Final Project Report: dbFS Audio Meter

ENGR 478: Design with Microcontrollers

Electrical Engineering Department

San Francisco State University

Spring, 2020

By Moshe Stanylov

mstanylov@mail.sfsu.edu

918669614

Introduction

In this project I've implemented a dbFS (decibel Full Scale) meter with the TM4C123 microcontroller. This type of meter is commonly used in digital audio interfaces and DAWs (Digital Audio Workstation) to make sure the signal is not clipping when sampled.
Clipping occurs when the analog signal exceeds the range of the ADC (Analog to Digital Converter). In the case of the TM4C123 the maximum range is $3.3v$, If the incoming analog signal is equal or above $3.3v$ it will be sampled as 4095 in the case of a 12bit ADC.

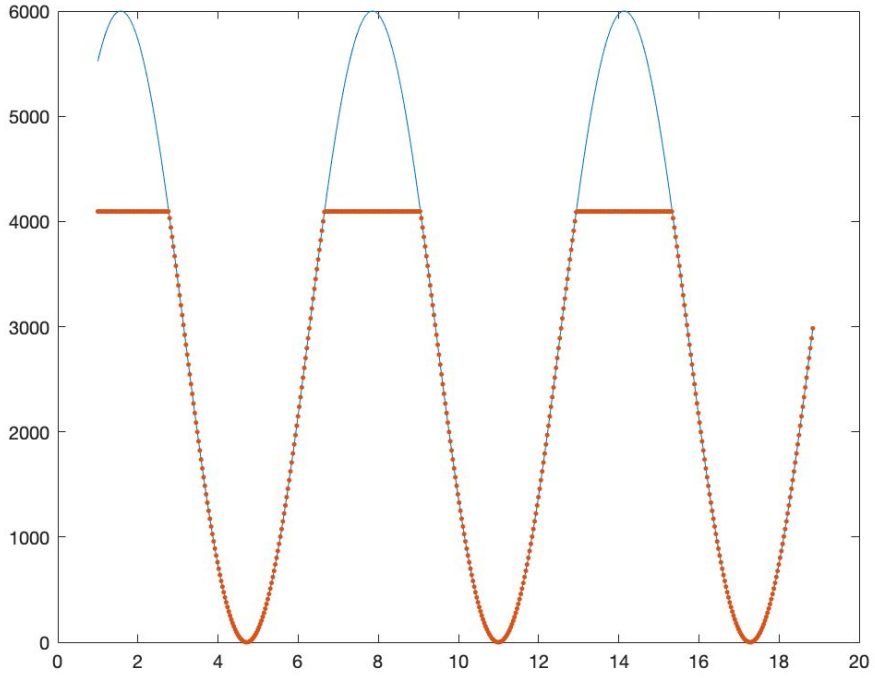

Fig.1 - clipped signal (orange) relative to input signal (blue)

In order to avoid clipping we need to monitor the sampled signal and make sure the input is within the range of the ADC. With digital audio signals we usually use a dbFS meter to achieve that. dbFS is a standard used to measure the magnitude of the signal relative to the range of the ADC. The maximum value on a dbFS meter is 0db and it corresponds with the maximum value the ADC can produce. In the case of the TM4C123, If the sample is 4095 the meter will show 0db, if it's below 4095 the meter will show the gain in db between the sample and the full scale of the ADC. We use the formula $20log(sample/maximum)$ to get the db value and round according to the meter resolution.

For example if the input is $1.7v$ it will be sampled as 2109, we can get the dbFS reading by using the formula $20log(2109/4095) = -5.76db$. the reading can then be rounded to fit the meter resolution.

Fig.2 - dbFS meter reading of $-5.8db$ in a digital audio workstation

Another important feature of the dbFS meter is the clipping indicator. If the reading on the meter was 0db even for a slight moment it is very likely the signal was clipped, but it is not always possible to notice the brief peak on the meter. There needs to be an obvious indication that the signal hit the ceiling. This is where the clipping indicator comes in, the indicator light will turn on if the signal hits 0db and will remain on until the user resets it.

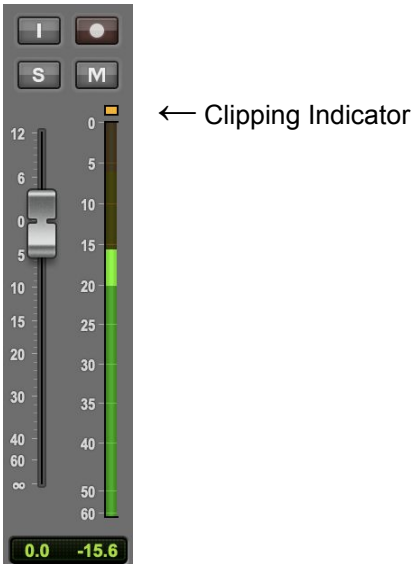

Fig.3 - clipping indicator is on meaning the signal was clipped previous to the -15.6 reading

The goal of this project is to implement a dbFS meter using the TM4C123 microcontroller and an array of 12 LED's. Such meters are used on professional audio interfaces to make sure the signal on a specific audio input channel is not clipping when sampled.

Fig.4 - dbFS meters for input channels 1 and 2 on the UAD apollo audio interface

Hardware Design

The system receives an instrument level signal, this can be from a guitar pickup, microphone etc. The preamp module amplifies the signal by a fixed gain and features a volume control. After the amplification, the signal is passed to the ADC where it is sampled and converted to a 12 bit digital number. The LED array is used as a dbFS meter with a $3db$ resolution and a clipping indicator. The LEDs are lit based on the software implementation on the TM4C123 microcontroller. The onboard switch SW2 is used as a reset button for the clipping indicator.

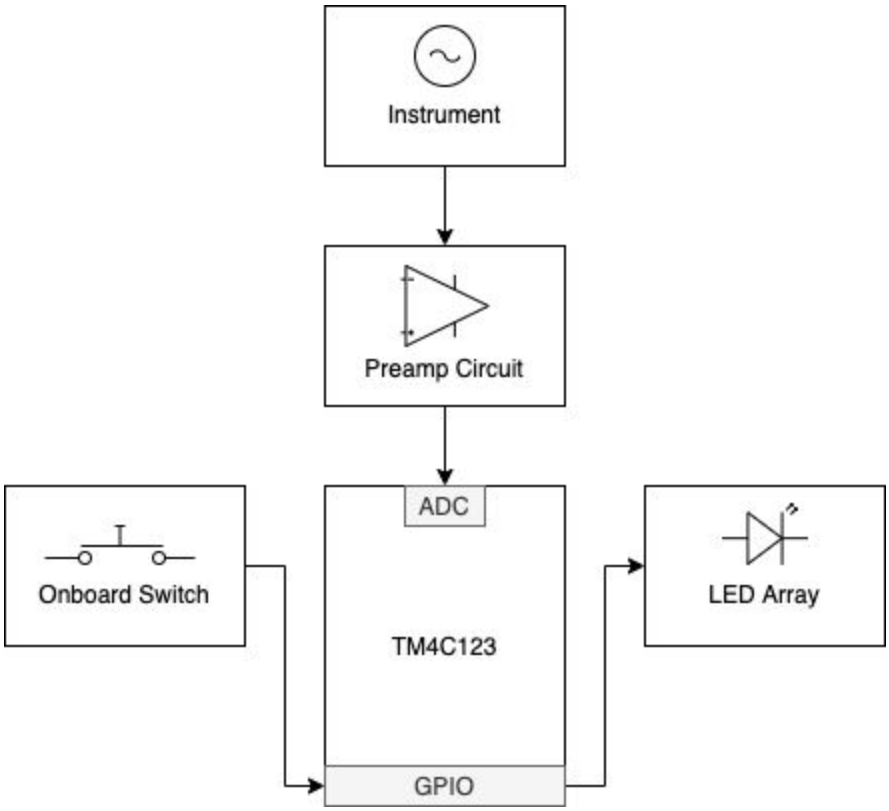

Fig.5 - system overview

Preamplifier

The preamp circuit is using the LM741 operational amplifier in a non inverting configuration with a fixed gain of $25.6v/v$. I've decided to use this gain factor since the output from a guitar is usually in the $100 \sim 200\ mv$ range, and a 25.6 gain factor allowed me to get close enough to $3.3v$ with the resistors I had available.

I've used a potentiometer between the opamp output and the ADC pin in order to control the amplitude of the signal before it is sampled by the ADC.

For the supply rails of the LM741 I've used two $9v$ batteries, one of the batteries is set up in a flipped polarity to create the negative voltage required by one of the supply rails. The ground connections of the batteries were connected through a switch so the amp can be turned on and off. A schematic of the amplifier can be seen in figure 6.

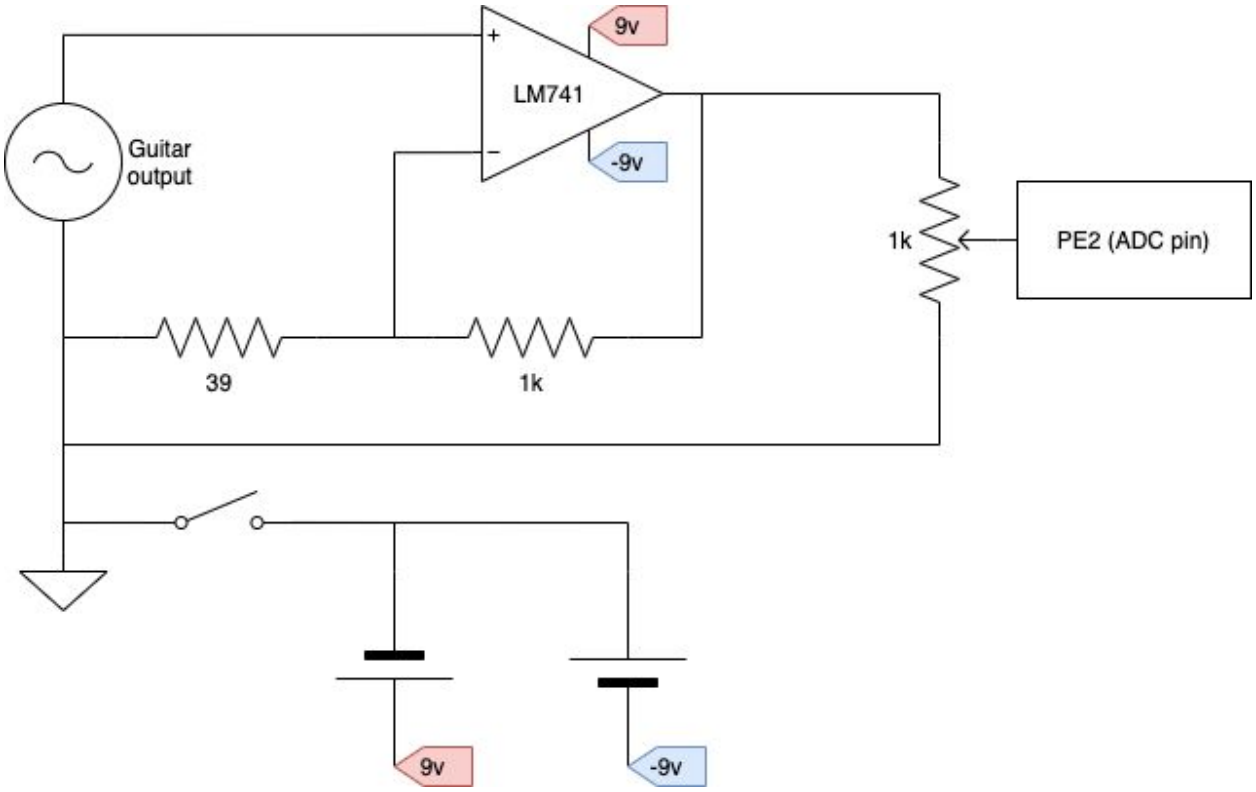

Fig.6 - amplifier circuit

LED Array

I've used an array of 12 LEDs to serve as the dbFS meter. Each LED was connected with a 330Ω resistor in series to limit the current draw.

The top LED in the array represents $0db$ and serves as the clipping indicator, each LED below the clipping indicator represents a $-3db$ increment.

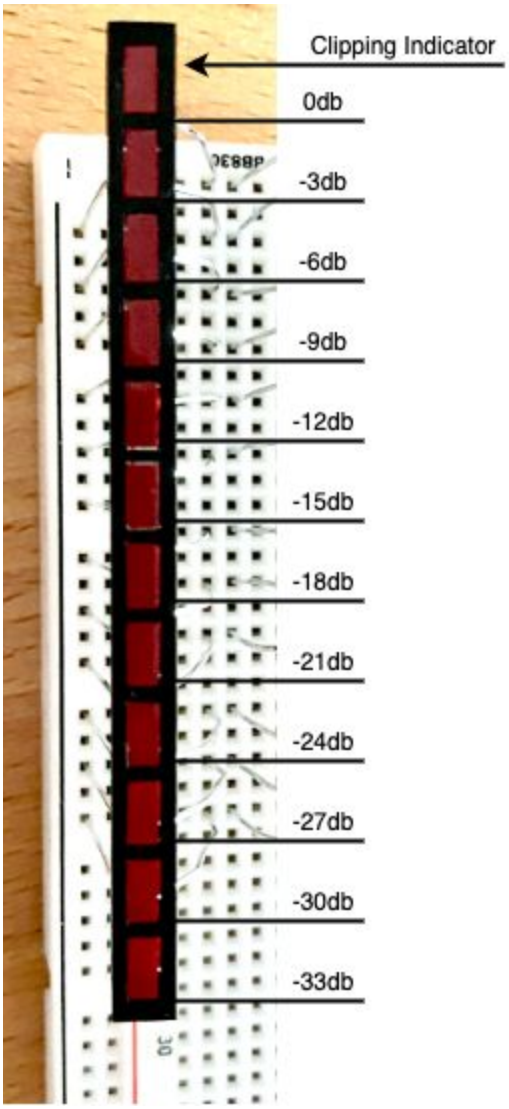

Fig.7 - LED array

Microcontroller connections

The following diagram shows the connections between the main modules and the microcontroller. The output GPIO pins are connected to the LEDs, The input GPIO pin is connected to the onboard switch and the ADC input pin is connected to the preamp circuit. The ground pin is connected to the cathode of the LEDs and the ground of the preamp circuit.

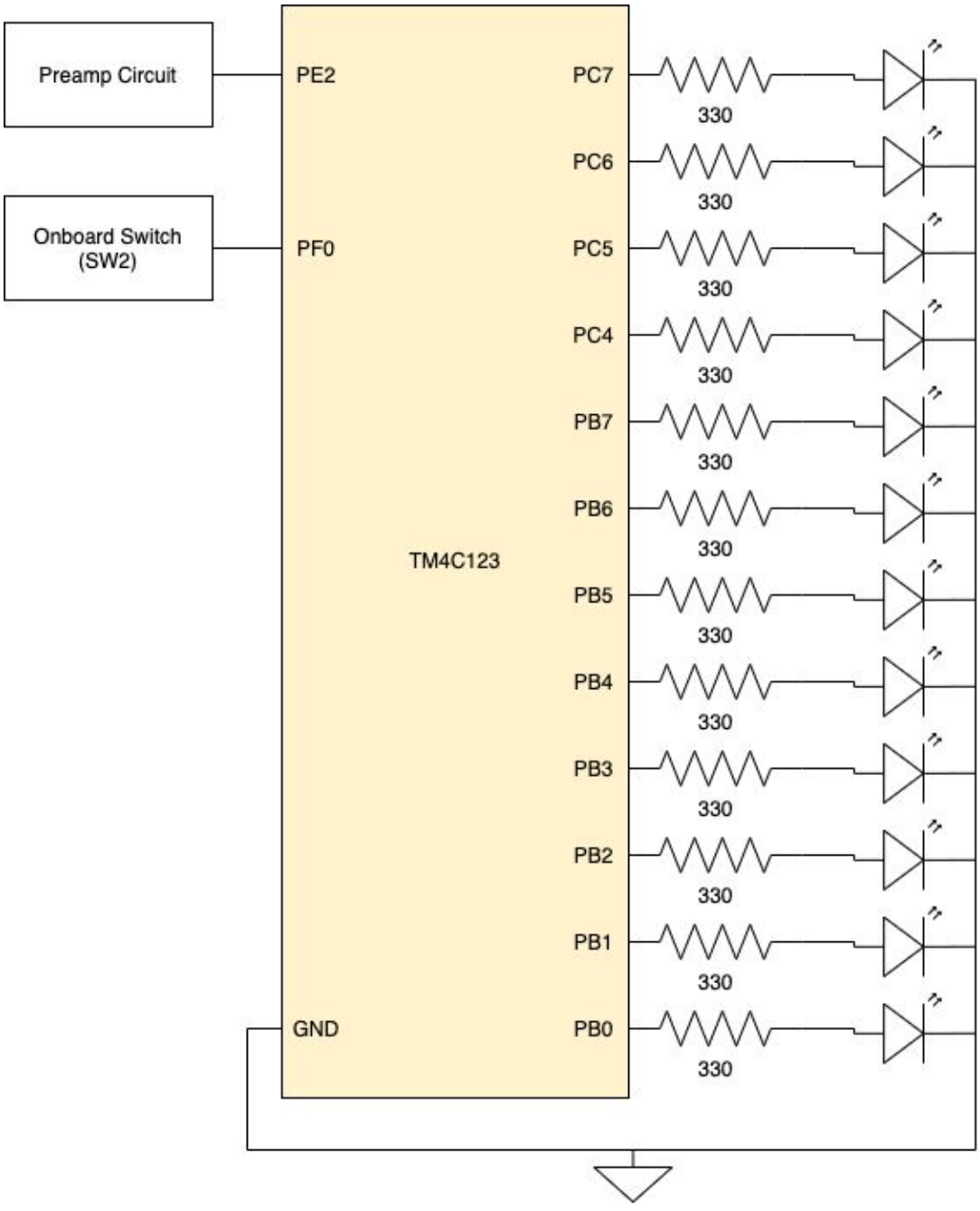

Fig.8 - Schematic of the TM4C123 pins

Software Design

The software uses one function for calculations, two initialization functions and an interrupt handler. The main function is using software polling to decide what LEDs should light up. The software is written using the math.h and tivaware libraries.

CalculateThresh

The CalculateThresh function is used to calculate the thresholds to be used by the if statements in the main function. The function receives a meter resolution value in db and returns an array of thresholds. The first threshold is set to be the full scale of the ADC. Each consecutive threshold is calculated by multiplying the previous threshold by the multiplier. The multiplier calculation is $multiplier = 10^{dbValue/20}$ this calculation is based on the decibel formula $20log(gain) = db$.

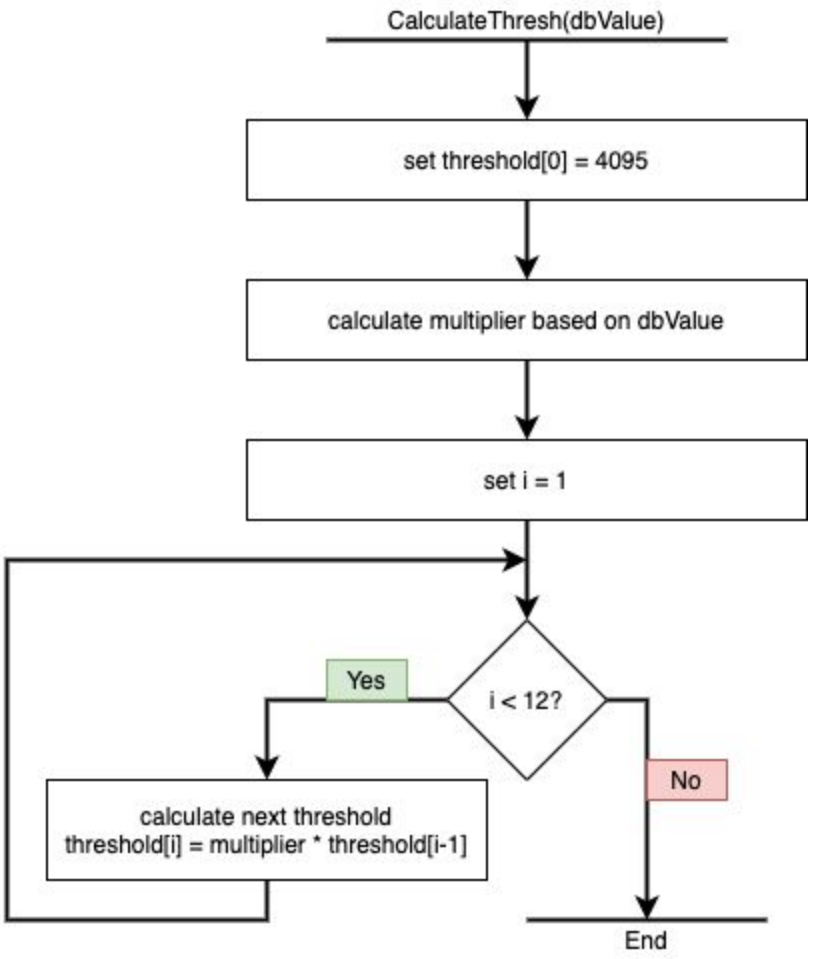

Fig.9 - Flowchart for CalculateThresh function

Initializations and Interrupts

PinInit is a function used to configure the GPIO inputs and outputs. The output pins are using positive logic while the input pin is using negative logic. The function enables the clock for ports B, C and F, and then configures the GPIO pins and enables the pull up register for the switch on pin PF0.

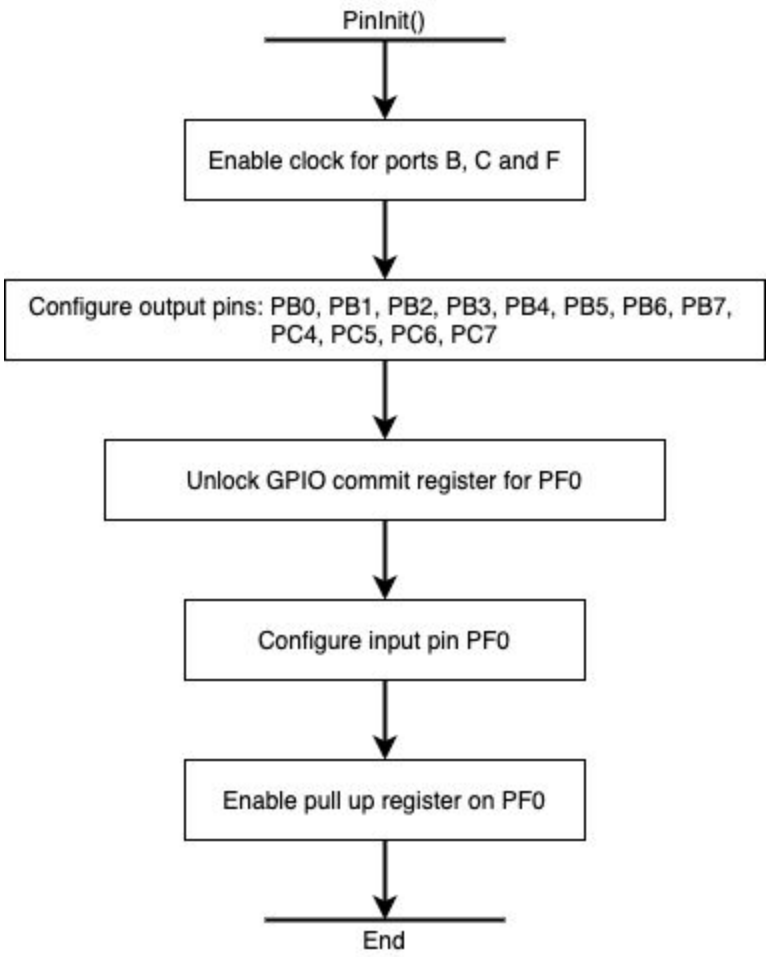

Fig.10 - Flowchart for PinInit function

ADC0_Init is a function used to initialize and configure the ADC. it starts by configuring the system clock to be 40MHz. I chose to use this clock speed since it is convenient and much higher than the nyquist frequency of the audible range. The function then enables port E and configures pin PE2 as an ADC pin.

The sample sequence on ADC0 is configured to work with SS1 and to be processor triggered with a priority of 0. The sample sequence has only one step, meaning the first sample is the last sample in a sequence. The last configuration in this function is to enable interrupts and arm the interrupt on ADC0.

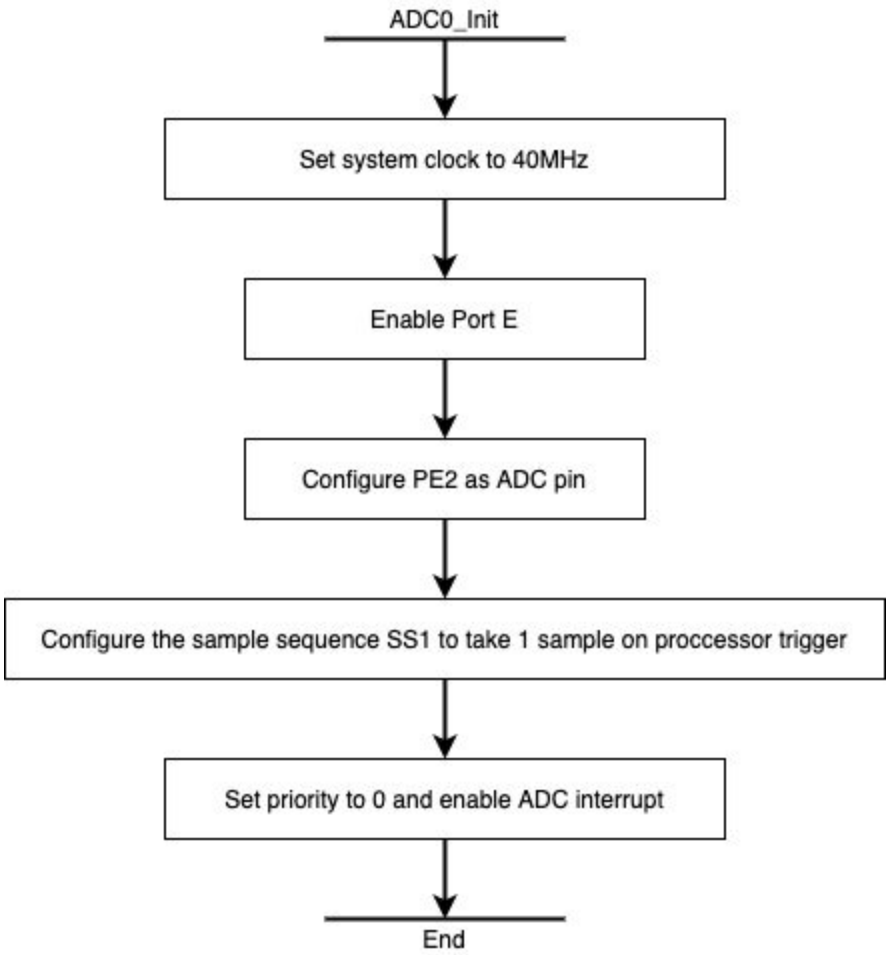

Fig.11 - ADC0_Init flow chart

The ADC0_Handler is a function that handles the data from the ADC, it starts by clearing the interrupt and then triggers the sample sequence.
It collects the sampled data from the sample sequence and sends it to the variable ui32Sample. The variable ui32Sample is a global variable that is later used in the main functions polling process to determine which LEDs should light up.

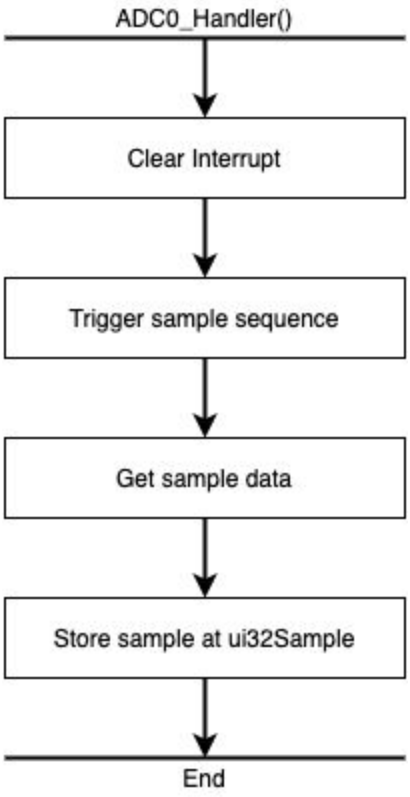

Fig.12 - ADC0_Handler flow chart

Main

The main function calls all of the predefined functions to calculate the thresholds and initialize the GPIO and ADC, it then enters a while loop. In the loop the function polls the status of SW2, if the switch is pressed the clipping indicator is turned off. Then a number of if statements poll the sample to determine which LEDs to turn on or off. If the sample is not smaller than one of the thresholds, the clipping indicator is turned on. The clipping indicator can only be turned off if SW2 is pressed.

The full process can be seen in the software flow chart on the next page.

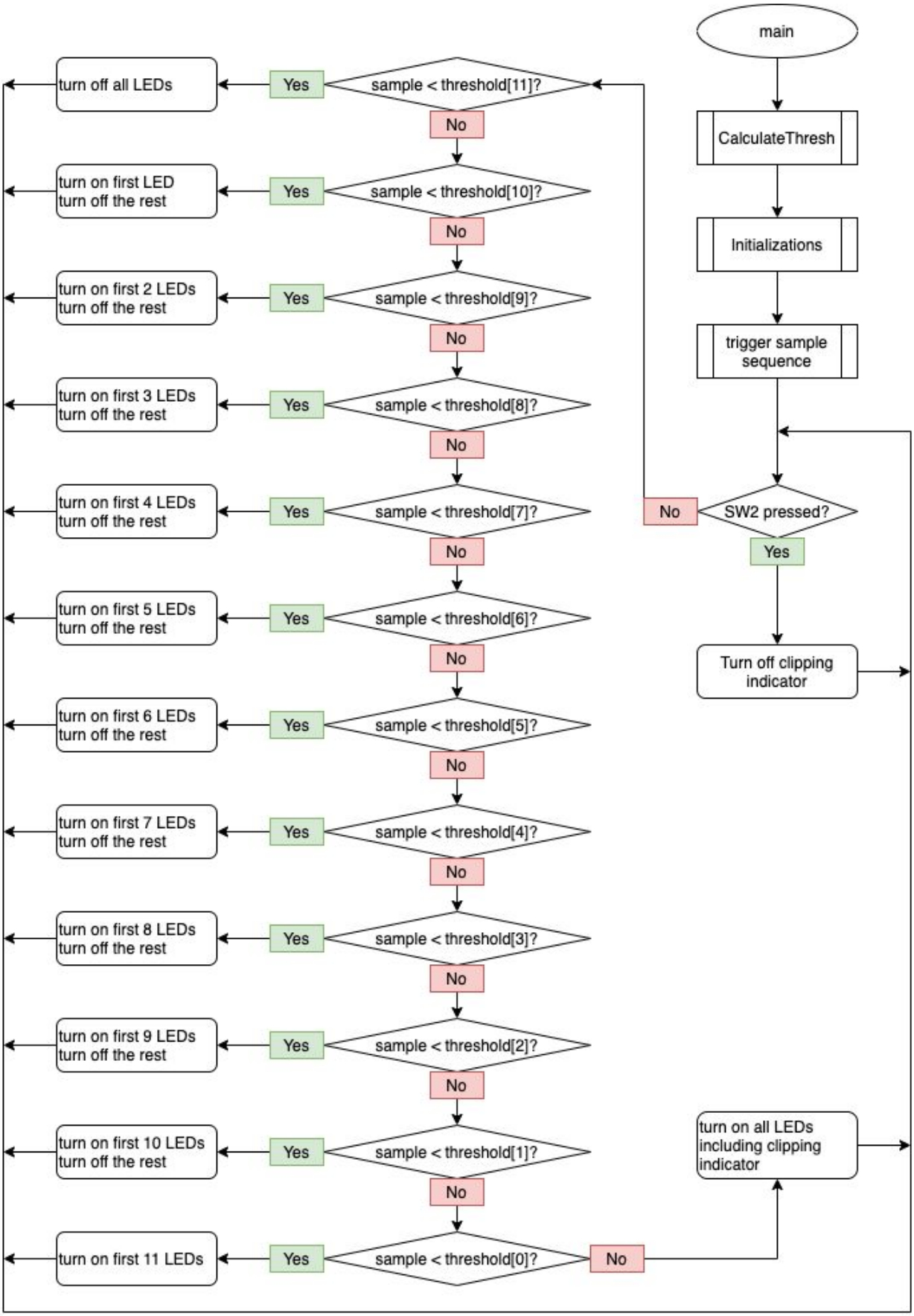

Fig.13 - software flow chart

Experiments

I’ve tested the system by playing guitar through the system, the meter seemed to behave in the desired way and the clipping indicator and clipping indicator reset switch worked without any problems.

To further evaluate the system performance I’ve connected a test signal to the input of the system and observed the meter while changing the signal parameters. The signal was a sine wave that I generated using a signal generator plugin in the Avid Pro Tools DAW.

Fig.14 signal generator plugin in Pro Tools

The signal was generated by software and converted to analog signal using a Focusrite Scarlett 2i2 audio interface. I’ve adjusted the interface output so that the reading is 0db on the meter when the signal generator is at 0db.

Then I changed the level parameter on the signal generator to see if the reading on the meter matched the signal generator, I’ve repeated the experiment for 3 different frequencies and recorded the results on the next page.

Results and Discussion

Signal Generator volume	Reading on meter (1 kHz signal)	Reading on meter (10 kHz signal)	Reading on meter (20 kHz signal)
0	0	-12db	-24db
-3db	-3db	-12db	-24db
-6db	-6db	-12db	-24db
-9db	-9db	-12db	-24db
-12db	-12db	-12db	-24db
-15db	-15db	-15db	-24db
-18db	-18db	-18db	-24db
-21db	-21db	-21db	-24db
-24db	-24db	-24db	-27db
-27db	-27db	-27db	-30db
-30db	-30db	-30db	-33db
-33db	-33db	-33db	n/a

Table1 - Results from first experiment

After finding that the results are off for higher frequencies I assumed that the clock speed is not high enough to accommodate for higher frequencies. I changed the clock speed of the system to be 80Mhz and repeated the experiment, the results of the second experiment are in table 2.

Signal Generator volume	Reading on meter (1 kHz signal)	Reading on meter (10 kHz signal)	Reading on meter (20 kHz signal)
0	0	0	0
-3db	-3db	-3db	-3db
-6db	-6db	-6db	-6db
-9db	-9db	-9db	-9db
-12db	-12db	-12db	-12db

-15db	-15db	-15db	-15db
-18db	-18db	-18db	-18db
-21db	-21db	-21db	-21db
-24db	-24db	-24db	-24db
-27db	-27db	-27db	-27db
-30db	-30db	-30db	-30db
-33db	-33db	-33db	-33db

Table2 - Results from second experiment

After the adjustment to the clock frequency the system works as desired over the entire frequency range.

Conclusion and Future Work

In this project I've implemented a dbFS meter on the TM4C123 microcontroller. The system samples the signal and generates a meter reading relative to the full scale of the ADC. if I had more time I would add a software function that would process the signal by convolving it with an impulse response and then send the signal to an DAC using the SSI protocol.

Source Code

```
#include <stdint.h>
#include <stdbool.h>
#include <math.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "driverlib/adc.h"
#include "driverlib/interrupt.h"
#include "inc/tm4c123gh6pm.h"
#include "driverlib/gpio.h"
#include "inc/hw_gpio.h"

//global variables
uint32_t ui32ADC0Value[1]; //data array to store samples from ADC SS1
volatile uint32_t ui32Sample; //sample to be used in meter
volatile float threshold[12]; //data array to store thresholds from CalculateThresh()

//pin initialization
void PinInit()
{
   //enable clock for GPIO ports B, C, F
   SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
   SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
   SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);

   //configures output pins
   GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE, GPIO_PIN_0 + GPIO_PIN_1 + GPIO_PIN_2 + GPIO_PIN_3 + GPIO_PIN_4
       + GPIO_PIN_5 + GPIO_PIN_6 + GPIO_PIN_7);
   GPIOPinTypeGPIOOutput(GPIO_PORTC_BASE, GPIO_PIN_4 + GPIO_PIN_5 + GPIO_PIN_6 + GPIO_PIN_7);

   //unlock the GPIO commit register
   HWREG(GPIO_PORTF_BASE + GPIO_O_LOCK) = GPIO_LOCK_KEY;
   HWREG(GPIO_PORTF_BASE + GPIO_O_CR) = 0x1;
   //configure input pin
   GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, GPIO_PIN_0);
   //enable pull up register
   GPIO_PORTF_PUR_R |= 0x01;
}
```

```
//ADC initialization
void ADC0_Init(void)
{
   // configure the system clock to be 80MHz
   SysCtlClockSet(SYSCTL_SYSDIV_2_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);
   //enable clock for ADC0 and GPIO port E
   SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
   SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
   //configure ADC pin
   GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_2 | GPIO_PIN_2);
   //allow the clock to be fully activated
   SysCtlDelay(2);

   //disable ADC0 before the configuration is complete
   ADCSequenceDisable(ADC0_BASE, 1);
   //configure the sequence to use (ADC0, SS1, processor-trigger, priority 0)
   ADCSequenceConfigure(ADC0_BASE, 1, ADC_TRIGGER_PROCESSOR, 0);
   //configure step 0 of the sequence to be the last
   ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_CH1|ADC_CTL_IE|ADC_CTL_END);

   //configure ADC0 SS1 interrupt priority as 0
   IntPrioritySet(INT_ADC0SS1, 0x00);
   //enable interrupt 31 in NVIC (ADC0 SS1)
   IntEnable(INT_ADC0SS1);
   //arm interrupt of ADC0 SS1
   ADCIntEnableEx(ADC0_BASE, ADC_INT_SS1);
   //enable ADC0
   ADCSequenceEnable(ADC0_BASE, 1);
}

//interrupt handler
void ADC0_Handler(void)
{
   //clear interrupt
   ADCIntClear(ADC0_BASE, 1);
   //trigger sample sequence
   ADCProcessorTrigger(ADC0_BASE, 1);
   //get data from sample sequence
   ADCSequenceDataGet(ADC0_BASE, 1, ui32ADC0Value);
   //put step 0 of the sample sequence into ui32Sample
   ui32Sample = ui32ADC0Value[0];
```

```
}

//calculate thresholds based on meter resolution in db
void CalculateThresh(float dbValue)
{
   threshold[0] = 4095;
   float multiplier = pow(10,(dbValue/20));
   for (int i = 1; i < 12; i++)
   {
       threshold[i] = multiplier*threshold[i-1];
   }
}

int main(void)
{
   CalculateThresh(-3); //calculate thresholds for -3db resolution
   PinInit();  //initialize GPIO pins
   ADC0_Init();    //initialize ADC
   IntMasterEnable();  //globally enable interrupt
   ADCProcessorTrigger(ADC0_BASE, 1);  //trigger sample sequence
   while (1)
   {
       //is SW2 is pressed, reset clipping indicator
       if(GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_0)!=0x01)
       {
           GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7, 0x00);
       }
       //turn on the appropriate LEDs based on sample
       else if (ui32Sample<threshold[11])
       {
           GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_0, 0x00);
           GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_1, 0x00);
           GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0x00);
           GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, 0x00);
           GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0x00);
           GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_5, 0x00);
           GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_6, 0x00);
           GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_7, 0x00);
           GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_4, 0x00);
           GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_5, 0x00);
           GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, 0x00);
       }
```

```
else if (ui32Sample<threshold[10])
{
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_0, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_1, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_5, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_6, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_7, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_4, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_5, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, 0x00);
}
else if (ui32Sample<threshold[9])
{
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_0, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_1, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_5, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_6, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_7, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_4, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_5, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, 0x00);
}
else if (ui32Sample<threshold[8])
{
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_0, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_1, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_5, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_6, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_7, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_4, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_5, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, 0x00);
}
```

```
else if (ui32Sample<threshold[7])
{
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_0, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_1, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_5, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_6, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_7, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_4, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_5, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, 0x00);
}
else if (ui32Sample<threshold[6])
{
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_0, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_1, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_5, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_6, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_7, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_4, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_5, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, 0x00);
}
else if (ui32Sample<threshold[5])
{
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_0, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_1, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_5, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_6, 0x00);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_7, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_4, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_5, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, 0x00);
}
```

```
else if (ui32Sample<threshold[4])
{
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_0, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_1, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_5, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_6, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_7, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_4, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_5, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, 0x00);
}
else if (ui32Sample<threshold[3])
{
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_0, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_1, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_5, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_6, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_7, 0xFF);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_4, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_5, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, 0x00);
}
else if (ui32Sample<threshold[2])
{
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_0, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_1, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_5, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_6, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_7, 0xFF);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_4, 0xFF);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_5, 0x00);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, 0x00);
}
```

```
else if (ui32Sample<threshold[1])
{
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_0, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_1, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_5, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_6, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_7, 0xFF);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_4, 0xFF);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_5, 0xFF);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, 0x00);
}
else if (ui32Sample<threshold[0])
{
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_0, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_1, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_5, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_6, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_7, 0xFF);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_4, 0xFF);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_5, 0xFF);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, 0xFF);
}
else
{
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_0, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_1, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_2, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_3, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_5, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_6, 0xFF);
    GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_7, 0xFF);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_4, 0xFF);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_5, 0xFF);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, 0xFF);
    //turn on clipping indicator
```

```
            GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7, 0xFF);
        }
    }
}
```