Arduino Mega Coding Tutorials to Use with the Cosmic Ray Detector DAQ Electronics

Gabe Kim / Fatima Saleem / Sonia Rostami / Lily Carter / Sam Resto

Queensborough Community College

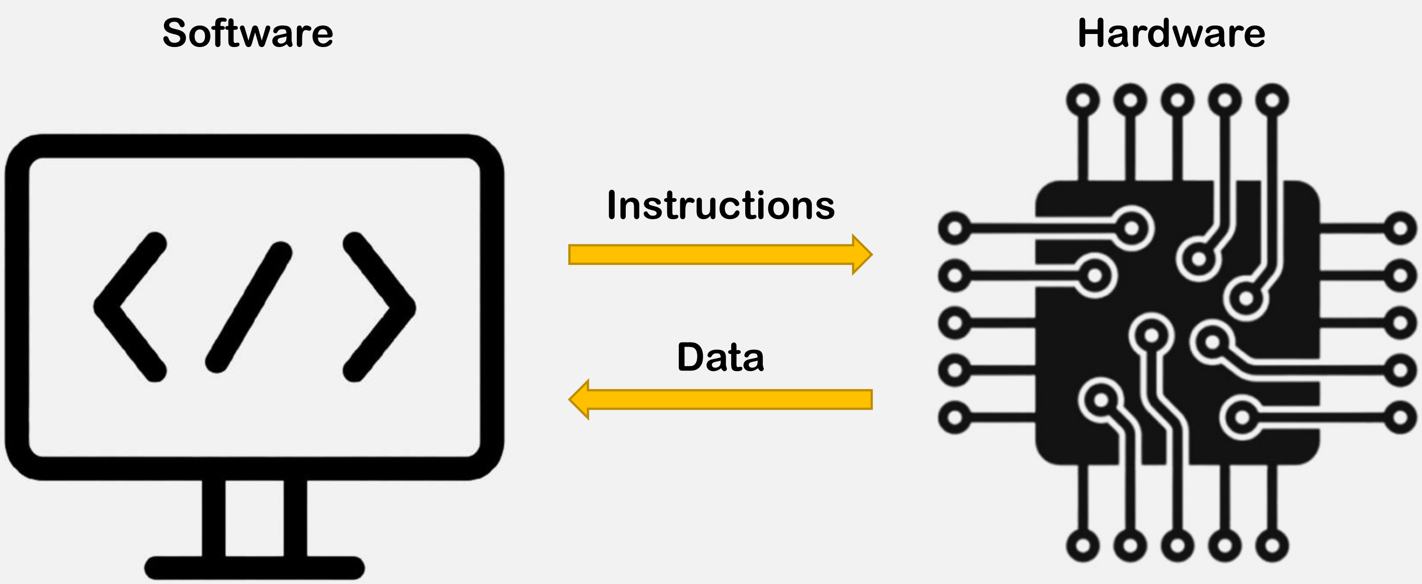
Mentor: R. Armendariz

Module I

Hardware Overview

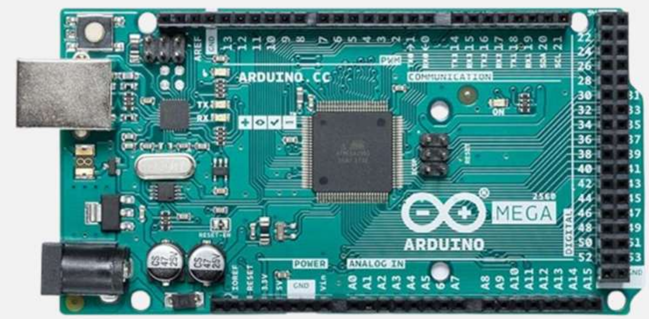
What is Arduino?

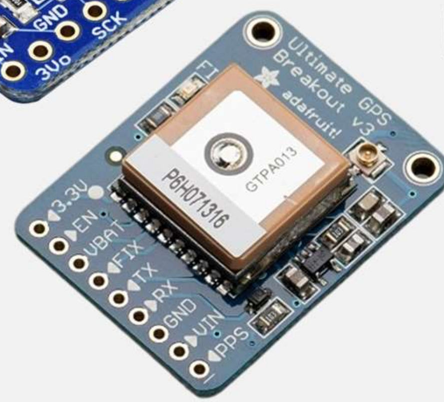
Arduino is an open-source electronics platform that combines hardware and software to create interactive projects. It utilizes a variety of microcontroller-based boards, which can be programmed using the Arduino IDE (Integrated Development Environment), a software application where you write the code that tells the Arduino what to do.

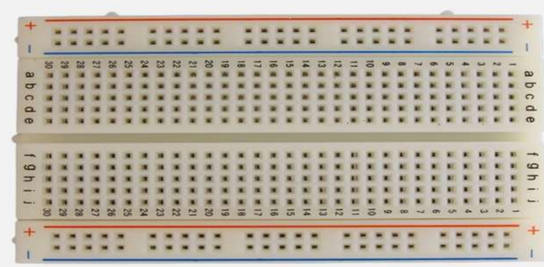


Hardware Overview

Your first step should be to familiarize yourself with the hardware you'll be using. Understanding the purpose and function of each component is important for resolving troubleshooting issues and designing effective circuits. It also helps prevent damage by ensuring safe connections and simplifies the integration of components into your projects, improving overall functionality and reliability.







Microcontroller

Arduino Mega 2560

The Arduino Mega 2560 is a type of microcontroller, which is a small computer on a single circuit board. It is used to control various electronic devices and projects. Imagine it as the "brain" that tells other parts what to do.

Here’s how it works:

•**Inputs:** It can take signals from sensors (like a temperature sensor or a GPS chip) that provide information.

•**Processing:** The Arduino uses this information to make decisions based on a program (set of instructions) that you write in your code.

•**Outputs:** After processing the inputs, it can control things like lights, motors, or sounds by sending signals to them.

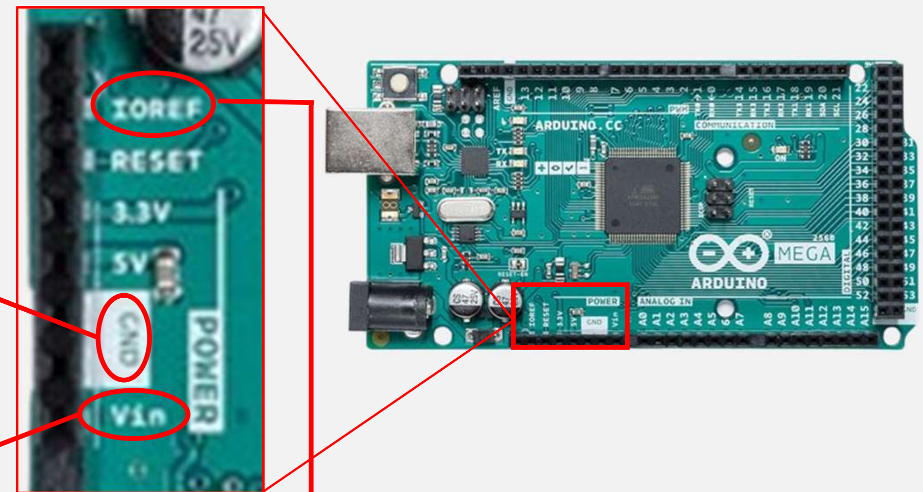
Microcontroller – Arduino Mega 2560

Microcontroller – Power Pins

Arduino Mega 2560 Pins

The **GND pins** on the Arduino Mega 2560 are used to complete the electrical circuit by providing a common ground. There are multiple GND pins on the board, and they are all interconnected internally. Any device or sensor connected to the board that requires power must also be connected to one of the GND pins to ensure proper current flow and circuit stability.

The **Vin pin** is used to supply external power to the Arduino Mega 2560 when it is not connected to a computer via USB. You can connect a power source like a battery or an external power adapter to this pin. The voltage input should typically be between 7-12V. The onboard voltage regulator then steps this down to the 5V required to power the board. When the board is powered through Vin, it provides a way to power both the board and external components without relying on USB power.

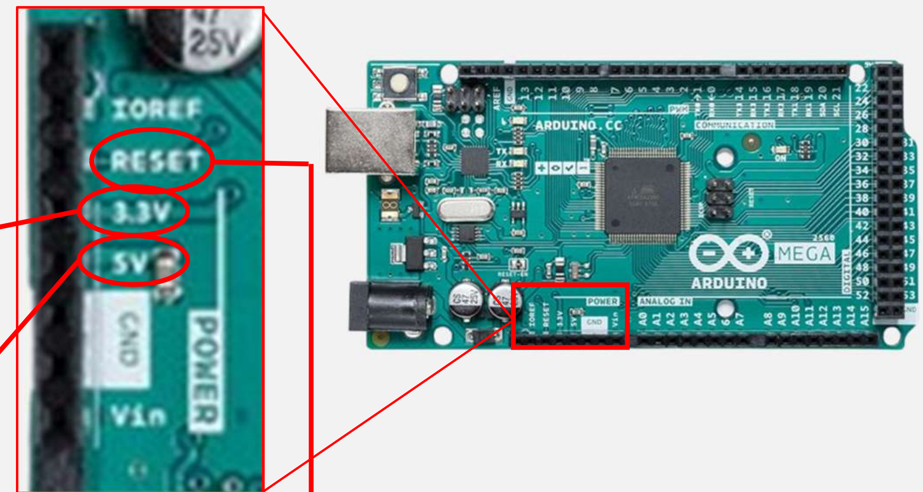


The **IOREF pin** on the Arduino Mega 2560 provides the reference voltage for the board's input/output pins, typically 5V, letting external components know the board's **logic voltage**. This is important because different devices may operate at different logic levels (e.g., 3.3V or 5V), and the IOREF pin helps them adapt to the correct voltage. **Logic voltage** refers to the voltage levels used to represent digital signals (binary 1/0), where "high" is typically 5V and "low" is 0V. This pin ensures safe communication and prevents potential damage from voltage mismatches.

Microcontroller – Power Pins

Arduino Mega 2560 Pins

The **3.3V pin** provides a regulated 3.3V output for powering components or sensors that operate at lower voltages. This is especially useful for devices that require lower voltage, as connecting them to the 5V pin might damage them. The 3.3V pin is powered by an onboard voltage regulator, ensuring that it delivers a stable 3.3V output.



The **5V pin** supplies a regulated 5V output, which can be used to power external components or sensors that require 5V to operate. This pin is powered either through the USB connection or via the voltage regulator when the board is powered through the Vin pin. It provides a stable 5V, making it convenient for powering devices directly from the board without needing an additional power supply.

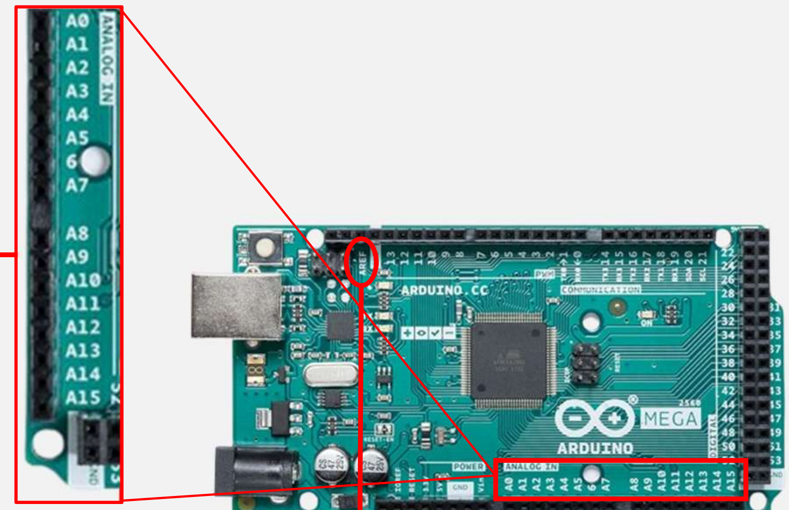
The **Reset pin** is used to reset the microcontroller on the Arduino Mega 2560. When you connect this pin to ground, it triggers a manual reset of the board. This can be useful in situations where you need to restart the board and reload the program, without disconnecting power or hitting the physical reset button. It's often used in circuits where automatic or remote resets are required.

Microcontroller – Analog Pins

Arduino Mega 2560 Pins

The pins in the **analog section (A0 – 15)** of the Arduino Mega 2560 are used for reading analog input signals and converting them into digital values that the microcontroller can process.

- **Function:** These pins are used to read varying voltage levels from sensors or other input devices (e.g., temperature sensors). They can measure a range of voltages between 0V and 5V.
- **Analog-to-Digital Conversion (ADC):** The Arduino Mega has a 10-bit ADC, which means it can convert the analog input into a digital value between 0 and 1023. For example, 0V would be read as 0, and 5V would be read as 1023, with values in between representing the corresponding voltage.
- **Analog input signals** are continuous, variable electrical signals that can take on a range of values, unlike digital signals, which are either on (1) or off (0).



The **AREF pin (Analog Reference Pin)** on the Arduino Mega 2560 is used to set a custom reference voltage for the analog-to-digital converter (ADC). By default, the Arduino uses 5V as the reference voltage, meaning that it maps input voltages between 0 and 5V to a digital range of 0 to 1023.

However, by connecting a different voltage to the AREF pin (typically between 1.1V and 5V), you can adjust this range to match the expected input from your sensors, improving accuracy. For example, if your sensor outputs a maximum of 2.5V, you can set the AREF to 2.5V, and the Arduino will map the input more precisely across the 0 to 1023 range. However, setting the AREF voltage too low (like 0.005V) would reduce the resolution significantly and might result in unusable or noisy data.

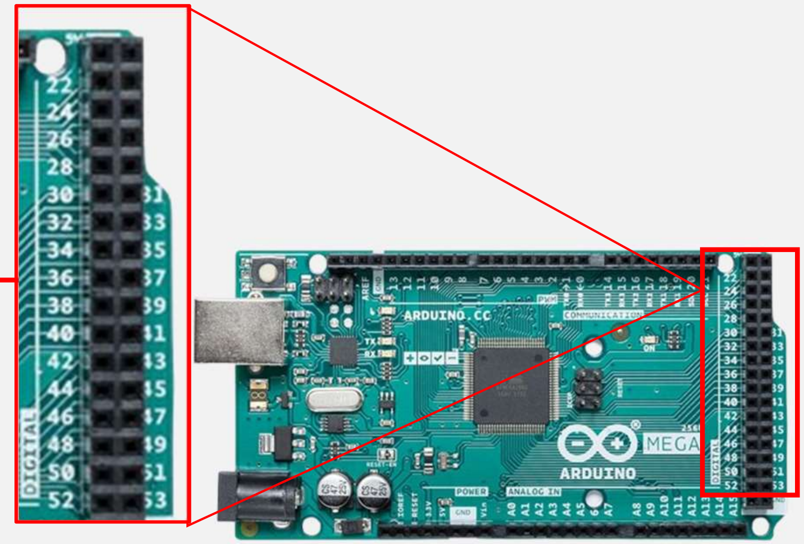
Microcontroller – Digital Pins

Arduino Mega 2560 Pins

The **digital pins** on the Arduino Mega 2560 are used for input and output of digital signals. These signals can only have two states: **HIGH** (on) or **LOW** (off). When used as inputs, digital pins can read the state of external devices like buttons, switches, or sensors, detecting whether they are in an on/off state. When used as outputs, the digital pins can control devices such as LEDs, motors, or relays, by sending a HIGH (5V) or LOW (0V) signal to turn them on or off.

- **Pins D22 to D53** can be used as digital inputs or outputs.
- **HIGH/LOW Logic:** The pins output 5V when set to HIGH and 0V when set to LOW.

*** The digital pins can be utilized by code written in the Arduino IDE to control the function of external components.

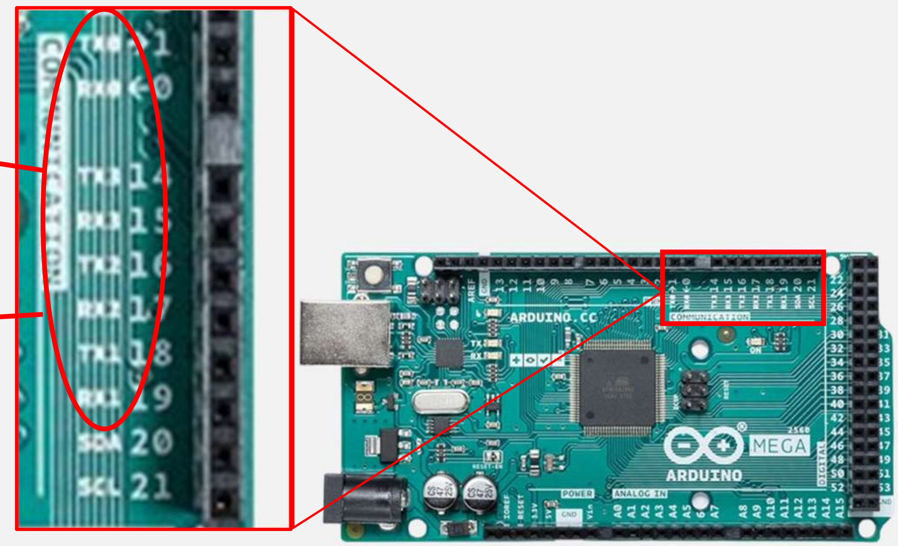


Microcontroller – Communication Pins

Arduino Mega 2560 Pins

TX Pins (TX0, TX1, TX2, TX3): These pins are used to transmit data from the Arduino to other devices. Each TX pin corresponds to a specific serial port: TX0 is associated with Serial, TX1 with Serial1, TX2 with Serial2, and TX3 with Serial3. When the Arduino sends data, it sends it out through the appropriate TX pin, which can be connected to the RX pin of another device for communication.

RX Pins (RX0, RX1, RX2, RX3): These pins are used to receive data from external devices into the Arduino. Each RX pin corresponds to a specific serial port: RX0 is associated with Serial, RX1 with Serial1, RX2 with Serial2, and RX3 with Serial3. When data is sent from another device, it enters the Arduino through the appropriate RX pin, allowing the microcontroller to process the incoming data.



What is Serial Data?

All TX pins operate using the same serial communication protocol (UART). This means they all transmit data in the same way, using the same format (typically a start bit, 8 bits of data, then a stop bit).

Bit-by-Bit Transmission: In serial communication, data is transmitted one bit after another.

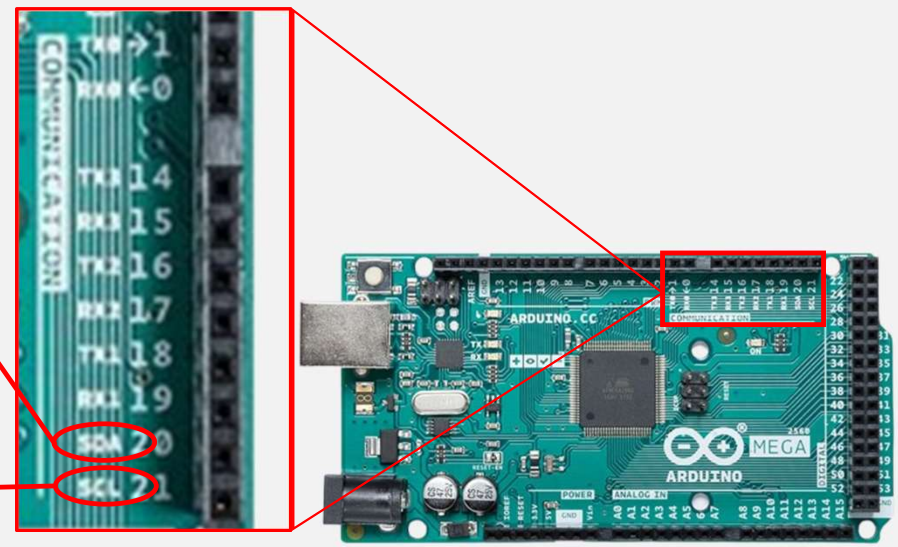
Start and Stop Bits: In asynchronous serial communication, each data packet typically starts with a start bit and ends with one or more stop bits. This helps the receiving device know when to start and stop reading the incoming data

Data Rate: Serial communication is characterized by its baud rate, which is the number of signal changes or symbols sent per second. This is important for determining the speed of data transmission.

Microcontroller – Communication Pins

Arduino Mega 2560 Pins

The **SDA pin** is the Serial Data Line used for I2C (Inter-Integrated Circuit) communication. It is responsible for carrying the data being transmitted between the master device (e.g., the Arduino) and one or more slave devices (such as sensors, displays, or other microcontrollers). The SDA line is bidirectional, allowing data to flow in both directions, depending on the communication needs.



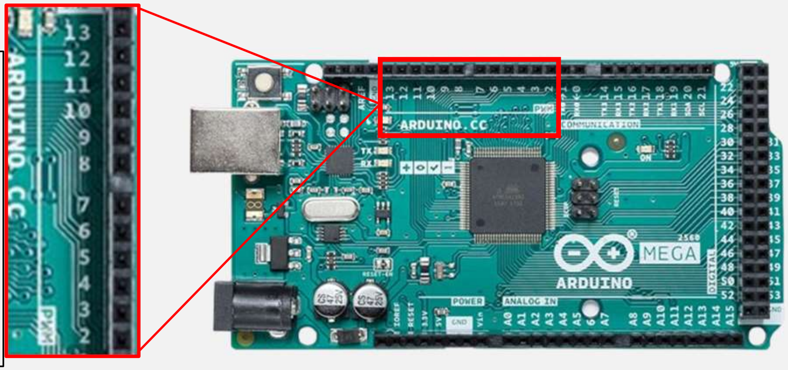
The **SCL pin** is the Serial Clock Line for I2C communication. It provides the clock signal that synchronizes the data transmission over the SDA line. The master device generates the clock signal on the SCL pin, which ensures that both the master and slave devices are synchronized in terms of timing during data exchange. The SCL line is essential for coordinating when data bits are read from or written to the SDA line.

Microcontroller – PWM Pins

Arduino Mega 2560 Pins

On the Arduino Mega 2560, the following pins are capable of generating PWM signals and here are their default assigned timers:

- **Pin 2**: PWM Output (Timer 3) **Pin 3**: PWM Output (Timer 3)
- **Pin 4**: PWM Output (Timer 0) **Pin 5**: PWM Output (Timer 3)
- **Pin 6**: PWM Output (Timer 4) **Pin 7**: PWM Output (Timer 4)
- **Pin 8**: PWM Output (Timer 4) **Pin 9**: PWM Output (Timer 2)
- **Pin 10**: PWM Output (Timer 2) **Pin 11**: PWM Output (Timer 1)
- **Pin 12**: PWM Output (Timer 1) **Pin 13**: PWM Output (Timer 0)



Timer Differences

Feature	Timer 0 (4,13)	Timer 1 (11,12)	Timer 2 (9,10)	Timer 3 (2,3,5)	Timer 4 (6,7,8)
Bit Resolution	8-bit	16-bit	8-bit	16-bit	16-bit
Count Range	0 to 255	0 to 65,535	0 to 255	0 to 65,535	0 to 65,535
Special Functions	Basic timing, PWM	Input capture, output compare, high-resolution PWM	Basic timing, PWM	Input capture, output compare, high-resolution PWM	Input capture, output compare, high-resolution PWM
Description	Used for general-purpose timing and PWM.	High-resolution timing and precise PWM.	Similar to Timer 0, used for PWM generation.	Advanced timing and PWM capabilities.	Similar to Timer 3, provides high-resolution PWM.

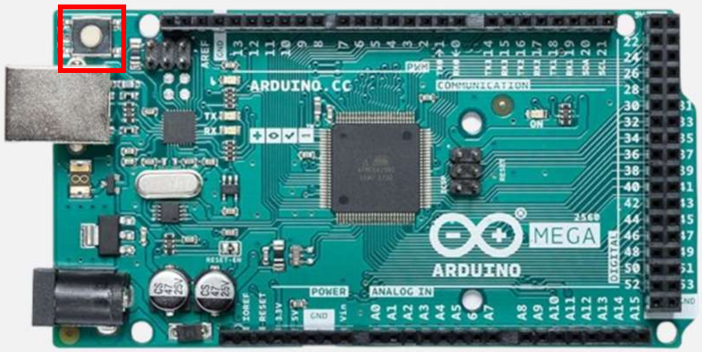
Microcontroller – Reset Button

Arduino Mega 2560 Reset Button

On the Arduino ATmega2560, the reset button essentially restarts the microcontroller, stopping all current processes and returning it to the initial state. Here's what happens when you press it:

- **Program Restart:** The microcontroller stops its current program, clears any ongoing tasks, and restarts from the beginning of the loaded code.
- **Memory and Variables:** Temporary variables and states stored in RAM are cleared. Non-volatile storage (EEPROM and Flash memory) remains unchanged, meaning stored code and saved data stay intact.
- **Bootloader Activation:** If the reset occurs while the Arduino is connected to a computer (like during programming), it briefly enters bootloader mode, allowing it to accept new code if uploading.

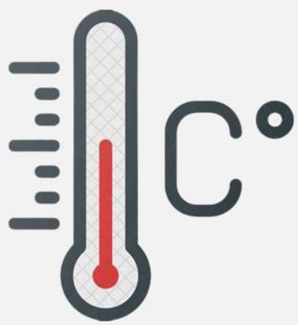
The reset button is handy for debugging, allowing you to restart the Arduino without unplugging it or cycling power.

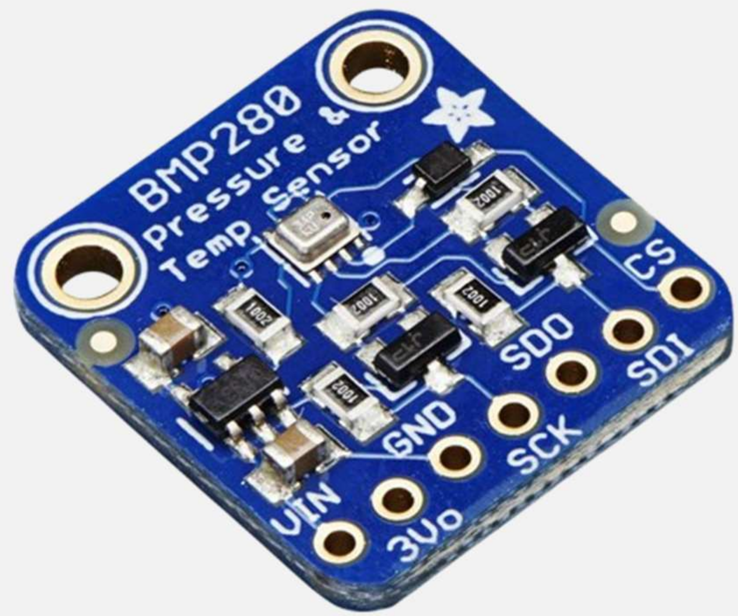


Temp. & Pressure Sensor

Adafruit BMP280 – Barometric Pressure and Temperature Sensor

The BMP280 sensor measures the **temperature** and **barometric pressure** of the surrounding air or environment in which it is placed. It detects the ambient temperature and the atmospheric pressure and is responsible for sending that data to the microcontroller.



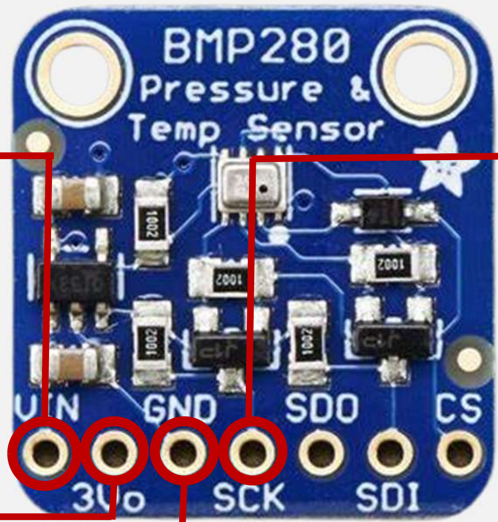


Temp. & Pressure Sensor

Adafruit BMP280 Sensor Pins

The **VIN (Voltage Input) pin** on the BMP280 sensor is used to connect the sensor to a power supply. It typically accepts a voltage range of 3.3V to 5V, allowing the sensor to operate. This pin provides the necessary power for the sensor to function and perform measurements of temperature and barometric pressure.

The **3Vo (3 Volt) pin** is used to provide a regulated 3.3V output. This pin can be used to power other components in your project, such as sensors or small devices that require 3.3V. It allows you to simplify your wiring by drawing power from the BMP280 module instead of using a separate power supply.



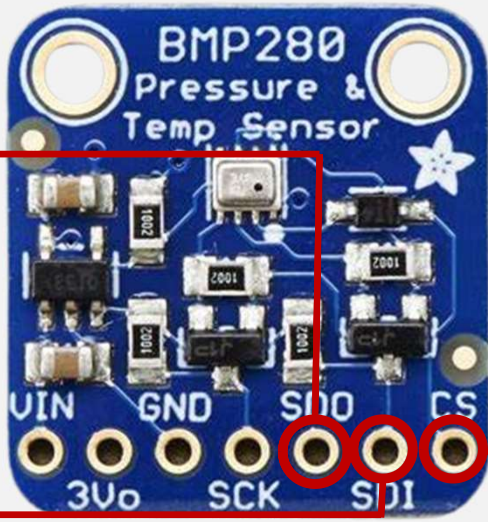
The **SCK pin** (Serial Clock) is used in SPI (Serial Peripheral Interface) communication to provide a timing signal from the master device (usually a microcontroller) to the BMP280 sensor.
In this setup, the SCK pin acts as a clock that synchronizes the data exchange between the master and the sensor. When the microcontroller sends a signal on the SCK pin, it indicates to the BMP280 when to read or send data. This ensures that both devices are working in harmony, allowing for accurate communication. Without the SCK pin, the sensor and microcontroller wouldn't be able to coordinate their data transfer effectively, potentially leading to errors in the information exchanged.

The **GND pin** (Ground) on the BMP280 sensor is used to establish the ground connection for the sensor. This pin completes the electrical circuit by providing a common reference point for voltage levels. It should be connected to the ground (negative) terminal of your power supply or microcontroller. Proper grounding is essential for the sensor to function correctly, as it helps to stabilize the signals and prevent noise in the system.

Temp. & Pressure Sensor

Adafruit BMP280 Sensor Pins

The **SDO pin** (Serial Data Out) on the BMP280 sensor is used to send data from the sensor to the microcontroller. When the microcontroller requests information, such as temperature or pressure readings, the BMP280 transmits that data back through the SDO pin. Essentially, it acts as a communication line, allowing the sensor to share its measurements with the microcontroller for further processing in your project.



The **SDI pin** (Serial Data In) on the BMP280 sensor is used to receive data from the microcontroller in SPI (Serial Peripheral Interface) mode. When the microcontroller sends commands or configuration settings to the BMP280, it does so through the SDI pin. This pin allows the sensor to receive instructions about what data to collect or how to operate, enabling communication between the sensor and the microcontroller.

The **CS pin** (Chip Select) on the BMP280 sensor is used to manage communication in systems where multiple devices share the same connection. When the microcontroller wants to communicate with the BMP280, it pulls the CS pin low to indicate that this particular sensor is selected for data exchange. This prevents confusion by ensuring that only the chosen device responds to commands, while others remain inactive. Essentially, the CS pin acts as a switch, allowing the microcontroller to focus on one device at a time for clear and organized communication.

LED Backpack Counter

Adafruit LED Backpack Counter

The Adafruit LED Backpack Counter is a small, ready-to-use display that shows numbers (and sometimes letters) on its LED digits. It's perfect for projects where you need to display things like scores, timers, counters, or other numerical data.

LED Backpack Counter

Adafruit LED Backpack Counter

The **D pin** corresponds to the SDA line and is responsible for carrying the actual data. This is where the microcontroller sends instructions, like which digits or symbols to display and which segments of the LEDs to light up. This line is bi-directional, meaning that it not only allows the microcontroller to send commands but also enables the LED Backpack to send acknowledgments or other responses back if needed.

The **C pin**, on the other hand, corresponds to the SCL line, which provides the clock signals necessary to synchronize the data transfer. The clock ensures that the data on the D pin is transmitted and received in a well-timed and organized manner, preventing errors during communication.

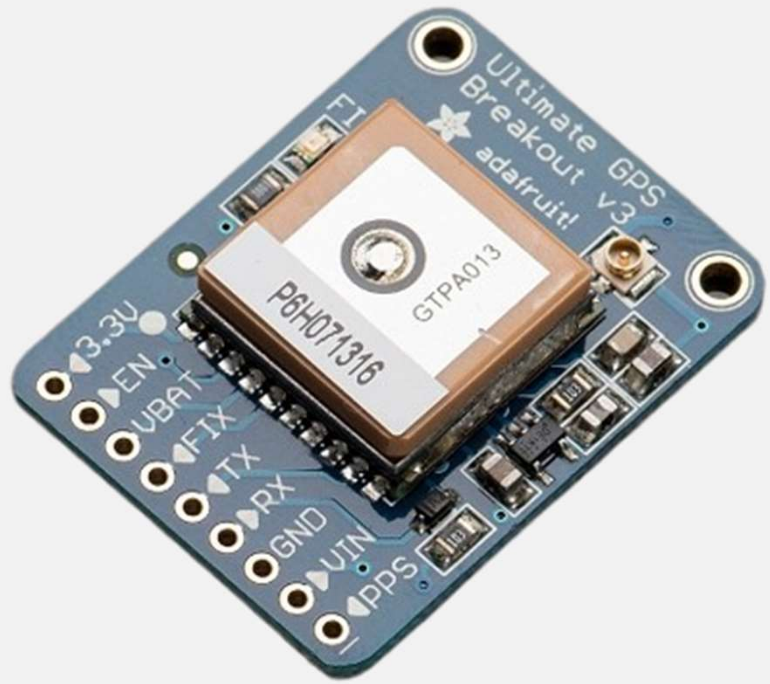
The **+** and **-** pins on the Adafruit LED Backpack are used to supply power to the module. The **+ pin** is the positive power input, also referred to as **VCC**. It provides the voltage needed to power the LED display and its onboard controller chip. The **- pin**, on the other hand, is the ground connection, referred to as **GND**. It serves as the return path for the electrical current, completing the circuit. The **- pin** must be connected to the ground pin of the microcontroller or power source to ensure proper operation.

GPS Module

Adafruit Ultimate GPS Breakout v.3

The **Adafruit Ultimate GPS Breakout v3** is a compact GPS module designed for accurate location tracking and navigation. It integrates easily with microcontrollers and offers a variety of features to enhance project functionality.

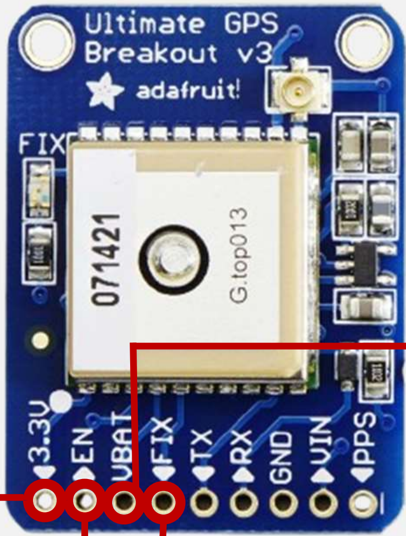
- **Accurate GPS Data:** Provides precise location information, including latitude, longitude, and altitude.Speed and Direction: Calculates speed and movement direction for navigation purposes.
- **NMEA Output:** Outputs data in standard NMEA format for easy integration with microcontrollers like Arduino or Raspberry Pi.
- **Timekeeping:** Offers accurate time data based on GPS signals, including UTC (Coordinated Universal Time).
- **Battery Backup:** Includes a battery backup option to maintain real-time clock and satellite information, ensuring faster GPS fixes after power loss.



GPS Module

Adafruit Ultimate GPS Breakout v.3 Pins

The **3.3V pin** provides a regulated 3.3V output. This pin can be used to power other components or sensors in your project that require a 3.3V power supply. It allows users to draw power directly from the GPS module, simplifying connections and reducing the need for additional power sources.



The **VBAT pin** is used to connect a battery for backup power. This pin allows the GPS module to maintain its real-time clock and satellite information even when the main power is turned off. By connecting a battery to the VBAT pin, the module can quickly acquire GPS signals when power is restored, reducing the time it takes to get a location fix. This feature is particularly useful in battery-operated projects where power may be intermittent.

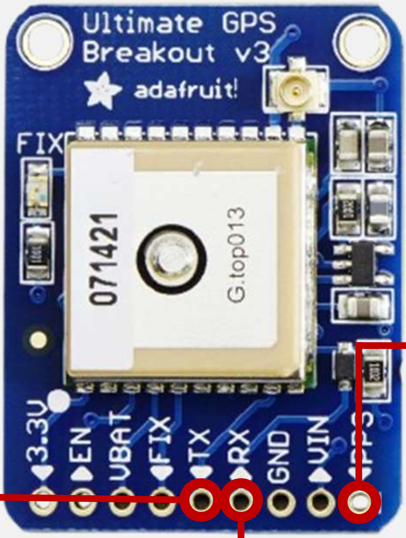
The **EN pin** (Enable pin) is used to enable or disable the GPS module's functionality. When the EN pin is pulled high (connected to a voltage source), the GPS module is activated and begins receiving GPS signals. Conversely, pulling the EN pin low (connecting it to ground) puts the module into a low-power sleep mode, reducing power consumption. This feature is particularly useful in battery-powered projects, allowing users to save energy when the GPS functionality is not needed.

The **FIX pin** serves as an indicator of the GPS module's fix status. When the pin is high, it signifies that the GPS has successfully acquired a valid position fix and is receiving usable GPS data. If the pin is low, it means the GPS is still searching for a signal or has not established a reliable fix. **Status Indicator:** The FIX pin provides a simple way to monitor whether the GPS module is functioning correctly and has a valid location.

GPS Module

Adafruit Ultimate GPS Breakout v.3 Pins

The **TX pin** (Transmit pin) is used to send data from the GPS module to a microcontroller or other devices. When the GPS module has processed location data—such as latitude, longitude, speed, and time—it transmits this information through the TX pin in standard NMEA format. This allows the microcontroller to receive and interpret the GPS data for use in various applications, such as navigation or location tracking. We'll review NMEA formatted data elsewhere.



The **PPS pin** (Pulse Per Second pin) provides a highly accurate timing signal that is synchronized with GPS time.
*** The GPS module may require periodic synchronization with GPS satellites to maintain accurate time and position data, especially after power loss or when it first acquires a signal.

Functions of the PPS Pin:
•**Accurate Timing:** The PPS pin emits a pulse at the start of each second, corresponding precisely to GPS time. This allows for extremely accurate timekeeping, to within a few nanoseconds.
•**Digital Output:** The output is a digital signal, typically transitioning from low to high at the start of each second, making it easy to interface with microcontrollers and other digital systems.

The **RX pin** (Receive pin) is used for receiving data from a microcontroller or other devices. When the microcontroller sends commands or configuration settings to the GPS module, it does so through the RX pin.

*** The **VIN pin** and the **GND pin** have the same functions as previously described for the BMP280 Sensor.

Module II

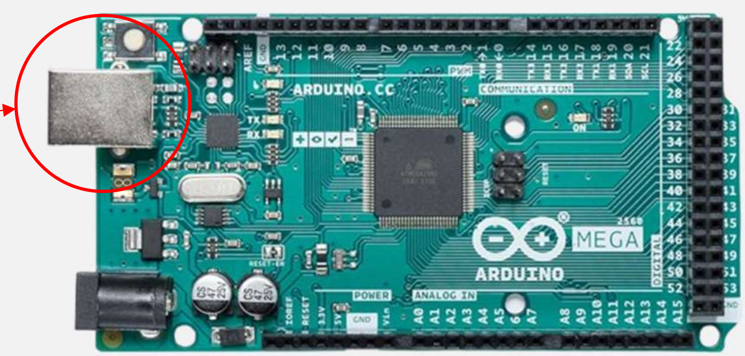
Software Overview – Arduino IDE

Arduino IDE

Arduino IDE (Integrated Development Environment)

Now that we are familiar with the hardware we'll be working with, let us introduce the software.

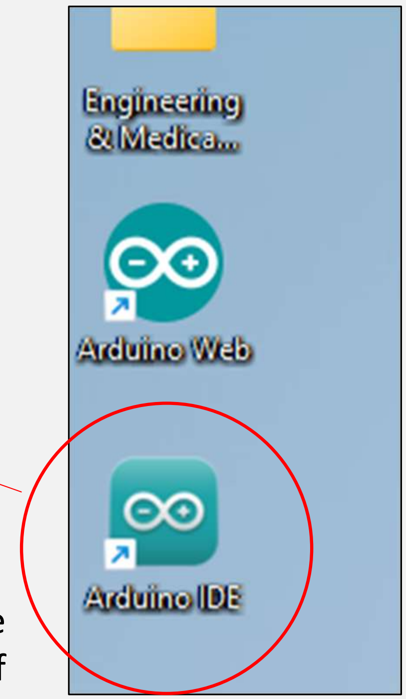
<u>1st Step:</u> Plug in the USB Type B cable into the Arduino board and the computer. This will allow you to communicate to the microcontroller from your computer and vice-versa.



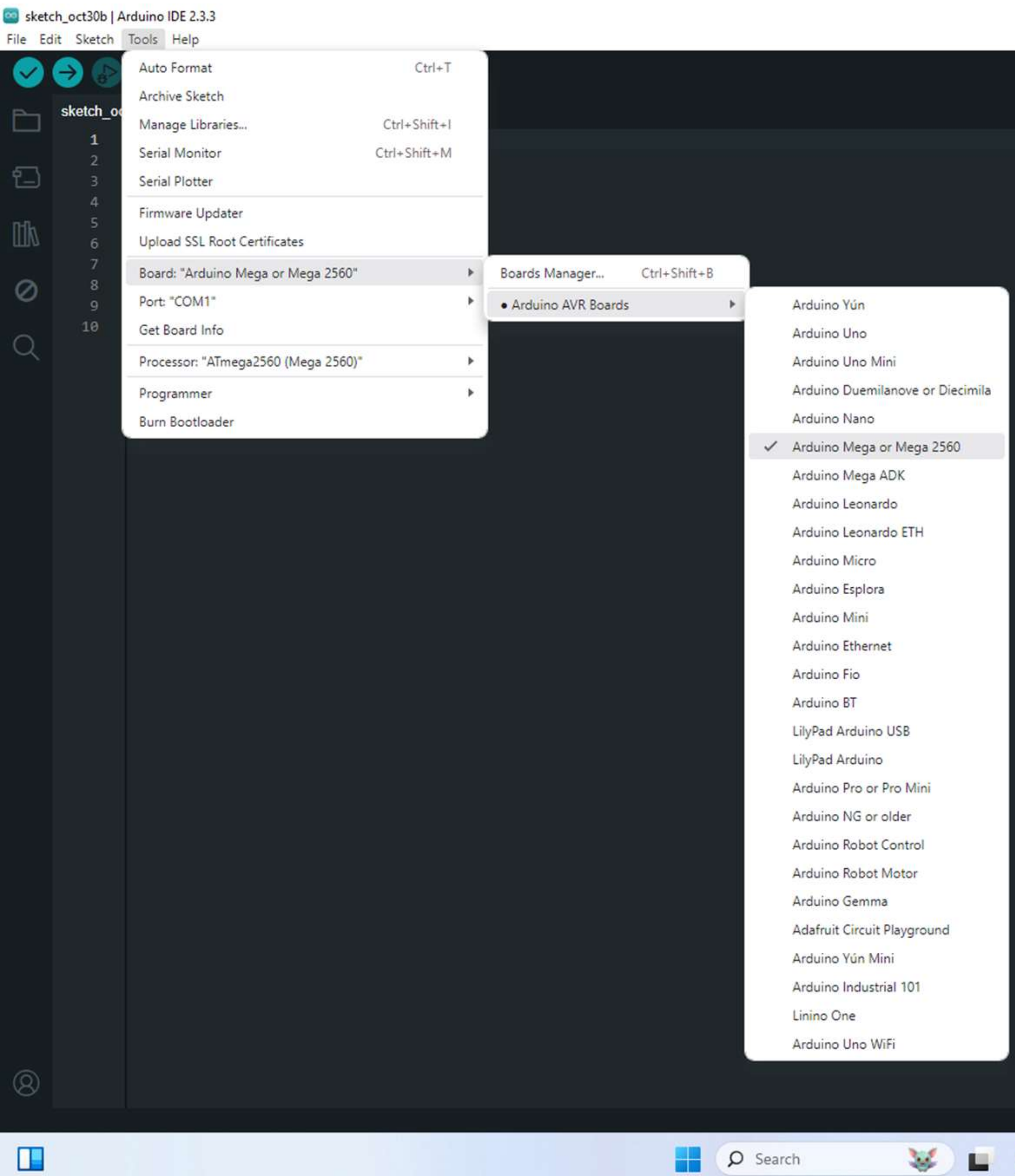
Arduino IDE

Arduino IDE (Integrated Development Environment)

<u>2nd Step:</u> Find the icon on your desktop to open up the Arduino IDE.

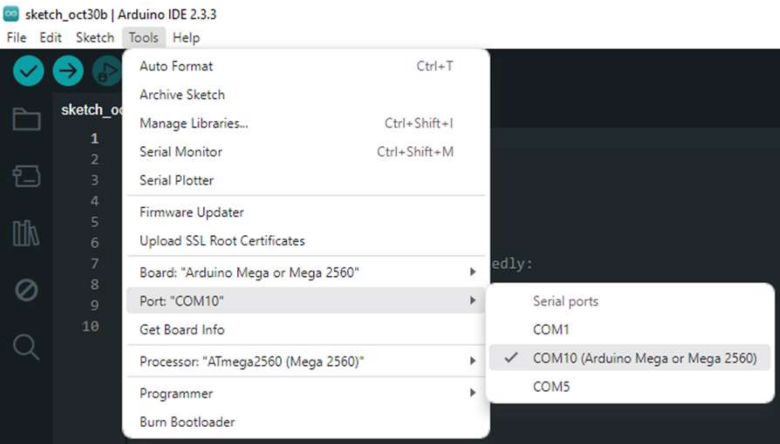


The code you write in the Arduino IDE is called a ***sketch***, and the Arduino compiler handles all the setup to convert it into machine language for the microcontroller. It utilizes a simplified subset of C++ with a few custom libraries simplifying C++ to be more accessible for prototyping and hardware interaction.



Check for proper board selection:

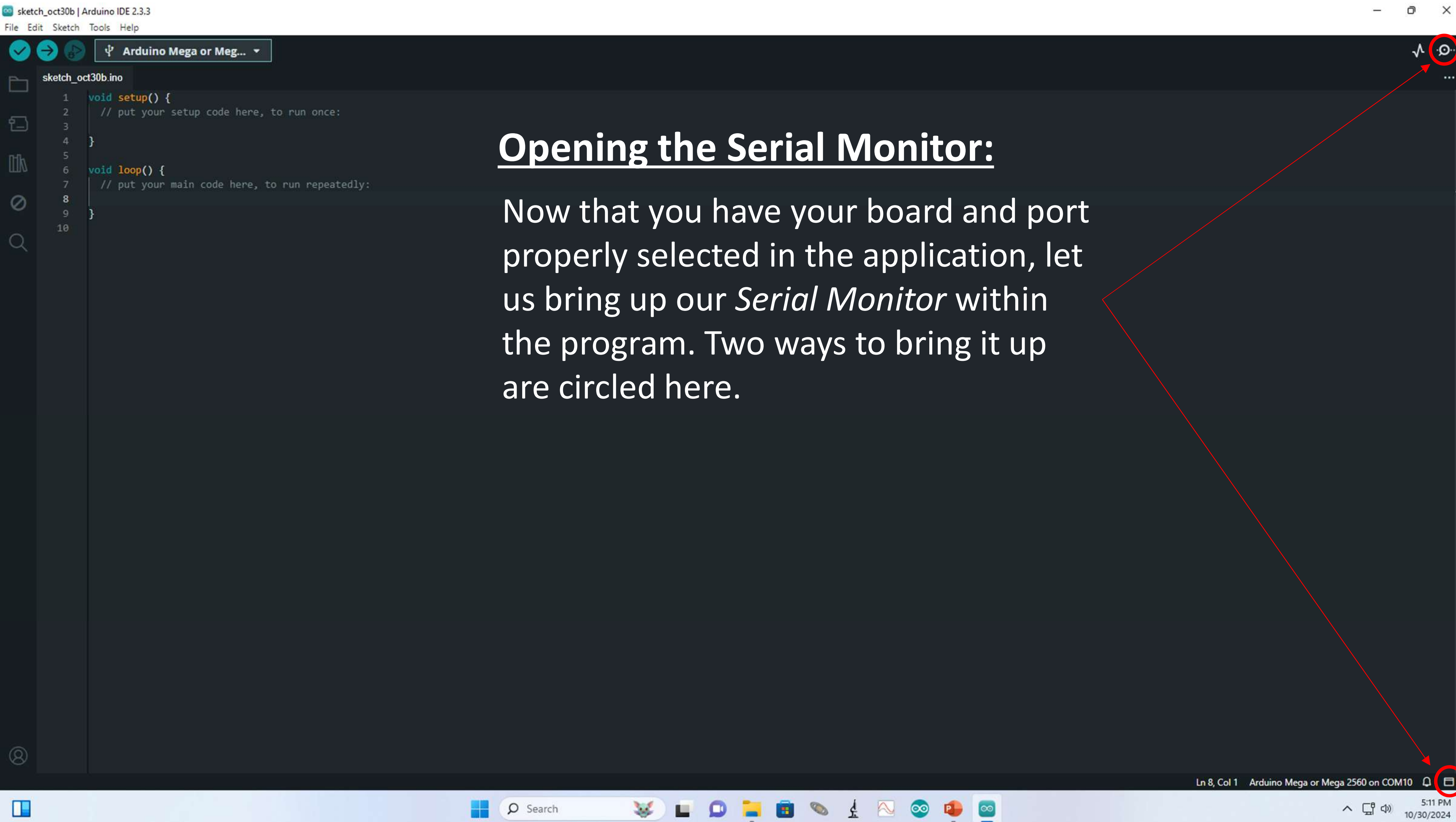
1. Navigate to the toolbar and select ***Tools***.
2. Go to ***Boards*** > ***Arduino AVR Boards***
3. Select the model of the board you are using. In this case, it is the ***Arduino Mega*** *or* ***Mega 2560.***



Check for proper port selection:

1. Navigate to the toolbar and select ***Tools***.
2. Go to ***Ports***.
3. *Select the COM port that shows the model of your microcontroller alongside it.*

*** Occasionally, the security software installed on campus computers will flag the Arduino, blocking its connection. If you have connection issues with the Arduino board, try closing the IDE, unplugging the board and reconnecting it. Using the reset button on the Arduino board may also help.



Opening the Serial Monitor:

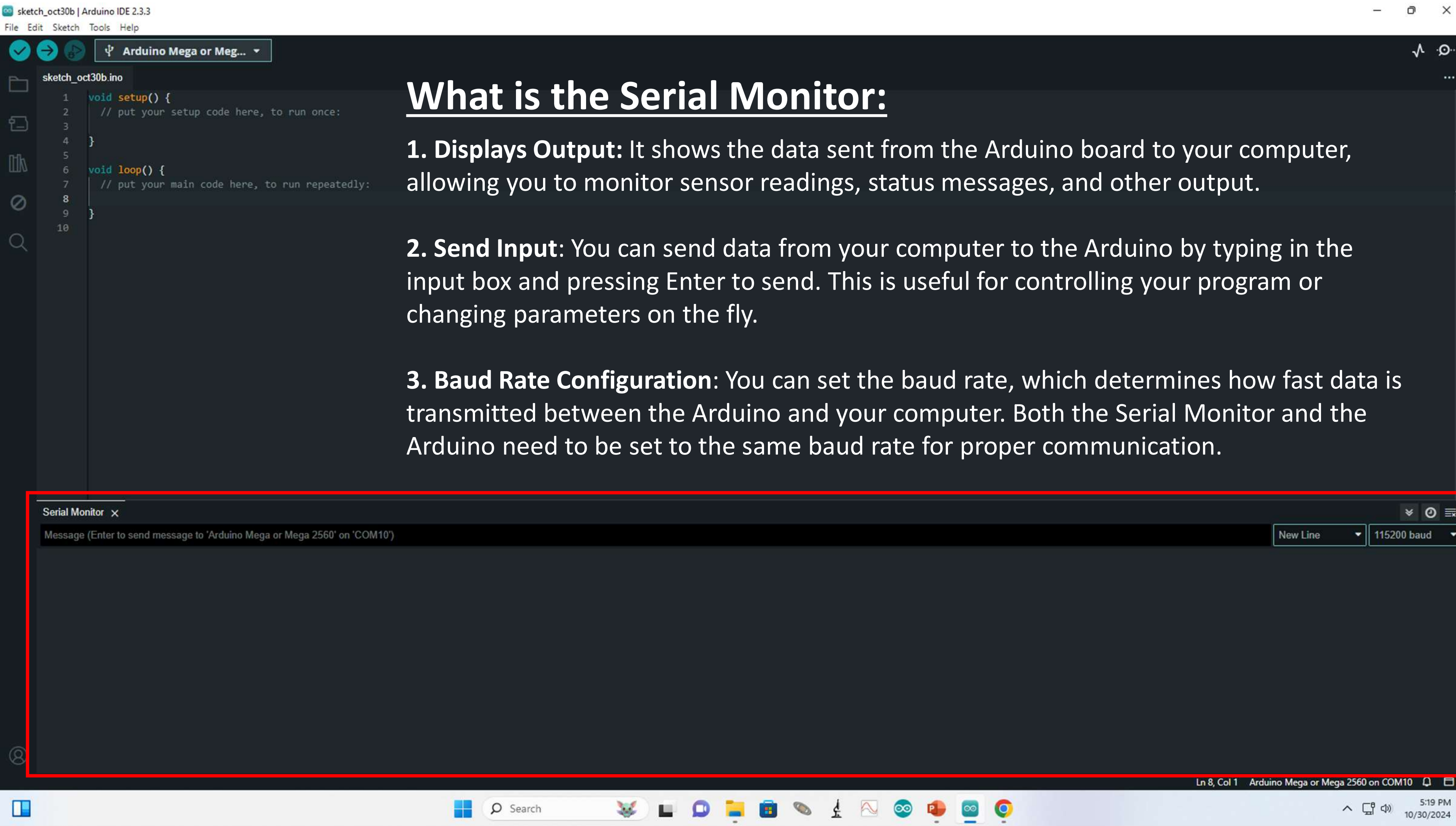
Now that you have your board and port properly selected in the application, let us bring up our *Serial Monitor* within the program. Two ways to bring it up are circled here.

What is the Serial Monitor:

1. Displays Output: It shows the data sent from the Arduino board to your computer, allowing you to monitor sensor readings, status messages, and other output.

2. Send Input: You can send data from your computer to the Arduino by typing in the input box and pressing Enter to send. This is useful for controlling your program or changing parameters on the fly.

3. Baud Rate Configuration: You can set the baud rate, which determines how fast data is transmitted between the Arduino and your computer. Both the Serial Monitor and the Arduino need to be set to the same baud rate for proper communication.



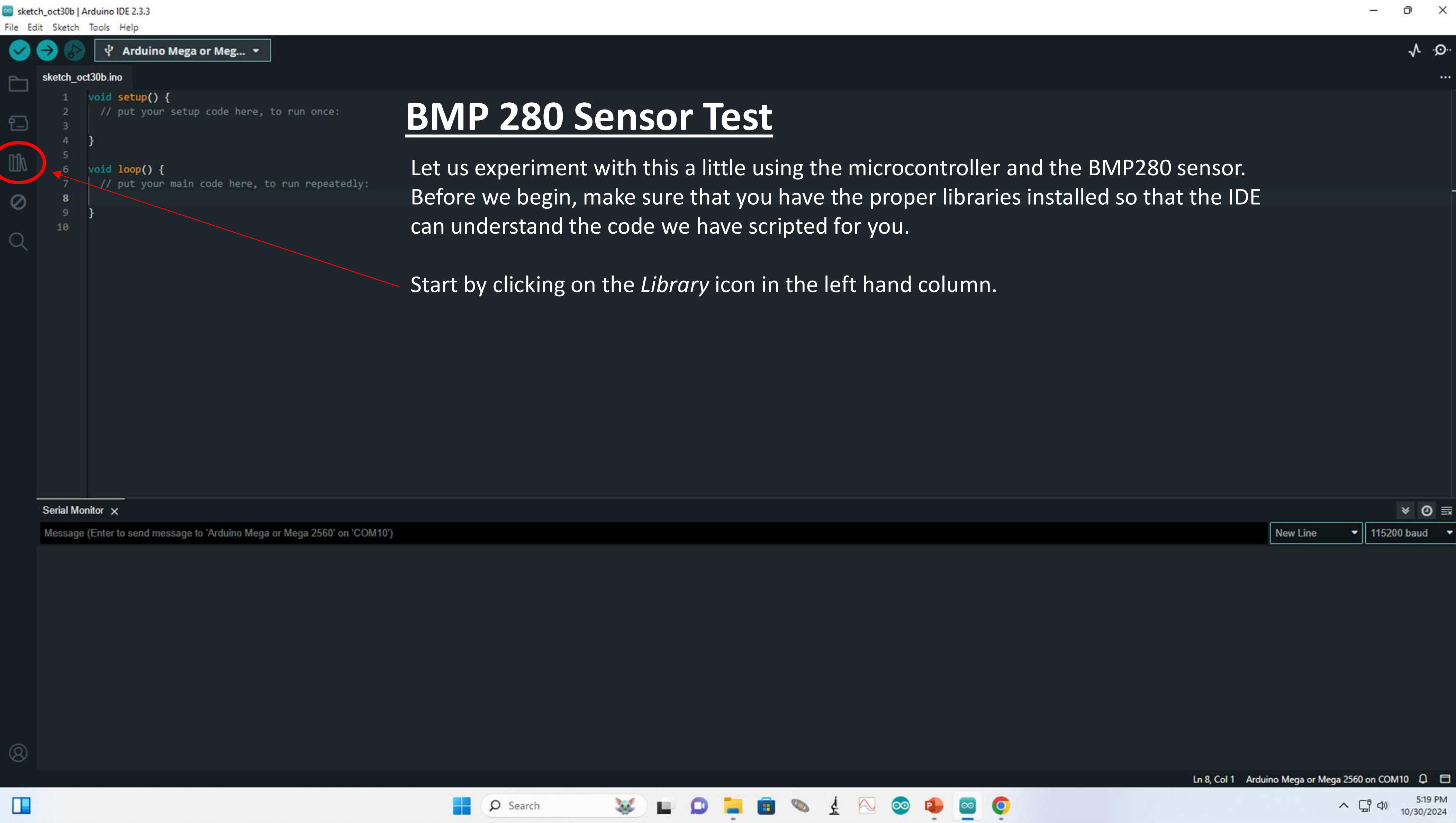
Module III

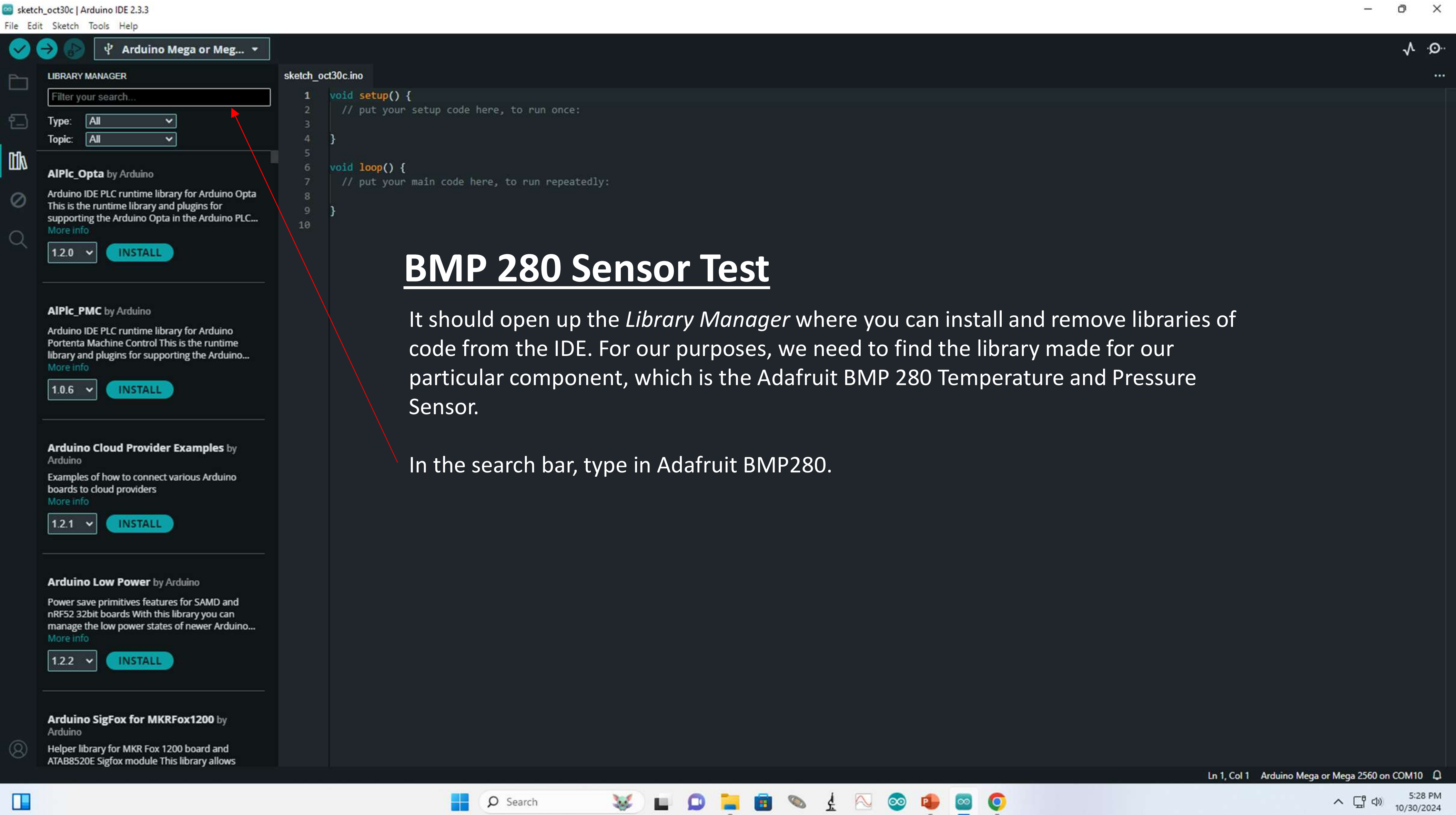
BMP 280 Sensor Test:
Ambient Pressure, Temperature & Altitude

BMP 280 Sensor Test

Let us experiment with this a little using the microcontroller and the BMP280 sensor. Before we begin, make sure that you have the proper libraries installed so that the IDE can understand the code we have scripted for you.

Start by clicking on the *Library* icon in the left hand column.

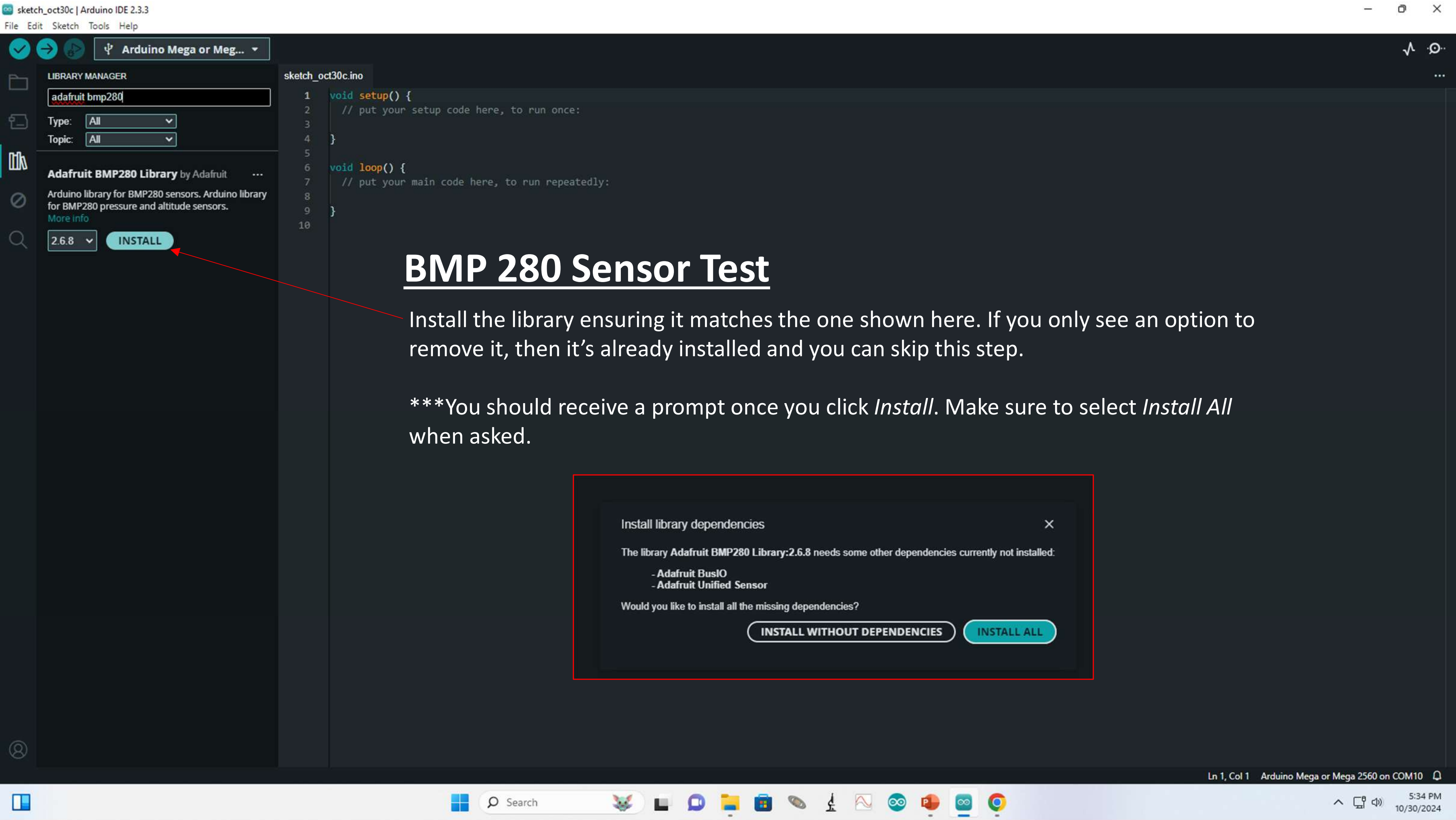




BMP 280 Sensor Test

It should open up the *Library Manager* where you can install and remove libraries of code from the IDE. For our purposes, we need to find the library made for our particular component, which is the Adafruit BMP 280 Temperature and Pressure Sensor.

In the search bar, type in Adafruit BMP280.



BMP 280 Sensor Test

Install the library ensuring it matches the one shown here. If you only see an option to remove it, then it's already installed and you can skip this step.

***You should receive a prompt once you click *Install*. Make sure to select *Install All* when asked.

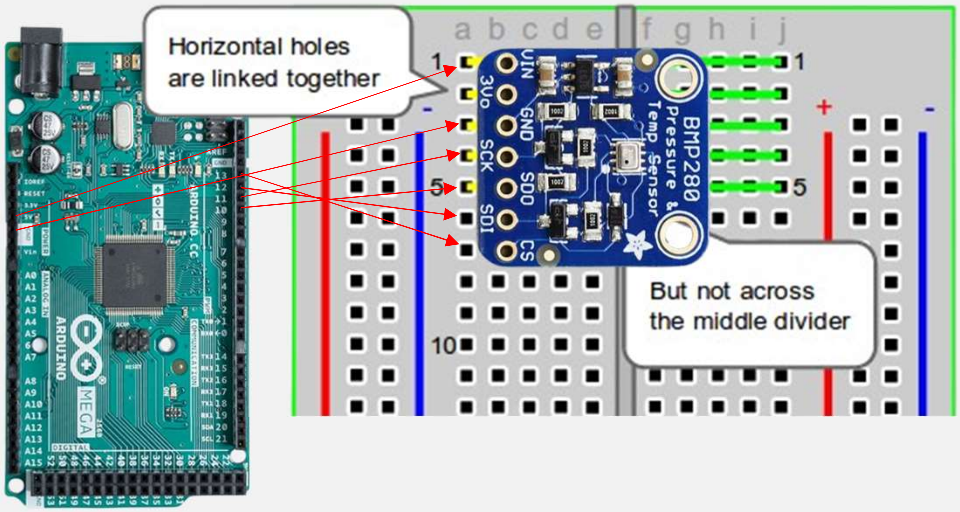
BMP 280 Sensor Test

Using the breadboard

1. Connect the BMP 280 Sensor to the breadboard.
2. To connect the BMP280 to the Arduino, the jumper wire must be placed into the breadboard slot next to the sensor's pin.
3. Once your jumper wires are connected to the BMP280, make the following connections to the Arduino:

Connections

Arduino	BMP 280
(Power) 5V	VIN
(Power) GND	GND
(PWM) Pin 10	CS
(PWM) Pin 11	SDI
(PWM) Pin 12	SDO
(PWM) Pin 13	SCK



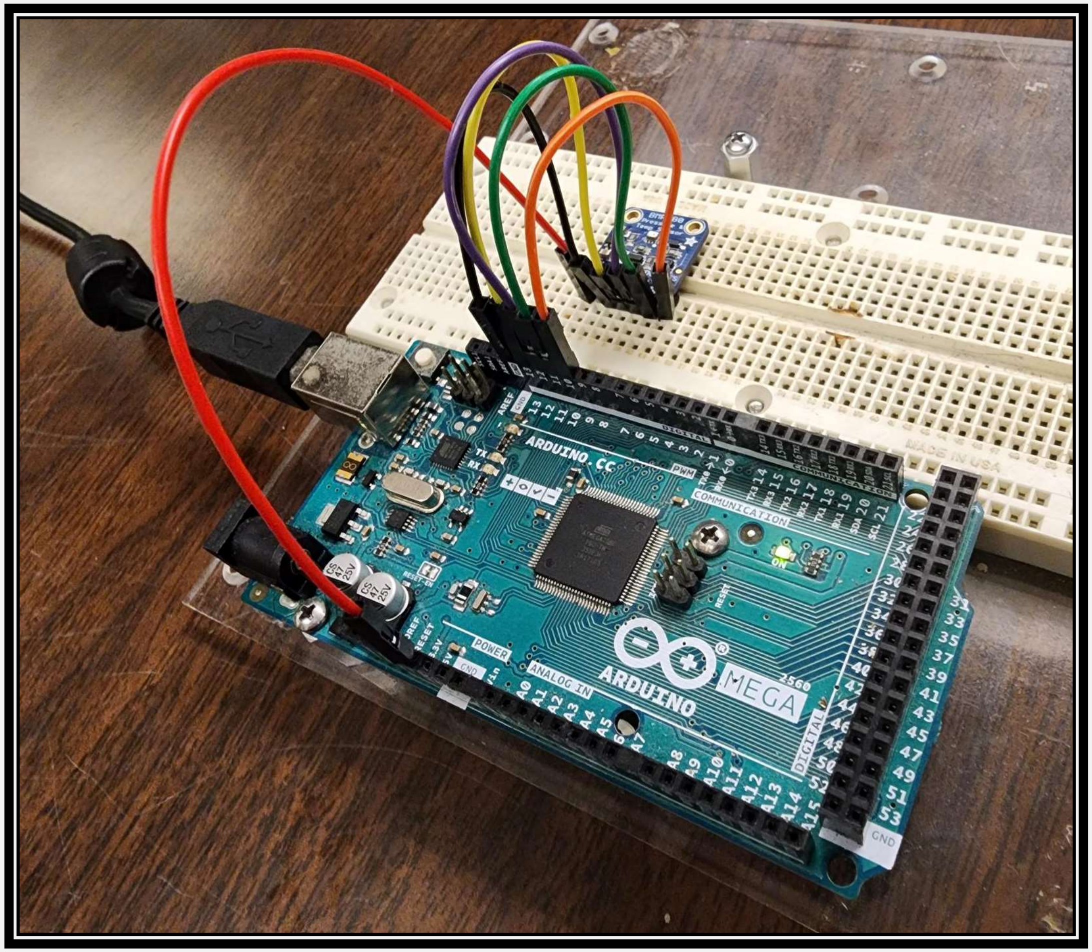
BMP 280 Sensor Test

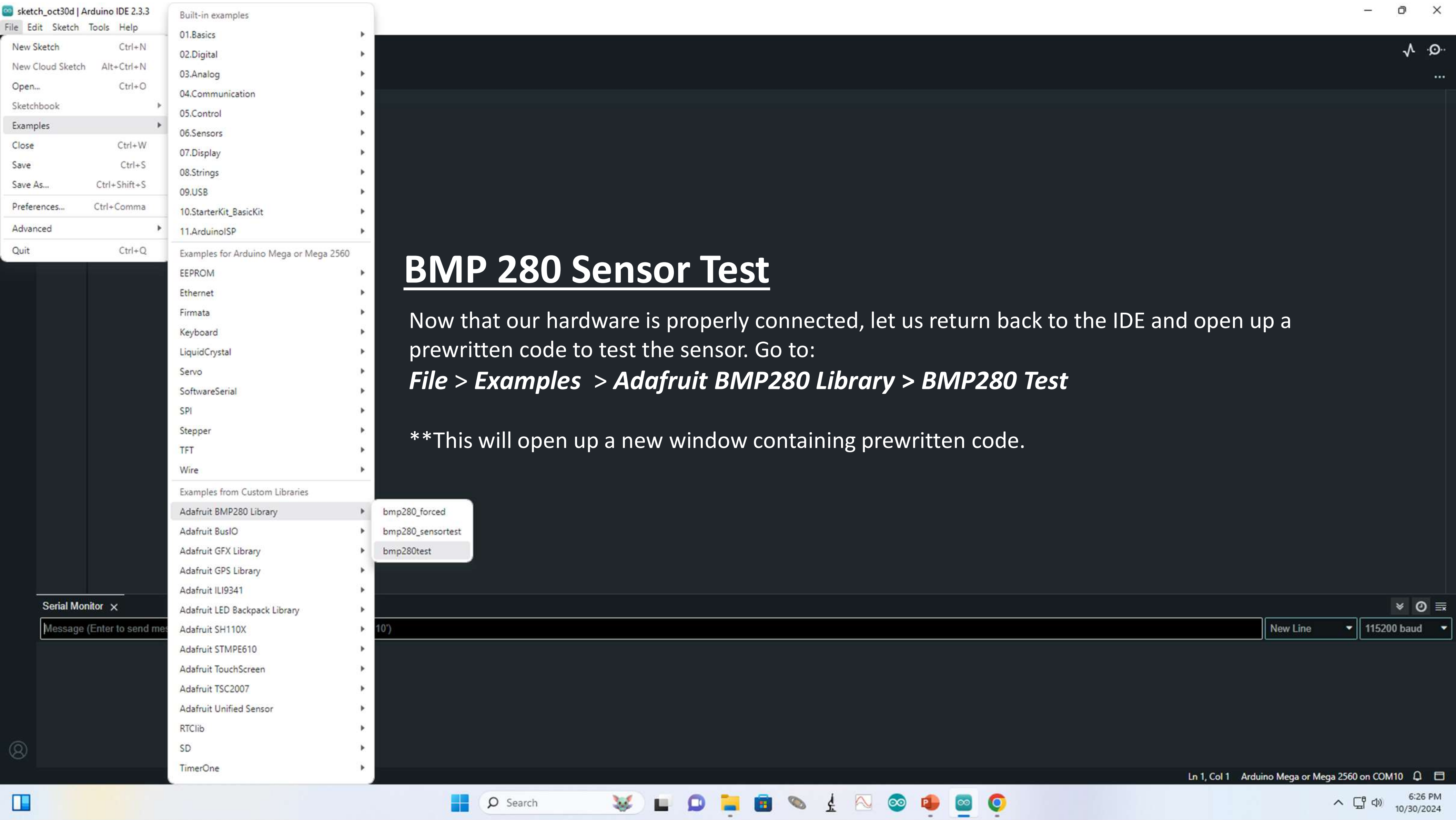
Setup

Here is what your setup should look like!

Connections

Arduino	BMP 280
(Power) 5V	VIN
(Power) GND	GND
(PWM) Pin 10	CS
(PWM) Pin 11	SDI
(PWM) Pin 12	SDO
(PWM) Pin 13	SCK





BMP 280 Sensor Test

Now that our hardware is properly connected, let us return back to the IDE and open up a prewritten code to test the sensor. Go to:

File* > *Examples* > *Adafruit BMP280 Library* > *BMP280 Test

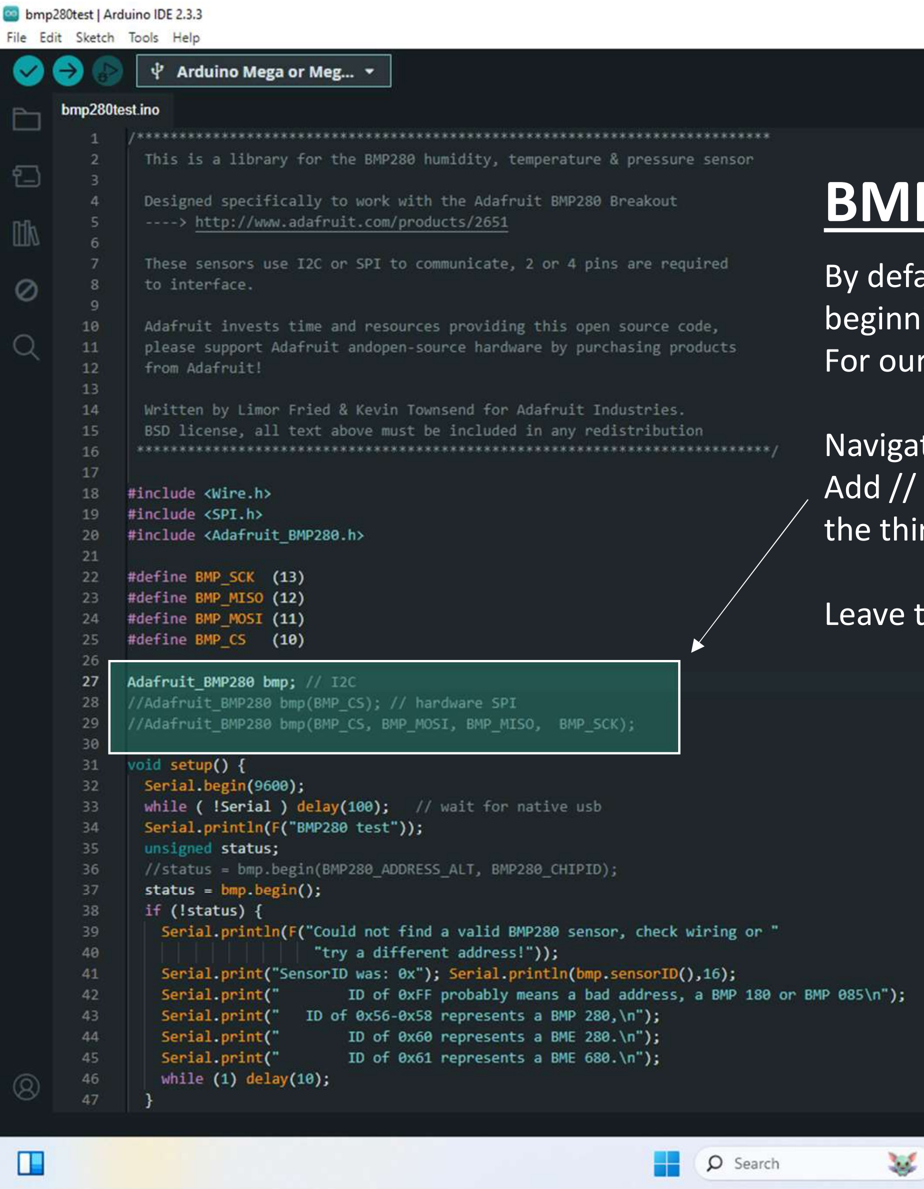
**This will open up a new window containing prewritten code.

BMP 280 Sensor Test

By default, there is some code that is "commented out". This is done by beginning the line with a double forward slash (i.e. //), effectively disabling it. For our purposes here, we will need to enable them.

Navigate to the text highlighted here.
Add // to the beginning of the first line, disabling it, and remove the // from the third line to enable that one.

Leave the second line alone.



Reminder: You can use the button in the top right to pull up the serial monitor.

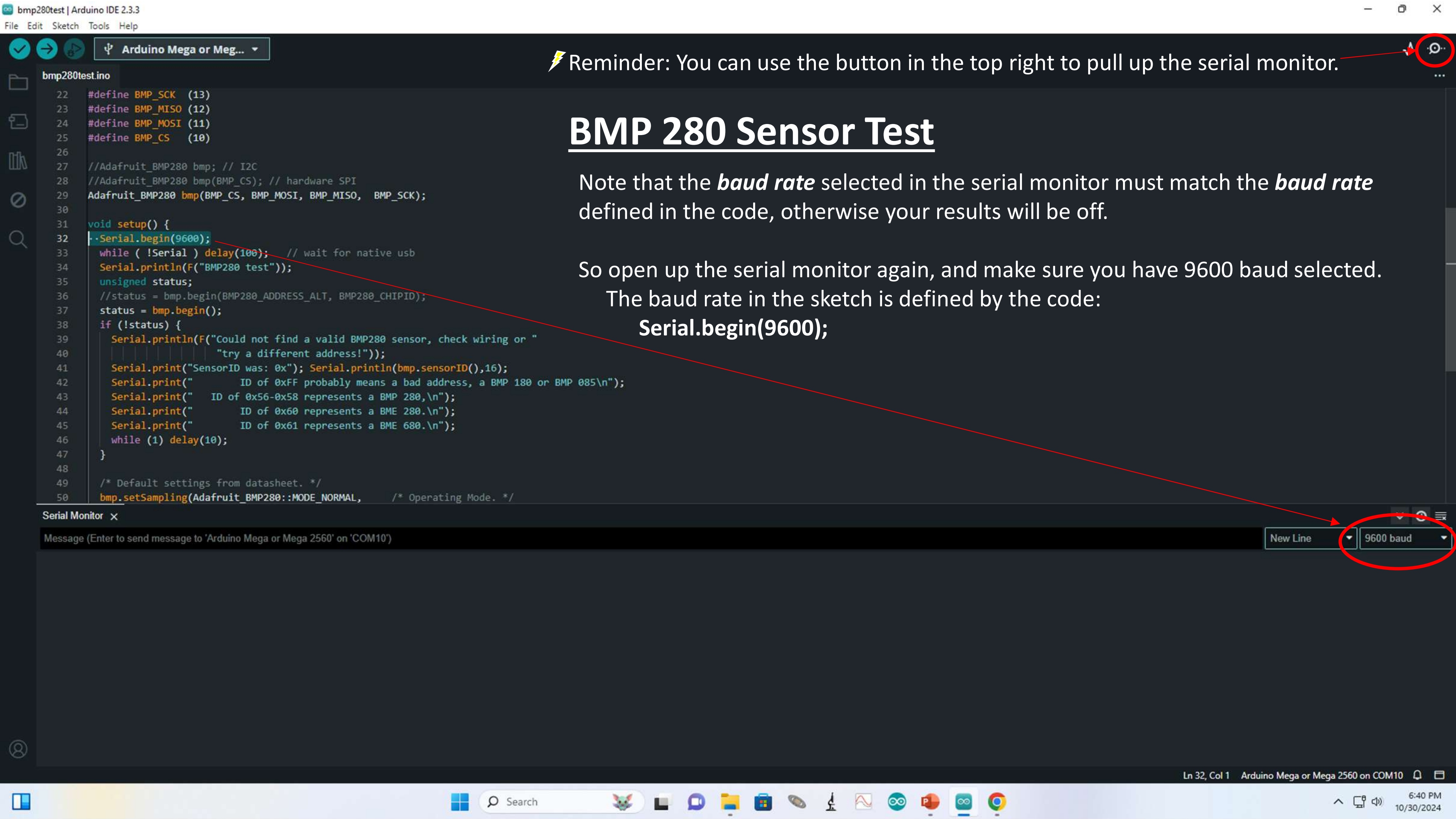
BMP 280 Sensor Test

Note that the ***baud rate*** selected in the serial monitor must match the ***baud rate*** defined in the code, otherwise your results will be off.

So open up the serial monitor again, and make sure you have 9600 baud selected.

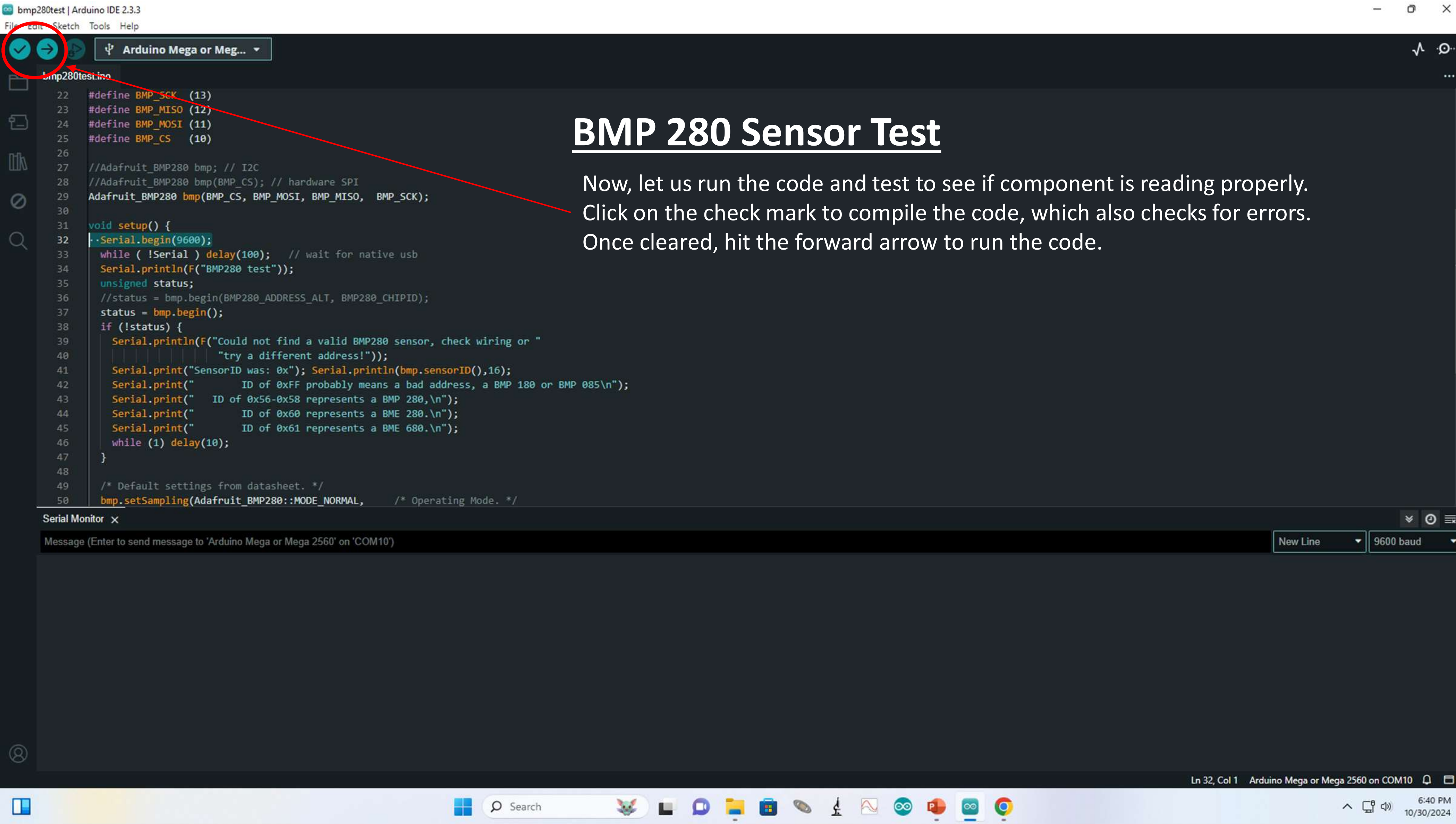
The baud rate in the sketch is defined by the code:

Serial.begin(9600);



BMP 280 Sensor Test

Now, let us run the code and test to see if component is reading properly. Click on the check mark to compile the code, which also checks for errors. Once cleared, hit the forward arrow to run the code.

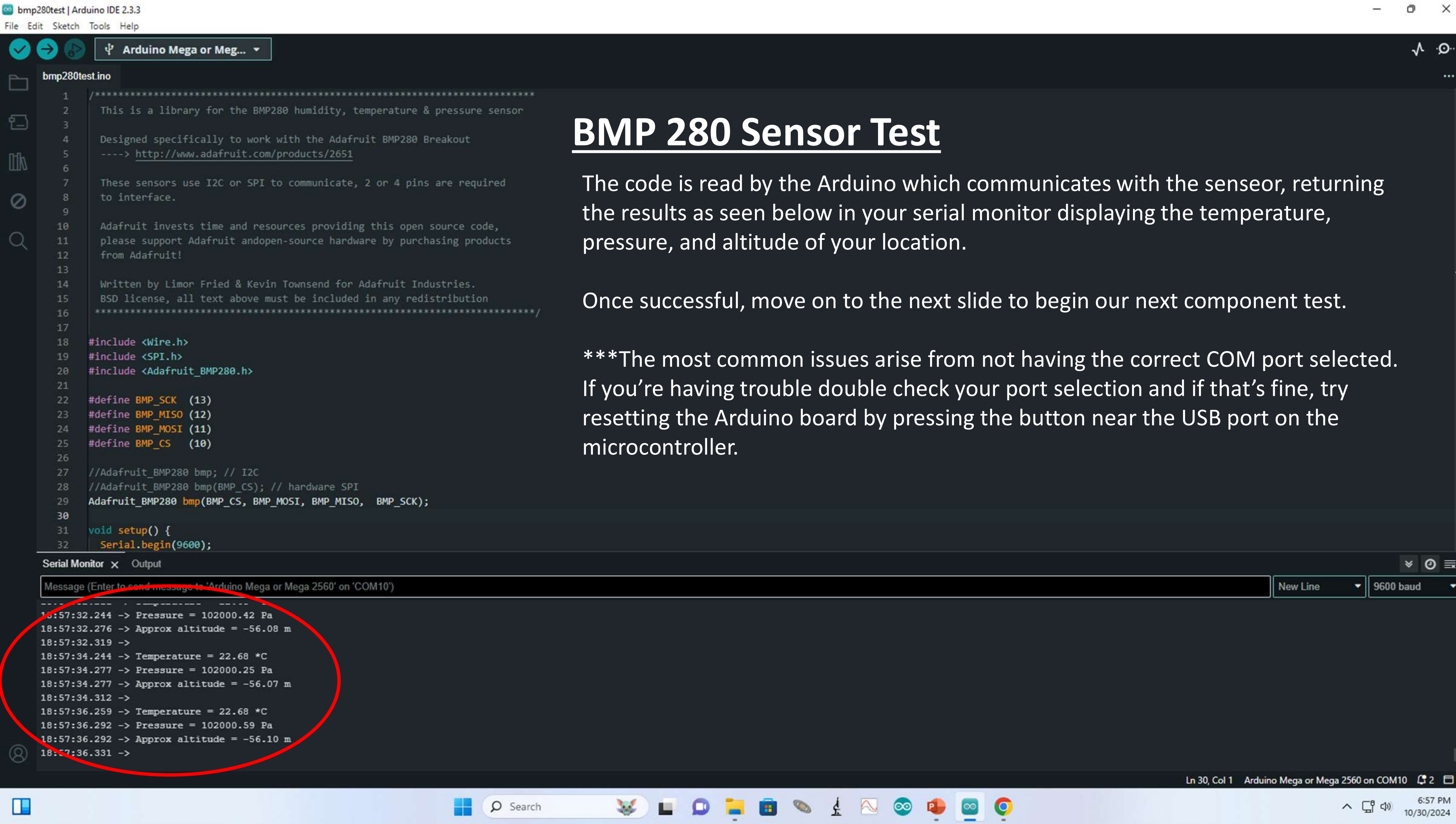


BMP 280 Sensor Test

The code is read by the Arduino which communicates with the senseor, returning the results as seen below in your serial monitor displaying the temperature, pressure, and altitude of your location.

Once successful, move on to the next slide to begin our next component test.

***The most common issues arise from not having the correct COM port selected. If you’re having trouble double check your port selection and if that’s fine, try resetting the Arduino board by pressing the button near the USB port on the microcontroller.

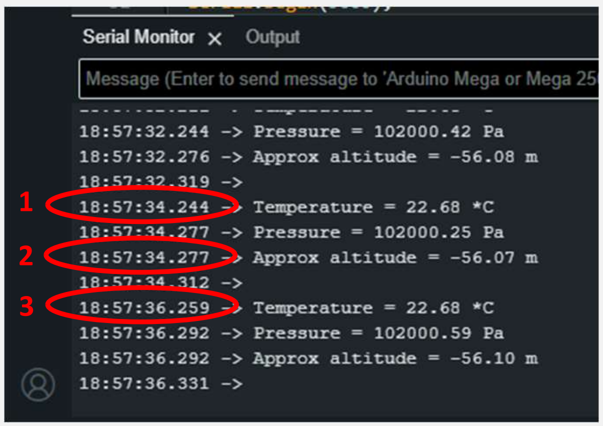


BMP 280 Sensor Test

Identifying Data Transmission Speed

There are a few pieces of information you can gather from this test:

(a) temperature, pressure, and altitude of your location
(b) the length of time it takes to send a set of data (temperature, pressure, & altitude)
(c) how often it sends this data



To find (b) the length of time it takes to send a set of data, subtract the timestamps of the first and last data points from the set. (1 from 2)

34.277 – 34.244 =.033 seconds to transmit a complete set of data (33 ms)

To find (c) how often it sends data, subtract the timestamps of the first items from two consecutive sets of data. (1 from 3)

36.259 – 34.244 = 2.015 seconds between data transmission

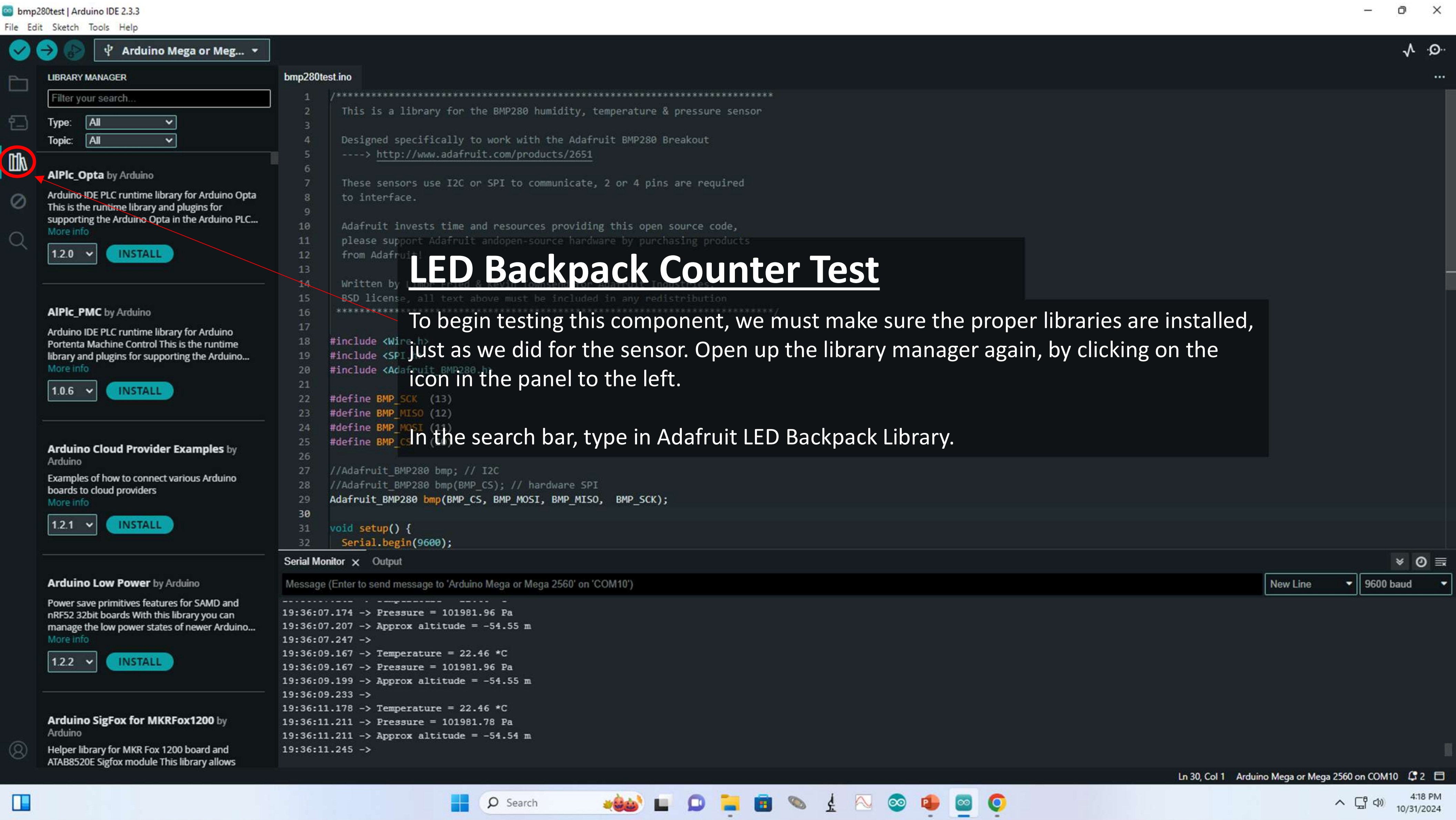
Module IV

LED Backpack Counter Test w/ Arduino Generated Pulses

LED Backpack Counter Test

To begin testing this component, we must make sure the proper libraries are installed, just as we did for the sensor. Open up the library manager again, by clicking on the icon in the panel to the left.

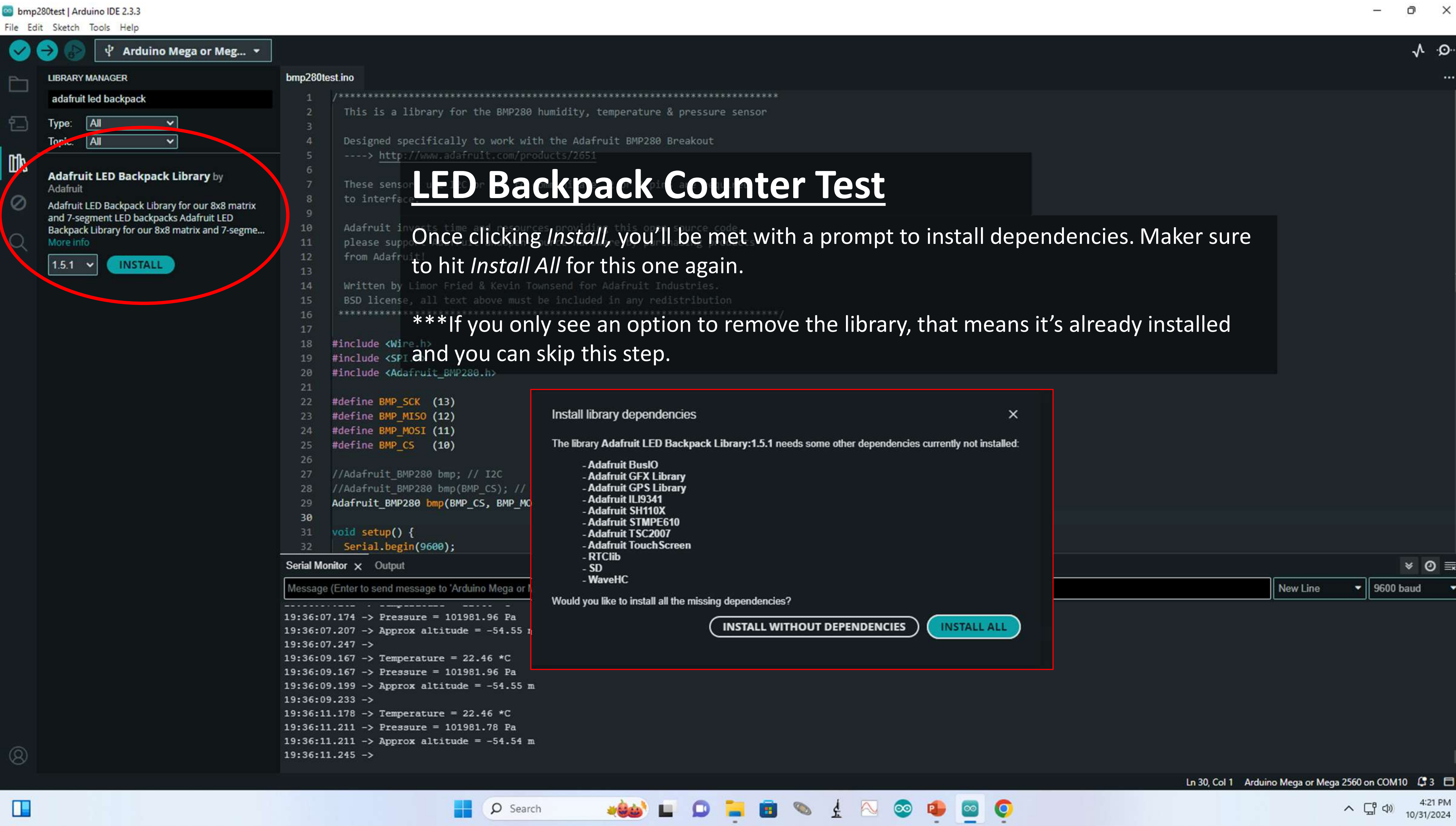
In the search bar, type in Adafruit LED Backpack Library.



LED Backpack Counter Test

Once clicking *Install*, you'll be met with a prompt to install dependencies. Maker sure to hit *Install All* for this one again.

***If you only see an option to remove the library, that means it's already installed and you can skip this step.



LED Backpack Counter

Testing the LED Backpack Counter

1. Now, let us connect the LED Backpack counter to the breadboard as you did with the sensor.
2. To connect the LED Backpack Counter to the Arduino, the jumper wire must be placed into the breadboard slot next to the sensor's pin.
3. Connect a jumper wire to each of the pins on the counter. Once your jumper wires are connected to the LED Backpack Counter, make the following connections to the Arduino:

Connections

Arduino	LED Counter
(Power) 5V	+
(Power) GND	-
(Comm) SDA	D
(Comm) SCL	C

*** In parentheses are the sections of the Arduino in which those pins are located.
*** The LED won't light up until you run the code.

LED Backpack Counter

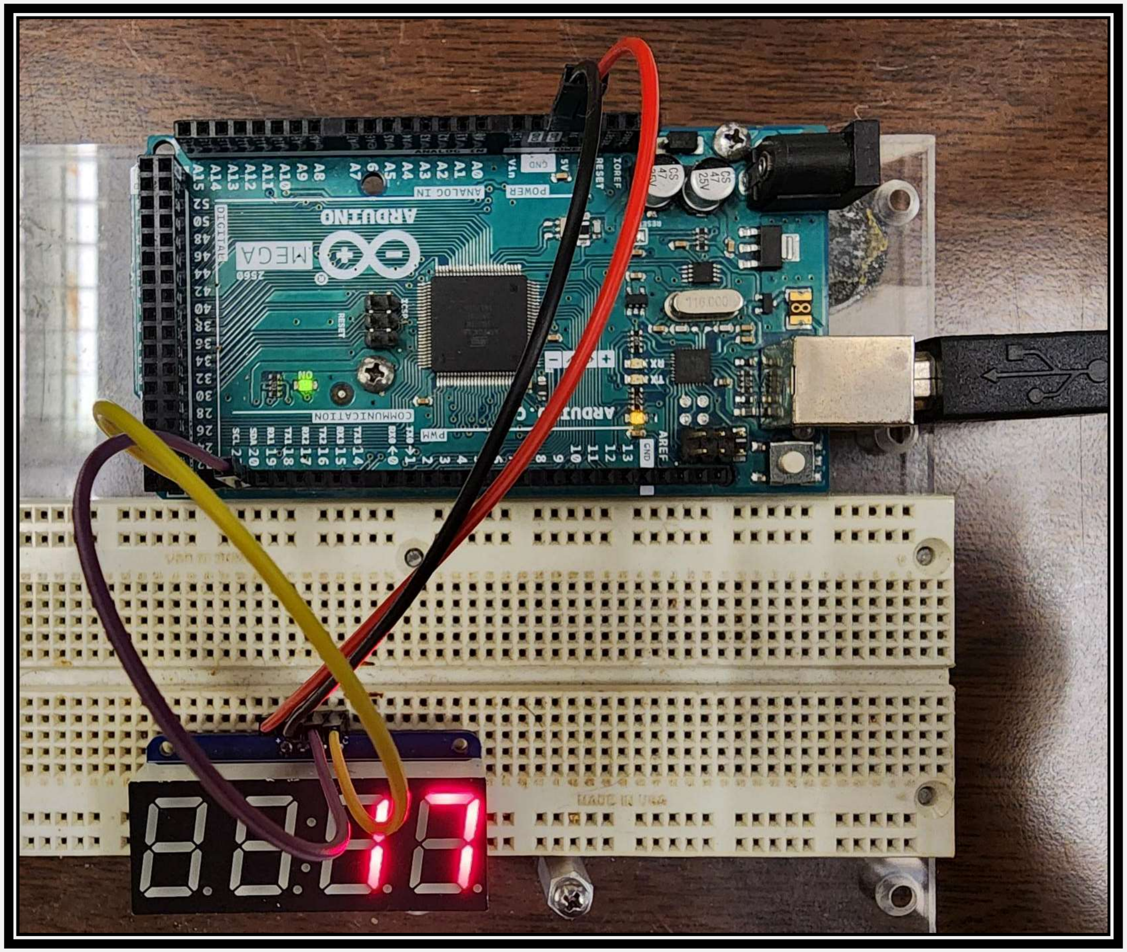
Setup

Here is what your setup should look like!

Connections

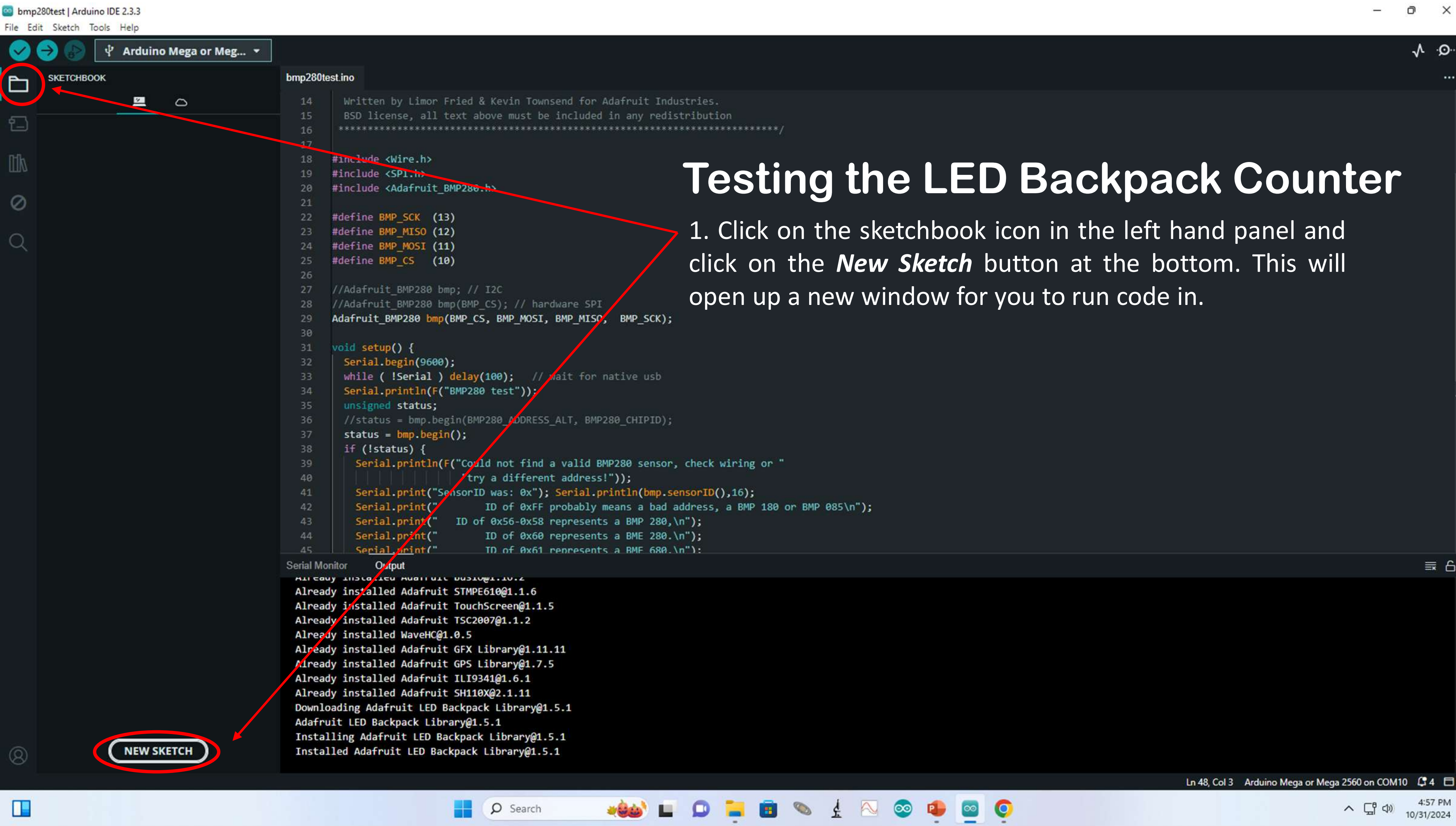
Arduino	LED Counter
(Power) 5V	+
(Power) GND	-
(Comm) SDA Pin 20	D
(Comm) SCL Pin 21	C

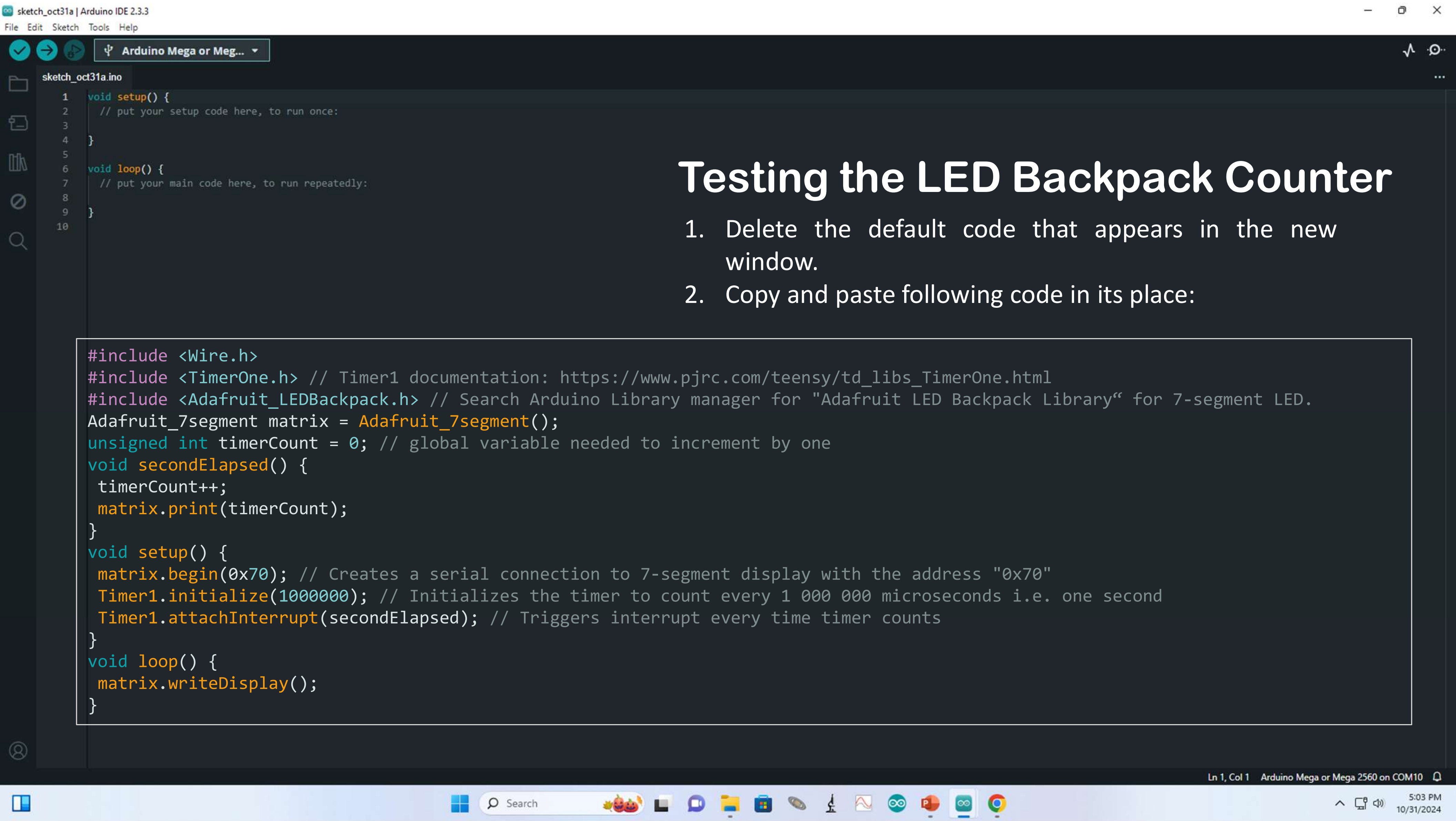
*Keep in mind your LED won't light up until you run the code.



Testing the LED Backpack Counter

1. Click on the sketchbook icon in the left hand panel and click on the ***New Sketch*** button at the bottom. This will open up a new window for you to run code in.

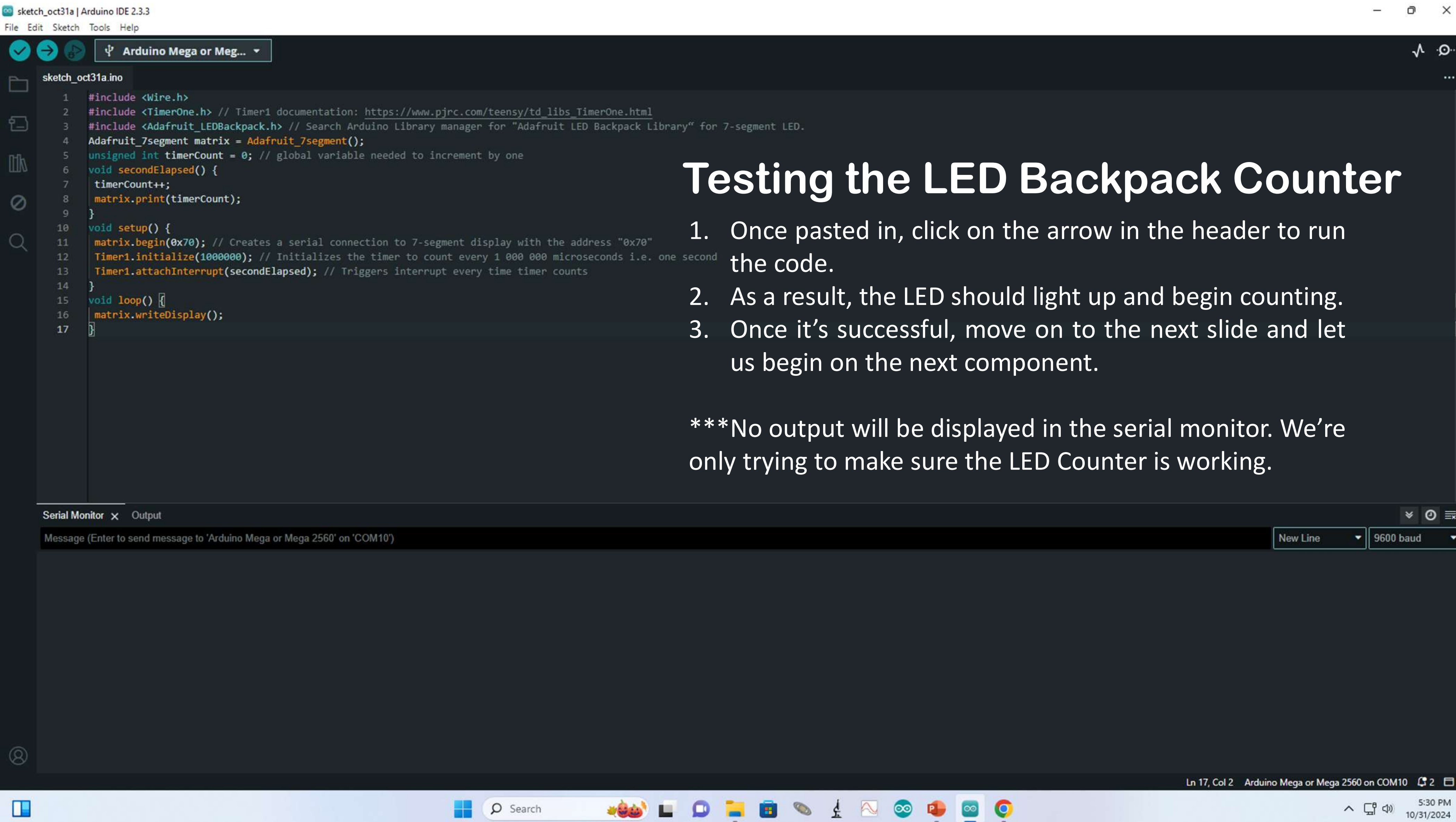




Testing the LED Backpack Counter

1. Delete the default code that appears in the new window.
2. Copy and paste following code in its place:

```
#include <Wire.h>
#include <TimerOne.h> // Timer1 documentation: https://www.pjrc.com/teensy/td_libs_TimerOne.html
#include <Adafruit_LEDBackpack.h> // Search Arduino Library manager for "Adafruit LED Backpack Library“ for 7-segment LED.
Adafruit_7segment matrix = Adafruit_7segment();
unsigned int timerCount = 0; // global variable needed to increment by one
void secondElapsed() {
 timerCount++;
 matrix.print(timerCount);
}
void setup() {
 matrix.begin(0x70); // Creates a serial connection to 7-segment display with the address "0x70"
 Timer1.initialize(1000000); // Initializes the timer to count every 1 000 000 microseconds i.e. one second
 Timer1.attachInterrupt(secondElapsed); // Triggers interrupt every time timer counts
}
void loop() {
 matrix.writeDisplay();
}
```



Testing the LED Backpack Counter

1. Once pasted in, click on the arrow in the header to run the code.
2. As a result, the LED should light up and begin counting.
3. Once it's successful, move on to the next slide and let us begin on the next component.

***No output will be displayed in the serial monitor. We're only trying to make sure the LED Counter is working.

Module V

Arduino Timer Testing
w/ Arduino Generated Pulse

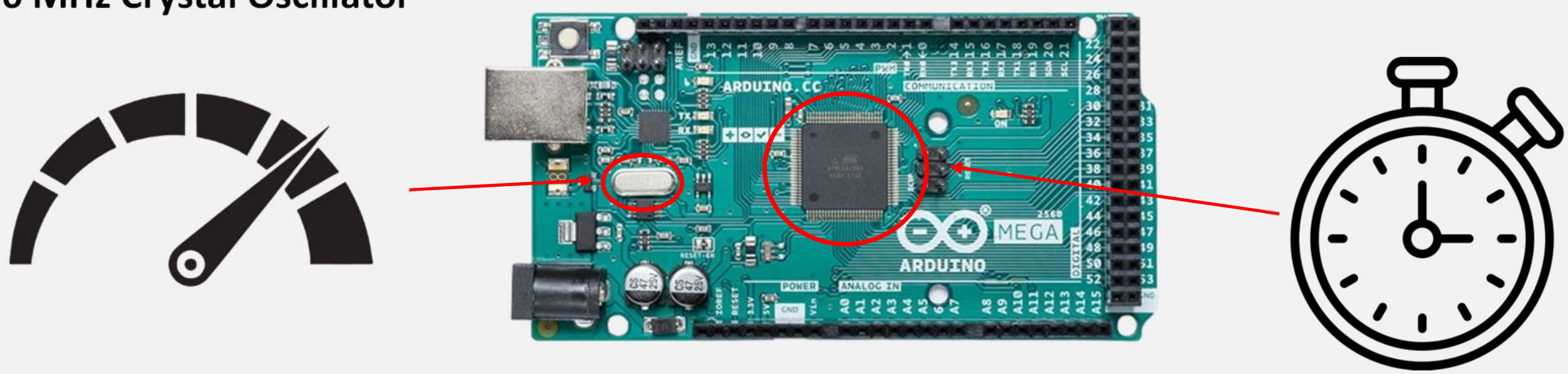
Key Concepts

Arduino Components

Let’s take a step back now and go over two components of the Arduino board and critical concepts that you’ll need to grasp before moving on. Those are the:

16 MHz Crystal Oscillator

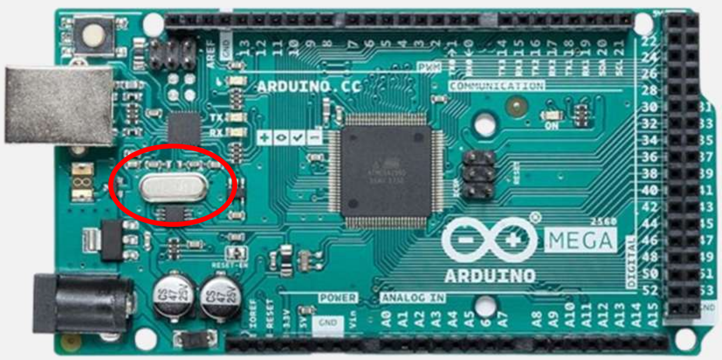
16-Bit Timer



Key Concepts

16 MHz Frequency Crystal Oscillator

The ATmega2560 is operating with a 16MHz (megahertz) frequency clock oscillator, meaning the microcontroller can perform up to 16 million operations per second, or 16 million clock cycles each second. A clock cycle, or “tick”, is the basic unit of time for the microcontroller. The frequency of the oscillator determines the time period for each tick.



Theoretical Timer Tick Period

$$\frac{\text{1 second}}{\text{16,000,000 ticks}} = \text{62.5 nanoseconds per tick}$$

* This formula shows us how we arrive at the theoretical time it takes for the Arduino clock to “tick” once. It doesn’t tick once a second like a normal clock, it would, theoretically, tick once every 62.5 nanoseconds.

*****However, due to temperature fluctuations in the environment, aging effects of the crystal, and programming with interrupts in the code, the Timer Tick Period will not match what we would expect as outlined above and so tests are needed to determine the True Timer Tick Period, or at least a more accurate one.**

Key Concepts

Timers w/ Binary Systems

The timer on board the ATMega 2560 is using a 16-bit counter which utilizes a binary number system The binary number system is a method of representing numbers using only two symbols: **0** and **1**. It is the foundation of all modern computing systems because it aligns well with digital electronics, where circuits have two states: ON (1) and OFF (0).

Binary Place	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
Value	32,768	16,384	8,192	4,096	2,048	1,024	512	256	128	64	32	16	8	4	2	1
Binary Number	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Binary to Decimal

1111111111111111 = 65,535

A 16-bit counter with all bits set to 1 has a maximum value of 65,535. This is important because, with a 16-bit binary system, the counter can only count up to this number. However, our clock oscillator operates at 16,000,000 ticks per second, far exceeding the counter's capacity within a single cycle.

Key Concepts

Decimal Numbering System

The Arduino uses a binary number system but as a comparison, let’s look at how we use numbers using the decimal number system. In the decimal number system, each digit in a number represents a different place value depending on its position. Let's break down the number **9,432.**

9,432

Place	10^3	10^2	10^1	10^0
Value	1,000	100	10	1
Digit	9	4	3	2

The rightmost digit (2) is in the **ones** place. This means it represents:
$2 \times 10^0 = 2 \times 1 = 2$
The next digit (3) is in the **tens** place. This means it represents:
$3 \times 10^1 = 3 \times 10 = 30$
The next digit (4) is in the **hundreds** place. This means it represents:
$4 \times 10^2 = 4 \times 100 = 400$
The leftmost digit (9) is in the **thousands** place. This means it represents:
$9 \times 10^3 = 9 \times 1000 = 9{,}000$

When we add up the values, we get the total for the number it represents:

$$\mathbf{9{,}000 + 400 + 30 + 2 = 9{,}432}$$

Key Concepts

Binary Numbering Systems

The binary system works much the same way as the decimal numbering system only that the values of the positions are no longer powers of 10, but powers of 2. A limitation for the binary system is that it only can use two digits, 0 and 1 for each position. This means that even though this number may look like 1,011 from the decimal system it is not equal in value. Note also, binary numbers do not use comma separators.

<u>**1011**</u>

Place	2^3	2^2	2^1	2^0
Value	8	4	2	1
Digit	1	0	1	1

The rightmost digit (1) is in the **ones** place. This means it represents:
$1 \times 2^0 = 1 \times 1 = 1$
The next digit (1) is in the **twos** place. This means it represents:
$1 \times 2^1 = 1 \times 2 = 2$
The next digit (0) is in the **fours** place. This means it represents:
$0 \times 2^2 = 0 \times 4 = 0$
The leftmost digit (1) is in the **eights** place. This means it represents:
$1 \times 2^3 = 1 \times 8 = 8$

When we add up the values, we get the total for the number it represents:

8 + 0 + 2 + 1 = 11 1011 = 11

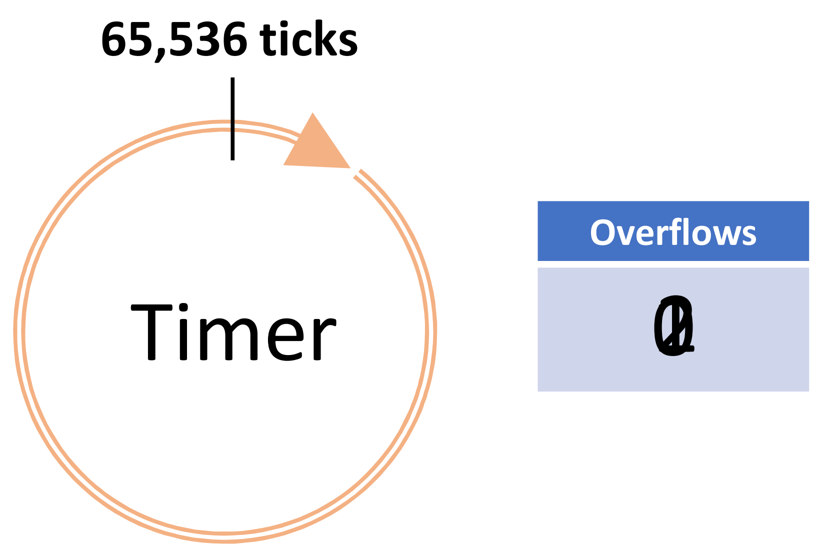
Key Concepts

Working Around Limitations Using Code

To address the limitation of only being able to count up to 65,535, we wrote code to keep track of how many times the counter reaches its maximum value of 65,535. When the counter hits this limit, it resets to 0 and starts counting again. Each time this happens, we increment a variable in the code called **"Overflows"**. For example, two overflows would correspond to a total count of **65,536 × 2.**

***Notice, we multiply the *Overflows* value by **65,536**. This is because the timer treats the value 0 as a tick.

To watch the animation, click the *Animations* Tab from the ribbon above, and click Preview to see how the timer affects the *Overflows* variable.



Identifying True Arduino Timer Tick Period

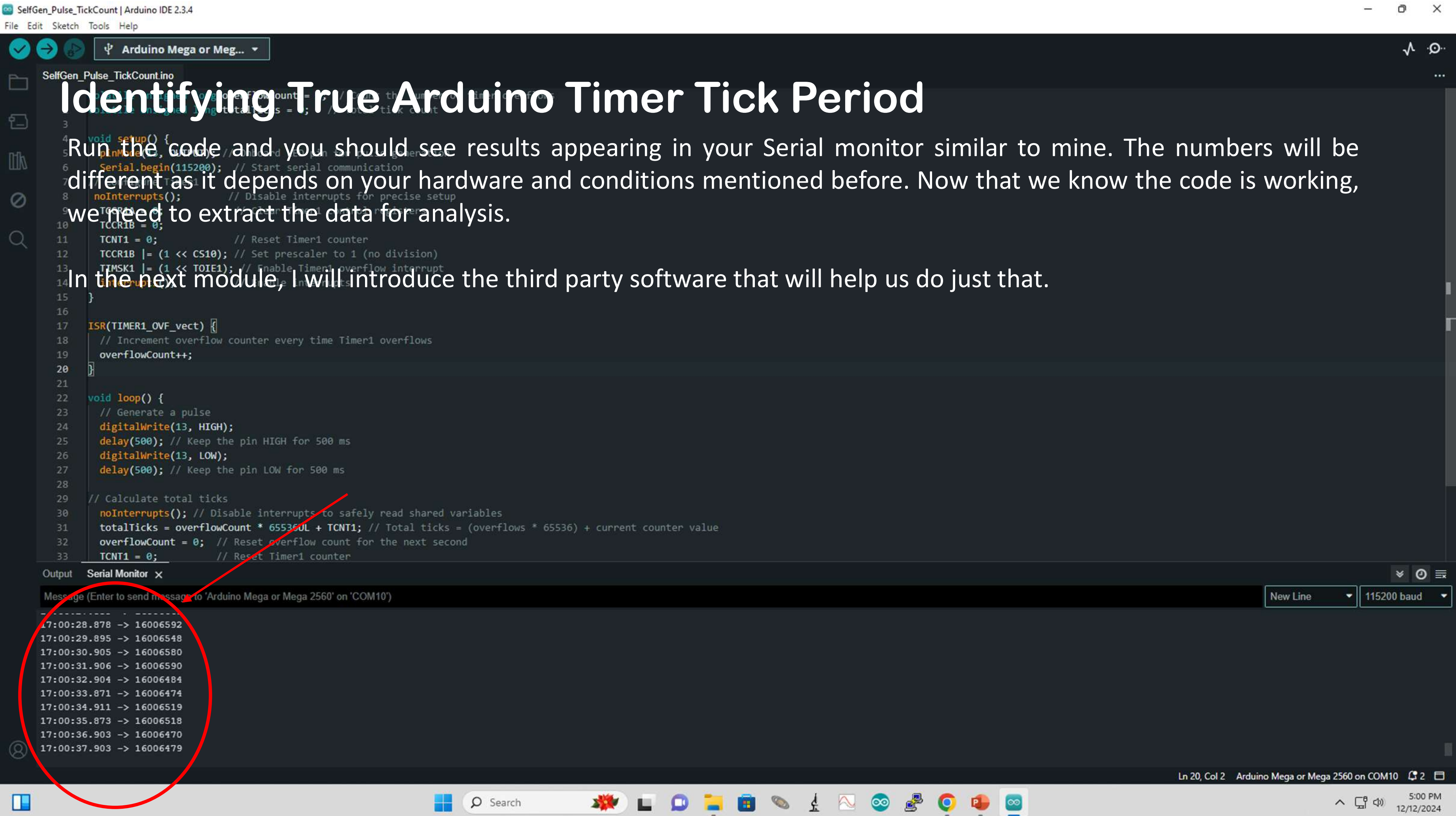
Now we will be testing for the True Arduino Timer Tick Period. To do this, disconnect all of your previous jumper wires, create a new sketch, and paste the code below.

```
volatile unsigned long overflowCount = 0; // Count the number of timer overflows
volatile unsigned long totalTicks = 0;    // Total tick count

void setup() {
  pinMode(13, OUTPUT); // Onboard LED pin for pulse generation
  Serial.begin(115200);  // Start serial communication
// Configure Timer1
 noInterrupts();        // Disable interrupts for precise setup
  TCCR1A = 0;           // Clear Timer1 control registers
  TCCR1B = 0;
  TCNT1 = 0;             // Reset Timer1 counter
  TCCR1B |= (1 << CS10); // Set prescaler to 1 (no division)
  TIMSK1 |= (1 << TOIE1); // Enable Timer1 overflow interrupt
 interrupts();           // Enable interrupts
}

ISR(TIMER1_OVF_vect) {
  // Increment overflow counter every time Timer1 overflows
  overflowCount++;
}

void loop() {
  // Generate a pulse
  digitalWrite(13, HIGH);
  delay(500); // Keep the pin HIGH for 500 ms
  digitalWrite(13, LOW);
  delay(500); // Keep the pin LOW for 500 ms

// Calculate total ticks
  noInterrupts(); // Disable interrupts to safely read shared variables
   totalTicks = overflowCount * 65536UL + TCNT1; // Total ticks = (overflows * 65536) + current counter value
   overflowCount = 0; // Reset overflow count for the next second
   TCNT1 = 0;         // Reset Timer1 counter
  interrupts();       // Re-enable interrupts

// Print the total ticks
  Serial.println(totalTicks);
}
```

Identifying True Arduino Timer Tick Period

Run the code and you should see results appearing in your Serial monitor similar to mine. The numbers will be different as it depends on your hardware and conditions mentioned before. Now that we know the code is working, we need to extract the data for analysis.

In the next module, I will introduce the third party software that will help us do just that.



Module VI

Setting Up Putty for Data Extraction

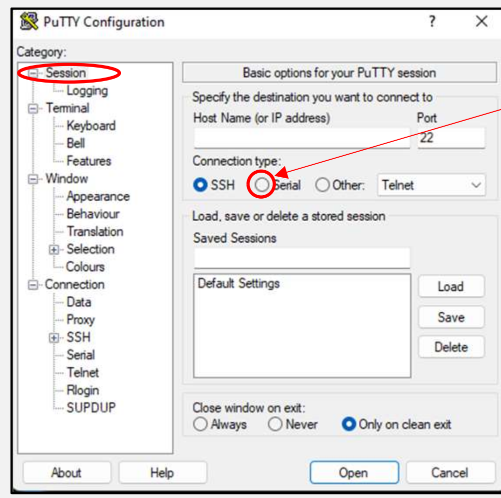
Putty

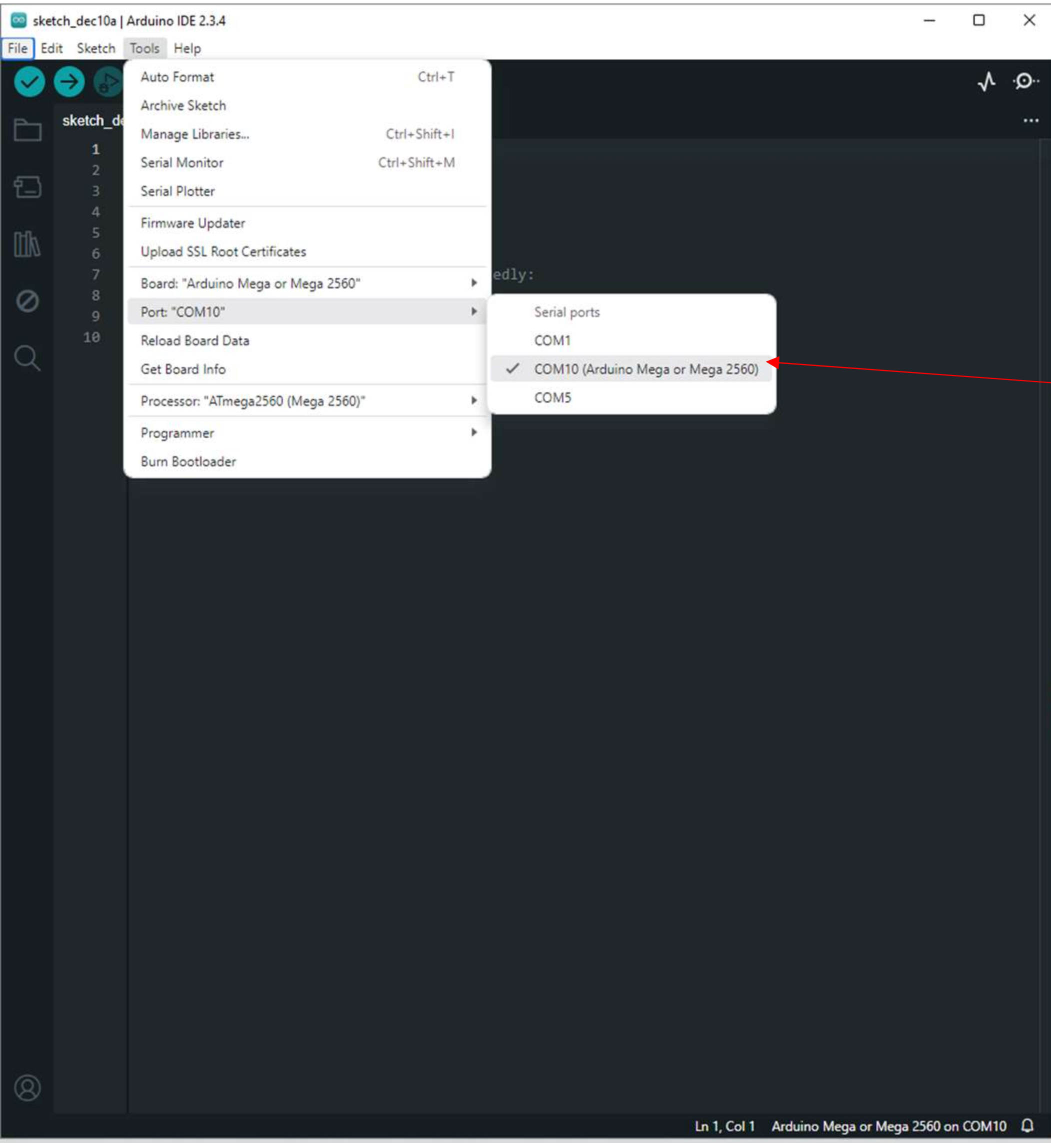
Putty Settings

Putty is an application that captures information sent to your computer from the Arduino by tapping into the serial port. With that said, you CANNOT have the serial monitor in the Arduino IDE open when running Putty as it will create a conflict and not run.

<u>1st Step:</u> Find the icon on your desktop to open up the Putty application. If not on your desktop, type Putty into your search bar to locate it in your PC.

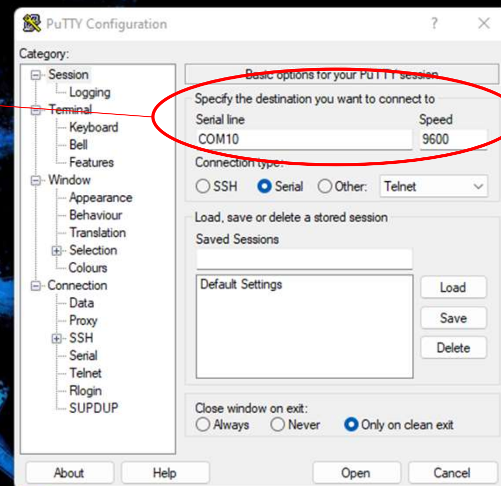
<u>2nd Step:</u> In the *Session* menu, change the connection type to serial.





Putty Settings

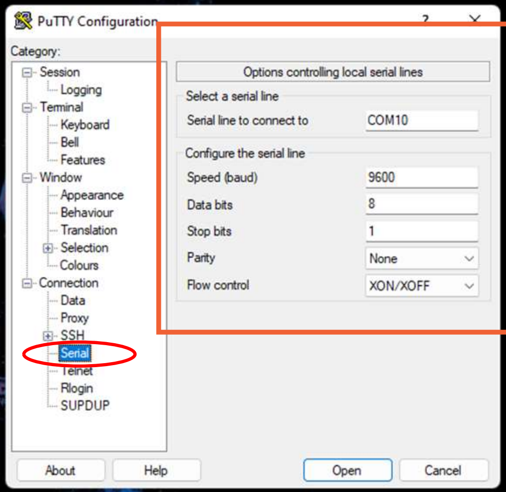
<u>3rd Step:</u> Make sure the Serial Line in Putty is set to the same port and baud rate the Arduino is using to communicate.



*To do this, navigate to *Tools > Port,* and identify which COM line is being used. Remember, the speed set here should match baud rate specified in the code you use while using Putty.

Putty Settings

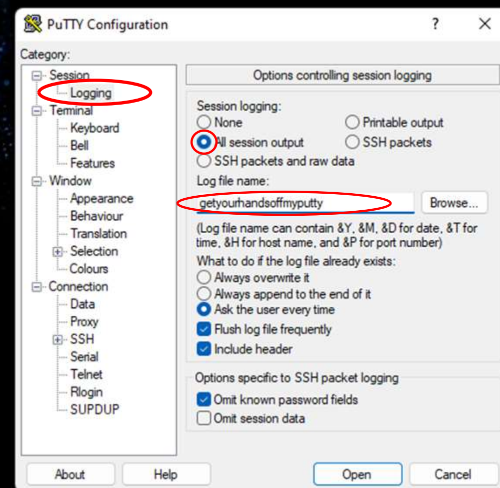
<u>4th Step:</u> Navigate to the *Serial* menu item in the category tree and configure your settings as such. The Serial Line and Speed you set are dependent on your Arduino IDE configuration. The speed should match the baud rate from the code and the Serial Line should match the port you are using. The rest of the options should be identical to mine.

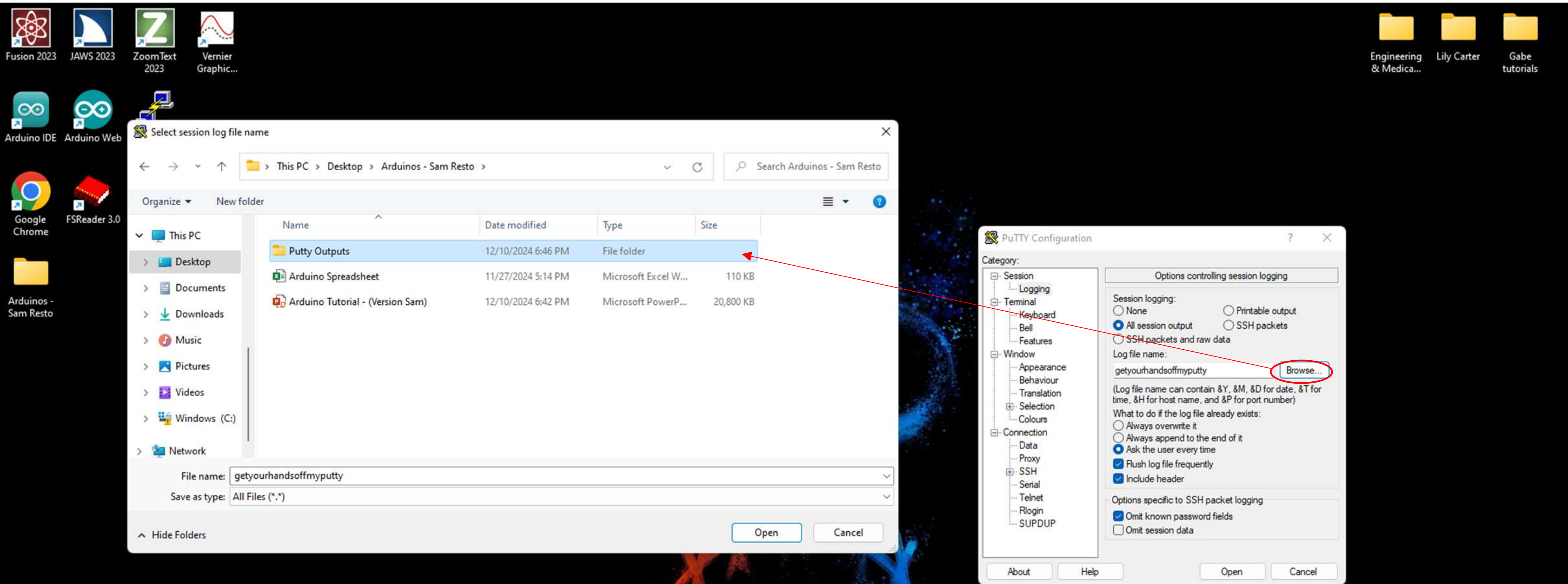


Configuring Putty to Save Data

5th Step: Navigate to the *Logging* menu item in the category tree and configure your settings to match that shown here.

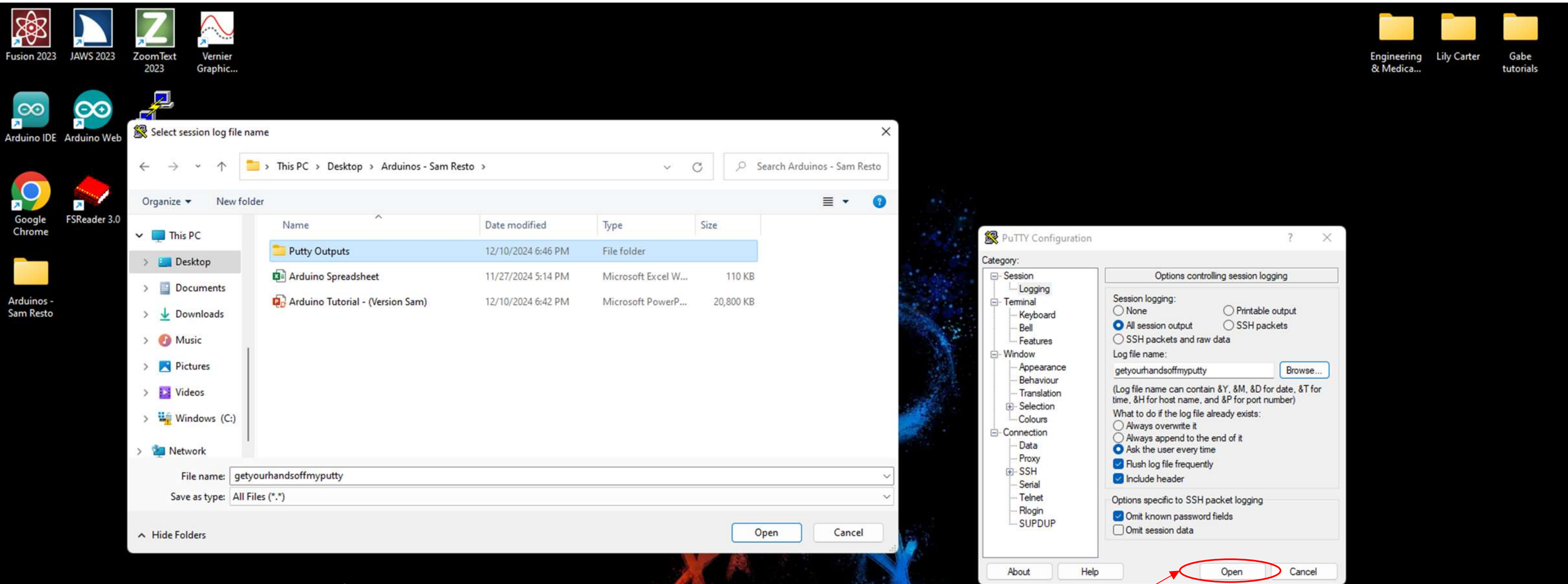
- Select *All session output*
- Rename your file however you like in the *Log file name* space given.





Configuring Putty to Save Data

<u>6th Step:</u> You can set the destination for the file to be saved in by clicking browse and selecting any location you'd like, a folder, the desktop, etc. I suggest creating a folder for your own research and keeping your data organized as I did for myself here.

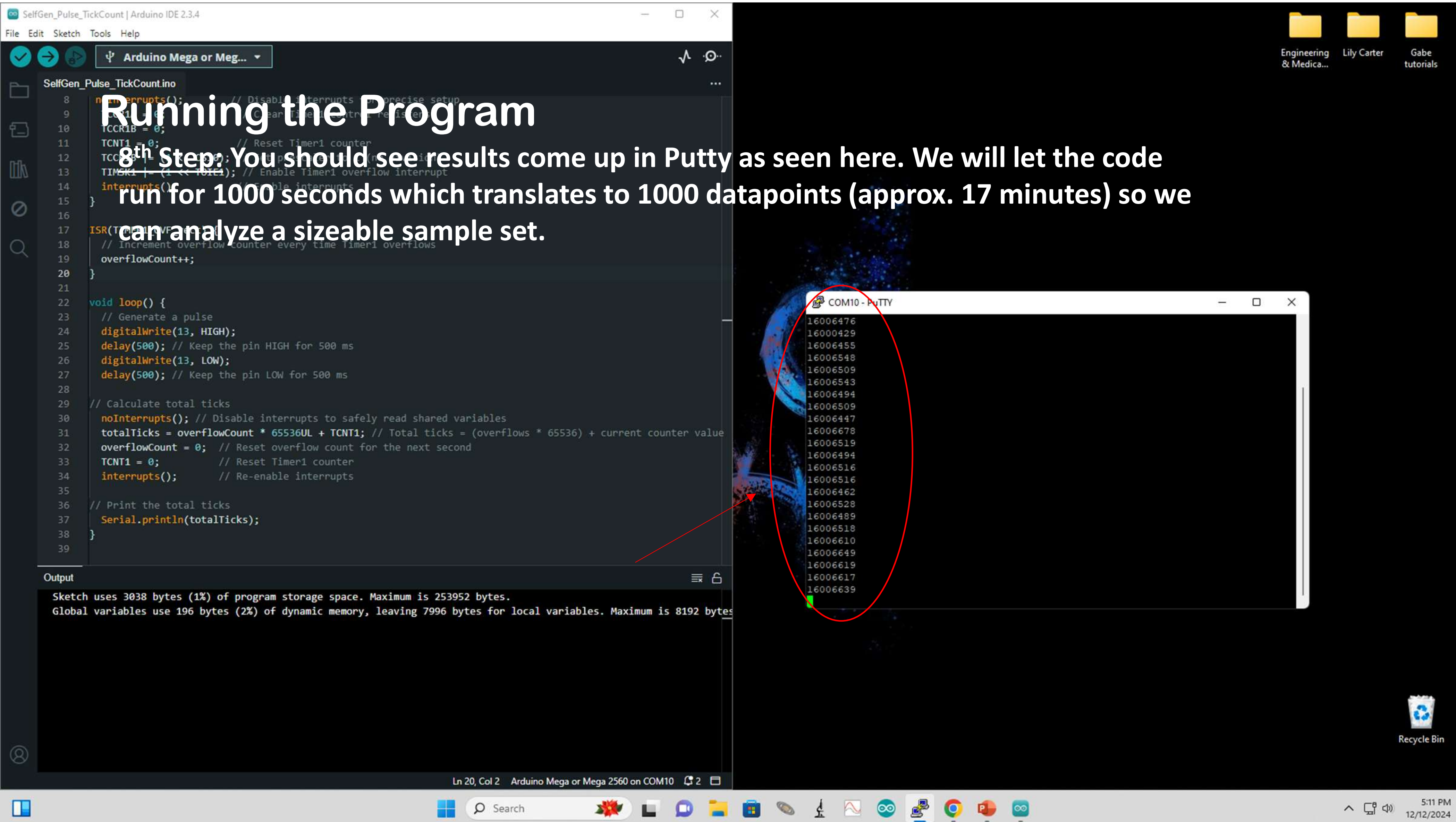


Running the Program

7th Step: Now that PuTTY is set up and ready to capture information from the IDE, the next step would be to click *Open* to start the connection. However, we won't do that just yet. Before PuTTY can receive data from the Arduino, you need to close the Serial Monitor in the Arduino IDE. This is because both the Serial Monitor and PuTTY use the same communication ports, which can create a conflict if both are open at the same time.

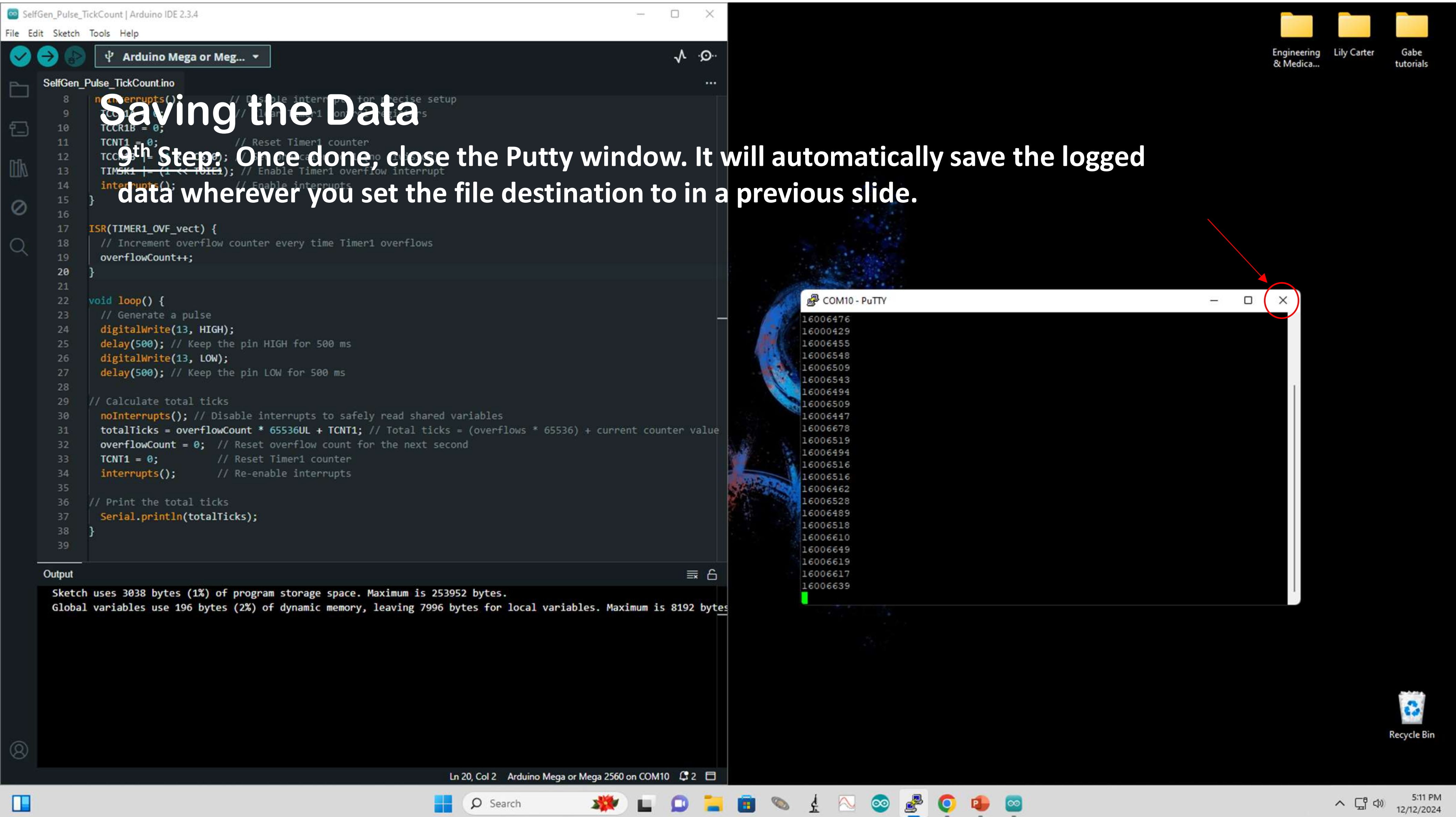
Running the Program

8th Step: You should see results come up in Putty as seen here. We will let the code run for 1000 seconds which translates to 1000 datapoints (approx. 17 minutes) so we can analyze a sizeable sample set.



Saving the Data

9th Step: Once done, close the Putty window. It will automatically save the logged data wherever you set the file destination to in a previous slide.



Module VII

Simple Data Analysis using Excel

Module VII

Testing Arduino Timer Against Signals Generated by the GPS Breakout v.3 PPS Pin

Ultimate GPS Breakout v.3

Testing the GPS Module

1. Now, let us connect the GPS module to the breadboard as you did with the LED backpack.
2. To connect the GPS module to the Arduino, the jumper wire must be placed into the breadboard slot next to the sensor's pin.
3. Connect jumper wires to make the following connections:

Connections

Arduino	GPS Module
(Power) 5V	VIN
(Power) GND	GND
(PWM) 2	PPS

*** In parentheses are the sections of the Arduino in which those pins are located.

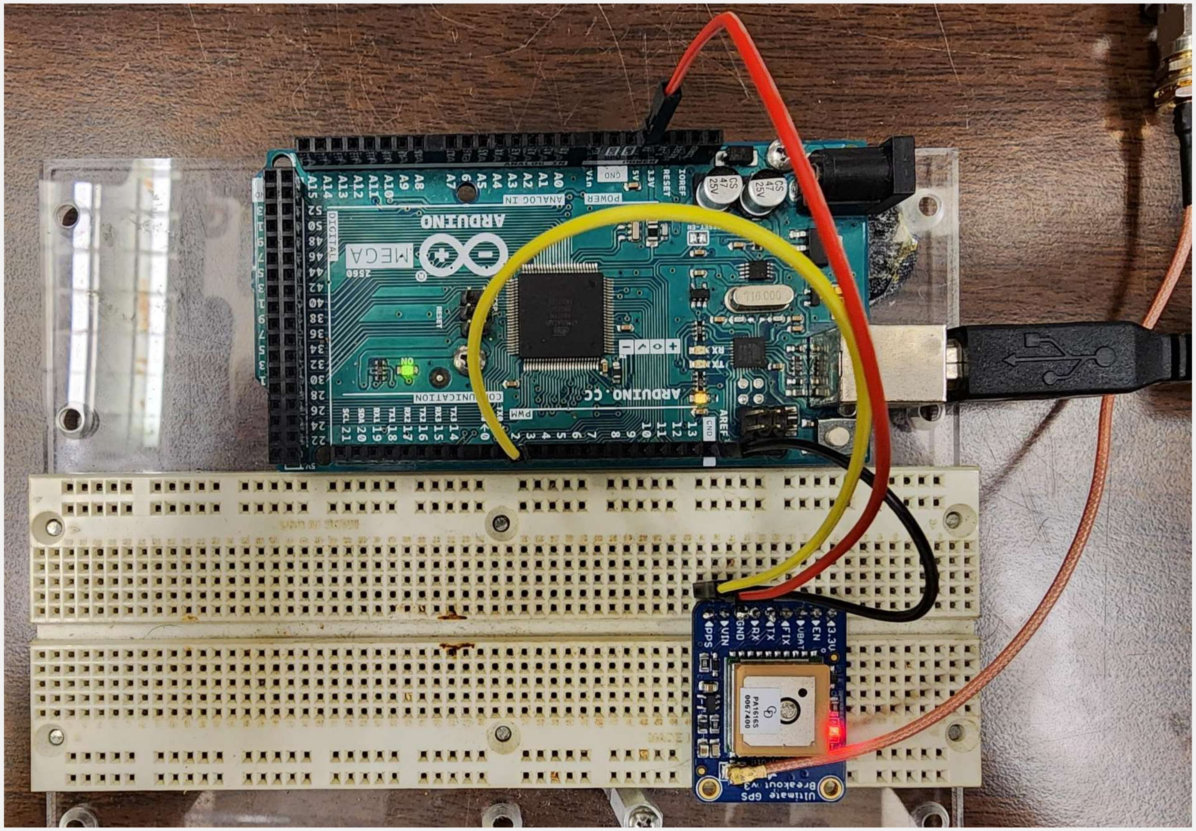
Ultimate GPS Breakout v.3

Setup

Here is what your setup should look like!

Connections

Arduino	GPS Module
(Power) 5V	VIN
(Power) GND	GND
(PWM) 2	PPS



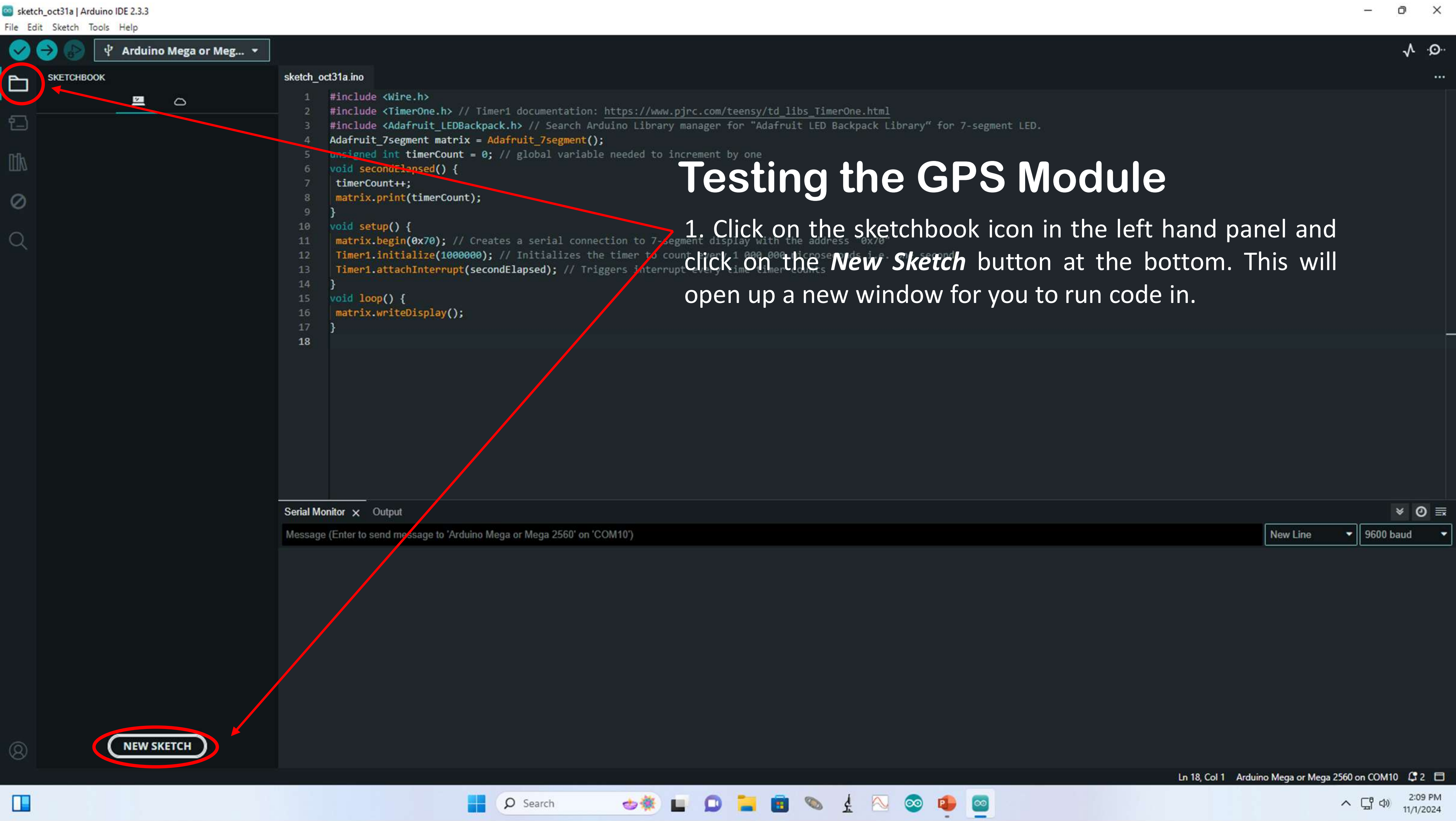
Ultimate GPS Breakout v.3

Testing the GPS Module

1. Once you connect the jumper wires to power the module, the FIX light will start blinking on and off. This indicates that the module hasn't acquired your location yet.
2. To resolve this, connect the GPS cables (hanging down over each workstation) to the port circled here.
3. After connecting, allow a few minutes for the module to establish your position. The process is complete when the blinking slows down to about once every fifteen seconds.

Testing the GPS Module

1. Click on the sketchbook icon in the left hand panel and click on the ***New Sketch*** button at the bottom. This will open up a new window for you to run code in.



Testing the GPS Module

1. Delete the default code that appears in the new window.
2. Copy and paste following code in its place:

```
#define PPS_PIN 2 // The pin we're attaching to the PPS signal from the GPS unit
 volatile unsigned int overflows = 0;
 volatile unsigned int overflowsSincePPS = 0;
 volatile unsigned int lastTimer1 = 0;
 volatile bool recentPPS = false;
 ISR(TIMER1_OVF_vect) // This is called whenever Timer/Counter 1 overflows
{
 overflows++; // Increases the "overflows" variable by 1
}
void setup() {
 Serial.begin(115200);
 delay(1000);
 pinMode(PPS_PIN, INPUT);
 TCCR1A = 0; // Sets entire TCCR1A--Timer1 Control Register A--to 0
 TCCR1B = bit(CS10); // Turns on the Timer1 clock and sets it to increment every clock cycle
 TCCR1C = 0; // Timer 1 Control Register C set to 0
 TCNT1 = 0; // Initialize timer/counter 1's value to 0
 TIMSK1 = bit(TOIE1); // Timer/Counter1's interrupt mask register; TOIE1 is the timer/Counter1 overflow interrupt enable
 Serial.println("Starting up...");
 attachInterrupt(digitalPinToInterrupt(PPS_PIN), PPSHandler, RISING);
}
void PPSHandler() { // Since this is an interrupt we should do as little as possible here. Serial writes take a lot of clock cycles, so we save that for the loop.
 lastTimer1 = TCNT1;
 TCNT1 = 0; // Resets Timer1 Count
 overflowsSincePPS = overflows;
 overflows = 0;
 recentPPS = true;
}
void loop(){
 if (recentPPS) {
  noInterrupts();
  uint32_t overflowsTemp = overflowsSincePPS;
  uint32_t lastTimerTemp = lastTimer1;
  interrupts();
  Serial.print("Overflow:");
  Serial.println(overflowsTemp);
  Serial.print("Timer1:");
  Serial.println(lastTimerTemp);
  Serial.print("ClockCycles:");
  Serial.println(overflowsTemp << 16 | lastTimerTemp); // Equivalent to overflowsTemp * 2^16 + lastTimerTemp
 recentPPS = false;
 }
}
```

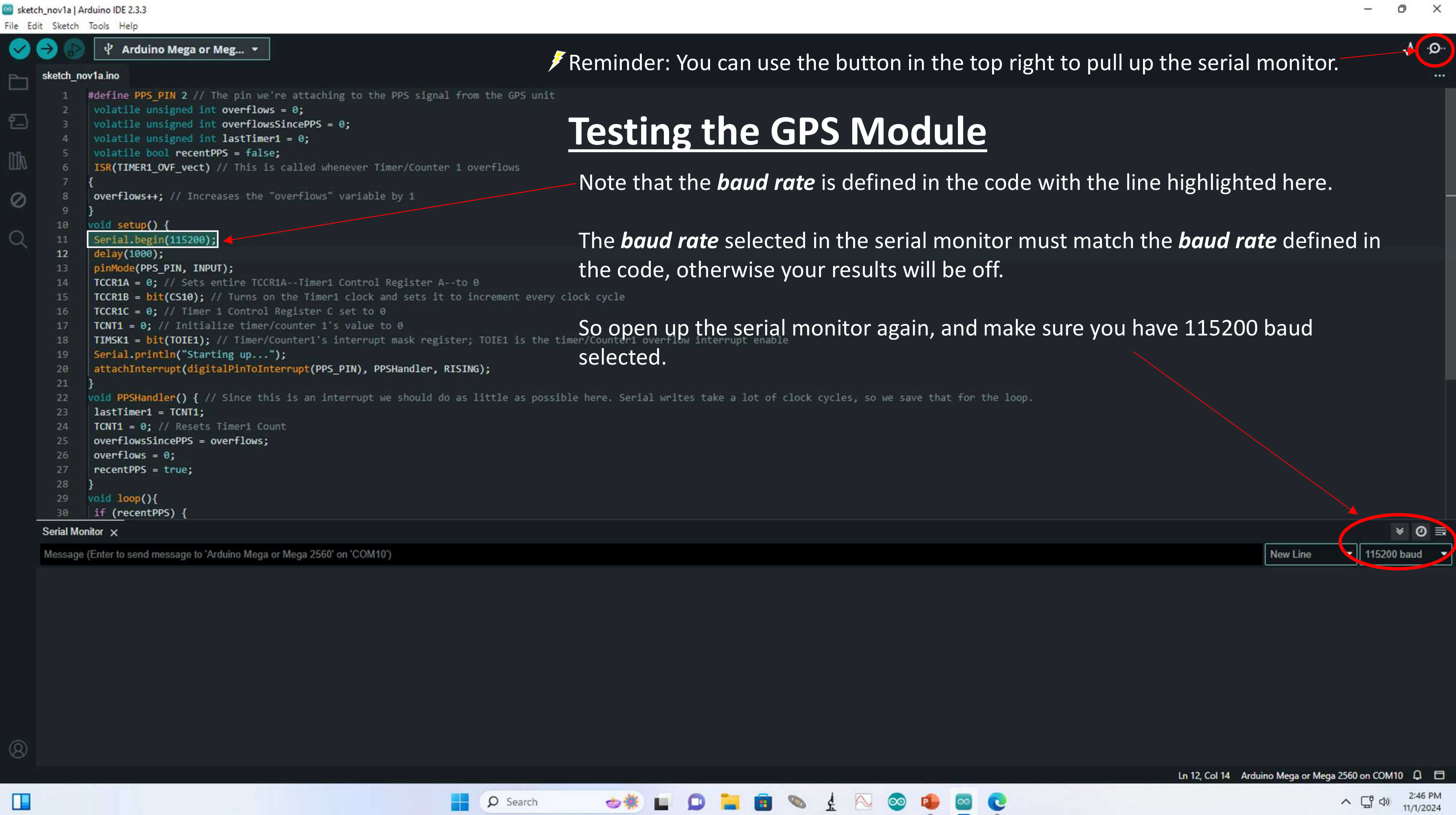
Testing the GPS Module

Reminder: You can use the button in the top right to pull up the serial monitor.

Note that the ***baud rate*** is defined in the code with the line highlighted here.

The ***baud rate*** selected in the serial monitor must match the ***baud rate*** defined in the code, otherwise your results will be off.

So open up the serial monitor again, and make sure you have 115200 baud selected.



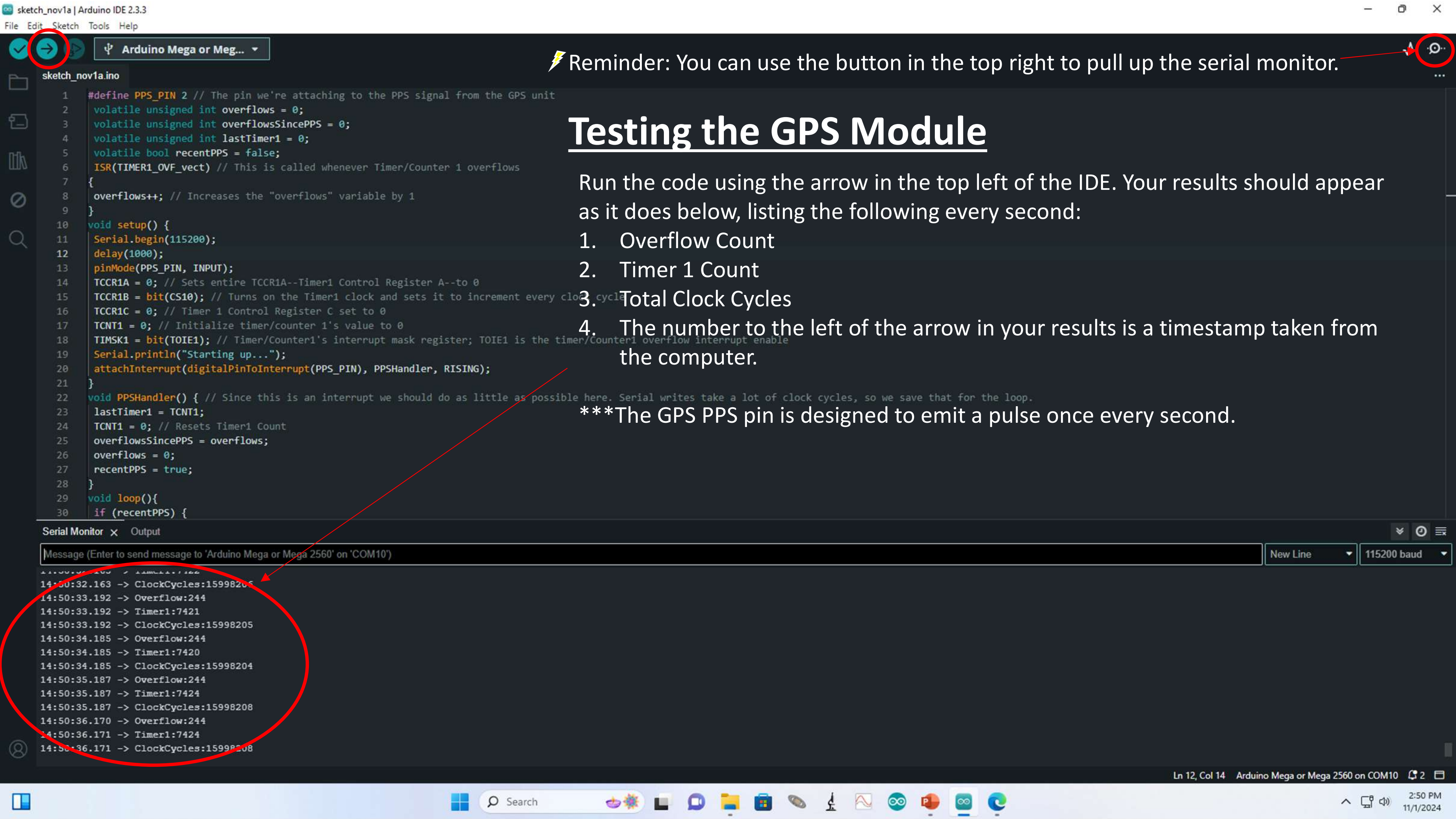
Testing the GPS Module

Reminder: You can use the button in the top right to pull up the serial monitor.

Run the code using the arrow in the top left of the IDE. Your results should appear as it does below, listing the following every second:

1. Overflow Count
2. Timer 1 Count
3. Total Clock Cycles
4. The number to the left of the arrow in your results is a timestamp taken from the computer.

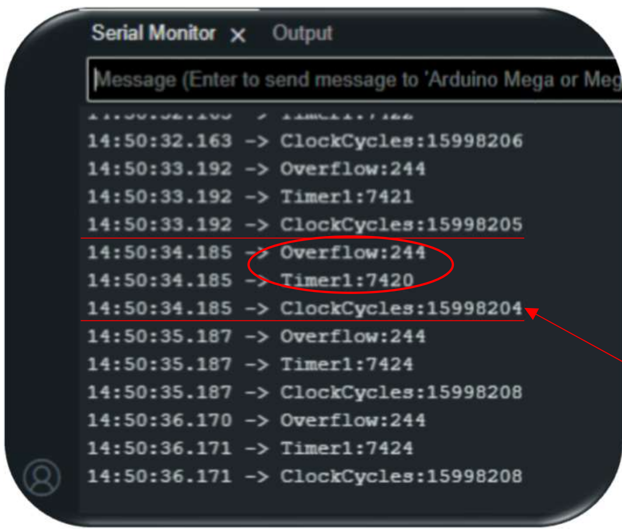
***The GPS PPS pin is designed to emit a pulse once every second.



Key Concepts

Applying What We Learned

From the results displayed in the Serial Monitor, we can observe the time elapsed between each signal received from the GPS PPS pin. This elapsed time is represented by two key values: the Overflow value and the Timer1 value. The Overflow value reflects the number of complete timer overflows that have occurred, while the Timer1 value indicates the leftover timer ticks that have not yet accumulated enough to increment the Overflow counter. Together, these values provide a precise measurement of the time interval between signals.



Using my results, you can see that 244 Overflows plus 7420 individual ticks occurred in between that signal and the last. Knowing these values, we can calculate the elapsed time by doing the following:

$$(244 \times 65{,}536) + 7420 = 15{,}998{,}204 \text{ ticks}$$

You may have noticed that this number matches the *ClockCycles* value from the same timestamp. In the code, we have programmed the Arduino to perform the necessary calculation, allowing us to see the total number of ticks between pulses from the PPS pin.

Module VII

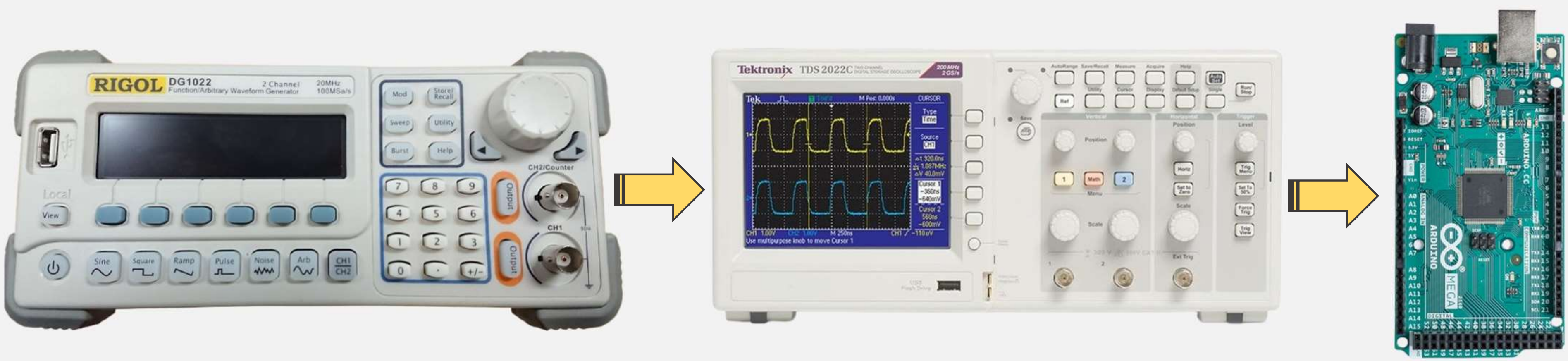
Arduino Timer Testing w/ Pulse Generator

Pulse Generator & Oscilloscope

Connecting the Pulse Generator & Oscilloscope

For this part we will be using the Pulse Generator and Oscilloscope in tandem with the Arduino board to test out some more code.

Before we can do this, you need to make sure your equipment is set up properly.



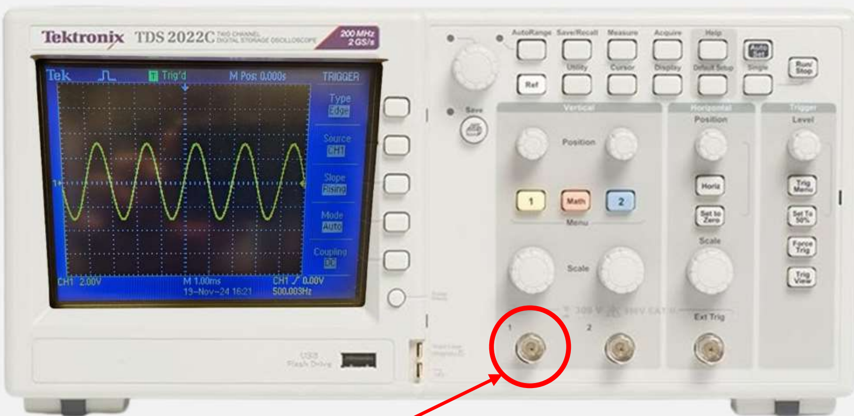
Pulse Generator & Oscilloscope

Connecting the Pulse Generator & Oscilloscope

The first connection we'll need to secure is the BNC cable to the Oscilloscope. Connect the BNC Cable from CH1 (Channel 1) on the Pulse Generator to Channel 1 on the Oscilloscope.

Pulse Generator

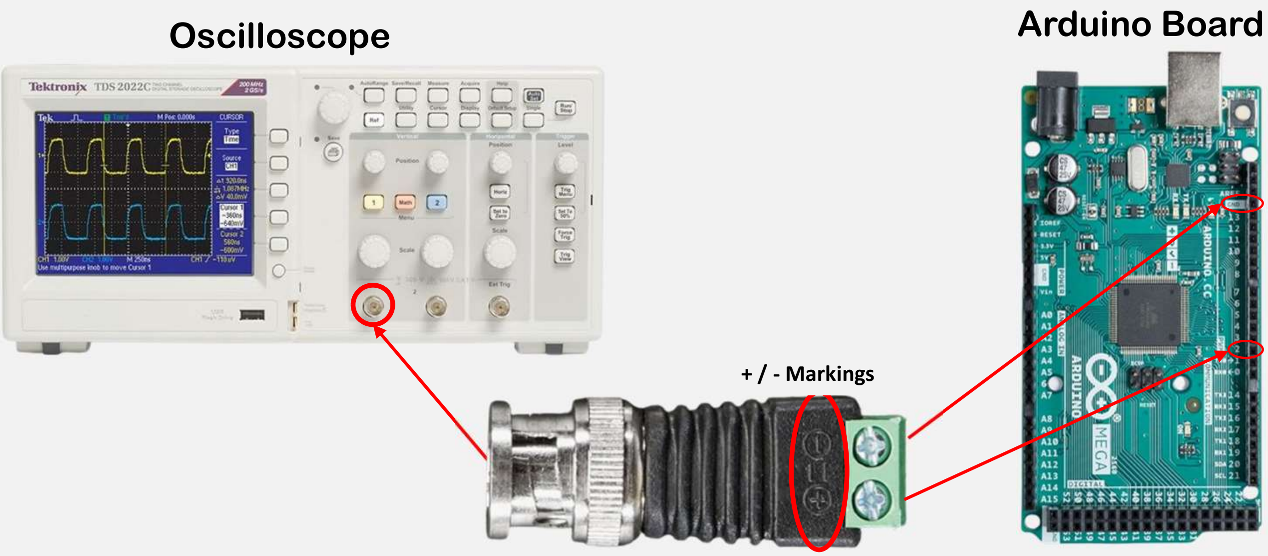
Oscilloscope



Pulse Generator & Oscilloscope

Connecting the Pulse Generator & Oscilloscope

The other end of the BNC Cable should be equipped with an adaptor that has two slots for jumper wires so that you can attach them to the Arduino board. The adaptor slots for the jumper wires are marked with a positive and negative sign. Use this to make the connections in the table below.



Connections

BvNC Cable	Arduino
+	2 (PWM)
-	GND (PWM)

*** In parentheses are the sections of the Arduino in which those pins are located.

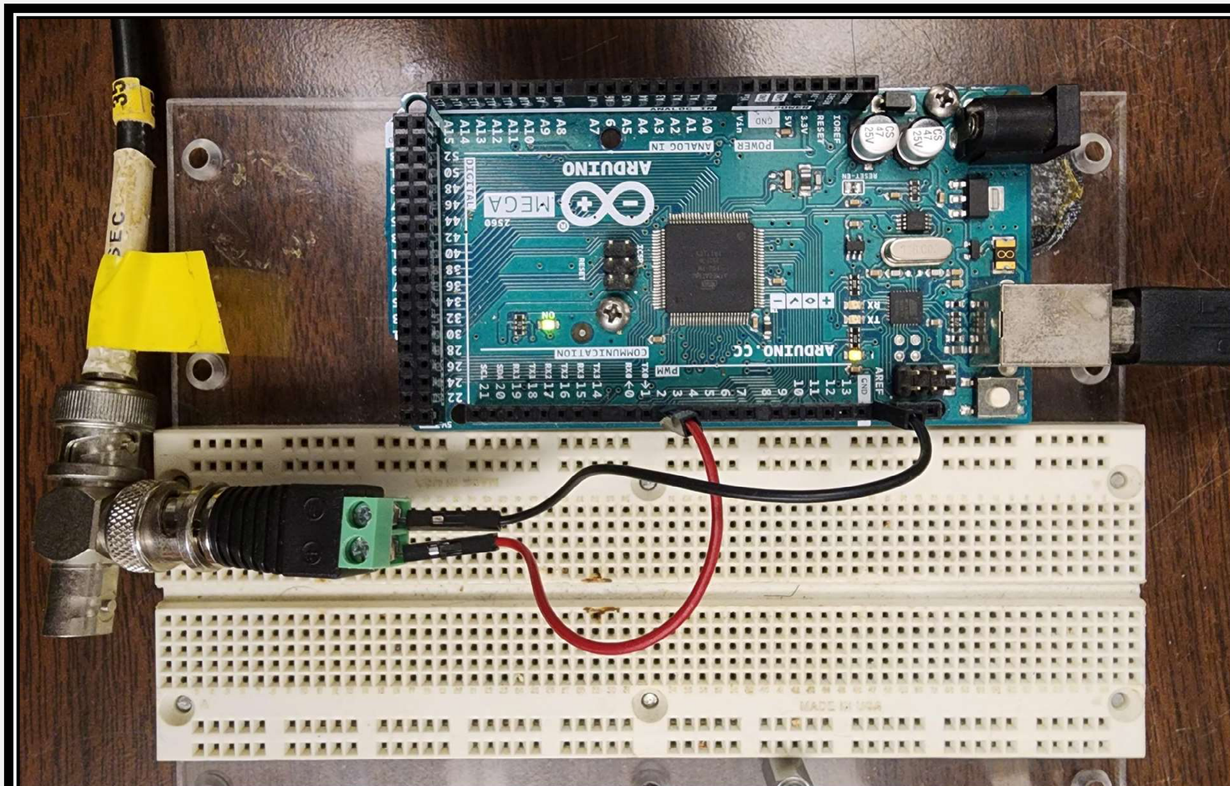
Pulse Generator & Oscilloscope

Setup

Here is what your setup should look like!

Connections

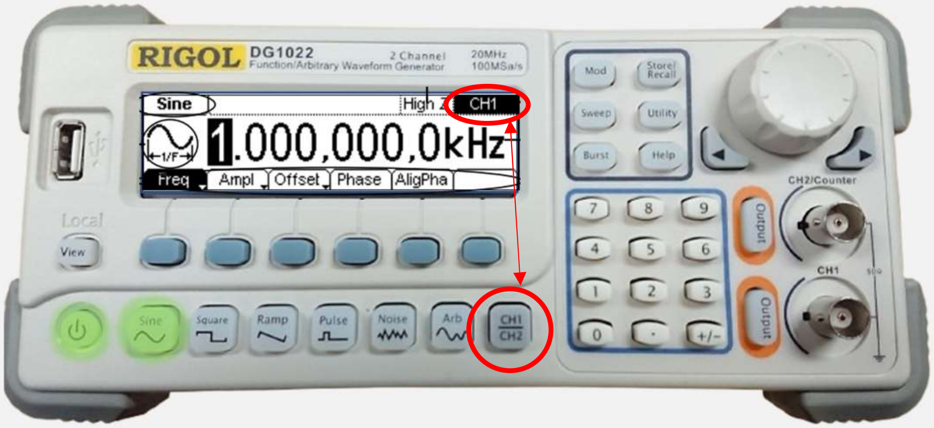
BNC Cable	Arduino
+	2 (PWM)
-	GND (PWM)

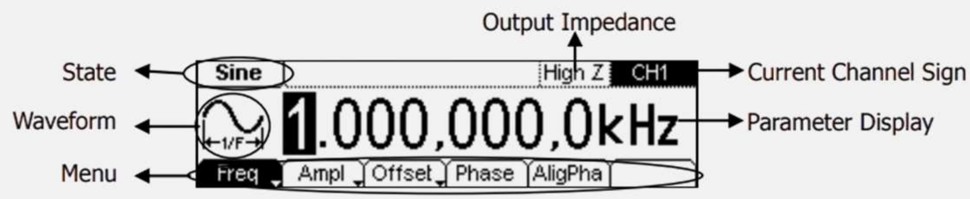


Pulse Generator & Oscilloscope

Setting Up The Pulse Generator

Now lets adjust the settings on the Pulse Generator.





Turning on the pulse generator, the display will light up and you'll see a screen similar to the one above. Ensure that the Current Channel Sign is set to CH1. If set to CH2, press the CH1/CH2 button at the bottom of the panel to toggle between the two.

Pulse Generator & Oscilloscope

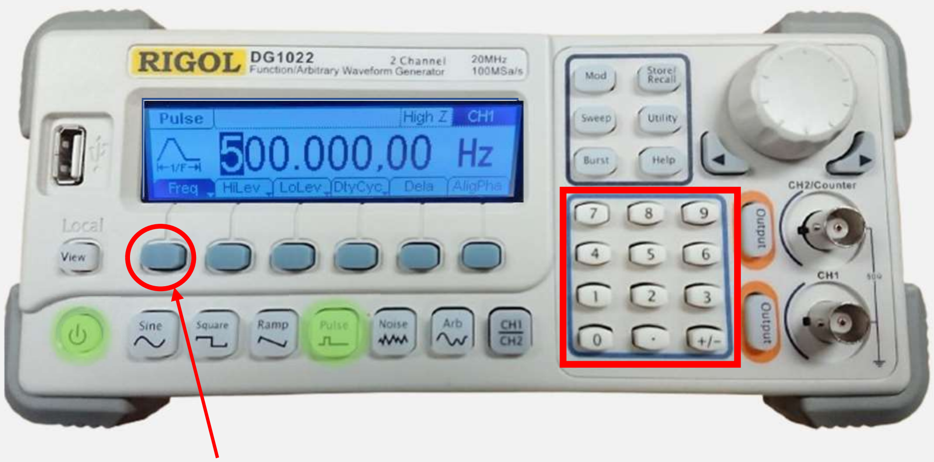
Setting Up The Pulse Generator

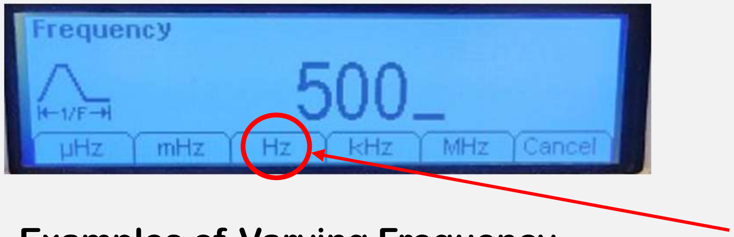
Change the type of waveform generated to a pulse. Pulses are ideal for measuring how systems respond to quick, temporary changes. You'll notice the waveform image and state shown in the display will change as well.

Pulse Generator & Oscilloscope

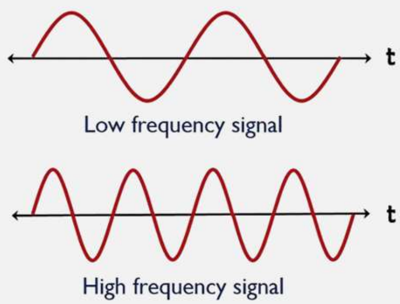
Adjusting the Frequency

The frequency determines how many pulses per second the pulse generator emits. So at 500Hz, it'll send out 500 pulses per second.





Examples of Varying Frequency

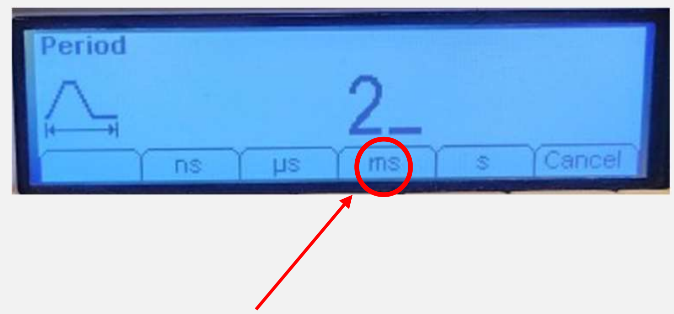


Pressing the blue button under the *Freq* (Frequency) menu item will toggle the screen from *Period* to *Frequency* and vice-versa. If it is already on *Frequency*, adjust the value to 500Hz by entering 500 using the number keypad outlined in red above and finalizing your entry by selecting the appropriate scale from the new menu that appears underneath your input. In our case, we want to choose Hz (hertz).

Pulse Generator & Oscilloscope

Adjusting the Period

The period is how long it takes for a pulse to complete one full cycle.



Toggle the menu into Period mode, by hitting the blue button underneath the *Freq* menu item and enter the desired time. For our purposes, let us enter 2ms by entering 2 with the number keypad and finalize it by pressing the blue button underneath the *ms* (milliseconds) menu item.

***I show you how to adjust the period so that you know how, but the period is automatically set once you enter the frequency as the two measures are related. A 500Hz frequency waveform will always have a 2ms period.

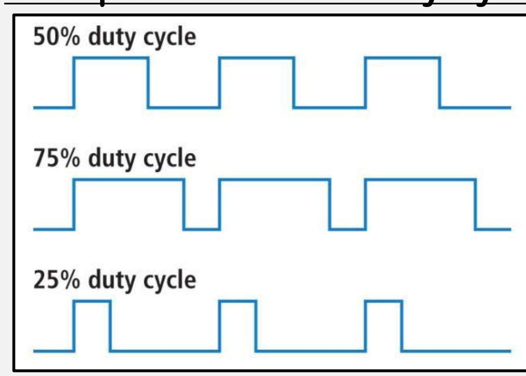
Pulse Generator & Oscilloscope

Adjusting the Duty Cycle

The Duty Cycle refers to the percentage of the waveform that is in the ON (high voltage) state.

Press the button below *DtyCyc* (Duty Cycle) and it will display its current setting. Enter 10 using the number keypad and finalize your input by pressing the button below the percentage sign in the newly displayed menu.

Examples of Various Duty Cycles



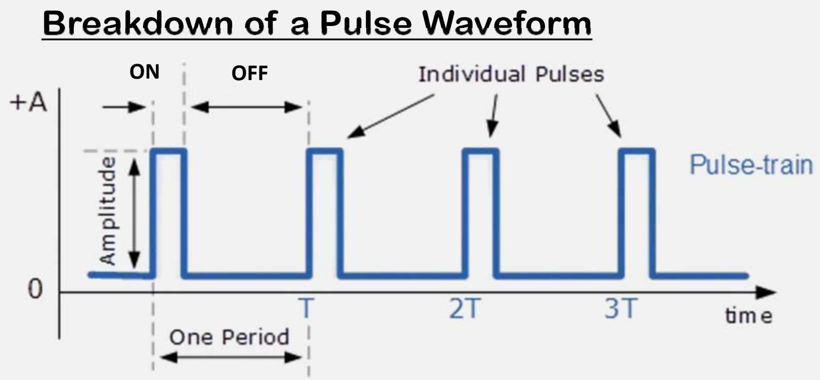
*As seen on the oscilloscope.

**Note that pressing the *DtyCyc* button twice will toggle it between *DtyCyc* and *Width.* So, if you only see *Width* on your screen, just press the button below the menu option to change it to *DtyCyc.*

Pulse Generator & Oscilloscope

Adjusting the High Level

The High Level refers to the amplitude (height) of the waveform.



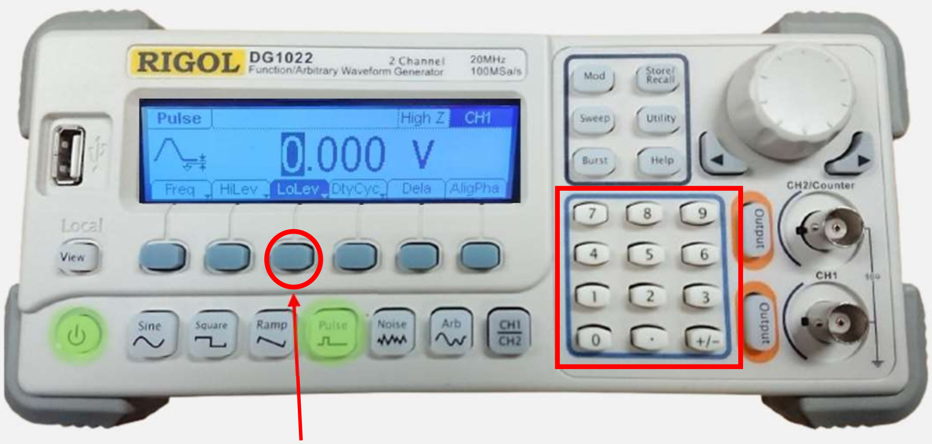
Press the button below *HiLev* (High Level) and it will display its current setting. Enter 5 using the number keypad and finalize your input by pressing the button below the appropriate scale in the newly displayed menu. For this exercise, select V (volts).

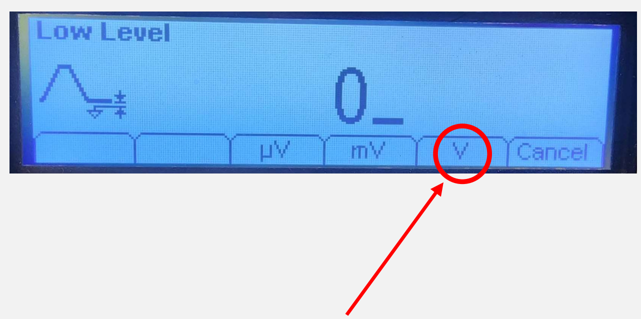
**Note that pressing the *HiLev* option twice will toggle it between *Ampl* and *HiLev.* So, if you only see *Ampl* on your screen, just press the button below the menu option to change it to *HiLev.*

Pulse Generator & Oscilloscope

Adjusting the Low Level

The Low Level refers to the baseline or lowest voltage point of the waveform.





Press the button below *LoLev* (Low Level) and it will display its current setting. Enter 0 using the number keypad and finalize your input by pressing the button below the appropriate scale in the newly displayed menu. For this exercise, select V (volts).

**Note that pressing the *LoLev* option twice will toggle it between *Offset* and *LoLev.* So, if you only see *Offset* on your screen, just press the button below the menu option to change it to *LoLev.*

Pulse Generator & Oscilloscope

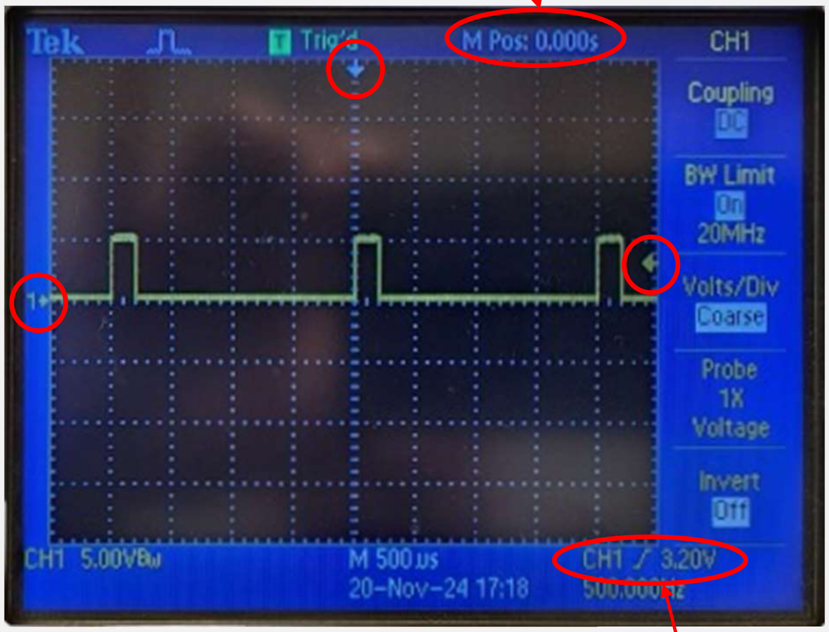
Sending Out the Signal

***If your screen looks different than the one shown here, try cycling through the various view modes by pressing the *View* button until it's similar to mine.

Near the BNC cable connected to CH1, there is a button labeled *Output.* It must be lit up in order for its signal to be sent out to the oscilloscope. If it is not lit up, press the button and you should see the waveform it is generating show up on the display of your oscilloscope.

Pulse Generator & Oscilloscope

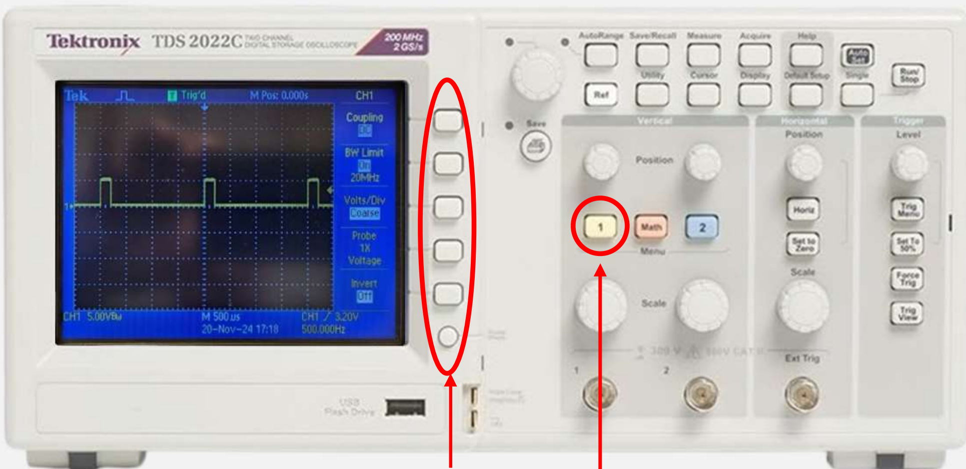
Setting Up The Oscilloscope



Before we can verify the equipment reading, let us run through some settings to make sure the oscilloscope is doing what we want it to. A good place to start is to adjust the position of all the cursors. The horizontal, vertical and trigger level cursor. Adjust the knobs so that the arrows are set to 0 at all points which will align each one with the x or y axis. ***The vertical cursors position will appear on the screen as you turn the knob.

Pulse Generator & Oscilloscope

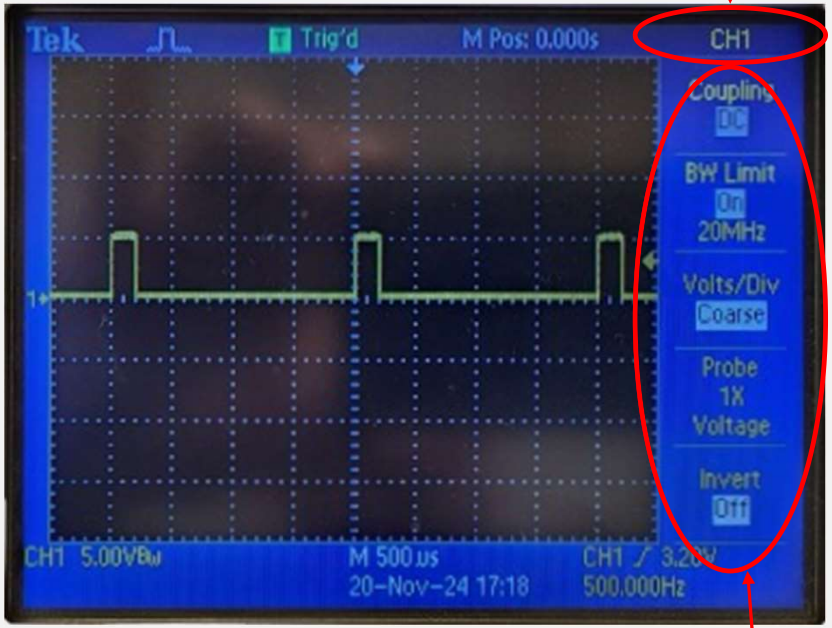
Setting Up The Oscilloscope



Menu Cycle Buttons

Channel 1 Menu Button

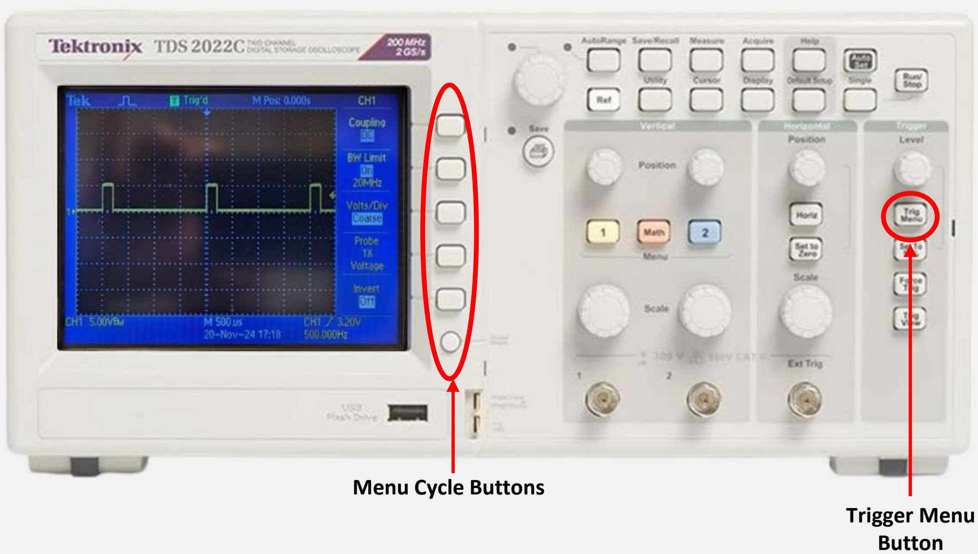
Current Menu Displayed

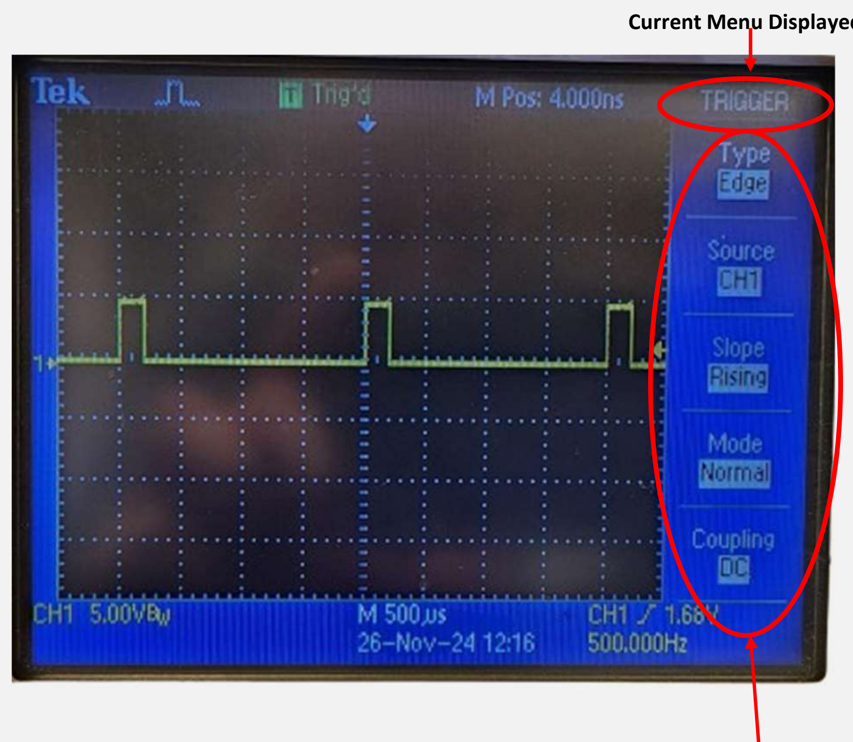


Press the Channel 1 button on your oscilloscope. This will display the Channel Menu on your screen as seen above. Make sure your settings match what is circled on the screen here. If not, cycle through the options by pressing the buttons alongside each menu item until it is identical.

Pulse Generator & Oscilloscope

Setting Up The Oscilloscope

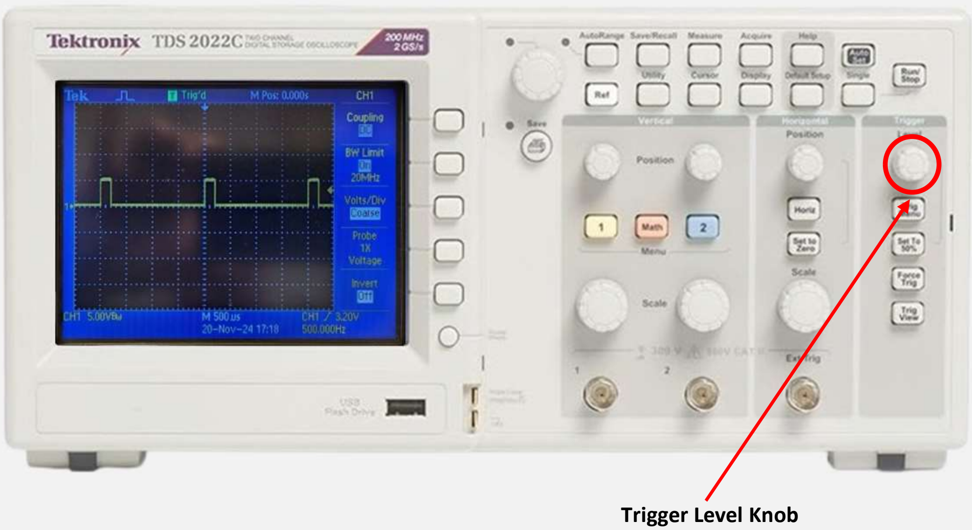


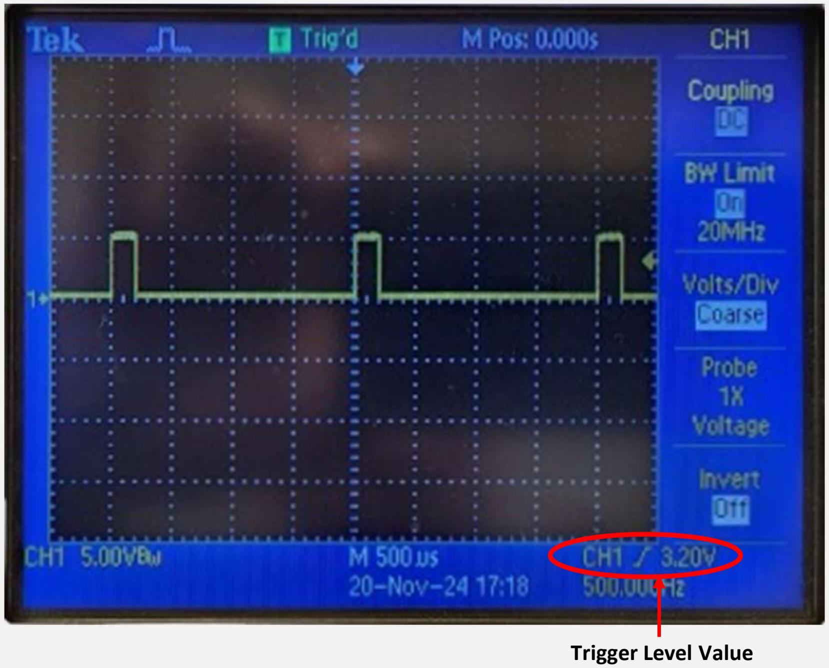


Press the Channel 1 button on your oscilloscope. This will display the Channel Menu on your screen as seen above. Make sure your settings match what is circled on the screen here. If not, cycle through the options by pressing the buttons alongside each menu item until it is identical.

Pulse Generator & Oscilloscope

Stabilizing the Signal





Most likely at this point, your signal is unstable, meaning its moving around the screen. To stabilize it, adjust the trigger level to whatever value you need to so that the wave remains still. It should be some value in between your low and high level, which in our case, is 0 and 5 volts respectively.

Pulse Generator & Oscilloscope

Setting the Appropriate Scale

The knob highlighted in purple controls the voltage scale (y-axis). Adjusting this knob changes how much voltage each box in the grid represents. In this case, I have my scale set to 5 volts, so each box represents 5 volts.

The knob highlighted in yellow controls the time scale (x-axis). Adjusting this knob changes how much time each box in the grid represents. In this case, I have my scale set to 500µs (microseconds), so each box represents 500µs.

Pulse Generator & Oscilloscope

Verify the Reading

As you recall, we set our pulse generator to a period of 2ms (milliseconds) and a high level of 5 volts. Judging from the signal seen on our oscilloscope, we can see that everything is in working order. The amplitude is one box high, representing 5 volts, and the period, the length of time from the start of one pulse to the next, is 2000 microseconds which is equal to 2 milliseconds.

Testing the accuracy of the Arduino Timer

1. Open up a new sketch, delete the default code that appears in the new window.
2. Copy and paste following code in its place:
3. Note the baudrate here is 115200 as seen in the line of code “Serial.begin(115200)”. Ensure the baud rate in your IDE is set to the same value.
4. Run the code with the arrow icon in the top left.

sketch_oct31a | Arduino IDE 2.3.3

File Edit Sketch Tools Help

Arduino Mega or Meg...

sketch_oct31a.ino

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:
```

```
#include <SPI.h> // Allows you to communicate with SPI (Serial Peripheral Interface) devices, with the Arduino as the master device
#include <Wire.h> // Enable this line if using Arduino Uno, Mega, etc.
// Signal pins are given a name, Global variables
#define triggerPin 2 // Trigger signal pin
// Interrupt service routine
// This function must be implemented, so that the TCNT1 counter counts
ISR(TIMER1_OVF_vect)
{
}
// All Arduino programs must contain a setup() and loop() functions
void setup() {
Serial.begin(115200); // Starts the serial monitor, sets baudrate to "115200" BPS
pinMode(triggerPin,INPUT); // Sets the digital pin 2 as an input
delay(1000); // Pauses the program for one second at the moment of open
// Initializes the Timer1 registers (16-bit timer -- counts from 0 to 65535 ad nauseam). Timer interrupts pauses the execution of the loop() function for a predefined number of seconds.
// Timer1 is a 16-bit timer, so the timer will increase its value until it reaches its maximum count before reverting to 0. This enables the program to run a different set of commands. Once executed, the program resumes at the same position.
TCCR1A = 0; // Sets entire TCCR1A--Timer1 Control Register A--to 0
TCCR1B = 0; // Timer 1 Control Register B set to 0 (The physical address of timer1)
TCCR1C = 0; // Timer 1 Control Register C set to 0
TCNT1 = 0; // Initialize timer/counter 1's value to 0
TIMSK1 = _BV(TOIE1); // Timer/Counter1's interrupt mask register; TOIE1 is the timer/Counter1 overflow interrupt enable
TCCR1B = 1; // Timer 1 Control Register B set to 1
attachInterrupt(digitalPinToInterrupt(triggerPin), Trigger, RISING); // Interrupts execution of the program when a trigger signal is received. The "Trigger" function is subsequently executed
}
void Trigger(){
unsigned int temp = TCNT1; // Only positive integers are required
Serial.print("TCNT1 value: ");
Serial.println(temp); // Prints the value stored at temp
}
void loop() {
// No lines are necessary here
}
```

Ln 1, Col 1 Arduino Mega or Mega 2560 on COM10

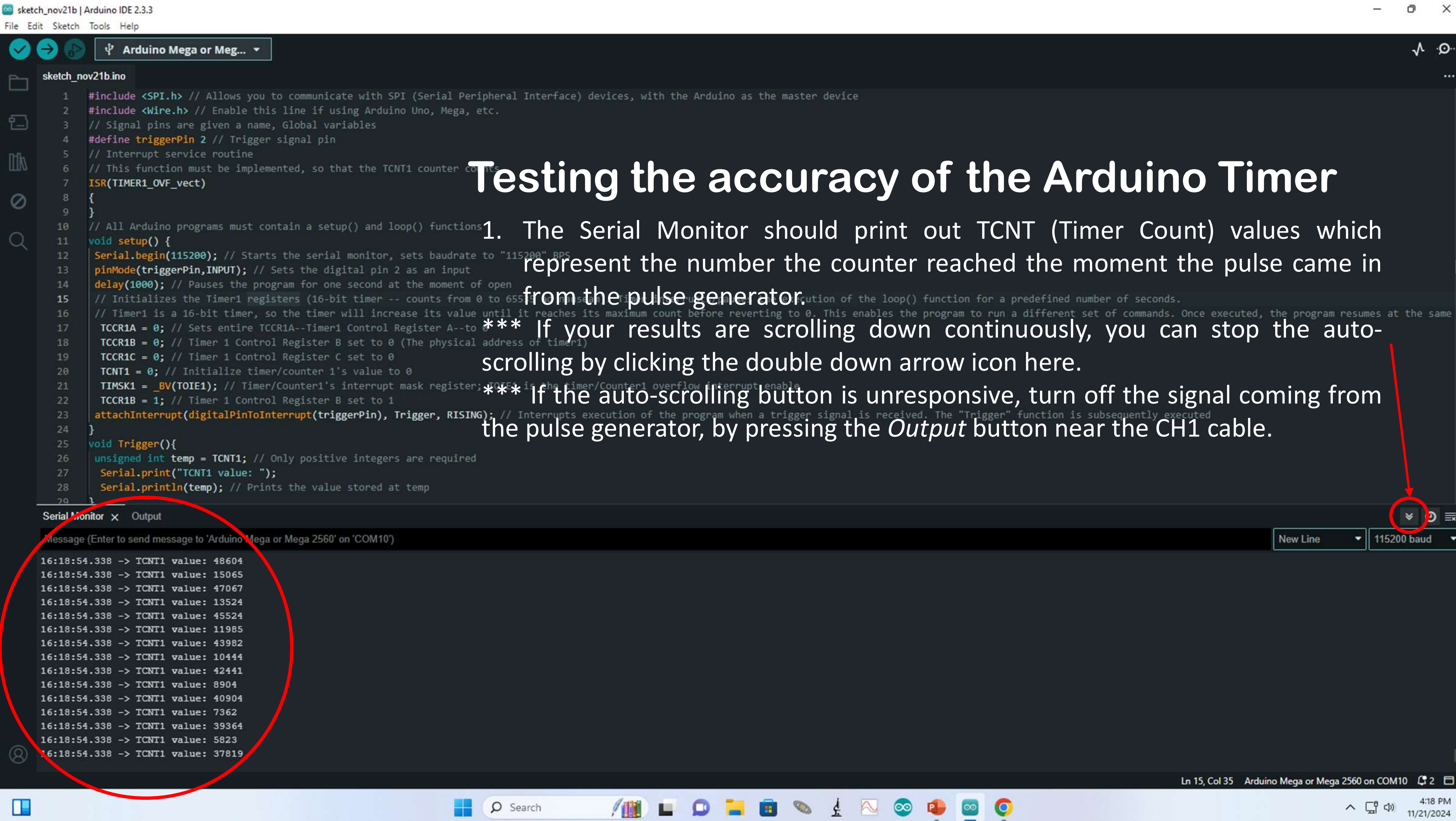
5:03 PM 10/31/2024

Testing the accuracy of the Arduino Timer

1. The Serial Monitor should print out TCNT (Timer Count) values which represent the number the counter reached the moment the pulse came in from the pulse generator.

*** If your results are scrolling down continuously, you can stop the auto-scrolling by clicking the double down arrow icon here.

*** If the auto-scrolling button is unresponsive, turn off the signal coming from the pulse generator, by pressing the *Output* button near the CH1 cable.

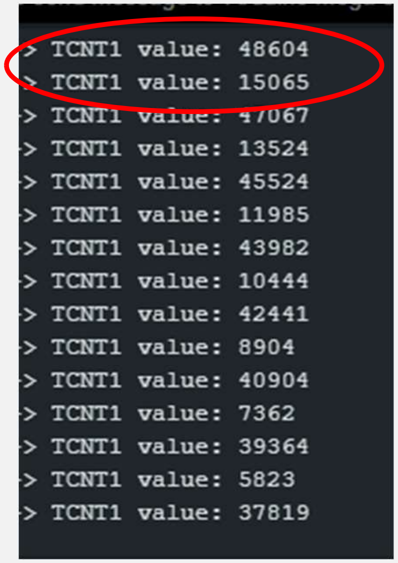


Arduino Timers

Testing Arduino Timer Accuracy

Now that we have a better understanding of how the Arduino Timer works, let us use this knowledge to test its accuracy.

The last snippet of code I gave you logged the number the Arduino Timer reached as each pulse came in from the Pulse Generator. Knowing that we set the Pulse Generator to 2ms pulses we would expect the counter to have 2ms in between each printout as there are 2ms in between each pulse.



To test this, select any two consecutive readings. Keep in mind that your numbers will differ from mine, but the results should be similar. I'll use the first two readings here.

The first pulse occurred at 48,604 ticks, and the next pulse came in at 15,065 ticks.

We know that the timer counts up to 65,535 before resetting to 0. To determine how many ticks occurred between each pulse and calculate the elapsed time, subtract the first reading from the maximum tick count, then add the second reading.

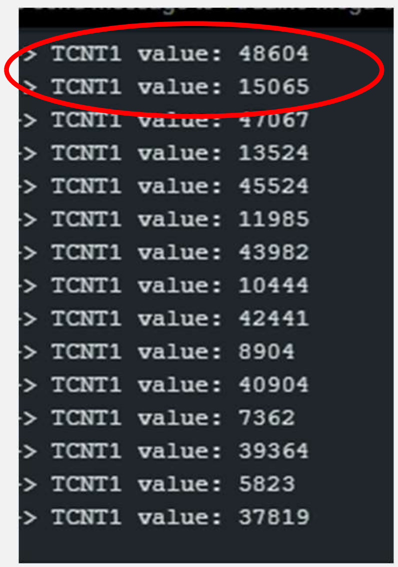
65,535 – 48,604 = 16,931 ticks before the timer count reset to 0

0 + 15,065 = 15,065 ticks after the timer count reset to 0

Arduino Timers

Testing Arduino Timer Accuracy

Recalling the calculation we did before to demonstrate the duration of each tick was 62.5 nanoseconds, we can now determine the accuracy of our Arduino Timer. Again, it should be 2ms (equivalent to 2 million nanoseconds) as was set on the Pulse Generator.



16,391 ticks before the timer count reset to 0

+ 15,065 ticks after the timer count reset to 0

31,996 = ticks in between each pulse

$$\frac{\textbf{1 second}}{\textbf{16,000,000 ticks}} = \textbf{62.5} \text{ nanoseconds per tick}$$

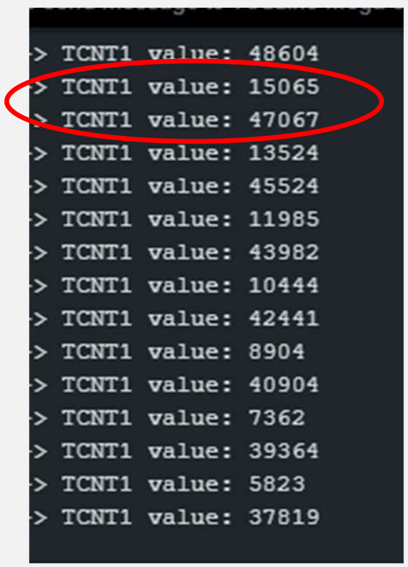
31,996 x **62.5 = 1,999,750** nanoseconds

* As you can see the reading is extremely close to perfect, off by only 250ns.

Arduino Timers

Testing Arduino Timer Accuracy

Keep in mind that if we had chosen a different set of consecutive readings, one where the counter didn't reset, like the readings circled below, you could simply subtract the smaller reading from the larger one and multiply the difference by 62.5 to get your final result.



47,067 – 15,065 = 32,002 ticks in between each pulse

32,002 x **62.5 = 2,000,125 nanoseconds between each pulse**

* Again the reading is extremely close to what we expected at 2ms.
* To convert nanoseconds to milliseconds you can divide the number by one million.

Flying Solo:
Perform this experiment twice more with the following settings for your pulse generator and test the Arduino's accuracy before moving on to the next slide:

- 3ms period
- 1ms period

Arduino Timers

Higher Frequencies / Shorter Periods

You may have noticed that your 1ms calculations may appear inaccurate, with the difference of the TCNT1 counts exceeding expected values by over 50%.

The baud rate of your Arduino determines how fast data is transmitted (in bits per second) and must align with the pulse signal frequency being measured. If the baud rate is too low, the Arduino may struggle to keep up with rapid signal changes, resulting in data loss or inaccuracies.

With a 1 kHz signal (1ms period), increasing the baud rate to 230,400 can improve accuracy by ensuring the Arduino processes data quickly enough to match the pulse frequency. Optimal results require matching the baud rate to your signal's frequency and system needs. Experiment with different baud rates while considering factors like noise, interference, and hardware quality.

Arduino Timers

Higher Frequencies / Shorter Periods

Test out a higher baud rate for the 1ms period and see if it brings you more accurate results.

Don’t forget to change the line of code responsible for setting the baud rate too! It should match the baud rate setting in the IDE.

```
// All Arduino programs must contain a setup() and loop() functions
void setup() {
 Serial.begin(115200); // Starts the serial monitor, sets baudrate to "115200" BPS
 pinMode(triggerPin,INPUT); // Sets the digital pin 2 as an input
 delay(1000); // Pauses the program for one second at the moment of open
 // Initializes the Timer1 registers (16-bit timer -- counts from 0 to 65535 ad nauseam). Time
 // Timer1 is a 16-bit timer, so the timer will increase its value until it reaches its maximu
  TCCR1A = 0; // Sets entire TCCR1A--Timer1 Control Register A--to 0
  TCCR1B = 0; // Timer 1 Control Register B set to 0 (The physical address of timer1)
  TCCR1C = 0; // Timer 1 Control Register C set to 0
  TCNT1 = 0; // Initialize timer/counter 1's value to 0
```

Baud Rates

Baud Rates

Knowing that a higher baud rate results in faster data transmission, you might be tempted to always set a high baud rate. This does, however, come with drawbacks.

Increased Susceptibility to Noise: High baud rates make communication more sensitive to electrical noise and interference, which can lead to corrupted data. This is especially problematic in environments with significant electromagnetic interference (EMI).

Higher Error Rate Over Long Distances: If the Arduino is communicating over a longer cable, higher baud rates are more likely to encounter errors due to signal degradation.

Baud Rates

Baud Rates

Knowing that a higher baud rate results in faster data transmission, you might be tempted to always set a high baud rate. This does, however, come with drawbacks.

Excessive CPU Overhead: A high baud rate increases the frequency at which the Arduino's processor handles serial interrupts, leaving less processing power for other tasks and potentially slowing down the system.

Waste of Resources: For longer signal periods or slower signals, a high baud rate is unnecessary and inefficient. It forces the system to process more data than required, which can be wasteful in terms of energy and processing time.

When writing code for the Arduino, it's important to experiment with various baud rates to determine the one that delivers the most accurate readings.

Module IX

Retrieving NMEA Data from the GPS Breakout v.3

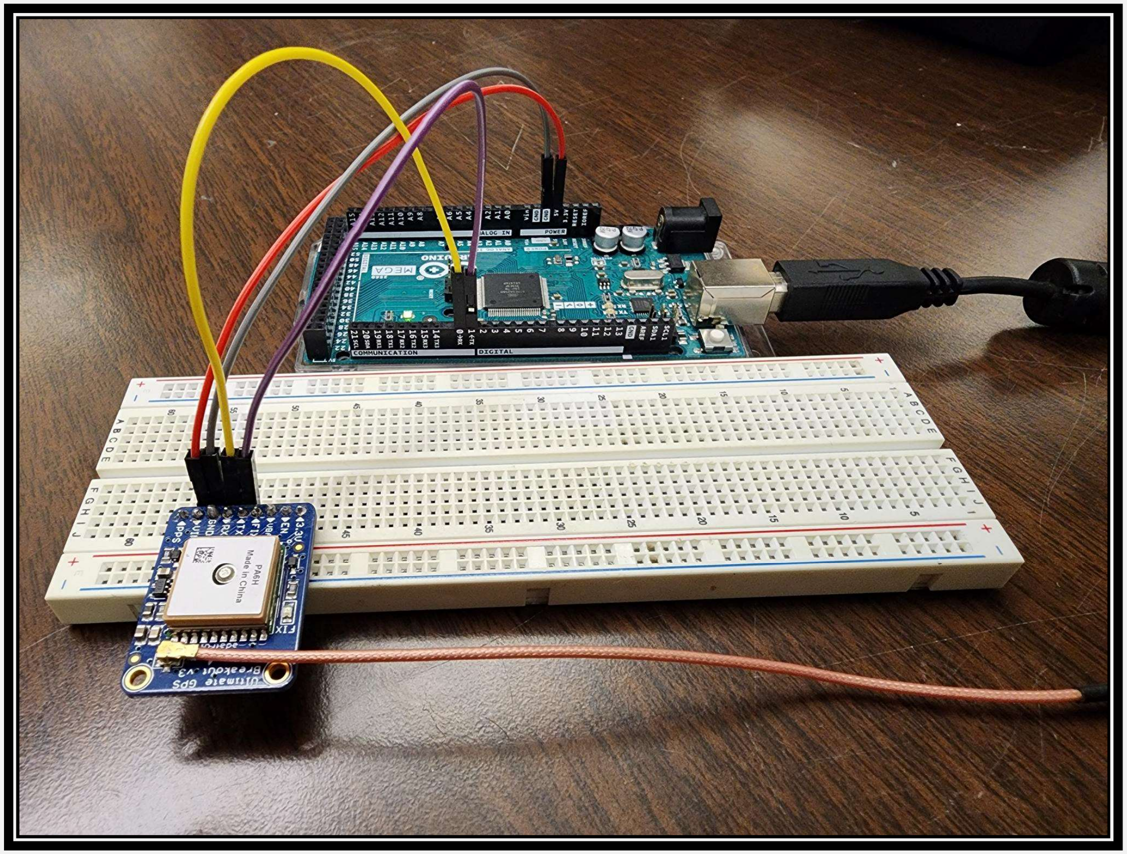
Retrieving NMEA Data

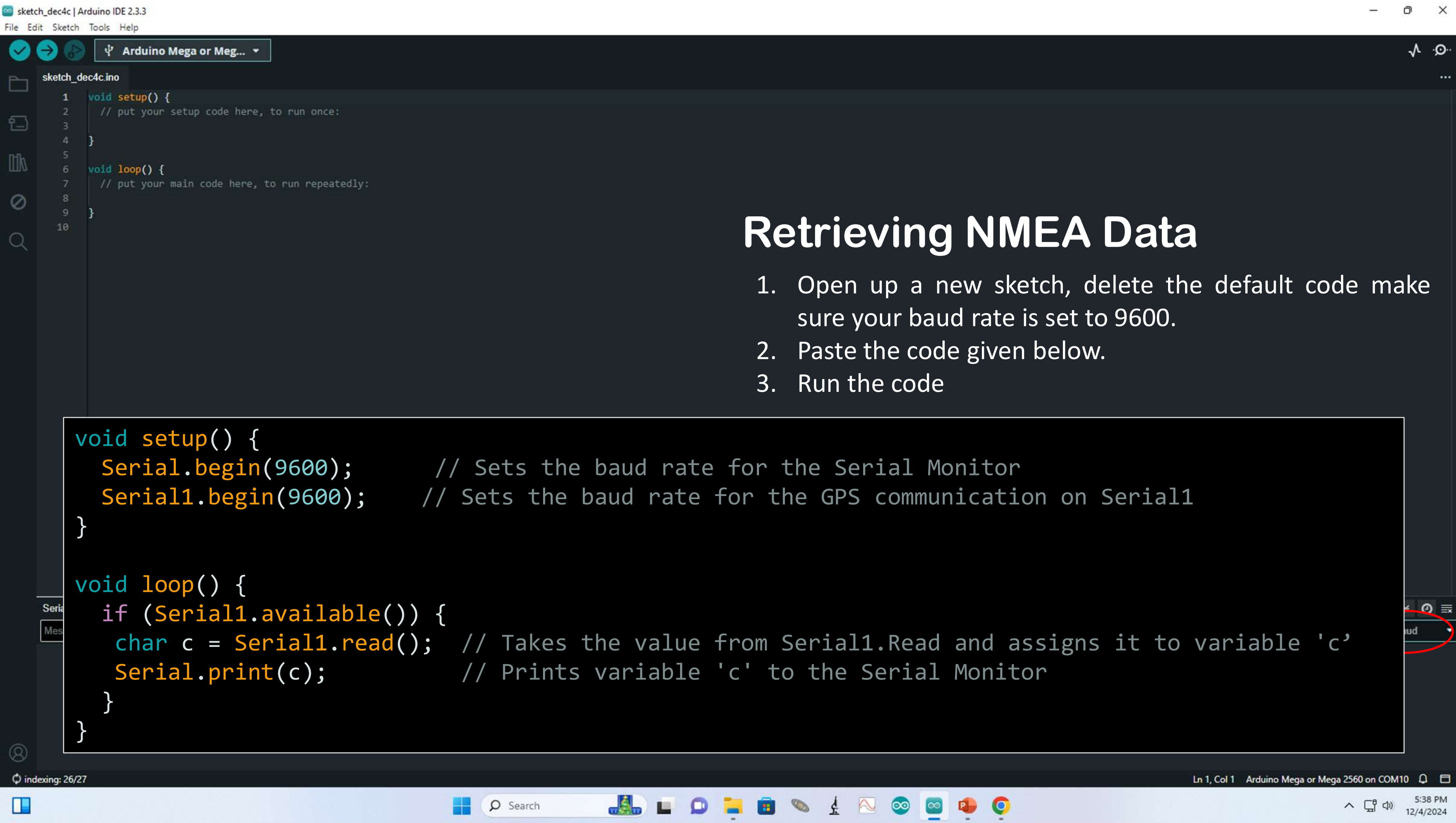
Setup

Here is what your setup should look like! Make the following connections:

Connections

Arduino	GPS
(Power) GND	GND
(Power) 5V	VIN
(Comm) TX1	RX
(Comm) RX1	TX





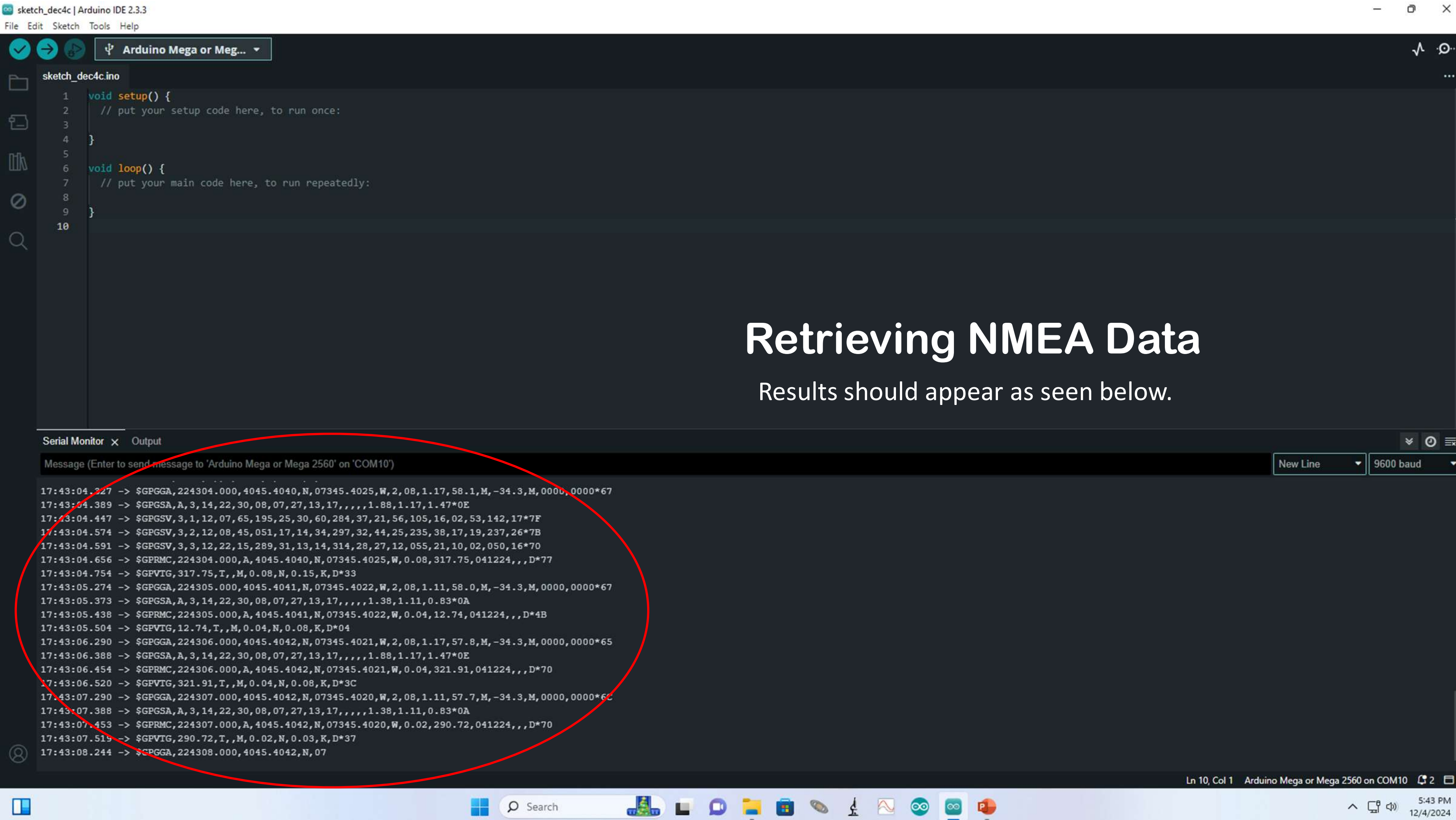
Retrieving NMEA Data

1. Open up a new sketch, delete the default code make sure your baud rate is set to 9600.
2. Paste the code given below.
3. Run the code

```
void setup() {
  Serial.begin(9600);       // Sets the baud rate for the Serial Monitor
  Serial1.begin(9600);    // Sets the baud rate for the GPS communication on Serial1
}

void loop() {
  if (Serial1.available()) {
   char c = Serial1.read();  // Takes the value from Serial1.Read and assigns it to variable 'c’
   Serial.print(c);          // Prints variable 'c' to the Serial Monitor
  }
}
```

Retrieving NMEA Data

Results should appear as seen below.



Understanding NMEA Data

Sentence Types Output by Adafruit GPS Breakout v.3

As you may have noticed in your results, there are the five sentence types that the Adafruit Ultimate GPS Breakout v.3 is capable of transmitting.

Sentence Type	Data Type
GGA	Global Positioning System Fixed Data
GSA	GNSS DOP and Active Satellites
GSV	GNSS Satellites in View
RMC	Recommended Minimum Specific GNSS Data
VTG	Course Over Ground and Ground Speed

*GNSS stands for Global Navigation Satellite System.

Understanding NMEA Data

GPGGA (GGA Sentence Type)

GPGGA Sentence Types follow this sequence and format, printed in order from left to right, and separated by commas.

Data	Example	Unit	Description
Message ID	$GPGGA		GGA Protocol Header
UTC Time	080754.000		hhmmss.sss
Latitude	3342.6618		ddmm.mmmm
N/S Indicator	N		N = North, S = South
Longitude	11751.3858		dddmm.mmmm
E/W Indicator	W		E = East, W = West
Position Fix Indicator	1		Range 0 – 6, * See Position Fix Indicator Table (next slide)*
Satellites Used	10		Range 0 - 12
HDOP	1.2		Horizontal Dilution of Precision
MSL Altitude	27.0	Meters	
Units	M	Meters	
Geoid Separation	-34.2	Meters	Geoid-to-ellipsoid separation. Ellipsoid altitude = MSL Altitude + Geoid Separation
Units	M	Meters	Null fields when DGPS is not used
Age of Diff. Corr.		Seconds	
Diff. Ref. Station ID	0000		
Checksum	*5E		

Understanding NMEA Data

GPGGA (GGA Sentence Type)

These are the values and meanings for the Position Fix indicator used in the GPGGA Sentence Type

Position Fix Indicator

Value	Description
0	Fix Not Available
1	GPS SPS Mode, Fix Valid
2	Differential GPS (DGPS), SPS Mode, Fix Valid
3 - 5	Not Supported
6	Dead Reckoning Mode, Fix Valid

Understanding NMEA Data

GPGGA (GGA Sentence Type Example)

Here is an example of a GGA sentence type from the previous output

Sentence Breakdown

$GPGGA, 224304.000, 4045.4040, N, 07345.4025, W, 2, 08, 1.17, 58.1, M, -34.3, M, 0000, 0000*67

- $GPGGA — Message ID
- 224304.000 — Timestamp (10:43:04.000)
- 4045.4040, N — Latitude
- 07345.4025, W — Longitude
- 2 — Position Fix Indicator
- 08 — Satellites Used
- 1.17 — HDOP
- 58.1 — MSL Altitude
- M — Unit
- -34.3 — Geoid Separation
- M — Unit
- 0000 — Age of Diff. Corr.
- 0000 — Diff. Ref. Station ID
- *67 — Checksum

Understanding NMEA Data

GPGSA (GSA Sentence Type)

GPGSA Sentence Types follow this sequence and format, printed in order from left to right, and separated by commas.

Data	Example	Unit	Description
Message ID	$GPGSA		GSA Protocol Header
Mode 1	A		A or M, *See Mode 1 table*
Mode 2	3		Range 1 – 3, *See Mode 2 table*
Satellite Used	07		SV on Channel 1
Satellite Used	02		SV on Channel 2
…			
Satellite Used			SV on Channel 12
PDOP	1.8		Position Dilution of Precision
HDOP	1.0		Horizontal Dilution of Precision
VDOP	1.5		Vertical Dilution of Precision
Checksum	*33		

Understanding NMEA Data

GPGSA (GSA Sentence Type)

These are the values and meanings for the Mode 1 and Mode 2 entries from the previous table.

Mode 1

Data	Description
M	Manual – forced to operate in 2D or 3D mode
A	2D Automatic – allowed to automatically switch 2D/3D

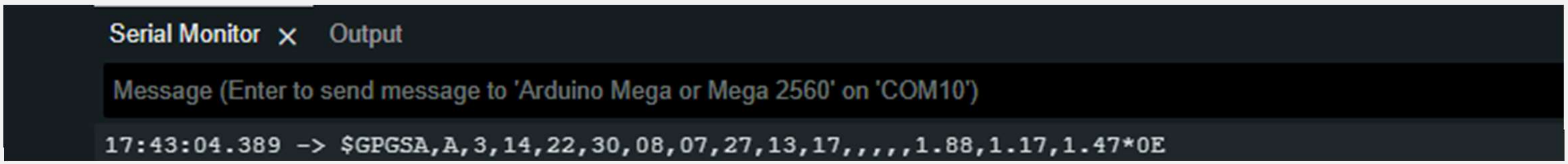
Mode 2

Data	Description
1	Fix not available
2	2D (<4 SVs used)
3	3D (<3 SVs used

Understanding NMEA Data

GPGSA (GSA Sentence Type Example)

Here is an example of a GSA sentence type from the previous output



Sentence Breakdown

Message ID Mode 2 PDOP VDOP

$GPGSA, A, 3, 14, 22, 30, 08, 07, 27, 13, 17,,,,, 1.88, 1.17, 1.47*0E

Mode 1 Satellites Used HDOP Checksum

Understanding NMEA Data

GPGSV (GSV Sentence Type)

GPGSV Sentence Types follow this sequence and format, printed in order from left to right, and separated by commas.

Note the ellipsis in the left most column indicates that the information repeats until all Satellite ID’s and associated information are given.

Data	Example	Unit	Description
Message ID	$GPGSV		GSV Protocol Header
Number of Messages	2		Range 1-3
Message Number	1		Range 1-3
Satellites in View	08		
Satellite ID	07		Channel 1 (Range 1-32)
Elevation	79	Degrees	Channel 1 (Max. 90)
Azimuth	048	Degrees	Channel 1 (True, Range 0 – 359)
SNR (C/N0)	42	dBHz	Range 0 – 99, null when not tracking
….			
Satellite ID	27		Channel 3 (Range 1-32)
Elevation	21	Degrees	Channel 3 (Max. 90)
Azimuth	138	Degrees	Channel 3 (True, Range 0 – 359)
SNR (C/N0)	35	dBHz	Range 0 – 99, null when not tracking
Checksum			

Understanding NMEA Data

GPGSV (GSV Sentence Type Example)

Here is an example of a GSV sentence type from the previous output

```
Serial Monitor ×   Output
Message (Enter to send message to 'Arduino Mega or Mega 2560' on 'COM10')
17:43:04.447 -> $GPGSV,3,1,12,07,65,195,25,30,60,284,37,21,56,105,16,02,53,142,17*7F
17:43:04.574 -> $GPGSV,3,2,12,08,45,051,17,14,34,297,32,44,25,235,38,17,19,237,26*7B
17:43:04.591 -> $GPGSV,3,3,12,22,15,289,31,13,14,314,28,27,12,055,21,10,02,050,16*70
```

Sentence Breakdown

Message ID   Message #   Satellite ID   Azimuth   Satellite ID   Azimuth   Satellite ID   Azimuth   Satellite ID   Azimuth   Checksum

$GPGSV, 3, 1, 12, 07, 65, 195, 25, 30, 60, 284, 37, 21, 56, 105, 16, 02, 53, 142, 17*7F

# of Messages   Satellites in View   Elevation   SNR (C/N0)   Elevation   SNR (C/N0)   Elevation   SNR (C/N0)   Elevation   SNR (C/N0)

Understanding NMEA Data

GPRMC (RMC Sentence Type)

GPRMC Sentence Types follow this sequence and format, printed in order from left to right, and separated by commas.

Data	Example	Unit	Description
Message ID	$GPRMC		RMC Protocol Header
UTC Time	161229.487		hhmmss.sss
Status	A		A = Data Valid, V = Data not Valid
Latitude	3723.2475		ddmm.mmmm
N/S Indicator	N		N = North, S = South
Longitude	12158.3416		dddmm.mmmm
E/W Indicator	W		E = East, W = West
Speed Over Ground	.13	Knots	
Course Over Ground	309.62	Degrees	
Date	120598		ddmmyy
Magnetic Variation		Degrees	E = East, W = West
E/W Indicator	E		E = East
Mode	A		A = Autonomous, D = DGPS, E = DR
Checksum	*10		

Understanding NMEA Data

GPRMC (RMC Sentence Type Example)

Here is an example of a RMC sentence type from the previous output

Sentence Breakdown

$GPRMC, 224304.000, A, 4045.4040, N, 07345.4025, W, 0.08, 317.75, 041224,,,D*77

- $GPRMC — Message ID
- 224304.000 — UTC Time
- A — Status
- 4045.4040, N — Latitude
- 07345.4025, W — Longitude
- 0.08 — Speed Over Ground
- 317.75 — Course Over Ground
- 041224 — Date
- D — Mode
- *77 — Checksum

Understanding NMEA Data

GPVTG (VTG Sentence Type)

GPVTG Sentence Types follow this sequence and format, printed in order from left to right, and separated by commas.

Data	Example	Unit	Description
Message ID	$GPVTG		RMC Protocol Header
Course	309.62	degrees	Measured Heading
Reference	T		True
Course		degrees	Measured heading
Reference	M		Magnetic
Speed	0.13	knots	Measured horizontal speed
Units	N		Knots
Speed	0.2	km/hr	Measured horizontal speed
Units	K		Kilometers per hour
Mode	A		A = Autonomous, D = DGPS, E = DR
Checksum	*23		

Understanding NMEA Data

GPVTG (VTG Sentence Type Example)

Here is an example of a RMC sentence type from the previous output

Sentence Breakdown

Message ID · Reference · Speed · Speed · Mode

$GPVTG, 317.75, T, , M, 0.08, N, 0.15, K, D*77

Course · Reference · Units · Units · Checksum

Arduino Timers

Identifying True Arduino Timer Tick Period

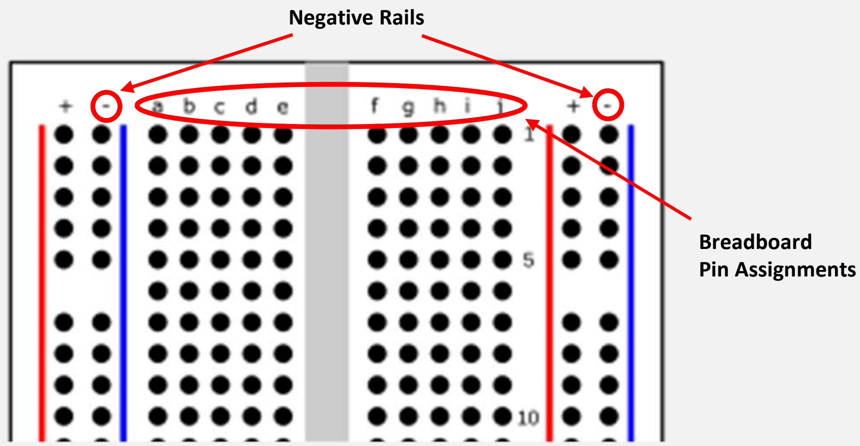
Now we will be testing for the True Arduino Timer Tick Period.

To do this, disconnect all of your previous jumper wires and reconfigure it accordingly:

Connections

Arduino	Breadboard	Breadboard
(Power) GND	Negative Rail	
(PWM) 11	A10	
(PWM) 2	B10	
	E10 (Resistor)	F10 (Resistor)
	J10	Negative Rail

* Note for the E10 and F10 pins on the breadboard, you must connect them using a resistor, not a jumper wire.



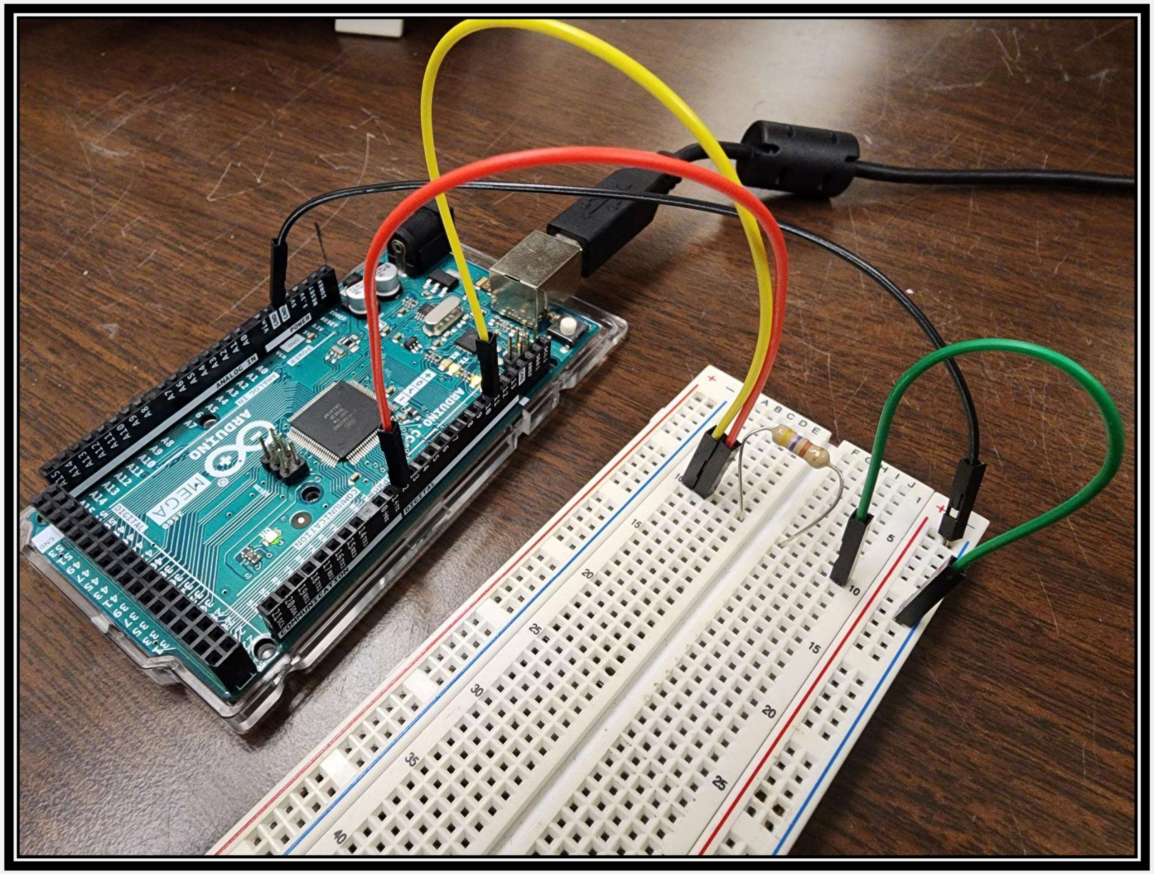
Pulse Generator & Oscilloscope

Setup

Here is what your setup should look like!

Connections

Arduino	Breadboard	Breadboard
(Power) GND	Negative Rail	
(PWM) 11	A10	
(PWM) 2	B10	
	E10 (Resistor)	F10 (Resistor)
	J10	Negative Rail



Testing the accuracy of the Arduino Timer

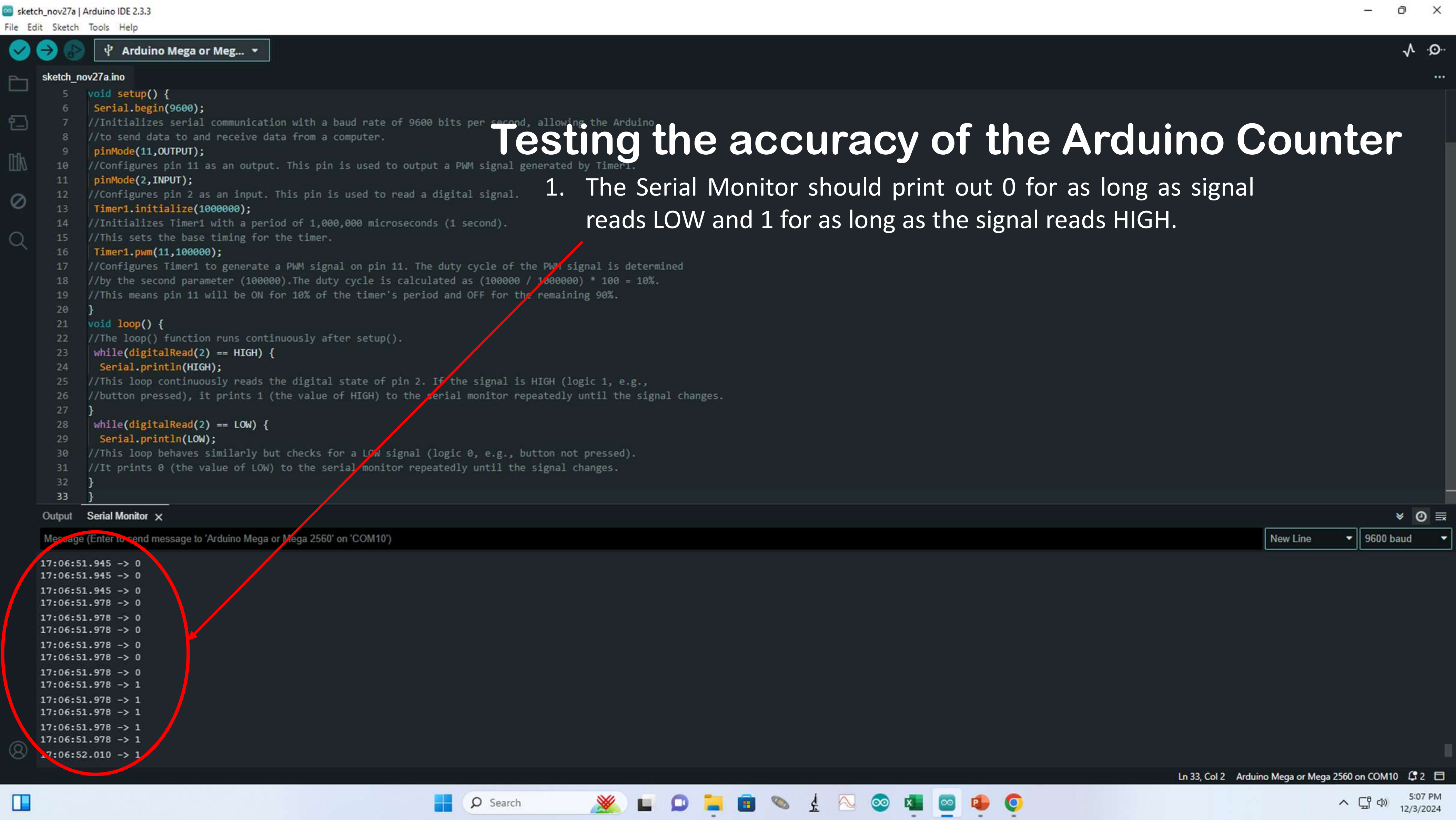
```
#include <TimerOne.h>
//This line includes the TimerOne library, which provides functions for configuring and controlling
//Timer1 on Arduino boards. This is commonly used for precise timing tasks like generating PWM signals
//or periodic interrupts.
void setup() {
 Serial.begin(9600);
//Initializes serial communication with a baud rate of 9600 bits per second, allowing the Arduino
//to send data to and receive data from a computer.
 pinMode(11,OUTPUT);
//Configures pin 11 as an output. This pin is used to output a PWM signal generated by Timer1.
 pinMode(2,INPUT);
//Configures pin 2 as an input. This pin is used to read a digital signal.
 Timer1.initialize(1000000);
//Initializes Timer1 with a period of 1,000,000 microseconds (1 second).
//This sets the base timing for the timer.
 Timer1.pwm(11,100000);
//Configures Timer1 to generate a PWM signal on pin 11. The duty cycle of the PWM signal is determined
//by the second parameter (100000).The duty cycle is calculated as (100000 / 1000000) * 100 = 10%.
//This means pin 11 will be ON for 10% of the timer's period and OFF for the remaining 90%.
}
void loop() {
//The loop() function runs continuously after setup().
 while(digitalRead(2) == HIGH) {
  Serial.println(HIGH);
//This loop continuously reads the digital state of pin 2. If the signal is HIGH (logic 1, e.g.,
//button pressed), it prints 1 (the value of HIGH) to the serial monitor repeatedly until the signal changes.
}
 while(digitalRead(2) == LOW) {
  Serial.println(LOW);
//This loop behaves similarly but checks for a LOW signal (logic 0, e.g., button not pressed).
//It prints 0 (the value of LOW) to the serial monitor repeatedly until the signal changes.
}
}
```

1. Open up a new sketch, delete the default code that appears in the new window.
2. Copy and paste following code in its place:
3. Note the baudrate here is 9600 as seen in the line of code "Serial.begin(9600)". Ensure the baud rate in your IDE is set to the same value.
4. Run the code with the arrow icon in the top left.

* I recommend reading the comments left in the code to understand the instructions being sent to the Arduino.

Testing the accuracy of the Arduino Counter

1. The Serial Monitor should print out 0 for as long as signal reads LOW and 1 for as long as the signal reads HIGH.

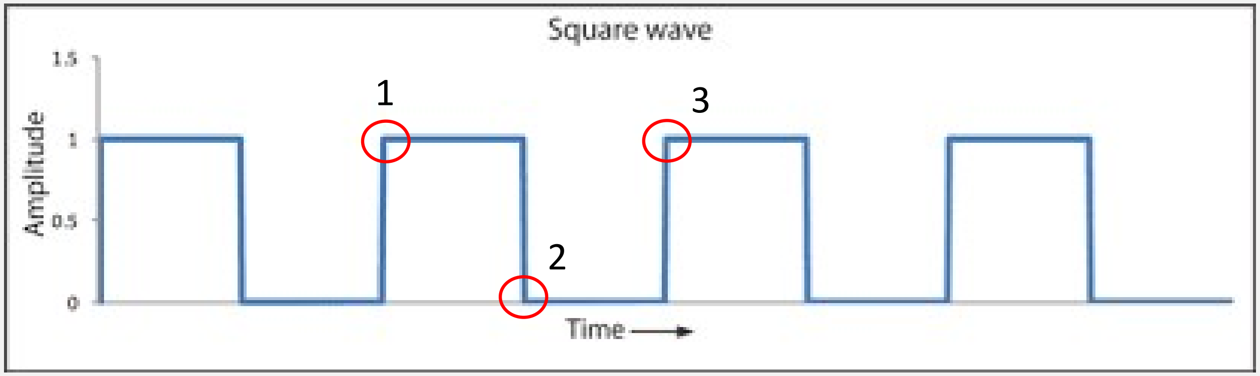


Arduino Timers

Further Testing of Arduino Timer Accuracy

As the comments in the code described, we set Timer1 to send a pulse once every second that has a pulse width of 10%. So our expected results are that the pulses come in every second, and remain in the ON (HIGH) state for a tenth of a second (100 milliseconds).

To find this out, we must log three timestamps as outlined below:



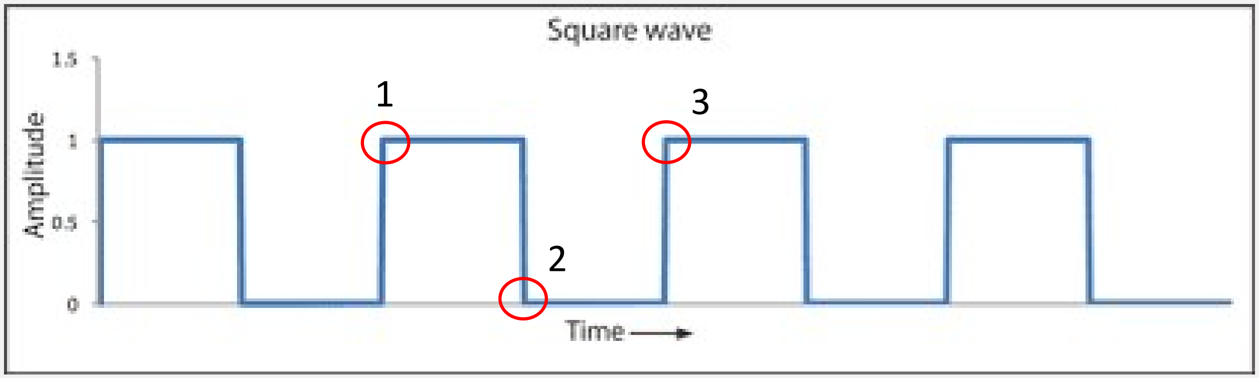
1. The timestamp when it changes from 0 to 1.
2. The timestamp of the very next instance it changes back from 1 to 0.
3. The timestamp of the very next instance it changes back from 0 to 1.

*With this data we can calculate how accurate the Arduino is.

To help visualize the information that we are taking, I've included this chart above.

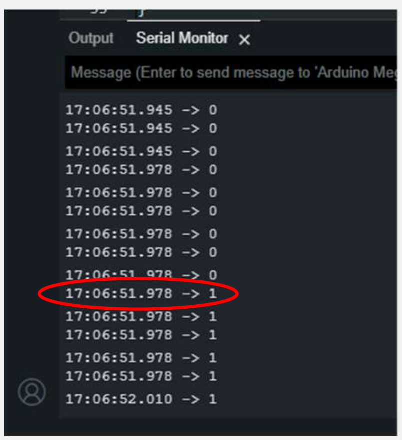
Arduino Timers

Further Testing of Arduino Timer Accuracy

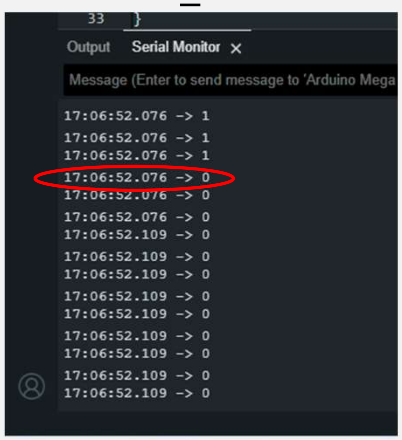


1. The difference between points 1 and 2 will give us the time the signal remained in the ON (HIGH) state.
2. The difference between points 1 and 3 will give us the time that passes in between pulses.

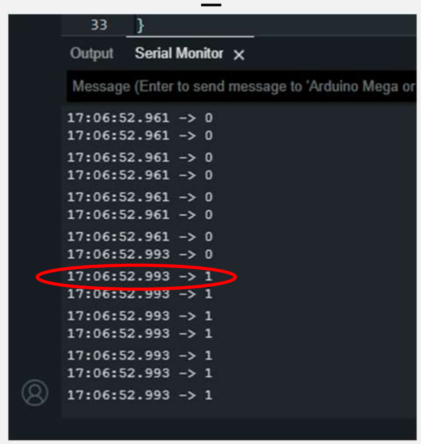
1



2



3



These were the timestamps that I collected. Obviously, your numbers will be different but the math remains the same.
As the time stamps are all from within the same minute, we can drop that when doing our calculations.

Arduino Timers

Further Testing of Arduino Timer Accuracy

No.	Timestamp
1	51.978
2	52.076
3	52.993

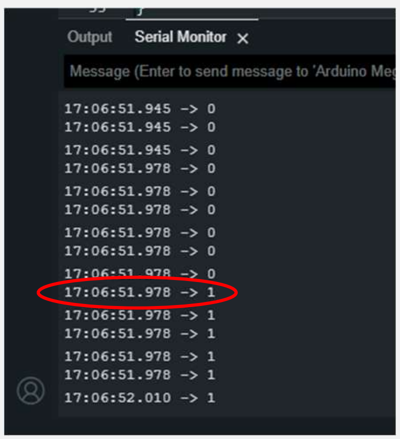
Find the **pulse width** (subtract timestamp 1 from timestamp 2)

52.076 – 51.978 = .098 seconds (98 milliseconds)

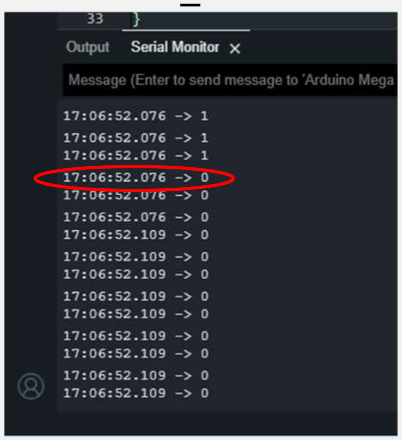
Find the **pulse period** (subtract timestamp 1 from timestamp 3)

52.993 – 51.978 = 1.015 seconds

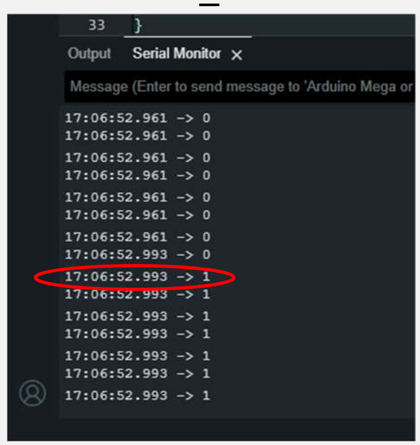
1



2



3



Remember, we were expecting a pulse period of 1 second and a pulse width of a tenth of a second (100 milliseconds).
The results are close to what was expected but now we want to quantify the deviation observed here.

Integrating Putty in Timer Testing

To quantify the deviation, we need more than one datapoint. We need a sample set of datapoints that we can use to determine the average deviation across. To do this, we use Putty to extract the information and from there we export it to Excel.

Step 1: Close the Serial Monitor in the Arduino IDE.

Step 2: Click *Open* in the Putty window.