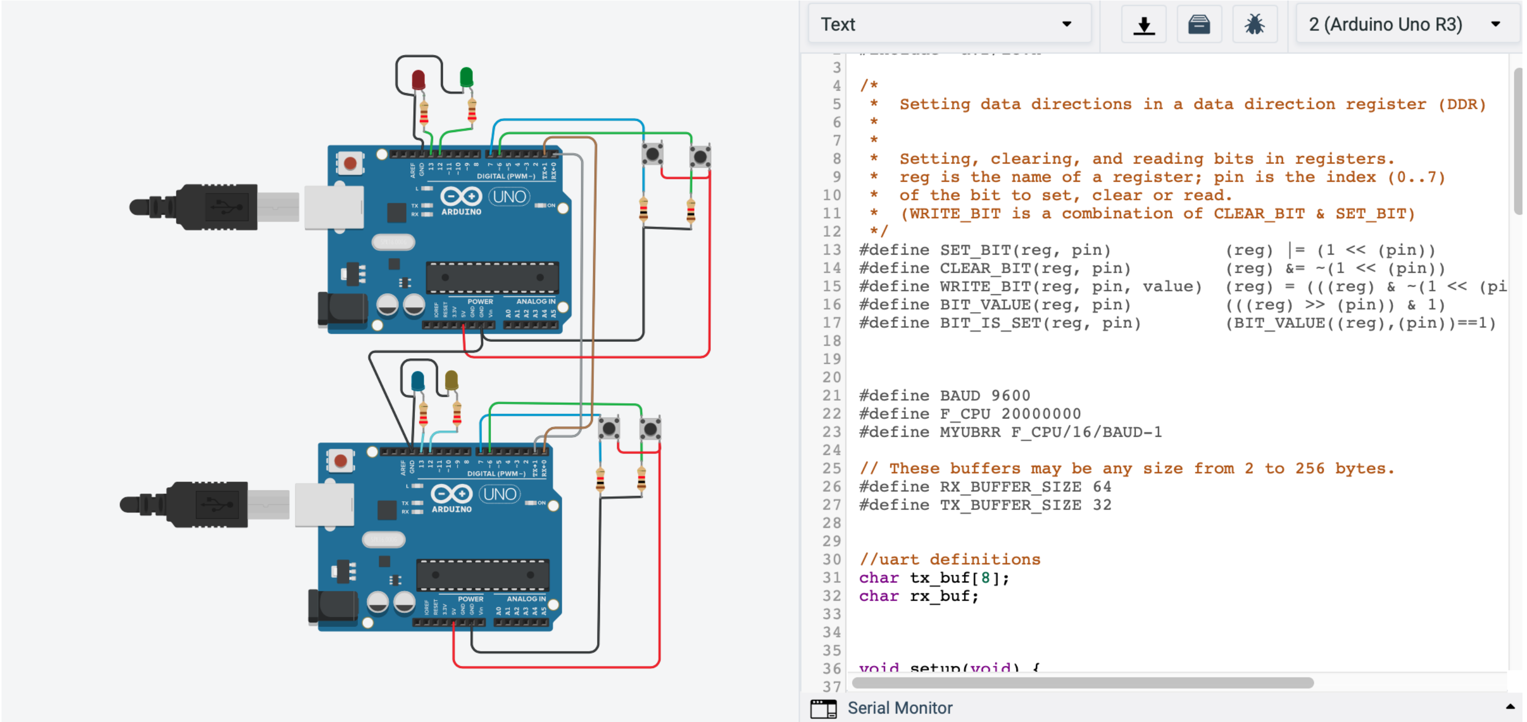
```
    UDR0 = data;            /* Put data into buffer, sends the data */

}

//receive data
unsigned char uart_getchar(void){

    /* Wait for data to be received */
     while ( !(UCSR0A & (1<<RXC0)) );

   return UDR0;

}
```

What this does:

- Both programs are very similar in the way they perform the task. Differences are mainly in the characters used to trigger events on each microcontroller.
- In **Setup**
  - Pins B5 and B4 are set as outputs
  - Pins D6 and D7 are set as inputs.
- In **process**
  - Pins associated with each switch are evaluated, if switch is pressed then a character is sent via uart.
  - uart is read, and depending on the character one of the LEDs is turn on and the other off.

[TinkerCad](https://www.tinkercad.com/things/3QF0CjuryIr)version of this program:

https://www.tinkercad.com/things/3QF0CjuryIr



- ***Note in serial (uart) communications pins you need to connect are `TX` and `RX` `( and GND).`***
    - Please **`do not`** connect **`any other pin between microcontrollers.`**

Additional exercices:

- Design a system based on microcontrollers (2x) that exchange data via uart.
    - The first microntroller send the string "Hello from Microcontroller 1", then the second microcontroller receive the string and add "From Microcontroller 2:" at the beginning of the received string and send it back to microcontroller 1. **Hint**: Example 2 is a good starting point for this exercices. The receiver function will need to be modified to receive strings.

Summary

- UART is a critical functionality in emebedded system. Often, to debug program, configure inner parameters in the program, send state of the program, etc.
- Timers and interrupts are the preferred way to impletement uart routines, once covered in future lectures we will modify uart routines to use interrupts.
- Serial communications also refer to other protocols such as SPI, I2C and USB.

The End
