TESTING WIRELESS DATA TRANSMISSION WITH THE XBEE WIRELESS TRANSCEIVER

By Thomas Stocklin

Mentor: Raul Armendariz

5/2/25, QCC Physics Department

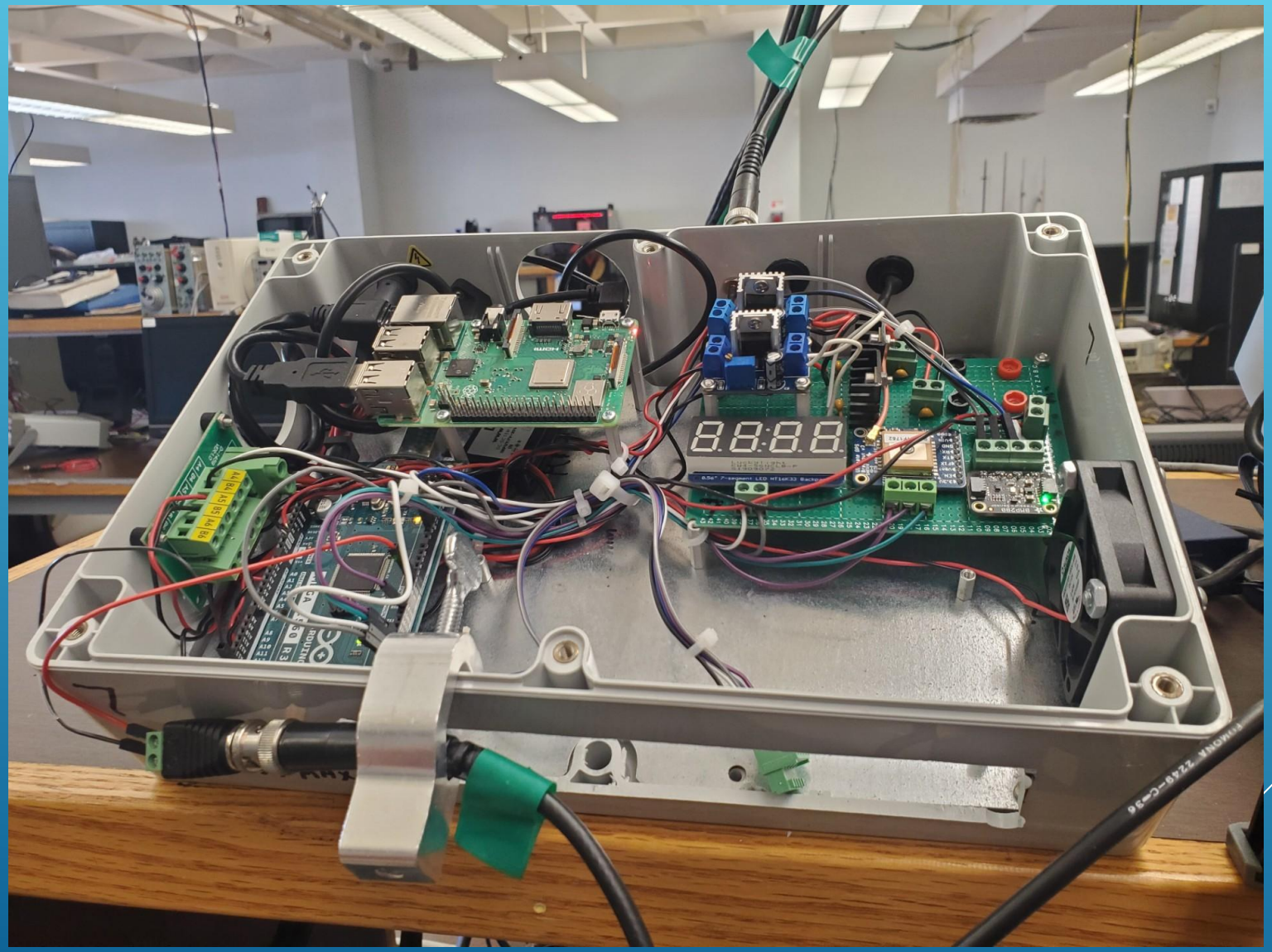
THE COSMIC RAY DETECTOR

- Currently, the Physics Department at Queensborough Community College is engaged in a project to build and test a series of Cosmic Ray Detectors. These devices take the form of small electronics boxes that house a data acquisition circuit (or DAQ). This circuit operates in conjunction with a plastic scintillator, photomultiplier tubes, and an Arduino microcontroller to to detect Muon showers, calculate their energy, and log the exact time of their appearance.
- In order to generate a timestamp for when these showers occur, the current DAQ configuration utilizes a pair of Adafruit GPS modules, one housed inside of the box and one outside. When a coincidence (that is, two signals occurring at the same time) is detected by the scintillator, a GPS antenna at a nearby window acquires a NMEA data timestamp from satellites orbiting the earth. This timestamp signal is then transmitted through a ceiling-mounted wire to the DAQ box and it's GPS receiver. This module then sends this data to the Arduino, and the Arduino then uses this data, alongside a periodic signal produced by the the GPS module, to calculate a more accurate timestamp for the shower, down to the microsecond.
- As this project continues, the Physics department is looking to expand placement of these boxes outside of the QCC Physics Lab, installing them at various locations around campus and even as far as Brookhaven National Laboratory in Suffolk County. Unfortunately, some of these locations have regulations that do not allow the use of ceiling-mounted wires. As a result, there has been ongoing research into substituting the wired connection with a wireless connection and, to that end, we he have been experimenting with the Xbee 3 Wireless Transceiver.

THE XBEE 3 WIRELESS TRANSCEIVER

- The XBee 3 was developed by Digi International and is primarily intended for use with the Arduino and other microcontroller systems. Lightweight and affordable, the this transceiver boasts an indoor transmission range of up to 100m, making it ideal for the limitations of this project.
- When the connected to the GPS module placed at the window, one XBee, programmed as a wireless transmitter, collects the NMEA data through a wired connection to the antenna and then converts that signal it into it's own unique frame data structure. The XBee will then send this data wirelessly to a second XBee which has been housed in the DAQ box in lieu of it's GPS module. This XBee will then pass this data on to the Arduino for processing.
- Photos of a prototype circuit that utilizes this model have been included in the next two slides for convenience. The first circuit simulates the GPS/XBee transceiver setup as described above, while the second simulates the receiver XBee and the Arduino components of the DAQ box.

Current Cosmic Ray Detector DAQ Box



XBee Transmitter Prototype Circuit

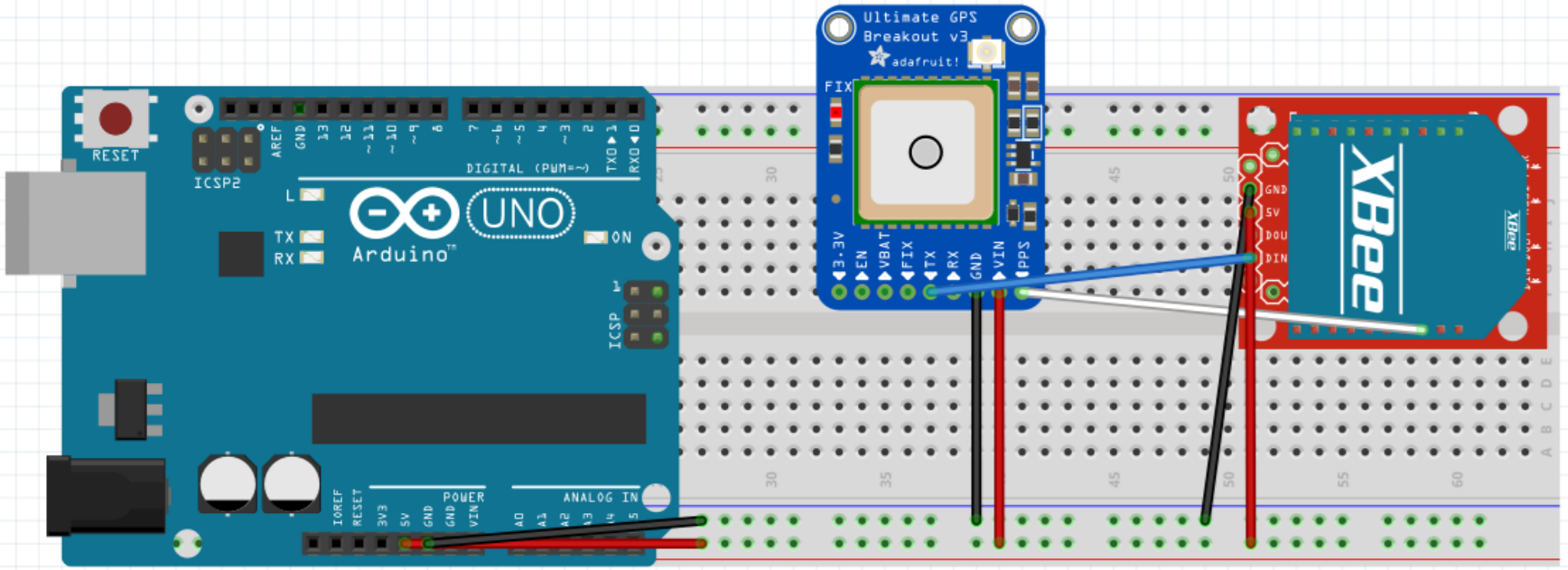
Transmitter Circuit Connections

Arduino 5V to Board Power
Arduino GND to Board GND

Board Power to GPS 5V
Board GND to GPS GND

Board Power to XBee 5V
Board GND to XBee GND

GPS TX to XBee DIN
GPS PPS to XBee DIO2

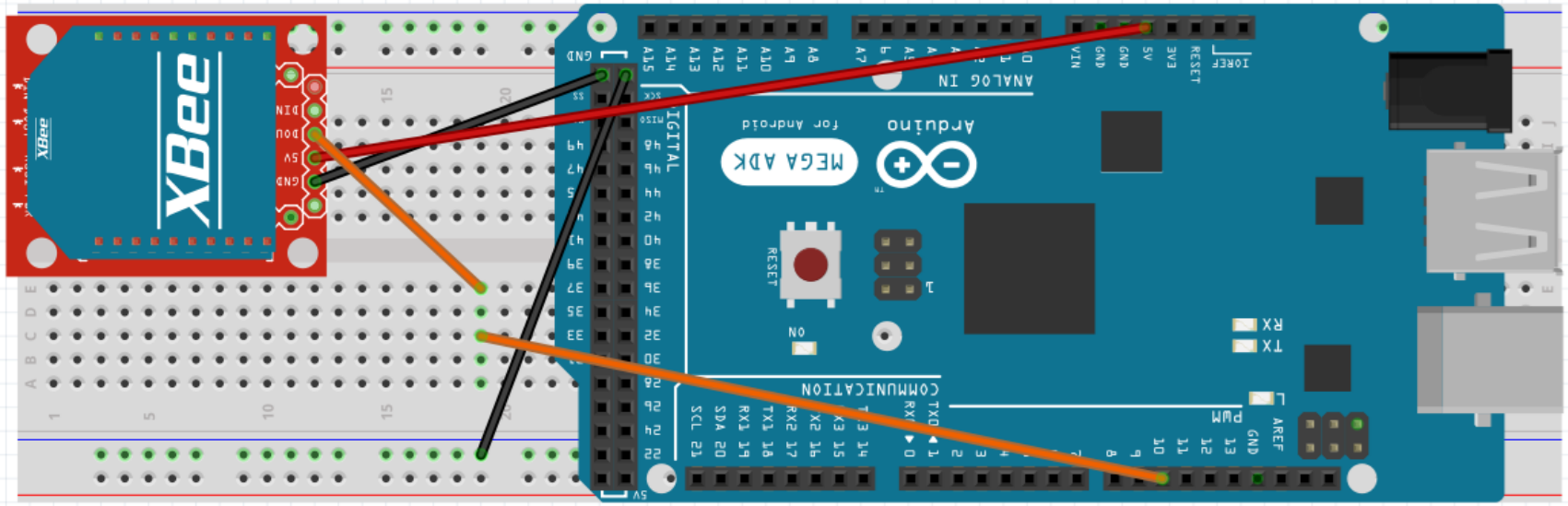


fritzing

XBee Receiver/DAQ Prototype Circuit

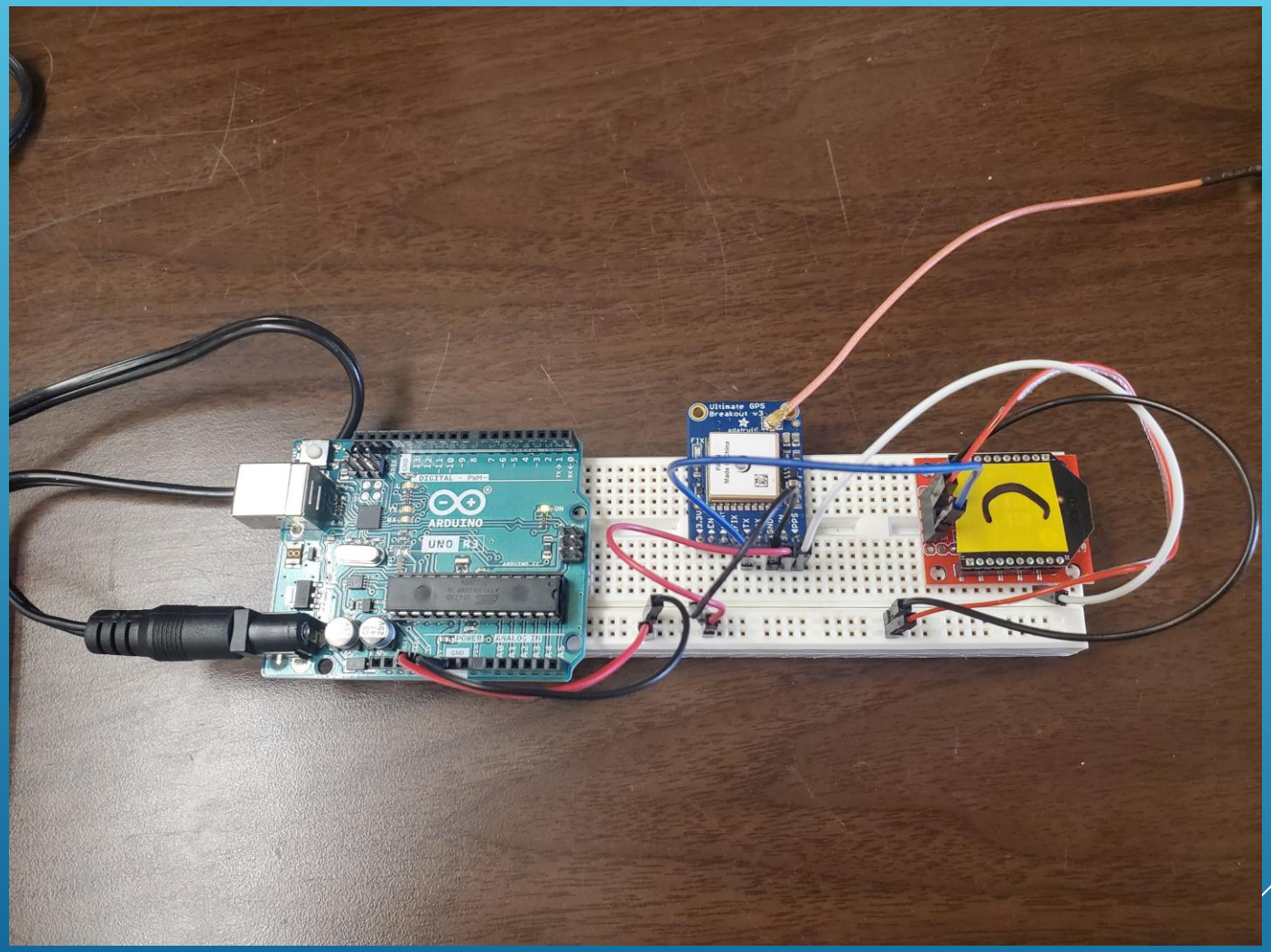
Receiver Circuit Connections

XBee GND - Arduino GND
XBee 5V - Arduino 5V
XBee DOut - Arduino Pin 10
Arduino GND - Board GND

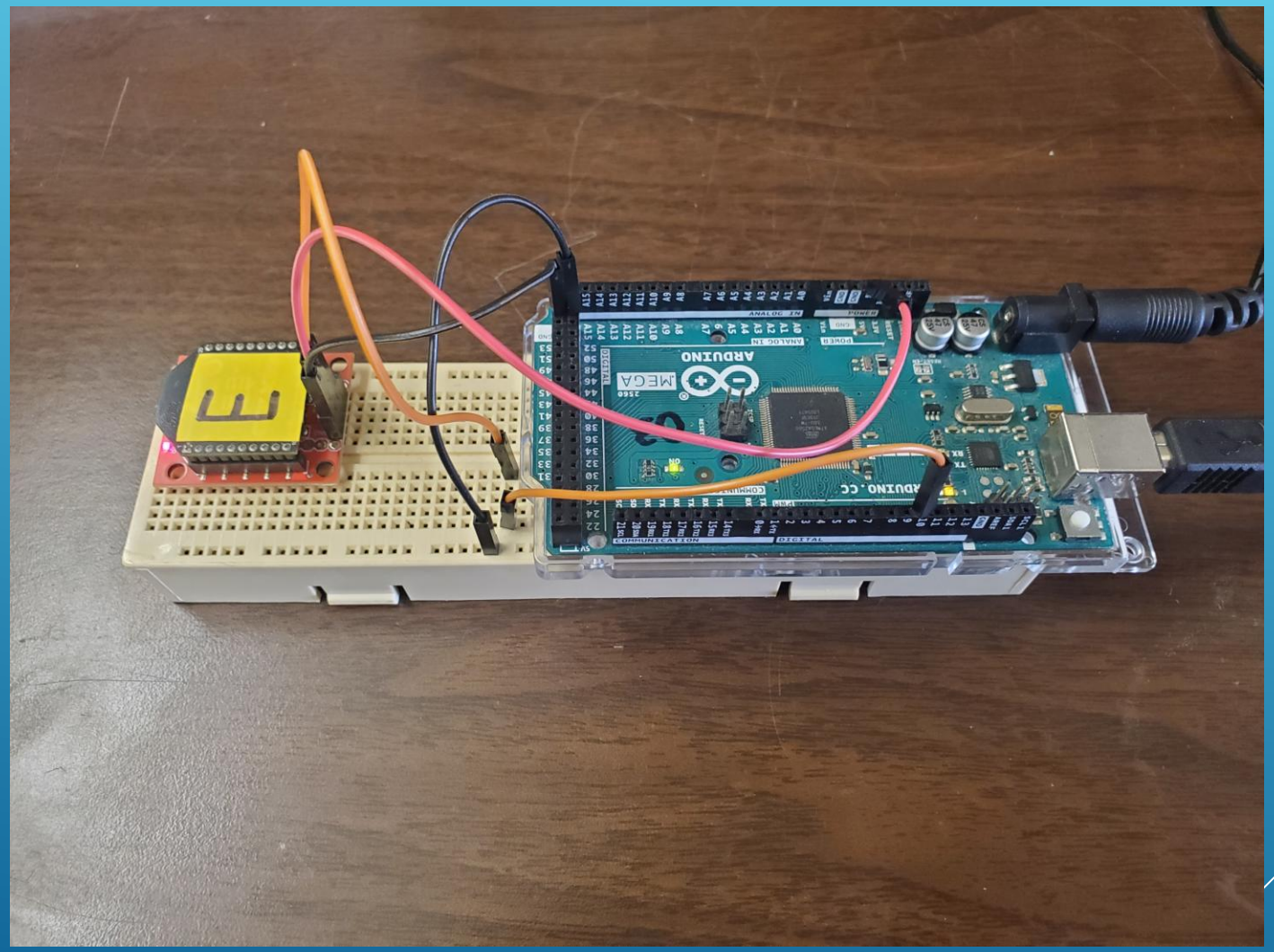


fritzing

XBEE TRANSMITTER PROTOTYPE CIRCUIT

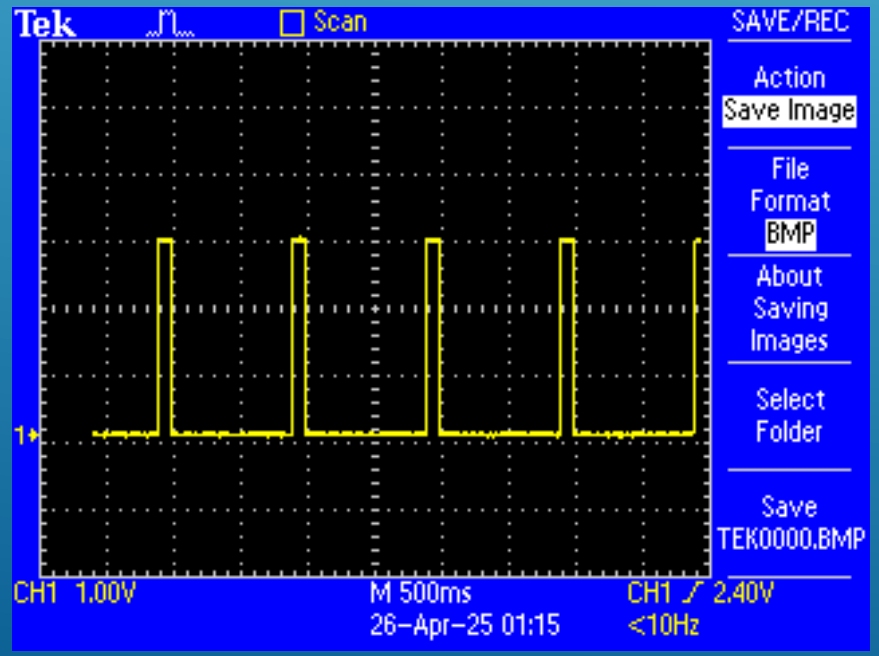


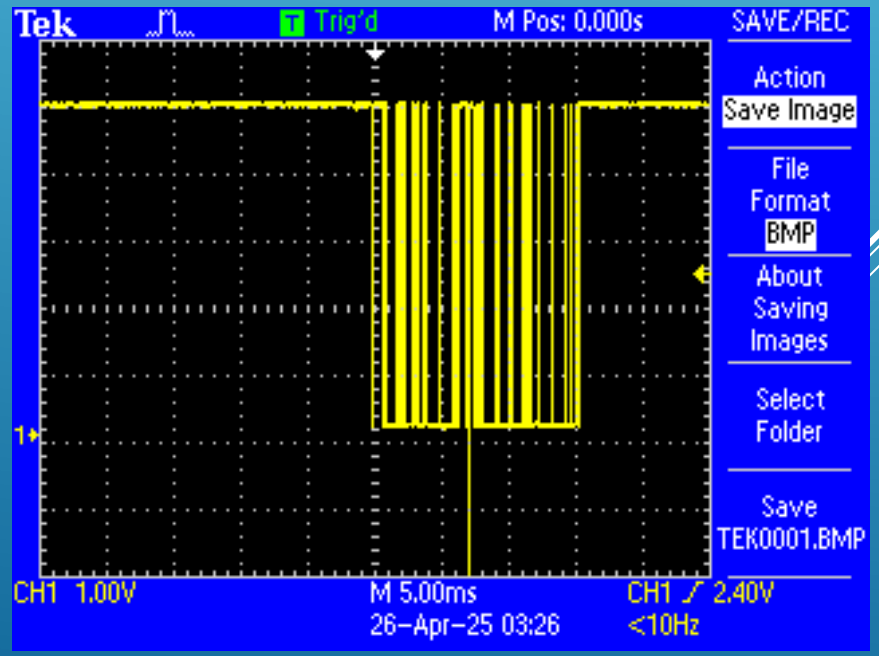
XBEE RECEIVER/DAQ PROTOTYPE CIRCUIT



PPS PULSE

- The heart of the timing calculations that we use for this project is the Adafruit Ultimate GPS's PPS (pulse per second) pin. This pin provides a simple 5V digital signal at a 10% duty cycle, meaning that the signal goes high for 100ms and low for 900ms in a repeating pattern.
- These high signals, when they are compared to the time that detector cosmic ray signals are received from the DAQ circuit can be used to calculate the exact time that the signal was received.





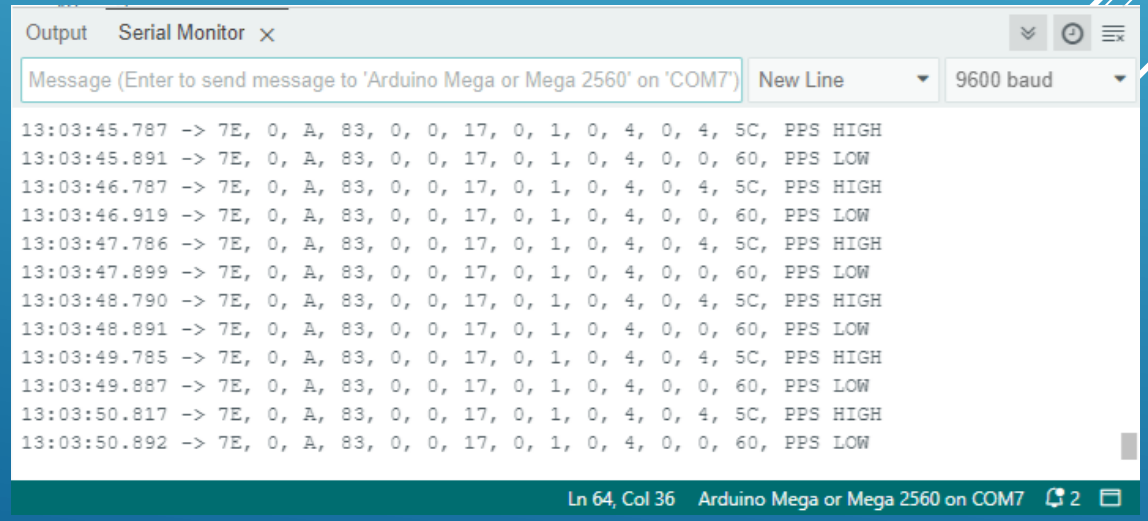
FRAME DATA

- Once the XBees have been properly configured, they will send and receive signals using a specific data frame structure as determined by the XBee's API.
- The transmitter XBee will take in data from a wired connection, and then format that data for transmission by adding unique frames to the signal that allow the receiver module to recognize the data.
- The individual frames take the form of ascii characters, but for readability our code will output the data as hexadecimal bytes instead.

INITIAL PPS CODE

```
void loop() {
  getData();
  if (newPPSData == true && receivedChars[0] == startMarker){
    printPPSData();
    newPPSData = false;
  }
  else{
    newPPSData = false;
  }
}

void getData() {
  if(XBee.available() > 0 && newPPSData == false){
    rc = XBee.read();
      if (rc == startMarker){
        recvInProgress = true;
      }
      if (recvInProgress == true){
        receivedChars[ndx] = rc;
        ndx++;
    }
  }
  if (ndx == ppsChars && receivedChars[3] == 131){
    receivedChars[ndx] = '\0';
    recvInProgress = false;
    newPPSData = true;
    ndx = 0;
  }
}
void printPPSData(){
  for (int i = 0; i < ppsChars; i++){
    Serial.print(receivedChars[i], HEX);
    Serial.print(", ");
    }
}
```

- In order to parse the data, the Arduino will read the incoming characters from the receiver XBee one by one. When it receives the character that corresponds to the start byte (7E), it will begin to write all of the incoming characters into an array.
- Once that array has reach the standard size of a PPS frame, 14 bytes, the program will then output each individual frame of the signal to the serial monitor, as shown below.

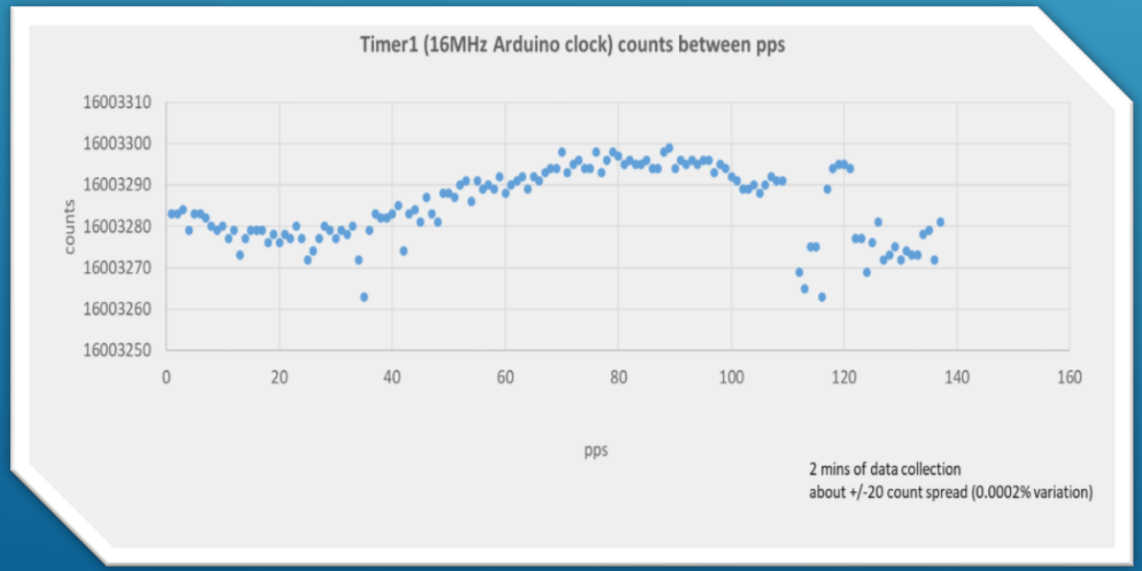


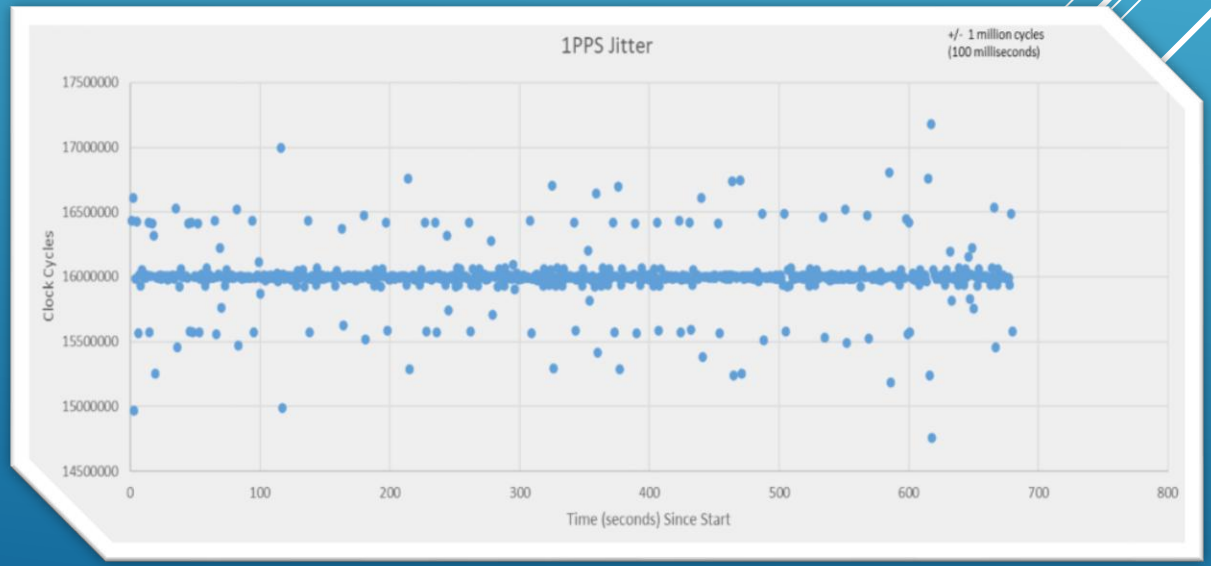
TIMESTAMPING WITH THE ARDUINO

- The main reason that the Cosmic Ray Detector box needs the PPS signal is that the GPS cannot provide a timestamp as precise as a microsecond, only a second. As such, these finer measurements need to be calculated with the Arduino's internal timers and registers.
- All computers contain an internal crystal oscillator clock that is used for internal timing purposes. This clock will oscillate with a fixed frequency, and this frequency will determine how many operations the computer will able to perform in a single second. The Arduino Mega 2560 boasts a clock speed of 16MHz, meaning that it's clock will oscillate once every 62.5 nanoseconds, or 16 times a microsecond.
- As such, all iterations of this project have used specific coding functions to keep track of the number of clock cycles that pass between successive PPS pulses, i.e. that pass once a second
- Once this wireless iteration of the project is complete, the program will be able to gather multiple timestamps to determine a.) the time within a second that a PPS pulse is received and b.) the time within a second that signal from a Muon coincidence is received. The program will then calculate the number of clock cycles that pass between these two signals in order to get the exact microseconds that the data is received. This value will then be appended to the GPS timestamp in order to get the exact time the cosmic ray signal was detected.

TESTING FOR PPS JITTER

- Due to the nature of wireless transmission, the signals we are using for timing will be considerably less accurate than those gathered from a wireless connection.
- Tests from previous students have determined that the wired connection's signals would have an average jitter of ±1.25microseconds , meaning that these signals would, on average, be received by the Arduino 1.25 microsecond faster or slower than expected.
- Previous tests with the XBee wireless connection, on the other hand, would display a maximum jitter of ±62.5 milliseconds. In attempting to write new code for the project, I wanted to experiment with new methods of parsing the data in order to see if these numbers were truly accurate, and if they could potentially be improved.

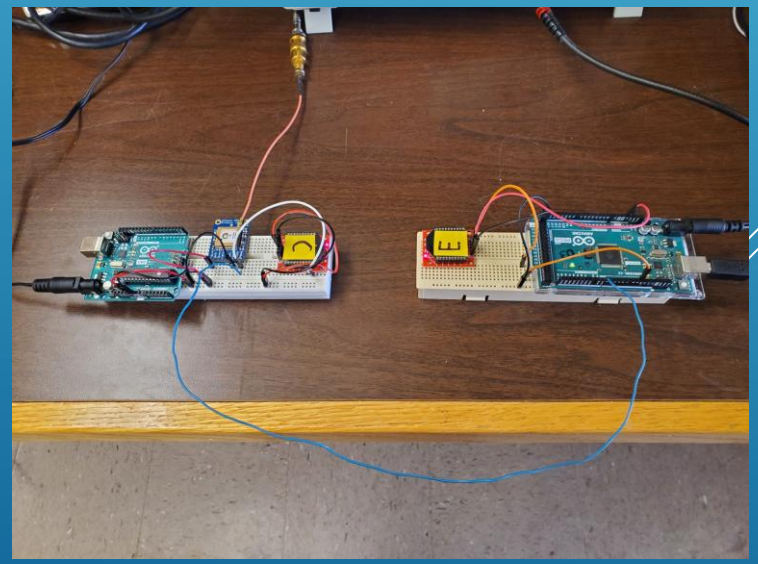




PPS JITTER CODE

```
 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() {
 lastTimerPPS = TCNT1;
 overflowsSincePPS = overflows;
}

void loop(){
 getData();
 if (newPPSData == true && receivedChars[0] == startMarker){
    printData();
    newPPSData = false;
  }
  else{
    newPPSData = false;
  }
}

void getData() {
  if(XBee.available() > 0 && newPPSData == false){
    rc = XBee.read();
      if (rc == startMarker){
        lastTimerXBEE = TCNT1;
        overflowsSinceXBEE = overflows;
        TCNT1 = 0; // Resets Timer1 Count
        overflows = 0;
        recentXBEE = true;
        recvInProgress = true;
      }
      if (recvInProgress == true){
        receivedChars[ndx] = rc;
        ndx++;
    }
  }
  if (ndx == ppsChars && receivedChars[3] == 131){
    receivedChars[ndx] = '\0';
    recvInProgress = false;
    newPPSData = true;
    ndx = 0;
  }
}
```

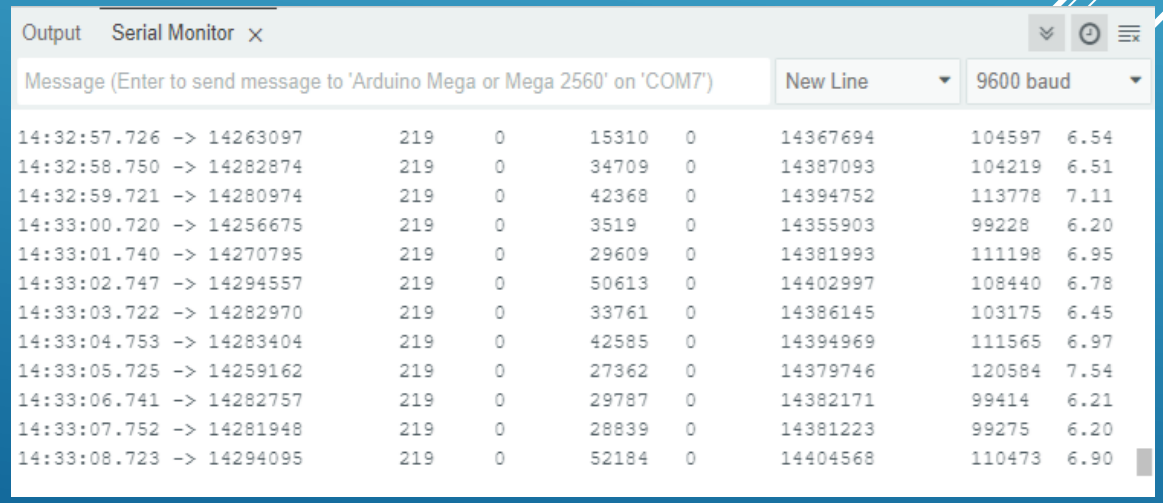
- Because the Arduino's internal registers that are used for the timers have a maximum size of 64Kb, or 65535 bits, a separate variable needs to increment every time the timer overflows. The exact number of clock cycles is then calculated by multiplying the number of overflows by 65535 and then adding the remaining contents of the timer register.
- In order to determine the jitter of the wireless signal, code was written that would gather two different timer values, one when the PPS pin on the GPS module goes high, and another when the XBee on the receiver end gets the start byte (7E) of the PPS transmission. The signal from the PPS pin was gathered by running a wire directly from the PPS pin to the Arduino Mega, as shown below, and interrupting the program to gather the timer values when this PPS pin goes high.



PPS JITTER CODE

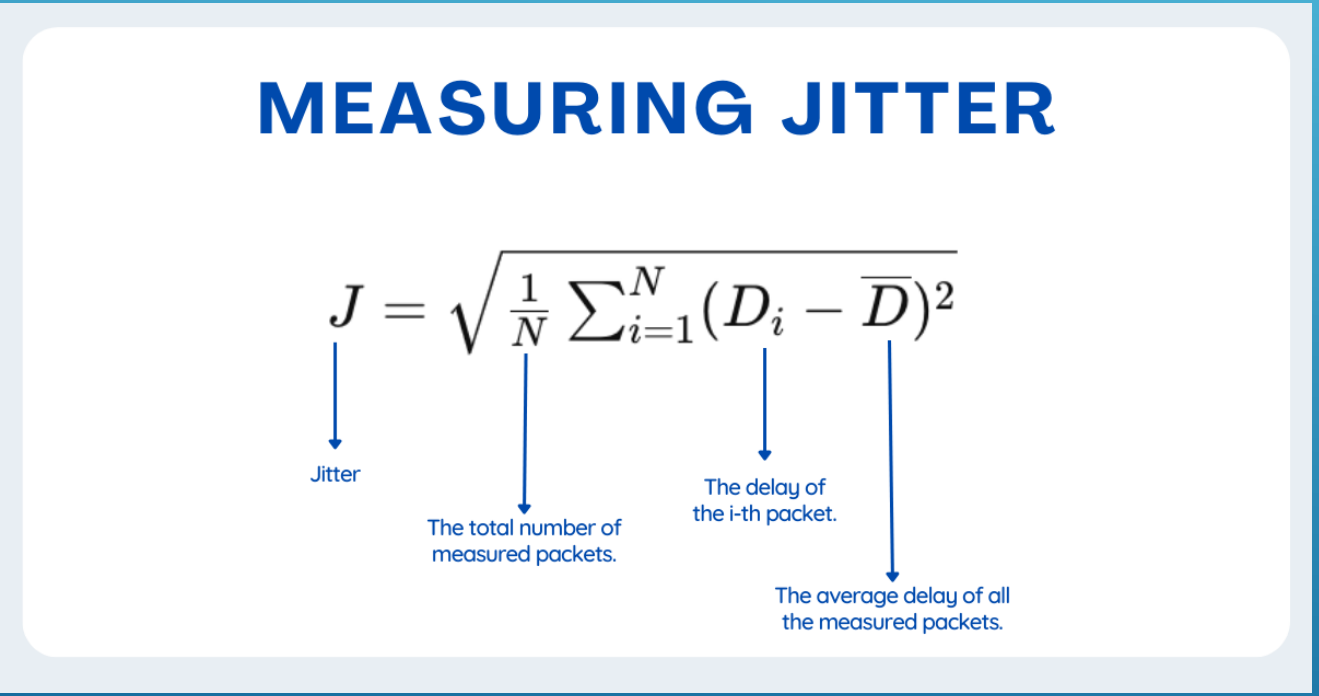
```
void printData(){
  if (recentXBEE) {
    noInterrupts();
    uint32_t overflowsTempPPS = overflowsSincePPS;
    uint32_t lastTimerTempPPS = lastTimerPPS;
    overflowsTempXBEE[ndx1] = overflowsSinceXBEE;
    lastTimerTempXBEE[ndx1] = lastTimerXBEE;
    uint32_t ppsCycles = overflowsTempPPS << 16 | lastTimerTempPPS;
    uint32_t xbeeCycles = (overflowsTempXBEE[0] + overflowsTempXBEE[1]) << 16 |
(lastTimerTempXBEE[0] + lastTimerTempXBEE[1]);
    uint32_t transmissionGap = xbeeCycles - ppsCycles;
    double transmissionGapMillis = transmissionGap * (62.5 * pow(10, -6));
    interrupts();

    if(receivedChars[8] == 1 && receivedChars [12] == 0x4){
      Serial.print(ppsCycles ); // Equivalent to overflowsTemp * 2^16 + lastTimerTemp
      Serial.print("\t");
      Serial.print(overflowsTempXBEE[0]);
      Serial.print("\t");
      Serial.print(overflowsTempXBEE[1]);
      Serial.print("\t");
      Serial.print(lastTimerTempXBEE[0]);
      Serial.print("\t");
      Serial.print(lastTimerTempXBEE[1]);
      Serial.print("\t");
      Serial.print(xbeeCycles); // Equivalent to overflowsTemp * 2^16 + lastTimerTemp
      Serial.print("\t");
      Serial.print(transmissionGap);
      Serial.print("\t");
      Serial.println(transmissionGapMillis);
    }

    ndx1++;
    if (ndx1 == 2){
      ndx1 = 0;
      for(int i = 0; i < 2; i++){
        overflowsTempXBEE[i] = 0;
        lastTimerTempXBEE[i] = 0;
      }
    }
  recentXBEE = false;
  }
}
```

- Because the XBee can only gather the PPS data by change detection (high to low or low to high), there is currently no way to prevent the XBee from transmitting an extra signal when the PPS pin goes low. This means that for every one second period of the PPS pin's cycle, two wireless signals will be transmitted to the XBee; and two timer reading will need to be gathered and then added together. The timer counters will then be reset once the high signal from the XBee is received, and the data will then be processed and displayed.
- Because, as noted, wireless transmissions travel slower, the high transmission will be received slightly after the PPS pin on goes high. As a result, if you subtract the sum of the two XBee timer readings from the PPS timer reading will produce the exact number of clock cycles between the PPS pin on the GPS going high and the XBee receiving the correct PPS wireless signal. The result of these calculations have been displayed below, with the two numbers on the right representing the number of clock cycles and it's corresponding value in milliseconds respectively.



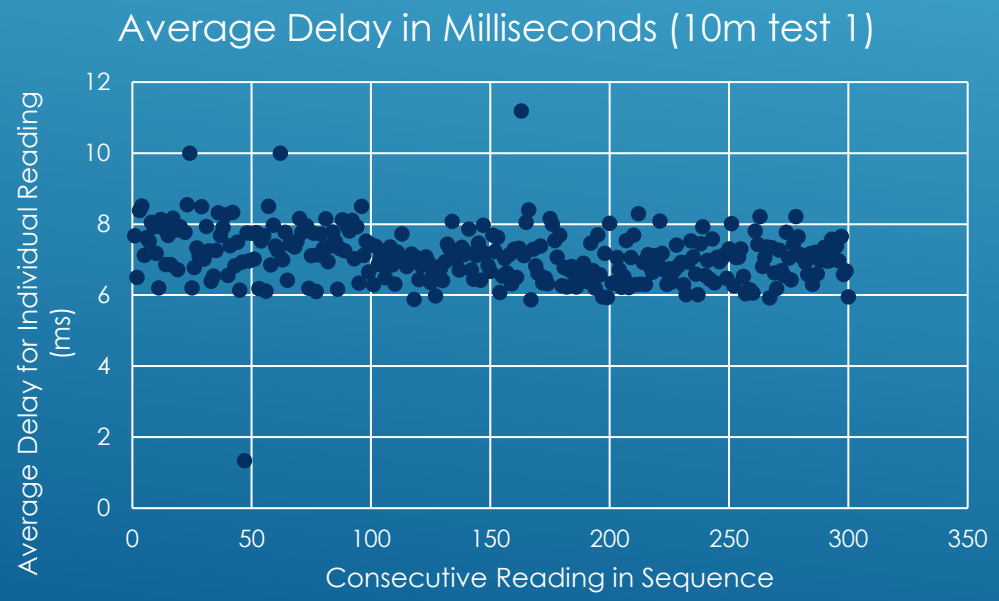
CALCULATING JITTER

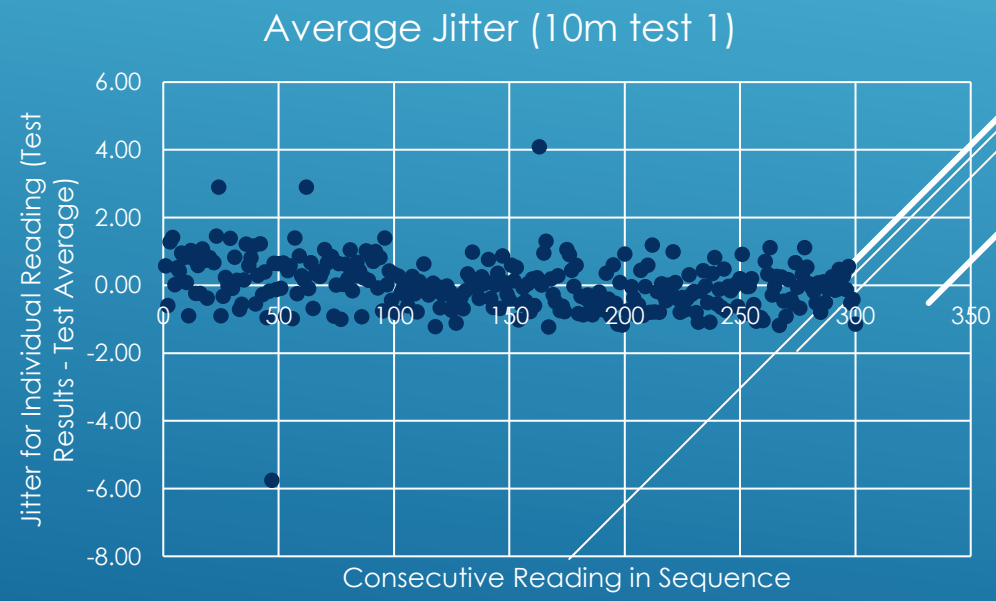
- In order to find a suitable average of the clock cycle readings, along with an average jitter, clock cycle data was collected across five consecutive five minute tests where the two XBees were separated by a distance of 10m (with a suitable length of wire running from the GPS module on one end to the Arduino on the other). With each of these tests, approximately 300 clock cycle reading were taken, each one second long, and the data was processed using the formula below to get an average jitter for the entire test.



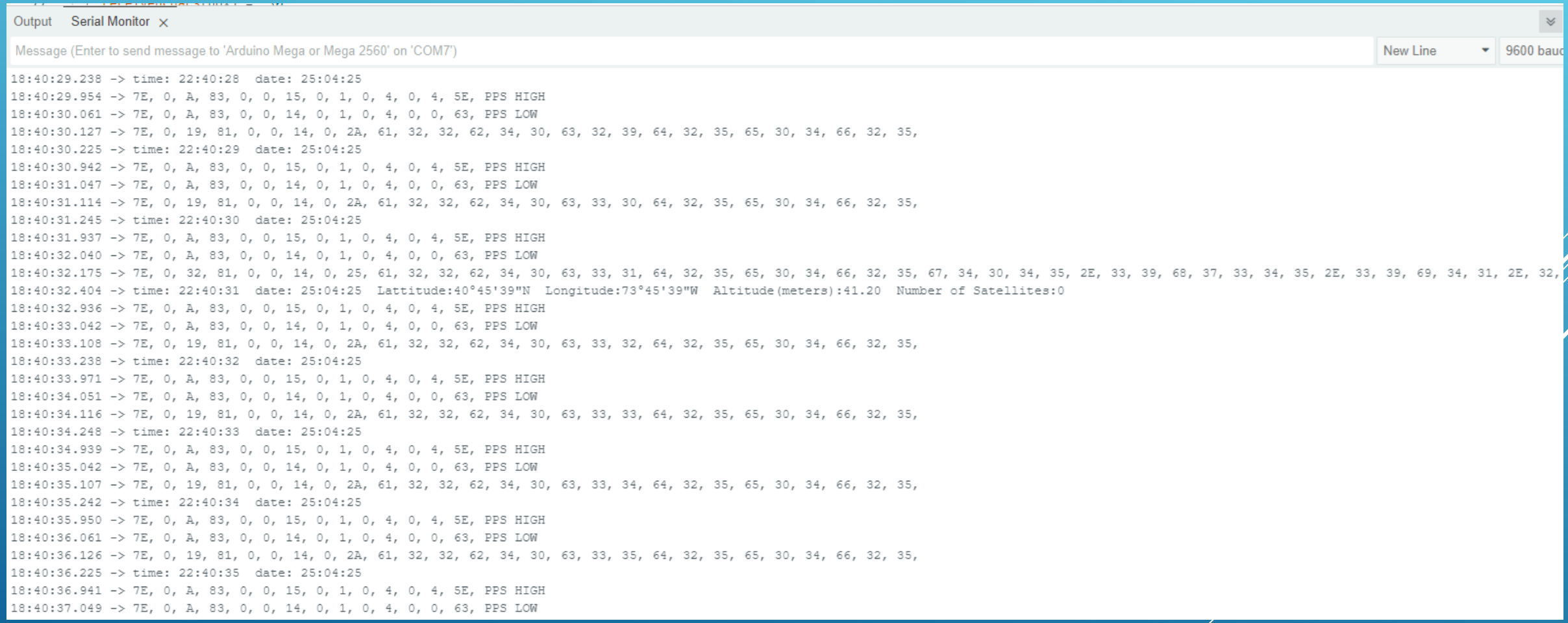
RESULTS

- The results of the five tests show that on average, a delay of 8.5ms can be expected between the physical (wired) PPS pulse and the reception of the pulse by the XBee.
- Additionally, an average jitter of only 930 microseconds was observed between each of the individual XBee wireless high signals.
- Tests at ten meters lead to a handful of erroneous readings with each test due to bugs with the code, and these needed to adjusted in the final results. These errors were much more pronounced in the 100m tests, so further work needs to be done to allow the circuit to function at longer range.





FUTURE GOALS: GETTING THE NMEA TIMESTAMP



Acknowledgements:
David Buitrago, Physics, York College
Nikolai Baca, Physics, Queensborough Community College
Raul Armendariz Ph.D., Physics, Queensborough Community College