CAB202 Topic 10 – Serial Communication

Authors:

- Lawrence Buckingham, QUT (2017)

Contents

Roadmap

Previously:

7. Teensy – Introduction to Microcontrollers; Digital Input/Output; Bitwise operations.
8. LCD Display – sending digital signals to a device; directly controlling the LCD display.
9. Timers and Interrupts – asynchronous programming.

   Assignment 2 task was published.

This week:

10. Serial Communication – communicating with another computer.

Still to come:

11. Analogue to Digital Conversion; Pulse Width Modulation; Assignment 2 Q&A.
12. Bread-boarding a Teensy – live demonstration.
13. Last minute Assignment 2 Q+A.

References

Recommended reading:

- Blackboard→Learning Resources→Microcontrollers→atmega32u4 datasheet.pdf.
- PJRC UART Library: https://www.pjrc.com/teensy/uart.html
- PJRC USB Serial Library: https://www.pjrc.com/teensy/usb_serial.html
- Sparkfun SPI tutorial: https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi

---

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.

LCD: One-way serial communication

- The LCD covered in Lecture 8 is an example of a device which is controlled via serial communication.
- TeensyPewPew has six electrical connections between the ATMega32U4 chip and the LCD.
    - `C7` → LCD backlight.
    - `B4` → LCD Reset pin.
    - `B6` → LCD Serial Data Input pin.
    - `B5` → LCD Serial Data/Command pin.

- `D7` → LCD Chip Select pin.
- `F7` → LCD Serial Clock pin.

- Four of these are used to send data to the LCD, but the serial communication is done with two wires:
  - `F7` carries a clock signal which tells the LCD when to read a bit.
  - `B6` is carries the bits.
  - `D7` *tells the LCD that data is arriving, but here we are interested in the signalling process used to send the data*.
- Transmitting a byte looks like this:

- Data transmission works as follows:
  - The sender provides the clock signal.
  - The clock signal has a period equal to the transmission time for a single bit.
  - To send a bit, the sender:
    1. Clears the clock pin;
    2. Writes the bit value to the data pin;
    3. Sets the clock pin.
  - The receiver monitors the clock signal, and when it detects a transition from 0 to 1 on its clock pin, it reads a bit value from its incoming data pin.
- On the TeensyPewPew, communication to the LCD must be done with hand-written code.
  - See `lcd.c` in the `cab202_teensy` library.
  - This technique is known as *bit banging* or *bit bashing*: https://en.wikipedia.org/wiki/Bit_banging .
  - Bit-bashing can be problematic:
    - While the CPU is tied up sending and receiving data nothing else can be done.
    - CPU load can cause data errors if the receiver code is not able to process incoming bits fast enough.

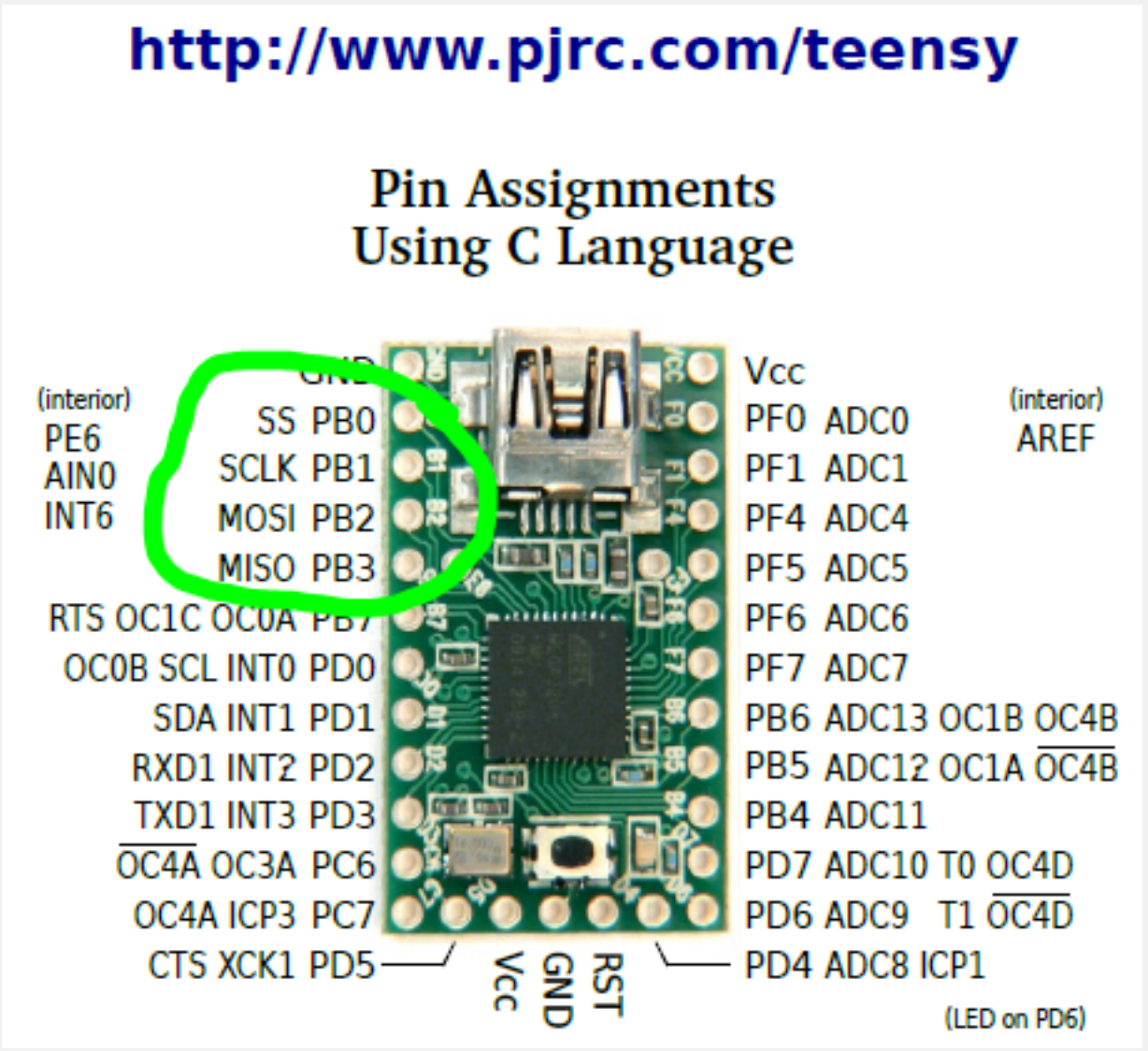
Serial Peripheral Interface (SPI)

Overview

- The ATMega32U4 includes a hardware implementation of the Serial Peripheral Interface (SPI).
- Applications:
  - LCD displays.
    - We cannot use SPI to drive our LCD, because SPI runs on pins B0, B1, B2, and B3, and we have already connected these pins to other digital IO (B0 and B1 are connected to the joystick; B2 and B3 are connected to LEDs).
  - Sensors.
  - External storage devices such as SD cards.
- Under SPI, one device is in charge: the *master*.
  - The master device manages interaction with one or more *slave* devices.
  - Where multiple slaves are controlled, each is allocated a numeric address which is used to ensure

transmissions go to the correct receiver.

- SPI uses 4 pins:



Pin	Function
`SS`	*Slave Select* – tells connected slaves to send or receive data. ◦ Where multiple slaves are connected, each may have a separate dedicated **`SS`** pin. ◦ **`SS`** can be configured as an output when we only intend to be the master. ◦ When we may need to share master role, **`SS`** can be configured as an input. Details may be found on P182.
`SCK`	*Clock* – slave monitors this pin and uses it to determine when to read and write bits.
`MOSI`	*Master Out, Slave In* – serial data from master to slave.
`MISO`	*Master In, Slave Out* – serial data from slave to master.

- **The SPI interface on the ATmega16U4/ATmega32U4 is also used for program memory and EEPROM downloading or uploading. See page 360 for serial programming and verification.**

SPI register usage

- SPI by the following registers.
  - Refer Data sheet, Chapter 17.
  - **`SPCR`** – SPI control register (Page 182).

Pin	Name	Interpretation
7	**`SPIE`**	Interrupt enable: generate SPI interrupt when transfer is complete
6	**`SPE`**	SPI Enable
5	**`DORD`**	Data Order: 1 = LSB first; 0 = MSB first
4	**`MSTR`**	Master/Slave select: 1 = master; 0 = slave
3	**`CPOL`**	Clock polarity: 1 = high when idle; 0 = low when idle
2	**`CPHA`**	Clock phase: 1 = sample on trailing edge; 0 = sample on leading edge
1	**`SPR1`**	Clock rate select bit 1 (see **`SPI2X`** below)
0	**`SPR0`**	Clock rate select bit 0 (see **`SPI2X`** below)

- **SPSR** – SPI status register (Page 183-4).

Pin	Name	Interpretation
7	**SPIF**	Interrupt flag: set when transfer complete; see data sheet for detail
6	**WCOL**	Write Collision flag
0	**SPI2X**	Clock rate select bit 2 (see following table)

- Clock rate: 8 values, encoded by combining **SPI2X**, **SPR1**, **SPR0**. F_{Osc} is the CPU clock frequency.

SPI2X	**SPR1**	**SPR0**	SCK frequency
0	0	0	$F_{Osc}/4$
0	0	1	$F_{Osc}/16$
0	1	0	$F_{Osc}/64$
0	1	1	$F_{Osc}/128$
1	0	0	$F_{Osc}/2$
1	0	1	$F_{Osc}/8$
1	1	0	$F_{Osc}/32$
1	1	1	$F_{Osc}/64$

- **SPDR** – SPI data register.
  - Data register is used to read and write.
  - Writing to the register initiates data transmission.
  - Reading from the register causes data to be read from the shift register receive buffer.

Programming SPI

- The code samples below show how to:
  - Configure SPI in master or slave roles;
  - Transmit a byte;
  - Receive a byte.
  - *Warning* This code cannot be tested easily on a PewPew. It should be considered indicative rather than definitive.
- Initialise master (data sheet, P180):

```
void SPI_MasterInit(void) {
        /* Set MOSI and SCLK output, all others input */
        DDRB |= (1 << PB2) | (1 << PB1);

        /* Enable SPI, Master, set clock rate f_osc/16 */
        SPCR = (1 << SPE) | (1 << MSTR) | (1 << SPR0);
}
```

  - **MOSI** is **B2**, **SCLK** is **B1**, so the data direction register is updated to allow output on those pins.
  - **SS** is on **B0**, so if you don't intend to use slave mode, you could set **PB0** as output as well.
  - Set appropriate bits in SPI control register.
- Transmit byte from master (data sheet, P180):

```
void SPI_MasterTransmit(char cData) {
        /* Start transmission */
        SPDR = cData;

        /* Wait for transmission complete */
        while ( !(SPSR & (1 << SPIF)) ) {}
}
```

  - Transfer is initiated by writing the data to **`SPDR`**.
  - After initiation, wait until the data has been sent.

- Initialise slave (data sheet, P181):

```
void SPI_SlaveInit(void) {
        /* Set MISO output, all others input */
        DDRB |= (1 << PB3);

        /* Enable SPI */
        SPCR = (1 << SPE);
}
```

- Receive byte from master (data sheet, P181):

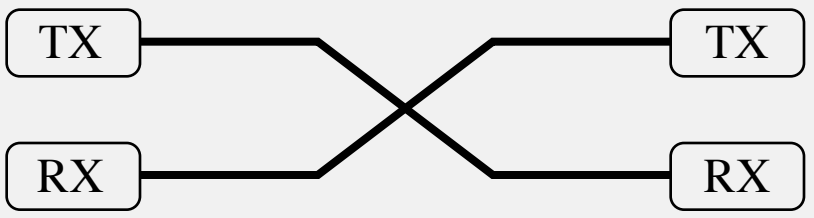
```
char SPI_SlaveReceive(void) {
        /* Wait for reception complete */
        while ( !(SPSR & (1 << SPIF)) ) {}

        /* Return Data Register */
        return SPDR;
}
```

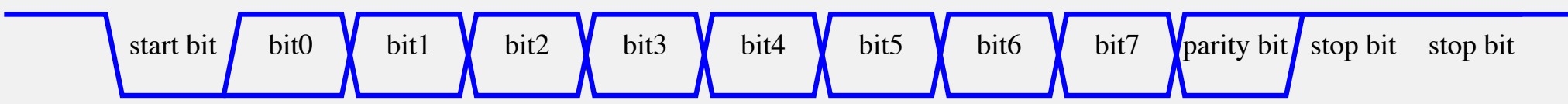
UART (Universal Asynchronous Receiver-Transmitter)

Introduction

- Reference: Data sheet, Chapter 18 (Pages 186–213).
- 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 Teensy).
  - 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.
- **UDR1** – 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 **UDRE1** bit must be set to enable data transmission (see **UCSR1A** below).
- **UCSR1A** – USART Control and Status Register 1 A

Pin	Name	Interpretation
7	**RXC1**	USART Receive Complete flag – can generate an Receive Complete interrupt (see **RXCIE1** bit).
6	**TCX1**	USART Transmit Complete flag – can generate Transmit Complete interrupt.
5	**UDRE1**	USART Data Register Empty – transmit buffer is ready to receive new data. Can generate a Data Register Empty interrupt (see **UDRIE1** bit).
4	**FE1**	Frame Error – always clear this bit when writing to **UCSR1A** (this register).
3	**DOR1**	Data Overrun – always clear this bit when writing to **UCSR1A** (this register).
2	**UPE1**	USART Parity Error – always clear this bit when writing to **UCSR1A** (this register).
1	**U2X1**	Double transmission speed: 0 → Normal speed; 1 → Double speed.
0	**MPCM1**	Multi-processor Communication Mode.

- **UCSR1B** – USART Control and Status Register 1 B

Pin	Name	Interpretation
7	**RXCIE1**	RX Complete Interrupt Enable.
6	**TXCIE1**	TX Complete Interrupt Enable.
5	**UDRIE1**	USART Data Register Empty Interrupt Enable.
4	**RXEN1**	Receiver enable.
3	**TXEN1**	Transmitter enable.
2	**UCSZ12**	Character Size bit 2 (combined with **UCSZ11** and **UCSZ10**).
1	**RXB81**	Receive data bit 8 – the ninth bit of a 9-bit character received (when operating with 9-bit characters).
0	**TXB81**	Transmit data bit 8 – the ninth bit of a 9-bit character (when operating with 9-bit characters).

- **UCSR1C** – control and Status Register 1 C

Pin	Name	Interpretation
7	**UMSEL11**	USART mode select bit 1.
6	**UNSEL10**	USART mode select bit 0: combine with **UMSEL11**. **UMSEL11** / **UNSEL10** / Mode **0** / **0** / Asynchronous **0** / **1** / Synchronous **1** / **1** / Master SPI
5	**UPM11**	Parity mode, bit 1.
4	**UPM10**	Parity mode, bit 0: combine with **UPM11**. **UPM11** / **UPM10** / Mode **0** / **0** / Disabled **1** / **0** / Even parity **1** / **1** / Odd parity
3	**USBS1**	Stop bits: 0 → 1 stop bit; 1 → 2 stop bits.
2	**UCSZ11**	Character size bit 1.
1	**UCSZ10**	Character size bit 0. Combine with **UCSZ11** and **UCSZ12**. **UCSZ12** / **UCSZ11** / **UCSZ10** / Character size **0** / **0** / **0** / 5 bits **0** / **0** / **1** / 6 bits **0** / **1** / **0** / 7 bits **0** / **1** / **1** / 8 bits **1** / **1** / **1** / 9 bits
0	**UCPOL1**	Clock polarity. **UCPOL1** / Output to TxD1 pin / Input sampled on RxD1 pin **0** / Rising edge / Falling edge **1** / Falling edge / Rising edge

- **UCRS1D** – USART Control and Status Register 1 D

Pin	Name	Interpretation
7..2	**-**	Reserved.
1	**CTSEN**	CTS enable. See P209.
0	**RTSEN**	RTS enable. See P209.

- **UBRR1** – USART Baud Rate Registers.
  - A two-byte (16 bit) register which is used to define the baud rate.
  - 12 bits are used.
  - Optimal values of **UBRR1** for a CPU running at 8MHz are listed in the tables on page 212 of the datasheet. Fairly accurate approximations are obtained with the formulae:

Mode	Equation
Async normal	**UBRR1 = (F_CPU/8/BAUD - 1)/2**
Async double speed	**UBRR1 = (F_CPU/4/BAUD - 1)/2**

Sync master mode	**`UBRR1 = F_CPU/2/BAUD`** - 1

`UART` hardware programming

- Once the **`UART`** control registers have been configured, the programming model is extremely simple.
- For general purpose usage, the sample **`UART`** library at https://www.pjrc.com/teensy/uart.html is a useful resource. The code is elegant and economical.
- The sample code below is an excerpt from **`uart.c`** (available at the link provided).
- To initialise **`UART`**:

```
void uart_init(uint32_t baud) {
        cli();
        UBRR1 = (F_CPU / 4 / baud - 1) / 2;
        UCSR1A = (1<<U2X1);
        UCSR1B = (1<<RXEN1) | (1<<TXEN1) | (1<<RXCIE1);
        UCSR1C = (1<<UCSZ11) | (1<<UCSZ10);
        tx_buffer_head = tx_buffer_tail = 0;
        rx_buffer_head = rx_buffer_tail = 0;
        sei();
}
```

  What this does:

  1. Set **`UBRR1`** using the formula to convert from baud rate.
     - You can infer from the formula that double-speed asynchronous mode is in use.
  2. Enable double speed mode.
  3. Enable receive, transmit, and the receive-complete interrupt.
     - A receive-complete interrupt handler is also implemented (not shown here).
     - The ISR stashes multiple incoming bytes in a buffer to make it less likely to lose data.
  4. Set character size to 8 bits.
  5. Initialise transmit and read buffers.
     - These are arrays which hold incoming and outgoing characters.
     - When we want to send a character, it is first dropped int the buffer.
     - Later, an interrupt service routine will move data from the buffer to the UART Data Register.
- To send one character to a connected device:

```
void uart_putchar(uint8_t c) {
        uint8_t i;

        i = tx_buffer_head + 1;
        if ( i >= TX_BUFFER_SIZE ) i = 0;

        while ( tx_buffer_tail == i ); // wait until space in buffer

        //cli();
        tx_buffer[i] = c;
        tx_buffer_head = i;
        UCSR1B = (1 << RXEN1) | (1 << TXEN1) | (1 << RXCIE1) | (1 << UDRIE1);
        //sei();
}
```

  What this does:

  1. Waits until there is room in the transmit buffer for another character.
  2. Inserts the character into the transmit buffer.
  3. Enables the Data Register Empty interrupt.
     - The ISR (in **`uart.c`** but not shown here) will be executed when the transmitter is ready to send a character.

        - As soon as that happens, a character is grabbed from the buffer and sent.
        - Eventually the buffer empties out, and the ISR disables the interrupt to avoid wasting clock cycles.
    4. This approach means we usually do not have to wait for the character to be sent.
        - Our program can move on to do other work.
- To receive one character from a connected device:

```
uint8_t uart_getchar(void) {
        uint8_t c, i;

        while ( rx_buffer_head == rx_buffer_tail ); // wait for character

        i = rx_buffer_tail + 1;
        if ( i >= RX_BUFFER_SIZE ) i = 0;
        c = rx_buffer[i];
        rx_buffer_tail = i;
        return c;
}
```

What this does:

1. Waits until there is a character in the receive buffer.
2. Fetches the next character from the buffer.
3. Returns the character.

- In summary, **`uart.c`**:
    - Implements an asynchronous transmit function – the program can continue working while the data is sent by the **`UART`** circuit.
    - Implements a synchronous receive function – it waits for a character to arrive before returning.
    - If you examine the two interrupt handlers, it also illustrates how the Receive Complete and Data Register Empty interrupts can be used to orchestrate asynchronous send and receive operations.

Case Study – Bidirectional communication between two Teensies

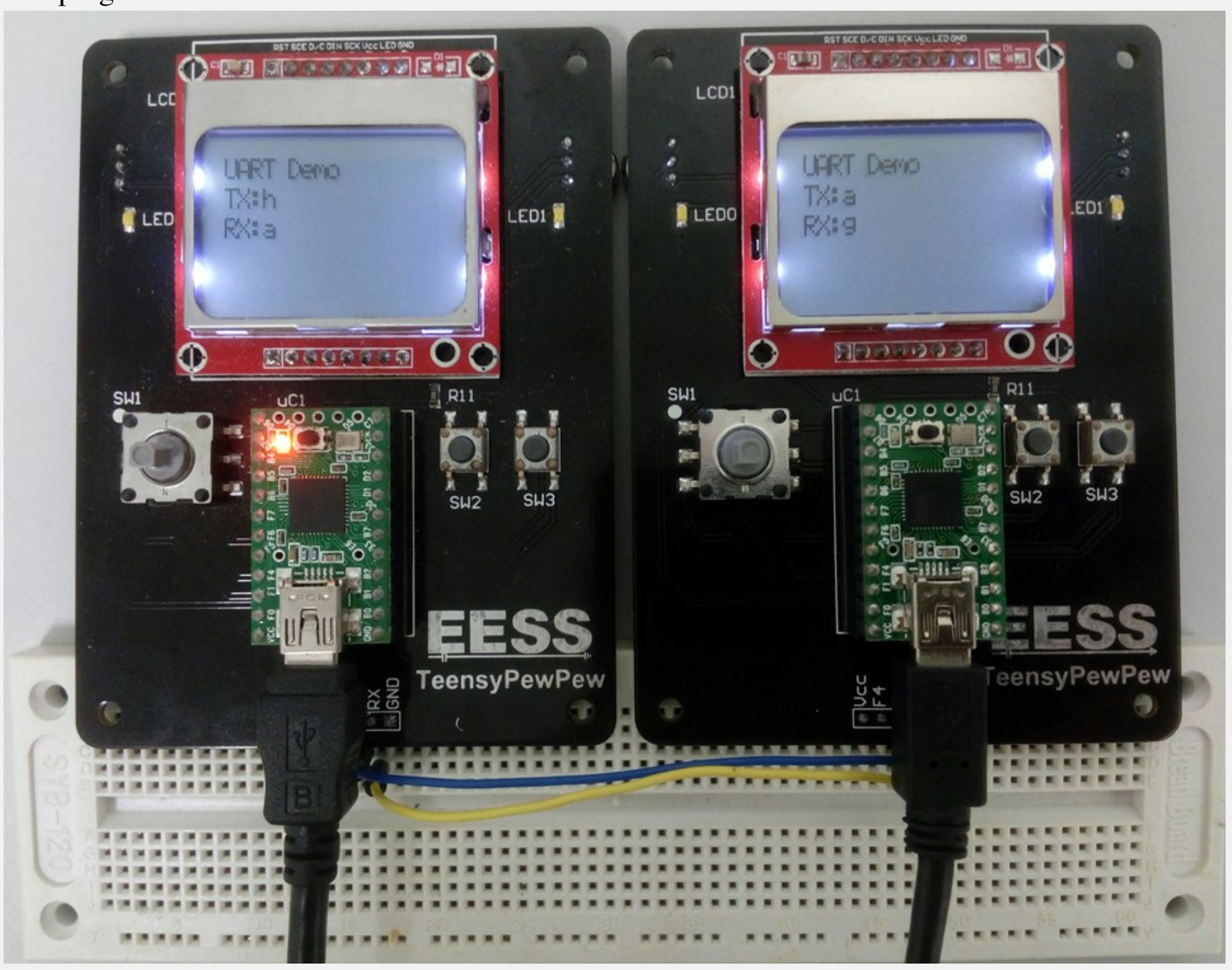
- A sample program, **`uart_hello.c`**, is listed below.

```
/*
**      uart_hello.c
**
**      Demonstrates bidirectional communication between two TeensyPewPew
**      over UART.
**
**      Lawrence Buckingham, QUT, October 2017.
**      (C) Queensland University of Technology.
*/
#include <stdint.h>
#include <string.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <cpu_speed.h>
#include <macros.h>
#include <graphics.h>

#include "uart.h"
```

- In this program, two TeenyPewPew devices talk to each other over the **`UART`** connection.
    - In **`setup`**:
        - **`UART`** is initialised, running at 9600 bits per second.
        - Timer 0 is enabled with pre-scale value 256 so that it will overflow about 122 times per second, and the Timer Overflow interrupt is enabled for Timer 0.
    - Each time **`process`** is called:
        - An incrementing character is sent to the other Teensy via **`uart_putchar`**.

        - The transmitted character is displayed on the LCD.
        - If an incoming character is available:
            - **`uart_getchar`** is called to get the character.
            - The received character is displayed.
    - If we start one Teensy and then delay for a little while before starting the second, the difference between the sent and received character shows up.
- The Timer 0 overflow **`ISR`** accumulates the elapsed time, and every time a second passes, it toggles the LED on pin **`D6`**.
- ***Listen up! The only pins you need to connect are `TX` and `RX`.***
    - Please **do not** connect **`VCC`** (power) to some random pin and hurt your Teensy...
    - Your tutor will have a breadboard and jumper cables in class if you want to try this.
    - The safest thing to do is to come to the tutorial.
    - FYI: replacement cost for fried Teenies is about $30.00.
    - Try not to destroy the device, thanks.
- To test the program, you will need:
    - Two TeensyPewPew devices;
    - A breadboard;
    - A pair of jumper cables;
        - Connect **`TX`** on the first device to **`RX`** on the second.
        - Connect **`RX`** on one first device to **`TX`** on the second.
- Install **`uart_hello.hex`** on both devices. They should talk to each other.
- The program looks like this when it runs:



USB Serial Programming

Introduction

- We can use USB to implement serial communication between our computer and a Teensy.
- Rather than taking a deep dive into implementation detail, we will use the PJRC **`usb_serial`** library.
- Applications:
  - Debugging – we can send diagnostic information to the computer display.
  - Game control – we might use the computer keyboard to control a program running on the Teensy.
- The API summarised in **`usb_serial.h`** contains the following functions:

```
// setup
void usb_init(void);                    // initialize everything
uint8_t usb_configured(void);           // is the USB port configured

// receiving data
int16_t usb_serial_getchar(void);       // receive a character (-1 if timeout/error)
uint8_t usb_serial_available(void);     // number of bytes in receive buffer
void usb_serial_flush_input(void);      // discard any buffered input

// transmitting data
int8_t usb_serial_putchar(uint8_t c);   // transmit a character
int8_t usb_serial_putchar_nowait(uint8_t c);  // transmit a character, do not wait
int8_t usb_serial_write(const uint8_t *buffer, uint16_t size); // transmit a buffer
void usb_serial_flush_output(void);     // immediately transmit any buffered output

// serial parameters
uint32_t usb_serial_get_baud(void);     // get the baud rate
uint8_t usb_serial_get_stopbits(void);  // get the number of stop bits
uint8_t usb_serial_get_paritytype(void);// get the parity type
uint8_t usb_serial_get_numbits(void);   // get the number of data bits
uint8_t usb_serial_get_control(void);   // get the RTS and DTR signal state
int8_t usb_serial_set_control(uint8_t signals); // set DSR, DCD, RI, etc
```

- Usage is similar to **`UART`** coding.
- In **`setup`**:
  - Call **`usb_init`** to establish a connection.
  - Call **`usb_configured`** to verify that we're connected.
- To read data from the computer:
  - **`usb_serial_available`** returns a non-zero value if data is available.
  - **`usb_serial_getchar`** returns a byte, or **`-1`** if no data.
- To send data to the computer:
  - **`usb_serial_putchar`** transmits a byte.
  - **`usb_serial_write`** transmits an array of bytes.
    - To send a string, use a cast operator:

```
#include <string.h>
char * s = "hello";
. . .
usb_serial_write((uint8_t *) s, strlen(s));
```

Transmit from Teensy to computer (usb_serial_hello.c)

- Here is a very simple "hello world" program for USB serial: **`usb_serial_hello.c`**.

```
/*
**      usb_serial_hello.c
**
**      Demonstrates serial transmission from Teensy to desktop via USB.
**
**      Lawrence Buckingham, QUT, October 2017.
**      (C) Queensland University of Technology.
*/
#include <stdint.h>
#include <string.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <cpu_speed.h>
#include <macros.h>
#include <graphics.h>

#include "usb_serial.h"

void setup_usb_serial( void );
```

- Start the program on your Teensy, then open a Putty/screen session.
- You should see a repeating message appear every 2 seconds.

Two-way communication between Teensy and computer (usb_serial_echo.c)

- This program reads a sequence of characters via USB serial and echoes them back to the sender with a message: **`usb_serial_echo.c`**.

```
/*
**      usb_serial_echo.c
**
**      Demonstrates character receive and transmit over USB serial.
**
**      Lawrence Buckingham, QUT, October 2017.
**      (C) Queensland University of Technology.
*/
#include <stdint.h>
#include <string.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include <cpu_speed.h>
#include <macros.h>
#include <graphics.h>
#include <stdio.h>
#include "usb_serial.h"

void setup( void );
```

- Start the program on your Teensy, then open a Putty/screen session.
- Type letters in your terminal, and what you type should come back.

Useful stuff

Makefile for the sample programs listed in this document

- To build the programs in this lecture:
  - Download the USB Serial and UART libraries from PJRC (See References).
    - Mac users will need to replace **`usb_serial.c`** with this: modified usb_serial.c.
  - Execute the **`Makefile`** in each of the two library source directories.
- Use the following **`Topic 10 Makefile`** to build the programs.
  - Verify and if necessary update paths to the location of **`libzdk.a`**, **`libcab202_teensy.a`**, **`usb_serial.o`**, **`uart.o`**

```
# Topic 10 Makefile (with USB Serial and UART)
# Lawrence Buckingham, October  2017.
# Latest revision: May 6 2019.
# Queensland University of Technology.

# Replace these targets with your target (hex file) name, including the .hex part.

TARGETS = \
        uart_hello.hex \
        usb_serial_hello.hex \
        usb_serial_echo.hex

# Set the name of the folder containing libcab202_teensy.a
CAB202_TEENSY_FOLDER=../cab202_teensy

# Set the name of the folder containing usb_serial.o
USB_SERIAL_FOLDER =../usb_serial
USB_SERIAL_OBJ =../usb_serial/usb_serial.o

# Set the name of the folder containing uart.o
```

The End
