1 Introduction

In this lab, you will construct your own digital oscilloscope. The Arduino microprocessor will perform two roles: it will act as a function generator to provide an input signal, and will acquire waveforms from its analog input, storing both the time and voltage read on the analog input. This data is downloaded to a PC via the same serial USB connection which you use to program the Arduino. The waveforms are then plotted using the matplotlib utility of Python.

Your scope will be nearly fully featured. It will have a trigger requirement, that will cause data to be collected only when the signal passes above a programmable threshold. It will have a delay feature, so that some number of samples collected before the trigger requirement is met will be included in the wave form.

2 Serial Output of the Arduino

From the course website, download the "DigitalScopeStart.ino" Arduino sketch and upload it to your Arduino using the Arduino software (Since this is only a file, the software should prompt you about making a directory folder to contain this sketch, which you should allow.) This program is a useful starting point for building your own digital oscilloscope. It includes the function generator from the previous lab, and includes a shell for the Digital Scope program, including a circular data buffer and the complete serial interface for requesting and receiving the scope data on the PC over the USB serial port.

To see how the serial interface works on the Arduino side, open a serial monitor using the Arduino software on your PC (making sure it is set to the same rate as the program: 9600 baud). When you open the serial connection, the Arduino initializes, which (in this sketch) includes sending the message "Arduino Initialized" on the serial port. This informs you that the Arduino is now ready and listening to the serial port for commands.

Send a single "a" character on the serial monitor (the "a" stands for "acquire data"). The Arduino receives this command, and, in response, sends back a payload of data to the PC via the serial link, which you should see in the serial monitor. The format of the payload is a single integer N (in this case 50) which tells the PC how many x and y coordinates will follow, then N lines of containing one x and one y coordinate. At this moment, we are just testing the Arduino serial input and output, so the payload consists of fake data (Your job will be to turn this into real scope data).

Code Details: The code which controls the serial interface includes these lines, which initialize the serial interface to 9600 baud, then prints an initialization message to the serial output:

```
// initialize serial interface:
Serial.begin(9600);
Serial.print("Arduino Initialized\n");
```

The handling of the "a" serial input, which causes data to be sent on the serial port is handled in the following special function:

```
void serialEvent() {
  while (Serial.available()) {
    char inChar = (char) Serial.read();
    if (inChar == 'a'){
      // this command overrides current state, we go to ACQUIRE:
      scope_state = ACQUIRE;
      digitalWrite(led, HIGH);
```

```
      return;
    }
  }
}
```

The `serialEvent()` function is called after each call to loop if serial input is available. So this loop simply looks for the "a" character, and sets the state of the digital scope to `ACQUIRE` mode if it is found. The led is set high and will remain lit until the requested data has been sent to the PC.

When the digital scope's circular buffer is full (a complete waveform has been captured) the scope state moves to FULL, and the timestamps and ADC input stored in the circular buffer (`cbuf_time` and `cbuf_adc`) is sent to the PC over the serial link:

```
  if (scope_state == FULL) {
    char line[50];
    sprintf(line, "\%d\n", cbuf_size);
    Serial.print(line);
    for(int i=0; i<cbuf_size; i++){
      sprintf(line, "\%ld \%d\n", cbuf_time[i], cbuf_adc[i]);
      Serial.print(line);
    }
    //data has been overloaded, go back to standby:
    scope_state = STANDBY;
    digitalWrite(led, LOW);
  }
```

3 PC Driver for the Arduino Digital Scope

Close your serial monitor before proceeding! You cannot access the serial port through the python tools described in this section if you already have a serial monitor open in the Arduino IDE.

We want to automate the communication between the PC and the Arduino (in the previous section, you controlled the Arduino from the serial monitor). This driver code is handled by a second Python program running on the PC. Download the "serial_eg.py" Python script from the course site. Run this script from the command line as "python serial_eg.py" from the directory (e.g. Desktop) where you have downloaded the serial_eg.py file to.

The program will prompt you for the serial port address, which you can usually read off from the bottom of the Arduino program which you ran in the previous step.

After connecting to the serial port you select, the python script will wait to receive the "Arduino Initialized" message from the Arduino. Then it will prompt you to press enter when you are ready to acquire data. Press Enter. The python script will send the acquire command (a character 'a' send over the serial port) and the Arduino will send its payload of x and y coordinates.

After receiving the data from the Arduino, the python script uses matplotlib to plot the data. If this works, you now have a way to acquire data that originated on the Arduino and display it on your PC. In other words, you have a data acquisition system.

For Convenience: If you are tired of entering the serial port address by hand, you can modify the `serial_eg.py` script to use a constant value appropriate for your setup instead of prompting the user each time. Just remember to change this if you serial port changes location!

Code Details: Note: if you want to use this serial python script on your laptop, you will need to make sure that you have `pyserial`, `matplotlib`, and `numpy` installed. You won't need to modify

the python code to complete this lab, but for the curious, the code which handles the serial interface includes waiting for the initialization message "Arduino Initialized"):

```
print ser.readline().strip()
```

send the"a" character to the serial output:

```
ser.write('a')
```

read the size n of the input data:

```
n = int(ser.readline().strip())
```

and finally read and parse n lines of input:

```
str = ser.readline().strip()
x,y = str.split()
```

4 Using the Function Generator Feature

Do not send signal from a lab function generator (or any other external power source) to the Arduino. If we had more time, we could devise circuits which map an arbitrary input voltage to the 0-5V range the Arduino analog inputs are capable of digitizing. Instead, we are only going to send signals that originate on the Arduino (and are already in this allowed range) as input to the analog inputs.

This sketch includes the complete function generator from the previous lab, on output pin 5. By default it is outputting a sine wave. Reproduce the low pass filter from the previous lab and check, with a lab scope, that you have a nice 10 Hz sine wave at the output of the filter.

Using a small wire, simply connect the output of your function generator filter circuit to the input of your Arduino, at, say, Analog Input 0.

5 Programing the Arduino Digital Scope

Do not send signal from a lab function generator (or any other external power source) to the Arduino. If we had more time, we could devise circuits which map an arbitrary input voltage to the 0-5V range the Arduino analog inputs are capable of digitizing. Instead, we are only going to send signals that originate on the Arduino (and are already in this allowed range) as input to the analog inputs.

The data for this demonstration code is fake data, as you can see from these lines in the `setup()` function which fill the circular buffer used to store the scope data:

```
// fill scope buffer with fake data:
 for (int i=0; i<cbuf_size; i++){
    cbuf_time[i] = i;
    cbuf_adc[i] = 10*(i%25);
 }
```

When the scope sees an acquire request on the serial port (a single character "a") this fake data is immediately sent on the serial port. No actual data is transmitted.

Your task for this lab is to replace this fake data payload with real data acquired by the Arduino using the analog inputs. You'll need to use `analogRead()` which you learned about in the first lab. **Trouble Shooting:** Some of the Arduinos have broken A0 or A1 inputs. In this case, just use another input (e.g. A5). The reading of this input is already implemented in the loop() function:

```
  unsigned long current_time = corrected_micros();
  unsigned current_input = analogRead(inp);
```

but the `current_input` isn't yet used.

To make this all work, you'll need to understand the Digital Scope state machine we'll be using, which has the following steps already defined:

```
enum State {STANDBY,ACQUIRE, ENABLED, TRIGGERED, FULL};
State scope_state;
```

but not all states are yet implemented. Here's the description and status of each state.

- `STANDBY` awaiting acquisition request on serial port (already implemented).
- `ACQUIRE` acquire request has been received on serial port (**partially implemented**).
- `ENABLED` we've already collected enough data to cover the delay period (before trigger) so we can now enable a trigger (**not yet implemented**)
- `TRIGGERED` the trigger requirement has been met (**not yet implemented**)
- `FULL` the circular buffer contains the complete waveform, offloading to the PC.

The demonstration you've been given does not correctly implement the `ACQUIRE`, `ENABLED` or `TRIGGERED` state. Instead, as soon as it enters the `ACQUIRE` state, it immediately goes to `FULL` state and begins downloading the fake data:

```
// for now, simply send buffer as is when acquire received:
  if (scope_state == ACQUIRE){
    scope_state = FULL;
  }
```

You'll want to comment out these lines of code and replace it will your own logic, which we'll discuss in the next sections.

6 No Trigger Requirement

For the first step, we can ignore the trigger and delay requirements and simply start collecting data immediately upon entering the ACQUIRE state. To do this, I would leave the `ENABLED` and `TRIGGERED` states unimplemented. Replace the ACQUIRE state logic with something like:

```
 if (scope_state == ACQUIRE){
    if (current_time > (last_sample_time + sample_period)){
      last_sample_time = current_time;
      cbuf_time[icbuf] = current_time;
      cbuf_adc[icbuf] = current_input;
      sample_count++;
      icbuf++;
      if (icbuf >= cbuf_size) icbuf=0;
    }
 }
```

Make sure you understand what this step does. It checks if enough time has passed to take another sample, If so, it records the time and adc value in the circular buffer at location `icbuf`. Then it increments this location. As this is a circular buffer, if the index reaches the end, it simply places this index back to the beginning. It also sets the `last_sample_time` to the `current_time` so that the next sample will be taken after a sufficiently long interval, and increments the `sample_count`.

To complete this portion, simply add logic to move the state from `ACQUIRE` to `FULL` when the `sample_count` is sufficiently large. The code already implements the `FULL` state, which downloads the data and returns the state to `STANDBY`.

When this stage is working, you should see nice sine wave output when you run `serial_eg.py`. However, you should notice that the phase of the function is completely random (it changes each time) because you have no trigger requirement.

7 Add a Trigger and Delay Requirement

For a fully featured scope, you need a trigger requirement and delay (ability to show wave form for some time before trigger requirement was met).

This should be implemented by using the `ENABLED` and `TRIGGERED` state as follows. You enter the `ENABLED` state after you have collected at least `delay_samples` samples. Once in the `ENABLED` state, you are simply waiting for the trigger requirement to be met. You continue to update the circular buffer exactly as for the `ACQUIRE` state. In fact, you should use that same block of code for all the states which collect data, by extending the condition to

```
if ((scope_state==ACQUIRE)||(scope_state==ENABLED)||(scope_state==TRIGGERED)){
 //...
}
```

and moving the state logic specific to each of these states into separate blocks.

Within the `ENABLED` state, if the trigger threshold is met, the `sample_count` is set back to zero, and the `TRIGGERED` state is entered. Within the `TRIGGERED` state, you continue to update the circular buffer, but, when you've collected enough events to fill the circular buffer (including the already collected `delay_samples`!) you move to the `FULL` state.

8 Collect Sine Wave, Square Wave, and Saw Wave Data

Once you have your scope working, collect some different wave forms by adjusting the mode of the function generator. Check that you can adjust the phase by adjusting the number of delay samples collected. Check that you can adjust the timescale by adjusting the `sample_period`.