**ELEC 291 Section 20C**

Lab 6

L2C

Team 2A

*Student name Student number Contribution percentage*

Andy Ruan 36863141 33.3%

Kevin Wong 32105132 33.3%

Clarence Su 36387132 33.3%

Contribution Summary:

Andy Ruan wrote the Arduino code for the ISR. He also wrote code to collect the voltages coming from the function generator and sending the data to Processing.

Kevin Wong created the Fritzing for the circuit and wrote documentation for the report

Clarence Su implemented the Processing code to read and graph the data from the Arduino. The GUI resembled an oscilloscope used in the laboratory.

**Introduction and motivations**

Lab 6 provides students to learn about reading signals and displaying the data. The motivation for this lab is for students to construct their very own oscilloscope. While building the oscilloscope, students will learn about AVR features of the Arduino, Analog to Digital on the Arduino and Graphing on Processing.

**Lab Description**

There were three components to this lab:

1. Setting up the hardware
2. Writing the Arduino Code
3. Building the Processing GUI

Combining all these components results in a circuit that reads the signal from the Function Generator by pressing on a push button. The signal will be converted from an analog to a digital and will be passed to Processing to graph the data like an oscilloscope.

**Setting up the hardware:**

The hardware setup for lab 6 was pretty simple. The components were Arduino, protoshield, mini breadboard, a resistor, a function generator and a push button.

First, the ground of the function generator needed to be connected to the ground of the Arduino. Following that, the output power of the function generator was connected to the analog pin of the Arduino in order to read the analog value and to convert the analog value to a digital value to collect voltage magnitude. The power of the Arduino was then connected to the breadboard in order to supply power to the other circuit elements such as the push button. The push button goes in series with a 10000 ohms resistor in order for the user to have a user-friendly mechanism to collect data from the function generator.

The function generator as its name suggests, allows the user to create various electrical waveforms over a wide range of frequencies. For this lab, we will be testing the circuit with a triangle wave, a square wave and a sine wave. The expected result is seeing a similar wave shape in Processing.

**Writing the Arduino Code:**

The team first implemented an Interrupt Service Routine (ISR). The ISR triggers on the rising edge of the momentary switch and sets a flag telling the main loop of the program to run. Using the Arduino’s ADC, the main loop periodically performs analog reading of the pin that is attached to the signal generator. Millis() is used to keep track of how long since the button’s rising edge has passed and reset the flag to disable the main loop after the minimum 15 seconds. This data is sent after a calculation to convert the value returned from analogRead() into a voltage between 0 and 5 volts, across Serial communications for use by Processing. Additionally, the value of micros() during each reading was also sent in order to determine the period of the wave form.

We found that the time between each reading was relatively slow during our initial prototyping, originally being around 1-2ms per reading. While we understood the limitations of the Arduino’s 16MHz processor, we did attempt to apply some techniques to increase our sampling rate. The first change was to reduce the prescale factor of the ADC clock from its default 128 to 16. Since the ADC clock runs at 16MHz divided by this prescale factor, this gives us more clock cycles in the same amount of time to perform the conversion, at the cost of some resolution. We managed to reduce our time per reading down to around 600us doing this alone.

Our team found that Serial communications after every reading caused this still somewhat significant delay. A small increase to performance we made was to increase the baud rate to 250000 from our default 9600. We also attempted to implement a buffer which would fill up before sending this data over and then filling another buffer. We found that this reduced the delay to just about 40us per reading which was a very significant difference. However, there was an even larger delay between the last reading of one buffer and the beginning of the next, upwards of 2-4ms depending on how large the buffer was. Additionally, this meant that the last value of a buffer would not always be coherent with the first value of the next. In an attempt to fix this, we decided to only send of the part of subsequent buffers that appended nicely to the previous, taking into account of the slope and value at the end of the previous buffer. The problem with this method was that at lower frequencies, entire buffers will have data points equal to 0.0 as all negative values of the waveform are returned as 0 by the ADC and the higher sampling rate. The amount of 0.0 datapoints sent over would be skewed by this logic in the code. At higher frequencies, this was less of an issue as we could consistently get non-zero values into the buffer. Ultimately, we decided to not implement this in our final sketch as in addition to these problems, we were bordering the point of tampering with the data. However, the partial completed code will be provided in Appendix IV.

**Building the Processing GUI:**

Our team tried to imitate an oscilloscope as close as possible by implementing a GUI similar to the oscilloscope used in the laboratory. The GUI was first constructed by creating a graph plot to determine the magnitude of the voltages received through the serial port.

The Processing GUI then uses serial to read the voltage and the time the data was collected in order to construct a graph. We implemented a current voltage value and a previous voltage value to record values in order to draw a line between them. Looping through all the values collected from the serial port, our team was able to implement a graph similar to the oscilloscope. When we generate a square wave on the function generator, we could plot a square wave in Processing. Similarly, we could plot a sine wave and a triangle wave in Processing when we create a sine wave or triangle wave on the function generator.

To draw the graph on the GUI windows, we used two global variables. One is the current data point and the other is previous data point. To plot the graph, we use line() function to connect two points. The algorithm used to translate voltage to the coordinate system used in the GUI is provided in the code in Appendix III. As for the x-axis, considering that a small change in frequency will affect the x-axis a lot and the time interval detected by the Arduino is not perfectly precise, we decided to implement a fixed scale x-axis for all frequency. Later, we implemented a feature that allow user to zoom in and zoom out in the x-axis by pressing the up and down button on the keyboard.

A major challenge was the implementation of the zoom in and out feature for the graph. Our team needed to construct the graphing grid with variables in order to dynamically change them with a click of a button. Two buttons were implemented for y-axis scaling. First, we have a hover function constantly checking if the user’s mouse is in the area of the button. If so, our program will resize the button that the mouse select to allow the user easily see which button they selected. The up button will check if the current size is the maximum. if not. it will scale the y-axis as the following 0.1->0.5, 0.5->1, 1->1. The down button will check if the current scale is at minimum. If not, it will scale the y-axis as the following 1->0.5, 0.5->0.1, 0.1->0.1. The scaling factor is one of the parameter that is used for the position algorithm for plotting the data point on the windows and scaling factor is declared as a global variable. Therefore, by changing the scale, the graph plot will change corresponding. Similarly, our team implemented two key events to scale the x-axis, it will check which key that the user presses on their keyboard. If the key pressed is the up arrow on the keyboard, it will increase the x-axis scaling factor so that it will stretch the graph horizontally. If the key pressed is the down arrow on the keyboard, it will check if the current scale is at the minimum. If not, it will scale down the x-axis scaling factor so that the graph will shrink horizontally. Due to time constraints, our team did not change the key events for scaling the x-axis to be in the Processing window like the up and down buttons for scaling the y-axis.

Another feature our team implemented was a cursor functionality similar to the oscilloscope. This cursor can calculate the voltage difference between two points.

A button was implemented for the user to find the voltage difference between two point and the period of the graph. Once the user pressed the button, the cursor function is activated. Then, the user can start sampling data. Once the user finds the section they want to measure, they could move their mouse to place the cursor line by clicking on desired area. The distance between two cursor line will be translated back to voltage difference by reverse the calculation for translating voltage to the point in the coordinate system. The computation code is provided in the code in Appendix III. This calculation takes into the consideration of the changing scale. Therefore, it takes the scaling factor as one of the parameter and will change the value automatically if the scale is changed by the user. As for the period, we store 400 data points (voltage and time) into two global buffer. The compute period algorithm will go into the buffer and find the first zero, and store that as the starting\_time. Then it will proceed finding the second zero, and store that as the ending\_time. The difference between starting\_time and ending\_time will be the difference for half period. By scaling the result by 2, we got the period for a complete cycle. We are only storing 400 data point because we are afraid that increasing the buffer size might affect the smoothness of real time rendering. That is, the Processing GUI becomes extremely slow and not responsive. Because of this, the computing period algorithm does not work every time, when the frequency is low(too few data point in that period).

**Conclusions**

This lab allowed students to have an opportunity to build their own oscilloscope using an Arduino, Processing and a function generator. This lab also allowed students to review tools used from other classes such as the function generator from ELEC 201. Additionally, students were able to improve their GUI development skills by writing code in Processing.

The team learned some lessons during this lab. One lesson was that no matter how good a piece of software is, there is always limitations from the hardware. In this lab, the Arduino was the limitation. It could only take read values from 0-5 volts otherwise it would be damaged. Due to this limitation, our Processing graph for oscilloscope could not imitate a real oscilloscope perfectly. Moreover, when passing data from Arduino to Processing using serial, there is delay that causes our sampling rate to lose approximately 600 microseconds of data points; therefore, the Processing graph is not perfect. If there was a different way to pass data points from Arduino to Processing, the graph generated in Processing would be much better.

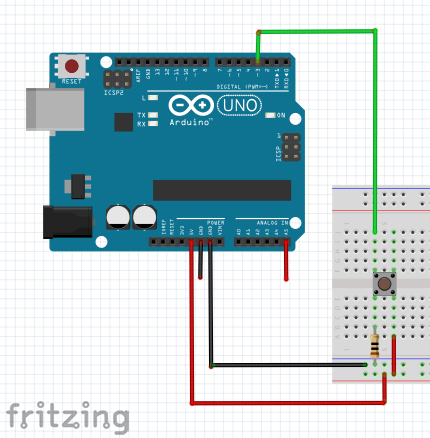
Another lesson learned was that constant testing of our software is extremely important. Instead of testing the GUI with the function generator after all our functions were implemented. We tested each function separately with the function generator after it was implemented. That way, we receive instant feedback if the code is not working as expected. If not, then our team could quickly debug and change the code to make the GUI better.

**References**

[1] https://learn.sparkfun.com/tutorials/analog-to-digital-conversion

**Appendix I**

Below is an image a Fritzing breadboard schematic for Lab 6:



**Figure 1. Fritzing Diagram**

NOTE:

The red wire coming out of the A5 analog pin is to display the input for the function generator. The black wire coming out of the ground pin is to display the ground for the function generator.

**Appendix II**

Below is the Arduino code for Lab 6:

|  |
| --- |
| // defines setting and clearing register bits  #ifndef cbi  #define cbi(sfr, bit) (\_SFR\_BYTE(sfr) &= ~\_BV(bit))  #endif  #ifndef sbi  #define sbi(sfr, bit) (\_SFR\_BYTE(sfr) |= \_BV(bit))  #endif  #define BUTTON\_PIN 3  #define VOLTAGE\_IN A5  const long samplingDuration = 15000;  int ISRflag = 0; // used to trigger main loop  void setup() {  Serial.begin(250000);  // set prescale to 16  sbi(ADCSRA, ADPS2);  cbi(ADCSRA, ADPS1);  cbi(ADCSRA, ADPS0);  attachInterrupt(digitalPinToInterrupt(BUTTON\_PIN), isr, RISING); // trigger ISR on rising edge of button  }  void loop() {  if ( ISRflag == 1 ) { // main loop runs after ISR detected  unsigned long prevMilli = millis(); // record start time in millisecond  unsigned long oriMicros = micros(); // record start time in microseconds  while ( millis() - prevMilli <= samplingDuration ) { // send data for samplingDuration milliseconds  Serial.print(analogRead(VOLTAGE\_IN) \* (5.0 / 1023.0)); // convertion from analogRead to 0-5V  Serial.print(",");  Serial.print(micros() - oriMicros); // time since we started sampling for this data point  Serial.print(",0\n");  }  ISRflag = 0; // reset ISRflag so we do not continue to loop  }  }  // ISR that sets a flag to trigger the main loop  void isr() {  ISRflag = 1;  } |

**Appendix III**

Below is the Processing Code for Lab 6:

|  |
| --- |
| import processing.serial.\*;  import java.util.\*;  //variable for color  float background\_color = 0;  float white =255;  float green=150;  float red = 255;  //font  PFont font;  //image  PImage onButton;  PImage upButton;  PImage downButton;  PImage leftButton;  PImage rightButton;  //button variables  boolean onButtonOver = false;  boolean upButtonOver = false;  boolean downButtonOver = false;  boolean cursorOn = false;  int buttonWidth = 50, buttonHeight = 50;  int onButtonX= 650, upButtonX=730, downButtonX=810;  int buttonY=550;  int cursorNum=0;  int cursorsY[]=new int[2];  //buffer for voltage and time data  float voltage\_buffer[] = new float[400];  long time\_buffer[] = new long[400];  Serial myPort;  //scale or inrecement for x and y axis  float scale\_factor = 1;  float x\_increment=0.2;  //variable used to compute period  float pre\_voltage = 0;  int pre\_time = 0;  float current\_voltage = 0;  int current\_time = 0;  int current\_x=0;  int current\_y=0;  int pre\_x=0;  int pre\_y=0;  long period=0;  //debug  int index=0;  float data\_inteval=35;  void setup() {  size(900, 670);  background(background\_color);  myPort = new Serial(this, Serial.list()[1], 250000);  myPort.bufferUntil('\n');  font = loadFont("BanglaMN-48.vlw");  onButton = loadImage("onButton.png");  onButton.resize(50, 50);  upButton = loadImage("upArrow.png");  upButton.resize(50, 50);  downButton = loadImage("downArrow.png");  downButton.resize(50, 50);  }  void draw() {  draw\_transparent\_rect(); //update the area whose information is changing  draw\_display\_windows(); //draw the windonws  draw\_addtional\_feature(); //draw other things  updateButtonStatus(); //check if button is pressed  drawCursor(); //draw the cursor  draw\_data(); //plot the data  }  //draw the transparent rectangle  void draw\_transparent\_rect() {  int middle\_y =(600-50)/2 +55;  noStroke();  fill(0, 50);  rect(0, 0, 50, 670);  rect(600, 0, 300, 670);  }  void draw\_display\_windows() {  int top = 50;  int bot = 600;  int mid = (600-50)/2 +50;  //frame  stroke(255);  strokeWeight(1);  line(top, top, bot, top);  line(top, bot, bot, bot);  line(top, top, top, bot);  line(bot, top, bot, bot);  strokeWeight(3);  line(top+2, mid, bot-2, mid);  //grid line  stroke(255);  strokeWeight(0);  //horizontal grid line  for (int i=1; i<11; i++) {  line(top, top+i\*55, bot, top+i\*55);  }  //vertical grid line  for (int i=0; i<11; i++) {  line(top+i\*55, top, top+i\*55, bot);  }  //label for axis  //y-axis  fill(255, 204, 0);  textFont(font, 16);  text("Voltage(v)", 0, top-20);  textFont(font, 12);  float init\_label\_y=5;  for (int i=0; i<11; i++ ) {  float label\_value = init\_label\_y\*scale\_factor-i\*scale\_factor;  String label = String.format("%.1f", label\_value);  text(label, top-24, top+i\*56);  }  //x-axis label  fill(255, 204, 0);  textFont(font, 16);  text("Time", bot+5, mid);  }  //additional function  void draw\_addtional\_feature() {  image(onButton, onButtonX, buttonY);  image(upButton, upButtonX, buttonY);  image(downButton, downButtonX, buttonY);  }  //check if the the mouse is on the up button  boolean overUpButton() {  if (mouseX >= upButtonX && mouseX <= upButtonX+buttonWidth &&  mouseY >= buttonY && mouseY <= buttonY+buttonHeight) {  return true;  } else {  return false;  }  }  //check if the the mouse is on the down button  boolean overDownButton() {  if (mouseX >= downButtonX && mouseX <= downButtonX+buttonWidth &&  mouseY >= buttonY && mouseY <= buttonY+buttonHeight) {  return true;  } else {  return false;  }  }  //check if the the mouse is on the cursor Button  boolean overOnButton() {  if (mouseX >= onButtonX && mouseX <= onButtonX+buttonWidth &&  mouseY >= buttonY && mouseY <= buttonY+buttonHeight) {  return true;  } else {  return false;  }  }  //check if one of the button is selected, if yes, resize the button to indicate the selection  //effect and set that button status to true  void updateButtonStatus() {  if (overOnButton()) {  onButtonOver = true;  downButtonOver = false;  upButtonOver = false;  onButton.resize(60, 60);  upButton.resize(50, 50);  downButton.resize(50, 50);  } else if (overUpButton()) {  onButtonOver = false;  downButtonOver = false;  upButtonOver = true;  onButton.resize(50, 50);  upButton.resize(60, 60);  downButton.resize(50, 50);  } else if (overDownButton()) {  onButtonOver = false;  downButtonOver = true;  upButtonOver = false;  onButton.resize(50, 50);  upButton.resize(50, 50);  downButton.resize(60, 60);  } else {  onButtonOver = upButtonOver = downButtonOver = false;  onButton.resize(50, 50);  upButton.resize(50, 50);  downButton.resize(50, 50);  }  }  //mouse click event and check wichi button is clicked  void mousePressed() {  if (onButtonOver) {  cursorOn();  fill(0, 250);  rect(50, 50, 550, 600);  } else if (upButtonOver) {  //scale up the y-axis  if (scale\_factor==0.5 )  {  scale\_factor=1;  } else if (scale\_factor==0.1 ) {  scale\_factor=0.5;  }  } else if (downButtonOver) {  //scale down the y-axis  if (scale\_factor==1 )  {  scale\_factor=0.5;  } else if (scale\_factor==0.5 ) {  scale\_factor=0.1;  }  } else if (cursorOn && cursorNum<2) {  //draw cursor line with the mouse coordinate  cursorsY[cursorNum]=mouseY;  cursorNum++;  }  }  //turn on cursor  void cursorOn() {  cursorOn = true;  cursorNum = 0;  }  //draw cursor  void drawCursor() {  if (cursorOn) {  int i=0;  for (i=0; i<cursorNum; i++) {  //draw the cursor line  strokeWeight(3);  stroke(#FFFF00);  line(52, cursorsY[i], 598, cursorsY[i]);  }  if (cursorNum==2) {  //compute the cursor value and the period of the data  //display the result on the window  float cursorValue = abs(cursorsY[1]-cursorsY[0])/55.0\*scale\_factor;  String stringValue = String.format("%.2f", cursorValue);  String toPrint = "Cursor Value: " + stringValue;  fill(white, white, white);  text(toPrint, 650, 100);  compute\_period();  String frequencyValue = String.format("%d", period);  String stringFreq = "Period: "+ frequencyValue +" ms";  text(stringFreq, 650, 150);  }  }  }  //read data  void serialEvent (Serial myPort) {  String inString = myPort.readStringUntil('\n');  String[] data;  if (inString != null) {  //parse the data  data = split(inString, ',');  current\_voltage = float(data[0]);  current\_time = int(data[1]);    //store the data into a buffer for later computation  if (index<400) {  voltage\_buffer[index]= current\_voltage;  time\_buffer[index]= current\_time;  }  index++;  //for debug  println("the index is " + index);  //index++;  //println(data[0]);  //println(data[1]);  }  }  //draw data on the window  void draw\_data() {  int middle\_y =(600-50)/2+50;  //debug  //println("current voltage is "+current\_voltage);    //draw the plot by connecting the previous data point and the current data point  current\_y = int((middle\_y - current\_voltage\*55/scale\_factor));  pre\_y = int((middle\_y - pre\_voltage\*55/scale\_factor));  strokeWeight(2);  stroke(#b0b031);  line(50+index\*x\_increment, pre\_y, 50+(index+25)\*x\_increment, current\_y);  if (50+(index+1)\*x\_increment>=595) {  index=0;  fill(0, 200);  noStroke();  rect(50, 0, 550, 670);  }  strokeWeight(0);  pre\_voltage = current\_voltage;  }  void compute\_period() {  boolean find\_init=false;  boolean find\_end=false;  long start\_time = 0;  long end\_time=0;  if (index>=400) {  for (int i=0; i<399; i++) {  //find the first zero in the period  if (voltage\_buffer[i]==0.0 &&  voltage\_buffer[i+1]!=0.0) {  find\_init = true;  start\_time = time\_buffer[i];  if (i<350) i+=10;  }  //find the second zero in the period after the first zero is found  if (find\_init && voltage\_buffer[i]==0.0) {  find\_end=true;  end\_time = time\_buffer[i];  }  if (find\_init && find\_end ) {  //after finding the first and second zero  //compute the period  period = (end\_time-start\_time)\*2/1000;  return;  }  }  }  }  //keyEvent  void keyPressed() {  if (key == CODED) {  if (keyCode == UP) {  //increase the x-scale if up key is pressed  x\_increment+=0.1;  } else if (keyCode == DOWN) {  if (x\_increment>0.2) {  //decrease the x-scale if down key is pressed  x\_increment-=0.1;  }  }  }  } |

Below is the Processing GUI for Lab 6:



Figure 2. Processing GUI

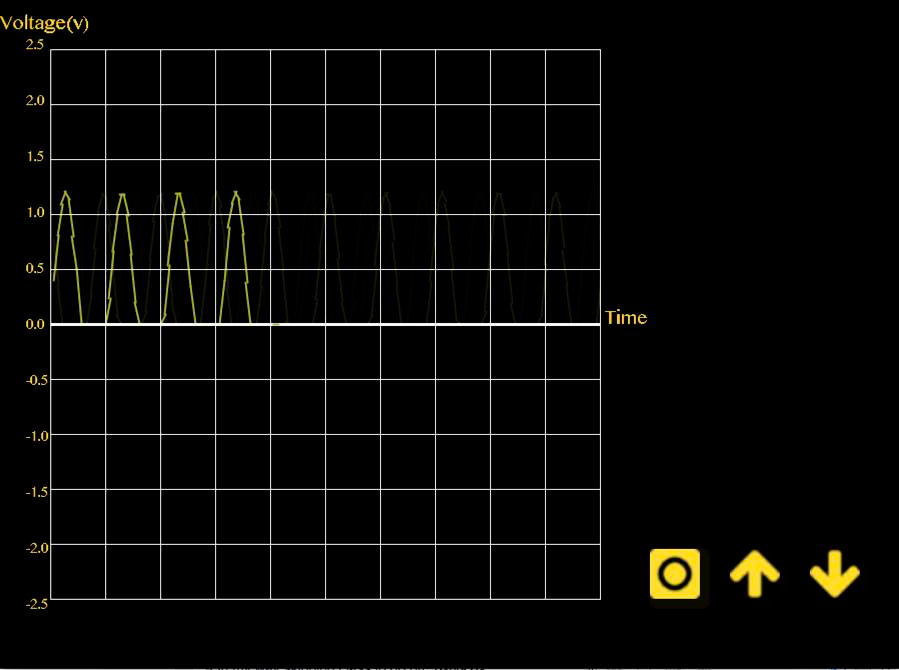


Figure 3. Processing GUI when reading data

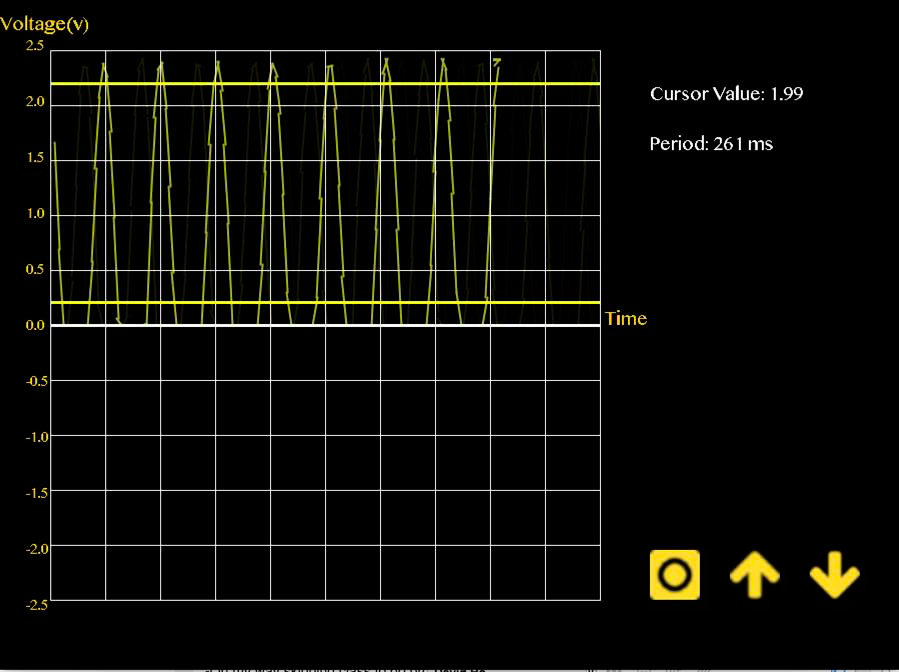


Figure 4. Processing GUI with cursor value to determine voltage difference

**Appendix IV**

Below is the partially completed Arduino buffer code our team tried to implement. However, due to time constraints and reasons described in the lab report, we decided not to complete it.

NOTE: This was not used in the demo. Appendix I contains the Arduino code for Lab 6.

|  |
| --- |
| // defines for setting and clearing register bits  #ifndef cbi  #define cbi(sfr, bit) (\_SFR\_BYTE(sfr) &= ~\_BV(bit))  #endif  #ifndef sbi  #define sbi(sfr, bit) (\_SFR\_BYTE(sfr) |= \_BV(bit))  #endif  #define BUTTON\_PIN 3  #define VOLTAGE\_IN A5  #define BUFFER\_SIZE 200  const long samplingDuration = 3000;  int nonZeroIndex = -1;  int ISRflag = 0;  int slopeFlag = -1;  int prevBufferEnd = 0;  void setup() {  Serial.begin(250000);  // set prescale to 16  sbi(ADCSRA, ADPS2);  cbi(ADCSRA, ADPS1);  cbi(ADCSRA, ADPS0);  attachInterrupt(digitalPinToInterrupt(BUTTON\_PIN), isr, CHANGE);  }  void loop() {  if ( ISRflag == 1 ) {  int data[BUFFER\_SIZE];  unsigned long prevMilli = millis();  int i = 0;  // fill the buffer  while ( millis() - prevMilli <= samplingDuration && i <= BUFFER\_SIZE) {  data[i] = (analogRead(VOLTAGE\_IN));  i++;  // send data in buffer after it has been filled  if ( i >= BUFFER\_SIZE || millis() - prevMilli >= samplingDuration ) {  int j = 0;  // calculate which part of the start location of the buffer to append to the last set of data points  switch ( slopeFlag ) {  case -1: // previous slope was negative  while ( (data[j + 1] > data[j] || data[j] > prevBufferEnd) && j < i )  j++;  break;  case 1: // previous slope was positive  while ( (data[j + 1] < data[j] || data[j] < prevBufferEnd) && j < i )  j++;  break;  case 0: // previous slope was 0  while ( data[j] == 0 )  j++;  break;  default:  while (data[j] == 0)  j++;  break;  };  // send data points, ignores 0s, record the index of the last non-zero value  while ( j < i ) {  if (data[j] != 0) {  Serial.print(data[j] \* (5.0 / 1023.0));  Serial.print(",0\n");  nonZeroIndex = j;  }  j++;  }  Serial.print("BREAK\n"); // debugging info to tell us when a bufferred set of data ends  i = 0; // reset i to 0 so we can fill a new buffer later  // calculates the slope of the end of the buffer that was just sent  if (nonZeroIndex == -1)  slopeFlag = 0;  else if (data[nonZeroIndex] < data[nonZeroIndex - 1]) {  slopeFlag = -1;  prevBufferEnd = data[nonZeroIndex];  }  else if (data[nonZeroIndex] > data[nonZeroIndex - 1]) {  slopeFlag = 1;  prevBufferEnd = data[nonZeroIndex];  }  else  slopeFlag = 0;  nonZeroIndex = -1; // resets the nonZeroIndex for next buffer sending  }    }  }  ISRflag = 0; // stop running main loop code after samplingDuration milliseconds  }  void isr() {  ISRflag = 1;  } |