Lab experience 5: Accelerometer-based CPR feedback device

Arduino Version

Objectives

- ➢ Gain experience using an electronic accelerometer
- ➢ Practice writing Arduino firmware
- ➢ Learn how to create a physical user interface with a push button and OLED display

Reading & new concepts

- Pull-up resistors
- Interrupt service routine (ISR)
- Software debouncing
- Moving average

Hardware

1. ADXL335 Accelerometer

As you might guess, an accelerometer measures acceleration, including the virtual acceleration produced by Earth's gravitational field. For this week's activities we are using an Analog Devices ADXL335 3-axis accelerometer installed on a printed circuit board with a voltage regulator and appropriate capacitors and connectors.[1] The ADXL335 has ±3g range; which should be plenty for the CPR project. This accelerometer uses a surface-micromachined capacitive sensing cell as described in its datasheet.[2]

In order to use the accelerometer board, we need to supply power at the Vin and GND pins, and measure from the X, Y and/or Z out pins. Zout is the most important for its use for CPR feedback; use of Xout and Yout is optional. The other pins may remain disconnected.

A header (row of pins) has been soldered to each board, and you will be given long jumper wires to allow you to connect the DAQ to the accelerometer board. The power input at Vin may be supplied by the 5 V supply on the DAQ, so the minimal setup consists only of the DAQ, accelerometers and connectors.

Figure 1. ADXL335 Accelerometer.

[1] https://www.adafruit.com/product/163

[2] https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL335.pdf

2. OLED Module

OLED displays are widely used in embedded systems because they are thin, light, and do not require backlight like LCD displays. The SSD1306 OLED driver is embedded inside the OLED module, and it contains a 128x64 bit SRAM buffer that stores the display pattern. We will use the Adafruit 1306 Library to write to this buffer via I2C. In addition, the Adafruit GFX Graphics Library contains high-level functions that simplify the drawing process. Learn how to draw more complicated graphics (not required for this lab) with Adafruit GFX Graphics Library:
https://learn.adafruit.com/adafruit-gfx-graphics-library/overview

Figure 2. An OLED module. This particular one has a display size of 0.96 inches (128x64 pixels). The top 16 rows of pixels are yellow, and the remaining 48 rows are blue. It may look big in this figure, but it is only slightly larger than a U.S. quarter.

3. Pull-Up Resistors

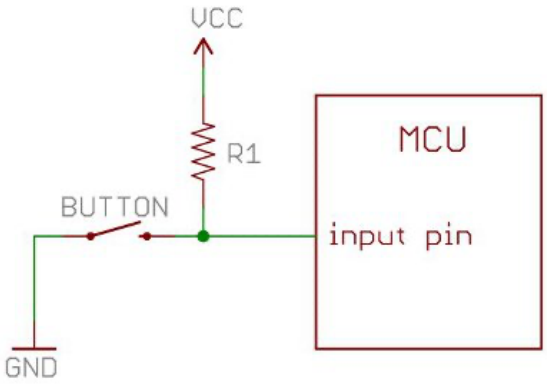


Figure 3. An input pin is "pulled up" by a resistor. This prevents the voltage from "floating" when the switch is open. Figure source: https://learn.sparkfun.com/tutorials/pull-up-resistors/all

Suppose that we are reading the voltage from a grounded switch, what should the input voltage be when the switch is open? If the input pin is not connected to anything else, then the voltage is at an indeterminate value. Sometimes people also refer to a disconnected pin like this as a "floating" pin. To mitigate this, we can connect a resistor of a large value, usually around 10 kΩ, to "pull" the pin to VCC whenever the switch is open (Figure 3). On Arduino, almost every pin comes with

an internal pull-up resistor. To enable the pull-up resistor on an digital pin, simply do: `pinMode(pinNumber, INPUT_PULLUP)` during setup.

Software

1. Interrupt Service Routine (ISR)

Suppose we want to check if the button has been pressed by the user, we can have an if-statement inside the `loop()` function that checks the pin's state every loop iteration. However, this means that we run the risk of missing a button press when the program is busy doing something else. Fortunately, in most microcontrollers there exist a few digital pins that, when detected changes in voltage level, can raise a flag that signals to the control unit to execute another set of instructions, thereby interrupting the current program (Figure 4). The set of instructions executed in response to an interrupt is called an Interrupt Service Routine (ISR).

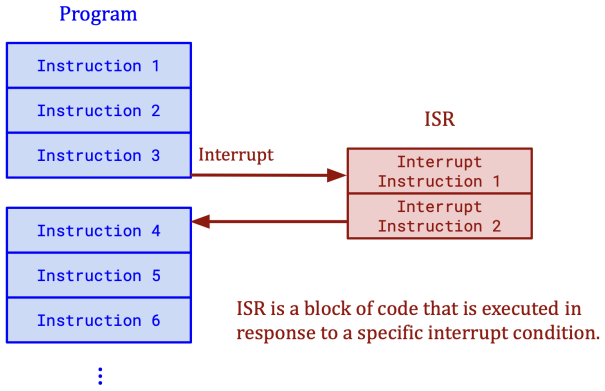


Figure 4. How an Interrupt Service Routine "interrupts" the main program.

On Arduino Uno, only pin 2 and 3 support interrupts. In this lab, we connect a push button to one of those pins. Then, we will write an ISR that will change the contents of the OLED display whenever the button is pressed.

2. Debouncing a button

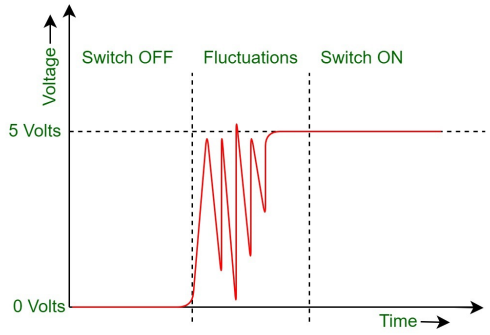


Figure 5. The voltage trace of a bouncy switch when it is switched on. Figure source: https://www.geeksforgeeks.org/switch-debounce-in-digital-circuits/

Mechanical push buttons, especially the cheap ones, bounce when they are pressed (Figure 5). This can be really frustrating because, for each button press, multiple edge transitions can occur, and this triggers multiple ISRs in quick sucession. An easy way to solve this is to use an RC low-pass filter–we can put the capacitor in parallel with the push button to smooth out the waveform. This is hardware debouncing. But in this lab, we will learn to debounce a push button with code, i.e., software debouncing. The idea is simple–after an ISR is called, we specify a "refractory" period, typically 200 ms, during which no other edge transitions should be registered. This post "Debounce Code - One Post to Rule Them All" introduces over 10 variations of software debouncing. However, we will use our own variation that works with an ISR.

3. Calculating position from acceleration

In order to measure the CPR compression depth, we need to double-intergrate the acceleration. However, the acceleration we get from the accelerometer typically contains a large DC component that come from the earth's gravitational acceleration and the baseline voltage of the accelerometer. This DC component also tends to drift over time, so a simple subtraction does not work. Instead, we rely on filters to attenuate these low frequency components. Since we have not talked about how to implement filters in Arduino, we will use a simple method to calculate the position instead. First, let's assume the CPR motion is sinusoidal without offset, then the acceleration of can be approximated with:

$$A\,cos(2\pi ft\ +\ \phi)$$

where *2A* is the peak-to-peak magnitude of the acceleration; f is the CPR rate; and ϕ is the phase. Integrate this expression twice, we get the expression for the position:

$$\frac{-A}{(2\pi f)^2} cos(2\pi ft\ +\ \phi)$$

This means that our CPR depth is proportional to A/f^2. We will need to find the proportionality constant empirically. This method is crude, but it helps us avoid the instability of numerical integration.

Pre-lab calculations

1. Verify from the device specifications that the DAQ's 5 V terminal can supply adequate voltage and current for the accelerometer board.

2. Determine the theoretical voltage-to-acceleration conversion in SI units.

3. Determine the predicted maximum acceleration if the CPR rate is 120 compressions per minute and the compression depth is 50 mm.

Pre-lab programming

In this pre-lab, you will create an Arduino script that can calculate the CPR rate and depth based on the accelerometer output. We decided to record a tutorial that will guide you through the pre-lab programming because C programming is not a prerequisite of the course. Also, not having access to an Arduino board at home means that you will not be able to test if your code behaves correctly. Nevertheless, please feel free to do as much as the programming yourself before watching the tutorial. Keep in mind that there are often many ways of implementing each step, and there is no right or wrong implementation. The following steps are just our recommendations, and you are welcome to try out your own algorithms.

0. Head to the official Arduino website: https://www.arduino.cc/en/software. Download and install Arduino Integrated Development Environment (IDE). The IDE is free, but please feel free to make a few dollars of contribution.

1. Open an Arduino IDE. This should open a new .ino file with only the skeleton code containing two functions: `setup()` and `loop()`.

2. [Establishing serial communication] In the `setup()` function, initialize serial communication with a baud rate of 9600 bits per second.

3. From now on, you should click the checkmark on the top-left ("verify") every step of the way to confirm that your program does compile.

4. [Collecting data from the accelerometer] In the `loop()` function, do an `analogRead()` from channel A0, and save the result in an integer variable named `analogValue`.

5. [Keeping track of maximum and minimum readings] In the `loop()` function, write code to keep track of the maximum and minimum `analogValue` in real-time. For example, if `analogRead()` returns 5 in the first loop iteration, 8 in the second, then 10, then finally 3, the maximum should be 10, and the minimum should be 3.

6. [Implementing the elastic envelope] Our goal is to estimate the amplitude A of the acceleration formula $A\ cos(2\pi ft + \phi)$. However, the previous step only finds the max and min accelerations **of all time**. In order to keep track of the max and min value of the most recent values, we need to "forget" the previous max and min. We can store only the most recent N values in an array, and find the max and min values of the array in every loop iteration. But doing so would use a lot of memory. Here, we will use an alternative approach to "decay" the max and min values toward the mean. You can think of the max and min values as the envelope of the sinusoid, and this method gives this envelope elasticity as it is constantly narrowing inward

(Figure 6). The width of the envelope is the approximation of *2A*. We can update the max and min values with the following formulas:

- $stepSize = (maxReading - minReading) \times \alpha$
- $newMaxReading = maxReading - stepSize$
- $newMinReading = minReading + stepSize$

where α is a constant. Set α to 0.01 for now. You can experiment with different values when you come to the lab.

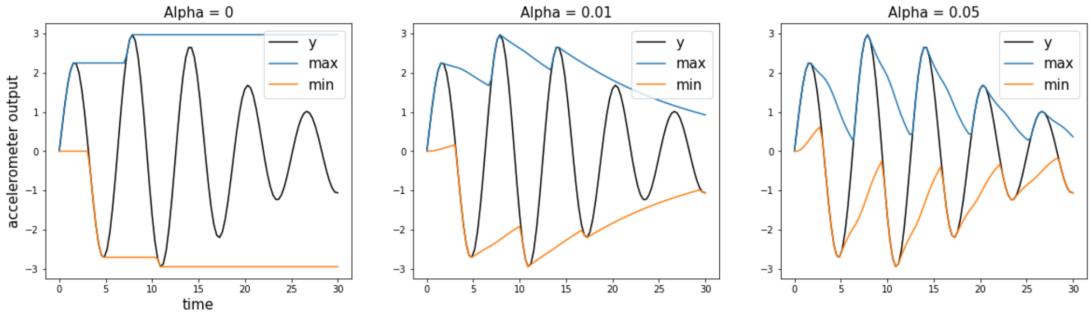


Figure 6. Keeping track of max and min values with different α values.

7. [Calculating CPR rate] Write code to calculate the CPR rate. We cannot just count the total number of peaks and divide by time because it would be the average rate of the entire duration. Instead, we want to know the current rate of CPR. Let's define two threshold values:

- $upperThreshold = maxReading \times 0.7 + minReading \times 0.3$
- $lowerThreshold = maxReading \times 0.3 + minReading \times 0.7$

Having two thresholds allows us to implement hysteresis in peak detection. A CPR period will now be defined as the time it takes for the `analogValue` to (1) cross the *upperThreshold* from a lower value; (2) cross the *lowerThreshold* from a higher value; then (3) cross the *upperThreshold* from a lower value again (Figure 7). Measure this period with the built-in `micros()` function. Take the inverse of this period to calculate the CPR rate. Store this rate in a variable. We will display the rate on an OLED display in the lab.

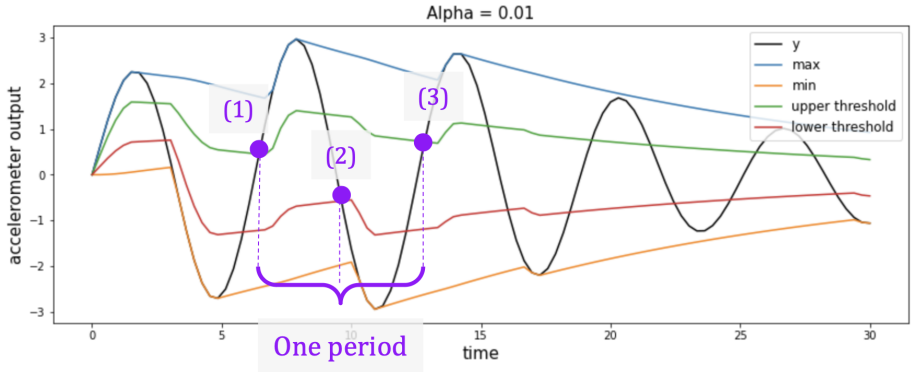


Figure 7. The length of the period is measured with hysteresis.

8. [Convert accelerometer output to acceleration] Write a function to convert the accelerometer output first from the range [0, 1023] to volts, then from volts to m/s^2.

9. [Calculating CPR depth] As mentioned in the beginning of this handout, the CPR depth is estimated to be proportional to A/f^2. Use a proportional constant of 0.7 for now, calculate the CPR depth and store it in a variable. We will display this depth on an OLED display in the lab.

10. [For advanced students: Implementing moving average in a separate file] Write code to calculate the moving averages of the CPR rate and depth. This makes our program less susceptible to noise and sudden movements. Please see **Appendix 1** at the end of the handout for our implementation.

In-lab activities

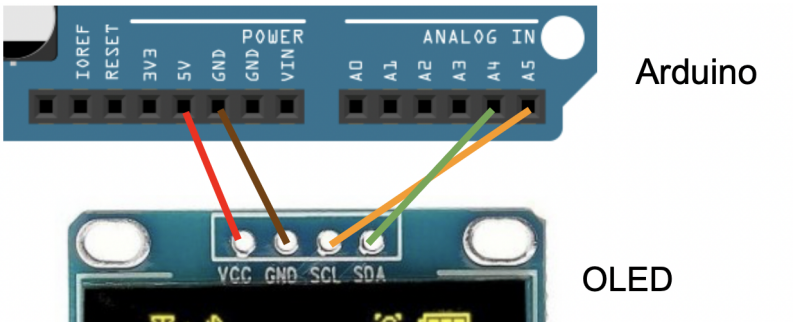
1. Connect the ADXL335 accelerometer to Arduino Uno according to the following:

Arduino Uno pin	ADXL335 pin
5V	Vin
GND	GND
A0	Zout

2. Add a serial print statement in your program to print `analogValue`, i.e., the accelerometer reading from analog input A0.

3. [Visualize accelerometer output] Upload your script to Arduino Uno. Then, go to Tools > Serial Plotter to visualize the accelerometer output. Apply sinusoidal, CPR-like movement to your accelerometer and observe the results.

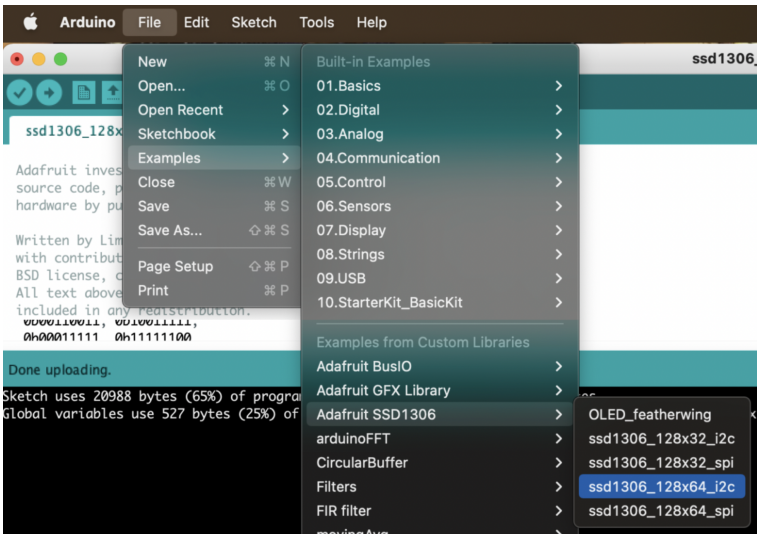
4. [Wire the OLED module to the Arduino] Wire according to the following diagram:



5. [Confirm your OLED display is working properly] Downloading the library necessary for driving the OLED display. In your Arduino IDE, go to Tools > Manage Libraries. Download/install the following libraries (and their dependencies, if any):

- Adafruit GFX Library
- Adafruit SSD 1306

Then, go to File > Examples > Adafruit SSD1306 > ssd1306_128x64_i2c.



Change the SCREEN_ADDRESS variable to **0x3C.** This is the I2C address of the OLED module we use in this class.

```
// Declaration for an SSD1306 display connected to I2C (SDA, SCL pins)
// The pins for I2C are defined by the Wire-library.
// On an arduino UNO:       A4(SDA), A5(SCL)
// On an arduino MEGA 2560: 20(SDA), 21(SCL)
// On an arduino LEONARDO:   2(SDA),  3(SCL), ...
#define OLED_RESET     4 // Reset pin # (or -1 if sharing Arduino reset pin)
#define SCREEN_ADDRESS 0x3C ///< See datasheet for Address; 0x3D for 128x64, 0x3C for 128x32
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);
```

Upload the example sketch to your Arduino, and you should see some vibrant animations starting with the Adafruit logo appearing on your OLED screen (it might take a while initially). If not, check the wiring or hit the reset button on Arduino.

Note: On the back of your OLED, you will see the I2C address says 0x78 instead of 0x3C. We know from the prefix 0x that those numbers are written in hexadecimal. Converting 0x78 from hex to binary becomes 0111 1000. In standard I2C convention, the least significant bit (LSB) is used as a read/write flag, and a zero in this case means that you may write to this I2C address. Many manufacturers incorrectly incorporate this read/write flag when converting the binary I2C addresses to hex, that is, 0x78 is actually an invalid I2C address. You should really just use the first 7 bits. Therefore, after removing the LSB (shifted right), we get 0011 1100, which corresponds to 0x3C in hex.

6. [Establish communication with the OLED display] Now go back to your CPR monitor script. On the top of your own sketch, include the following preliminaries:

```
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#define SCREEN_ADDRESS 0x3C
#define DISPLAY_WIDTH 128
#define DISPLAY_HEIGHT 64
```

```
#define OLED_RESET -1 // 1 if sharing Arduino reset pin
Adafruit_SSD1306 display(DISPLAY_WIDTH, DISPLAY_HEIGHT, &Wire, OLED_RESET);
```

In the `setup()` function, initialize serial communication with the OLED display with the following:

```
display.begin(SSD1306_SWITCHCAPVCC, SCREEN_ADDRESS);
```

This command establishes the I2C communication between your Arduino and OLED. `SWITCHCAPAVCC` tells the Arduino to switch on the OLED internal display voltage. After this, include the command:

```
display.clearDisplay(); // clear buffer
```

This is because, by default, the display buffer starts with the Adafruit logo. But we would like to display our own message! So this command allows us to start with a clean slate.

7. [Show the CPR rate on the OLED display] Write a function named `displayRate()` that take a rate parameter and display it on the OLED display. For example, if I want to display out the following message:

I can use the following commands:

```
display.setTextSize(2); // Default size is 1
display.setTextColor(1); // Draw white text (must include!)
display.setCursor(0,0); // Start at top-left corner
display.println(F("    Lie\n  Detector"));
display.setTextSize(1);
display.println("");
display.println(F("  Tell us the truth!"));
display.display();
```

Note: Notice that the OLED will only update if you call the function `display.display()`. Otherwise, you are just modifying the display buffer. Also, I wrapped the string with the `F()` function. By doing so, I told the compiler that I wanted this string to be stored in the flash memory (where the program is kept), thus freeing up more space for other data in the RAM.

8. [Connecting the button to Arduino] Let's use a push button to control whether to display the CPR rate or depth on the OLED display. Wire one end of the button to

digital pin 2, and the other end to the ground. Then, in the `setup()` function, enable the pull-up resistor of this pin.

9. [Write an ISR] Initialize a global, volatile boolean variable named `showRate`. Because we will display only one information (either rate or depth) at the time, we will use this variable to toggle what is being displayed.

```
volatile bool showRate = true;
```

Note: the variable `showRate` must be declared `volatile` to prevent the compiler from optimizing it away.

Then write an ISR to toggle this variable:

```
void isrSwitch() {
  showRate = !showRate;
}
```

10. [Attach the ISR to pin 2] In the `setup()` function, include the following:

```
attachInterrupt(digitalPinToInterrupt(2), isrSwitch, FALLING);
```

Note: The `FALLING` mode means that whenever pin 2 goes from high to low, the `isrSwitch()` function will be called. There are also `RISING` and `CHANGE` modes. The `digitalPinToInterrupt()` is a built-in function that converts the pin number to the *interrupt number*.

11. [Show the CPR depth on the OLED display] Write a function named `displayDepth()` that take a depth parameter and display it on the OLED display. In the `loop()` function, write a conditional statement that calls `displayDepth()` if showRate is true, and calls `displayDepth()` otherwise. Upload the script to Arduino and try it out!

12. [Debouncing ISR button] As we mention at the beginning of this handout, mechanical button bounces when they are pressed. This can make the user really frustrated because it often takes several presses just to get the right information on the OLED. To mitigate this, write code to implement a "refractory" period of 200 ms during which no other edge transitions should be registered. Use the built-in `millis()` function for timing. We have included our version in **Appendix 2**, but try it out yourself before looking at our implementation.

13. Apply sinusoidal, CPR-like movement to your accelerometer and observe the results.

14. Read the Lab 5 Report section below and collect any data or observations that you would like to include in your report.

15. Demonstrate your working VI to an instructor.

Deliverables

1. Upload your completed `.ino` file to the Canvas drop box.

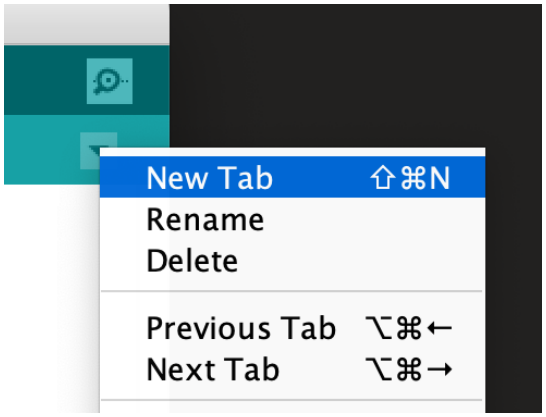
2. Each team member should write a document that includes the items listed below. The technical content such as specifications, test results, standards that state how CPR should be done, circuit diagram(s) and equations may be shared among the teammates if you choose to, but the writing should be by each individual author. Grading will primarily be based on completeness, rather than on the quality of writing. However, if you believe that you have written the document to the best of your ability and would like feedback on your writing, please include a comment saying so when you upload your documents to Canvas.

- Purpose of a CPR feedback device (1-2 paragraphs)
- Relevant standard(s) that govern performance of CPR (one paragraph);
  You should at least consider the instructions in Section 8 of the AHA's CPR guidelines:
  https://eccguidelines.heart.org/circulation/cpr-ecc-guidelines/part-5-adult-basic-life-support-and-cardiopulmonary-resuscitation-quality/#
- Specifications for your device (targets - you do not have to meet them in lab 5)
- The system you developed, including
  - Hardware
  - Software
  - Mathematics
- Tests you ran
- Results of the tests, including estimated measurement uncertainty
- Discussion, including suggestions for further development (2-3 paragraphs)

Appendix 1: Implementing moving average buffer in a separate file

Moving average is such a commonly used tool that we want to place it into a separate file, so we can reuse it later. The names of tool files like these will often ends with `.h`, which stands for "header." For larger projects, the header files will only contain the declarations, while the implementations can be found in `.cpp` files. Because this is only a small project, we place both the declarations and implements of the moving average filter in the header file named `MovingAvg.h`.

1. [Opening a new file] On the upper right hand corner of the Arduino IDE, click on the downward arrow > New Tab. Name the new file `MovingAvg.h`



2. [Adding an include guard] To prevent the header file from being included multiple times, it is customary to wrap the entire content of the file with the conditional compilation directives:

```
#ifndef MOVING_AVG_H_
#define MOVING_AVG_H_

// Put your code here...

#endif // MOVING_AVG_H_
```

The code between `#ifndef` and `#endif` will only be compiled if `MOVING_AVG_H_` is defined beforehand. Here, we `#define MOVING_AVG_H_` between `#ifndef` and `#endif` so that it will be defined exactly once.

3. [Implement the MovingAvg class] Our moving average filter will be made of a circular buffer, a write pointer, a `push()` function that will add a new value to the circular buffer, and a `getMovingAvg()` function to get the moving average. Using a class is a good way to group those attributes and functions together. There are other C++ concepts that we do not have time to go over in the course, so instead, we will just show you our implementation of `MovingAvg.h` on the next page.

```
/*
 * A simple moving average buffer.
 */

#ifndef MOVING_AVG_H_
#define MOVING_AVG_H_

template<typename T, size_t S, typename ST>
class MovingAvg {
public:
  MovingAvg() {
    // Populate buffer with zeros.
    for (int i = 0; i < S; i++) {
      circularBuffer[i] = static_cast<T>(0);
    }
    writeIndex = 0;
    sum = static_cast<ST>(0);
  }

  void push(T value) {
    if (isnan(value) || isinf(value)) {
      return;
    }
    // Store oldest value in a temp variable before overwriting it
    T oldestValue = circularBuffer[writeIndex];
    circularBuffer[writeIndex] = value;

    // Increment pointer
    writeIndex++;
    writeIndex = (writeIndex >= S) ? 0 : writeIndex;

    // Update size
    bufferSize++;
    bufferSize = (bufferSize > S)? S : bufferSize;

    // Update sum
    sum += value - oldestValue;
  }

  float getMovingAvg() {
    if (bufferSize == 0) {
      return 0.0;
    }
    return sum / static_cast<float>(bufferSize);
  }

private:
  T circularBuffer[S];
  int writeIndex;
  int bufferSize;
  ST sum;
};

#endif // MOVING_AVG_H_
```

4. Here is an example use of the MovingAvg class:

```
#include “MovingAvg.h”

// Construct a MovingAvg object named analogValues with a integer circular buffer
// of size 50. The data type of the sum variable is long to prevent integer overflow.
MovingAvg<int, 50, long> analogValues;

void setup() { Serial.begin(9600); }

void loop() {
  analogValues.push(analogRead(A0));  // Push a new value into the buffer
  Serial.println(analogValues.getMovingAvg());  // Print the current moving average
}
```

Appendix 2: Software debouncing with ISR

1. Create a new global volatile boolean variable named `isInterrupted`, initialize it to false.

2. Rewrite the `irsSwitch()` function such that its only job now is to change the variable `isInterrupted` to true.

3. When `isInterrupted` is true, toggle the variable showRate once. Then, mark the current time using the `millis()` function, which will return how many milliseconds since the program starts. As long as the difference between the current time is less than 200 milliseconds, the program will not react to another interrupt. See our implementation below:

```
#define DEBOUNCE_PERIOD_MS 200
const byte interruptPin = 2;
bool showRate = true;
volatile bool isInterrupted = false;  // ISR flag

void setup() {
  // The other setup lines are not shown

  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin), isrSwitch, FALLING);
}

void loop() {
  // The other lines are not shown

  // Debouncing ISR button
  static unsigned long lastInterruptTime = millis();  // will overflow in ~50 days
  if (isInterrupted) {
    int interruptInterval = millis() - lastInterruptTime;
    if (interruptInterval > DEBOUNCE_PERIOD_MS) {
      showRate = !showRate;
    }
    isInterrupted = false;
    lastInterruptTime = millis();
  }

  // Decide what to display:
  if (showRate) {
    displayRate(cprRate);
  } else {
    displayDepth(cprDepth);
  }
}

void isrSwitch() {
  isInterrupted = true;
}
```