Sparkfun Codec Radio

or the "Sparkfun Codec Thing"

John Talbert - March 2023

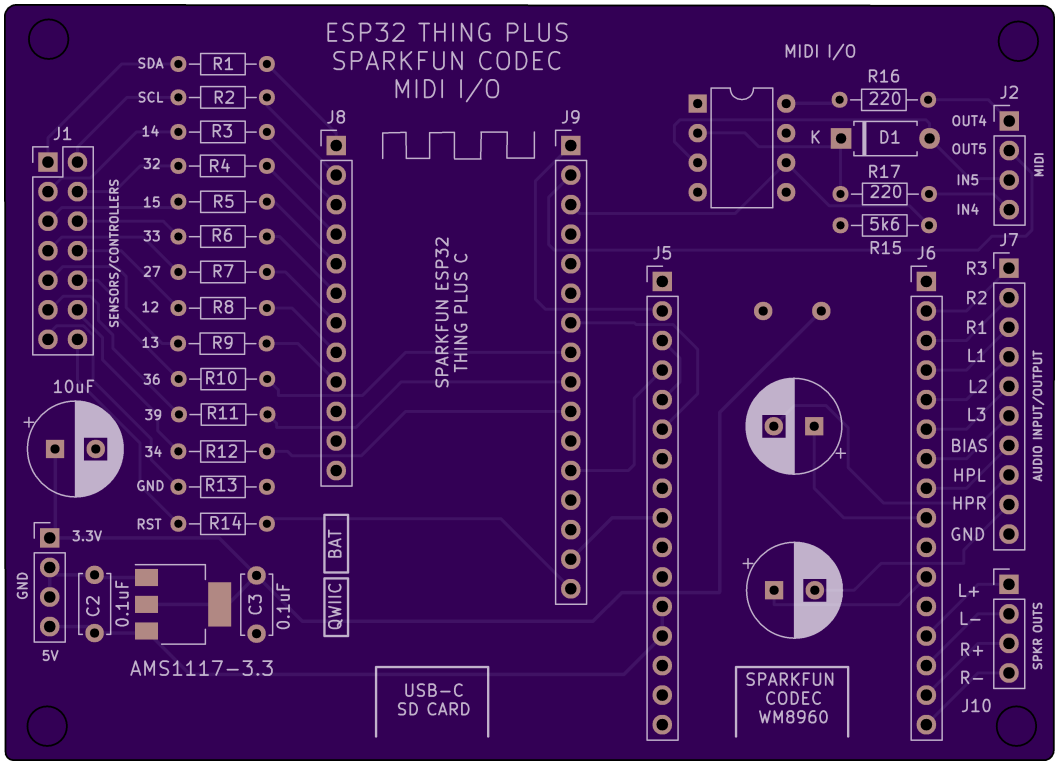
Table of Contents

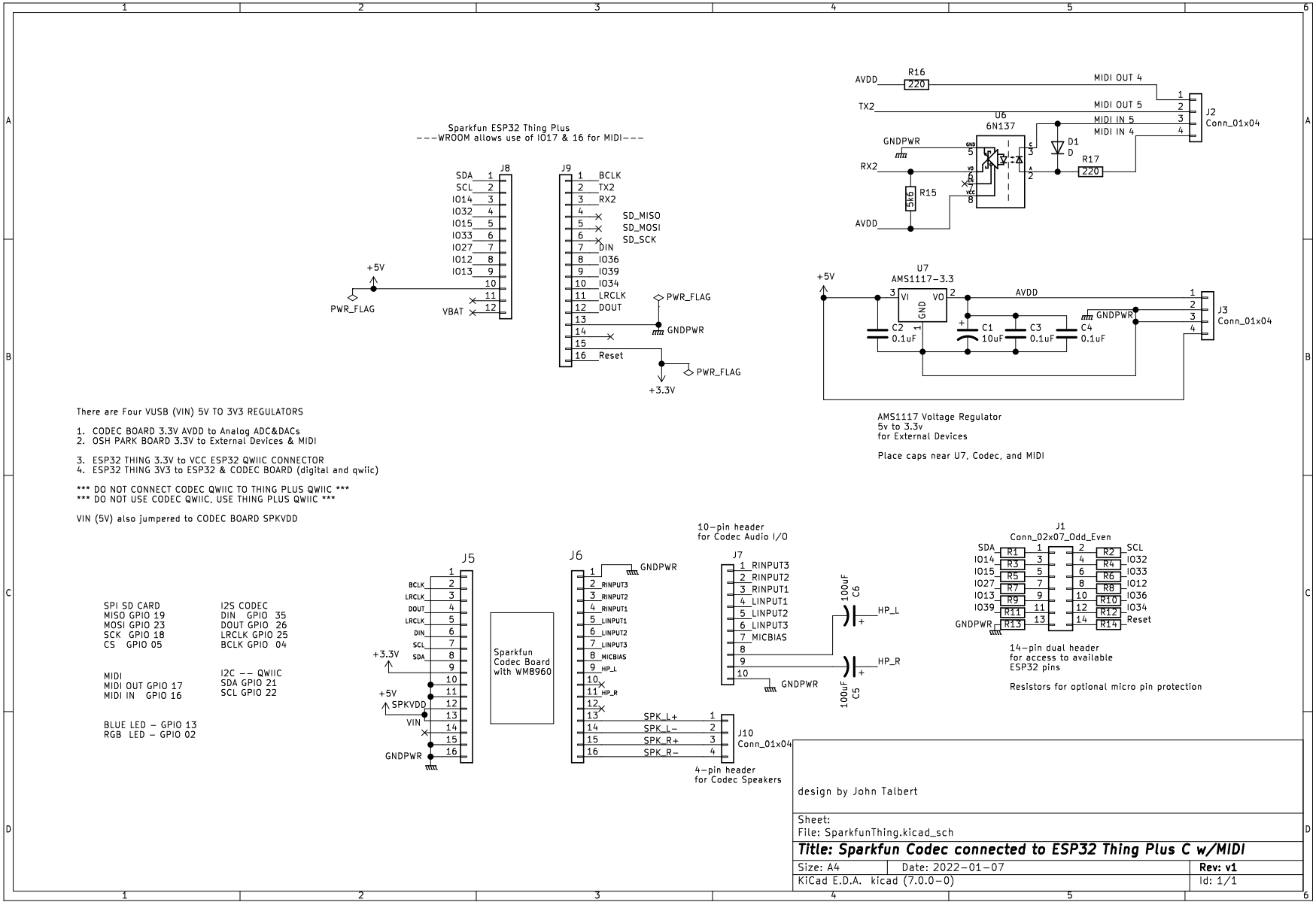
The Board

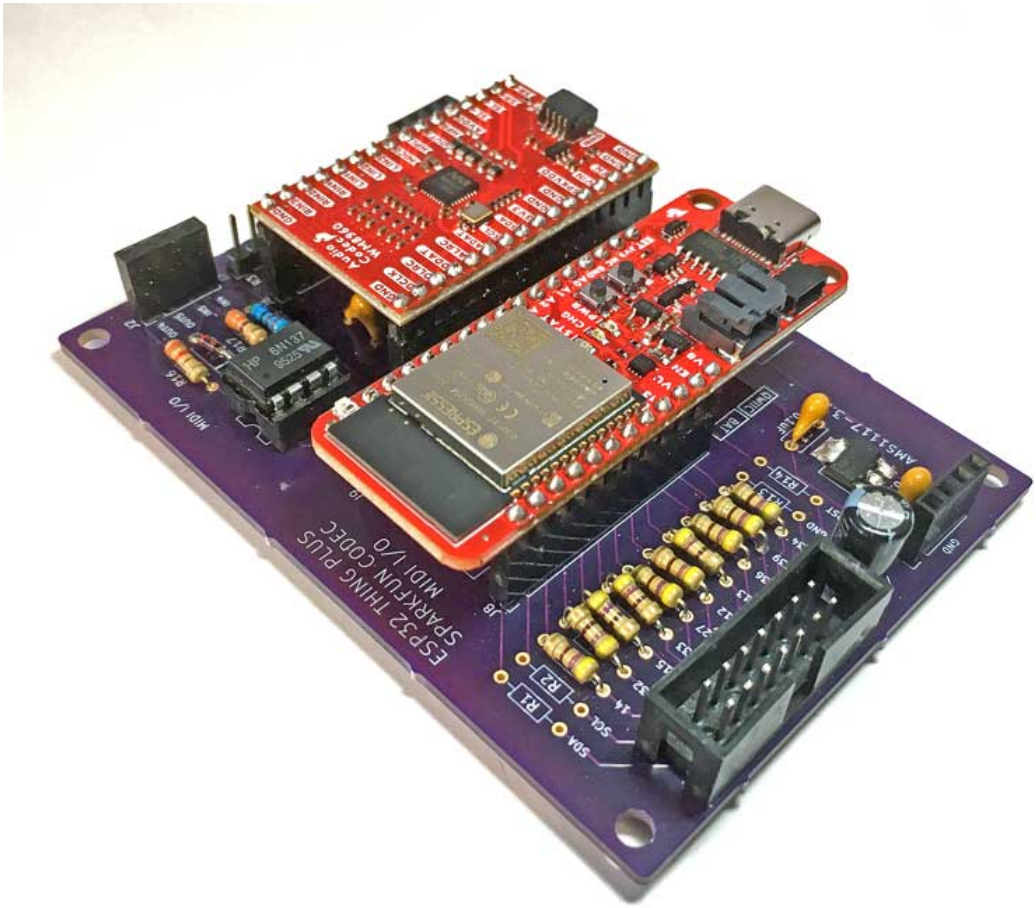
This Radio Enclosure incorporates a PCB board that connects the Sparkfun Codec module to a Sparkfun ESP32 Thing Plus C processor module. The Codec module (https://www.sparkfun.com/products/21250) uses a WM8960 codec by Cirrus Logic (https://datasheetspdf.com/pdf-file/1365067/CirrusLogic/WM8960/1). The ESP32 Thing Plus (https://www.sparkfun.com/products/20168) uses a WROOM 32E Processor. Check the Sparkfun website for full documentation and tutorials.

The PCB board includes a MIDI Input/Output interface and a convenient GPIO breakout header for connecting external sensors and controllers to available GPIO pins. A Sparkfun Qwiic connector is available on both the ESP32 module and the Codec module, and an SD Card Slot is included on the underside of the ESP32 Thing Plus module.

The PCB board is available from OSHPark circuit board fabricators (https://oshpark.com) under "Shared Projects" by john.talbert@oberlin.edu at a cost of about $45 for a minimum of 3 boards.




Sparkfun ESP32 Thing Plus
---WROOM allows use of IO17 & 16 for MIDI---
J8
SDA 1
SCL 2
IO14 3
IO32 4
IO15 5
IO33 6
IO27 7
IO12 8
IO13 9
10
11
12
VBAT
+5V
PWR_FLAG
J9
1 BCLK
2 TX2
3 RX2
4
5 SD_MISO
6 SD_MOSI
7 SD_SCK
8 DIN
9 IO36
10 IO39
11 IO34
12 LRCLK
13 DOUT
14
15
16 Reset
PWR_FLAG
GNDPWR
PWR_FLAG
+3.3V
There are Four VUSB (VIN) 5V TO 3V3 REGULATORS
1. CODEC BOARD 3.3V AVDD to Analog ADC&DACs
2. OSH PARK BOARD 3.3V to External Devices & MIDI
3. ESP32 THING 3.3V to VCC ESP32 QWIIC CONNECTOR
4. ESP32 THING 3V3 to ESP32 & CODEC BOARD (digital and qwiic)
*** DO NOT CONNECT CODEC QWIIC TO THING PLUS QWIIC ***
*** DO NOT USE CODEC QWIIC, USE THING PLUS QWIIC ***
VIN (5V) also jumpered to CODEC BOARD SPKVDD
SPI SD CARD
MISO GPIO 19
MOSI GPIO 23
SCK GPIO 18
CS GPIO 05
MIDI
MIDI OUT GPIO 17
MIDI IN GPIO 16
BLUE LED – GPIO 13
RGB LED – GPIO 02
I2S CODEC
DIN GPIO 35
DOUT GPIO 26
LRCLK GPIO 25
BCLK GPIO 04
I2C -- QWIIC
SDA GPIO 21
SCL GPIO 22
AVDD
R16
220
MIDI OUT 4
TX2
MIDI OUT 5
U6
6N137
MIDI IN 5
MIDI IN 4
J2
Conn_01x04
GNDPWR
D1
D
RX2
R17
220
5K6
R15
AVDD
U7
AMS1117-3.3
+5V
VI
VO
GND
AVDD
GNDPWR
C2
0.1uF
C1
10uF
C3
0.1uF
C4
0.1uF
J3
Conn_01x04
AMS1117 Voltage Regulator
5v to 3.3v
for External Devices
Place caps near U7, Codec, and MIDI
10-pin header
for Codec Audio I/O
J5
BCLK
LRCLK
DOUT
LRCLK
DIN
SCL
SDA
+3.3V
+5V
SPKVDD
VIN
GNDPWR
Sparkfun
Codec Board
with WM8960
J6
GNDPWR
RINPUT3
RINPUT2
RINPUT1
LINPUT1
LINPUT2
LINPUT3
MICBIAS
HP_L
HP_R
SPK_L+
SPK_L-
SPK_R+
SPK_R-
J7
RINPUT3
RINPUT2
RINPUT1
LINPUT1
LINPUT2
LINPUT3
MICBIAS
GNDPWR
100uF
C6
100uF
C5
HP_L
HP_R
J10
Conn_01x04
4-pin header
for Codec Speakers
J1
Conn_02x07_Odd_Even
SDA
IO14
IO15
IO27
IO13
IO39
GNDPWR
SCL
IO32
IO33
IO12
IO36
IO34
Reset
14-pin dual header
for access to available
ESP32 pins
Resistors for optional micro pin protection
design by John Talbert
Sheet:
File: SparkfunThing.kicad_sch
Title: Sparkfun Codec connected to ESP32 Thing Plus C w/MIDI
Size: A4
Date: 2022-01-07
Rev: v1
KiCad E.D.A. kicad (7.0.0-0)
Id: 1/1


ESP32 THING PLUS
SPARKFUN CODEC
MIDI I/O
MIDI I/O
AMS1117-3.3


SDA
SCL
R1
R2
14
32
15
33
27
12
13
36
39
34
GND
R13
RST
R14
3.3V
0.1uF
0.1uF
AMS1117-3.3
SENSORS/ICONTROLLERS
ESPRESSIF
ESP32-WROOM-32E
MIDI I/O
R16
R17
OUT4
OUT5
IN5
IN4
J2
J5
Audio Codec WM8960
GND
BCLK
DLRC
DDAT
ALRC
ADAT
SCL
SDA
3V3
GND
GND
SPKVDD
VIN
EN
GND
GND
RIN3
RIN2
RIN1
LIN1
LIN2
LIN3
MICD
HPL
OUT3
HPR
AVDD
SL
SL
SR
SR
STAT
CHG
PWR
BOOT

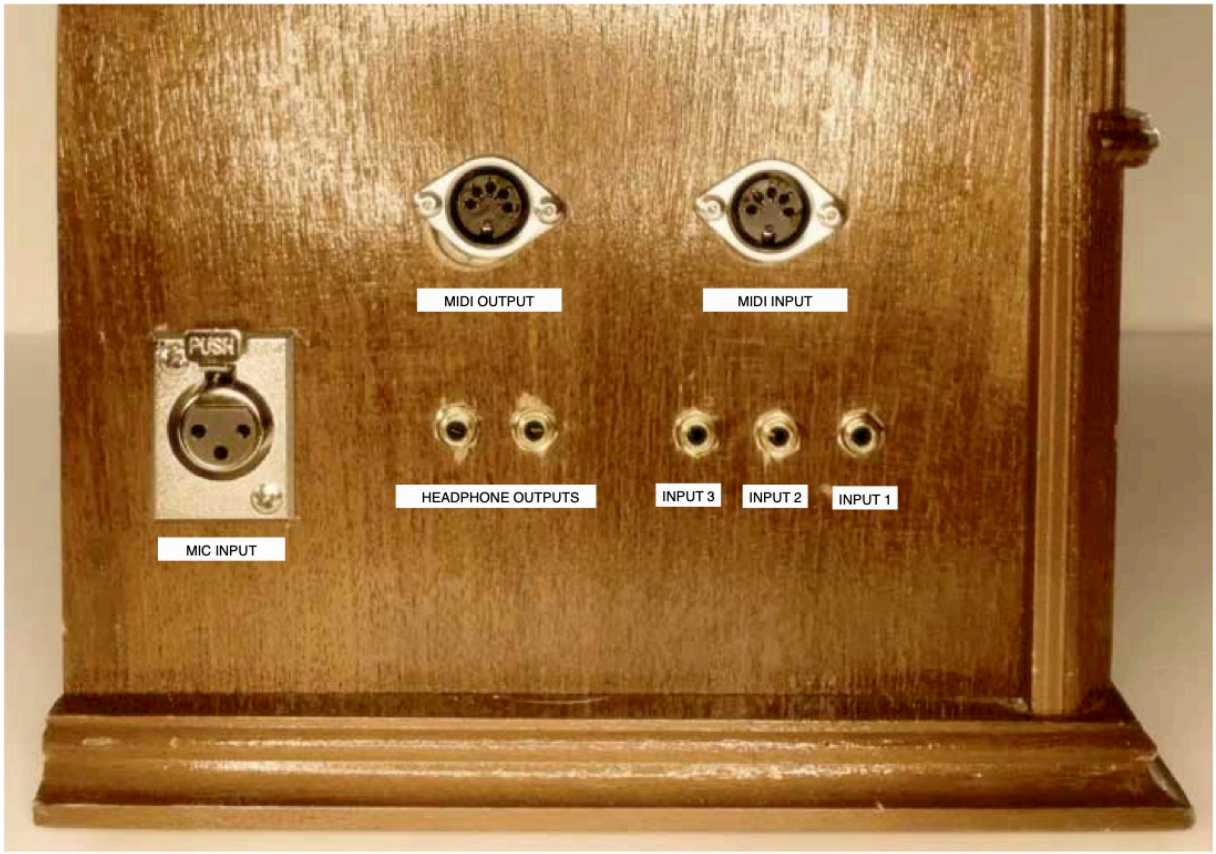
The Radio Components

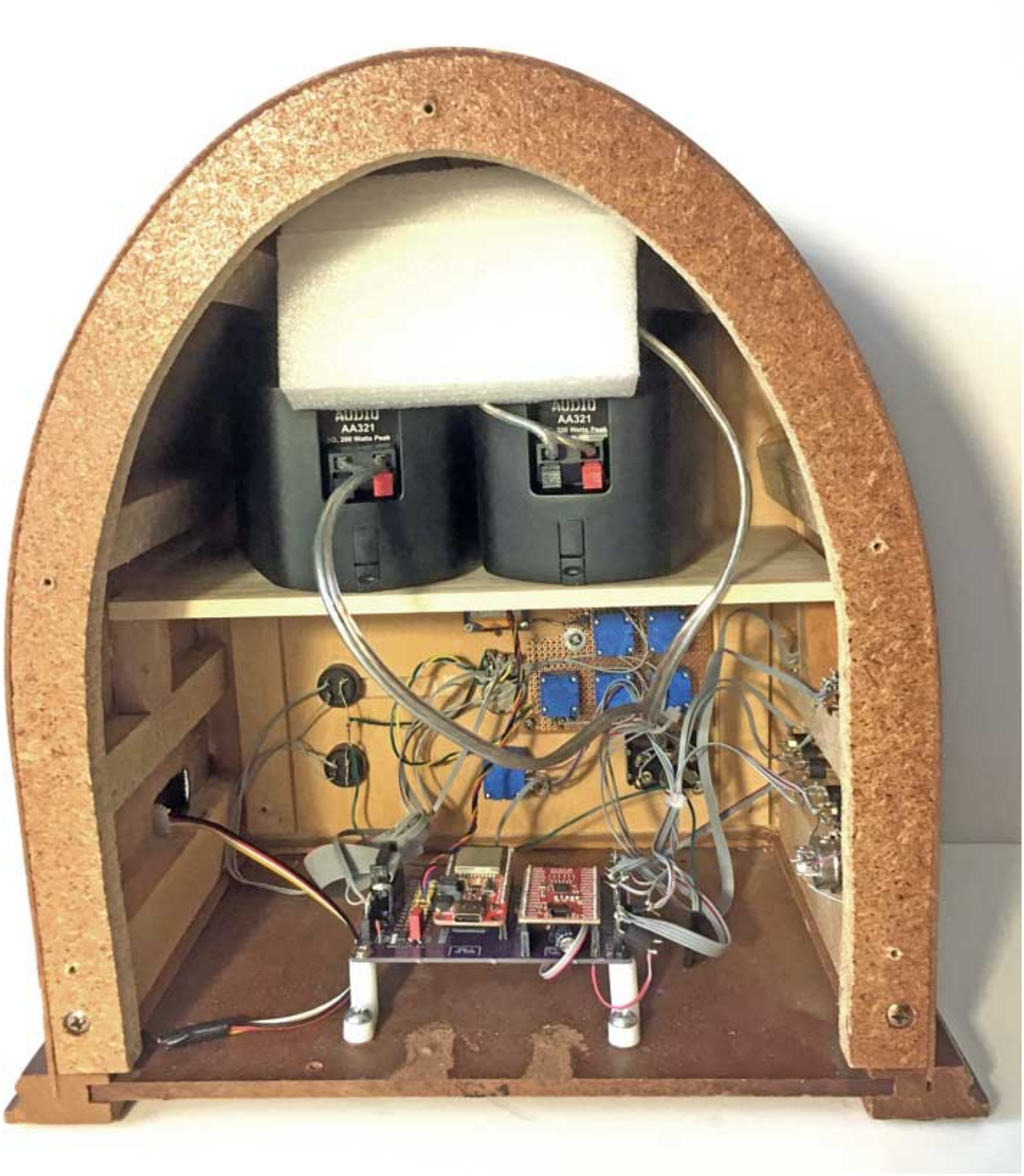
PCB Board with Sparkfun Thing Plus ESP32 and Sparkfun WM8960 Codec
MIDI Input and Output using 5-pin DIN
Two 8 Ohm Speakers - Acoustic Audio AA321B
OLED Display, 128x64 Pixels, MSP430 0.9 inch, Qwiic I2C

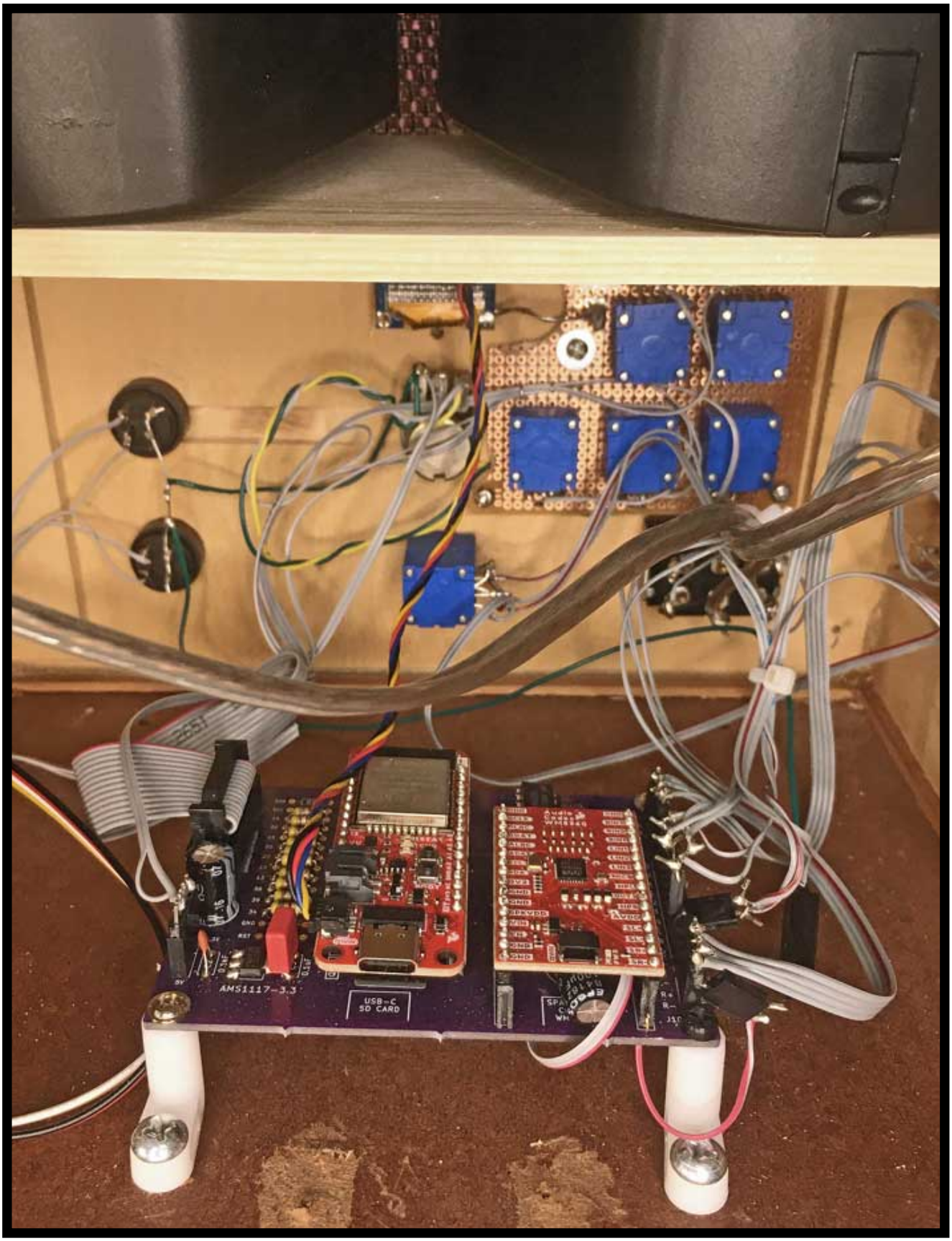
Three Stereo Codec Inputs
Volume Control for Input 3
Headphone Stereo Codec Outputs
Codec Speaker Amp Outputs
On/Off Switch for Speakers
Microphone Input (using Inputs 1 and 2)

Six Potentiometer Sensors
Two Pushbutton Sensors
Two LEDs
Trill Square Pad Sensor, Qwiic I2C


STEREO SPEAKERS
POT1 14
POT2 32
DISPLAY
POT6 34
KEY1 15
LED1 13
POT3 39
POT4 36
POT5 33
TUNING
TONE
VOL
KEY2 27
LED2 12
SPEAKER ON/OFF
INPUT3 VOLUME
THOMAS
Collectors Edition Radio


MIDI OUTPUT
MIDI INPUT
PUSH
HEADPHONE OUTPUTS
INPUT 3
INPUT 2
INPUT 1
MIC INPUT

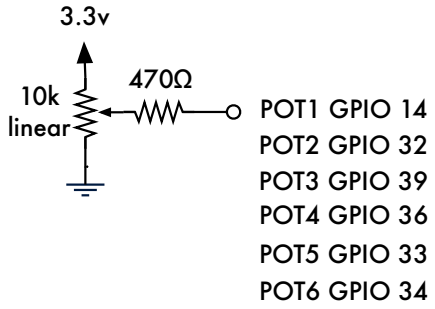

AUDIO
AA321
AUDIO
AA321

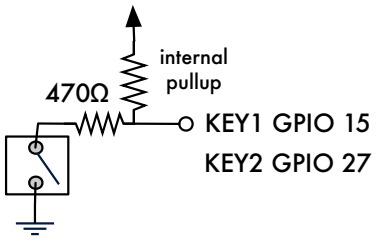

AMS1117-3.3
USB-C
SD CARD

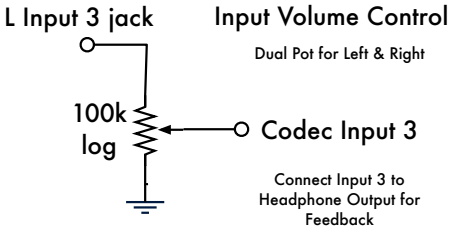
Radio Circuits

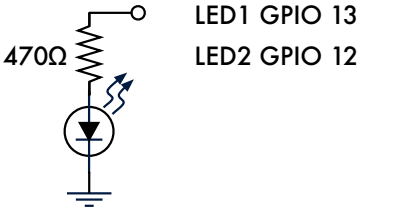
ESP32 / Sparkfun Codec Thing
Sensor — Controller Circuits

John Talbert 2023



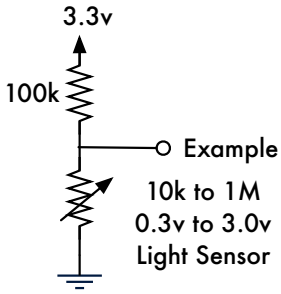


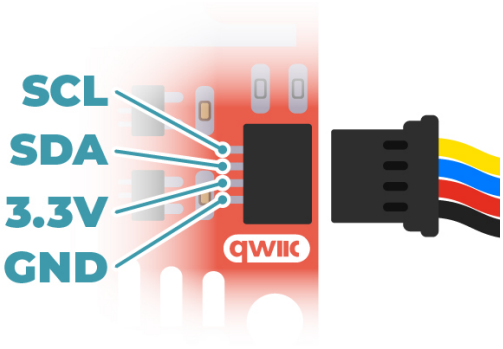


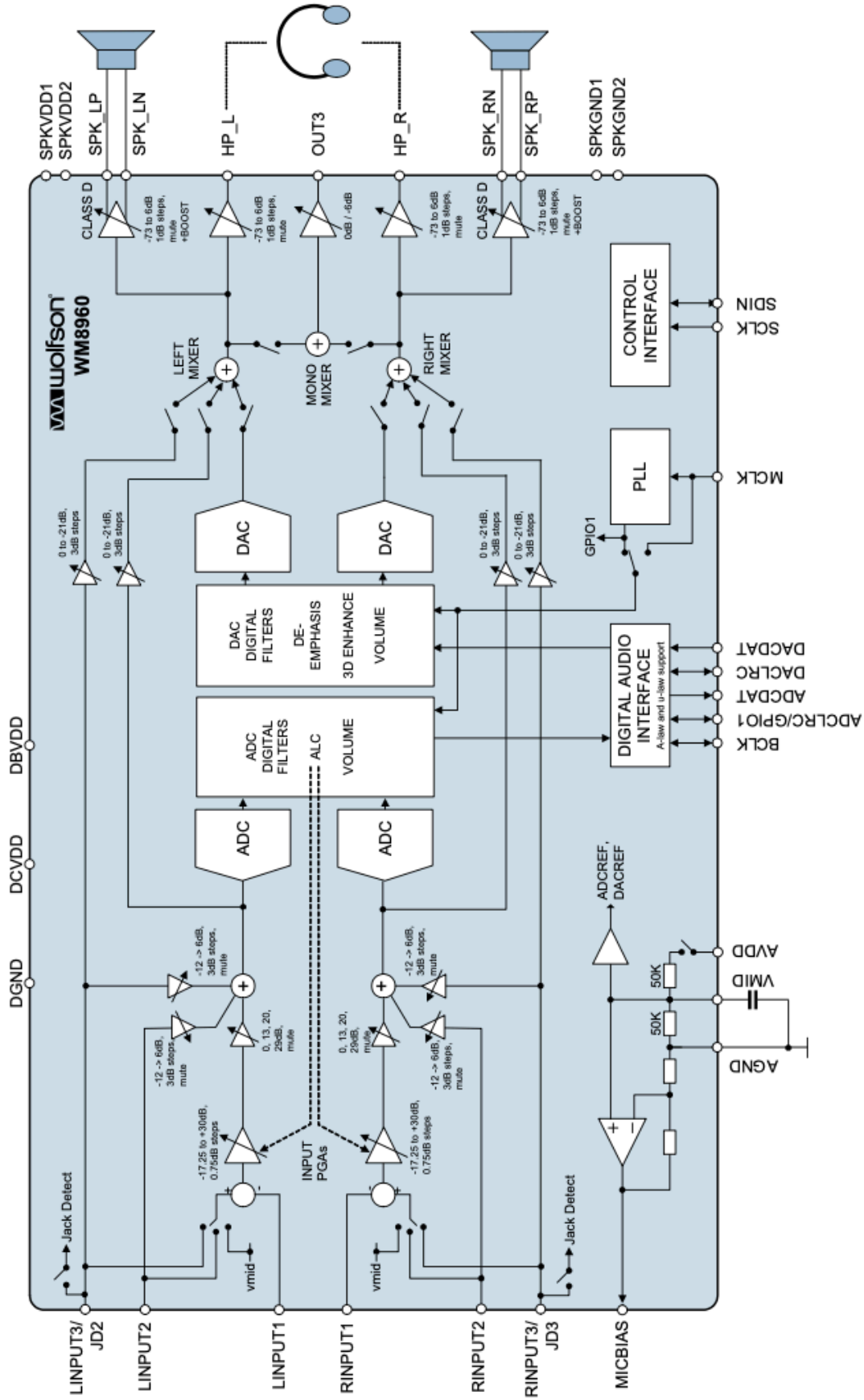


Microphone Input

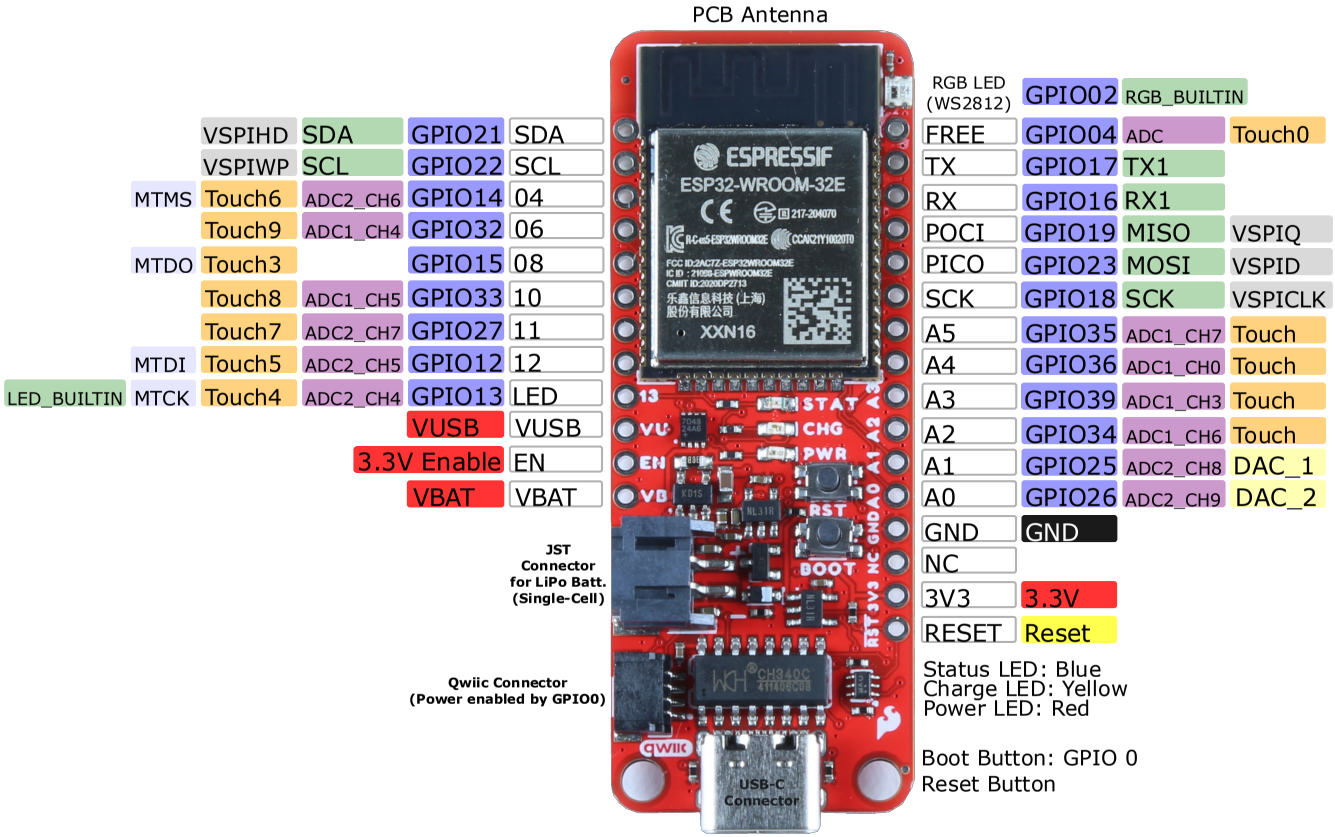
1 Ground
2 Codec Left Input 2
3 Codec Left Input 1






SPKVDD1
SPKVDD2
SPK_LP
SPK_LN
HP_L
OUT3
HP_R
SPK_RN
SPK_RP
SPKGND1
SPKGND2
CLASS D
-73 to 6dB 1dB steps, mute +BOOST
-73 to 6dB 1dB steps, mute
0dB / -6dB
-73 to 6dB 1dB steps, mute
CLASS D
-73 to 6dB 1dB steps, mute +BOOST
wolfson
WM8960
LEFT MIXER
MONO MIXER
RIGHT MIXER
CONTROL INTERFACE
SDIN
SCLK
PLL
MCLK
GPIO1
0 to -21dB, 3dB steps
DAC
DAC
DAC DIGITAL FILTERS
DE-EMPHASIS
3D ENHANCE
VOLUME
DIGITAL AUDIO INTERFACE
A-law and u-law support
DACDAT
DACLRC
ADCDAT
ADCLRC/GPIO1
BCLK
ADC DIGITAL FILTERS
ALC
VOLUME
ADC
ADC
DBVDD
DCVDD
DGND
-12 -> 6dB, 3dB steps, mute
0, 13, 20, 29dB, mute
-17.25 to +30dB, 0.75dB steps
INPUT PGAs
ADCREF, DACREF
AVDD
VMID
50K
50K
AGND
Jack Detect
vmid
vmid
Jack Detect
LINPUT3/ JD2
LINPUT2
LINPUT1
RINPUT1
RINPUT2
RINPUT3/ JD3
MICBIAS

SparkFun ESP32 Thing Plus (USB-C) (WRL-20168)



Power
ESP32 VCC Range: 3.0V-3.6V
VBAT: Direct to battery (and charger)
VUSB: Direct to USB (5V)
3V3: Direct to 3.3V regulator (700mA)
ESP32-S2 Current Consumption:
· WiFi: 380mA (peak)
· Light-Sleep: 800µA
· Deep-Sleep: 10-150µA

Power Control
EN pin - Power for board
GPIO0 - Power to Qwiic Connector

ESP32-WROOM
Dual-Core Xtensa 32-bit LX7
Up to 240MHz
520KB SRAM
16MB SPI Flash

Multiplexed I/Os allow up to
13x 12-bit ADC Channels
3x SPI Interfaces
3x UART Interfaces
2x I2C Interfaces
2x I2S Interface
16x 20-bit PWM Channels
2x 8-bit DACs
8x Capacitive Touch Inputs

Wireless
WiFi: 802.11 b/g/n (up to 150Mbps)
WPA/WPA2/WPA2-Enterprise/SPS
Bluetooth:Bluetooth 4.2/BLE

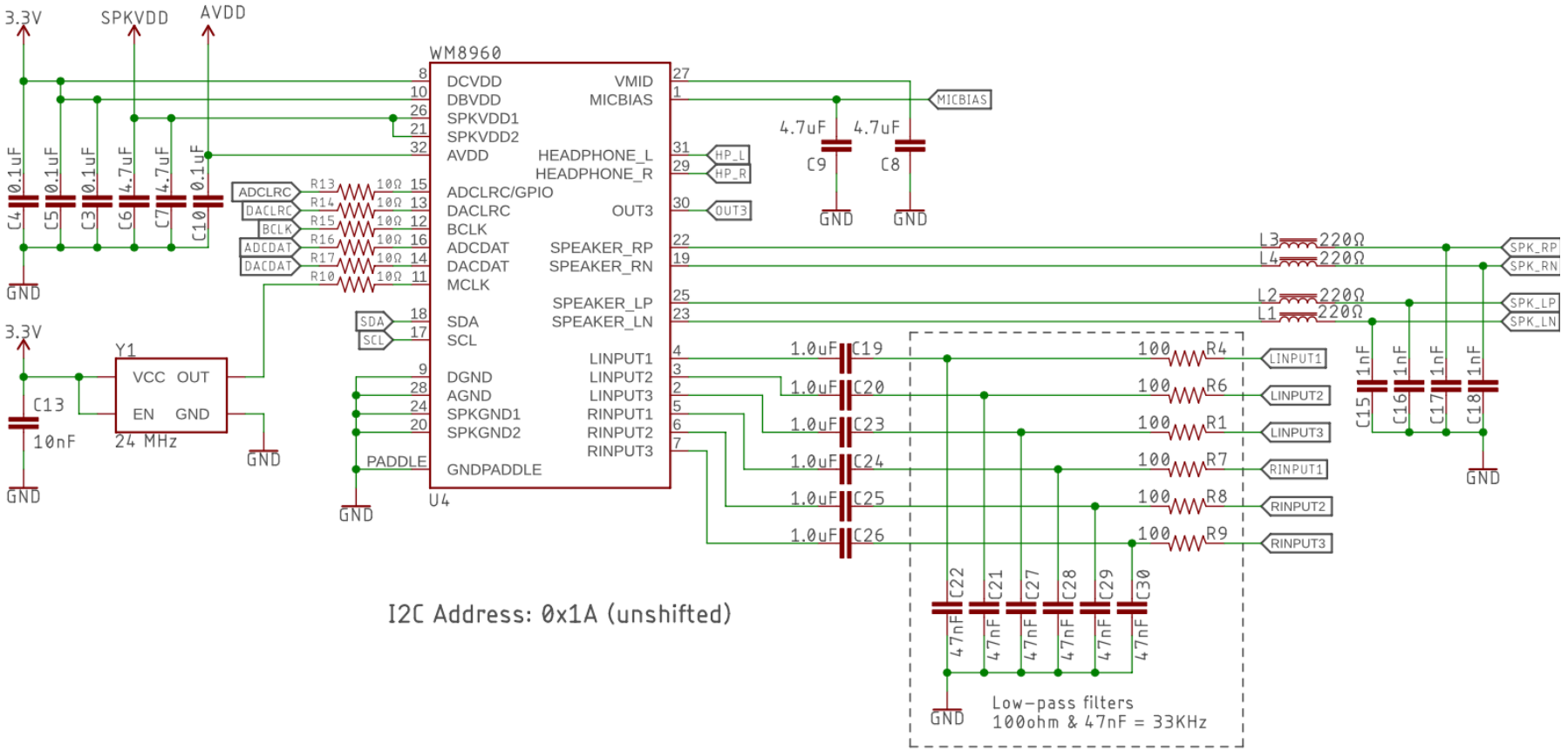
Other*
Temp sensor (-40°C to 125°C)
Hall Sensor
SD/SDIO/MMC Host Controller
· µSD card slot CS: GPIO05 SS
CAN Bus

*On datasheet, but may not be supported yet

sparkfun ELECTRONICS

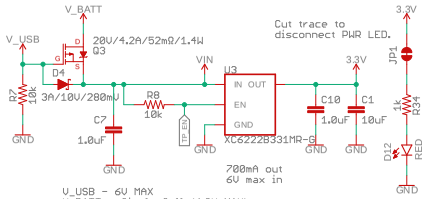
Audio Codec

Supply Voltage Ranges
VIN: 3.7–5.5V*
DCVDD: 1.7–3.6V
DBVDD: 1.7–3.6V
AVDD: 2.7–3.6V
SPKVDD: 2.7–5.5V
* If VIN is used for SPKVDD, then VIN shouold be limited to 5.5V by SPKVDD max range

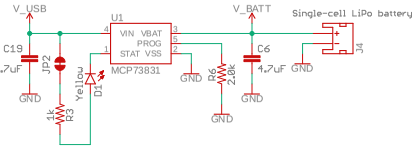


I2C Address: 0x1A (unshifted)

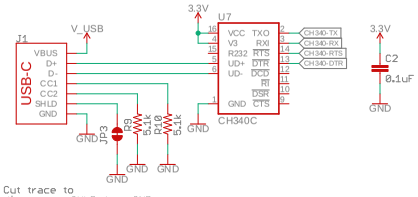
System Regulator



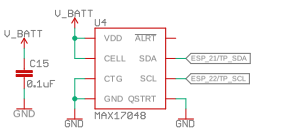
Lithium-Polymer Battery Charger



USB-to-Serial Converter



Fuel Gauge

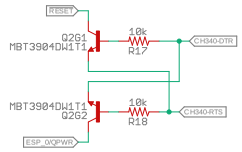


Auto-Reset

Boot Mode Configuration

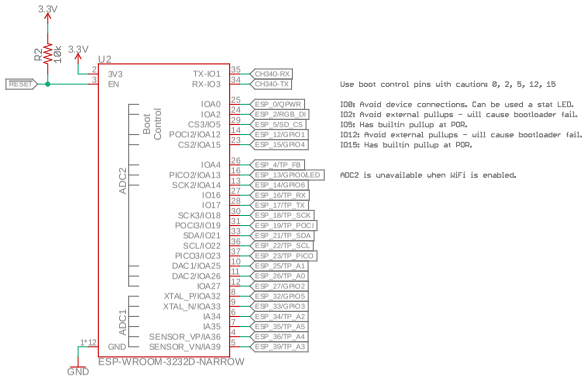
Pin	Run	Bootloader	Default
IO0	1	0	1
IO2	x	0	0
IO5	1	x	1
IO12	0	0	0
IO15	x	x	1

IO12 = MDTI, IO15 = MDTO

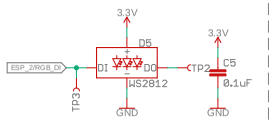


If DTR is LOW, toggling RTS from HIGH to LOW resets to run mode.
If RTS is HIGH, toggling DTR from LOW to HIGH resets to bootloader.

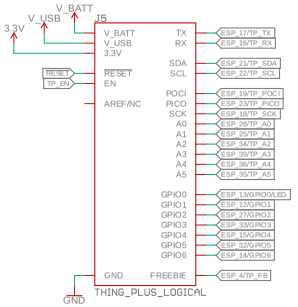
Espressif ESP32-WROOM



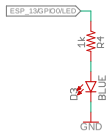
RGB Status LED



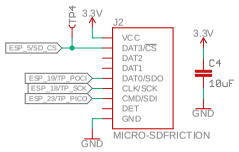
ThingPlus Form Factor



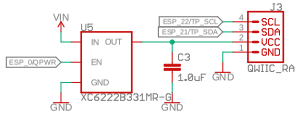
Status LED



microSD Connector



Qwiic Connector



GPIO0/Boot Button

Reset Button

TITLE: SparkFun_Thing_Plus_ESP32_WROOM_C	
Design by: Nathan Seidle Revised by: Alex Brudner	REV: v40
Date: 7/7/2022 1:25 PM	Sheet: 1/1

Software Solutions

The software used here is from a complete Codec/ESP32 Package described on my website at https://www.jtalbert.xyz/ESP32/ Please read the PDF files from the site for more information: Codec Software and SparkfunCodec.

OLED Display Software

The software for the OLED uses the Library Adafruit_SSD1306 version 2.5.7, along with the Adafruit GFX Graphic Library. The Adafruit website has a useful GFX Graphics Library Tutorial.

The OLED code is installed in the software package's **task.cpp** file as one of several other function "tasks" defined in that file. It's format is similar to the NeoPixel task built for the LillyGo TAudio Codec Board and described in the PDF LillyGoSoftware.

The OLED Display board uses an I2C interface which requires the TwoWire Library which is already initialized for the Codec in the package. Several devices can use the same I2C interface as long as they have different "addresses". Note also that the Display's four I2C interface lines were rewired to the Sparkfun Qwiic connector standard.

Here is a listing of code specific to the OLED Display found in the **task.cpp** file.

```
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~OLED FUNCTIONS - ADAFRUIT LIB~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#define SCREEN_WIDTH 128 // OLED display width, in pixels
#define SCREEN_HEIGHT 64 // OLED display height, in pixels

// Declaration for an SSD1306 display connected to I2C (SDA, SCL pins)

#define OLED_RESET -1 // Reset pin # (or -1 if sharing Arduino reset pin)
#define SCREEN_ADDRESS 0x3C  // 0x3D for 128x64, 0x3C for 128x32

Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~OLED DISPLAY TASK~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
/*************************************************************************
 This is an example for our Monochrome OLEDs based on SSD1306 drivers
 The Yellow/Blue Display has a fixed upper banner of yellow
 and main lower screen of blue pixels.

 Pick one up today in the adafruit shop!
 ------> http://www.adafruit.com/category/63_98

 This example is for a 128x64 pixel display using I2C to communicate
 3 pins are required to interface (two I2C and one reset).

 Adafruit invests time and resources providing this open
 source code, please support Adafruit and open-source
 hardware by purchasing products from Adafruit!

 Written by Limor Fried/Ladyada for Adafruit Industries,
 with contributions from the open source community.
 BSD license, check license.txt for more information
 All text above, and the splash screen below must be
 included in any redistribution.

*************************************************************************/

void displaytask(void* arg)
{
  // SSD1306_SWITCHCAPVCC = generate display voltage from 3.3V internally

  if(!display.begin(SSD1306_SWITCHCAPVCC, SCREEN_ADDRESS)) {
    Serial.println(F("SSD1306 allocation failed"));
    for(;;); // Don't proceed, loop forever
  }
    display.invertDisplay(true);
    // True will do black text on yellow/blue background.
    // False will do yellow/blue text on black background.

    display.setTextColor(SSD1306_WHITE); // Draw white(yellow/blue)text
    display.setTextSize(2);              // Draw 2X-scale text

while(true)  //Using OLED Display Library Functions
{
  //testdrawline();   // Draw many lines, from Library Examples
  //testdrawrect();   // Draw rectangles (outlines), from Library Examples

  display.clearDisplay();
  display.setCursor(0,0); // Start at top-left corner

  display.println("Thrsh/Dst"); // Displays in top yellow "banner" region
  display.print(""); display.print(myPedal->threshold);
  display.print(" "); display.println(myPedal->distortion);

  display.print(" Mix "); display.print(myPedal->mix);

  display.display();

  vTaskDelay(500);  // Update only every 1/2 second

} // End of while(1)
} // End of task
```

This particular Display board seems to only work with an I2C Screen Address of 0x3C (`#define SCREEN_ADDRESS 0x3C`) in spite of the Library directions to use 0x3D for a 128x64 pixel display, and in spite of the other two addresses printed on the back of the board.

The yellow and blue pixels for this particular display board are not programmable. The yellow pixels are fixed at the top quarter of the screen as a kind of banner. The lower three quarters of the display are fixed with blue pixels.

After the code **#define** labels, the first order of business is to declare an instance object of the Library Display Class **Adafruit_SSD1306** and call the object "**display**". Once this is done the **display** object can access all the Display Library's available class methods using the dot operator, such as **display.print( )**.

The Library includes several fun example sketches that show off its graphic display capabilities. The display functions found in the examples files can be defined at this point in **task.cpp** before the main display task. These functions make liberal use of the GFX library display methods using the "**display.**method" dot operator. The only necessary change to these functions is to replace all the Arduino **delay( )** commands with the non-blocking RTOS command **vTaskDelay( )**.

Finally, the main **displaytask( )** function is defined in the **task.cpp** file. It starts off with the **display.begin( )** method from the Adafruit Library. Then after a few display setup calls, the function enters an infinite **while(1)** loop to continually display a number of the effects program variables defined in **set_module.cpp** and accessed with the **myPedal->** class module object and its pointer operator.

Setting the text size to "2" (`display.setTextSize(2);`) allows for 4 lines of text on this small .9" display. The Adafruit Library coders have done a great job in making it easy to display text by using the familiar Arduino Serial functions **print( )** and **println( )**. Note the RTOS delay line `vTaskDelay(500)` which sets the display update rate at about 1/2 a second. During this half second delay other tasks, created in the file **task.cpp** and initiated in **taskSetup( )**, can execute their own code, an indispensable feature of the **FreeRTOS** real time operating system.

```
void taskSetup()
{

//decoding button presses and other digital sensors
xTaskCreatePinnedToCore(buttontask, "buttontask", 4096, NULL,
AUDIO_PROCESS_PRIORITY, NULL,0);

//decoding potentiometer and other analog sensors
```

```
xTaskCreatePinnedToCore(controltask, "controltask", 4096, NULL,
AUDIO_PROCESS_PRIORITY, NULL,0);

//audio frame monitoring task used by systemMonitor
xTaskCreatePinnedToCore(framecounter_task, "framecounter_task", 4096, NULL,
AUDIO_PROCESS_PRIORITY, NULL,0);

//Neo Pixel Task originally for LillyGo TAudio board with 19 LEDs
//xTaskCreatePinnedToCore(neopixeltask, "NeoPixeltask", 4096, NULL, 5,
NULL,0);

//OLED Display Task for SSD1306 0.96" OLED Display Module, Adafruit Library
xTaskCreatePinnedToCore(displaytask, "OLED_Displaytask", 4096, NULL, 5,
NULL,0);

}
```

NeoPixel Software

The Neo Pixel Task was commented out (disabled) in the **taskSetup( )** shown immediately above. However, the Sparkfun Thing Plus ESP32 does include one on-board Neo-Pixel. The NeoPixel programming, described in the PDF LillyGoSoftware, operates similarly to the OLED Display, with its own Library and task functions. If you want to un-comment this task, the one NeoPixel is programmed to change color with one of the effects controls.

Codec Speakers

The WM8960 includes differential stereo speaker outputs for a left and right speaker. The Codec amplifier is rated as a class D speaker driver and can drive 1W into 8Ω speakers.

For users looking to power the speakers with a separate power supply, you can cut the jumper between the pads labeled as SPKVDD and VIN on the back of the board and connect your own power supply to SPKVDD. In this enclosure, SPKVDD, power for the Modem Speakers, comes from VIN which in turn originates from 5 volts on the USB cable connected to the Sparkfun Thing Plus ESP32 board.

Class D speaker amplifiers are highly efficient designs. They use modulated high frequency pulses (705.6kHz in this case) to create the audio out. The inductance of the speaker coil becomes part of the class D amplifier's output filter to get rid of the high frequency modulating signal, leaving only the audio. This means that only a passive coil speaker should be connected to the Codec's Speaker outputs, and specifically, it

must be an 8 Ohm Speaker.

The Codec Speaker Outputs can easily be enabled with two Codec methods:

```
codec.enableSpeakers();  // disableSpeakers( ) to disable
codec.setSpeakerVolumeDB(0.00); // Valid dB settings are -74.0 up to +6.0
```

These can be placed in the **codec_sets( )** function in the file **set_codec.cpp**. The following separate controls are also available:

```
codec.enableLeftSpeaker( )
codec.disableLeftSpeaker( )
codec.enableRightSpeaker( )
codec.disableRightSpeaker( )
```

Sample Rates

The usual sample rate is 44.1KHz as declared in **set_settings.h** with the line

```
#define SAMPLE_RATE   (44100)
```

The codec must be set up with this rate and that happens in **set_codec.cpp** inside the function **codec_sets( )** with this code:

```
// CLOCK STUFF, These settings will get you 44.1KHz sample rate,
// and class-d speaker amp modulation rate at 705.6kHz

codec.enablePLL(); // Needed for class-d amp clock
codec.setPLLPRESCALE(WM8960_PLLPRESCALE_DIV_2);
codec.setSMD(WM8960_PLL_MODE_FRACTIONAL);
codec.setCLKSEL(WM8960_CLKSEL_PLL);
codec.setSYSCLKDIV(WM8960_SYSCLK_DIV_BY_2);
codec.setBCLKDIV(4);
codec.setDCLKDIV(WM8960_DCLKDIV_16);
codec.setPLLN(7);
codec.setPLLK(0x86, 0xC2, 0x26); // PLLK=86C226h

//codec.setADCDIV(0); // Default, 000 (what we need for 44.1KHz)
//codec.setDACDIV(0); // Default, 000 (what we need for 44.1KHz)
codec.setWL(WM8960_WL_16BIT);
```

As you can see, the set up is rather complicated since other clocks are being set up along with the ADC/DAC sample rate. However, the WM8960 Datasheets have a couple useful tables to help figure it all out -- Table 40 (pg 58) and Table 45 (pg 61). Some instructions on how a 44.1KHz sample rate is set up can be found in the file **codec.cpp**.

Be aware that the Master Clock (MCLK) for the Sparkfun Codec board is 24MHz from an external hardware clock on the board. This is the master clock from which all the other system clocks, including the Sample Rate, are derived.

Sometimes a lower Sample Rate is useful, to get longer delay times for echo effects or more time to perform involved effects calculations, for instance. The following function was created to offer a choice between 44.1KHz and 32KHz sample rates, to be placed in the file **set_codec.cpp** and used in the function **codec_sets( )** replacing the code shown above.

```
void setWM8960SampleRate()
 {
   // These settings will get you 44.1KHz or 32KHz sample rate,
   // as set by the label SAMPLE_RATE from set_settings.h
   // and class-d speaker amp modulation rate at 705.6kHz

  codec.enablePLL(); // PhaseLockLoop
  codec.setPLLPRESCALE(WM8960_PLLPRESCALE_DIV_2);
  codec.setSMD(WM8960_PLL_MODE_FRACTIONAL);
  codec.setCLKSEL(WM8960_CLKSEL_PLL);
  codec.setSYSCLKDIV(WM8960_SYSCLK_DIV_BY_2);
  codec.setBCLKDIV(4);
  codec.setDCLKDIV(WM8960_DCLKDIV_16);

  if(SAMPLE_RATE == 41000){  // From set_settings.h
      codec.setPLLN(7);
      codec.setPLLK(0x86, 0xC2, 0x26); // PLLK=86C226h, SYSCLK=11.2896MHz
      codec.setADCDIV(0); // default = 1
      codec.setDACDIV(0); // default = 1
  }
  else if(SAMPLE_RATE == 32000){  // From set_settings.h
      codec.setPLLN(8);
      codec.setPLLK(0x31, 0x26, 0xE8); // PLLK=3126E8, SYSCLK=12.288MHz
      codec.setADCDIV(1); // 1.5
      codec.setDACDIV(1); // 1.5
  }
  codec.setWL(WM8960_WL_16BIT);

 }  // End of setWM8960SampleRate()
```

Trill Touch Pad

A Trill Touch Pad was installed on the right side of the Radio Enclosure. Trill sensors are made by Bella, a "maker platform for creating beautiful interaction". There is a Trill GitHub Library available for the Arduino. These sensors work through an I2C

interface.

So far, I haven't yet incorporated this sensor into my Effects Software Platform. This PDF will be updated later. For now I'm just collecting the available programming resources.

Software Effects

The software used here is from a complete Codec/ESP32 Package described on my website at https://www.jtalbert.xyz/ESP32/ Please read the PDF files from the site for more information: Codec Software and SparkfunCodec.

Most of the Effects programming will happen in the files **main.cpp**, **set_module.cpp** and **set_module.h**.

The **set_settings.h** file that holds all the **#define** labels like GPIO pin assignments for the Sparkfun ESP32/Codec PCB is shown below. It is applied in all the example effects to be discussed.

```
#ifndef SETTINGS_H_
#define SETTINGS_H_

  #pragma once
  #include "codec.h"
  #include <Arduino.h>
  #include "driver/i2s.h"

  #define SAMPLE_RATE     (44100)
  #define BITS_PER_SAMPLE (16)
  #define CHANNEL_COUNT 2

  //Sparkfun Codec/ESP32 Thing Plus C PIN ASSIGNMENTS
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~

  #define POT1 14
  #define POT2 32
  #define POT3 39
  #define POT4 36
  #define POT5 33
  #define POT6 34

  #define LED1 13  //onboard Blue LED, ESP32 Sparkfun Thing Plus
  #define LED2 12

  #define KEY1 15
  #define KEY2 27

  #define TOUCH_THRESHOLD 30  //for touch switches
```

```
//ESP32-Codec I2S PIN SETUP
#define I2S_NUM               (0)
//#define IS2_MCLK_PIN (0) //onboard Osc Chip, MCLK of 24MHz, on Sparkfun
Codec
#define I2S_BCLK    (4)   //BCLK, SCK, SCLK
#define I2S_LRC     (25)  //LRC, WS, ADCLRC, DACLRC, LRCLK  -- Left/Right
channel indicator
#define I2S_DIN     (35)  //DIN, ADCDAT, SD --  data into ESP32 from ADC
output
#define I2S_DOUT    (26)  //DOUT, DACDAT, SDO -- data out of ESP32 to DAC
input

// I2C address (7-bit format for Wire library)
//#define WM8960_ADDR 0x1A  //left on codec.h
// I2C on Qwiic Connector
#define Codec_SDA      21
#define Codec_SCK          22
#define I2C_MASTER_SCL_IO 22
#define I2C_MASTER_SDA_IO 21
#define I2C_SDA        21
#define I2C_SCL        22

#define I2C_MASTER_NUM 1    //I2C port number for master dev
#define I2C_MASTER_FREQ_HZ 100000
#define I2C_MASTER_TX_BUF_DISABLE 0
#define I2C_MASTER_RX_BUF_DISABLE 0

//SD Card Reader Settings

#define SD_CARD_CS   5
#define SD_CARD_MISO 19
#define SD_CARD_MOSI 23
#define SD_CARD_CLK  18

#define SAMPLES_BUFFER_SIZE 1024

//NEO PIXEL SETTINGS
#define PIN 2  //Built in RGB on ESP32 Thing Plus
#define NUM_LEDS 1
#define BRIGHTNESS 5

//audio processing frame length in samples (L+R) 64 samples (32R+32L) 256
Bytes
//Used as size of i2s input and output buffers
#define FRAMELENGTH    256
//audio processing priority
#define AUDIO_PROCESS_PRIORITY  10

//SRAM used for DMA = DMABUFFERLENGTH * DMABUFFERCOUNT * BITS_PER_SAMPLE/8
* CHANNEL_COUNT
//Lower number for low latency, Higher number for more signal processing
time
//Must be value between 8 and 1024 in bytes
#define DMABUFFERLENGTH 128

//number of above DMA Buffers of DMABUFFERLENGTH
#define DMABUFFERCOUNT  8

  // processor timing variables for system monitor, also included in
```

task.cpp

```
  extern unsigned int runningTicks;
  extern unsigned int usedticks;
  extern unsigned int availableticks;
  extern unsigned int availableticks_start;
  extern unsigned int availableticks_end;
  extern unsigned int usedticks_start;
  extern unsigned int usedticks_end;
  extern unsigned int processedframe;
  extern unsigned int audiofps;

 void I2S_init(void);

#endif
```

Stereo Chorus Effect

This effect was copied from the Deeptronics/Blaskstomp project upon which my Effects Software Package was based. It uses the **oscillator** class and the **fractionalDelay** class from the DSP file, **bsdsp.cpp**.

set_module.h

```
#ifndef MODULE_H_
#define MODULE_H_

#include "controller_mod.h"
#include "bsdsp.h"

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~ DSP Class Declarations (bsdsp files) ~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 extern fractionalDelay delay1;
 extern fractionalDelay delay2;
 //extern bool x;
 //extern bool y;
  extern oscillator lfo1;
  extern oscillator lfo2;

//Create a child class derived from controllerModule
//The controller_module sets up all Pot, Switch, and LED pin, mode, and
actions

class controller_module:public controllerModule
{
  public:
  float depth;
  float freq;
  float beatFrequency;
  float phaseDiff;
  bool asynch;
```

```
  bool stereo;

  void init();
  void onButtonChange(int buttonIndex);
  void onControlChange(int controlIndex);
};

//controller_module myPedal declaration with extern
extern controller_module *myPedal ;

#endif
```

Note the `#include "bsdsp.h` line. Two instances of the DSP **fractionalDelay** Class and two instances of the DSP **oscillator** Class are declared -- **delay1**, **delay2**, **lfo1**, **lfo2**. Four controller variables are declared for **depth**, **freq**, **beatFrequency**, and **phaseDiff**. Two boolean switch variables are declared for **asynch** and **stereo**. As you can see, just this short header file lists all the basic controller components of the effect.

set_module.cpp

```
#include "set_module.h"
#include "set_settings.h"
#include "set_codec.h"

//controller_module myPedal definition
controller_module *myPedal = new controller_module();

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~ DSP Class Definitions (bsdsp files) ~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  fractionalDelay delay1;
  fractionalDelay delay2;
  bool x = delay1.init(3); //init for 3 ms delay
  bool y = delay2.init(3); //init for 3 ms delay
  oscillator lfo1;
  oscillator lfo2;

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~ CONTROLLER MODULE CLASS DEFINITIONS ~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Define the controllerModule functions declared in set_module.h
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::init()  //effect module class initialization
 {
  name = "Stereo Chorus";

  // Set up pin Modes for the switches and LEDs
  // For mode details, see control_task() and button_task() in task.cpp
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(KEY1, INPUT_PULLUP);  //internal pullup
```

```
  pinMode(KEY2, INPUT_PULLUP);

  //setting up the buttons
  button[0].name = "mute";
  button[0].mode = BM_MOMENTARY;
  button[0].touch = false;
  button[0].pin = KEY1;

  button[1].name = "LED";
  button[1].mode = BM_TOGGLE;
  button[1].touch = false;
  button[1].pin = KEY2;

  //add gain control
  control[0].name = "Rate";
  control[0].mode = CM_POT;
  control[0].levelCount = 128;
  control[0].pin = POT1;

  //add range control
  control[1].name = "Depth";
  control[1].mode = CM_POT;
  control[1].levelCount = 128;
  control[1].pin = POT2;

  control[2].name = "F/P Diff";
  control[2].mode = CM_POT;
  control[2].levelCount = 128;
  control[2].pin = POT3;

  control[3].name = "Input Mode";
  control[3].mode = CM_SELECTOR;
  control[3].levelCount = 2;  //0:mono 1:stereo
  control[3].pin = POT4;

  control[4].name = "Sync Mode";
  control[4].mode = CM_SELECTOR;
  control[4].levelCount = 2;
  control[4].pin = POT5;


  freq=5;
  depth=0.5;
  beatFrequency=2.5;
  stereo = 1;
  asynch = 1;
  lfo1.setFrequency(freq);
  lfo2.setFrequency(freq+beatFrequency);
 }
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::onButtonChange(int buttonIndex)
{
  switch(buttonIndex)
  {
    case 0: //main button state has changed
    {
      if(button[0].value) //if effect is activated
      {
        //codec.analogBypass(false);
        codec.disableDacMute();
```

```
        digitalWrite(LED1, HIGH);
      }
      else //if effect is bypassed
      {
        //codec.analogBypass(true);
        codec.enableDacMute();
        digitalWrite(LED1, LOW);
      }
      break;
    }
    case 1: //the button[1] state has changed
    {
      if(button[1].value) // just test LED and Switch
      {digitalWrite(LED2, HIGH);}
      else
      {digitalWrite(LED2, LOW);}
      break;
    }
  }
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::onControlChange(int controlIndex)
{
  switch(controlIndex)
  {
    case 0: //rate
    {
      freq = 0.5 + 10 * (float)control[0].value/127.0;
      lfo1.setFrequency(freq);
      lfo2.setFrequency(freq + beatFrequency);
      break;
    }
    case 1:  //depth
    {
      depth = 1.49 * (float)control[1].value/127.0;
      break;
    }
    case 2:  //phase or frequency difference
    {
      beatFrequency = 5 * (float)control[2].value/127.0;
      phaseDiff = (float)control[2].value;
      lfo2.setFrequency(freq + beatFrequency);
      break;
    }
    case 3:  //depth
    {
      stereo = (bool)control[3].value;
      break;
    }
    case 4:  //depth
    {
      asynch = (bool)control[4].value;
      break;
    }
 }
}
```

The first thing accomplished in the **set_module.cpp** file is the creation of instances for all the Classes used in the Effect. A pointer to **myPedal** is created, the main instance

object of the **controller_module** child class. **delay1** and **delay2** are instance objects of the **fractionalDelay** DSP class. **lfo1** and **lfo2** (low frequency oscillators) are instance objects of the **oscillator** DSP class. Both delay instances are initialized with **delay1.init(3)** and **delay2.init(3)**. This will create buffers that hold 3 milliseconds of samples given the defined **SAMPLE_RATE**. The number of samples in buffer = (samples per second) * (0.003 seconds)). These **init( )** methods return boolean **true** if the buffer build was successful.

Next, the **controller_module** child **init( )** method is defined, to be executed later in the main loop of **main.cpp** with the line `myPedal->init( )`. Here **pinMode**s are set up for two switches and two LEDs. Then the control properties required for two switches and five pots are configured. Finally, all the new variables used to hold the pot and switch values are defined and given initial values. At this point we can also assign some of these variables to the inputs of some DSP methods. The **setFrequency( )** method of the **lfo1 oscillator** class object will get its frequency from the variable **freq**. The other low frequency **oscillator, lfo2**, will get a slightly higher frequency, **freq+beatFrequency**.

One pushbutton is set up in the **controller_module** method **onButtonChange( )** to either enable the Chorus Effect or bypass it and indicate which with an LED. Another switch is just turns on or off an LED.

The action of 5 pots are configured in the **controller_module** method **onControlChange( )**. Most of them just transfer the pot value to one of the variables defined above after a bit of mathematical adjustments. These variable values will then be used in the main effects code loop. A couple of the pot values are used right away to set the frequency of the low frequency **oscillator** objects using the **oscillator** class **setFrequency( )** method.

main.cpp

```
#include <Arduino.h>
#include "set_settings.h"
#include "set_module.h"
#include "set_codec.h"
#include "task.h"
#include <SD.h>
#include "sd_play.h"

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~SETUP~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup()
{
  Serial.begin(115200);
  while(!Serial);
```

```
  delay(3000);

  //~~~~~~~~~~~codec is initialized  See Codec.cpp~~~~~~~~~~~~~~~~
  //~~~~i2c is initialized within codec.init() with initI2C()~~~~~~

        Wire.begin();
        Serial.println("Initialize Codec Codec ");
        codec.begin();
        codec_sets();
        Serial.println("Codec Init success!!");

  //~~~~~~I2S  See set_settings.cpp for I2S ~~~~~~~~~~~~~~~

        I2S_init();

  //~~~~~~~~~~~~~~Monitor (can be commented out)~~~~~~~~~~

        Serial.println("I2S/SD setup complete");
        runSystemMonitor();  //for testing only

} //Setup End


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~MAIN LOOP~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop()
{
    size_t readsize = 0;
    int16_t rxbuf[FRAMELENGTH], txbuf[FRAMELENGTH]; //128 L+R signed 16 bit
samples
    float rxl, rxr, txl, txr;  //left and right single samples, processed
as floats

    myPedal->init();
    taskSetup();

  while(1){   //signal processing loop

  setDebugVars(myPedal->depth, myPedal->freq, myPedal->phaseDiff,
myPedal->beatFrequency);

  /*
   read 256 samples = FRAMELENGTH (128 Left+Right signed samples). It's
also the size of buffers.
   read 2 bytes for each 16 bit (2 byte) sample (FRAMLENGTH*2)
   rxbuf[] and txbuf[] defined with signed 16 bit integers (int16_t) and of
FRAMELENGTH size.
 */
  //gather some input samples into receive buffer from the DMA memory,
  i2s_read(I2S_NUM_0, rxbuf, FRAMELENGTH*2, &readsize, 20);

  for (int i=0; i<(FRAMELENGTH); i+=2) {  //process samples one at a time
from buffers

    rxl = (float) (rxbuf[i]) ;   //convert sample to float
    rxr = (float) (rxbuf[i+1]) ;
```

```
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    //~~~~~~~~~~stereoChorus Processing~~~~~~~~~~~~
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    delay1.write(rxl);
    delay2.write(rxr); //write anyway, no matter it's stereo or mono input

    lfo1.update();
    lfo2.update();
    float dt1 = (1 + lfo1.getOutput())* myPedal->depth;
    float dt2;
    if(myPedal->asynch == 0) //asynchronous
      dt2 = (1 + lfo2.getOutput())* myPedal->depth;
    else  //synchronous
      dt2 = (1 + lfo1.getOutput(myPedal->phaseDiff))* myPedal->depth;

    txl = (0.7 * rxl) + (0.7 * delay1.read(dt1));
    if(myPedal->stereo) //if stereo input
      txr = (0.7 * rxr) + (0.7 * delay2.read(dt2));
    else //if mono
      txr = (0.7 * rxl) + (0.7 * delay1.read(dt2));
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    txbuf[i]   = ((int16_t) txl) ; //convert sample back to integer
    txbuf[i+1] = ((int16_t) txr) ;
  }
  // play processed receive buffer by loading transmit buffer into DMA
memory
  i2s_write(I2S_NUM_0, txbuf, FRAMELENGTH*2, &readsize, 20);

} // End of while(1) loop
} // End of Main Loop
```

The Stereo Chorus Effect processing is found in the middle of the inner **while(1)** loop within the main **loop( )**. Here is a general description of what's going on there.

The code first loads the input signal sample into the two circular delay buffers. The index, **dt1** and **dt2**, into each of these delay buffers determines the amount of delay. The two low frequency oscillator outputs multiplied by the **depth** control are applied to the two delay buffer indices. This results in an oscillating amount of delay in the two delay lines, one oscillating a bit faster than the other. Finally, the output samples are generated as an equal mix of the original signal samples and the delayed samples, the left channel given a different delay from the right.

One **if/else** section sets up a stereo or mono output depending on the boolean value "**stereo**". Another **if/else** section sets up a different **dt2** delay index calculation depending on the boolean value "**asynch**".

Stereo Chorus on SD Playback

This effect uses a modified version of the the Stereo Chorus effect described above and applies it to an audio WAV file recorded onto an SD card instead of a Codec audio Input. The Sparkfun Thing Plus ESP32 includes an SD card slot on the bottom of the circuit board just below the USB port.

Here the "**stereo**" and "**asynch**" switches are dropped from the effects processing and the **frequency** and **depth** control values are widened for more intense effects.

Playback from an SD card was first introduced and described in the project PDF LillyGo. For this capability, the Effects Software Package was expanded with new classes in the files **sd_play.h** and **sd_play.cpp**.

set_module.h

```
#ifndef MODULE_H_
#define MODULE_H_

#include "controller_mod.h"
#include "bsdsp.h"

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~ DSP Class Declarations (bsdsp files) ~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 extern fractionalDelay delay1;
 extern fractionalDelay delay2;
 //extern bool x;
 //extern bool y;
  extern oscillator lfo1;
  extern oscillator lfo2;

//Create a child class derived from controllerModule
//The controller_module sets up all Pot, Switch, and LED pin, mode, and
actions

class controller_module:public controllerModule
{
  public:
  float depth;
  float freq;
  float beatFrequency;
  float pan;

  void init();
  void onButtonChange(int buttonIndex);
  void onControlChange(int controlIndex);
};

//controller_module myPedal declaration with extern
extern controller_module *myPedal ;
```

```
#endif
```

set_module.cpp

#include "set_module.h"

```
#include "set_settings.h"
#include "set_codec.h"

//controller_module myPedal definition
controller_module *myPedal = new controller_module();

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~ DSP Class Definitions (bsdsp files) ~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  fractionalDelay delay1;
  fractionalDelay delay2;
  bool x = delay1.init(3); //init for 3 ms delay
  bool y = delay2.init(3); //init for 3 ms delay
  oscillator lfo1;
  oscillator lfo2;

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~ CONTROLLER MODULE CLASS DEFINITIONS ~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Define the controllerModule functions declared in set_module.h
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::init()  //effect module class initialization
 {
  name = "Stereo Chorus";

  // Set up pin Modes for the switches and LEDs
  // For mode details, see control_task() and button_task() in task.cpp
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(KEY1, INPUT_PULLUP);  //internal pullup
  pinMode(KEY2, INPUT_PULLUP);

  //setting up the buttons
  button[0].name = "mute";
  button[0].mode = BM_MOMENTARY;
  button[0].touch = false;
  button[0].pin = KEY1;

  button[1].name = "LED";
  button[1].mode = BM_TOGGLE;
  button[1].touch = false;
  button[1].pin = KEY2;

  //add gain control
```

```
  control[0].name = "Rate";
  control[0].mode = CM_POT;
  control[0].levelCount = 128;
  control[0].pin = POT1;

  //add range control
  control[1].name = "Depth";
  control[1].mode = CM_POT;
  control[1].levelCount = 128;
  control[1].pin = POT2;

  control[2].name = "Pan";
  control[2].mode = CM_POT;
  control[2].levelCount = 128;
  control[2].pin = POT3;

  freq=5;
  depth=0.5;
  pan=0.5;
  beatFrequency=2.5;
  lfo1.setFrequency(freq);
  lfo2.setFrequency(freq+beatFrequency);
 }
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::onButtonChange(int buttonIndex)
{
  switch(buttonIndex)
  {
    case 0: //main button state has changed
    {
      if(button[0].value) //if effect is activated
      {
        //codec.analogBypass(false);
        codec.disableDacMute();
        digitalWrite(LED1, HIGH);
      }
      else //if effect is bypassed
      {
        //codec.analogBypass(true);
        codec.enableDacMute();
        digitalWrite(LED1, LOW);
      }
      break;
    }
    case 1: //the button[1] state has changed
    {
      if(button[1].value) // just test LED and Switch
      {digitalWrite(LED2, HIGH);}
      else
      {digitalWrite(LED2, LOW);}
      break;
    }
  }
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::onControlChange(int controlIndex)
{
  switch(controlIndex)
  {
    case 0: //rate
```

```
    {
      freq = 0.5 + 10 * (float)control[0].value/127.0;
      lfo1.setFrequency(freq);
      lfo2.setFrequency(freq + beatFrequency);
      break;
    }
    case 1:  //depth
    {
      depth = 1.49 * (float)control[1].value/127.0;
      break;
    }
    case 2:  //pan
    {
      pan = (float)control[2].value/127.0;
      break;
    }

 }
}
```

main.cpp

```
#include <Arduino.h>
#include "set_settings.h"
#include "set_module.h"
#include "set_codec.h"
#include "task.h"
#include <SD.h>
#include "sd_play.h"

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

SDplay mySDplay; //create an instance of SDplay class

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~SETUP~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup()
{
  Serial.begin(115200);
  while(!Serial);
  delay(3000);

  //~~~~~~~~~~~codec is initialized  See Codec.cpp~~~~~~~~~~~~~~~~~
  //~~~~i2c is initialized within codec.init() with initI2C()~~~~~~

        Wire.begin();
        Serial.println("Initialize Codec Codec ");
        codec.begin();
        codec_sets();
        Serial.println("Codec Init success!!");

  //~~~~~~I2S  See set_settings.cpp for I2S ~~~~~~~~~~~~~~~

        I2S_init();

         mySDplay.SDCardInit();
```

```
        mySDplay.OpenWaveFile();

  //~~~~~~~~~~~~~~~Monitor (can be commented out)~~~~~~~~~~

        Serial.println("I2S/SD setup complete");
        runSystemMonitor();  //for testing only

} //Setup End


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~MAIN LOOP~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop()
{
    size_t readsize = 0;
    byte txbuf[SAMPLES_BUFFER_SIZE]; //128 L+R signed 16 bit samples
    float rxl, rxr, txl, txr;  //left and right single samples, processed
as floats

    myPedal->init();
    taskSetup();

  while(1)
  {   //signal processing loop

    setDebugVars(myPedal->depth, myPedal->freq, myPedal->pan,
myPedal->beatFrequency);

    //gather some input samples into Samples buffer from the SD wavfile,
      mySDplay.ReadFile(mySDplay.Samples);

    for (int i=0; i<(SAMPLES_BUFFER_SIZE); i+=4)  //process samples one at
a time from Samples[]
    {
       rxl = (float)((int16_t)(mySDplay.Samples[i+1] << 8) |
mySDplay.Samples[i]);   // Left sample float
       rxr = (float)((int16_t)(mySDplay.Samples[i+3] << 8) |
mySDplay.Samples[i+2]); // Right sample float

      //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      //~~~~~~~~~~stereoChorus Processing~~~~~~~~~~~~~
      //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      delay1.write(rxl);
      delay2.write(rxr);

      lfo1.update();
      lfo2.update();

      float dt1 = (1 + lfo1.getOutput())* myPedal->depth;
      float dt2 = (1 + lfo2.getOutput())* myPedal->depth;

        txl = ((1 - myPedal->pan) * rxl) + (myPedal->pan *
delay1.read(dt1));
        txr = ((1 - myPedal->pan) * rxr) + (myPedal->pan *
delay2.read(dt2));
      //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
        txbuf[i]   = ((int16_t) txl) & 0xff ;  // Left sample loaded as two
bytes
        txbuf[i+1] = ((int16_t) txl) >> 8;
        txbuf[i+2] = ((int16_t) txr) & 0xff ;  // Right sample loaded as
two bytes
        txbuf[i+3] = ((int16_t) txr) >> 8;

    } // End of for loop

     // play processed transmit buffer by loading txbuf into DMA memory
    mySDplay.FillI2SBuffer(txbuf);

  } // End of while(1) loop
}   // End of Main Loop
```

Stereo Echo with Feedback

This is a basic Echo Delay with separate delay and mix controls on each stereo channel and one feedback control for both left and right.

The **fractionalDelay** DSP class was used with two instantiated objects, **delay1** and **delay2**, each given a buffersize of 260msec, which is the maximum delay time for the echo. The ESP32 has limited memory space available for these two delay buffers. Basically, if you go over the limit the effects program will crash, exhibited by continually restarting. The **SAMPLE_RATE** was lowered to 32KHz to allow for larger maximum delays. How this was accomplished is described in a previous section.

The 0 to 256ms delay times for each channel originate directly from two control pots. There seems to be some jitter in these pot readings causing some output noise. To minimize this noise, the delay pot readings are entered from a pushbutton switch instead of being continuously variable.

The original Radio Enclosure idea was to connect a volume controlled Codec Input3 to the headphones output and use it for effect feedback. This turned out to be much less responsive than the calculated control pot feedback and thus needs further exploration.

I've included the OLED Display task in the code below. It will display all 5 control values updated every half second - **delay1**, **mix1**, **delay2**, **mix2**, **feedback**.

set_module.h

```
#ifndef MODULE_H_
#define MODULE_H_
```

```
#include "controller_mod.h"
#include "bsdsp.h"

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~ DSP Class Declarations (bsdsp files) ~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 extern fractionalDelay delay1;
 extern fractionalDelay delay2;

//Create a child class derived from controllerModule
//The controller_module sets up all Pot, Switch, and LED pin, mode, and
actions

class controller_module:public controllerModule
{
  public:
  float delay_mix1;
  float delay_mix2;
  int delay_time1;
  int delay_time2;
  int dt1;
  int dt2;
  float feedback;

  void init();
  void onButtonChange(int buttonIndex);
  void onControlChange(int controlIndex);
};

//controller_module myPedal declaration with extern
extern controller_module *myPedal ;

#endif
```

set_module.cpp

```
#include "set_module.h"
#include "set_settings.h"
#include "set_codec.h"

//controller_module myPedal definition
controller_module *myPedal = new controller_module();

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~ DSP Class Definitions (bsdsp files) ~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  fractionalDelay delay1;
  fractionalDelay delay2;
  bool x = delay1.init(260); //init for delay
  bool y = delay2.init(260); //init for delay

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~ CONTROLLER MODULE CLASS DEFINITIONS ~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Define the controllerModule functions declared in set_module.h
```

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::init()  //effect module class initialization
 {
  name = "Echo Reverb";

  // Set up pin Modes for the switches and LEDs
  // For mode details, see control_task() and button_task() in task.cpp
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(KEY1, INPUT_PULLUP);  //internal pullup
  pinMode(KEY2, INPUT_PULLUP);

  //setting up the buttons
  button[0].name = "mute";
  button[0].mode = BM_MOMENTARY;
  button[0].touch = false;
  button[0].pin = KEY1;

  button[1].name = "Enter Delay";
  button[1].mode = BM_MOMENTARY;
  button[1].touch = false;
  button[1].pin = KEY2;

  //add gain control
  control[0].name = "DelayMix1";
  control[0].mode = CM_POT;
  control[0].levelCount = 128;
  control[0].pin = POT1;

  //add range control
  control[1].name = "DelayTime1";
  control[1].mode = CM_POT;
  control[1].levelCount = 256;
  control[1].pin = POT2;

  control[2].name = "DelayMix2";
  control[2].mode = CM_POT;
  control[2].levelCount = 128;
  control[2].pin = POT3;

  control[3].name = "DelayTime2";
  control[3].mode = CM_POT;
  control[3].levelCount = 256;
  control[3].pin = POT4;

  control[5].name = "Feedback";
  control[5].mode = CM_POT;
  control[5].levelCount = 128;
  control[5].pin = POT6;

  delay_mix1 = 0.5;
  delay_mix2 = 0.5;
  delay_time1 = 0;
  delay_time2 = 0;
  feedback = 0;
 }
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::onButtonChange(int buttonIndex)
{
  switch(buttonIndex)
```

```
  {
    case 0: //main button state has changed
    {
      if(button[0].value) //if effect is activated
      {
        //codec.analogBypass(false);
        codec.disableDacMute();
        digitalWrite(LED1, HIGH);
      }
      else //if effect is bypassed
      {
        //codec.analogBypass(true);
        codec.enableDacMute();
        digitalWrite(LED1, LOW);
      }
      break;
    }
    case 1: //the button[1] state has changed
    {
      if(button[1].value) // enter delay times on button press
      {digitalWrite(LED2, HIGH); dt1=delay_time1; dt2=delay_time2; }
      else
      {digitalWrite(LED2, LOW);}
      break;
    }
  }
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::onControlChange(int controlIndex)
{
  switch(controlIndex)
  {
    case 0: //delay mix1
    {
      delay_mix1 = (float)control[0].value/127.0;
      break;
    }
    case 1:  //delay time1
    {
      delay_time1 = control[1].value ;
      break;
    }
    case 2:  //delay mix2
    {
      delay_mix2 = (float)control[2].value/127.0;
      break;
    }
    case 3:  //delay time2
    {
      delay_time2 = control[3].value ;
      break;
    }
    case 5: //delay mix1
    {
      feedback = (float)control[5].value/127.0;
      break;
    }
 }
}
```

main.cpp

```
#include <Arduino.h>
#include "set_settings.h"
#include "set_module.h"
#include "set_codec.h"
#include "task.h"
#include <SD.h>
#include "sd_play.h"

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~SETUP~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup()
{
  Serial.begin(115200);
  while(!Serial);
  delay(3000);

  //~~~~~~~~~~~codec is initialized  See Codec.cpp~~~~~~~~~~~~~~~~~
  //~~~~i2c is initialized within codec.init() with initI2C()~~~~~~

        Wire.begin();
        Serial.println("Initialize Codec Codec ");
        codec.begin();
        codec_sets();
        Serial.println("Codec Init success!!");

  //~~~~~~I2S  See set_settings.cpp for I2S ~~~~~~~~~~~~~~~

        I2S_init();

  //~~~~~~~~~~~~~~~Monitor (can be commented out)~~~~~~~~~~

        Serial.println("I2S/SD setup complete");
        runSystemMonitor();  //for testing only

} //Setup End


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~MAIN LOOP~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop()
{
    size_t readsize = 0;
    int16_t rxbuf[FRAMELENGTH], txbuf[FRAMELENGTH]; //128 L+R signed 16 bit
samples
    float rxl, rxr, txl, txr;  //left and right single samples, processed
as floats

    myPedal->init();
    taskSetup();

  while(1){   //signal processing loop

  setDebugVars(myPedal->delay_mix1, myPedal->delay_time1,
myPedal->delay_mix2, myPedal->delay_time2);
```

```
  /*
   read 256 samples = FRAMELENGTH (128 Left+Right signed samples). It's
also the size of buffers.
   read 2 bytes for each 16 bit (2 byte) sample (FRAMLENGTH*2)
   rxbuf[] and txbuf[] defined with signed 16 bit integers (int16_t) and of
FRAMELENGTH size.
 */
  //gather some input samples into receive buffer from the DMA memory,
  i2s_read(I2S_NUM_0, rxbuf, FRAMELENGTH*2, &readsize, 20);

  for (int i=0; i<(FRAMELENGTH); i+=2) {  //process samples one at a time
from buffers

    rxl = (float) (rxbuf[i]) ;   //convert sample to float
    rxr = (float) (rxbuf[i+1]) ;

     //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    //~~~~~~Delay with Feedback Processing~~~~~~~~~
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    delay1.write( ((1 - myPedal->feedback) * rxl) + (myPedal->feedback *
txl) );
    delay2.write( ((1 - myPedal->feedback) * rxr) + (myPedal->feedback *
txr) );

    txl = ((1.0 - myPedal->delay_mix1) * rxl) + (myPedal->delay_mix1 *
delay1.read(myPedal->dt1));
    txr = ((1.0 - myPedal->delay_mix2) * rxr) + (myPedal->delay_mix2 *
delay2.read(myPedal->dt2));

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    txbuf[i]   = ((int16_t) txl) ; //convert sample back to integer
    txbuf[i+1] = ((int16_t) txr) ;
  }
  // play processed receive buffer by loading transmit buffer into DMA
memory
  i2s_write(I2S_NUM_0, txbuf, FRAMELENGTH*2, &readsize, 20);

} // End of while(1) loop
} // End of Main Loop
```

displaytask in task.cpp

```
void displaytask(void* arg)
{
  // SSD1306_SWITCHCAPVCC = generate display voltage from 3.3V internally
  if(!display.begin(SSD1306_SWITCHCAPVCC, SCREEN_ADDRESS)) {
    Serial.println(F("SSD1306 allocation failed"));
    for(;;); // Don't proceed, loop forever
  }

    display.invertDisplay(true);
    display.setTextColor(SSD1306_WHITE); // Draw white text
    display.setTextSize(2);             // Draw 2X-scale text

while(true)  //OLED Display functions below defined above in task.cpp
```

```
{
  display.clearDisplay();
  display.setCursor(0,0);             // Start at top-left corner

  display.println(" Mix/Delay"); // display in top yellow "banner" region
  display.print(" "); display.print(myPedal->delay_mix1);
  display.print(" "); display.println(myPedal->delay_time1);
  display.print(" "); display.print(myPedal->delay_mix2);
  display.print(" "); display.println(myPedal->delay_time2);
  display.print(" Fdbck "); display.print(myPedal->feedback);

  display.display();

  vTaskDelay(500);

} // End of while(1)
} // End of task
```

Amplitude Modulation of SD file

This effect performs amplitude modulation separately on the left and right signals of a WAV file played back from an SD card.

Two oscillator objects, **lfo1** and **lfo2**, are created from the DSP oscillator class. Their frequencies, controlled from two pots, range in values from 0.5Hz to 1 or 2KHz. When multiplied with the audio signal, the result ranges from a tremolo to full AM modulation. A pan control for each channel controls the mix between the original signal and the amplitude modulated signal.

The pan control could just as easily control a mix between the pure oscillator signal and the modulated SD signal. The oscillators here use a sine wave table but could just as easily use any waveshape stored in a waveform table. Any number of oscillator instances could be created. What is being described here is the beginnings of an electronic synthesizer built from **bsdsp** file classes.

set_module.h

```
#ifndef MODULE_H_
#define MODULE_H_

#include "controller_mod.h"
#include "bsdsp.h"

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~ DSP Class Declarations (bsdsp files) ~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  extern oscillator lfo1;
  extern oscillator lfo2;
```

```
//Create a child class derived from controllerModule
//The controller_module sets up all Pot, Switch, and LED pin, mode, and
actions

class controller_module:public controllerModule
{
  public:
  float freq1;
  float freq2;

  float pan1;
  float pan2;

  void init();
  void onButtonChange(int buttonIndex);
  void onControlChange(int controlIndex);
};

//controller_module myPedal declaration with extern
extern controller_module *myPedal ;

#endif
```

set_module.cpp

```
#include "set_module.h"
#include "set_settings.h"
#include "set_codec.h"

//controller_module myPedal definition
controller_module *myPedal = new controller_module();

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~ DSP Class Definitions (bsdsp files) ~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  oscillator lfo1;
  oscillator lfo2;

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~ CONTROLLER MODULE CLASS DEFINITIONS ~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Define the controllerModule functions declared in set_module.h
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::init()  //effect module class initialization
 {
  name = "Amplitude Modulation";

  // Set up pin Modes for the switches and LEDs
  // For mode details, see control_task() and button_task() in task.cpp
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(KEY1, INPUT_PULLUP);  //internal pullup
  pinMode(KEY2, INPUT_PULLUP);

  //setting up the buttons
```

```
  button[0].name = "mute";
  button[0].mode = BM_MOMENTARY;
  button[0].touch = false;
  button[0].pin = KEY1;

  button[1].name = "LED";
  button[1].mode = BM_TOGGLE;
  button[1].touch = false;
  button[1].pin = KEY2;

  //add gain control
  control[0].name = "Freq1";
  control[0].mode = CM_POT;
  control[0].levelCount = 128;
  control[0].pin = POT1;

  //add range control
  control[1].name = "Freq2";
  control[1].mode = CM_POT;
  control[1].levelCount = 128;
  control[1].pin = POT2;

  control[2].name = "Pan1";
  control[2].mode = CM_POT;
  control[2].levelCount = 128;
  control[2].pin = POT3;

  control[3].name = "Pan2";
  control[3].mode = CM_POT;
  control[3].levelCount = 128;
  control[3].pin = POT4;

  freq1=5;
  freq2=5;
  pan1=0.5;
  pan2=0.5;
  lfo1.setFrequency(freq1);
  lfo2.setFrequency(freq2);
 }
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::onButtonChange(int buttonIndex)
{
  switch(buttonIndex)
  {
    case 0: //main button state has changed
    {
      if(button[0].value) //if effect is activated
      {
        //codec.analogBypass(false);
        codec.disableDacMute();
        digitalWrite(LED1, HIGH);
      }
      else //if effect is bypassed
      {
        //codec.analogBypass(true);
        codec.enableDacMute();
        digitalWrite(LED1, LOW);
      }
      break;
    }
```

```
    case 1: //the button[1] state has changed
    {
      if(button[1].value) // just test LED and Switch
      {digitalWrite(LED2, HIGH);}
      else
      {digitalWrite(LED2, LOW);}
      break;
    }
  }
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::onControlChange(int controlIndex)
{
  switch(controlIndex)
  {
    case 0: //freq Left
    {
      freq1 = 0.5 + 2000 * (float)control[0].value/127.0;
      lfo1.setFrequency(freq1);
      break;
    }
   case 1: //freq Right
    {
      freq2 = 0.5 + 1000 * (float)control[1].value/127.0;
      lfo2.setFrequency(freq2);
      break;
    }
    case 2:  //pan1
    {
      pan1 = (float)control[2].value/127.0;
      break;
    }

    case 3:  //pan2
    {
      pan2 = (float)control[3].value/127.0;
      break;
    }

 }
}
```

main.cpp

```
#include <Arduino.h>
#include "set_settings.h"
#include "set_module.h"
#include "set_codec.h"
#include "task.h"
#include <SD.h>
#include "sd_play.h"

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

SDplay mySDplay; //create an instance of SDplay class

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~SETUP~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
void setup()
{
  Serial.begin(115200);
  while(!Serial);
  delay(1000);

  //~~~~~~~~~~~codec is initialized  See Codec.cpp~~~~~~~~~~~~~~~~~
  //~~~~i2c is initialized within codec.init() with initI2C()~~~~~~

        Wire.begin();
        Serial.println("Initialize Codec Codec ");
        codec.begin();
        codec_sets();
        Serial.println("Codec Init success!!");

  //~~~~~~I2S  See set_settings.cpp for I2S ~~~~~~~~~~~~~~~

        I2S_init();

         mySDplay.SDCardInit();

        mySDplay.OpenWaveFile();

  //~~~~~~~~~~~~~~Monitor (can be commented out)~~~~~~~~~~

        Serial.println("I2S/SD setup complete");
        runSystemMonitor();  //for testing only

} //Setup End


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~MAIN LOOP~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop()
{
    size_t readsize = 0;
    byte txbuf[SAMPLES_BUFFER_SIZE]; //128 L+R signed 16 bit samples
    float rxl, rxr, txl, txr;  //left and right single samples, processed
as floats

    myPedal->init();
    taskSetup();

  while(1)
  {   //signal processing loop

    setDebugVars(myPedal->freq1, myPedal->freq2, myPedal->pan1,
myPedal->pan2);

    //gather some input samples into Samples buffer from the SD wavfile,
      mySDplay.ReadFile(mySDplay.Samples);

    //process samples one at a time from Samples[]
    for (int i=0; i<(SAMPLES_BUFFER_SIZE); i+=4)
    {
       rxl = (float)((int16_t)(mySDplay.Samples[i+1] << 8) |
mySDplay.Samples[i]);   // Left sample float
```

```
       rxr = (float)((int16_t)(mySDplay.Samples[i+3] << 8) |
mySDplay.Samples[i+2]); // Right sample float

      //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      //~~~~~~~~~Modulation Processing~~~~~~~~~~~~~
      //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        lfo1.update();
        lfo2.update();

        //txr = 0x7fff * myPedal->pan2 * lfo2.getOutput(); // lfo only
        //txl = 0x7fff * myPedal->pan1 * lfo1.getOutput();

        txl = ((1 - myPedal->pan1) * rxl) + (myPedal->pan1 * (rxl *
lfo1.getOutput()));
        txr = ((1 - myPedal->pan2) * rxr) + (myPedal->pan2 * (rxr *
lfo2.getOutput()));
      //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

        txbuf[i]   = ((int16_t) txl) & 0xff ;  // Left sample loaded as two
bytes
        txbuf[i+1] = ((int16_t) txl) >> 8;
        txbuf[i+2] = ((int16_t) txr) & 0xff ;  // Right sample loaded as
two bytes
        txbuf[i+3] = ((int16_t) txr) >> 8;

    } // End of for loop

     // play processed transmit buffer by loading txbuf into DMA memory
    mySDplay.FillI2SBuffer(txbuf);

  } // End of while(1) loop
}   // End of Main Loop
```

Transfer Function Distortion

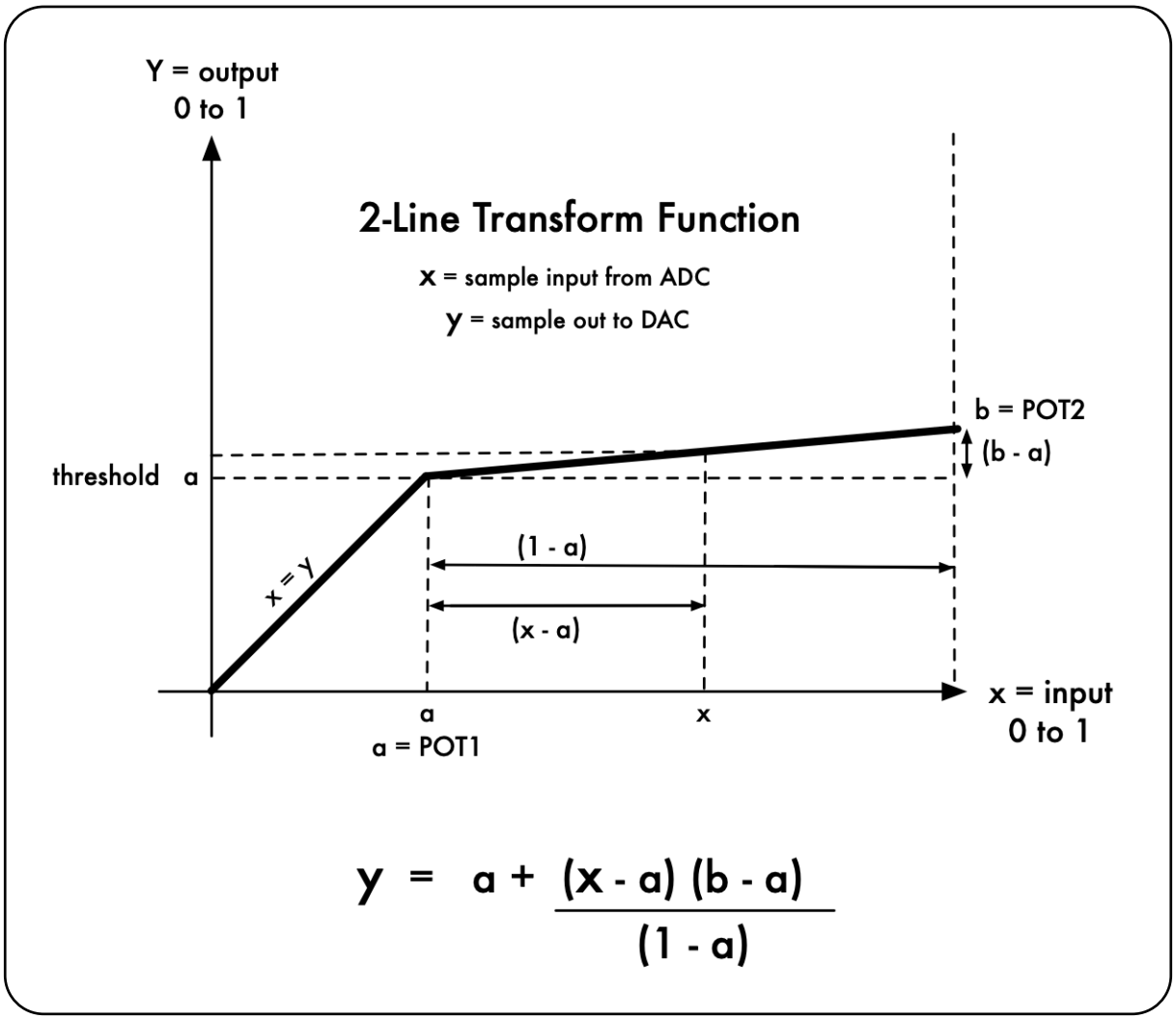
Think of a transfer function as an array of alternative audio sample values. A 16-bit sample will have 32768 possible positive values, running from 0 to 32767, and the same number of negative values. Build an array of 32768 sample values. Given any input sample, use it as an address into the array and replace its 16-bit value with the alternative value stored in the array at that address. Do the same for the negative sample values.

The type of distortion depends on your choice of array values. For no distortion just make the array element value equal to its array address. Kind of boring. How about at some threshold sample value such as 20000 don't continue to increment the array value with the address value, just keep it at 20000 for the remaining addresses. The result is an output signal with its tops and bottoms chopped off, which is the familiar "fuzz" box distortion. However, there are so many other possibilities which can be explored here.

The size of this array is perhaps too large for the current ESP32 available memory space. The DSP file **bsdsp.cpp** include a waveShaper class with an array of 512 floating point values ranging from 0 to 1. Luckily, the ESP32 has a fast floating point processor to translate this 0 to 1 range back and forth from the original range by simple floating point multiplication and division by 32767.

For this Distortion effect, however, I won't be using an array. I'll be going algebraic on you with floating sample values from 0 to 1.

An equation is derived from the straight line "transform" function shown on the graph in the figure below. The x-axis is ADC input sample values from zero to 1. The y-axis is DAC output sample values from zero to 1.



For low sample values from zero to a "threshold" the straight line function is very simple: x = y, or input sample value is equal to the output sample value - no change from input to output. A threshold value is set by Pot 1 ("a" on the graph) and is the start of a second straight line function. The end of this second straight line, at y input = 1, is set by Pot 2. Let's explore some specific possibilities for this second straight line function.

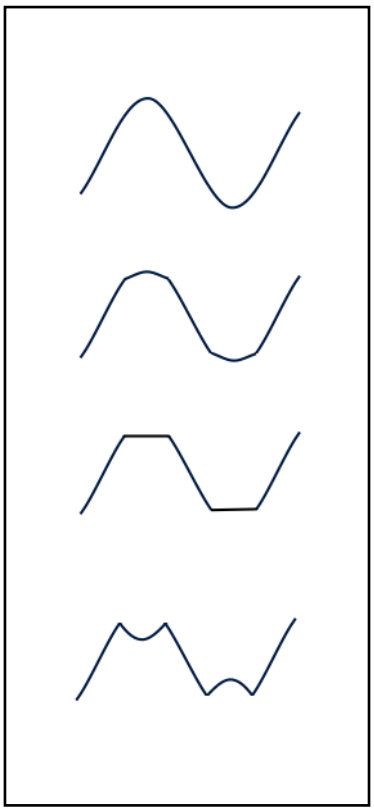
If Pot 2 sets a value of y=1 at x=1, then this second straight line is just a continuation of the slope of the first line setting x = y for all possible input sample values. The rather boring result is that there is no change from the input to the output.

If Pot 2 sets a value that is equal to Pot 1, the threshold, then the second line lies flat at that same threshold value for all input values from threshold to 1. Any input values coming in above the threshold will be ignored and set to equal to that one threshold value resulting in a clipped waveform, which is a hard clip Fuzz distortion.

If Pot 2 sets an end value between the Pot 1 value and 1. The output signal peaks above the threshold will be a "squished" version of the input. The amount of squish can vary between the two examples described above, from zero squish (x = y, when Pot2 = 1) to complete clipping (x = threshold, when Pot2 = Pot1). This becomes a type of audio "compressor".

A somewhat unusual case happens when Pot 2 sets an end point below the Pot 1 threshold value resulting in a downward sloping line. In this case the input signal peaks are compressed, or "squished" as described above, but they are also inverted - an upward hump now becomes a downward hump. This introduces a prominent harmonic in the signal that is twice the signal's base frequency (an octave above).

For a sinewave input, the output results are shown below as the Pot 2 values are moved down from 1 to below the Pot 1 threshold value.



In the sketch below the calculations shown in the graph are implemented. For input values below the Pot 1 threshold the output value is unchanged. For input values above the threshold, the algebraic equation shown is used to calculate the output sample value. The same calculations are made for the negative going parts of the input signal.

set_module.h

```
#ifndef MODULE_H_
#define MODULE_H_

#include "controller_mod.h"
#include "bsdsp.h"

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~ DSP Class Declarations (bsdsp files) ~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//Create a child class derived from controllerModule
//The controller_module sets up all Pot, Switch, and LED pin, mode, and
actions

class controller_module:public controllerModule
{
  public:
  float threshold;
  float distortion;
```

```
  float mix;

  void init();
  void onButtonChange(int buttonIndex);
  void onControlChange(int controlIndex);
};

//controller_module myPedal declaration with extern
extern controller_module *myPedal ;

#endif
```

set_module.cpp

```
#include "set_module.h"
#include "set_settings.h"
#include "set_codec.h"

//controller_module myPedal definition
controller_module *myPedal = new controller_module();

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~ DSP Class Definitions (bsdsp files) ~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~ CONTROLLER MODULE CLASS DEFINITIONS ~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

// Define the controllerModule functions declared in set_module.h
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::init()  //effect module class initialization
 {
  name = "Transfer Function Distortion";

  // Set up pin Modes for the switches and LEDs
  // For mode details, see control_task() and button_task() in task.cpp
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(KEY1, INPUT_PULLUP);  //internal pullup
  pinMode(KEY2, INPUT_PULLUP);

  //setting up the buttons
  button[0].name = "mute";
  button[0].mode = BM_MOMENTARY;
  button[0].touch = false;
  button[0].pin = KEY1;

  button[1].name = "LED";
  button[1].mode = BM_MOMENTARY;
  button[1].touch = false;
  button[1].pin = KEY2;

  //add gain control
  control[0].name = "Threshold";
  control[0].mode = CM_POT;
  control[0].levelCount = 256;
  control[0].pin = POT1;
```

```
  //add range control
  control[1].name = "Distortion";
  control[1].mode = CM_POT;
  control[1].levelCount = 256;
  control[1].pin = POT2;

  control[5].name = "Mix";
  control[5].mode = CM_POT;
  control[5].levelCount = 256;
  control[5].pin = POT6;

  threshold = 1.0;
  distortion = 0;
  mix = 0;
 }
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::onButtonChange(int buttonIndex)
{
  switch(buttonIndex)
  {
    case 0: //main button state has changed
    {
      if(button[0].value) //mute
      {
        //codec.analogBypass(false);
        codec.disableDacMute();
        digitalWrite(LED1, HIGH);
      }
      else //if effect is bypassed
      {
        //codec.analogBypass(true);
        codec.enableDacMute();
        digitalWrite(LED1, LOW);
      }
      break;
    }
    case 1: //the button[1] state has changed
    {
      if(button[1].value) // enter distortion values on button press
      {
        digitalWrite(LED2, HIGH);
      }
      else
      {digitalWrite(LED2, LOW);}
      break;
    }
  }
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::onControlChange(int controlIndex)
{
  switch(controlIndex)
  {
    case 0: //Threshold start of Distortion
    {
      threshold = (float)control[0].value/255.0;  // 0 to 1
      break;
    }
    case 1:  //Distortion as Function Slope
    {
```

```
      distortion = -1.0 + (float)control[1].value/130.0 ; // -1 to +1
      break;
    }
    case 5: //Pan between input and distortion  // 0 to 1
    {
      mix = (float)control[5].value/255.0;
      break;
    }
  }
}
```

main.cpp

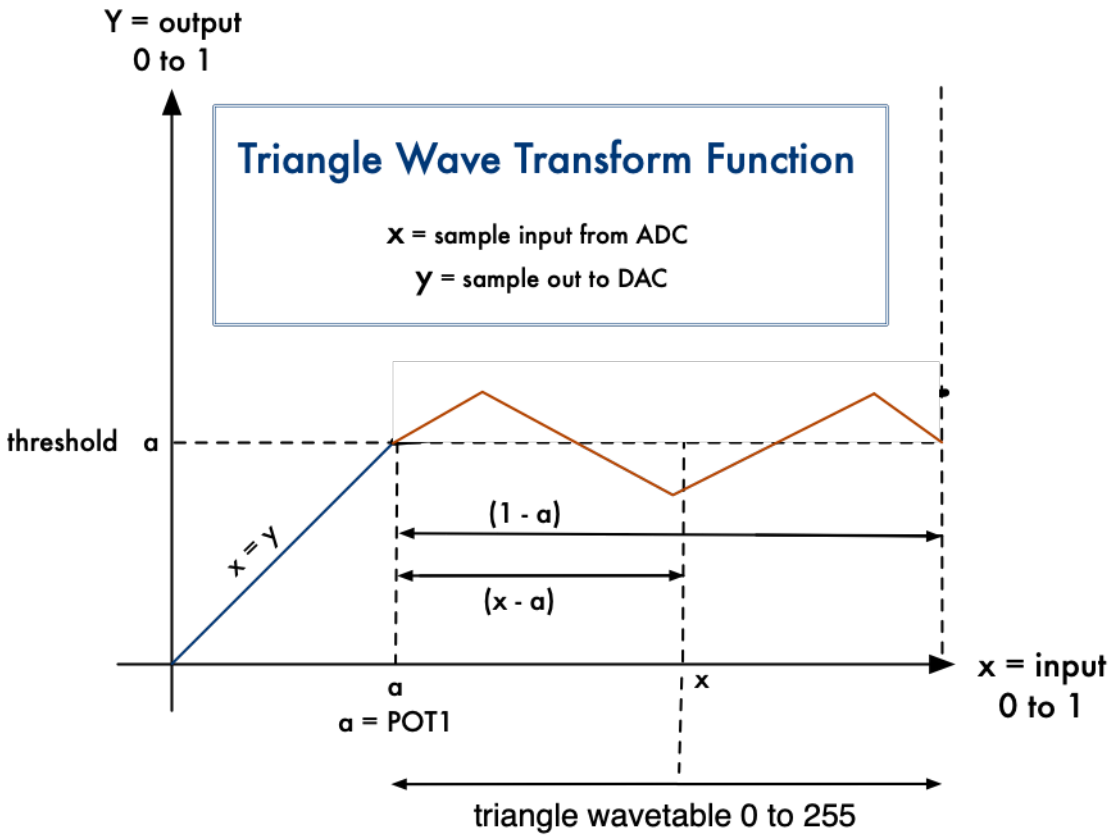
```
#include <Arduino.h>
#include "set_settings.h"
#include "set_module.h"
#include "set_codec.h"
#include "task.h"
#include <SD.h>
#include "sd_play.h"

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~SETUP~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup()
{
  Serial.begin(115200);
  while(!Serial);
  delay(3000);

  //~~~~~~~~~~~codec is initialized  See Codec.cpp~~~~~~~~~~~~~~~~~
  //~~~~i2c is initialized within codec.init() with initI2C()~~~~~~

        Wire.begin();
        Serial.println("Initialize Codec Codec ");
        codec.begin();
        codec_sets();
        Serial.println("Codec Init success!!");

  //~~~~~~I2S  See set_settings.cpp for I2S ~~~~~~~~~~~~~~~

        I2S_init();

  //~~~~~~~~~~~~~~Monitor (can be commented out)~~~~~~~~~~

        Serial.println("I2S/SD setup complete");
        runSystemMonitor();  //for testing only

} //Setup End

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~MAIN LOOP~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop()
{
    size_t readsize = 0;
    int16_t rxbuf[FRAMELENGTH], txbuf[FRAMELENGTH]; //128 L+R signed 16 bit
```

```
samples
    float rxl, rxr, txl, txr;  //left and right single samples, processed
as floats
    float rx, tx;
    float a, b;
    bool rx_negative;

    myPedal->init();
    taskSetup();

  while(1){   //signal processing loop

  setDebugVars(myPedal->threshold, myPedal->distortion, myPedal->mix, 0);

  //gather some input samples into receive buffer from the DMA memory,
  i2s_read(I2S_NUM_0, rxbuf, FRAMELENGTH*2, &readsize, 20);

        a = myPedal->threshold ;
        b = myPedal->distortion;

  for (int i=0; i<(FRAMELENGTH); i+=2) {  //process samples one at a time
from buffers

    rxl = (float) (rxbuf[i]) ;   //convert sample to float
    rxr = (float) (rxbuf[i+1]) ;

    rx = (0.5 * rxl) + (0.5 * rxr);  // change to mono signal
    rx = rx / 32767;  // samples 0 to 1

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    //~~~~~~~~~~Distortion Processing~~~~~~~~~~~~~~
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    if(rx < 0) { rx_negative = 1; rx = -rx; } // Negative sample value
    else {rx_negative = 0;}

    //Transfer Function
    if(rx > a) {  tx = a + ( ((b - a) * (rx - a)) / (1 - a)); }

    else { tx = rx; }

    if(rx_negative) { rx = -rx;  tx = -tx; }

    rx = rx * 32767;
    tx = tx * 32767;
    txl = ((1 - myPedal->mix) * rx) + (myPedal->mix * tx);
    txr = txl;  // Mono

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    txbuf[i]   = ((int16_t) txl) ; //convert sample back to integer
    txbuf[i+1] = ((int16_t) txr) ;
  }
  // play processed receive buffer by loading transmit buffer into DMA
memory
  i2s_write(I2S_NUM_0, txbuf, FRAMELENGTH*2, &readsize, 20);

} // End of while(1) loop
} // End of Main Loop
```

Transfer Distortion Triangle Wave

This unique distortion/modulation effect expands upon the straight line transfer function of the previous effect. There is still a movable threshold value that defines where the distortion starts on the input peaks. However, the single sloping distortion line is replaced with the sloping lines of a triangle wave oscillator with control over its gain and frequency.

4 controllers — threshold, triangle frequency, triangle gain, distortion mix



x-phase = 255*(x-a)/(1-a)

phase-increment = 255 * frequency/sample_rate

x-triangle-phase = 255 * fractional_part_of { x-phase * phase-increment / 255 }

output_sample = triangle.getOutput(x-triangle-phase) * triangle_gain

getOutput( ) is from bsdsp.cpp oscillator class

set_module.h

```
#ifndef MODULE_H_
#define MODULE_H_

#include "controller_mod.h"
#include "bsdsp.h"
#include "dsptable.h"

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~ DSP Class Declarations (bsdsp files) ~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//Create a child class derived from controllerModule
//The controller_module sets up all Pot, Switch, and LED pin, mode, and
actions

extern oscillator triangle;


class controller_module:public controllerModule
{
  public:
  float threshold;
  float frequency;
  float depth;
  float mix;
  float aa;

  void init();
  void onButtonChange(int buttonIndex);
  void onControlChange(int controlIndex);
};

//controller_module myPedal declaration with extern
extern controller_module *myPedal ;

#endif
```

set_module.cpp

```
#include "set_module.h"
#include "set_settings.h"
#include "set_codec.h"

//controller_module myPedal definition
    controller_module *myPedal = new controller_module();

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~ DSP Class Definitions (bsdsp files) ~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    oscillator triangle;

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~ CONTROLLER MODULE CLASS DEFINITIONS ~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
// Define the controllerModule functions declared in set_module.h
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::init()  //effect module class initialization
 {
  name = "Transfer Triangle Distortion";


  // Set up pin Modes for the switches and LEDs
  // For mode details, see control_task() and button_task() in task.cpp
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(KEY1, INPUT_PULLUP);  //internal pullup
  pinMode(KEY2, INPUT_PULLUP);

  //setting up the buttons
  button[0].name = "mute";
  button[0].mode = BM_MOMENTARY;
  button[0].touch = false;
  button[0].pin = KEY1;

  button[1].name = "LED";
  button[1].mode = BM_MOMENTARY;
  button[1].touch = false;
  button[1].pin = KEY2;

  //setting up pots
  control[0].name = "Threshold";
  control[0].mode = CM_POT;
  control[0].levelCount = 127;
  control[0].pin = POT1;

  control[1].name = "Frequency";
  control[1].mode = CM_POT;
  control[1].levelCount = 256;
  control[1].pin = POT2;

  control[2].name = "Depth";
  control[2].mode = CM_POT;
  control[2].levelCount = 256;
  control[2].pin = POT3;

  control[5].name = "Mix";
  control[5].mode = CM_POT;
  control[5].levelCount = 256;
  control[5].pin = POT6;

  threshold = 1.0;
  frequency = 1;
  depth = 0;
  mix = 0;
  triangle.setFrequency(frequency);
  triangle.setWaveTable(triangle_table);
 }

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::onButtonChange(int buttonIndex)
{
  switch(buttonIndex)
  {
```

```
    case 0: //main button state has changed
    {
      if(button[0].value) //mute
      {
        codec.disableLoopBack();
        //codec.disableDacMute();
        digitalWrite(LED1, HIGH);
      }
      else //if effect is bypassed
      {
        codec.enableLoopBack();
        //codec.enableDacMute();
        digitalWrite(LED1, LOW);
      }
      break;
    }
    case 1: //the button[1] state has changed
    {
      if(button[1].value) // enter threshold values on button press
      {
        digitalWrite(LED2, HIGH);
        aa = threshold;
      }
      else
      {digitalWrite(LED2, LOW);}
      break;
    }
  }
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::onControlChange(int controlIndex)
{
  switch(controlIndex)
  {
    case 0: //Threshold start of Distortion
    {
      threshold = control[0].value << 8;  // 0 to 32512
      break;
    }
    case 1: //Frequency of Triangle Distortion
    {
      frequency = 1.0+(400*(float)control[1].value/255.0); // 1Hz to 400Hz
      triangle.setFrequency(frequency);
      break;
    }
    case 2:  //Depth of Triangle Distortion
    {
      depth =  (float)control[2].value/255.0; // 0 to 1
      break;
    }
    case 5: //Pan between input and distortion  // 0 to 1
    {
      mix = (float)control[5].value/255.0;
      break;
    }
 }
}
```

main.cpp

```
#include <Arduino.h>
#include "set_settings.h"
#include "set_module.h"
#include "set_codec.h"
#include "task.h"
#include <SD.h>
#include "sd_play.h"

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~SETUP~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup()
{
  Serial.begin(115200);
  while(!Serial);
  delay(3000);

  //~~~~~~~~~~~codec is initialized  See Codec.cpp~~~~~~~~~~~~~~~~~
  //~~~~i2c is initialized within codec.init() with initI2C()~~~~~~

        Wire.begin();
        Serial.println("Initialize Codec Codec ");
        codec.begin();
        codec_sets();
        Serial.println("Codec Init success!!");

  //~~~~~~I2S  See set_settings.cpp for I2S ~~~~~~~~~~~~~~~

        I2S_init();

  //~~~~~~~~~~~~~~~Monitor (can be commented out)~~~~~~~~~~

        Serial.println("I2S/SD setup complete");
        runSystemMonitor();  //for testing only

} //Setup End


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~MAIN LOOP~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loop()
{
    size_t readsize = 0;
    int16_t rxbuf[FRAMELENGTH], txbuf[FRAMELENGTH]; //128 L+R signed 16 bit
samples
    float rxl, rxr, txl, txr;  //left and right single samples, processed
as floats
    float rx, tx;  //mono input-output samples, -1 to +1 floats
    float a, b;
    float phase;
    bool rx_negative;
    float table_rate = 255.0/(float)SAMPLE_RATE;

    myPedal->init();
    taskSetup();
```

```
  while(1){   //signal processing loop

  setDebugVars(myPedal->threshold, myPedal->frequency, myPedal->depth,
myPedal->mix);

  //gather some input samples into receive buffer from the DMA memory,
  i2s_read(I2S_NUM_0, rxbuf, FRAMELENGTH*2, &readsize, 20);

      //a = myPedal->aa ;  // Enter  with pushbutton
        a = myPedal->threshold ;

  //process samples one at a time from buffers
  for (int i=0; i<(FRAMELENGTH); i+=2) {

    rxl = (float) (rxbuf[i]) ;   //convert sample to float
    rxr = (float) (rxbuf[i+1]) ;

    // change to mono signal, 0 to +/- 32767 input sample values
    rx = (0.5 * rxl) + (0.5 * rxr);

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    //~~~~~~~~~Distortion Processing~~~~~~~~~~~~~
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    // same transfer function on positive and negative signal peaks

    if(rx < 0) { rx_negative = 1; rx = -rx; }  // Flip negative samples
    else {rx_negative = 0;}


    if(rx > a)
    {
       phase = ( (rx - a)/(32767.0 - a)) * table_rate * myPedal->frequency;

       // fractional part of phase * 255 wavetable size
       phase = (phase - (float)(int(phase))) * 255.0;

            triangle.setPhase(0);
            tx = triangle.getOutput(phase)* myPedal->depth;

            tx = a + (32767.0 * tx);
            if(tx>32767.0) {tx=32767.0;}  // 0 to 32767 output sample
    }
    else { tx = rx; }  // No sample changes below threshold


    if(rx_negative) { rx = -rx;  tx = -tx; } //Re-Flip negative samples

    //mix of input and distorted
    txl = ((1 - myPedal->mix) * rx) + (myPedal->mix * tx);
    txr = txl;  // Output Mono

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    txbuf[i]   = ((int16_t) txl) ; //convert sample back to integer
    txbuf[i+1] = ((int16_t) txr) ;
  }
  // play processed receive buffer by loading transmit buffer into DMA
memory
```

```
  i2s_write(I2S_NUM_0, txbuf, FRAMELENGTH*2, &readsize, 20);

} // End of while(1) loop
} // End of Main Loop
```