Audio Volume Unit Meter
Firmware Readiness Report

By Team 10:
Ethan Nagelvoort 821234668
Robert Esposito 822379994
Carlos A. Callejas Dominguez 821226920
Ardiana Krasniqi 820354659

Executive Summary:

The purpose of this document is to describe the firmware design for the Audio Volume Unit Meter. The firmware for this project uses an Arduino Due to read seven bandpass filters used to filter an audio signal into different frequency bins and multiplex seven shift registers, which each drive eight LEDs based on the strength of the audio signal in each bin. This report will describe how we leveraged the hardware and software features of the Arduino Due to achieve the firmware requirements of the Audio Volume Unit Meter. These features include the Analog-to-Digital Converter (ADC), timers, interrupt service routines, and digital output pins. This report will show that the firmware is ready to be implemented into the overall Audio Volume Unit Meter. The biggest challenge when developing this firmware were the timing requirements, such as the sample rate of our ADC and multiplexing our LEDs using shift registers. We were able to overcome this challenge by creating prototypes for each part of our firmware to test its functionality. This report includes a description of the system process, a state diagram of the system, and test cases to validate that our system functions correctly.

System Process Description:

Our system operates based on the following system diagram:

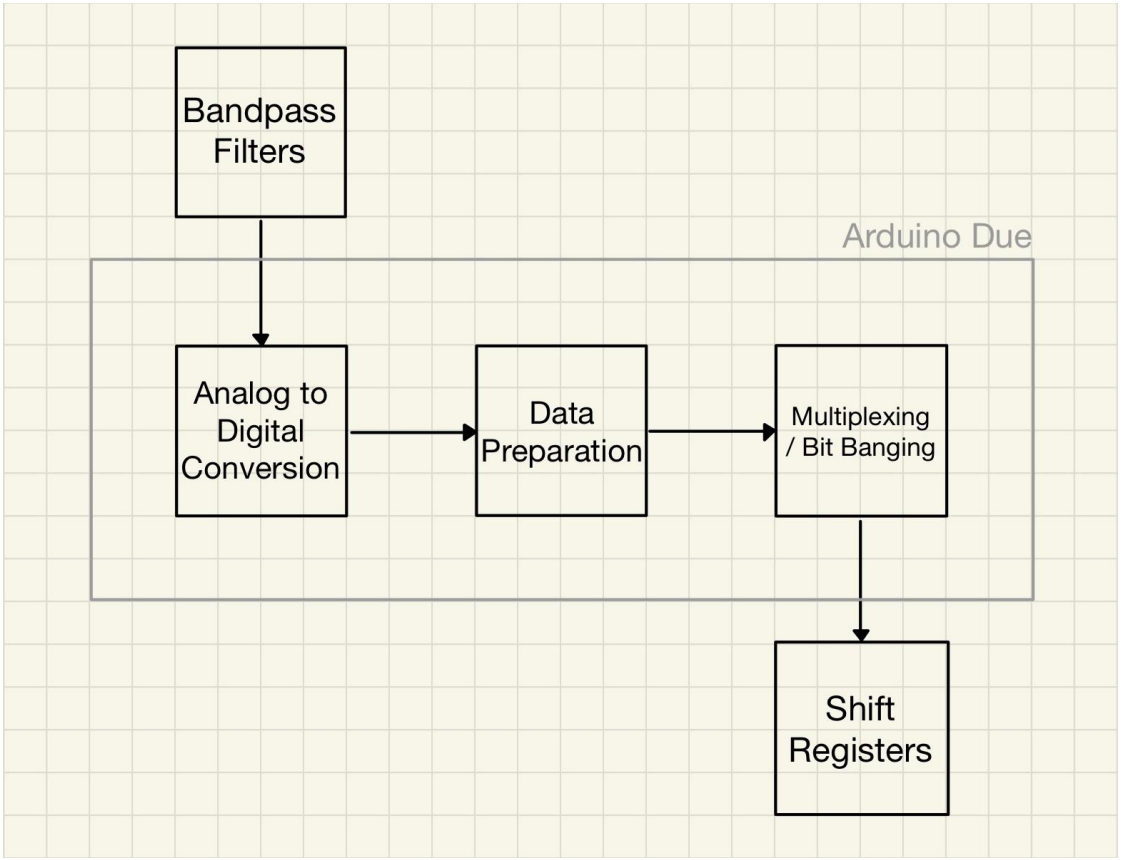

Figure 1: System Diagram

The Arduino Due will have its analog pins, pins 0 through 6, attached to the different bandpass filters that represent the seven frequency bins that the VU Meter detects. The ADC on the Arduino Due is set to freerun mode, which will constantly read these seven analog pins. The ADC values are sampled at a rate of 10kHz and are placed into an array. This array gets sent into a for loop where each ADC value of the array goes through an if-else statement. This if-else statement determines how many LEDs turn on for the corresponding frequency bins by comparing the values read by the ADC to the quanta that lies at the center between each -3 dB cutoff. We determined these quanta by first calculating the dB value of all the possible inputs of our 10 bit ADC using the formula: $dB = 20log(\frac{quanta}{1023})$.Using this list of dB values, we determined the quanta that corresponded with the closest value to the center of our -3 dB cutoffs. The figure below shows the cutoff values used and how many LEDs turn on based on the cutoffs:

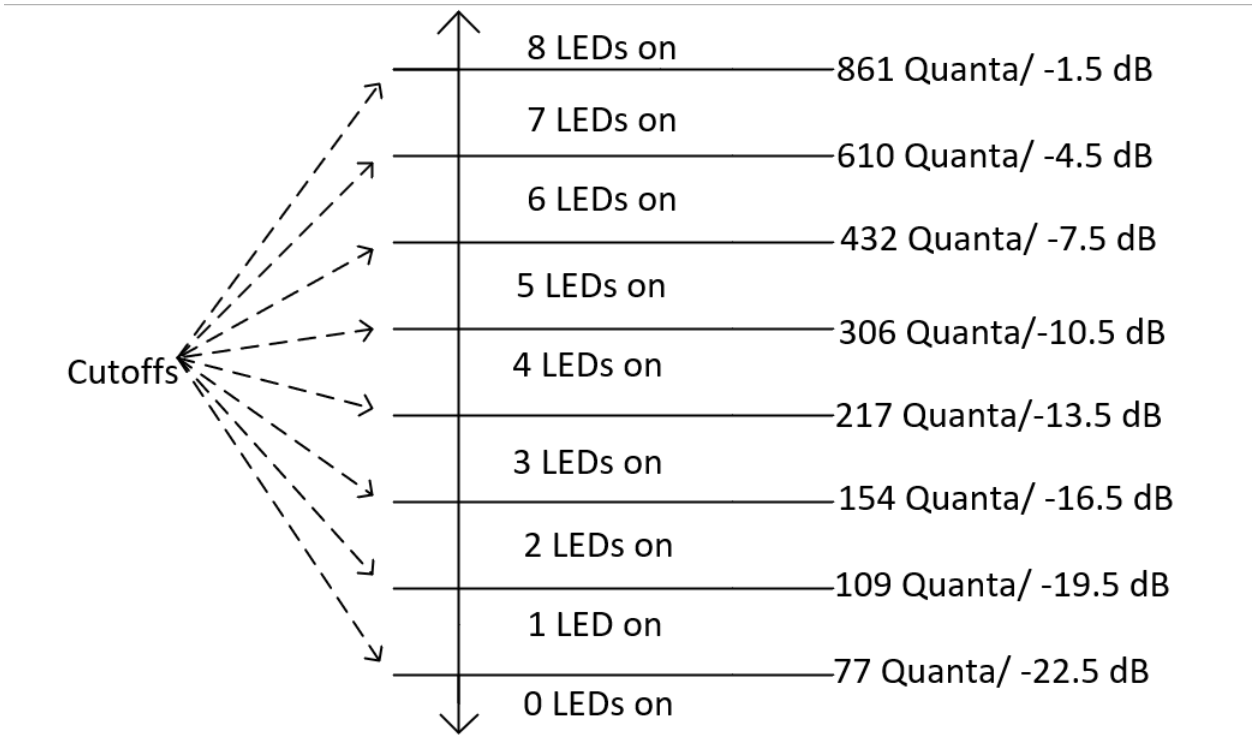

Figure 2: ADC/dB cutoffs showing when LEDs turn on

The number of LEDs that are supposed to turn on will be stored in global variable n and then this variable is used in the formula, 0xFF >> (8 - n). This formula is used to determine the number of 1's needed to turn on *n* LEDs. For example, if *n = 5*, then 0xFF would be shifted to the right by 3, resulting in 0x1F (or 00011111 in binary). These values are then stored in a global array called *data*.

The LEDs are multiplexed using shift registers through a process called bit banging, which is achieved by toggling an output pin using a timer to generate a clock signal. The interrupt service routine for this timer operates based on the following timing diagram.

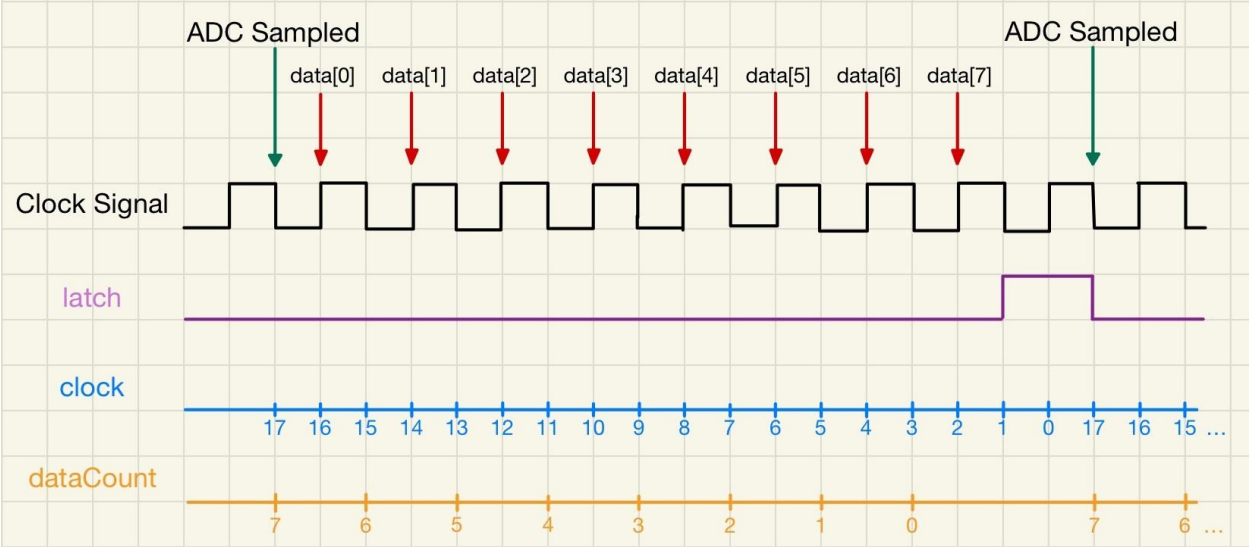

Figure 3: Timing Diagram

Eight clock cycles are needed to get the values in the array *data* into our shift registers (one bit per clock cycle), along with an additional clock cycle needed to latch the registers and drive the LEDs. The interrupt generates this clock signal by toggling one of our digital output pins between high and low, which represent the positive and negative edges of the clock. The interrupt is triggered eighteen times per sample period, 16 to get the values from *data* into the shift registers, and 2 in order to latch the shift registers. On the negative edge of the clock, each bit is isolated from the seven other bits by ANDing the current bit with a 1 and the rest with 0, then shifting until the current bit is the LSB. Thus, the result of this operation will either be a 00000001 or a 00000000, which represents the next bit that will go into the shift registers, 1 or 0. The current bit is then sent to the shift register by setting the output pins to either high (1) or low (0). Then on the positive edge, the bit is sent to the first flip-flop of the shift register, and each of the previous bits are shifted forward accordingly. Once all of the data is in the shift register, the shift register is then latched using another output pin and the LEDs turn on. The variable *clock* is reset and the ADC is sampled again. The entire system process can be seen in the following state diagram.

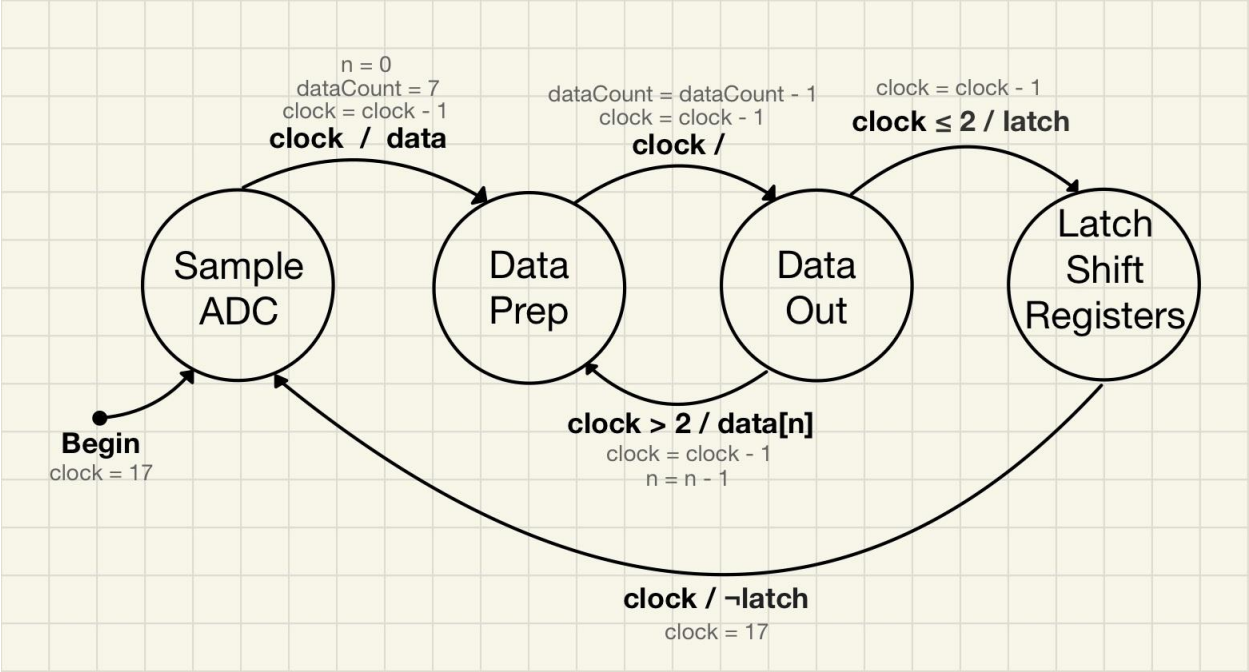

Figure 4: State Diagram of System

Validations:

Test	Description and Coverage	Procedure	Results
1	The ADC was tested by connecting a 10K potentiometer to one analog pin and a photoresistor to another analog pin. The expected result of this test was to read ADC values simultaneously on both pins. This would verify that our sampling rate is valid.	The ADC was set to freerunning mode. Then the ADC readings were viewed through the serial monitor. Then the 10K potentiometer was turned and a hand was placed over the photoresistor.	As the 10K potentiometer was turned and the photoresistor was covered, the ADC values that were read on the serial monitor were changed.

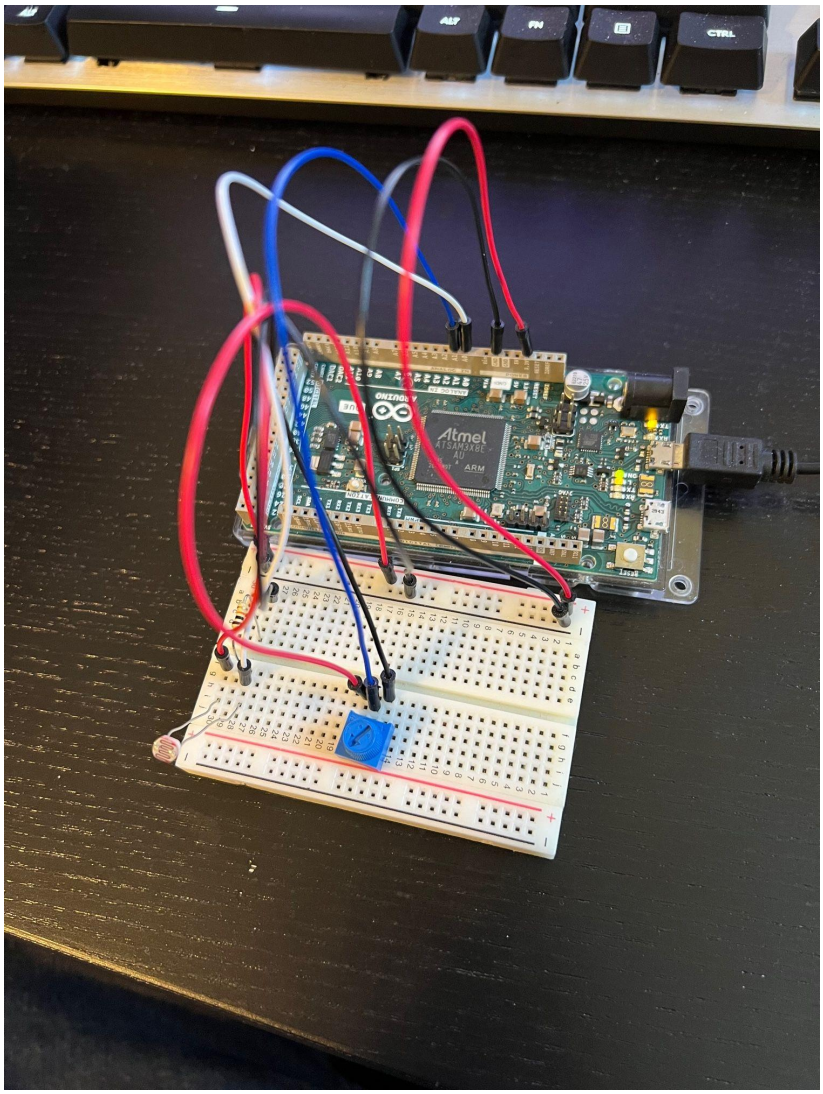

Figure 5: Photoresistor and Potentiometer connected to ADC

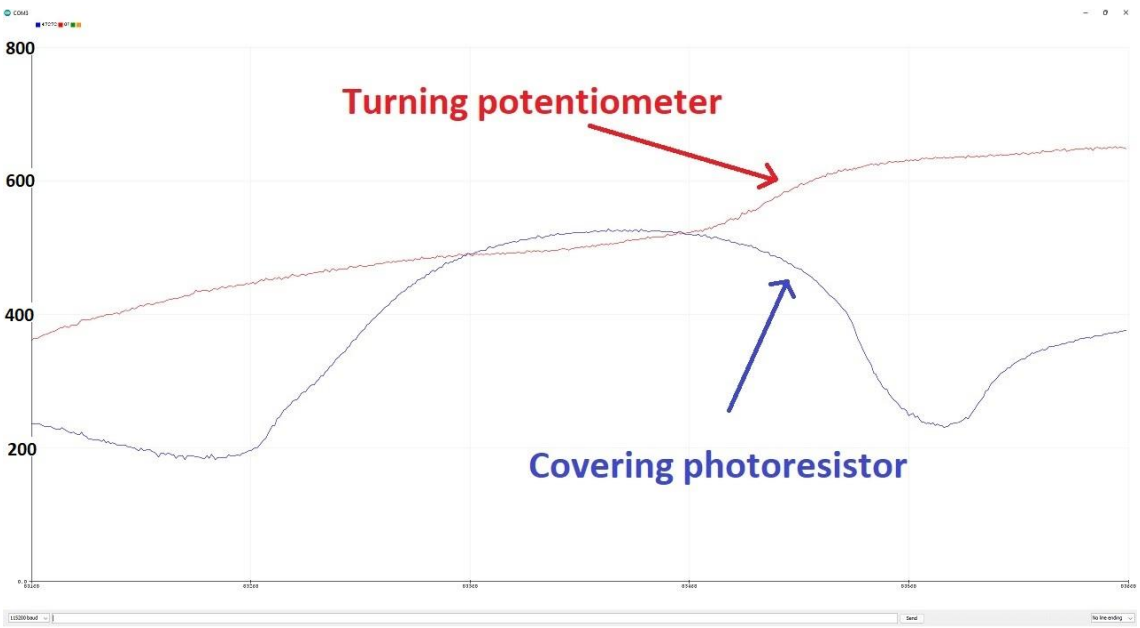

Figure 6: Serial plot of the two ADC readings

Test	Description and Coverage	Procedure	Results
2	The bit banging was tested by having the Arduino Due hooked up to a photoresistor and an LED test board. The test board holds 64 LEDs in a 8 by 8 configuration with eight shift registers. This represents the LED configuration that will be similar to the VU Meters. The expected result for this test was that the amount of LEDs that turn on will change based on the amount of light entering the photoresistor. This would verify the bit banging was working as intended	One photoresistor was connected to all seven of the Arduino Due's analog pins. The photoresistor was then covered by a hand and lit by a flashlight to change the values being read by the ADC. This also ensured the shift registers were able to drive the LEDs and wouldn't flicker.	In the ambient room light, 6 LEDs were on which was used as a baseline for the test. When the photoresistor was covered, less LEDs turned on depending on how much of our hand we put over it. When it was lit by a flashlight, all of the LEDs turned on. The LEDs did not flicker during this test, thus confirming our bit banging was working as expected.

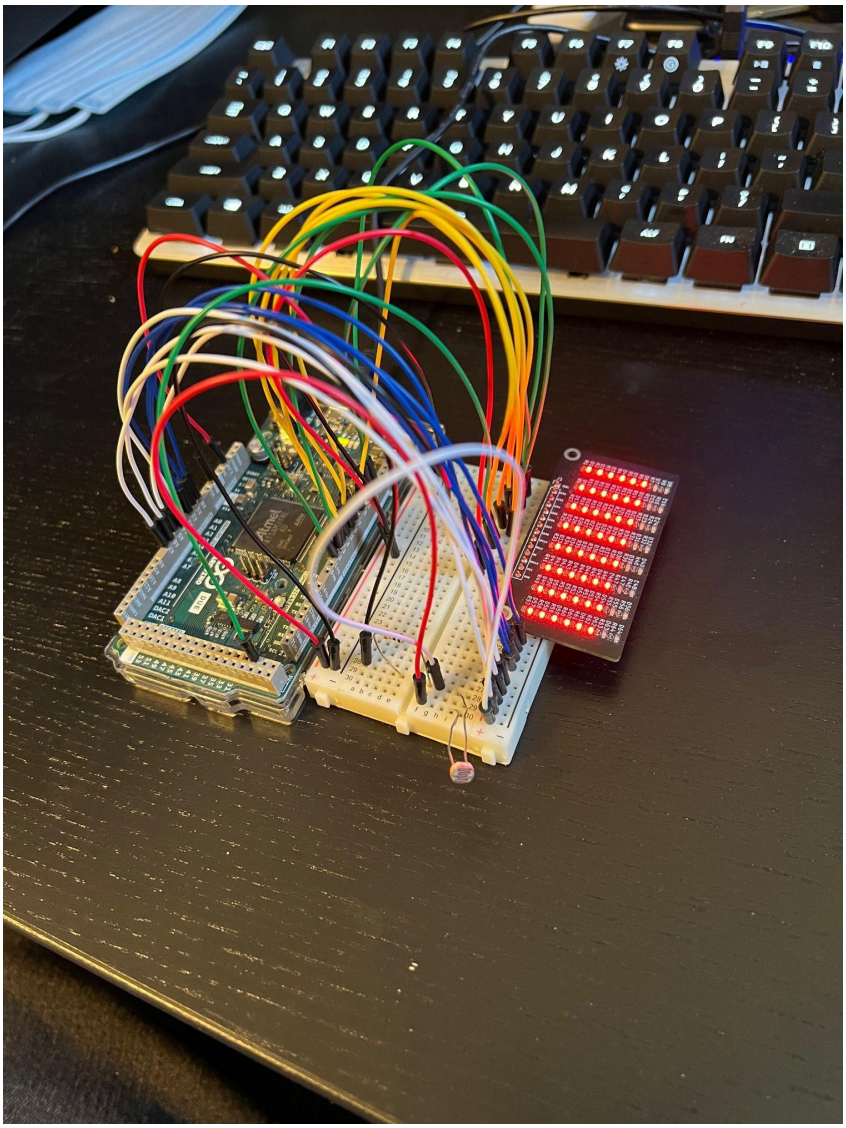

Figure 7: Photoresistor in ambient room light

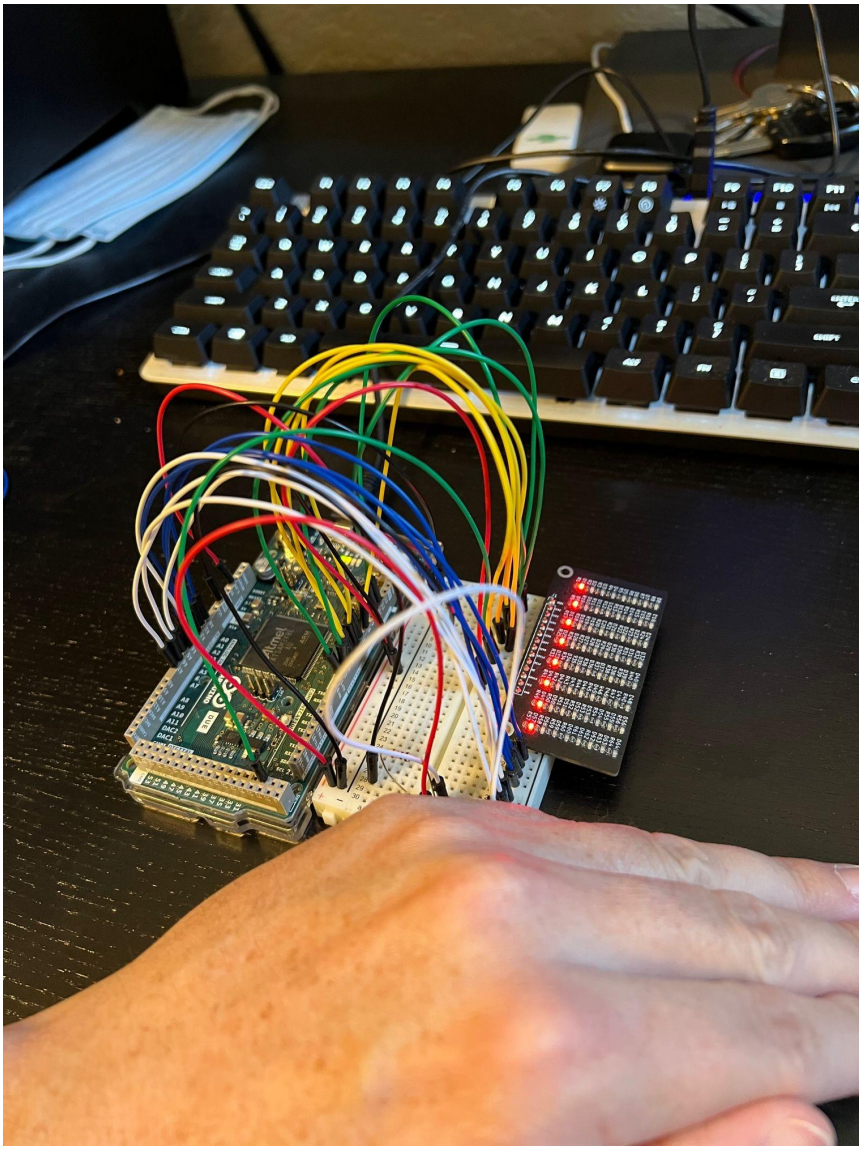

Figure 8: Photoresistor covered by hand

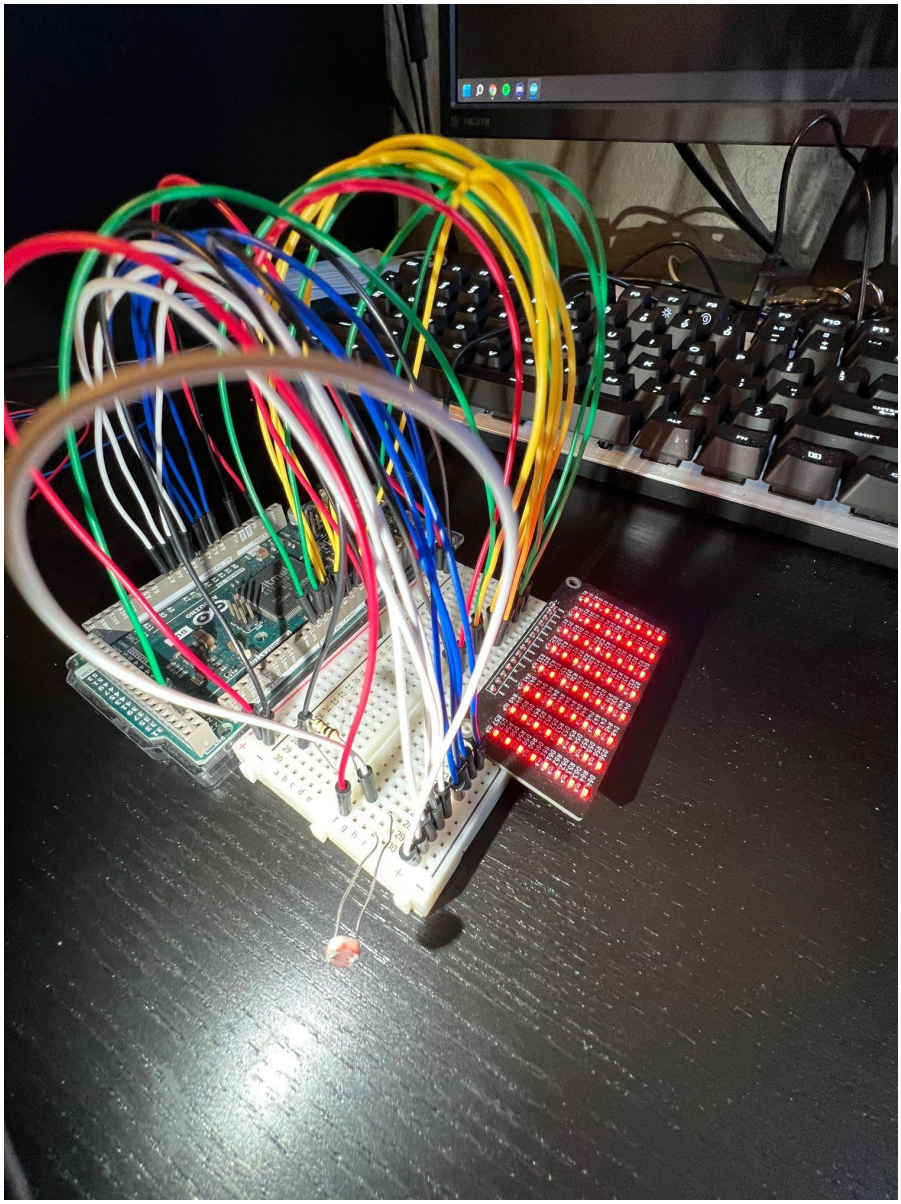

Figure 9: Photoresistor lit by flashlight

Test	Description and Coverage	Procedure	Results
3	The ADC cutoffs were tested by having the Arduino Due hooked up to the same test board. Then ADC values were manually placed into the code to replicate what the bandpass filters would output into the Arduino Due. The expected result of this test was that the correct amount of LEDs would turn on based on the cutoffs. This would verify that our cutoffs worked correctly.	Three different sets of ADC values were manually placed into the code. One to test right before the cutoffs (76, 108, 153, 216, 305, 431, 609, 860), at the cutoffs (77, 109, 154, 217, 306, 432, 610, 861), and right after the cutoffs (78, 110, 155, 218, 307, 433, 611, 862). Then we verified if the correct amount of LEDs turned on.	The set of ADC values used to test right after the cutoffs and right at the cutoffs both lit up the LEDs in a staircase like configuration. The first column of LEDs had 1 LED on and the second column had 2 LEDs on, etc. Each column of LEDs had the right amount of LEDs turned on for the ADC values entered. The set of ADC values used to test right before the cutoffs also had the LEDs light up in a staircase like configuration but the number of LEDs that lit in the columns decreased by one. Hence, the first column now has 0 LEDs on and the second column would now have 1 LED on, etc.

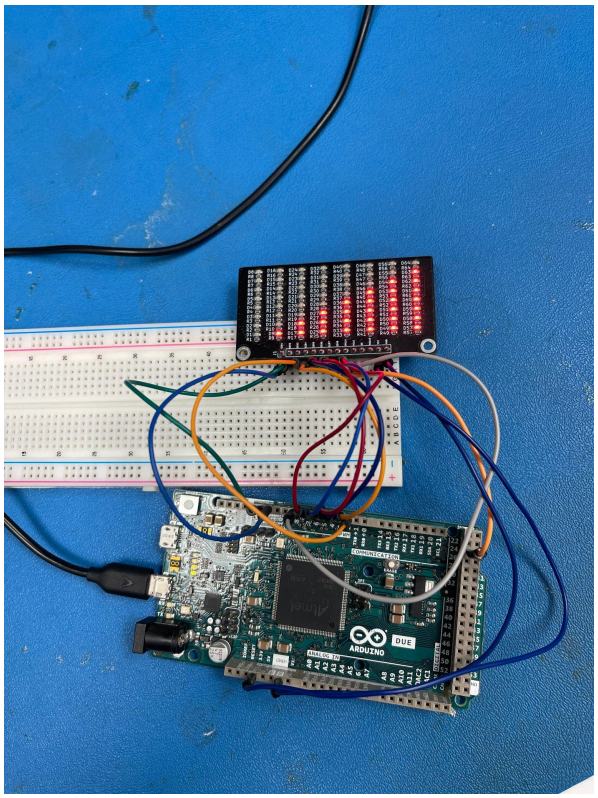

Figure 10: ADC values right before the cutoffs

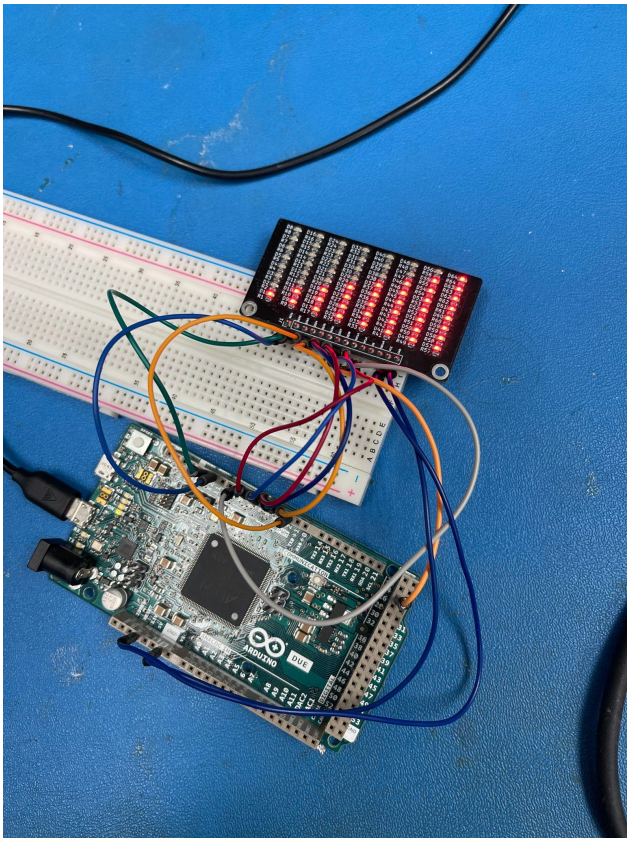

Figure 11: ADC values right at the cutoffs

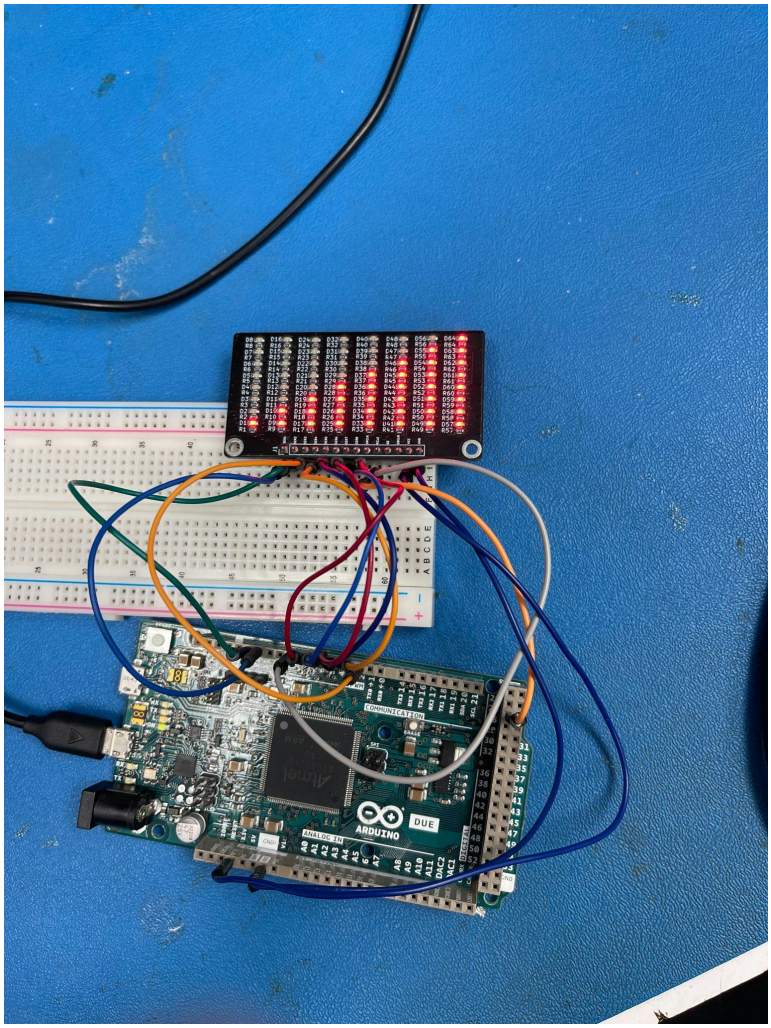

Figure 12: ADC values right after the cutoffs

Test	Description and Coverage	Procedure	Results
4	The sample rate was tested by connecting an oscilloscope to the output pin of the generated clock frequency. The expected result is nine clock cycles in a 10 kHz sample rate. This would verify that our code operates at the required sample rate.	Connect an oscilloscope to the clock output pin and then run the code. Capture the timing through the oscilloscope.	After capturing the timing, we got a frequency of 90 KHz. That frequency is equal to 9 clock cycles per sample period.

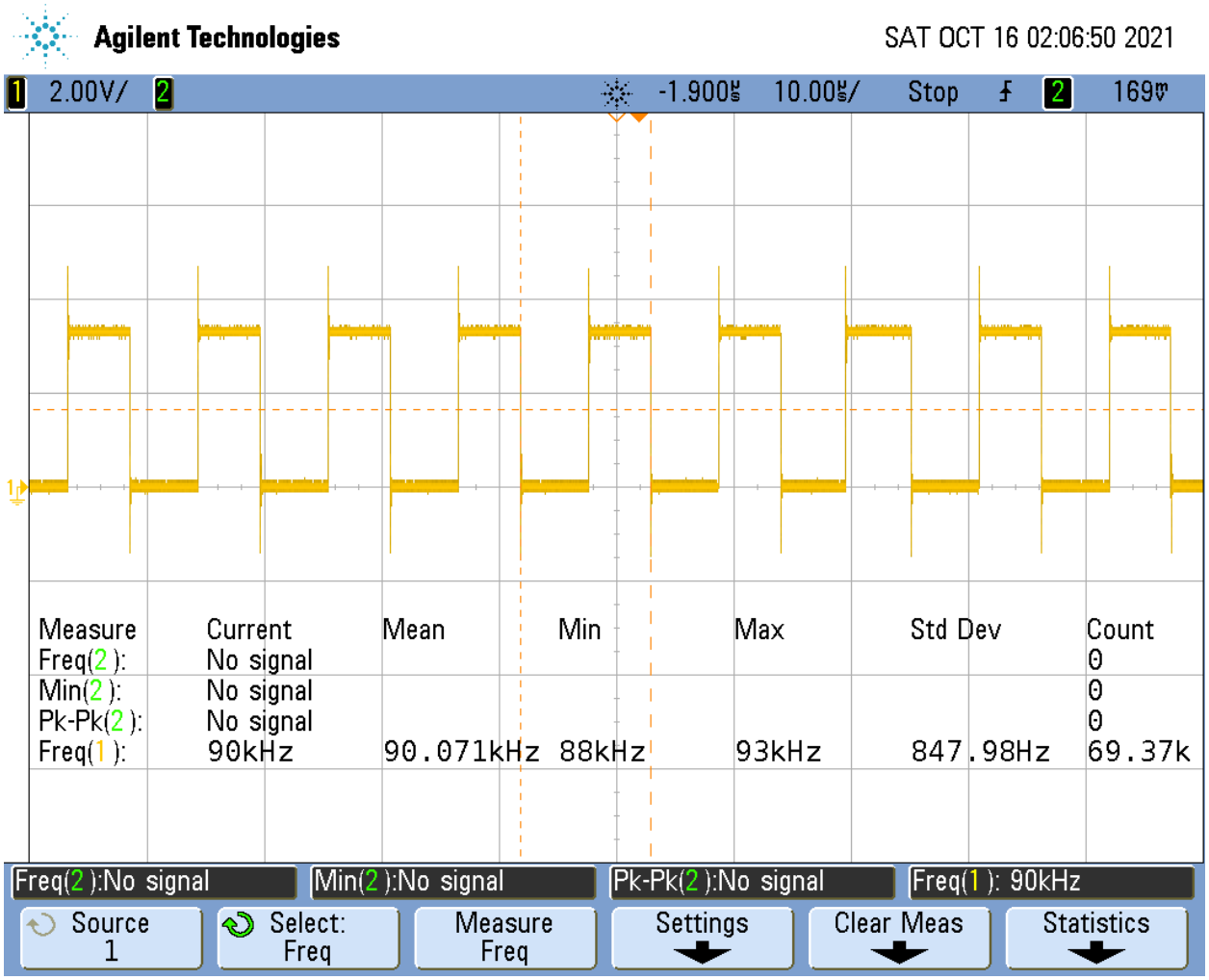

Figure 13: Oscilloscope showing 90kHz clock signal

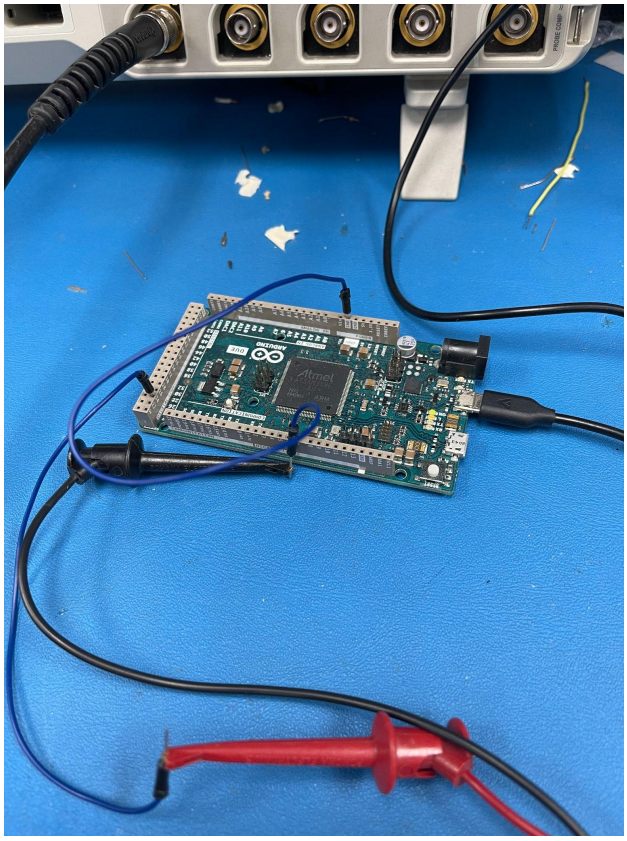

Figure 14: Oscilloscope connected to clock pin

Firmware Code:

```
/*
 * FILENAME : VUMeterFirmware.ino
 * DATE     : June 7th, 2021
 *
 * AUTHOR 1 : Robert Esposito
 * AUTHOR 2 : Ethan Nagelvoort
 * COURSE   : EE/CompE 496B - Senior Design B
 * TEAM     : Team 10 - VU Meter
 *
 * DESCRIPTION :
 *    This program will read from the Arduino Due's ADC and then
 *    bit bang the results to shift registers.
 *
 * LAST MODIFIED : 10/15/2021
 *
 */

volatile int clockCount = 17;
volatile int dataCount = 7;
int a[8];
uint8_t n = 0;
uint8_t data[8] = {0,0,0,0,0,0,0,0};
uint8_t temp[8];

void setup(){
 /**************** ADC SETTINGS ****************/
 ADC->ADC_MR |= ADC_MR_FREERUN | ADC_MR_LOWRES;
 ADC->ADC_CR = ADC_CR_START;
 ADC->ADC_CHER = ADC_CHER_CH7 | ADC_CHER_CH6 | ADC_CHER_CH5 | ADC_CHER_CH4 | ADC_CHER_CH3 | ADC_CHER_CH2 |
ADC_CHER_CH1 | ADC_CHER_CH0;

 /*************** OUTPUT PIN SETTINGS ****************/
 //PA15 (Pin 24) = CLOCK.
 PIOA->PIO_OER |= PIO_OER_P15;
 PIOA->PIO_OWER |= PIO_OWER_P15;

 //PC22 (Pin 8) = LATCH.
 PIOC->PIO_OER |= PIO_OER_P22;
 PIOC->PIO_OWER |= PIO_OWER_P22;

 //PD7 (Pin 11) = Shift Register 0.
 PIOD->PIO_OER |= PIO_OER_P7;
 PIOD->PIO_OWER |= PIO_OWER_P7;

 //PD8 (Pin 12) = Shift Register 1.
 PIOD->PIO_OER |= PIO_OER_P8;
 PIOD->PIO_OWER |= PIO_OWER_P8;

 //PB25 (Pin 2) = Shift Register 2.
 PIOB->PIO_OER |= PIO_OER_P25;
 PIOB->PIO_OWER |= PIO_OWER_P25;

 //PC28 (Pin 3) = Shift Register 3.
 PIOC->PIO_OER |= PIO_OER_P28;
 PIOC->PIO_OWER |= PIO_OWER_P28;
```

```
  //PC26 (Pin 4) = Shift Register 4.
  PIOC->PIO_OER |= PIO_OER_P26;
  PIOC->PIO_OWER |= PIO_OWER_P26;

  //PC25 (Pin 5) = Shift Register 5.
  PIOC->PIO_OER |= PIO_OER_P25;
  PIOC->PIO_OWER |= PIO_OWER_P25;

  //PC24 (Pin 6) = Shift Register 6.
  PIOC->PIO_OER |= PIO_OER_P24;
  PIOC->PIO_OWER |= PIO_OWER_P24;

  //PC23 (Pin 7) = Shift Register 7.
  PIOC->PIO_OER |= PIO_OER_P23;
  PIOC->PIO_OWER |= PIO_OWER_P23;

  /*************** TIMER AND INTERRUPT SETTINGS ***************/
  PMC->PMC_PCR = PMC_PCR_EN | PMC_PCR_CMD | (ID_TC0 & 0x7F);
  TC0->TC_CHANNEL[0].TC_CMR = TC_CMR_WAVE | TC_CMR_WAVSEL_UP_RC | TC_CMR_TCCLKS_TIMER_CLOCK1;
  TC0->TC_CHANNEL[0].TC_RC = 233;
  TC0->TC_CHANNEL[0].TC_IER = TC_IER_CPCS;
  NVIC_EnableIRQ(TC0_IRQn);
  TC0->TC_CHANNEL[0].TC_CCR = TC_CCR_CLKEN | TC_CCR_SWTRG;

  /*************** CLOCK START ***************/
  PIOA->PIO_ODSR |= PIO_ODSR_P15;
}

void loop(){
  if (clockCount < 0){
    clockCount = 17;
  }

  /*************** SAMPLE ADC ***************/
  if (dataCount < 0){
    dataCount = 7;
    a[0]=ADC->ADC_CDR[7];   // a[0] (analog pin 0) = sub bass filter.
    a[1]=ADC->ADC_CDR[6];   // a[1] (analog pin 1) = bass filter.
    a[2]=ADC->ADC_CDR[5];   // a[2] (analog pin 2) = lower midrange filter.
    a[3]=ADC->ADC_CDR[4];   // a[3] (analog pin 3) = midrange filter.
    a[4]=ADC->ADC_CDR[3];   // a[4] (analog pin 4) = upper midrange filter.
    a[5]=ADC->ADC_CDR[2];   // a[5] (analog pin 5) = presence filter.
    a[6]=ADC->ADC_CDR[1];   // a[6] (analog pin 6) = brilliance filter.
    a[7]=ADC->ADC_CDR[0];   // a[7] (analog pin 7) = entire frequency filter.

    for(int i = 0; i < 8; i++){
      if(a[i] < 77) {
        n=0;                      //0 LEDs.
      }
      else if(a[i] < 109){
        n=1;                      //1 LEDs.
      }
      else if(a[i] < 154){
        n=2;                      //2 LEDs.
      }
      else if(a[i] < 217){
        n=3;                      //3 LEDs.
```

```
    }
    else if(a[i] < 306){
      n=4;                    //4 LEDs.
    }
    else if(a[i] < 432){
      n=5;                    //5 LEDs.
    }
    else if(a[i] < 610){
      n=6;                    //6 LEDs.
    }
    else if(a[i] < 861){
      n=7;                    //7 LEDs.
    }
    else{
      n=8;                    //8 LEDs.
    }
    data[i] = 0xFF >> 8 - n;
   }
  }
}

/**************** INTERRUPT SERVICE ROUTINE ****************/
void TC0_Handler(){

 TC0->TC_CHANNEL[0].TC_SR;
 PIOA->PIO_ODSR ^= PIO_ODSR_P15;

 if (clockCount > 2){
   PIOC->PIO_ODSR &= ~PIO_ODSR_P22;

   if ((clockCount & 0x01) != 0){
     for (int i = 0; i < 8; i++){
       temp[i] = data[i] & (0x01 << dataCount);
       temp[i] >>= dataCount;
     }
     dataCount--;

     //Shift Register 0.
     if (temp[0] == 0x01){
       PIOD->PIO_ODSR |= PIO_ODSR_P7;
     }
     else{
       PIOD->PIO_ODSR &= ~PIO_ODSR_P7;
     }

     //Shift Register 1.
     if (temp[1] == 0x01){
       PIOD->PIO_ODSR |= PIO_ODSR_P8;
     }
     else{
       PIOD->PIO_ODSR &= ~PIO_ODSR_P8;
     }

     //Shift Register 2.
     if (temp[2] == 0x01){
       PIOB->PIO_ODSR |= PIO_ODSR_P25;
     }
```

```
    else{
     PIOB->PIO_ODSR &= ~PIO_ODSR_P25;
    }

    //Shift Register 3.
    if (temp[3] == 0x01){
     PIOC->PIO_ODSR |= PIO_ODSR_P28;
    }
    else{
     PIOC->PIO_ODSR &= ~PIO_ODSR_P28;
    }

    //Shift Register 4.
    if (temp[4] == 0x01){
     PIOC->PIO_ODSR |= PIO_ODSR_P26;
    }
    else{
     PIOC->PIO_ODSR &= ~PIO_ODSR_P26;
    }

    //Shift Register 5.
    if (temp[5] == 0x01){
     PIOC->PIO_ODSR |= PIO_ODSR_P25;
    }
    else{
     PIOC->PIO_ODSR &= ~PIO_ODSR_P25;
    }

    //Shift Register 6.
    if (temp[6] == 0x01){
     PIOC->PIO_ODSR |= PIO_ODSR_P24;
    }
    else{
     PIOC->PIO_ODSR &= ~PIO_ODSR_P24;
    }

    //Shift Register 7.
    if (temp[7] == 0x01){
     PIOC->PIO_ODSR |= PIO_ODSR_P23;
    }
    else{
     PIOC->PIO_ODSR &= ~PIO_ODSR_P23;
    }
   }
  }
  else{
   PIOC->PIO_ODSR |= PIO_ODSR_P22;
  }
  clockCount--;
}
```