Lab 1: Arduino Programing and Serial Communication

1. Introduction

Microcontrollers are used to control many types of systems, including microwave ovens, robotic systems, and automobile components (engines, transmissions, braking, traction control, etc.). Microcontrollers are small, relatively inexpensive, and operate at ever-increasing speeds, which makes them ideal for control applications where a large computer is not necessary, or the space is not available. In this course, we will use the Arduino Mega and Arduino Uno microcontroller boards, which are based on ATMEL's ATmega series of microcontroller chips. They both use the same Arduino software, but the Mega has more input and output capabilities than the Uno. The following introduction and laboratory procedures are presented to provide an understanding of the basic functions and operation of these microcontrollers. In this lab, you will cover the basics of syntax, timing, serial monitor, and serial/wireless communication.

We will also use this lab to review and practice proper soldering technique, which will help you be successful in upcoming lab and project activities.

1.1 Lab Objectives

- Learn Arduino Program Structure
- Learn methods for Serial and Wireless Communication using Arduino
- Learn strategies for data packaging and communication timing
- Learn proper soldering technique

1.2 Project Objectives

- Brainstorming concepts

1.3 Lab Hardware

- 1 – Arduino Uno
- 1 – Arduino Mega
- 2 – USB cable
- 1 – Pair of Xbee radios and shields
- 1 – Jumper wire kit

1.4 Project Hardware

- Hardware details

2. Laboratory Concepts

2.1 Arduino Basics

The Arduino is a small single-board microcontroller with simple open source software and hardware. This board was originally developed in 2005 in Italy with the thought of developing a cheap, simple, and open-architecture microcontroller. Several varieties of these boards are available for users, including the Arduino Mega, Nano, and Uno.

Powering the boards

The Arduino can be powered in 3 ways: connecting an external power source between 7 to 12 volts to either the power jack or to the V_{in} pin, or connecting the Arduino to a computer with a USB cable. If the Arduino is powered with an external power source it will automatically regulate the voltage so that the Arduino outputs 5V. Using this method, the Arduino can directly power any device so long as the Arduino's max current draw (40 mA) is not exceeded. If the Arduino is powered with a USB cable, the maximum current draw of the USB port must not be exceeded to not damage the USB port on the computer.

Arduino programming language

The Arduino is programmed using the cross-platform Arduino software which can be downloaded from the Arduino website (http://arduino.cc/en/Main/Software). A program is called a sketch in Arduino and is very similar to other programming languages such as C.

The biggest difference between these two languages is that while C has a "main" function, the Arduino has "setup" and "loop" functions. The "setup" function is run once each time the board is powered up or reset. Its purpose is to initiate the operations needed for your code. The "loop" function runs continuously after the "setup" function has finished. The processor will execute the operations inside the "loop" as fast as it can and then begin again at the top of the loop. Again, this process repeats indefinitely.

The Arduino runs the code line by line. Different lines of code will take different amounts of time to run. For instance, assigning a number to a variable or simple integer math (addition and subtraction) takes less time than multiplication or division. Other tasks such as analog to digital conversion (A/D), serial communication, and performing floating point math will take orders of magnitude more time. Sketches can consist of comments, libraries, variables, and functions.

Comments

Comments are for the sake of documentation so that the programmer and others will have an easier time understanding the code. Comments describe what that line(s) of code does in plain English. There are two ways to make a comment:

- // comments the characters to the right of the backslash. This method comments one line at a time.
- /* before and */ after a comment will comment everything between the backslashes. This can be used to comment multiple lines.

Libraries

Libraries are made by advanced users to make complicated functionalities usable by users of any level. For example, including the Servo library is done by using the #include command, which imports functions for controlling a servo. Some libraries are included in the Arduino IDE. Others can be found for hardware online. They are usually in a zip file and need to be extracted to the libraries folder of the Arduino IDE system files. Another way is to import them through the Arduino IDE (Sketch>>Import Library).

Variables

Variables are how data is stored. Each variable has a name, type, and a value associated with it. The name of a variable can be anything that contains only letters, underscores, or numbers. Variables must start with a letter and are case sensitive (i.e. "variable_name" is not the same as "Variable_Name"). To help the TA's and fellow students understand your code, pick names that describe the purpose of your variables.

The type of a variable informs the program what kind of variable it should be expecting. Once a variable is set to be of a certain type, it cannot be redefined. A variable is given a type when it is first created. The figure below shows Arduinos most common data types.

If the value of a variable exceeds the range of its data type it is called overflow, which would result in its value resetting back to the other end of the variable's range.

Name	Description	Size (Bytes)	Range	
			Min	Max
char	character or small integer	1	-128	127
unsigned char	character or small integer	1	0	255
int	Integer	2	-32768	32767
unsigned int	positive Integer	2	0	65535
long	Long Integer	4	-2147483648	2147483648
unsigned long	positive long integer	4	0	4294967295
boolean	Boolean value	1	FALSE (0)	TRUE (1)
float	Floating point number	4	+/- 3.4e +/- 38 (7digits)	
double	Double percision float	8	+/- 1.7e +/- 308 (15digits)	

Figure 1. Arduino data types with descriptions, memory size, and data range.

2.2 Common Functions

Just like any other programming language, functions can be written in Arduino to organize code and make debugging much easier. Users can write functions, import functions from libraries, or use the functions built into the Arduino software. Below is a list of functions you will need for this lab. A more complete list and descriptions can be found at https://www.arduino.cc/en/Reference/HomePage.

Mandatory functions: *setup()* and *loop()*

These two functions must be included in every sketch. The *setup()* function is only called once at the beginning of the code. The *loop()* function is the heart of a sketch and is repeatedly called until the Arduino is powered off.

Pin Input/Output

- `pinMode(pin, mode)`: Configures the specified pin to behave either as an input (reads a voltage) or an output (outputs a voltage).
- `digitalWrite(pin, value)`: Output a HIGH (5V) or a LOW (0V) voltage from a digital pin. Note that on most Arduino boards this can be used on pin 13 (once its mode has been set to output) to control the on-board LED.
- `digitalRead(pin)`: Reads the voltage at a specified digital pin, either HIGH (5V) or LOW (0V).
- `analogWrite(pin, value)`: Writes an analog value (PWM wave) to a pin. The 'value' is the duty cycle: between 0 (0% duty cycle) and 255 (100% duty cycle). Note that on most Arduino boards this can be used on pin 13 (once its mode has been set to output) to control the on-board LED.
- `analogRead(pin)`: Reads the voltage from the specified analog pin.

Time

- `millis()`: Returns the number of milliseconds since the Arduino board began running the current program.
- `micros()`: Returns the number of microseconds since the Arduino board began running the current program.
- `delayMicroseconds(us)`: Pauses the program for the time specified (in microseconds).

2.3 Serial Communication

One of the basic functions of microcontrollers is Serial Communication, which allows for communication between your microcontroller and your computer, or between multiple microcontrollers and/or other peripheral devices.

Arduino Serial Communication Hardware

Arduinos perform serial communication on dedicated serial ports. Each serial port is bidirectional, and requires two pins, a TX pin for transmitting data, and an RX pin for receiving data. The Mega has four hardware serial ports, while the Uno only has one. The USB connection made between the Arduino and the computer uses the first serial port on any Arduino board. This means digital pins 0 and 1 (labeled RX and TX, respectively) are dedicated to communicating with a computer and should not be used for other purposes. Otherwise, you will interfere with the ability of the computer to upload new code to the board. On the Uno, which only has a single dedicated serial port, this can be a problem if you wish to communicate with another device other than a computer. However, an external software package called SoftwareSerial.h can be used to convert

two PWM pins on any Arduino board into a new serial port. The Mega, on the other hand, has three additional dedicated serial ports which use the pins labeled TX1/RX1 through TX3/RX3.

Serial Communication Protocol

Serial communication is digital in nature, meaning information is sent and received as a sequence of binary on/off signals called bits (1 or 0). Arduinos use a UART (Universal Asynchronous Receiver/Transmitter) serial communication protocol that sends/receives information one byte at a time (1 byte = 8 bits). Both the sender and the receiver need to be expecting the data to be sent at the same rate. This is called the BAUD Rate. The UART communication protocol has been standardized for various baud rates. For example, 9600 baud (bits per second) is a common rate that we will use in this lab. If the sender and receiver are expecting different baud rates they will incorrectly interpret the incoming data.

One byte can represent 2^8=256 different numbers (i.e. integers 0-255). Thus, it's straightforward to transmit an integer number between 0 and 255. However if we want to transmit anything else (e.g. larger integers, floating point numbers, letters and other characters), we need some way to encode the information as bytes. A common coding scheme (which the Arduino employs), is the ASCII (American Standard Code for Information Interchange) standard (pronounced ask-ee). Each keyboard character (including letters, numbers, symbols, control commands) is represented by a unique integer number between 0 and 255 (see ASCII standard table for reference). For example, capital 'A' has an ASCII value of 64, lower case 'a' has an ASCII value of 97, the number '4' treated as a character has an ASCII value of 52, and an exclamation point '!' has an ASCII value of 33.

Sending Information

Two common functions for sending information via serial communication are the *print()* and *write()* functions. These functions are similar, but have some key differences. The *print()* function is straightforward because it always treats its input argument as a string of ASCII characters, converts each character to an ASCII value, and then sends that ASCII value. Here's a few examples:

Code:	Information being sent:
Serial.print("Hello!");	72 101 108 108 111 33
Serial.print(100);	49 48 48
Serial.println("Hi");	72 105 10

Table 1: Serial.print() examples.

Note that in the 2nd example, the *print()* function doesn't care whether you put the number in quotes or not. Notice in the 3rd example we only typed two characters but three numbers were sent. This is because the *println()* function adds a newline or linefeed (LF) control character (ASCII code is 10) to the end of whatever information you input. An alternate way to achieve the same result would be *Serial.print("Hi\n")*, where the '\n' is code (a.k.a. escape sequence) for newline.

The *write()* function, on the other hand, does not automatically do an ASCII conversion on every input. If you write a number between 0 and 255, it will send that number. If you *write()*

anything other than a number, it will convert it to its corresponding ASCII value and then send it just like the *print()* function does. Here are a few examples:

Code:	Information being sent:
Serial.print(100);	49 48 48
Serial.write(100);	100
Serial.write(38);	38
Serial.write("Hi");	72 105
Serial.write(1000);	232

Table 2: Serial.print() and Serial.write() examples.

Note that in the last row of Table 2 we tried to send the number 1000, but the value 232 was sent instead. This is because the *write()* function can only send one byte at a time. In this case, the value of 1000 overflows one byte, and wraps around to a value of 232. (We'll discuss binary overflow in Lab 2).

Now that we understand how the *print()* and *write()* functions work we need to take a moment to understand the Serial Monitor because this will be a common place for us to look at the results of calling *Serial.print()* and *Serial.write()*. The Serial Monitor is the window in the Arduino App on your PC that lets you send and receive information from your Arduino via the USB tether. The Serial Monitor always uses a reverse ASCII conversion to decode the information that is sent to it. It assumes ASCII values are being sent to it, and therefore converts the numbers it receives into ASCII characters before outputting them to the screen. Here are some examples:

Code:	Information being sent:	Serial monitor output:
Serial.print("Hello!");	72 101 108 108 111 33	Hello!
Serial.print(100);	49 48 48	100
Serial.write(100);	100	d
Serial.write(1000);	232	؟

Table 3: Serial Monitor examples.

Note that if we want the Serial Monitor to output "100" we need to use *Serial.print(100)*, or we need to call *Serial.write(49); Serial.write(48); Serial.write(48);*. We see that *Serial.write(100)* results in the ASCII character "d" appearing because that corresponds to the ASCII value of 100.

Note that in the last example, the Serial Monitor outputs a backward question mark. This is because the ASCII value of 232 corresponds to the character è (from the extended ASCII table) which Arduino doesn't know.

Receiving Information

As previously discussed, serial data is sent one byte at a time. The device on the receiving end of the connection receives and automatically stores those bytes of data in a buffer, which is a spot set aside in the Arduino's RAM. When the receiving Arduino executes a *Serial.read()* statement, it retrieves the first available byte in its serial buffer. The next time the receiving Arduino executes a *Serial.read()* the next byte will be retrieved, and the prior byte will be lost

unless you stored it or otherwise did something with it. This highlights that the *Serial.read()* function handles only one byte at a time, even if the total amount of information received was larger than one byte.

The serial buffer on the Arduino is a finite resource, and it will fill with data unless your code reads or clears it. (The size of the serial buffer varies by model.) You can check if the buffer is full with *Serial.overflow()*. See:
https://www.arduino.cc/en/Reference/SoftwareSerialOverflow

Recall that each byte received is a number from 0 to 255. If we store this information in a numeric type of variable (int, float, long, byte) it will remain as a number. But if we store the received information in a char type variable it will be converted to an ASCII character.

Info being received:	Code:	Result:	Serial monitor output:
72	**int** info = Serial.read(); Serial.**write**(info);	info = 72;	H
72	**int** info = Serial.read(); Serial.**print**(info);	info = 72;	72
72	**char** info = Serial.read(); Serial.**write**(info);	info = 'H';	H
72	**char** info = Serial.read(); Serial.**print**(info);	info = 'H';	H

Table 4: Storing serial data.

The result column shows what a receiving device would store, the serial monitor output column shows what would happen if you then used *print()/write()* after receiving the info in a certain way. This highlights that you need to be careful and intentional with how you send and receive information between devices and when using the serial monitor.

Data Packaging

It is often the case that more than one value needs to be transmitted between two devices using serial communication. When this happens, the receiving device needs to know what each number it is receiving should correspond to. In these instances, you need to structure and send your data in an intentional way that is easy to decode and use once received. This is called packaging your data or sending a data packet. One typical way is to have a starting byte of information indicate the beginning of your data packet. This is sometimes called an indicator or flag. For the lab exercises below, you will use the integer 255 as the flag. Whatever value is selected, you **MUST** ensure that this flag value is unique and not achievable by any of the other variables present in the data packet. For example, in this lab, no other piece of information in the data packet being sent is allowed to be 255 (you can check the code when you get to this step of the lab to confirm that this statement is true). After sending the flag, you then send all your information in a specific, intentional order. The receiving side then waits until it sees the flag, and then knows to read the subsequent data and store it in the correct variables.

Using even this simple example of data packaging provides great protection against receiving data in the wrong order. Suppose, for example, a transmitting device turns on before the

receiver, and starts repeatedly transmitting 10 bytes of data. When the receiving device turns on and starts receiving data, it might start receiving in the middle of the transmission. By waiting until it sees the start flag, the receiver knows for sure it is starting at the beginning of the data.

Decoding this particular data packaging structure is achieved as follows. Your receiving code will wait until the necessary number of bytes becomes available. For example, if you are sending three pieces of information (including the start bit), the receiving code will wait until three or more bytes are available to process. Next, the first byte is observed and compared to what is known to be the flag value. If this is not the expected value, this byte should be discarded, and the next byte investigated. Once the expected starting value is found the remainder of the data packet can be sequentially analyzed in the order in which it is expected to be seen. Pseudocode representing this receiving process follows.

```
If 3 bytes of data are available
    If Serial.read() equals startFlagValue
        Variable1 = Serial.read()
        Variable2 = Serial.read()
```

In other cases, we might need to include integers larger than 255, or floating point numbers, in our data package. One simple way to do this is to send and receive them as a string of characters using *Serial.print()* and *Serial.readStringUntil()*. For example, if you want to transmit the numbers 543 and 1.234, you might send them using *Serial.print("534,1.234 ")*, with a space at the end. On the receiving end, you could then use the following code to parse the string and convert it to numbers:

```
String myString1 = Serial.readStringUntil(",");
String myString2 = Serial.readStringUntil(" ");
int x = toInt(myString1);
float y = toFloat(myString2);
```

The *readStringUntil()* command will read as many characters from the buffer as necessary until it encounters the specified character (in this case a space).

2.4 Soldering Practices

Read the guidelines below and watch the video at https://www.youtube.com/watch?v=IpkkfK937mU.

Using Flux

While many variations of solder exist, one of the common differentiations is flux core vs. non-flux core. The solder you find in the lab should all be flux-core. Flux is a key component of successful soldering. Flux cleans oxidation off the metal surfaces that you are bonding. Without using flux, your solder joints may be weak or not form at all.

Clean Your Iron's Tip

The iron's ability to transfer heat is greatly reduced when covered in burnt flux and oxidation. You can clean a heated tip by wiping it on a wet sponge or using a dry cleaner. It is recommended that you do this every time you pick up the iron. After cleaning, tin the tip by touching a small amount of solder to it, which helps transfer heat and creates a solder bridge to start your joint.

Prepping Your Work

Taking the time to prepare your work correctly can make soldering the connection much easier.

- Clean components: Flux will only take care of minor oxidation. Steel wool, or sometimes an eraser can be used to clean your components.
- Clamping your work: unfortunately, engineers have not evolved a third hand yet, so it is often useful to clamp your work to hold it together.
- Tinning components: often your work can be made easier by tinning first. To tin a component, hold the iron against one side and apply the solder to the opposite side. This allows the solder to flow cleanly and creates a smooth, even finish. Tinning is recommended when using stranded wire, or when connecting a wire to a header pin.
- Heat sink: some components such as transistors and diodes are sensitive to heat. Attaching a heat sink clip between the component and the joint will dissipate some heat.

Soldering a Joint

First heat the joint by placing the iron against both the component and pad. Second, make a heat bridge by touching the solder between the iron and component (if you have enough solder on the tip, this may not be necessary). Finally, touch the solder to the opposite side of the component allowing it to spread and cover the joint. Make sure you remove your solder feed before removing the iron! This should only take a few seconds to complete; longer applications of heat can damage components and cause pads to lift away from boards.

Desoldering

Two common ways to remove solder are by using solder wick or a solder sucker. Solder wick is a braided weave of copper strands coated in flux; place the wick over the joint and heat using the iron on top of the wick allowing the solder to absorb into the braid. A solder sucker sucks up molten solder; press the plunger down, heat up the joint, then quickly replace the iron tip with the sucker and press the release button.

What is a Good Joint

A good solder connection will resemble a shiny Hershey's Kiss. A dull or textured finish is called a 'cold joint' and has not made a successful bond – it is often caused by poor heating.

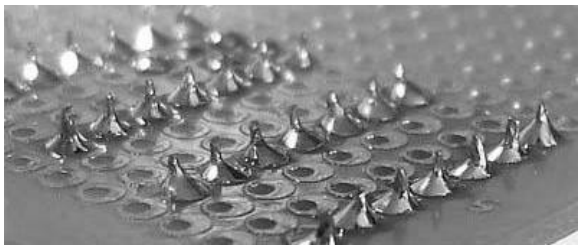

Figure 3. Examples of good solder joints.
http://store.curiousinventor.com/guides/how_to_solder/heat_solder

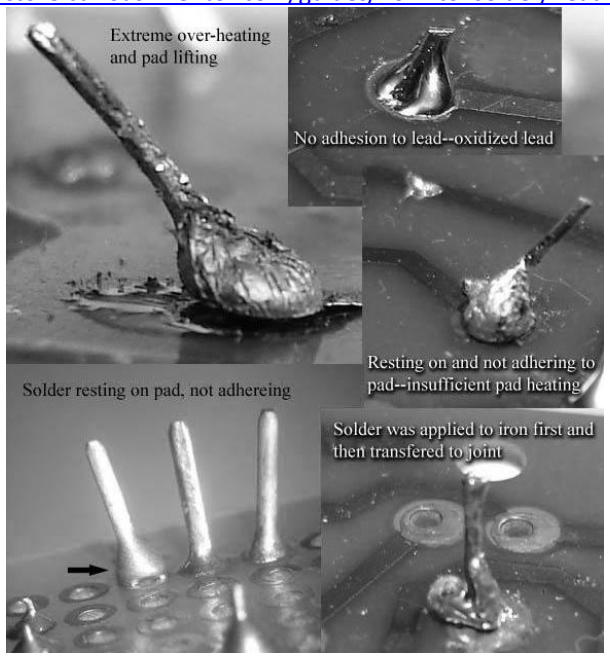

Figure 4. Examples of bad solder joints from
http://store.curiousinventor.com/guides/how_to_solder/heat_solder

Strain Relief

It is not uncommon for wires to be pulled on during operation. It is important, therefore, to relieve the strain at your joints. There are a number of ways to do this, a few are listed below.

- Heat Shrink: heat shrink covers solder joints, which protects from accidental shorting and reinforces the solder joint. Slide the tube over your wire before soldering the joint (be sure it is far enough away from your joint that the heat from soldering won't shrink it). Once the joint is soldered slide the tubing over your joint and heat using a heat gun.
- Wire Ties: use wire ties to secure wiring at key locations. One such application is securing wires from motor terminals. Fold the wires back along the motor and secure a wire tie around the motor housing and wires; this will keep the wires from pulling at motor terminals which can easily break.
- Bundling: Bundling groups of wires can keep them from becoming tangled and pulling at each other. Bundling also provides a means of organization.

3. Pre-Lab Exercises

Many of the answers can be found in the lab handout, but some require you to investigate the Arduino documentation online.

1. Arduino basics
   a. Where in the Arduino IDE (version 2 or later) do you set the Arduino board and port to connect to a device?
   b. How do you import a new library into the Arduino IDE and what is the syntax for adding the library to a specific sketch?
   c. How many times do the setup and loop functions run in a sketch?
   d. If you want to declare analog pin 7 as a variable, what is the syntax (assume pin variable name is `inputPin`)?
   e. Where do you place and what is the syntax to setup a digital pin as an input? (assume digital pin 10 is the pin being used).
   f. What is the syntax that is used to prepare the Serial monitor for display at a baud rate of 115200?
   g. If you want to light up the UNO onboard LED, what is the pin you would use?
   h. What is the maximum current that can be supplied through any I/O Arduino pins?
   i. What is the syntax to print the value of the variable "val" to the serial monitor with a new line each time it prints?

2. Arduino Serial communication
   a. The USB connection between an Arduino and a computer uses the board's primary dedicated serial port. Which digital pins are associated with this primary serial port?
   b. What does the function *Serial.available()* do?
   c. What are the numeric values of the bytes of data being sent based on the following code: *Serial.println("I'm 22!");*
   d. What will appear in the Serial Monitor based on the following code? (This code assumes an Arduino Uno sending info across the mySerial channel which was set up using the SoftwareSerial.h library, and an Arduino Mega receiving that information on the Serial1 channel):

      Sending Code:
      mySerial.print(9);

      Receiving Code:
      int data = Serial1.read();
      Serial.print(data);

4. Laboratory Exercises

4.1 Hello World

This exercise will allow you to practice setting up an Arduino sketch and printing to the Serial monitor. You will write a sketch that prints the running time of the sketch as well as "Hello World" at 2 second intervals.

4.1.1 Prepare Arduino connections and sketch settings.

- Using the USB cable, plug in the Arduino Uno to the computer.
- Set the board and port to the correct device

4.1.2 Prepare a sketch

- Open a new file – File >> New
- Save file – File >> Save, save as "Lab 1 Hello World" in a new desktop folder or on a USB flash drive.

4.1.3 Write "Hello World" code

- Initialize a variable called ***time*** and ***timeold*** as ***unsigned long*** before void setup().

```
unsigned long time = 0;
unsigned long timeold = 0;
```

- Setup Serial Monitor (in void setup) – the serial monitor should run at a baud rate of 9600

```
Serial.begin(9600);
```

- In void loop(), Set the value of ***time – time*** should be set as the current run time each loop in milliseconds.

```
time = millis();
```

- In void loop(), Write a logic statement that prints the following statement every 2 seconds without using the delay function.

  Hello World – 2 sec
  Hello World – 4 sec
  Hello World – 6 sec
  …

```
if(time-timeold >= 2000){          //Check if the time difference
  Serial.print("Hello World - "); //Print text to serial monitor
  Serial.print(float(time)/1000); //Print time in sec
  Serial.println(" sec");         //Print sec
  timeold = time;                 //Set time for previous print
}
```

4.1.4 Compile and Upload

Once you have finished, have a TA check your code.

4.2 Serial Communication

This exercise will run you through the process of setting up the Serial communication channel between an Arduino Uno and Mega. These sketches will allow you to type through the Serial monitor on one device and print it on the other device.

4.2.1 Download the **BasicCommsUno.ino** and **BasicCommsMega.ino** sketches from the class page.

- These sketches contain the basic setup for serial communication between two Arduinos.
- **BasicCommsUno.ino** creates a UART channel using the digital pins 10 and 11 (RX, TX). Using the SoftwareSerial library you can set up a comms channel with any two digital pins not being used.
- **BasicCommsMega.ino** uses a second dedicated serial port, Serial1, which is on pins 18(TX) and 19(RX).

4.2.2 Wire up the Arduino Uno and Mega communication channel

- From the jumper wire kit connect the digital pin 19 of the Mega to the digital pin 11 of the Uno. (Uno sending to Mega receiving).
- From the jumper wire kit connect the digital pin 10 of the Uno to the digital pin 18 of the Mega. (Mega sending to Uno receiving).
- From the jumper wire kit connect the GND pin of the Uno to the GND pin of the Mega.

4.2.3 **BasicCommsUno.ino:** Develop a communication method - add both of the following protocols to the void loop

- Sending protocol – the following lines are code for a sending protocol

```
// If something is typed into the Serial Monitor
if (Serial.available()) {
  // Read it and send it to the other Arduino
  mySerial.println(Serial.readStringUntil('\n'));
}
```

- Receiving protocol – the following lines are code for a receiving protocol

```
// If something it sent from the other Arduino
if (mySerial.available()) {
  // Read it and display it on the Serial monitor
  Serial.println(mySerial.readStringUntil('\n'));
}
```

4.2.4 **BasicCommsMega.ino:** Add the same lines of code as above to the Mega file but replacing “mySerial” with “Serial1”.

4.2.5 Save, compile, and upload to both the Mega and Uno.

- Press the Verify, check mark to verify
- Upload the files to the Uno and Mega

- Make sure the correct board and port are set for each file.
- Press the right arrow to upload
- You should be able to send a typed message in the Uno Serial Monitor and have it appear on the Mega Serial Monitor.

Once you have finished, have a TA check your code.

4.3 Wireless Communication

Like the Serial communication exercise, this exercise will run you through the process of setting up the wireless communication channel with the Arduino Uno and Mega. In this exercise, you will use the Uno to send pretend LED and servo motor commands to the Mega using a packaging method. The Uno should be able to send two numbers. One value should be a 0 or a 1. The other should be a value between 0 and 254. The Mega should turn on the onboard LED if the first value is 1 and print the second value to the serial monitor.

4.3.1 Set up your files

- Save a copy of BasicCommsUno as UnoSending
- Save a copy of BasicCommsMega as MegaReceiving

4.3.2 Prepare Mega and Xbee

- Remove the two jumper wires; the Xbee protocol remains the same.
- Connect the Xbee shields to the Uno and Mega normally. This is done by aligning the Xbee shield pins with the corresponding Arduino pins and inserting it on top. The Xbee shield uses pins 2 and 3 for serial communication.
- Make sure the switches on both Xbee shields are set to DLINE. DLINE means the pins using the serial communication are the SoftwareSerial pins and not the dedicated TX and RX used with the USB (Figure 2).

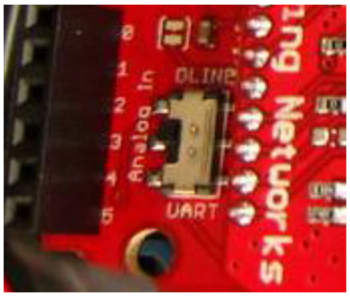

Figure 2. Xbee communication pin selection switch.

- On the shield inserted into the Mega, connect pin 2 to 19 and pin 3 to 18 using jumper wires.
- In UnoSending, set the SoftwareSerial pins to 2 and 3.

4.3.3 Once you have installed the XBee shields, and connected the jumper wires, check to make sure your hardware is working. You do this by running the same code on your Uno and Mega as from the wired communication procedures. Whether your systems are wired together or sending data wirelessly via XBees, the code is the same, and therefore should continue to work the same way now that you have switched from wired communication to wireless.

- If your communication is not working at this point, but was previously working for the wired communication, ask a TA for help checking whether your hardware is wired and functioning properly.

4.3.4 Using the protocols described below create the communication described in the section summary above. You may need to create new variables.

- UnoSending blink and Sine wave servo motor commands

```
t_ms = millis();
t = float(t_ms) / 1000.0;

// Blinking an LED without using a delay() function
if (t - t_old > 1) {
  if (LEDval == 0) {
    LEDval = 1;
  } else {
    LEDval = 0;
  }
  t_old = t;
}

// Sine wave for a servo
servoAngle = int(125 * sin(t)) + 127;
```

- UnoSending communication protocols. The following lines are code for a sending protocol. This is one method of packaging data. A flag variable is sent, followed by one byte of data at a time in a specific, intentional order. There are many other methods for packaging data.

```
mySerial.write(255);        // Send starting flag value
mySerial.write(LEDval);     // Send LED command
mySerial.write(servoAngle); // Send servo command
delay(20);
```

- MegaReceiving communication protocols. The following lines are code for a receiving protocol. (Be sure to initialize and set the LEDpin as an output.)

```
if (Serial1.available() > 2) {          // 3 or more bytes available
  if (Serial1.read() == 255) {          // Find the starting flag value
    receivedLEDValue = Serial1.read();  // 1st piece of data is the LED value
    receivedServoAngle = Serial1.read();// 2nd piece of data is the servo angle
  }
}
// The following lines of code are called continuously
digitalWrite(LEDpin, receivedLEDValue);
Serial.println(receivedServoAngle);
```

4.3.5 Compile and Upload

- Press the Verify check mark to verify
- Press the right arrow to upload

Once you have finished, have a TA sign off:

4.4 Soldering Practice

During this exercise you will practice soldering pins and wires to a prototyping ("proto") board. You will also apply heat shrink to provide strain relief.

4.4.1 Obtain the following materials

- Protoboard
- Scrap wire
- Solder
- Header pins (a male set and a female set)
- Heat shrink tubing
- Safety glasses

4.4.2 Go to a soldering station to practice the soldering techniques

4.4.3 Practice doing the following, a couple times each

- Using safety glasses
- Solder wire to the short end of a male header pin
- Solder female header pins to the protoboard
- Apply heat shrink to the solder joint on the male headers (leave at least one solder joint exposed for inspection)

4.4.4 During practice check with the TA to make sure your soldering joints are done properly.

Once you have finished, have a TA sign off.

Once you have cleaned up your work area have your TA sign off.

5. Post-Lab Exercises

1. List five or more Arduino functions that you think will be helpful in programming your robot and describe what they would do for your robot.

2. Based on this semester's competition, discuss how you could use Xbee wireless communication. What commands/information you would send from the Arduino Uno, and at what point during the system's operations? How would your system store this information and/or what actions would it take based on it?

3. Imagine you need to transmit three pieces of data from an Arduino Uno (being used as a remote control) to an Arduino Mega (which is operating an RC car that you built). The three pieces of information are: a servo angle of 120, a motor power value of 0, and an LED color value of 'r'.
    a. Provide code, or pseudocode, for the Uno of how you would package this data to send it.
    b. Provide code, or pseudocode, for the Mega of how you would receive this data packet and store the correct values in the correct variables.

6. Project Milestone 1

For Lab 2, each student should prepare at least 1 brainstorming concept (sensors, mechanisms, etc), as well as an accompanying description of the robot's operation and autonomy, for completing the following objectives of the competition course:

- Driving/navigating around the playing area
- Obtaining/sensing blocks
- Delivering/placing blocks

As well as individual concepts for each of these objectives, provide a combined robot concept that could complete all the objectives. This can be a combination of the individual concepts you created above or new concepts. With your combined robot concept provide a concept for autonomous strategy. Your concepts should be combined into an organized document. This document will be used for discussion during the first project milestone meeting.

Your concepts should include drawings and descriptions of mechanisms, types of sensors, and control strategy. You may also include a description of how you will use wireless communications.

During Lab 2, you will be formed into teams. You will meet with your team and a TA to discuss Team formation and the next Project Milestone. During this meeting, you will present your brainstorming concepts from above. You should use hand drawings, digital sketches, PowerPoint, or any other organized method for presenting designs and descriptions. **Submit a scan of your concepts in PDF format to Canvas.**