



Exported for Brian O'Connell on Tue, 27 May 2025 19:31:50 GMT

# Programming 7 Pre-Lab: MATLAB/Arduino Serial Communication

This pre-lab lesson will get into establishing communication between MATLAB and your Arduino. You will need your Arduino kit for this Lab.

I had intended to use the long weekend to expand this a bit more (Adding more walkthrough videos and more code reference throughout to make it a bit more streamlined). Since I was sick, I've moved that to Friday afternoon. I'm releasing this now though for those that want to work on it before then.

## Serial Communication

You are going to have to collect data using a variety of testing apparatuses that will not necessarily have an embedded user interface or native means to

extract the data it collects. Most will send information out and expect you to capture it on your or some dedicated PC that's set up for that purpose. This is where serial communication comes in.

The connection of these components creates a system and that system must communicate with the various components. In order for that information swap to occur, they must share a common communication protocol. That's the essence of serial communication and any designer looking to



CC BY-SA 2.0 DEED  
Attribution-ShareAlike 2.0  
Generic [Bureau of Reclamation \(U.S. Department of the Interior\)](#).

work with embedded systems, be it robotics, automated systems, medical devices, real-time monitoring/testing technologies, must have a good understanding of that. So let's get you at least started here with some of the basics, enough to do some interesting stuff here and in your projects as well.

There are numerous existing standard and protocols for [serial communication](#), ranging from very simple (which is what we'll be sticking with here) to extremely complex. In general terms though, there are 2 broad types: **synchronous** and **asynchronous**.

## Synchronous Transmission

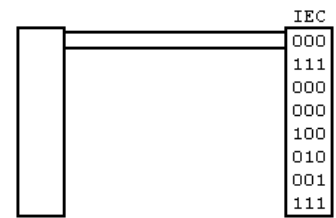
Synchronous transmission is the more advanced and complex category. In this form, the communication between the sender and receiver is fully-synced, meaning they're functioning on the same clock. This leads to no time-gap between data as well as . It's extremely efficient and much more reliable than asynchronous transmission. With that shared clock, large amount of data can be sent. It enables fast, reliable, gapless, real-time communication. It is much more complicated and therefore costly though, being very dependent on finely tuned systems to ensure all clocks are precisely synced. Telephone conversations, Video conferencing, & Online gaming required the use of synchronous transmission.

We are not going to work with something as complex as a synchronous transmission protocol today. I'm only informing you about this just to have a broad understanding. If you look for resources online about serial communication, I don't want you accidentally falling down a complex rabbit hole trying to achieve synchronous communication between your components. You may end up needing to use SPI or I2C in your project depending on what capabilities you incorporate and therefore what components you use. If you want to use RFIDs, color or gesture sensing, or some environmental sensors, then your system will have some synchronous communication between that component and your Arduino BUT you won't have to establish it. They should have libraries for that and tutorials on how to use them so that communication will already be handled for you. Again, just letting you know in case you come across synchronous transmission when researching some of those components.

## Asynchronous Transmission

We're going to be working with Asynchronous transmissions. This means that we'll be sending data

between our system components in the form of bytes or characters, 1 at a time. You can send larger amounts of information via this method but it's still only send 1 byte at a time but it does get queued up in the order received.



This method is slower but more economical. There are several limitations though. There's not a constant feed of data in this method; it's essentially a device sending out messages and you don't instantly know if the other devices have received it or if the message arrived in full. If synchronous is a phone conversation, then asynchronous is sending letters through the post office. You can't be 100% certain when it will arrive and be read. We'll work on some ways to plan for and accommodate these shortcomings in our examples.

### Question 1

What is the main difference between synchronous and asynchronous transmission in serial communication?

A

Synchronous transmission is slower but more economical, while asynchronous transmission is faster and reliable.

B

Asynchronous transmission uses less data while synchronous transmission uses a large amount of data.

C

Asynchronous transmission sends data one byte at a time and isn't certain of the exact arrival time, while synchronous transmission is fully-synced and more reliable.

D

Synchronous transmission functions on a separate clock, while asynchronous transmission functions on the same clock.

 HINT



 EXPLANATION



 Show Responses

 Show Answer

# Serial Communication Commands

Due to the nature of this, I'm going to start a little more boring than usual with just some basic overview of the commands used for Serial Communication in both platforms. The nature of these require some coordination between MATLAB and Arduino scripts and for them to be used in concert with one another. That makes 1 off examples for each command difficult since they'll, by necessity, also use several other new concepts and commands. So let's just get through this understanding portion and then we can do some bigger activities to learn about these in practice rather than having example scripts and command lines for each.

## Serial Communication in Arduino

Let's start with Arduino since you have some experience with using some of the commands already. The Serial Monitor in Arduino does just what the name implies, it monitors the connection to the Arduino for any serial communication then displays it. Let's start by going through the commands.

Arduino has [several commands](#) for handling basic serial communication through the USB connection. Just a reminder, the transmission (TX) and receiving (RX) wires of the USB are also connected to pins 0 and 1. Using those pins will disrupt serial communication. The commands we'll be using are all part of the Serial library, which is built in to the Arduino environment.

The first few you are familiar with. I'm still going to go over them but just in this context of what we'll use them for in this lab.

### **Serial.begin(speed)**

The [begin command](#) opens the port up for serial communication. The value of speed sets the baud rate, the data transmission rate in bits per second. The most common is 9600. You have to make sure that components communicating via Serial Communication are using the same baud rate.

You can see what communicating at different speeds can cause if you go into the Arduino IDE Serial Monitor and change the baud rate it's listening at (The drop down all the way on the right). You need an Arduino connected that's printing to the serial monitor to see the effect. If you change the rate, the messages start to become gibberish.

### **Serial.print(message)**

We'll use [print](#) the same way as we have before. Up until now we've been using it so we can monitor activity in our Arduino through the serial monitor. Now we'll use it to send information meant for MATLAB. MATLAB will essentially become our serial monitor, receiving the printed messages from the Arduino by listening over the USB port for them.

There are some new elements to consider though. Serial communication between these non-human devices is very dependent on adhering to certain formats. Each component is expecting certain information and will be set up to only handle information sent in a certain format. It won't be able to interpret any ambiguities or major deviations. We'll talk more about that later when going over creating serial communication protocols, but for now know that format really matters, down to the number of bytes sent and what data types those bytes represent. You can have greater control if you use some of the format control options for print. We'll be fine using simple messages for now but if your communications get more complicated, you'll want to check out some of the more efficient format controls [Serial.print\(\)](#) offers. The one everyone will need to remember is that **Serial.println()** adds an extra newline character to its output that you will have to account for in your communications.

Now for the new commands:

## Serial.available()

The [Serial.available\(\)](#) command returns the number of characters available to read from the serial port. This is commonly used to check if there is any information available in the **buffer**. Buffer is a colloquial term for where serial communication messages are stored while they wait to be read. The buffer size of an Arduino Uno is 64 bytes so it can not accommodate a message greater than 64 characters long.

**Serial.available()** is commonly used as a means for checking if there is a message waiting. In the forthcoming demo, you'll see it used in 2 ways. When the buffer is empty, *Serial.available()* returns the integer 0, which can be interpreted as a boolean false. When the buffer has anything in it, *Serial.available()* returns an unsigned int from 1 to 64 depending on the amount of info. That can be interpreted as a boolean true. We will use that as the conditional in our control statements to determine whether or not to access the buffer.

## Serial.read()

The [Serial.read\(\)](#) command accesses the buffer and retrieves the first byte in the incoming stream. It then returns that byte of information. The act of retrieving it, removes it from the buffer. How that byte is interpreted is based on how it's stored. If you run the command `char IncomingByte = Serial.read()` then it's stored as a character since it got assigned to a variable created as a

character. If you run the command `int IncomingByte = Serial.read()` then it's stored as a integer since it got assigned to a variable created as an integer.

This may be easier seen in action to understand:

## Video

**Please visit the textbook on a web or mobile device to view video content.**

If you would like to run this example yourself, you can get the code for it here:

[https://raw.githubusercontent.com/boconn7782/CourseCode/main/P7L\\_Demo1.ino](https://raw.githubusercontent.com/boconn7782/CourseCode/main/P7L_Demo1.ino)

To run, you just need to upload it to an Arduino, type messages into the Serial Monitor then send those to the Arduino to see the response.

`Serial.available()` and `Serial.read()` will be what you use most of the time for simple serial communication. We're going to show you a few more, some of which we'll use later just to showcase them, that can be useful for handling

## Serial.parseInt()

The [Serial.parseInt\(\)](#) command reads the buffer until it finds a valid integer then returns that integer. Everything it reads up until that integer gets eliminated from the buffer as its accessed. So if you have the message `'I ate 314 tacos'` in your buffer and then run the command:

```
int dataIn = Serial.parseInt();
```

The variable `dataIn` contains the value `314` now and the buffer is left with `' tacos'` in it. If your buffer had `'I ate too many tacos'` in it, so it contains no valid integers, then `dataIn` is set to `0` and the content of the buffer remains the same. If the message contains multiple integers, like `'I ate 314 tacos and 147 burritos'`, then the variable `dataIn` contains the value `314` now and the buffer is left with `' tacos and 147 burritos'` in it. Even though it contains

multiple integers, it will stop once it starts identifying non-integers in the stream so it doesn't get to the next integer.

This gets way overexplained here for those who want to dive deeper:

<https://www.programmingelectronics.com/parseint/> Also, to give credit where it is due, I got the 314 tacos example from them.

## Serial.parseFloat()

The [Serial.parseFloat\(\)](#) command functions similarly to `parseInt()` except it will look for a float instead.

As an aside, I've found this command to be more temperamental because it not only responds to numbers in the streams but periods as well. If you need to use it, adding buffer characters around any floats sent or received can be helpful.

## Serial.readString()

The [Serial.readString\(\)](#) command takes the entire contents of the buffer and returns it as a string. This is a pretty straightforward and usually more intuitive command. It can be useful enough that I will note it here but with how we'll be setting up our messaging, if you emulate that for your projects you'll likely not use this. It is useful when you're sending information that will be displayed by a component attached to the Arduino or will serve as a message passed along by the Arduino.

## Serial.setTimeout(time)

The [Serial.setTimeout\(\)](#) command applies to `Serial.parseInt()`, `Serial.parseFloat()`, and `Serial.readString()`. Each of those will access the stream until it finds what it is looking for, runs out of information in the buffer, or runs out of time. Running `Serial.setTimeout(time)` sets that time limit based on the value of time given in milliseconds. The default is 1000 ms. That can be a long time in terms of an Arduino script so, depending on fast you want your script to run, you can use this command to speed up the processes or even slow it down if it's taking a while for the complete messages to transmit to your Arduino.

### Question 2

What does the `Serial.available()` command do in the context of serial communication with non-human devices?


- A It checks if there is any accesible data in the buffer.
- B It retrieves the first byte in the incoming stream from the buffer.
- C It reads the buffer until it finds a valid integer then returns that integer.
- D It takes the entire contents of the buffer and returns it as a string.

 HINT



 EXPLANATION



 Show Responses

 Show Answer

### Question 3

Which pins on your Arduino should you avoid using?

Select All That Apply



- A A0
- B A1
- C A5
- D 0
- E 1
- F 7
- G 8
- H 10



I	11
---	----

 HINT



 EXPLANATION



Show Responses



Show Answer

## Serial Communication in MATLAB

MATLAB's communication across serial ports functions the same way as in Arduino. You need to open up a connection to a serial port and then you can either receive information along that line or transmit information along that line. The information is still sent and received byte by byte. MATLAB just has a much bigger buffer since it's on your laptop so it can receive and hold onto a lot more information before it needs to be processed and cleared.

### serialportlist

The [serialportlist](#) command returns a list of all actively connected serial devices. This will include virtual devices and bluetooth connections as well. When we get into actual implementation, we'll go over how to identify which one is your Arduino.

### serialport(port, baudrate)

The [serialport](#) command makes the connection to that serial device. It is not a direct 2-way connection, it's just connecting to the line that device is on so that MATLAB can transmit along that line and listen for data that comes in along it. The full command is **ArduinoObj = serialport(port, baudrate)** where **port** is the name of the serial port that you got from **serialportlist** and **baudrate** is the communication speed, which will typically be **9600**. The command itself stores that connection and associated information as an object which you need to save into a variable for future reference. I'm just arbitrarily referring to it as **ArduinoObj** here but it can be named whatever you want like any other variable.

### readline(device)

The [readline](#) command will connect to the buffer for the specified serial port and read in the next line as a string. It will read until it hits the terminator character, which is typically a newline

character. Once it reads that line, it's removed from the buffer, leaving the rest of the buffer data alone. The full command is `response = readline(device)` where `device` is the object created when you used the `serialport()` command. The information read from the device would be stored in `response`. There is also a [read](#) command which gives you some finer control of how much information you access from the buffer as well as how you interpret it. We'll primarily use `readline()` then parse the information we want from the string.

## Writeline(device, data)

The [writeline](#) command will connect to the buffer for the specified serial port and transmit some data along that line followed by a terminator character, the default being a newline character. If that data exceeds the buffer of the connected device, MATLAB is unaware and how the excess is handled is dependent on the connected device. The full command is `writeline(device, data)` where `device` is the object created when you used the `serialport()` command. It connects to it then writes value of `data` as a string to that device, attaching a terminator character to the end of the string. There is also a [write](#) command which allows you to write the data as a specified data type without an extra character but the connected device will still have some say in the datatype it interprets it as. We'll primarily use `writeline()` since the Arduino stores everything as generic bytes and generally interprets it as characters.

## flush(device)

The command `flush` clears the buffer of the selected serial port. This is useful in our circumstances because the Arduino will continuously run, sending a lot of information, while we may be running slower on MATLAB. This command clears out all previous messages and then MATLAB will then read the next message which would be the most up to date. The full command is `flush(device)` where `device` is the object created when you used the `serialport()` command. Since this completely empties the buffer, any following command to read the buffer will wait for new information or time out if the wait is too long.

Here's a demo to showcase the importance of utilizing this command before reading information from the serial port buffer.



**Video**

Please visit the textbook on a web or mobile device to view video content.

And here's the code used for this:

[https://raw.githubusercontent.com/boconn7782/CourseCode/refs/heads/main/P7\\_FlushDemo.m](https://raw.githubusercontent.com/boconn7782/CourseCode/refs/heads/main/P7_FlushDemo.m)

## writeread()

We will not use this one in our examples but the [writeread\(\)](#) command is very useful. It does require some better coordination between system components. Using readline and writeline is like passing notes back and forth in class without concern for what's actually in the other notes and no concern with the timing of when those notes get to their recipients or you get a return note, just the knowledge that they will get there. If you're passing a note that says "Will you go to prom with me?" you'd rather get a direct and immediate reply rather than waiting and that's what writeread() does. The full command is `response = writeread(device, command)` where `device` is the object created when you used the `serialport()` command and `command` is the string you want sent to it. The information read from the device would be stored in `response`. The common expectation with this command is that whatever device you're writing to is expecting a command and will send specific data in response to that command. This does require the scripts to be more in sync timing wise and it will timeout if a response takes too long. It's effectiveness is circumstantial.

### Question 4

What is the purpose of the 'serialportlist' command in MATLAB?

?

- A It flushes the buffer of the selected serial port.
- B It transmits data along the serial port.
- C It returns a list of all actively connected serial devices.

D	It creates an object for a connection to a serial port.
---	---

 HINT



 EXPLANATION



Show Responses



Show Answer

### Question 5

When reading information from the serial connection MATLAB just needs to run `readline()` to get the latest line of data in the serial port buffer.



A	True
---	------

B	False
---	-------

 HINT



 EXPLANATION



Show Responses



Show Answer

## Establishing a Protocol

Before we begin our activities, we need to address a more conceptual step in serial communication. We need to make sure we have rules in place for the format of the messages that will be sent back and forth between MATLAB and Arduino. For more complex systems, this can become equally complex for this simple 2 component system we only have a few things we need to establish.

- When will messages be read?
- What is the message format?

## When read?

This is typically right at the beginning on the Arduino side. We should think of it as another sensor in that we want that information at the beginning of our script so we can use it before making any decisions or taking any actions. The Arduino runs so quickly, that we can just assume that every message in the buffer gets read fast enough that we don't have to much more concern in than that.

On the MATLAB side, this is more complex and circumstantial. In our first activity, we'll treat it the same as with the Arduino, have it read at the beginning of our loop. We will need to consider that our MATLAB script will likely run slower than the Arduino since we're incorporating human readable elements which necessitates slowing things down. We have ways of handling that, which we'll go over in the activity, to ensure we're reacting to the most up to date data sent. It's also not uncommon to have the read happen only as required, when you do something within MATLAB to trigger a read. It's handled similarly but just even more important to ensure you're not reacting to old information.

## What format?

This can be whatever you want but short and simple is the best. You want some leading indicator, typically a character or phrase, followed by data. The receiver can look for that indicator and then parse out the data that follows it. There should also be some agreed upon number of characters that this all takes place in so every message is always the same length or at least under some maximum length.

For instance, I could program one side to send 'A275B085' based on wanting sensor A reading a value of 275 and sensor B reading a value of 85. When the receiver gets that message, it can see the character A in the buffer and know the next in line is an integer regarding information from sensor A connected to the sender component. It can parse that out and save it then continue to read the rest of the message. When it see B, it does the same with the understanding that the next number applies to sensor B. It can't inately know these things or figure it out on its own. It has to be pre-established and factor into the programming for each component.

## Our Protocol

For these activities, let's agree that this is our protocol:

- New messages will be read as soon as they arrive

- Messages will be no longer than 12 bytes
- Messages will contain data that is preceded by a specific character or set of characters used as identifiers/labels for the following data
  - We are only sending data as integers for these examples
- Every message will end with a newline character ‘\n’ – which we will call our terminating character



### Question 6

[Show Correct Answer](#)

[Show Responses](#)

Arrange the following into a good order of operations for programming a system of components that will utilize serial communication.

<b>A</b>	Test the input/output of the more complex component
<b>B</b>	Test communication between the system
<b>C</b>	Program the simpler component
<b>D</b>	Establish or agree upon a communication protocol
<b>E</b>	Plan out your initial logic through a psuedocode or flow chart
<b>F</b>	Program the more complex component
<b>G</b>	Test the input/output of the simpler componenet

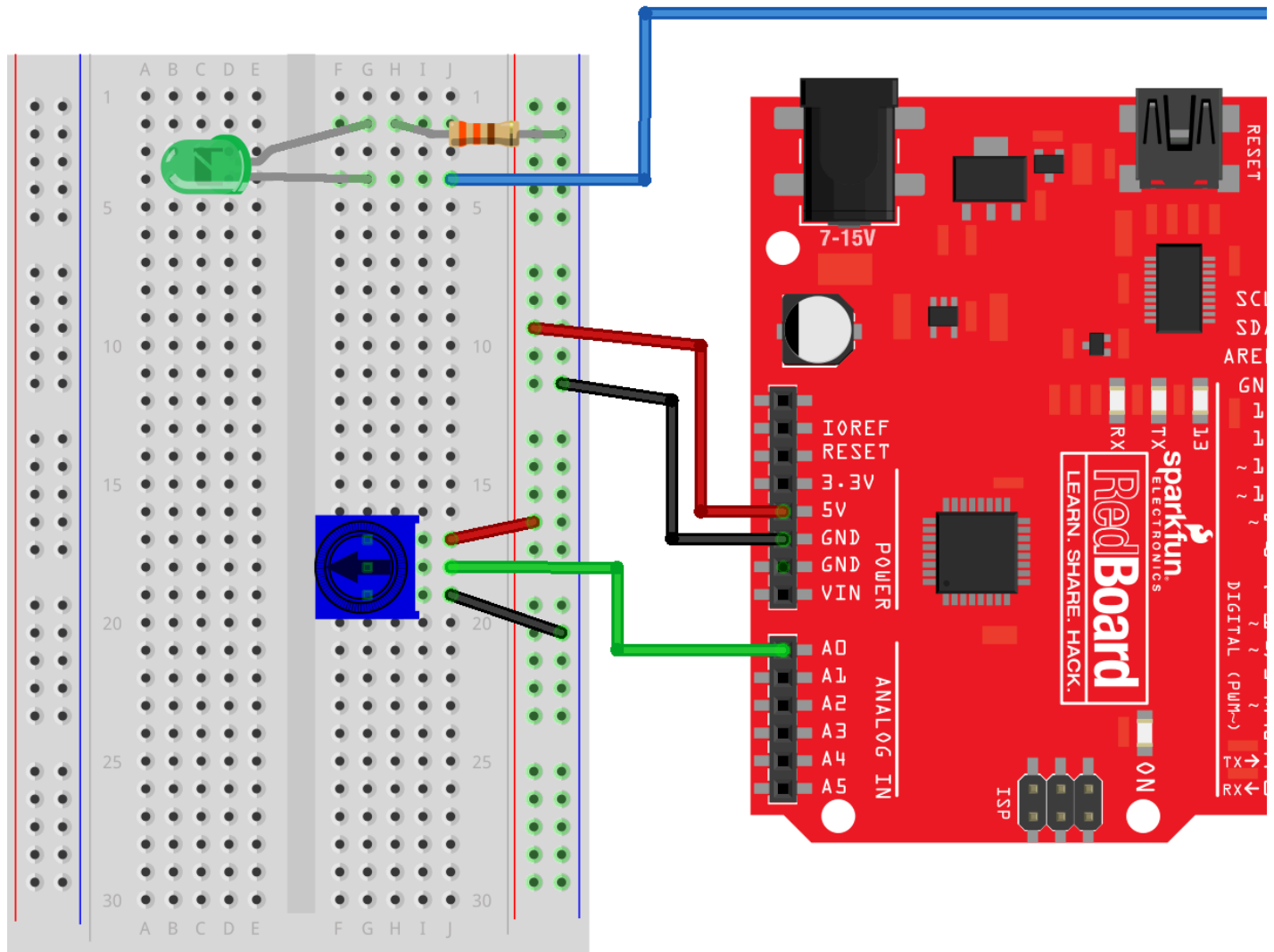
## P7L1: Sound to Light

Let's do an example where we are sending messages to an Arduino from MATLAB.

# The Receiver - Arduino

We'll start by creating the Arduino script. There's a benefit to starting with the Arduino since it has some built-in features that allow us to test and debug before involving MATLAB. In the Arduino IDE, we can develop fully the receiver side and use the Serial Monitor to send some initial test messages. After we've made sure the Arduino script works, then we connect to MATLAB and send from there.

First, create the following circuit:



**WARNING: THE LED IS FLIPPED IN THIS IMAGE. This will be fixed but be aware that the cathode and anode are connected backwards.**

It's just a single LED connected to pin 11 and a potentiometer connected to pin A0.

**WE WON'T USE THE POTENTIOMETER FOR PART 1.** I'm just including it now so you can build the circuit once and then get to focusing on the programming concepts around serial communication.

Here's what we're going to do:

- PROGRAM P7L1 Arduino – Reciever
- Initialize serial communication
- Adjust Serial Time Out
- WHILE 1
- IF Serial Available THEN
- Read a character from the buffer
- IF that character is 'a'
- Read buffer for an integer
- Write that value to the LED pin
- ENDIF
- ENDIF
- ENDWHILE
- END

Use the following to start an Arduino script for this part of the assignment:

[https://raw.githubusercontent.com/boconn7782/CourseCode/main/P7L1\\_ArduinoStarter.ino](https://raw.githubusercontent.com/boconn7782/CourseCode/main/P7L1_ArduinoStarter.ino) It

contains the pseudocode integrated into an Arduino script. It has gaps that we'll start filling in with the new commands.

Here's a demo of the intended behaviors you're looking for:

## Video

Please visit the textbook on a web or mobile device to view video content.



If you decide to swap out the audio, yours will be a little different but should functionally be the same in that it represents the audio levels played in MATLAB through an LED connected to your Arduino.

## Work our way through

Let's start filling in the gaps.

### Variables

The starter code contains most of the things you should already be familiar with, this includes some of the variables. We have the constant integer for the led pin value as well as a variable for storing the output value to write to that LED. We now need a character to hold information read from the buffer. Create an empty character (**char**) variable where indicated. It's later referred to as **SerCode** but you may change that to whatever descriptive variable name works for you. You'll just have to make sure to change that later.

### setup()

Here's where we want to start our serial communication as well as any other tasks we only need to run once. We need to **Initialize serial communication** so we can begin transmitting and receiving information via the USB port.

- Initialize your serial communication as you've done before for debugging.
  - Use the standard baud rate of 9600.

We're also going to use some commands later that are effected by the time out of the serial commands. So we want to ensure that we **Adjust Serial Time Out** so that we can mitigate that.

- Reset the time out value to 500 using the **setTimeout()** command.
  - Reference the earlier section for the exact syntax.

### loop()

Now for the things we'll do repeatedly. To start things off, we need to check and see **IF Serial Available**. If it's not, we don't want to bother accessing the buffer since any of those commands will take unnecessary time waiting to determine there's nothing there before moving on.

- Add the command for checking if there is information in the buffer to the if statement

When it's available, we want to **Read a character from the buffer**, taking a single character and storing it in the character variable we made earlier.

- Add the command for reading a character from the buffer and store it
  - If you decided not to use the name **SerialCode**, update the variable used.

Now that we have a character, we want to check **IF that character is 'a'**.

- Add a conditional to check if the retrieved character's value is equivalent to 'a'

If we received the correct identifying character, we'll **Read buffer for an integer**. The command we use here will read through multiple characters, parsing out any identified integer and returning that.

- Add the command to parse through the following data to identify and return an integer
  - Load that information into the variable DataIn

Now that we have the intended data from the sender, we can **Write that value to the LED pin**, setting the LED output based on an external information source.

## Test your script

Run your script and start the serial monitor. The nice thing about the Arduino IDE is that we can use it to send messages to the Arduino via the Serial Port. Type something like 'a200' to into the serial monitor and hit enter. The drop down to the right should be set to "New Line" so it adds a new line character to the end of every message. This should set your LED level to 200. Try a few more inputs.

If you think it's not responding fast enough, you can lower the time out value. Your timeout is too low when you enter 'a200' and the LED goes really dim. In that case, the `parseInt()` command read the '2' and then timed out fast enough that it didn't have time to retrieve the following '0' before thinking it had completely parsed the integer. If you achieve that, then I would double or even triple that time out value. The Arduino IDE is a very direct connection to the device and the serial connection through MATLAB will likely be slightly slower.

### Question 7

What value did you end up with for your timeout?

[Show Responses](#)[Show Answer](#)

## Video

Please visit the textbook on a web or mobile device to view video content.

## The Transmitter - MATLAB

So we've confirmed that your script can handle receiving commands through serial communication using the built-in serial monitor of the Arduino IDE. Now we're going to make it so those commands are coming from MATLAB. Here's the first step and it's extremely important - **COMPLETELY QUIT AND CLOSE YOUR ARDUINO IDE!!** The serial port can only maintain communication with 1 application at a time. The Arduino IDE automatically connects to your Arduino via the USB port so if it's running, MATLAB will not be able to find and communicate with your Arduino. So make sure you've completely closed the Arduino IDE before you even begin working in MATLAB.

Now let's create an Arduino script for this. We're going to create one that does the following:

- PROGRAM P7L1 MATLAB – Transmitter
- Initialize serial communication with Arduino
- Create an Audio Object
- Convert audio sample levels to representative values from 0 to 255
- WHILE Audio Object is playing

- Retrieve current audio sample
- Retrieve corresponding representative value from 0 to 255
- Create command for receiver Arduino using that representative value
- Send command to Arduino
- ENDWHILE
- END

Some of this pseudocode is very over-simplified like **Create an Audio Object** and most elements associated with it. This lesson is not about some of the advanced MATLAB features so that part is being yada yada yada'd past. You will be given all the code to handle those elements with comments if you want to explore those more on your own.

Use the following to start a MATLAB script for this part of the assignment:

[https://raw.githubusercontent.com/boconn7782/CourseCode/main/P7L1\\_MATLABStarter.m](https://raw.githubusercontent.com/boconn7782/CourseCode/main/P7L1_MATLABStarter.m) It contains the pseudocode integrated into a MATLAB script. It has gaps that we'll start filling in with the new commands. All the commands for creating the audio based demo are included and don't require anything on your part.

## Work our way through

Let's start filling in the gaps.

### Initialize serial communication

This first item, **Initialize serial communication with Arduino**, is something you'll want to remember as it's how you'll start any MATLAB script that you want to connect with an Arduino Uno. The steps involved are:

- Retrieve a list of ports with available serial devices
- Select the last one on that list
  - Your Arduino will typically be the last device on this list because it's the most recently connected. If not, see the debugging note below.
- Open up a serial connection to the Arduino over that port

You've already been given a single command that searches the available **serial ports** and returns a **list** of them. You just need to store that information so we can use it later. The provided script just uses the variable **X** since it's used in the very next line and never again. A descriptive variable is not always necessary in that circumstance.

Remember that we can use **end** as an index to indicate the very last item in an array. Use that to store the last item from **X** as **P**. Again, not the most descriptive variable name but it's being used

just once and in the next line so it's not hard to remember what it refers to.

**DEBUGGING NOTE:** You may not discover this until later when you actually try communicating over that serial port, in very few cases the last item in the serial port list array is not your Arduino Uno. In that uncommon case, here's what you do:

- Get the information from the Arduino IDE
  - Open the Arduino IDE and see what port it uses to connect to the Arduino
  - Write it down then close the Arduino IDE
  - Hard code that string into **P**
- Get the information from MATLAB
  - Unplug the Arduino
  - Run **serialportlist** in the command window and look at the results
  - Plug the Arduino back in
  - Run the command again and look at the results
    - The new element is the Arduino port
  - Hard code that string into **P**

The last part is to open the connection. MATLAB does this by creating an object, a data structure with properties and commands specific to it. We use the command to identify the desired **serial port** and baud rate we want to use then store the returned object. In this case, we're storing it in the variable **ArduinoObj**. Add the proper command and use our selected port, **P**, and a baud rate of 9600, the same baud rate that our Arduino Uno is set to, as the inputs.

## Create Audio Object

We don't have to do anything with this. I just included it to make the demo more interesting. If you'd like to do more with this though, you are welcome to read more into [using audio files with MATLAB](#) and may even swap out the built-in example with your own audio file if you wish. The gist of it is that MATLAB handles audiofiles as a combination of a array of audio samples, **y** in this case, and the sample rate by which their played through, **Fs** here.

I'd wait till you get the baseline demo completed first and come back though.

## Convert Audio Samples

**What is notable here** though is that we've planned for our communication with the receiving component. We, the programmer, knows what the Arduino Uno is looking for in the types of data it will be receiving. We know it wants integers and that it'll be using it as an LED output value, which ranges from 0-255. For a rudimentary system, particularly one where information is

traveling one way, we need to have that established protocol and manually ensure it gets followed throughout.

## Yada Yada Yada

There are some more elements of the script that are handled for you since they are associated with the audio demo. Since it's all happening in a short loop where we have other things we will be concerned with, here's the gist of what's happening just so you have the context.

- **WHILE Audio Object is playing**
  - **isplaying(AudioObject)** gives a boolean based on whether the audio object is currently playing or not
- **Retrieve current audio sample**
  - The **get** command retrieves properties from objects. In this case, the specific sample being played from the audio array the instant this command is called.
- **Retrieve corresponding representative value from 0 to 255**
  - The index of that sample is used to retrieve a representative value from 0 to 255 from the converted representative array created earlier.

## Create Command

Now we need to create the command. We tested earlier in Arduino with commands of the form 'a200', the character 'a' followed by an integer with a value of 0-255. We have the integer we want to use taken from the representative array we created earlier, stored in the variable **L**. We want to convert that to a string while adding an 'a' to the beginning. This sounds like something where you'd want to have some heavy format control, say like when using everybody's favorite command **fprintf()**. Now **fprintf()** outputs to the command window or a designated file but we just want it as a string. There's a similar command that functions the exact same as **fprintf()** except it returns a string locally rather than writing it externally, **sprintf()**.

- Use **sprintf()** to create your message for the receiver.
  - It should start with **a** then be followed by an *integer* taken from **L**.
  - Capture the output in **M**.

## Send Command

Our last task is to send the command to the Arduino. For this, we want to **write** that **line** of data to the serial port connected to our Arduino. Utilize the appropriate command for doing that in MATLAB.

- Add a command to sends the message to the intended receiver through the serial port

# Run your scripts

Now it's time to run your scripts.

1. Make sure Arduino has the correct script running
2. Close the Arduino IDE if it's not already
3. Run your script on MATLAB. Make sure sound is on.

Watch how the Arduino responds. The light and sound level may not perfectly line up but it should be reasonably close enough. You can experiment with changing the Time Out value on Arduino and the time between serial communications in MATLAB. Increasing or decreasing the window value of smoothdata, currently set to 1000, will also effect things.

## Video

**Please visit the textbook on a web or mobile device to view video content.**

## Expanding the concept

Since we didn't just send integers but rather used an identifier along with the integers, we can easily expand this to include multiple inputs. You could add some else-if statements to also be able to handle a message sent with data from another source. Right now, MATLAB can send 'a200' and your Arduino knows how to handle it and that it's intended for the output/code associated with 'a'. Expanding the IFTTT statement to handle more case would allow your system to handle commands like 'b100' and know how to handle that. The example 'Handel' only has one audio source in it but more complex audio files have multiple channels (like a different the left and right channel for your headphones). You could have an a different LED for each channel. That's just

expanding for this example. You can utilize this to handle giving commands to several different actuators connected to your Arduino.

## Submissions for P7L1

This is part 1 of 2. You will still need to complete Part 2 and submit all of the associated files for that along with this part. Please prepare and submit the following for this P7L1:

- The raw files
  - The .ino file for the Arduino portion
  - The .m file for the MATLAB portion
- A PDF of the code and circuit
  - Create a PDF that contains
    - A copy of the Arduino code
    - A copy of the MATLAB code
    - An image of your circuit
  - Either
    - start a doc to compile this info and then print to pdf
    - create individual pdfs from the raw files then combine them into 1 pdf
- A video showing your working circuit
  - You can upload the raw video file but PLEASE also go back to the assignment, after submitting, and add the videos as media comments. This makes the reviewing the assignment much easier in Canvas and that means happier (More forgiving) graders.

## P7L2: Gauging Resistance

Now let's go in the other direction.

### The Transmitter - Arduino

We'll start with Arduino again but mainly because there's nothing new here. We're just starting with a simple example where we have a sensor on Arduino taking in data and then sending a message out through the serial port based on that data. The built-in example

**AnalogInOutSerial** does everything we need. Start with that example and you'll even end up deleting about half of it. Don't forget to update the title block to make it your own:



1. `/*`
2. `Programming 7 Pre-lab 2`
3. `<First Initial>.<Last Name>`
4. `<Date of creation>`
5. `Gauging Resistance – Arduino as the Transmitter`
6. `Reads the potentiometer value then sends a`
7. `formatted message over the serial port with`
8. `that data converted from 0–100`
9. `*/`

You should already have the circuit in place. We're using the potentiometer for this part.

**We will not** be using the LED.

We're going to create a script that does the following:

- `PROGRAM P7L2 Arduino – Transmitter`
- `Initialize serial communication at 9600 bps`
- `WHILE 1`
- `Read the Analog In Value`
- `Map it to a range of 0 to 100 as an output value`
- `Print a message to the serial port using the output value`
- `Wait 2 ms for the analog-to-digital converter to settle`
- `ENDWHILE`
- `END`

For our message to Arduino, we kept it very simple. This is because of Arduino's limited ability to navigate more complex messages. [It has the ability](#) but it's a bit more advanced while MATLAB's ability to parse strings and character arrays is more intuitive. For that reason and to provide the demonstration, we're going to send a more complex message this time.

Have your message be '**S1:**' followed by the output value. Also include a few spaces afterwards. This serves a purpose later in MATLAB and will be explained then. So you should be outputting messages that look like this: '**S1:2** ', '**S1:47** ', or '**S1:100** '. Test your script and see what gets printed to the serial monitor. You may have to increase the delay to see but make sure you return it to the original value, 2, and upload your script before closing the Arduino IDE.

- There are really only 2 things you need to edit in the example to get the desired outcome
  - Convert the potentiometer input from 0 to 100
  - Print that converted value as part of a formatted message to the serial monitor

- Remember that you want a linebreak at the end of your message
- Everything else that's not associated with that and unused, can be deleted or commented out.
  - I recommend commenting out line by line and testing before deleting. That way if it doesn't run during a test or you get an error, you can uncomment that line to get it back to a working condition then rethink why that line might be necessary.

## Video

Please visit the textbook on a web or mobile device to view video content.

## The Receiver - MATLAB

Once you've created your Arduino script and confirmed it's outputting the messages you want for MATLAB, we can start building our receiver script in MATLAB.

So we've confirmed that your script can handle receiving commands through serial communication using the built-in serial monitor of the Arduino IDE. Now we're going to make it so those commands are coming from MATLAB. Here's the first step and it's extremely important - **COMPLETELY QUIT AND CLOSE YOUR ARDUINO IDE!!** The serial port can only maintain communication with 1 application at a time. The Arduino IDE automatically connects to your Arduino via the USB port so if it's running, MATLAB will not be able to find and communicate with your Arduino. So make sure you've completely closed the Arduino IDE before you even begin working in MATLAB.

Now let's create an Arduino script for this. We're going to create one that does the following:

- PROGRAM P7L1 MATLAB – Receiver
- Initialize serial communication with Arduino
- Create User Interface Objects – Gauge and Button
- WHILE 1
- Retrieve latest message from serial port
- IF message contains our identifier characters THEN
- Parse message for the sensor data
- Set Gauge Value based on that data
- ENDIF
- Check button to determine if script should end
- ENDWHILE
- END

Some of this pseudocode is very over-simplified like **Create User Interface Objects** and **Check button to....**. As before, this is not about the advanced MATLAB features so that part are being yada yada yada'd past again. You will be given all the code to handle those elements with comments if you want to explore those more on your own.

Use the following to start a MATLAB script for this part of the assignment:

[https://raw.githubusercontent.com/boconn7782/CourseCode/main/P7L2\\_MATLABStarter.m](https://raw.githubusercontent.com/boconn7782/CourseCode/main/P7L2_MATLABStarter.m) It contains the pseudocode integrated into a MATLAB script. It has gaps that we'll start filling in with the new commands. All the commands for creating the UI objects for the demo are included and don't require anything on your part.

Here's a demo of the intended behaviors you're looking for:

## Video

**Please visit the textbook on a web or mobile device to view video content.**

If you decide to swap out user interface components, yours will be a little different but should functionally be the same in that it represents the sensor readings from Arduino being displayed in MATLAB.

## Work our way through

Let's start filling in the gaps.

### Initialize serial communication with Arduino

We've done this before and you do it the same way here.

- Add your code to connect to the Arduino through Serial
  - Reference P7L1

### Create User Interface Objects - Gauge and Button

This is already done for you. We've created a gauge because that's an easy way to represent the potentiometer level in MATLAB. The button is because we're going to start a forever loop in our script. Since we need to repeatedly check the serial port, we're not just running through our script once like we've done in the past. We want this to run perpetually so we've included **WHILE 1 ... ENDWHILE**. Everything in between will loop continuously. We're using the button as a means to manually break the loop, allowing the script to end.

Just to abate some questions, [the User Interface objects](#) are like the serial object. They have properties unique to them like **ScaleColors**, **Style**, or **Value**. The [gauge](#), [button](#), and every other available user interface component has specific properties associated with it as well as callbacks, functions only accessible to that type of object. For more information on these, you can look through the MATLAB help and tutorials on [developing apps with user interfaces](#). MATLAB also has a feature called [App Designer](#) that has a bit of a steep learning curve but, once you know how to use it, makes developing user interfaces much simpler using features like drag and drop placement of components rather than having to programmatically set their **Position**.

Yes, that's a lot of words and several reference links for something we're just **yada yada yada'ing**. It's almost like your surreptitiously being given a heads up and prepared for some activities to come or useful resources for a major thing you're working on ...

### Retrieve latest message from serial port

We want to make sure we're working with the most up to date information from the Arduino. The Arduino has been sending signals and filling up the buffer ever since you plugged it into the USB.

The top line message might be from minutes ago. So we want to **clear the serial port buffer** of all of those old messages. Add a command to do just that, like water down a toilet... Now that the buffer is cleared, we know that the next message sent to it will be the latest. So we should **read** that **line** into a variable as a string. The starter code is set up to use the variable **ArdMsg**, short for Arduino Message.

- Add command to clear the serial port buffer
- Add command to read the latest line in the buffer into MATLAB as a string
  - Store that message in **ArdMsg**

**NOTE:** Clearing the buffer is not something you want to just once in your script but before every time you read the buffer. MATLAB is running through an environment meant to be primarily a human interface. Your scripts also reflect that by having delays significant enough the people can keep up with it. A microcontroller runs at astronomical speeds because it's all happening internally and with little regard for a human's perception. You can see this mentality in just the way they implement delays. Arduino's **delay()** command takes milliseconds as its input while MATLAB's **pause()** command works with seconds.

## IF message contains our identifier characters THEN

MATLAB has lots of options for scanning through a string to see if a specific substring is present. If you search for this online, you'll find lots of options from short algorithmic suggestions like converting it to characters then running a for loop to specific commands built-in to MATLAB over the years. A more recent addition is the **extract()** command, which is very powerful but equally complicated. I only mention it as it's another option they've added in recent additions.

I'm going to outright recommend using [the \*\*strfind\(\)\*\* command](#). This has been tangentially mentioned in other labs and available in the MATLAB resource. Here is where it's very applicable though. We have a known identifier in the message, **'S1: '**, that lets us know that we have a valid message from our Arduino and not junk, an incomplete message, or something formatted differently accidentally captured from another source. **strfind(A,B)** takes in two inputs, the string(A) you want to search and the pattern(B) you want to check and see if it's in it. It will return the starting index for every instance of the pattern in that string. If the pattern is not there, it'll return an empty matrix.

We're looking for the conditional **'message contains our identifier characters'**. Remember that anything non-zero is interpreted as True. So calling **strfind(A,B)** for a case where that command would provide you even a 1 in the case where B started with A, would be interpreted as true.

- Add a command call as the conditional in the if statement
  - Should be True when the message contains 'S1:'

## Parse message for the sensor data

This is another circumstance where there's a lot of options. The `extract()` command could do it or you could go as simple as converting the string to a character array then grabbing the integer characters, since you know where they are, and converting them to numbers, sort of like a manually implemented version of Arduino's `ParseInt` command. So you could do:

- `j = strfind(ArdMsg, 'S1:'); % Locate index of identifier phrase`
- `k = char(ArdMsg); % Convert to Character Array`
- `l = k(j+3 : j+(3+4)); % Pull out characters containing the data`
- `PotVal = str2num(l); % Convert to an integer`

Here's a more detailed breakdown of that approach:

We use `strfind()` to get the index of the start of our identifier characters. Next we convert that to a character array so we can work with the individual characters. We then pull out the characters that contain the integer. Since we know the identifier characters are 3 characters long, we add 3 to the known starting character and use that as the start of our index array. We also know the integer will never be more than 3 characters long. We had included some extra spaces in our message from the Arduino and this is why, so it could handle a single digit integer without errors. We then use `str2num()` to convert that subset of the character array to a number. This is not that complex and may be fairly intuitive for your current level and the commands you've worked with so far but there are better ways to do this.

A useful middle ground for complexity and efficiency though would be `sscanf()`. This is essentially `sprintf()` in reverse. It scans a string and reads formatted data. Try the following in your command window:

- `str = sprintf('S1:%d \n',95); % This mimics a message from your Arduino`
- `X = sscanf(str, 'S1:%d')`

The command will look for the matching elements in the string then interpret the portion that aligns with the indicator `%` based on the formatting operators given. This does what those 4 lines of code did earlier and wouldn't need the trailing spaces added to the Arduino message.

- Parse the message for the sensor data

- Save the returned integer into **PotVal**

pull out the characters from the index starting 3 after the start of the identifier phrase (Since we know the identifying phrase is 3 characters long) plus the next 4 characters (Since we know the integer will never be more than 3 characters long and that we have some extra spaces

## Set Gauge Value based on that data

This is pretty straightforward. We're updating the gauge property '**Value**' using the data extracted from the message. You can update the value of any UI object's properties this way, by setting them equal to something new.

## Check button to determine if script should end

You can also reference the value of any UI object's properties by calling it as it's shown here. MATLAB will update certain properties based on the user interaction with that object. The default value for an unselected button's Value is 0. When the button is pressed, it's Value becomes 1. You don't have to program that in, MATLAB handles those elements on the back end.

So when pressed, we get a 1 from that button value and therefore the statement becomes true here. It then calls the command **break** which manually breaks the loop it's within. This ends our forever loop allowing the script to end. We have some clean up elements that clear the Arduino Object and close the user interface as best practices.

## Run your scripts

Now it's time to run your scripts.

1. Make sure Arduino has the correct script running
2. Close the Arduino IDE if it's not already
3. Run your script on MATLAB.

Watch how the the Gauge responds as you turn the potentiometer. It should very quickly, if not instantaneously by your perception, update and move with the potentiometer movements. You can experiment with changing either the pause in MATLAB or the delays in Arduino. If you comment out the flush, you'll notice that there becomes a long delay in the gauge's response because it's reacting to older messages, not the most up to date.

# Video

Please visit the textbook on a web or mobile device to view video content.

## Debugging

Here are some common issues and how to debug them. If you come across anything not covered by this, contact me and we'll work through it asap. This is the first run of this assignment and, as much as I've gone over it, there's always something. I'll update this if any new issues arise but hopefully not:

- First of all, removing the semi-colon, `;`, from the end of a line so that it's output gets printed to the command window. That's a good way to check if the messages from the Arduino are being misread or if you're parsing them incorrectly.
  - I'd recommend doing this particularly for your `ArduinoObj` to make sure the object is being created properly. You should see something like the following if it worked.
    - `ArduinoObj =`
    - `Serialport with properties:`
    - `Port: "COM3"`
    - `BaudRate: 9600`
    - `NumBytesAvailable: 0`
  - You should also do this for your `ArdMsg` to make sure you're getting the messages you expected from the Arduino.
- The most likely issue is a timing one, where `flush()` actually flushes the buffer not between lines but in the middle of one, so you get a message that's not `"S1:95"` but rather `":95"` or `"1:95"` which we're not set up to handle.
  - To identify this issue:
    - Remove the semi-colon after your readline, so just have `ArdMsg = readline(ArduinoObj)` without the semi-colon. This will print `ArdMsg` to the command window and let you see what you're getting each run.
  - To fix the issue:
    - Repeat the `readline()`. The first `readline()` will read the incomplete message, removing it from the buffer. The next `readline()` will capture the next complete message from the Arduino.



- The cause:
  - Typically it's a speed issue. Your computer is reading the serial port just slightly slower or differently than MATLAB expects, so it interprets the space between individual characters received as the space between messages received. So `flush()` isn't catching it in the short pause between messages but between characters.
- Depending on how you formatted your message from Arduino, your `sscanf` may not be picking up the pattern correctly. If you get this error when running `g.Value = PotVal;` to update the gauge value: `'Value' must be a finite numeric, such as 10.` then you've likely have an `sscanf` issue.
  - This likely means that `PotVal = sscanf(ArdMsg, 'S1:%d');` is resulting in `PotVal` being set equal to `[ ]` because `sscanf` is expecting your `ArdMsg` to start with `'S1: '` but it actually starts with spaces or a tab, so it's not seeing the pattern in your string.
  - Adding a space to the beginning and end of your format specifier, so going from `'S1:%d'` to `' S1:%d '`, helps it to realize that the `S1:` doesn't have to be at the very beginning of the line. This is a slightly nuanced format issue but that's how string parsing tends to get.

## Expanding the concept

Being more efficient with our interpretation of the message can allow us to expand our capabilities. Let's say we're sending out 3 sensor values instead of 1. We can use `sscanf()` to extract them from the message easily, particularly if that's factored into our initial protocol. Try this:

- `str = sprintf('S1:%d S2:%d S3:%d \n',95,36,68); % This mimics a message from your Arduino`
- `X = sscanf(str, 'S1:%d S2:%d S3:%d')`

This returns an array of the 3 values from the message using one line of code. That's much easier than having to a couple lines of character array manipulation for each desired data point.

## Submissions for P7L2

This is part 2 of 2. You will still need to submit both with all of the associated files for each. Please prepare and submit the following for this P7L2: