ELECTRICAL ENGINEERING DEPARTMENT / CAL POLY STATE UNIVERSITY

EE143 Lab #7bc

Real Time Digital Signal Processing

Optional - Storage Scope and Spectrum Analyzer

PRELAB:

Review the following Arduino tutorial and web site links, as needed.
https://www.youtube.com/watch?v=fCxzA9_kg6s&list=PLA567CE235D39FA84
https://www.arduino.cc/en/Guide/Introduction Arduino software (IDE)
https://www.arduino.cc/en/Guide/Environment Arduino libraries
https://www.arduino.cc/en/Guide/Libraries Arduino Troubleshooting
https://www.arduino.cc/en/Guide/Troubleshooting

You may wish to locate an audio jack and cable that can be adapted for this lab, as well as a small speaker. See procedure.

PURPOSE:

- To use an Arduino in a practical application involving a signal processing and, optionally, to use a storage oscilloscope and spectrum analyzer
- To gain further practice bread boarding circuits, this time circuity peripheral to an Arduino
- To gain further practice in modifying Arduino code

This experiment relates to the following course learning objectives:

1. Ability to build a complete hardware / software Arduino system
2. Ability to analyze and evaluate data.

STUDENT PROVIDED EQUIPMENT:

1 Arduino
1 LM358 Op Amp
1 Breadboard
1 10kΩ potentiometer
1 20kΩ 1⁄4 Watt Resistor 1 10kΩ 1⁄4 Watt Resistor 1 4.3kΩ 1⁄4 Watt Resistor

Optional – needed optional parts of lab:
1 1/8" Stereo audio jack and cable. See procedure – audio cable will need to be cut and wires soldered to provide breadboard connections (use an old cable?)
1 Music/Signal source (such as a phone or audio from a computer)
1 Bookshelf-style speaker or powered speaker (or use the little speaker from DigiKey)

BACKGROUND:

This experiment provides an introduction to the concept of digital signal processing (or "DSP"). This method of processing signals uses arithmetic on a computer instead of the Op Amp-based methods used

in analog signal processing. Given the flexibility of processing via computer, the opportunities with DSP are vast. We will explore a few simple styles of processing. Also, as an optional part of the experiment, you'll have an opportunity to make a DSP-based filter and echo effect. (These require an audio jack and speaker, see below).

Level Shifting an Input Signal

Many signals are "two-sided" meaning they swing both positive and negative. Audio signals are two-sided, for example, typically swinging between +/- 1 Volt. To permit the Arduino to read these signals we need to perform a "level shifting" operation that adds a constant to an input signal, making it positive. The circuit shown in Figure 1 uses an LM358 Op Amp to accomplish the level shifting.

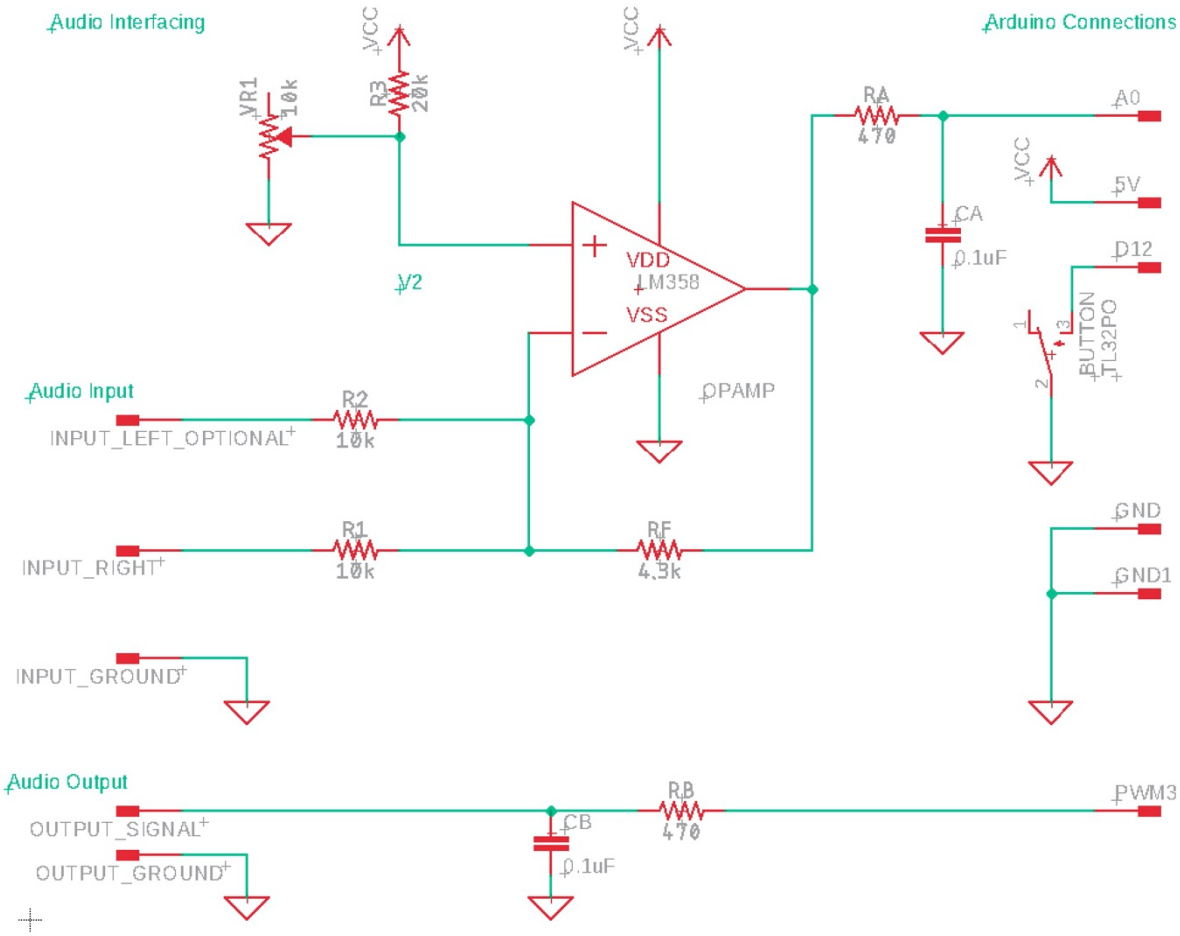

Figure 1. Audio input, level shifter and button interface for Arduino. RA, CA and RB, CB are all optional but will improve sound quality. The left audio input is also optional.

Note the voltage divider formed by the potentiometer and 20k resistor. It sets the non-inverting input to a constant value (V_2) that you can adjust. Because of the virtual short between the inputs of the Op Amp, the voltage at the inverting input is also V_2. Equating currents at the inverting input gives

$$\frac{V_{in} - V_2}{10k} = \frac{V_2 - V_{out}}{4.3k}$$

$$V_{out} = 1.43\, V_2 - 0.43\, V_{in}$$

Thus, we are able to adjust an incoming two-sided signal to be strictly positive, making it readable by the Arduino A/D. *Note that component values were constrained by the original parts kit. Also note that the RA, CA and RB, CB components are optional but help improve sound quality when working with audio signals (optional).*

PWM Output Signal

To generate an output voltage, we will use a pulse width modulated (or "PWM") signal. As shown in Figure 2, the PWM signal is binary and it is the average value of the signal that is associated with its useful voltage. The average is determined by the ratio of the "on time" Delta, to the period T. The reason the average is the dominant characteristic of a PWM signal is because the period, T, is typically very brief compared to the rate of change of other signals in an application. In this experiment the period is less than 1/30000 of a second.

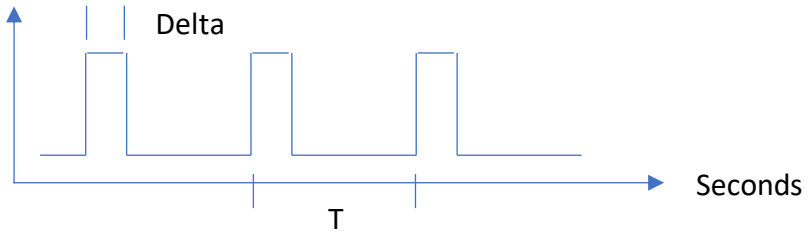

Figure 2. A PWM Signal.

To employ a PWM signal in the most proper fashion it is necessary to introduce a filter that helps to smooth out the rapid high/low variations, so that the average is effectively the only component of the signal present. In this experiment we do not have a proper filter to accomplish the smoothing. Rather we rely on the resistance of the Op Amp output and the inductance of a speaker to form an analog low pass filter (in the optional part of this experiment, involving audio signals). This filtering is not really sufficient but may help to illustrate the importance and usefulness of analog filtering in an application like this. You'll learn more about filters in EE 211.

Signal Processing

Having analog input and output voltages, we are ready to process signals on the Arduino. The Arduino system calls the function setup() once at power up and then calls the loop() function repeatedly, as long as power is applied. In DSP systems we typically want to control the rate of processing signals. In this case a delay is introduced that pauses the loop() function until 1/8000 of a second has elapsed. The function micros() is used to read the current time (since power up) in microseconds. See software listings in the Appendix.

In this lab you'll try several kinds of simple processing on the Arduino and various tests to verify the system is working properly. These tests include constant signals, a pass through where output equals the input, and a threshold operation similar to the analog continuity tester.

DSP - Optional

As an optional part of this lab you can try processing an audio signal with a simple digital filter and with an echo effect. For these kinds of operations, the value from the A/D (a "sample") is read and then stored to provide a brief history of the input signal.

The digital filter uses a very short history that just includes the two previous values of the input: X_{prev} and X_{prev2}. To compute an output for the filter, Y_{out}, the Arduino performs a calculation similar to:

$$Y_{out} = 0.333 * X_{in} + 0.333 * X_{prev} + 0.333 * X_{prev2}$$

Where X_{in} is the current input from the A/D. The above calculation is just an average of course, but it does constitute a simple filter. (Can you guess whether the filter would be high pass or low pass?) In a more general case, the 0.333 weighting values will vary for each term and the history of past output values are commonly included in the calculation as well. Changing these coefficients will change the type of filter (high- low, band-pass) as well as the filter cutoff. You'll learn more about (analog) filters in EE211 and you can learn more about digital filters in EE 328/368 and EE 419/459.

In the case of the echo effect an average is also computed. However, in this case the earlier input value is 1500 samples prior to the current input. This corresponds to a delay of 1500/8000 of a Second, or around 0.2 Sec. As you can appreciate, the averaging operation would also imply that filtering is occurring. However, the more dominantly perceived audio effect is an echo.

Signal Visualization – Optional

As another optional part of this lab you can explore wo methods to visualize a signal. One method is known as a storage oscilloscope and the other is a spectrum analyzer. In each case you'll play music or other audio into the Arduino. It will capture approximately 60 mSec of audio and transfer it up to the Serial Plotter utility (which is part of the Arduino IDE). See Figures 3 and 4. You'll learn more about oscilloscopes and spectrum analyzers in future classes, this experiment is intended to be an introduction only. Scopes depict signals as a function of time; spectrum analyzers present them as a function of frequency.

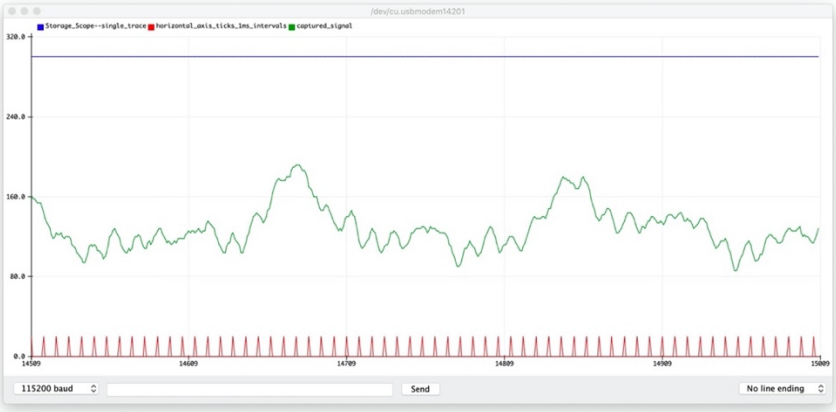

Figure 3. Storage scope displaying music. Red tick marks occur at 1 mSec intervals.

Figure 3 illustrates the storage scope mode. The captured signal (music) is plotted versus time. The red timing marks occur at 1mSec intervals. The numbers up the vertical axis indicate the value of the A/D as read by the Arduino, divided by 4 to convert from 10 bits to 8 bits. (Ignore the numbers along the horizontal axis – they are arbitrary.)

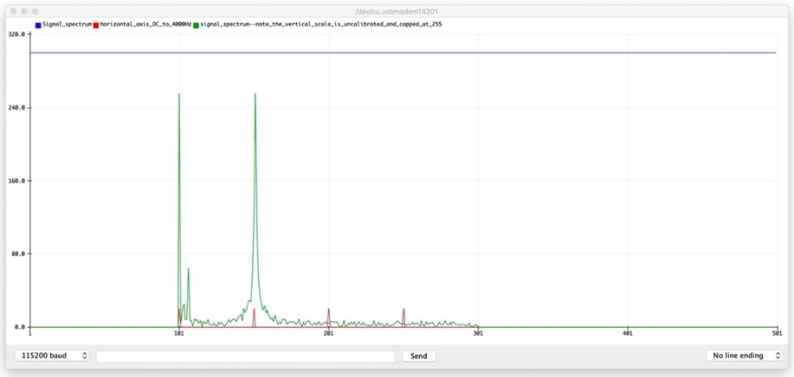

Figure 4. Spectrum analyzer displaying the frequency content of a tone at ~1000 Hz. Red tick marks occur at 1000 Hz intervals, ranging from DC to 4000Hz. Vertical axis is uncalibrated.

For these programs, the Arduino loops at 8000Hz. Each time through the loop the Arduino checks to see if the button has been pressed. As soon as the button input goes active (low) the Arduino captures the next 500 input values from the A/D, at the 8000Hz rate, and stores them in an array. Once all the values have been captured, they are transmitted up to the Serial Plotter program. The Plotter program is started from the Tools menu. One data capture and one upload are done per button press.

PROCEDURE and ANALYSIS:

1. Build the circuit shown in Figure 3. Use one of the Op Amps in the LM358. Only two connections are needed for the potentiometer, the middle wiper and one end. Leave the third connection open. Note that RA, CA and RB, CB can be omitted (you likely do not have these in your lab kit). Replace the resistors with a wire, the capacitors with a break or open.
2. Power the breadboard using the Arduino.
3. Copy and paste the Arduino program in Appendix 1. Find the comment that says

   ```
   // uncomment one of the four methods below to compute the output PWM value
   ```

4. Uncomment method 1 and assign a value to the PWM output of 255 `y_out = 255;` Then upload the program and measure the PWM output voltage. Record data in Table 1. Repeat for an output of 0 to the PWM.
5. Comment out method 1 and uncomment method 2 in the Arduino program. This is a pass thru with `y_out = x_in;` Upload this program. Adjust the potentiometer yield an offset voltage (V2) of approximately 1 Volt. Record your value in Table 1.
6. Build a voltage divider that provides some voltage under 2 Volts, using any available resistors and record the value of this voltage.
7. Connect one of the audio inputs to your voltage divider and leave the other open. Measure the output voltage of the Op Amp and record in Table 1.

8. Measure the output voltage of the PWM and record in Table 1, then complete % error calculations in Table 1.
9. Comment out method 2 and uncomment method 3 in the Arduino program. This is a threshold operation. Upload this program. Disconnect A0 from the Op Amp output and connect it to 0V and then to 5V and record the PWM output in Table 1.

Table 1. Signal Tests

Version of Software Processing	Measurement	Value
Version 1, y_out = 255;	PWM Output Voltage	
Version 1, y_out = 0;	PWM Output Voltage	
Version 2, y_out = x_in;	V2 offset voltage	
As above	Voltage of voltage divider, Vin	
Version 2, y_out = x_in; V divider as input	Output voltage of Op Amp	
As above	% Error in Op Amp Output	
As above	Output voltage of PWM	
As above	% Error in PWM Output	
Version 3, Threshold operation A0 at 0V	PWM Output Voltage	
Version 3, Threshold operation A0 at 5V	PWM Output Voltage	

THE REMAINING PORTIONS OF THIS EXPERIMENT ARE OPTIONAL

10. Prepare an 1/8" audio jack to use with a signal source, for example a phone or computer. See Figures 5 and 6. The circuit in Figure 1 accommodates a stereo input, but a mono source can be used (just drop the associated 10k resistor too. Carefully cut the audio cable, leaving plenty of length. Strip the outer sheath exposing a bare ground wire and inner, insulated conductor (red in Figure 6). Strip the insulated signal conductor and solder wires to each wire in the audio cable. Attach wires that can be inserted into your protoboard (orange in Figure 6).
11. Hook up a small speaker in a manner similar to the audio cable, so that you can connect it to signals on your breadboard.
12. Reconnect the A0 input if needed and switch to version 4 of the Arduino program in Appendix 1. Note the comment that describes switching to the internal analog reference.
13. Adjust your signal source (phone or computer) to half the maximum volume.
14. Connect one speaker wire to ground and the other to the output of the Op Amp. Adjust the volume level of your signal source and adjust the potentiometer for the best possible sound.
15. Download the storage scope program listed in Appendix 2 and run it. Open the Serial Plotter, via the Tools menu in the Arduino IDE. Set the baud rate in the serial plotter to 115200 (see the menu in the lower left).
16. Press the button and you should see a signal plot similar to Figure 3.

17. To run the spectrum analyzer program, change the Boolean variable “dft_mode” from false to true, then recompile and upload. Capture data by pressing the button – Note the spectrum may take up to 30 seconds to compute and display.

Figure 4. An 1/8” stereo audio jack.

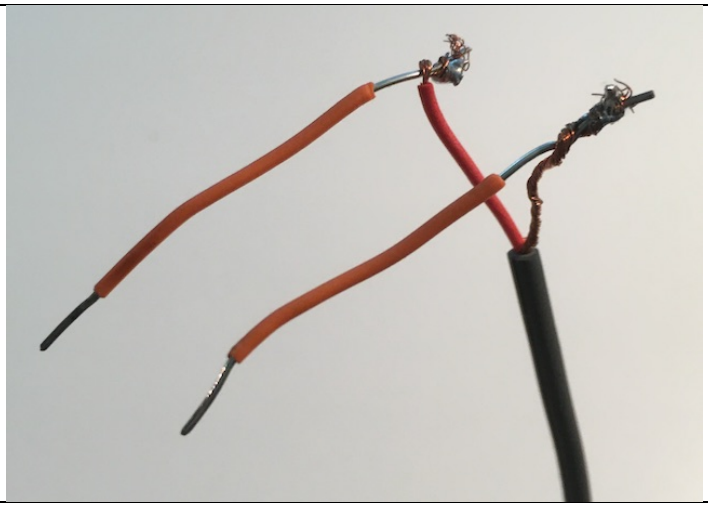

Figure 5. Audio cable with ground conductor and inner signal line. This is a mono signal cable.

DISCUSSION:

Document your efforts via:

- Describe challenges and troubleshooting, if any.

APPENDIX 1 – Software for Testing and Digital Filter

```
/*
  DSP - Example program

  This example code is in the public domain.
*/


////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////

// define sample rate using delta_time
//  with delta_time =  125;  the sample rate is 8000 Hz
//  with delta_time = 1000;  the sample rate is 1000 Hz
//  the function micros returns a count in microseconds but increments by 4, every 4 microseconds
unsigned long delta_time = 125;


////////////////////////////////////////////////////////////////////////
// micros return a count in microseconds but increments every 4 microseconds
unsigned long curr_time = 0;
unsigned long next_time = delta_time;


// variables for the current output, current input and a short history of the input, for
filtering
int y_out = 0;
int x_in = 0;
int x_prev = 0;
int x_prev2 = 0;

// pwm output pin, can be 3,5,6,9,10,11
int drv_pin = 3;

// Button connects to ground on pin 12
const int BUTTON = 12;



////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
// the setup routine runs once when you press reset, or when a program is loaded
void setup() {

  // set PWM frequency to 31372.55 Hz
  TCCR2B = TCCR2B & B11111000 | B00000001;

  // setup pin 3 for PWM output
  pinMode(3,OUTPUT);

  // setup button input
  pinMode(BUTTON, INPUT_PULLUP);

  // start RT operation
  curr_time = micros();
  next_time = curr_time + delta_time;

  // clear short history of input
  x_prev = 0;
  x_prev2 = 0;

  // set analog reference to an internally generated 1.1V
  // analogReference(INTERNAL);

  // ready!
  return;
}
```

```
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
// the loop routine runs over and over again forever:
void loop()
{
  // wait for next sample interval
  while( curr_time < next_time )
  {
    // get the current time in microseconds
    curr_time = micros();

    // handle case of wrap around with long int (once every 71 hours or so, if run continually)
    if(curr_time < 500) break;
  }

  // update the current time and the time to begin processing the next time thru the loop
  curr_time = next_time;
  next_time = curr_time + delta_time;

  // read A/D
  //  FYI, fastest rate for A/D (if continually triggered) 9615 Hz.
  x_in = analogRead(A0);

  // reduce 10 bit -> 8 bit
  // use right shift to avoid effort of division. sign extension not an issue: 0 <= x_in <= 1023
  // x_in = x_in / 4;
  x_in = x_in >> 2;

  // read the state of the pushbutton value:
  //   if button is pressed the buttonState is LOW:
  int buttonState = digitalRead(BUTTON);



  ////////////////////////////////////////////////
  // uncomment one of the four methods to compute the output PWM value

  // version 1 - output a constant value on PWM, ignoring the input
  y_out = 0;

  // version 2 - a pass thru with output = input
  // y_out = x_in;

  // version 3 - if the input is greater than 128, output high, otherwise output low voltage on
PWM
  // if(x_in < 128) y_out = 255;
  // else y_out = 0;


  /*
  // WHEN USING VERSION 4, UNCOMMENT THE analogReference(INTERNAL); LINE ABOVE, TO USE 0 <= A0 <=
1.1 V
  // version 4 - implement a simple DSP filter when button is pressed, otherwise output = input
  if(buttonState == LOW) y_out = (x_in + x_prev + x_prev2)/3;
  else y_out = x_in;

  // update signal history for simple filter
  x_prev2 = x_prev;
  x_prev = x_in;
  */



  ////////////////////////////////////////////////
  // set pwm output to current output
  analogWrite(drv_pin,y_out);


  // done with this iteration
  return;
}
```

APPENDIX 2 – Software for Storage Scope and Spectrum Analyzer

```
/*
  Storage Scope by Dr Fred DePiero

  Reads the A/D at 8000Hz and waits for a button press.
  When button is pressed, then next 500 inputs are saved and then uploaded for display
  Use with the Serial Plotter for display (Tools > Serial Plotter menu).

  This example code is in the public domain.
*/

////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
// define sample rate using delta_time
//  with delta_time =  125;  the sample rate is 8000 Hz
//  with delta_time = 1000;  the sample rate is 1000 Hz
//  the function micros returns a count in microseconds but increments by 4, every 4 microseconds
unsigned long delta_time = 125;

// select mode to compute a spectrum via DFT
bool dft_mode = false;
int DFT_LEN = 400;


////////////////////////////////////////////////////////////////////////
// an array to store the recent history of the analog input
//  captures a signal prior to upload
unsigned char *sig_cap;

// the signal plotter has a width of 500 samples
#define SIG_LEN 500

// define states associated with capture sequence
#define CAP_WAITING 1
#define CAP_ACTIVE 2
#define CAP_UPLOAD 3

int sig_cap_indx = 0;
int sig_cap_state = CAP_WAITING;



////////////////////////////////////////////////////////////////////////
// this tick period is used when displaying signals with the plotter
//  this interval is used to set the red ticks at the bottom of the plot
//  this interval is in SAMPLES. with a sample rate of 8000 Hz, 8 <-> 1 msec ticks
//  the plot has 500 samples horizontally
int tick_per = 8;


////////////////////////////////////////////////////////////////////////
//  micros count is in microseconds but increments every 4 microseconds
unsigned long curr_time = 0;
unsigned long next_time = delta_time;


// Button connects to ground on pin 12
const int BUTTON = 12;


////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
// the setup routine runs once when you press reset:
void setup() {

  // initialize serial communication at 115200 bits per second:
  Serial.begin(115200);

  // allocate array used for various purposes with analog input
  sig_cap = new unsigned char[ SIG_LEN ];
  if(sig_cap == NULL)
```

```
  {
    Serial.println("Cant allocate memory!");
  }

  // clear buffer
  for(int i=0;i<SIG_LEN;i+=1)
  {
    sig_cap[i] = 0;
  }

    // send plot info to serial plotter
    if(dft_mode == false)
    {
      Serial.print("Storage_Scope--single_trace");
      Serial.print(" ");
      Serial.print("horizontal_axis_ticks_1ms_intervals");
      Serial.print(" ");
      Serial.println("captured_signal");
    }
    else
    {
      Serial.print("Signal_spectrum");
      Serial.print(" ");
      Serial.print("horizontal_axis_DC_to_4000Hz");
      Serial.print(" ");
      Serial.println("signal_spectrum--
note_the_vertical_scale_is_uncalibrated_and_capped_at_255");
    }

  // setup button input
  pinMode(BUTTON, INPUT_PULLUP);

 // start RT operation
  curr_time = micros();
  next_time = curr_time + delta_time;

  // set analog reference to internally generated 1.1V
  analogReference(INTERNAL);

  // ready!
  return;
}


////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
// upload data to serial plotter
int send_pdata(int drv_sig,int resp_sig,int tsig_state)
{
  // send triplets of data, for 3 plotting curves
  int pmax = 300;

    Serial.print(pmax);
    Serial.print(" ");
    Serial.print(tsig_state);
    Serial.print(" ");
    Serial.println(resp_sig);

  return 0;
}



////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
// gen periodic impulses
//  this is done during non-real time processing hence computational effort doesnt matter as much
int gen_impulses(int n,int per,int amp_low,int amp_high)
{
  // find time index within period
  int n_mod = n % per;

  // amplitude logic
```

```
  if(n_mod == 0) return amp_high;

  return amp_low;
}


////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
// the loop routine runs over and over again forever:
void loop()
{
  // read the state of the pushbutton value:
  // check if the pushbutton is pressed. If it is, the buttonState is LOW:
  //  dont worry about debounce here, the first jump on the trampoline is when capture starts!
  int buttonState = digitalRead(BUTTON);
  if( (buttonState == LOW) && (sig_cap_state == CAP_WAITING) )
  {
    sig_cap_state = CAP_ACTIVE;
    sig_cap_indx = 0;
  }

  // START TIME CRITICAL SECTION

  // wait for next sample interval
  while( curr_time < next_time )
  {
    curr_time = micros();

    // handle case of wrap around (once every 71 hours or so)
    if(curr_time < 500) break;
  }
  curr_time = next_time;
  next_time = curr_time + delta_time;

  // fastest rate for A/D (if continually triggered) is 9615 Hz, FYI
  int sig = analogRead(A0);

  // reduce 10 bit -> 8 bit
  // use right shift to avoid effort of division. sign extension not an issue: 0 <= sig <= 1023
  // sig = sig / 4;
  sig = sig >> 2;

  // CLIP - data is stored as unsigned char, 0-255
  if(sig > 255) sig = 255;
  if(sig < 0) sig = 0;


  // CAPTURE data
  if(sig_cap_state == CAP_ACTIVE)
  {
    // capture signal, provided space available in buffer
    if(sig_cap_indx < SIG_LEN)
    {
      sig_cap[sig_cap_indx] = sig;
      sig_cap_indx += 1;
    }

    // if buffer full then done, switch to upload mode
    else
    {
      sig_cap_state = CAP_UPLOAD;
      sig_cap_indx = 0;
    }
  }


  // END TIME CRITICAL SECTION

  // if frame of data is complete then drop out of real time mode to upload data
  if(sig_cap_state == CAP_UPLOAD)
  {
    // transmit frame of data
    for(int i=0;i<SIG_LEN;i+=1)
```

```
    {
      // generate timing signal
      //  sample rate is 8000 Hz, 500 samples are captured (SIG_LEN)
      //  period is in samples!!! 8 <-> 1000 hz

      if(dft_mode == true) tick_per = 50;
      else tick_per = 8;

      // generate tick marks
      int tsig = gen_impulses(i,tick_per,0,20);

      // access captured signal
      int cap_sig = sig_cap[i];

      // compute DFT (for more info take EE 328 and EE 419, or Google)
      if(dft_mode == true)
      {
        int plot_i = i - 100;
        if(plot_i < 0)
        {
          cap_sig = 0;
          tsig = 0;
        }
        else if(plot_i<DFT_LEN/2)
        {
          int k = plot_i;
          double rsum = 0;
          double isum = 0;
          for(int n=0;n<DFT_LEN;n+=1)
          {
            double th = -2.0 * M_PI * k * n / DFT_LEN;
            rsum += sig_cap[n] * cos(th);
            isum += sig_cap[n] * sin(th);

            double mag = rsum * rsum + isum * isum;
            mag = sqrt(mag);

            mag = 255.0 * mag / DFT_LEN;
            if(mag > 255.0) mag = 255.0;
            if(mag < 0.0) mag = 0.0;

            cap_sig = (int) mag;
          }
        }
        else
        {
          cap_sig = 0;
          tsig = 0;
        }
      }

      // send data up to serial plotter
      send_pdata(0,cap_sig,tsig);
    }

    // following upload, chill
    delay(500);
    sig_cap_state = CAP_WAITING;

    // RE-start RT operation
    curr_time = micros();
    next_time = curr_time + delta_time;
  }

  return;
}
```

APPENDIX 3 – DSP Echo Effect

```
/*
  DSP - Echo Effect

  This example code is in the public domain.
*/


////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////

// define sample rate using delta_time
//  with delta_time =  125;  the sample rate is 8000 Hz
//  with delta_time = 1000;  the sample rate is 1000 Hz
//  the function micros returns a count in microseconds but increments by 4, every 4
microseconds
unsigned long delta_time = 125;



////////////////////////////////////////////////////////////////////////

// an array to store the recent history of the analog input
// array is used for multiple purposes
//  captures a signal prior to upload (see CAP_ONLY below)
//  saves history of input signal for echo (see RT_DSP below)
unsigned char *x_in_hist;

// need this long history of the input for the DSP echo
//  1500 is near the limit of available memory :(
#define XIN_HIST_LEN 1500

// indexes into the array for reading and writing into FIFO buffer
int xin_windx = 0;
int xin_rindx = 0;


// pwm output pin
int drv_pin = 3;    // 3,5,6,9,10,11


////////////////////////////////////////////////////////////////////////
//  micros count is in microseconds but increments every 4 microseconds
// unsigned long curr_time = millis();
unsigned long curr_time = 0;
unsigned long next_time = delta_time;

// a short history of the input, for pre-filtering
//  filtering controlled by filter_mode (above)
int y_out = 0;
int x_in = 0;
int x_prev = 0;
int x_prev2 = 0;

// Button connects to ground on pin 12
const int BUTTON = 12;



////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
// the setup routine runs once when you press reset:
void setup() {

  // initialize serial communication at 115200 bits per second:
  Serial.begin(115200);

  // set PWM frequency to 31372.55 Hz
  TCCR2B = TCCR2B & B11111000 | B00000001;

  // allocate array used for various purposes with analog input
  x_in_hist = new unsigned char[ XIN_HIST_LEN ];
  if(x_in_hist == NULL)
  {
    // a gasping call for help - probably wont be heard...!!!
    Serial.println("Cant allocate memory!");
    return;
  }

  // clear buffer
  for(int i=0;i<XIN_HIST_LEN;i+=1)
  {
    x_in_hist[i] = 0;
  }

  // set offset for reading and writing into buffer, for echo effect
  xin_windx = XIN_HIST_LEN - 2;
  xin_rindx = 0;

  // setup pin 3 for PWM output
  pinMode(3,OUTPUT);

  // setup button input
  pinMode(BUTTON, INPUT_PULLUP);

 // start RT operation
  curr_time = micros();
  next_time = curr_time + delta_time;

  // time index for PWM output drive signal
  // sig_drv_indx = 0;

  // clear short history of input
  x_prev = 0;
  x_prev2 = 0;

  // set analog reference to internally generated 1.1V
  analogReference(INTERNAL);

  // ready!
  return;
}



////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
// the loop routine runs over and over again forever:
void loop()
{
  // START TIME CRITICAL SECTION

  // wait for next sample interval
```

```
  while( curr_time < next_time )
  {
    // get the current time in microseconds
    curr_time = micros();

    // handle case of wrap around (once every 71 hours or so, if run continually)
    if(curr_time < 500) break;
  }

  // update the current time and the time to begin processing the next time thru the
loop
  curr_time = next_time;
  next_time = curr_time + delta_time;

  // read A/D
  //  FYI, fastest rate for A/D (if continually triggered) 9615 Hz.
  x_in = analogRead(A0);

  // reduce 10 bit -> 8 bit
  // use right shift to avoid effort of division. sign extension not an issue: 0 <=
x_in <= 1023
  // x_in = x_in / 4;
  x_in = x_in >> 2;


  // read the state of the pushbutton value:
  //   if button is pressed the buttonState is LOW:
  int buttonState = digitalRead(BUTTON);


  ////////////////////////////////////////////////
  // find output value for this time thru loop

  // default output is simply the input
  y_out = x_in;



  // do echo effect
  if(buttonState == LOW)
  {
    // write current input into FIFO
    x_in_hist[xin_windx] = x_in;

    // read older input from FIFO input
    int x_in_older = x_in_hist[xin_rindx];

    y_out = (x_in + x_in_older)/2;

    // advance read and write indecies
    xin_rindx += 1;
    if(xin_rindx >= XIN_HIST_LEN) xin_rindx = 0;

    xin_windx += 1;
    if(xin_windx >= XIN_HIST_LEN) xin_windx = 0;
  }


  // do DSP
  // if(buttonState == LOW) y_out = (x_in + x_prev + x_prev2)/3;

  // update signal history for simple filtering
  // x_prev2 = x_prev;
  // x_prev = x_in;
```

```
  // set pwm output to current output
  analogWrite(drv_pin,y_out);

  // END TIME CRITICAL SECTION

  return;
}
```