Lab 1: An Introduction to Arduino: From Flashing Lights to a Wireless Robot!

During the relatively short and time—consuming first few weeks of class, we will be working to get you familiar with a number of different platforms and tools. Those include using and developing for the Arduino environment (an easy-to-use and low-end platform), Linux device driver writing, PCB design, and working with multiple embedded platforms at once.

This lab will focus on making a rapid prototype and learning the Arduino environment.

> *The goal of prototyping is to achieve system functionality identical to the desired system or as close to it as necessary in key areas to determine system performance and cost prior to committing to a final solution. What this means is that during prototyping we modify the design to get a decent approximation of the real system at (hopefully) a fraction of the cost. This approximation may be a goal in itself, for example for the purposes of evaluating a design. The design modifications (or shifts) performed during prototyping can take on different forms, each leading to a different form of prototyping.*[1]

1. Prelab

1. Read the lab from start to finish.
2. Read https://docs.arduino.cc/software/ide/
3. Read https://docs.arduino.cc/hardware/uno-rev3/
4. Get familiar with Arduino, its IDE, and the associated syntax. Then take a look at **File>Examples** to get a sense of what has already been done.

Answer the following questions:

Q1. In Arduino-speak, what is a shield? Identify at least two shields, where you can get them from and how much they cost.

Q2. What processor family does Arduino primarily use? Name another programming environment that is commonly used to program that family. What are the pros and cons of Arduino compared to that environment?

Q3. In Arduino-speak, what is a "sketch"?

Q4. Look over https://docs.arduino.cc/learn/. Much of it is pretty basic.

a. What does **`pinMode()`** do and what are the options associated with it?

b. How is pin 13 different than most?

Q5. Look at https://www.arduino.cc/reference/en/

a. What does **`delay()`** do?

b. What does **`analogRead()`** do?

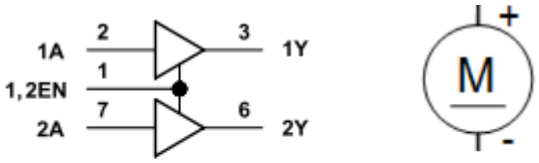
[1] Taken from http://www.signum.com/Signum.htm?p=art02.htm with only minor changes, though the ideas expressed are pretty standard.

Q6. Look at the H-bridge model L293D pin out (find it online).

a. Draw a diagram using the figures on the right showing how you would connect this H-bridge to a DC motor.

b. Specify the pins you would need to set on the h-bridge to move clockwise or counter-clockwise.

c. Explain how you would slow the motor down using PWM. Where would you apply the PWM signal?



Q7. Consider the following Python code. [2]

a. What does the **`import`** keyword do? Why do we import **`msvcrt`**?

b. What do you suppose **`serial.Serial()`**, **`ser.write()`** and **`msvcrt.getch()`** do? Be sure to address their arguments.

c. Briefly describe what this code does.

```
import msvcrt
import serial
ser = serial.Serial('COM38', 9600)
while 1:
      # Poll keyboard
      if msvcrt.kbhit():
            key = msvcrt.getch()
            if key == b'f':
                  ser.write(str.encode('C21FE'))
            elif key == b's':
                  ser.write(str.encode('C21SE'))
```

[2] Answering this question should take only 5 to 10 minutes. We just want you to have looked at the code and be aware that Python will be showing up. Also, be aware that kbhit doesn't work from bash or the IDLE program.

2. Inlab

The purpose of this lab is to gain a basic familiarity with the Arduino environment. Arduinos are a standard Atmel microcontroller with a *really* simplified programming environment. You should find that doing basic prototyping is really straightforward in this environment. In fact, Arduinos are really targeted toward non-programmers. This environment is important for you, a professional embedded engineer, to learn for three reasons:

- It is a powerful tool for quickly prototyping devices and getting basic interfacing working. There is a massive amount of device support out there and this can be a really useful way to either get a quick-and-dirty device created for someone or just for getting a proof-of-concept created.
- It provides some really solid interfaces so that people who don't understand the hardware can none-the-less use it. This can provide you with something to aspire to in your own designs for hardware interfaces.
- When others want to do basic embedded systems work without the needed background to do the job "right", you can point them to the Arduino environment and perhaps provide a bit of support.

In order to acquire that background, you will start by writing the "hello world" of embedded systems: getting a light blinking. We'll then move quickly to reading analog input, getting wireless communication working, controlling an H-bridge and getting a wireless vehicle up and running. All in this one lab! You might find https://www.arduino.cc/reference/en/ useful.

Note: some of these sections are significantly more guided than others. We have tried to find the right balance of having you searching and reading on your own while providing enough information that you aren't likely to waste your time in trivialities. So some sections (such as configuring your Bluetooth chips) are very directed (we don't want you wasting time figuring out their UI), while others more-or-less consist of "just go do it".

Part 1—Das Blinkenlights[3]

Let's get familiar with the Arduino programming environment. Open Arduino by double clicking the "infinity symbol." Dismiss (Cancel/No) all the Windows Security dialogs. This should bring you to a blank "sketch". We are going to make a simple little sketch for blinking the LED connected to pin 13. **Note that we won't need any includes or defines for this part; you only need to define a setup and loop function.** This is because the Arduino IDE transforms your sketch into a valid C++ program by adding the following lines of code:

```
#include "WProgram.h" // Definitions relevant to Arduino

void main() {
      setup();
      while(1) {
            loop();
      }
}
```

[3] See http://en.wikipedia.org/wiki/Blinkenlights if you are curious why this section is so named.

In setup, use pinMode (which you read about in the prelab) to make pin 13 an output. In `loop()`, use digitalWrite() and delay() to cause the LED to blink at 1 Hz with a 50% duty cycle. Note that both setup() and loop() should be declared as "void".

Now that the code is written, we need to specify which board we are using. Connect the Arduino via the USB port. Select the 'Select Board' dropdown and find the COM port used by the Arduino port (might require you to go to Window's Device Manager which you can search for in the start menu). Search for the 'Arduino Uno' if the board name does not automatically populate. Check to see if the code compiles. Press the check mark icon named "Verify." This will check for errors and compile your code. After the program compiles successfully, we still need to upload the program to the board. Then click the second icon on the upload icon . If you get an error, check to be sure you've selected the correct device and port. Verify that the Arduino is connected to the computer by plugging and unplugging while monitoring Device Manager->Ports (Com & LPT). If you cannot select a different COM port and cannot program your board, you may need to quit Arduino and restart it **with the board unplugged**. If that does not work, please power cycle your computer. You've written your first Arduino program!

G1. Have your GSI check that your code is flashing the LED at 1Hz.

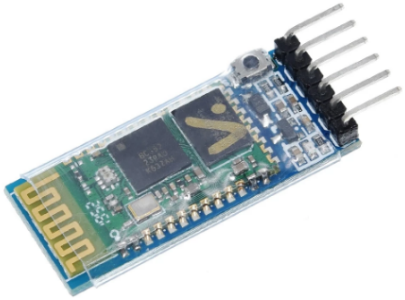
Using a breadboard, connect four digital outputs to four LEDs so you can light each of them individually. Include a ~1 KOhm resistor in series with each LED. Also make sure to share the Arduino's ground with the LED's. To recap, from a digital pin you will connect a resistor, the resistor will connect to an LED, and the LED will be connected to ground on the Arduino board. Write a quick test sketch to be sure you can individually control each LED. One interesting thing about the Arduino environment is that you *do* have access to the processor and its low-level interfaces just as if you were coding in C (because you are). Check out https://docs.arduino.cc/retired/hacking/software/PortManipulation/ and use the DDRx and PORTx to change the LEDs all at once.

For the last step of this section, hook up a potentiometer (pot) which we've supplied such that the middle pin of the pot is connected to one of the analog inputs found on the board while the other two pins on the pot are connected to 0V and 5V. Look through the Arduino documentation about how to work with analog inputs and make it so the number of LEDs lit is about proportional to how far the pot is turned (or, if you prefer, the MSBs of the value are displayed on the LEDs). For the analog input you do not need to use the DDRx and PORTx interface.

G2. Have your GSI check that your LEDs are properly controlled by the potentiometer. Show the GSI that you are using DDRx and PORTx to control the LEDs.

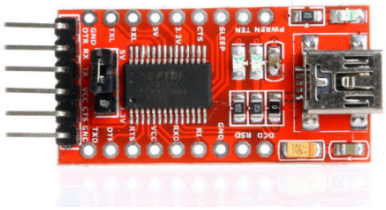
Part 2—BLE Wireless

For this part of the lab, you will create a wireless serial link (UART) between the workstation and the Arduino with a pair of low energy Bluetooth radio modules.



x2

To establish a UART connection from the workstation or your laptop, we will also need an FTDI USB to UART module.



If you have an FTDI board with a selectable 3.3/5.0-volt jumper, make sure it is set to 5 volts.

Arduino to BLE Module Connection

In this part we will construct and verify the Arduino to BLE module 1 connection. Make the following connections on your solderless protoboard. Check the silkscreen labels on each module to be sure of the pinouts. Here we are connecting between the workstation/laptop and the BLE module using the USB to UART converter on the Arduino. We are not yet connecting between the main microcontroller on the Arduino and the BLE module. This might be confusing at the moment, but this should be something you understand by the end of the lab.

Arduino, **TX** (pin 1)→ BLE Module 1 **TX**
Arduino, **RX** (pin 0) → BLE Module 1 **RX**

Arduino, 5v → BLE 3.6v – 6v
Arduino, GND→ BLE GND

The BLE module red LED should be blinking at several times a second if it is properly powered and working.

We will need to setup the module using AT commands over a UART serial link.

The module has to be put into the AT command mode first. To do this, power cycle the module while simultaneously holding down the little button in the lower corner of the BLE module. The simplest way

to power cycle the module is to pull the jumper wire connected to 5 volts in and out. If you were successful, the red led should now be blink much slower at about once a second.

Now let's see if we can send some AT commands to the module. Connect the USB cable to your Arduino, open Sketch and then open up the Tools→Serial Monitor. The baud rate will need to be set to 38400. This setting is in the lower right corner of the terminal window. The commands must be followed by the codes NL (/n) new line and CR (/r) carriage return. This sequence can be automatically appended with the setting in the lower right window.

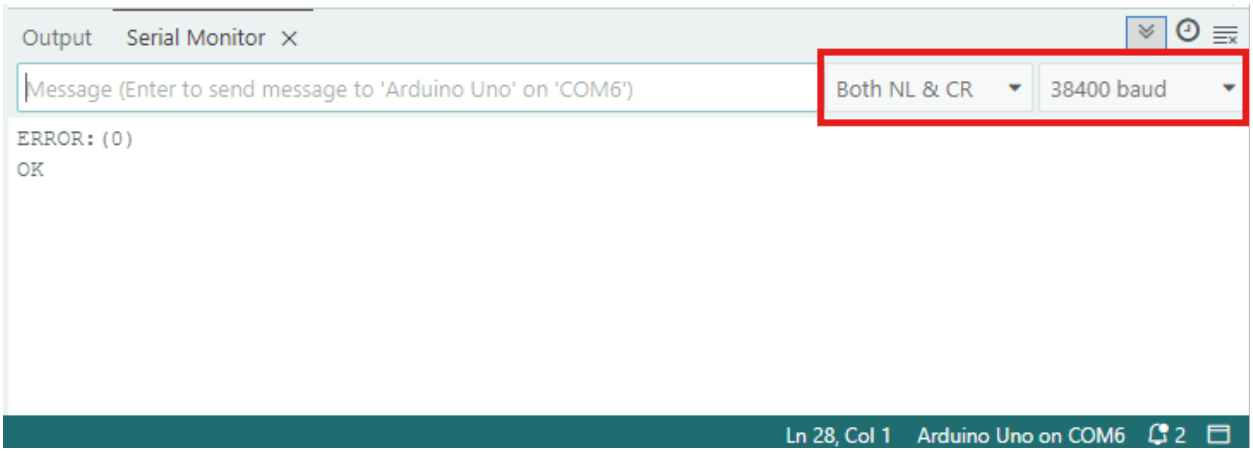
Send the letters "AT" until you get

```
OK
```

The first response may sometimes be

```
ERROR:(0)
```

Because of some initial error state of the module. This should clear up if you send "AT" again.



Workstation to BLE Module Connection

We are essentially going to do the same thing we did to setup a connection with the Arduino and BLE module, but with this time we will connect the BLE module to our workstation via the FTDI UART to USB module. Start by connecting your FTDI chip to the BLE module as follows.

FTDI, **TX** → BLE Module 2 **RX**
FTDI, **RX** → BLE Module 2 **TX**
FTDI, VCC → BLE 3.6v – 6v
FTDI, GND→ BLE GND

Connect the USB A cable to the FTDI chip to provide power.

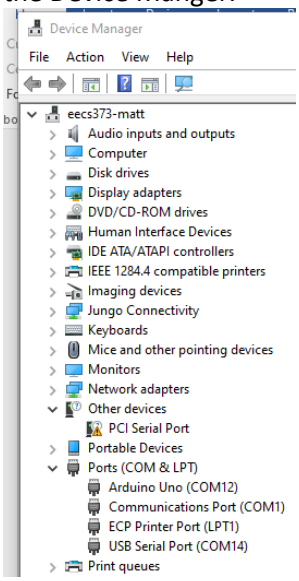
The BLE module red LED should be blinking at several times a second if it is properly powered and working.

We will need to setup the module using AT commands over a UART serial link.

The module has to be put into the AT command mode first. To do this, hold the little button down in the lower corner and power cycle the module. It is easiest to pull the jumper wire connected to 5 volts in and out. If you were successful, the red led should now be blink much slower at about once a second

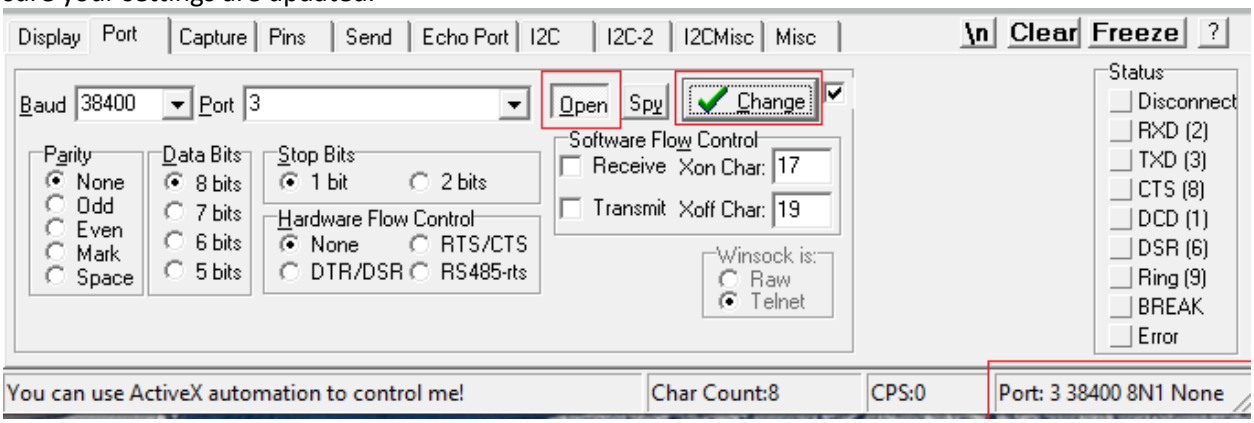
Now we will use a terminal program called `Realterm` to send and AT command to the module.

Open Realterm on the workstation. You will need to determine which com port the FTDI module is connected to. You can determine the port number in Windows 10 by looking under the Ports listing in the Device Manger.

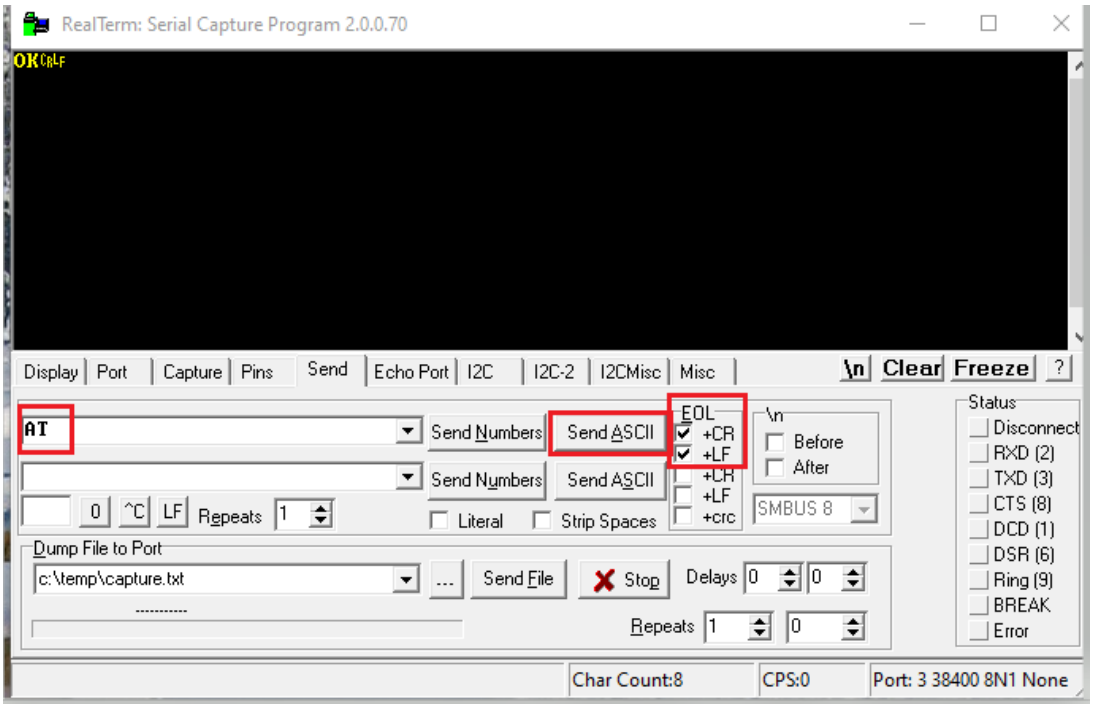


Usually it is the USB Serial Port. If you disconnect the FTDI chip the dependent port will be removed from this listing.

Once you have the port number you can specify it in Realterm in the Port tab. You will also need to set the Baud rate to 38400. Make sure the Open button is depressed and use the Change button to make sure your settings are updated.



Next select the Send tab, enter the AT command in the send field including /n and /r. Send ASCII and you should see a OK response. As before, the first response may sometimes be an error, but it should reply with an OK after a second attempt.



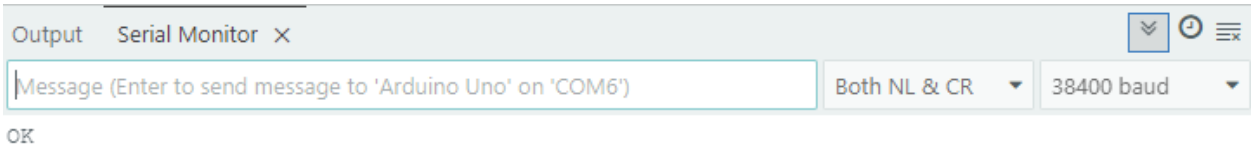
Pairing the BLE Modules

Now that we can communicate with the modules in the AT command mode we can configure the modules for 2 way serial communications.

Slave or Peripheral Configuration

BLE modules are paired in a slave/master or peripheral/center combination. The BLE module connected to the Arduino will be setup as a slave or peripheral module and the BLE module connected to the workstation will be the master.

Send AT+ORGL to the module connected to the Arduino. This will restore the module to its original state by resetting all configurations that were previously set. After sending this command, the module will respond with OK and then reset. We will have to put the module back into AT command mode by holding the button and cycling power.



Send AT+ROLE to the module connected to the Arduino. It should indicate the ROLE is 0 or peripheral which happens to be the default setting. If that command errors out (“ERROR:[0]”), try adding a ‘?’ after the command: “AT+ROLE?”. If this fixes it, you will need to use this syntax for all queries on this module.

```
Message (Enter to send message to 'Arduino Uno' on 'COM6')
OK
+ROLE:0
OK
```

We will need the physical address of this module to pair this module with the master/center module. Send AT+ADDR to get the address. You should see something like this.

```
Message (Enter to send message to 'Arduino Uno' on 'COM6')
OK
+ROLE:0
OK
+ADDR:98D3:31:F98CA6
OK
```

Keep in mind that the full address will always be in the form:

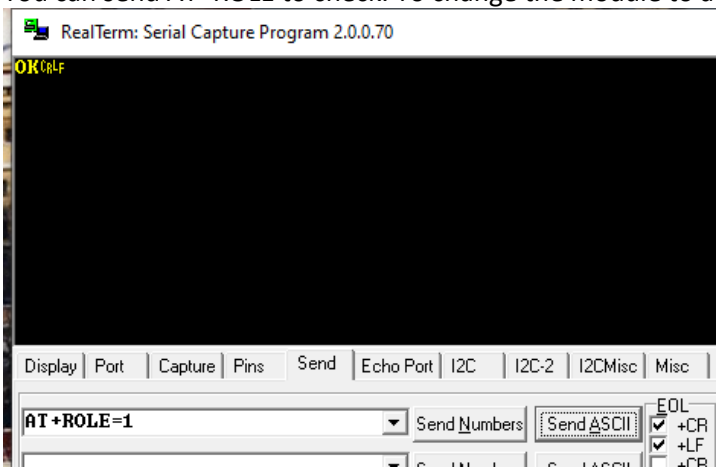
4-digits : 2-digits : 6-digits
XXXX:XX:XXXXXX

If the address shown by the AT+ADDR command has less digits, fill in the missing digits by adding leading zeroes as necessary (e.g. ef:34:e53 → 00ef:34:000e53).

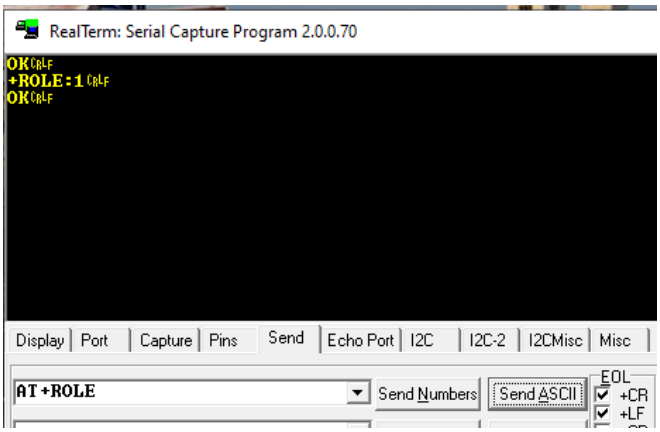
Save the address for the next part.

Master or Center Module Configuration

We will set the BLE module connected to the workstation or personal computer as the center module. We should reset this module as well by sending AT+ORGL. Afterwards we will need to put the module back into AT command mode. By default, the module should be configured as a peripheral or ROLE=0. You can send AT+ROLE to check. To change the module to a center role, send AT+ROLE=1.



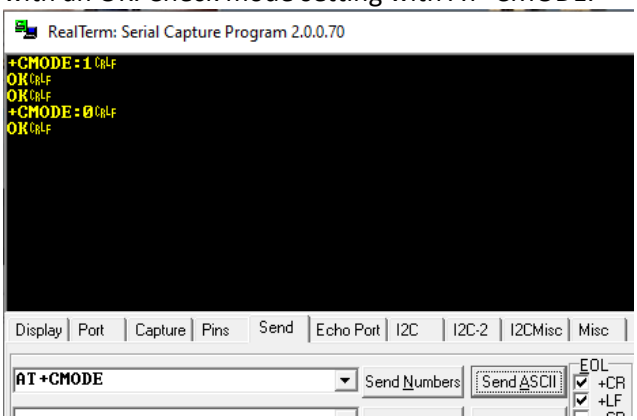
You should see an OK response. Double-check that your module is still in AT Mode, as sometimes this configuration change can reset the module. Once you have the module back into AT Mode, check that the role has changed by sending AT+ROLE



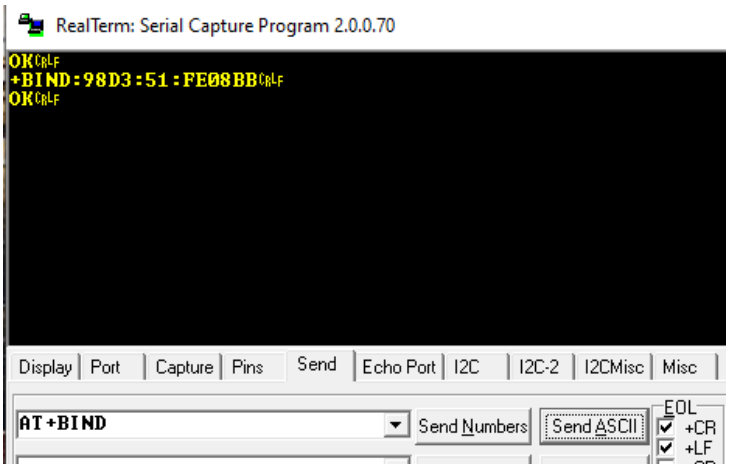
Paring the Modules

Now we are ready to pair the modules.

First we need to check the connection mode of the center/master module. Send the AT command AT+CMODE. The default is CMODE 1. This means the center module will connect to any BLE peripheral. We want to just connect our peripheral module. To do this we need to set CMODE to zero on the center/master module. Send the AT command AT+CMODE=0 to the center module. It should respond with an OK. Check mode setting with AT+CMODE.



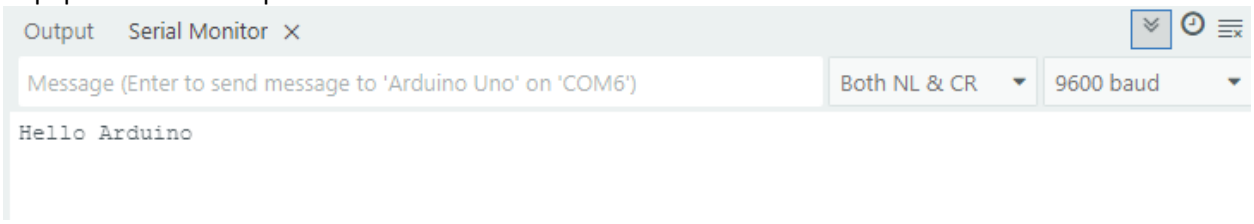
Next we need to tell the center module which peripheral to connect to. To do this we give it the physical address with the following AT command AT+BIND=physical address. In this example the command looks like AT+BIND=98d3,51,fe08bb. Notice the colons were replaced with commas. You should get an OK response from the center module. Send AT+BIND and the center module should respond with the address with colons replacing the commas.

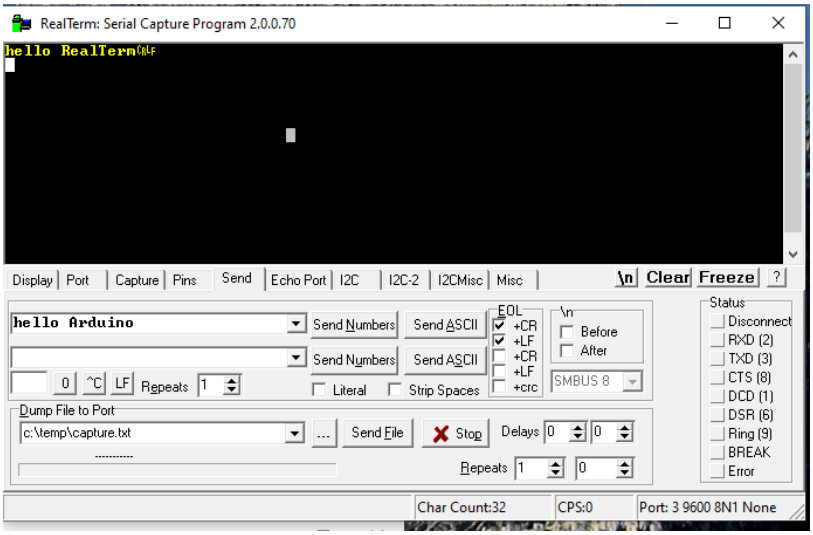


Testing the 2-way Serial Communications
Power cycle both modules. The LEDs will blink rapidly at first. Then they will blink in bursts of 2 blinks about every 2 seconds indicating the 2 modules are paired.

Next change the baud rate for the Arduino serial monitor and the RealTerm monitor to 9600 baud. Try sending a text message from both sides and you should see a successful transmission.

The modules are now paired. You do not have to repeat the configuration process. It can help the label the modules as master/center and slave/peripheral in case you remove them from the board with a slip of paper beneath the plastic sheath.





An Arduino Project

Now that we have a wireless serial interface, we will write an Arduino program to communicate wirelessly using the Arduino Serial functions. You may want to look at https://www.arduino.cc/en/Reference/Serial as a reference in using the Serial functions.

The program is required to echo each character that is sent with an "X" and a newline. So if you send "AB" from the workstation to the Arduino you should get back "AX" and then "BX" as a return from the Arduino back to the workstation. Send data via the master BLE terminal program (RealTerm) to the peripheral BLE connected to the Arduino setup wirelessly.

NOTE: The Arduino Sketch shares the Arduino TX RX pins so they need to be disconnect from the BLE peripheral module when you are uploading your program to the Arduino.

While setting up the BLE Module connected to the Arduino (The Peripheral BLE), RX must be connected to RX, TX to TX (as described above), to allow the Computer Serial Monitor to communicate with the BLE device. While the Arduino is running code, RX must be connected to TX, to allow the Arduino to communicate with the BLE Module.

GS. Demonstrate your working code. Explain the flow of data to the GSI.

Part 3—Wireless motor controller

Open up the terminal (right click in windows file explorer) in a location where you would like to save the files for the next part of the lab (i.e. N:\ drive). Clone the Lab1 repository:

```
git clone https://github.com/EECS473/Lab1.git
```

Open up the motor.ino file in the 'motor' directory. You can also find the code on the lab page, the "motor.txt" file. Disconnect the Bluetooth serial connections, compile and upload this program, then reconnect the Bluetooth. Use the BLE terminal program to build and send packets. Look over the code to find a sequence of characters that results in the response "moving forward". You may want to look at https://www.arduino.cc/en/Reference/Serial.

Q8. Examine the code.

a. What sequence of characters results in the response "FORWARD"? (There is an explanation below that will help, but try to understand the code first.)
b. Consider the packet xC21BE, where "x" is any arbitrary character. What will be the possible outputs? Under what circumstance (what value of x) if any will it not be recognized as a legal reverse packet?
c. What pins are used to control which motor?

Now go back to your terminal program and if possible, find a way to map a single key to send a packet (some terminal programs may not have this functionality and you may have to type in each packet. In that case you may want to find a better terminal program…). Confirm operation of the motor control program by sending the following commands: F->Forward, B->Backward, R->Right, L->Left, and S->Stop. But these commands are structured with the follow syntax. C21xE. C->Start of Packet, 2->Two byte command, 1->Motor Command, x->One of the above commands, E->End of Packet. Normally the Beginning and End of Packet (SoP/EoP) signifiers are non-ASCII bytes, which means bytes that don't translate into an ASCII letter, so that you can send arbitrary ASCII characters (think about trying to send a string for example).

G4. Demonstrate the working motor-control code to your GSI. Show what the motor outputs look like on an oscilloscope or logic analyzer.

Q9. What does the output of pin 5 look like on the scope when it is at "SPEED"? Notice that it is labeled as an analog output in the code and as PWM on the board. Explain both labels.

Part 4—Self-guided H-bridge

In this class there we are often expecting you to be able to figure things out largely on your own. That said, we also want to be sure we don't destroy our equipment. So we are asking you to wire everything up, but not to power your system until the GSI has looked over your setup.
We want you to use an H-bridge model L293D or SN754410 (they are equivalent) to drive a motor so that your wireless control system determines what direction the motor turns (or if it idles). Hook up the 5V output of the Arduino board as Vcc1 and **use the battery as Vcc2**. Get a DC motor to turn in different directions depending on the signal sent. Modify the code so that the idle output state (where no outputs are enabled) can be reached, you should assign a new command to reach this state ('I').

G5. Demonstrate your working motor (complete with an idle {'I'} option) to your GSI.

Part 5—Robot building time

Previously, you controlled a motor over a wireless link using an H-bridge with power coming from the lab station's power supply. We are now going to move the Arduino and its BLE module onto a robot. You will be driving both the Arduino and the motors off of the USB battery. Notice the robot is metal and conductive. To avoid damage, add insulating material between the chassis and the exposed solder joints on the bottom of the Arduino. The H-bridge circuit should also be insulated.

G6. Show your GSI you can control the robot using the same motor controller you used in Part 4. You might want to slow down the motors so you have time to type the fairly long commands into the terminal.

Note: We do not have enough batteries for everyone, so please return to the charging station after each lab. Try to keep batteries charged.

Part 6—Python control

```
import msvcrt
import serial
ser = serial.Serial('COM38', 9600)
while 1:
	# Poll keyboard
	if msvcrt.kbhit():
		key = msvcrt.getch()
		if key == b'f':
			ser.write(str.encode('C21FE'))
		elif key == b's':
			ser.write(str.encode('C21SE'))
```

Now you are going to combine the Python code written above, and your robot. In the next lab, we will extend this robot control interface substantially, but for now we just want a way to streamline sending commands. This python file should have already been downloaded when you ran git clone above. You can edit with notepad++ by running `start notepad++ "robot-control.py"` in the terminal. Add all the commands available in your Arduino sketch.

G7. Show your GSI that your python script works and allows for basic control of your robot. The robot should be moving around on the ground.

3. Post-lab

This post-lab has a few additional questions that generally require more thought and longer answers than the in-lab questions. Please submit the in-lab and post-lab questions as a single document. Both group members, working together, should do these together and submit just one document.

Wireless issues

Whenever we are working with wireless (radio) communication, sometimes it can be important to be aware of low-level details, including error correction/detection and bandwidth actually used. We

may need to understand these things due to bandwidth limitations or concerns about noise. For noise, we are mainly worried about three things:

- Our packet seeing interference from an external source and becoming corrupt.
- Random (or not so random…) noise that the receiver perceives as a valid packet.
- Random noise which happens before or after our packet causing our data to be lost or miscommunicated

Q10. Assume for now that our transmitter is just sending what we tell it to send. So if we send C21FE, only those characters (as ASCII characters) are sent out by the radio.

a. Say the sender sends a legal motor control packet for the scheme you used in lab. Could a single bit-flip (that is one bit of the message changing from a 0 to a 1 or a 1 to a 0) cause a different legal command (other than idle) to occur? If so, provide an example. If not, explain why not.

b. Say a random ASCII character is received right before your data. Rewrite the code so that all "xC21FE" packets will parse the code correctly, no matter the value of "x". Submit a diff of your code and the original code (using the Linux/Unix "diff" ideally).

Q11. In reality, most communication schemes send a fair bit more data, often including a start delimiter and checksum. Look at the example found at https://web.archive.org/web/20140820153750/http://www.digi.com/support/kbase/kbaseresultdetl?id=2206.

a. Why do you think there isn't an End Delimiter?

b. Calculate what the checksum would be for the example if the data packet were only 2 bytes: 0x0F and 0x1F.

c. If you send 512 bytes one byte at a time, how many bytes (total) would need to be transmitted? What if you sent all 512 at once?

d. If you are streaming a massive amount of data using the same generic frames, what is the % overhead of the protocol? (Think about what limits packet sizes.)

Arduino

Q12. Do a bit of research and explain the interrupt scheme used by the processor you are using. You may freely use web resources, but please cite your sources clearly. Your answer should include:

a. How are different interrupts distinguished?

b. How do you disable interrupts?

c. How do you clear interrupts?

d. Consider the sketch found at http://gonium.net/md/2006/12/20/handling-external-interrupts-with-arduino/. Explain what is going on in your own words.

Q13. There are many advantages and disadvantages to using Arduino and its development environment. Where do you think you'd use it? Where are you fairly certain you wouldn't use it?