Arduino Mega coding tutorials to use with the cosmic ray detector DAQ electronics

July 1, 2024

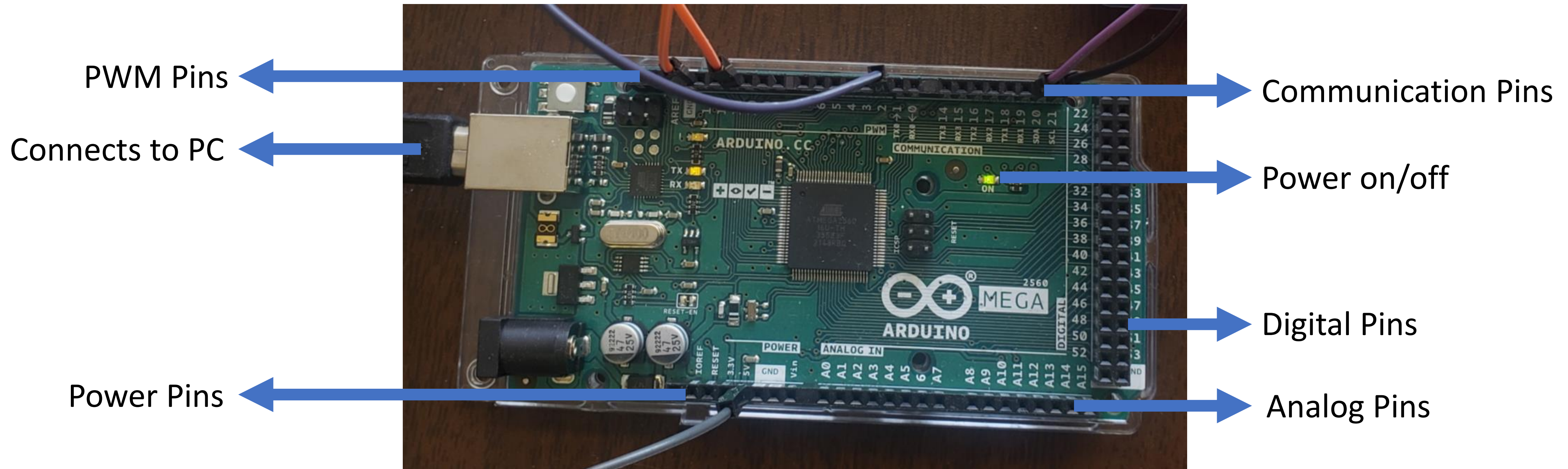
Fatima Saleem / Sonia Rostami + Lily Carter

Queensborough Community College

Mentor: R. Armendariz

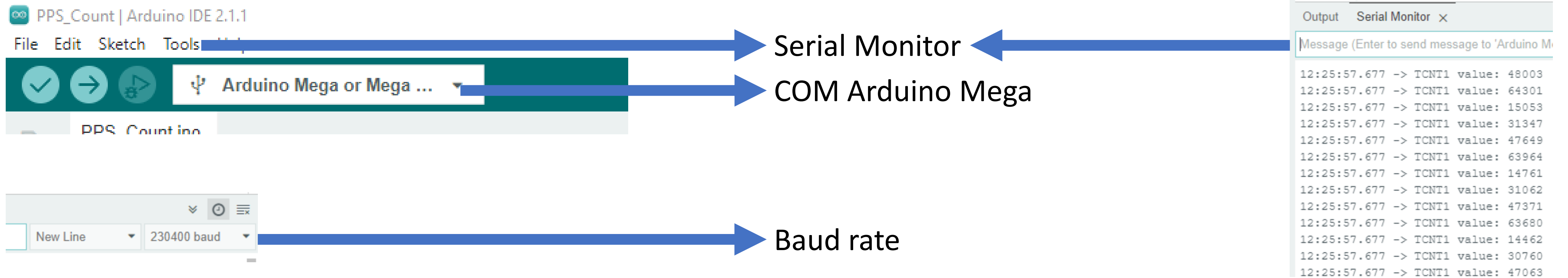
What is Arduino?

Arduino is a micro-controller that can be programmed to perform various tasks. It can be used for creative projects, like making things light up, move, or sense the environment. Arduino boards have different pins, including VIN, GND, SCK, etc. Jumper wires and a breadboard are used to connect Arduino to small devices like GPS, BMP, and larger devices like a pulse generator. The integrated project is controlled by code written using the Arduino IDE.



Arduino IDE

In my Arduino workflow, I relied on the Arduino IDE and libraries for coding experiments. I simply copy and paste provided codes, adjusting pins as needed. Verification involves checking COM settings and using the 'Tools' menu to access the Serial Monitor for real-time outputs, such as TCNT1 counts in Timer1 Verification. With inputs from the Arduino Mega, the code is uploaded, ensuring a smooth execution of experiments. You can adjust the baud rate to match the communication speed between the Arduino and the computer, allowing for optimal data transfer and monitoring during experimentation.



How to troubleshoot Arduino IDE

If you encounter unexpected outputs or error messages in the Arduino IDE, follow these steps to troubleshoot:

1. **Serial Connection:**
   - Select the correct COM port in the Arduino IDE under the 'Tools' menu.
   - Ensure the baud rate in the Serial Monitor matches your Arduino code.
2. **Hardware Connection:**
   - Double-check jumper wire connections as per instructions.
   - Confirm proper power supply to both the Arduino and the Integrated System.
3. **Library Installation:**
   - Verify successful installation of required libraries using the Library Manager.
4. **Code Review:**
   - Carefully inspect the copied code for syntax errors or logic issues.
   - Check for conditions in the code that may not be met.
5. **Serial.println() Statements:**
   - Confirm the proper functioning of Serial.println() statements.
   - Ensure the code reaches these statements by addressing any conditions preventing it.
6. **Serial Monitor Settings:**
   - Adjust 'Carriage return' or 'Newline' options in the Serial Monitor.
7. **Debugging:**
   - Insert additional Serial.println() statements for debugging at critical code points.

Brief explanation of bits and registers

In some of the code, we will be modifying certain registers in the microprocessor (in this case, the Atmel MEGA 2560, or ATmega2560). These registers control functions on the chip, and the one we are interested in is the Timer 1 function.
You can also read more about this here https://docs.arduino.cc/learn/programming/bit-math/

- Most registers we work with are 8-bit, meaning they have 8 1s and 0s, or can store 2^8 possible values
    - In a sequence like 1001 0101, the most significant digit is highlighted in red, whereas the least significant digit is blue.
    - The easiest way to get a 1 at a certain bit location is a bit shift to the left, represented as 1 << n in C++ code.
- The table below shows each bit shift value followed by the corresponding numerical value

7	6	5	4	3	2	1	0
128	64	32	16	8	4	2	1

- Note that each byte (8 bits) in also a number from 0-255, which is 256 possible values.
- The Arduino library has the function bit(n) that you can call to do this
- If you need multiple bits set in the same register, call bit(n) and combine them with the bitwise OR operator.
    - To get 1001 0000 we'd do bit(7) | bit(4)

To find the registers and what they do, we need to consult the datasheet. This can be found at https://www.microchip.com/en-us/product/atmega2560 in the documentation section. We will be citing the datasheet in this documentation, using the section of the datasheet in parentheses.

- Luckily for us, both the register names and the exact bit shifts for each register are already provided by the Arduino library because we selected the Arduino MEGA 2560 board
- For example, if we wanted to set the ICES4 and CS42 bits on the TCCR4B register, we'd just have to put

```
TCCR4B = bit(ICES4) | bit(CS42);
```

- In the setup code, and we'd be good to go!

Interrupts!

In embedded programming, we have a thing called an interrupt!

- You use interrupts everyday, for example, when you press a key on a keyboard. This sends an interrupt to the CPU on whatever computer you are using, and the operating system (the code that is always running, such as Windows, macOS, or Linux) has to handle it.

Every time an interrupt is triggered, an Interrupt Service Routine (ISR) is called. This is essentially a function that is run when the interrupt flag is set.

- To handle an interrupt with the Arduino MEGA 2560 in bare code, we'd need to declare a function ISR(INTERRUPT_NAME_vect), where INTERRUPT_NAME is defined in Table 14-1 in the datasheet, on page 102
- For many things in the Arduino library, there is a function like attachInterrupt(functionName), where functionName is another function declared somewhere else in the code, so we usually don't have to worry about this

This is a good resource to further your knowledge
https://gammon.com.au/interrupts

Things to be mindful of when using interrupts

- An interrupt is treated specially by the microprocessor. Because it is run outside of normal code, an interrupt cannot be triggered while we are in an interrupt (7.8).
- Since it is not in the normal code, sometimes the compiler will make optimizations that would break the code, so to tell it not to do that, we must use the volatile keyword when declaring a variable that we access in an interrupt
- When reading a multi-byte variable that we use in an interrupt, we should turn off interrupts using noInterrupts(); and then turn them back on immediately afterwards using interrupts(); so that the variable is not changed while we are reading it
  - A variable declared with int is 2 bytes, and a long is 4 bytes.
  - The Timer1 counter is 2 bytes so we have to be careful when reading from it

Measuring ambient pressure and temperature with the BMP280 sensor and Arduino Mega 2560

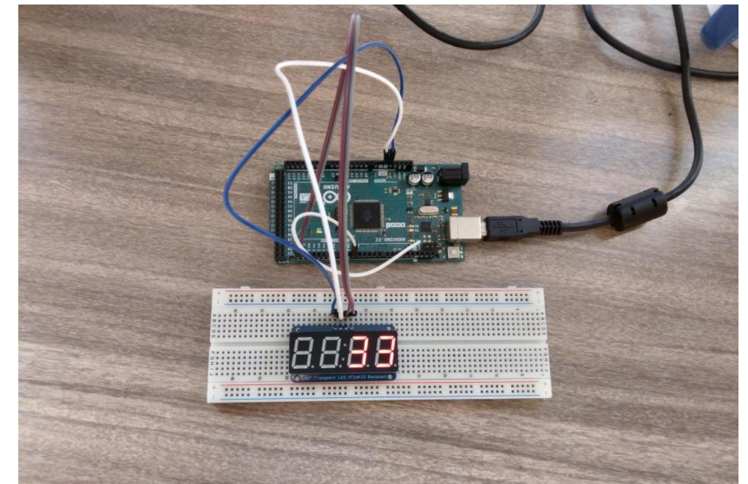
- Connect jumper wires from Arduino Power pins 5V and GND to BMP280 VIN and GND respectively (if code does not work try the other GND).
- Connect jumper wires from Arduino PWM pins 10, 11, 12, 13 to BMP280 pins CS, SDI, SDO, and SCK respectively
- Open a new Arduino sketch.
- Install the BMP280 library from the Arduino library manager.
- In the IDE go to FILE, EXAMPLES, BMP280 Library, open up bmp280test, a new sketch would open with the code.
- Towards the top of the code comment out “BMP280 bmp;” and uncomment “Adafruit_BMP280 bmp(BMP_CS, BMP_MOSI, BMP_MISO, BMP_SCK);” i.e. run //Adafruit_BMP280 bmp; // I2C //Adafruit_BMP280 bmp(BMP_CS); // hardware SPI Adafruit_BMP280 bmp(BMP_CS, BMP_MOSI, BMP_MISO, BMP_SCK);
- Make sure the serial monitor baud rate matches code baud rate which in this case is 9600. Highlighted is the baud rate command.

```
4   #include <BMP280.h>
5   Adafruit_BMP280 bmp;
6
7   void setup()
8   {
9     Serial.begin(9600);
10    delay(10);
11    Serial.println("BMP280 example");
12
13    Wire.begin(); //Join I2C bus
14    bmp280.begin();
15  }
```

Arduino code to use the LED Backpack counter (7 segment display) to count and display voltage pulses generated by TimerOne

- Connect jumper wire between:
  - Arduino 5V and 7 segment display pin +
  - Arduino SDA and 7 segment display pin D
  - Arduino SCL to 7 segment display pin C
- Install from Arduino Library manager.
- Adafruit_LEDBackpack.h and any needed associated libraries (search in library manager “Adafruit LED Backpack Library”) and TimerOne.h.
- Copy the code from next slide.
- When the code is running you should see in the Arduino Serial Monitor zeroes and ones appear as the LED counter counts

Libraries



```
#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();
}
```

Using Timer1 Functionality to count Arduino clock cycles to measure the 1 second time period between successive GPS Pulse-per-second pulses (PPS)

Background:

Pulse-per-second signal (PPS): An accurate electronic signal recurring every second, derived from the Adafruit GPS signal on the breadboard. The PPS signal lasts 50-100ms, pulsing high at 3.3V. Primarily used for time synchronization, with documented accuracy of 100-300 nanoseconds.

Note: Using the Serial Monitor for data capture relies on the system time (Windows).

- **PPS Program Objectives:**

PPS() function is called each time the PPS pulse rises, initiating Timer1 to increment up to 65,535. Overflows variable increments with each Timer1 overflow and is printed.
The program prints the number of Timer1 overflow occurrences between two PPS signals every second.
It also prints the number of clock cycles in 1 second, derived by multiplying Timer1 overflow occurrences by 2^16.
Overflow occurrences reset to 0 after 1 second; each line represents overflow occurrences between two PPS signals.

- **PPS Calculation Results:**

No actual results collected.
Considering a PPS signal frequency of 1 second

Understanding:

- These will help you understand the code without reading the datasheet

https://people.ece.cornell.edu/land/courses/ece3400/Timer1/timer1.html
https://forum.arduino.cc/t/capturing-timer-value-based-on-external-interrupt-leveraging-the-62-5nsec-16mhz-resolution-of-the-atmega328-on-the-arduino-uno/908807

Setup:

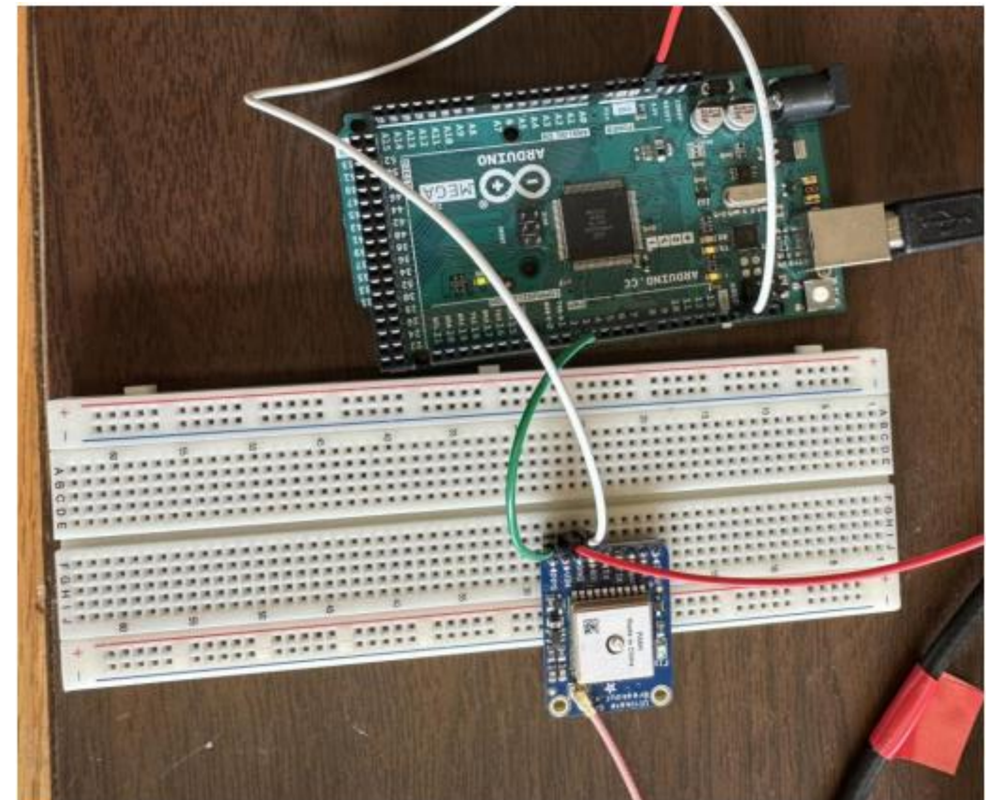
- Connect Arduino 5V, GND, PWM-2 to GPS VIN, GND, PPS respectively. (refer the right image for setup)
- Read the code from next two slides and understand what it does.
- Copy the code and run it.
- Check left image to see sample serial monitor output.

Output | Serial Monitor ×

Message (Enter to send message to 'Arduino Mega or Me... | Both NL & CR | 115200 baud

```
14:33:40.356 -> Overflow:244
14:33:40.356 -> Timer1:10133
14:33:40.356 -> ClockCycles:16000917
14:33:41.349 -> Overflow:244
14:33:41.349 -> Timer1:10131
14:33:41.349 -> ClockCycles:16000915
14:33:42.323 -> Overflow:244
14:33:42.323 -> Timer1:10132
14:33:42.323 -> ClockCycles:16000916
14:33:43.339 -> Overflow:244
14:33:43.339 -> Timer1:10132
14:33:43.339 -> ClockCycles:16000916
14:33:44.329 -> Overflow:244
14:33:44.329 -> Timer1:10133
14:33:44.329 -> ClockCycles:16000917
```

Ln 47, Col 2 Arduino Mega or Mega 2560 on COM7 2



```
#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;
  // Resets Timer1 Count
  TCNT1 = 0;
  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;
  }
}
```

Using Timer1 Functionality to count Arduino clock cycles to measure milliseconds between successive voltage pulses from a pulse generator (Timer1 Count Verification)

Explanation of Timer1:

- The Arduino Mega 2560 operates at a clock speed of 16 MHz.
- This results in a clock increment of 62.5 nanoseconds (1/16,000,000 seconds).
- Timer1 is an interrupt routine with a 16-bit length counter meaning it counts up to 2^{16} = 65,535 (including zero).
- Timer1 has a prescaler, which means it can count every 1, 8, 64, 256, or 1024 clock cycles (18.1).
- In Normal Mode, it counts to 65,535 units before overflowing back to 0 (17.9.1).
- Timer1 counts offer flexibility; they can confirm time elapsed between two outputs in the serial monitor.
- When Timer1 reaches 65,535, it has counted 4.096 milliseconds (65,536 counts)/(16,000,000 counts/sec)= 0.004096 sec)

Objective:

- Understand how to use Timer1 to measure the highest time resolution between pulses.

Important calculations:

- Difference in timer counts (consecutive counts without overflows) = TCNT1 count2 – TCNT1 count1
- Difference in timer counts (consecutive counts with overflows) = TCNT1 count after overflow + (65535 – TCNT1 count before overflow)
- Time Period = difference in timer counts/16,000,000

Setup:

- Pulse generator (Trigger) '+' and '-' to Arduino PWM2 and GND respectively.
- Set the pulse generator to the following settings one by one: (pulse generator explain on next slide)
  - 2ms and 500 Hertz
  - 3ms and 333.3 Hertz
  - 1ms and 1000 Hertz
- Perform this experiment with baud rate of 115200 bps.
- Copy the code from following slides.

Calculations:

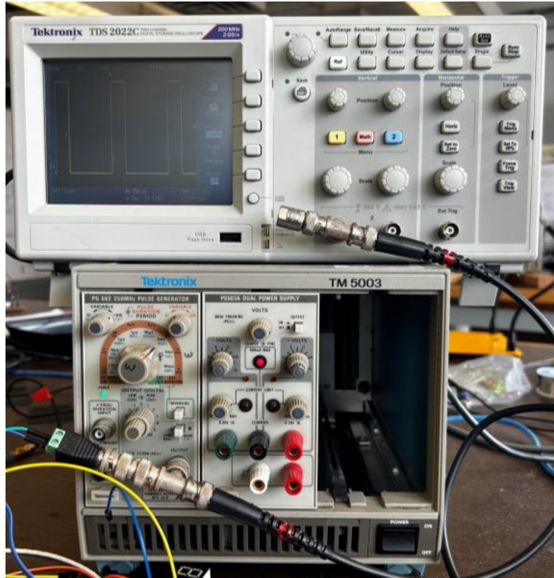
- Perform the calculations on previous slide to obtain a time periods closer to 2ms, 3ms, and 1ms.

Exception:

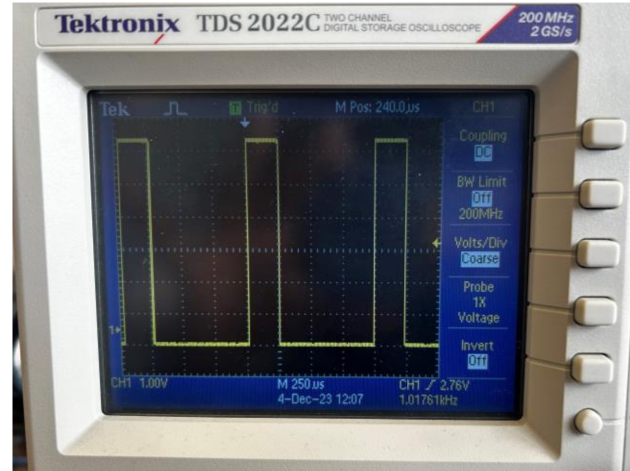
- You may observe that the readings for 1ms are inaccurate. The difference in TCNT1 counts obtained is nearly double the expected value. An explanation for this scenario and guidance on pulse generator settings are provided on the following slides after code.

Pulse Generator

Complete Setup



Oscilloscope Monitor



Pulse Generator

Orange knob for pulse duration, black knob for pulse period

Scaling

Zoom in and out of pulses for clarity

```
#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
}
```

Explanation for baud rate and exception for 1ms:

The Arduino baud rate to use depends on the pulse's signal frequency you are trying to measure with the Arduino; understanding your communication system is important. Baud rate is essentially the number of communications per second between the Arduino and the computer; while signal frequency is the rate at which the signal you are trying to measure oscillates. In the context of serial communication, baud rate is often used interchangeably with bits per second (bps).

When working with a pulse generator and serial communication, it's essential to consider the duration of each pulse and the time it takes for data bits to be transmitted. If the baud rate is set too low, it might not accurately capture and interpret each signal change, leading to inconsistent data reception.

In one case, setting the baud rate high, to 230,400, might provide better consistency because it allows for faster data transmission and more accurate interpretation of signal changes. If the signal frequency is 1 kHz (period is 1 ms), the time available for each signal change is limited, and a higher baud rate ensures that the serial communication can keep up with the rapid changes in the signal.

It's crucial to match the baud rate with the characteristics of the signal and the requirements of your communication system. Experimenting with different baud rates and observing the consistency of data reception is a common approach to finding the optimal configuration for your specific setup. Additionally, factors like noise, interference, and the quality of the communication hardware can also influence the choice of baud rate.