Sparkfun Codec with ESP32 Thing Plus C

or the "Sparkfun Codec Thing"

John Talbert - February 2023

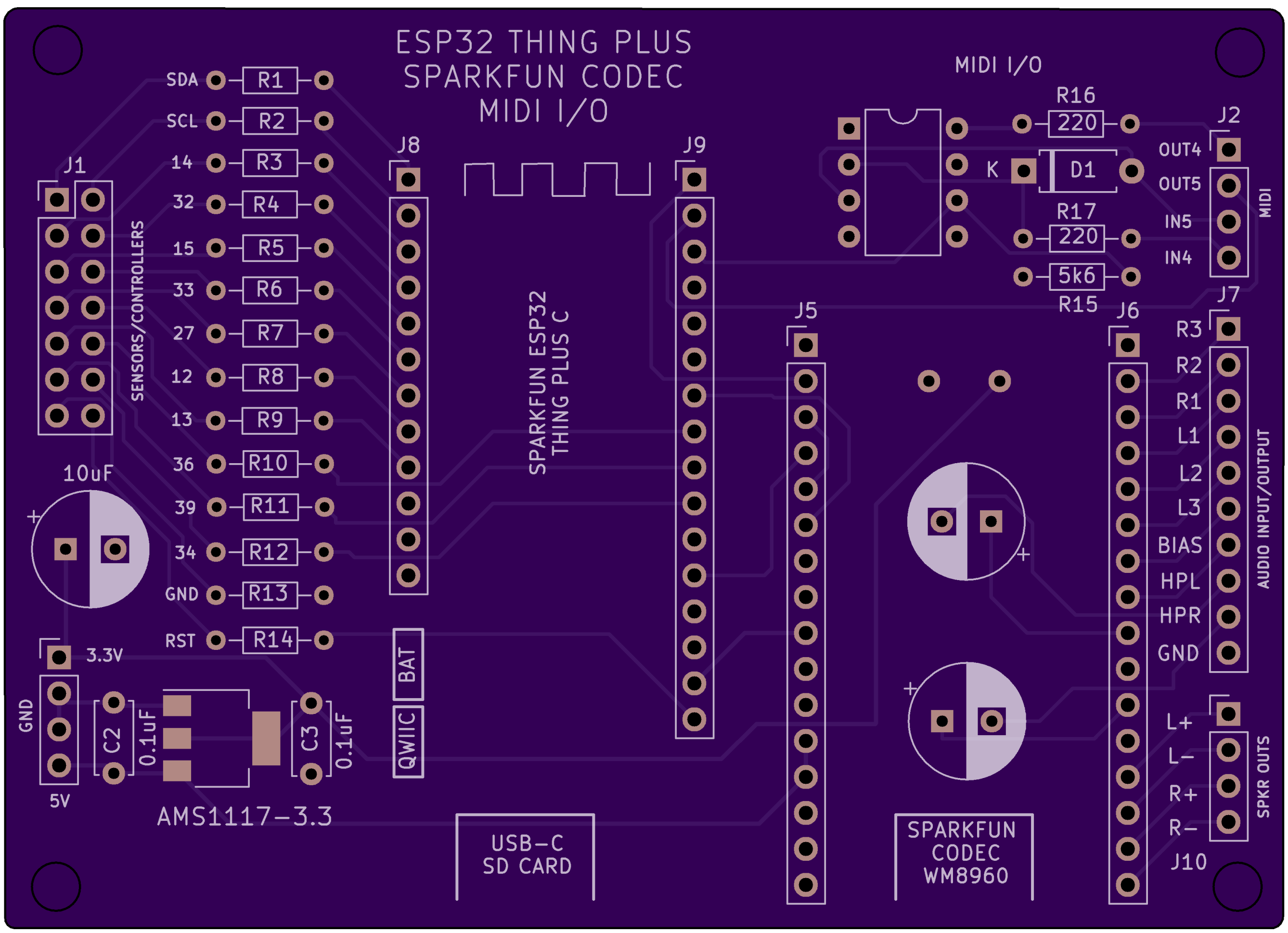


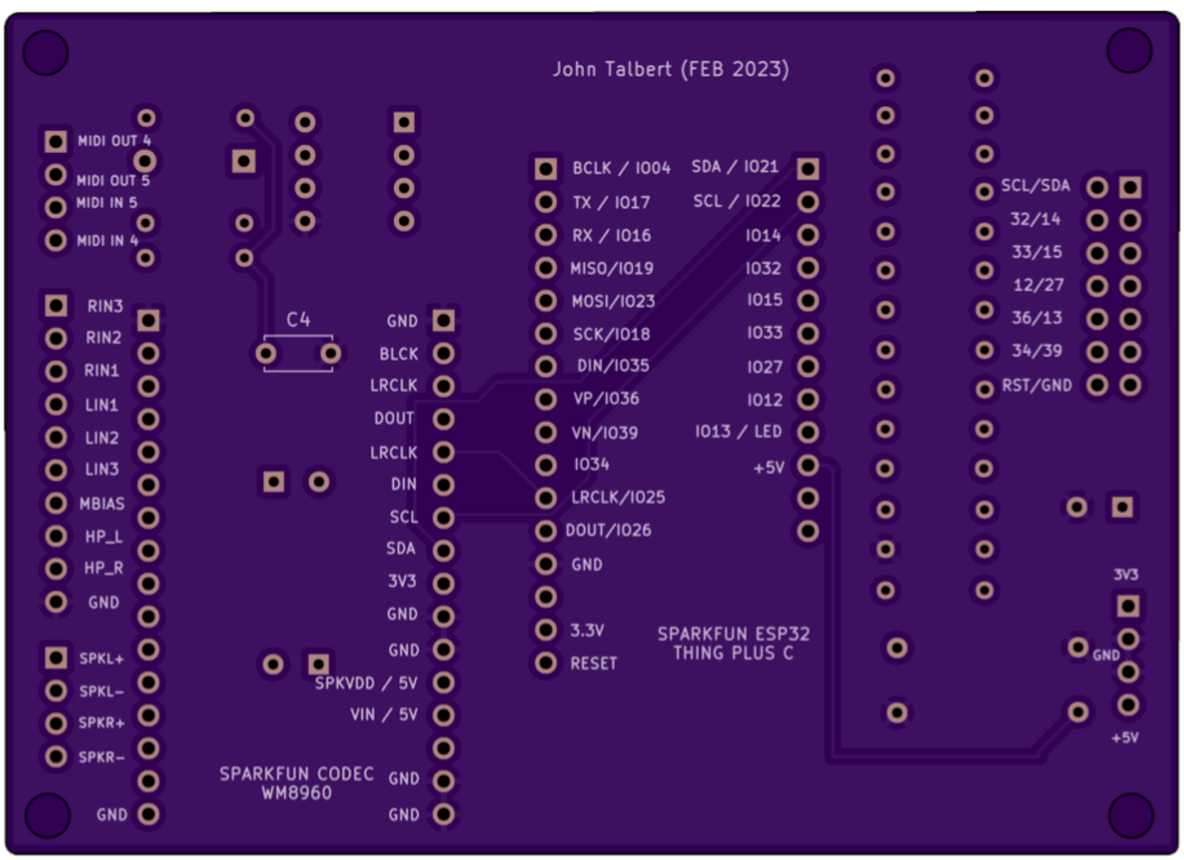
Table of Contents

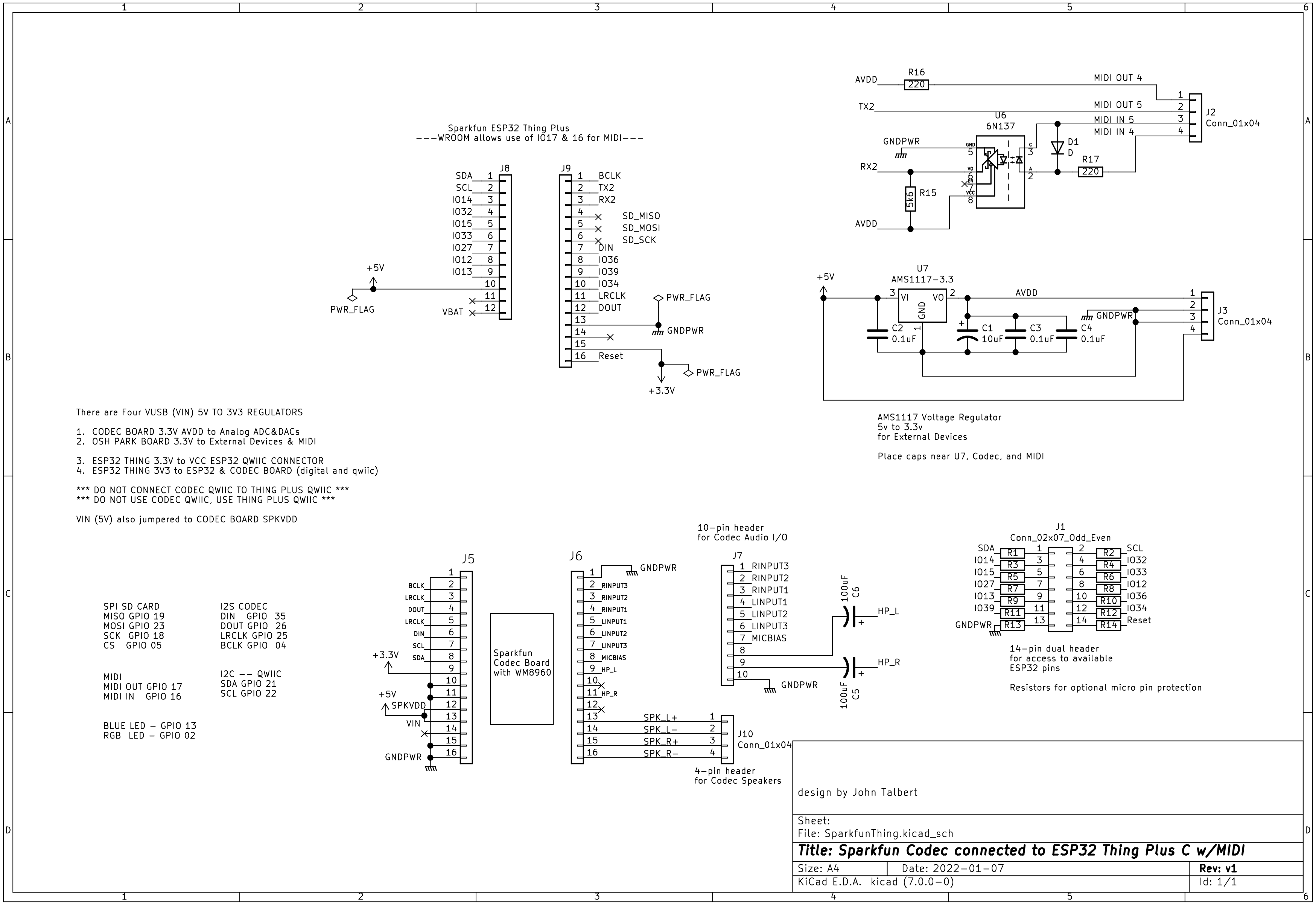
The Board

This PCB board 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 incorporates 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 SCL IO14 IO32 IO15 IO33 IO27 IO12 IO13 VBAT
+5V PWR_FLAG
J9
BCLK TX2 RX2 SD_MISO SD_MOSI SD_SCK DIN IO36 IO39 IO34 LRCLK DOUT Reset
PWR_FLAG GNDPWR +3.3V
R16 220 AVDD TX2 RX2 GNDPWR
U6 6N137
D1 D
R15 5k6
R17 220
MIDI OUT 4
MIDI OUT 5
MIDI IN 5
MIDI IN 4
J2 Conn_01x04
U7 AMS1117-3.3 VI VO GND
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
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
I2S CODEC
DIN GPIO 35
DOUT GPIO 26
LRCLK GPIO 25
BCLK GPIO 04
MIDI
MIDI OUT GPIO 17
MIDI IN GPIO 16
I2C -- QWIIC
SDA GPIO 21
SCL GPIO 22
BLUE LED - GPIO 13
RGB LED - GPIO 02
J5
BCLK LRCLK DOUT LRCLK DIN SCL SDA +3.3V +5V SPKVDD VIN GNDPWR
Sparkfun
Codec Board
with WM8960
J6
RINPUT3 RINPUT2 RINPUT1 LINPUT1 LINPUT2 LINPUT3 MICBIAS HP_L HP_R GNDPWR
10-pin header
for Codec Audio I/O
J7
RINPUT3 RINPUT2 RINPUT1 LINPUT1 LINPUT2 LINPUT3 MICBIAS GNDPWR
C6 100uF HP_L
C5 100uF HP_R
SPK_L+ SPK_L- SPK_R+ SPK_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
R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14
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

Power Connections

5V power to both modules comes from either the USB-C connector (**VUSB**) or the Battery connector (**VBAT**) on the Thing Plus module. This is variously labeled as **VIN**, **VUSB**, **+5V**. This 5 volt supply powers a total of four 5v to 3.3v voltage regulators.

1. The main regulator on the ESP32 Thing Plus module supplies 3.3 volts (labeled **3.3V** or **3V3**) at 700ma to the ThingPlus module. It is also connected, through the **3V3** pin, to the Codec Module to power the WM8960's digital circuitry as well as the Qwiic connector on the Codec Module.

2. A second voltage regulator on the ESP32 Thing Plus supplies 3.3 volts to the Qwiic connector on the Thing Plus module. Of the two available Qwiic connectors this is the best choice as it has its own dedicated regulator.

   *** Please do not connect the Thing Plus Qwiic to the Codec Qwiic when using the OSHPark PCB. Doing so will connect two regulator outputs together - not a good thing! The Sparkfun Tutorial Examples make this connection as an easy way to get power to the Codec module. This connection is unnecessary when using the OSHPark PCB.

3. A voltage regulator on the Codec Module is used to provide a clean 3.3 voltage source for the Codec's analog audio circuits. It is labeled **AVDD** and is also available on a module pin.

4. The PCB board has a AMS1117 regulator to supply 3.3 volts to any external sensors or controllers. It is also labeled **AVDD**, not to be confused with the Codec module's **AVDD**.

5. The 5v **VUSB** pin on the Thing Plus module connects to **VIN** and **SPKVDD** pins on the Codec module. **VIN** goes to the Codec module regulator that creates **AVDD**. **SPKVDD** goes to the WM8960 codec to power the speaker outputs on the chip.

   SPKVDD is connected to **VIN** both on the PCB board and with a jumper on the Codec module. Both must be cut if you want to power the Codec Speaker amps from another source.

Interfaces

Qwiic / I2C

Qwiic is a connection system by Sparkfun for I2C interface devices. The standard 4-pin connector carries **3.3v**, **GND**, **SDA** and **SCL**. The ESP32 Thing and the Codec module both have a Qwiic connector. The ESP32 Thing Qwiic connector has its own dedicated 3.3v voltage regulator while the Codec board's Qwiic connector makes use of the same 3.3v that powers both boards. Because of this arrangement, the Qwiic connector on the ESP32 Thing would be the preferred one to use.

Please note that the sample code examples from the Sparkfun Codec tutorials suggest connecting the ESP32 Qwiic to the Codec Qwiic for an easy power connection between the two boards. This is unnecessary when using the OSHPark PCB shown here and may even damage the board voltage regulators.

I2C is a 2-Wire serial communication interface with **SDA** carrying the data, and **SCL** carrying the data clock. The same two lines can service multiple devices as long as each device has its own unique address (https://learn.adafruit.com/i2c-addresses). Note that the WM8960 codec already makes use of this I2C interface for loading its setup registers. I2C is enabled from the "Arduino Wire Library" (https://docs.arduino.cc/learn/communication/wire).

A 14-pin header for external device connections on the OSHPark PCB includes **SDA** and **SCL**. Another 4-pin header provides power for external devices with a 3.3v source derived from a dedicated voltage regulator. External I2C devices can use these header connections as an alternative to the Qwiic connection.

I2S

Along with the I2C (Qwiic) interface the Sparkfun Thing Plus also has an I2S interface used by the Codec to move audio data between the ADCs and DACs.

The ESP32 microprocessor integrates the I2S interface with DMA (Direct Memory Access). Audio data movement from the ADCs to DMA memory blocks and from DMA memory blocks to the DACs at a specified sample rate can happen directly with little involvement from the ESP32 processor. While that automated audio data movement is happening, the program code can employ two specialized I2S functions to keep the DMA buffers filled, **i2s_read( )** and **i2s_write( )**. The I2S interface requires 4 to 5 ESP32 pins:

```
//ESP32-Codec PIN SETUP

//#define IS2_MCLK_PIN 0

#define I2S_NUM   0
#define I2S_BCLK  4  //BCLK, SCK, SCLK
#define I2S_LRC   25 //LRC, WS, ADCLRC, DACLRC, LRCLK -- Left/Right channel
#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
```

The above code lines were taken from the Codec Software Package file **set_settings.h** The pin label names are not completely standardized so that different manufacturers will use different pin designations. I have included some of them in the code comments. For **DIN** and **DOUT** it is important to realize that these designations are from the ESP32 perspective (not the Codec). It's audio data samples into (**DIN**) the ESP32 from the codec ADCs, and data out (**DOUT**) from the ESP32 to the codec DACs.

BCLK is the data clock, similar in function to the I2C **SCL**. **LRC** is another clock that indicates which audio channel, left or right, is presenting its data. Most any ESP32 pin can be used for each I2S function with some restrictions; for example, pin 35 is input only and thus is assigned to **DIN** (it won't work for **DOUT**).

MCLK is an I2S master clock usually associated with pin zero; however, the Sparkfun Codec uses a 24 MHz hardware clock instead. **NUM** is not a GPIO pin number, it is the I2S port number 0 or 1, usually set to zero.

SPI for the SD Card

A micro SD Card Slot is mounted on the underside of the ESP32 Thing Plus module directly beneath the USB connector. It uses an SPI interface with the following pins:

```
//SD Card Reader Settings

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

Read the PDF documentation for the LillyGo TAudio board at jtalbert.xyz/ESP32/ for a description of the **sd_play** Class Software files created to play back WAV audio files from a micro SD Card. For clues on how to set up the codec for this application read **Example 9: I2S Bluetooth** from the Sparkfun Codec Hookup Guide (https://learn.sparkfun.com/tutorials/audio-codec-breakout---wm8960-hookup-guide/all). For this application, codec Inputs from the ADCs must be disabled and only the DACs enabled.

MIDI Input/Output

A MIDI Interface is provided on the PCB. It uses pins **TX2** (IO17) and **RX2** (IO16) for MIDI Input and MIDI Output. A 6N137 opto-isolator chip is used in the MIDI IN circuit. A 4-pin header provides connections to standard 5-pin DIN MIDI Output and MIDI Input jacks.

Read the PDF documentation "ESP32_Codec" at jtalbert.xyz/ESP32/ for example code using the MIDI interface.

LEDs

The Sparkfun ESP32 Thing Plus module has a mounted Blue LED on GPIO pin 13. It also has one RGB WS2812 NeoPixel on GPIO pin 02. Read the PDF documentation for the LillyGo TAudio board at jtalbert.xyz/ESP32/ for a description of the code and library required for dealing with WS2812 NeoPixels.

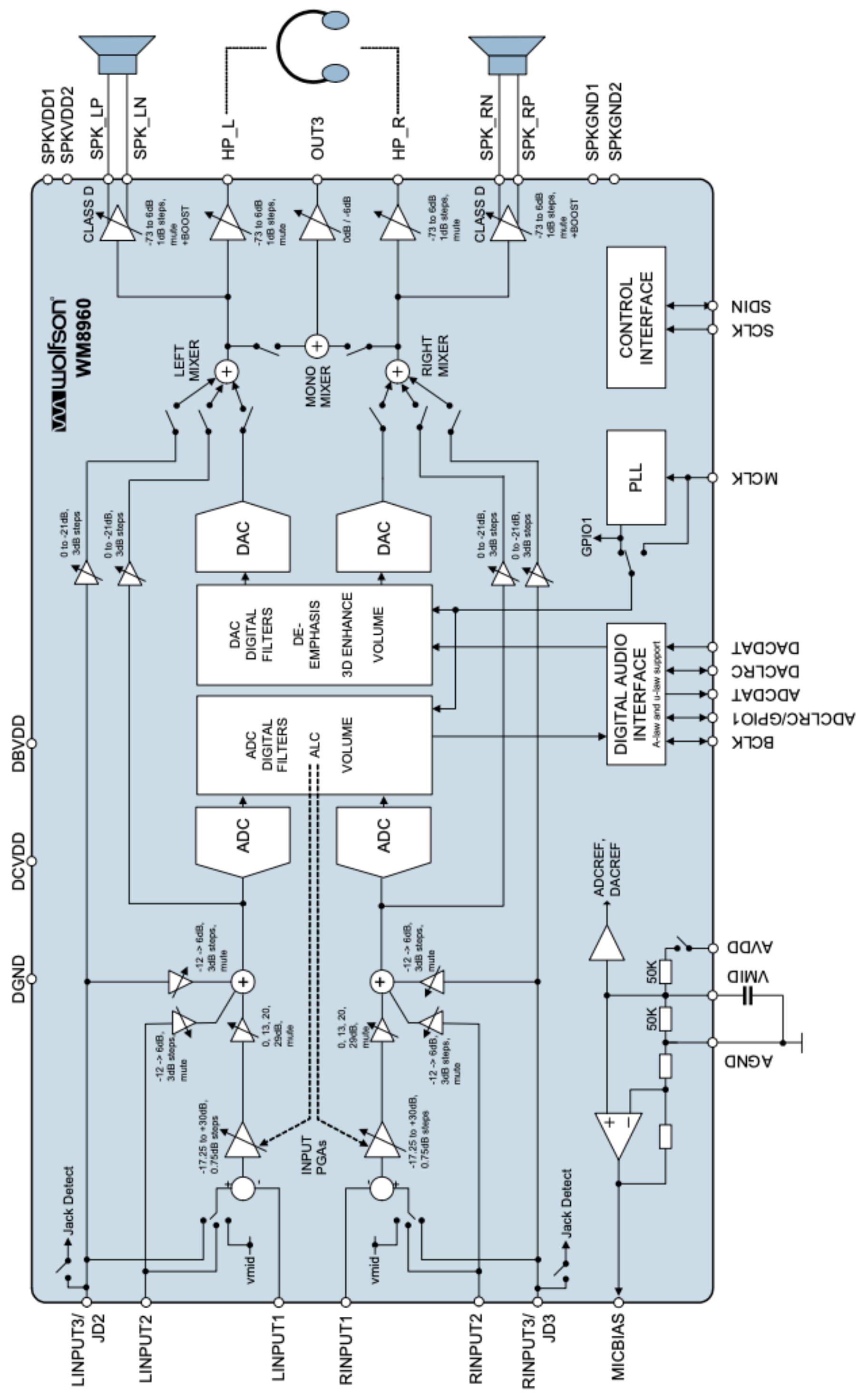
The WM8960 Codec

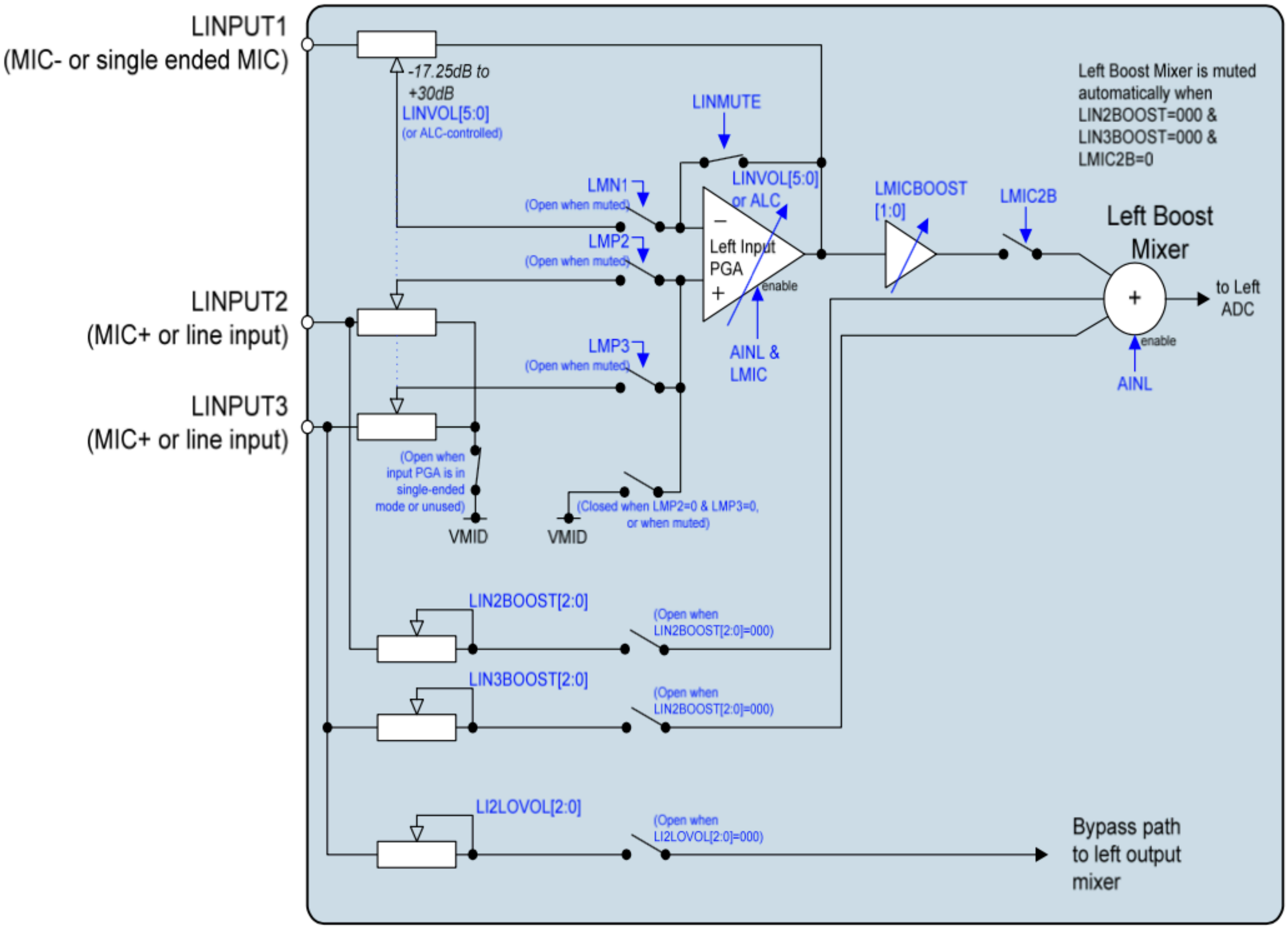
> The SparkFun Audio Codec Breakout - WM8960 is a low power, high quality stereo codec with 1W Stereo Class D speaker drivers and headphone drivers. The WM8960 acts as a stereo audio ADC and DAC, and communicates using I2S, a standard audio data protocol (not to be confused with I2C). This audio codec is chock full of features some of which includes advanced on-chip digital signal processing for automatic level control (ALC) for the line or microphone input, programmable gain amplifier (PGA), pop and click suppression, and its ability to configure I2S settings and analog audio path through software via I2C.

The above description is from the Sparkfun product page which also includes a link to a GitHub Arduino Library for the WM8960 (https://github.com/sparkfun/SparkFun_WM8960_Arduino_Library).

To understand what is going on in the WM8960 codec library you must reference the Wm8960 datasheet at https://datasheetspdf.com/pdf-file/1365067/CirrusLogic/WM8960/1 The codec has a total of 56 9-bit registers, all described in detail in the WM8960 datasheet. These registers are used to set up its numerous modes of operation. They are loaded from the ESP32 using I2C interface commands defined in the **codec.h** codec library file.

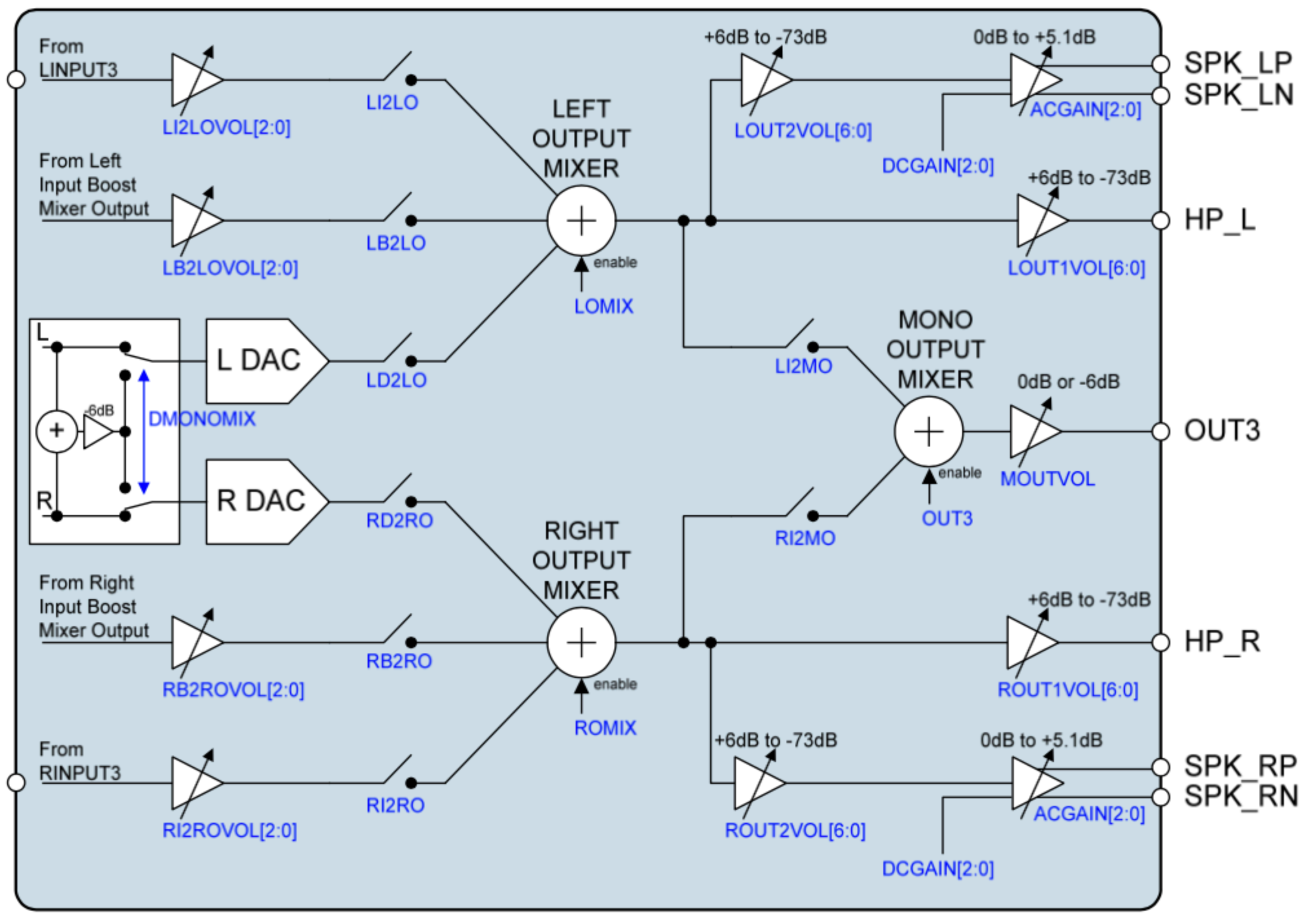
The Sparkfun codec library includes many simple functions (methods) to aid in setting up the codec and in making changes in real time; changes such as DAC output volume, muting, bypass, etc. Many of the register set functions are self-explanatory - enableAdcLeft( ), disableAdcRight( ), enableDacMute( ), disableDacMute( ), enableNoiseGate( ), setLeftAdcDigitalVolume( ). Others such as connectLMN1( ) will need some explanation, much of which you can get from the datasheet block diagrams shown here:




LINPUT1
(MIC- or single ended MIC)
-17.25dB to +30dB
LINVOL[5:0]
(or ALC-controlled)
LINMUTE
LMN1
(Open when muted)
LINVOL[5:0] or ALC
Left Input PGA
enable
LMP2
(Open when muted)
LMICBOOST [1:0]
LMIC2B
Left Boost Mixer is muted automatically when LIN2BOOST=000 & LIN3BOOST=000 & LMIC2B=0
Left Boost Mixer
to Left ADC
enable
AINL
AINL & LMIC
LINPUT2
(MIC+ or line input)
LMP3
(Open when muted)
LINPUT3
(MIC+ or line input)
(Open when input PGA is in single-ended mode or unused)
VMID
VMID
(Closed when LMP2=0 & LMP3=0, or when muted)
LIN2BOOST[2:0]
(Open when LIN2BOOST[2:0]=000)
LIN3BOOST[2:0]
(Open when LIN2BOOST[2:0]=000)
LI2LOVOL[2:0]
(Open when LI2LOVOL[2:0]=000)
Bypass path to left output mixer

OUTPUT MIXERS

Left and right analogue mixers allow the DAC output and analogue bypass paths to be mixed. Programmable attenuation and mute is available on the analogue bypass paths from LINPUT3, RINPUT3 and from the input boost mixers as shown in Figure 13. A mono mix of left and right output mixers is also available on OUT3.



The Sparkfun Codec Hookup Guide has 15 Example setups for the WM8960 Codec with code that includes the register set functions necessary for each setup.

Codec Effects Software Package

Short Description

My **Codec Effects Software Package** was first introduced with the "**ESP32 ES8388 Codec**" PCB project on my website at https://www.jtalbert.xyz/ESP32/. A full description and tutorial is presented there in the PDF document "**Codec Software**". Since then it has been expanded and successfully applied to a number of ESP32 Codec boards including the LyratT, A1S Audio Kit, PUCA DSP board, and LillyGo TAudio, and finally this PCB for the Sparkfun modules, all fully documented with PDFs and software downloads.

The main project goal was to create an audio effects software package for most any audio codec ESP32 board, with full documentation and tutorials, that is easier to use and understand than the usual ADF/IDF libraries. The **Software Package** tweaked for this Sparkfun Codec Thing PCB is available for download along with this PDF.

Coding was done on the **PlatformIO IDE** with an **Arduino Framework**, all from **Visual Studio Code (VSC)**. As such, with a few modification, it could be run from the **Arduino IDE**. Here is the **platform.io** file for the project.

```
[env:sparkfun_esp32s2_thing_plus_c]
platform = espressif32
board = sparkfun_esp32s2_thing_plus_c
framework = arduino
monitor_speed = 115200
lib_deps =
adafruit/Adafruit NeoPixel@^1.10.7
SPI
Wire
SD
sparkfun/SparkFun WM8960 Arduino Library@^1.0.3
```

Most of the package files are written as **.h/.cpp** file pairs (header/code files) with c++ OOP Class structures. The full capabilities of the ESP32 are utilized, special features such as dual core processing, FreeRTOS, Floating Point Unit (FPU) calculations, I2C, SPI, and Direct Memory Access (DMA) integrated with the I2S interface to access the Codec ADCs and DACs.

Below is a short description of the files that make up this **Codec Effects Software**

Package.

1. **codec** -- The driver for a specific codec, the WM8960 in this case.

2. **controller_mod** -- A base class container for analog and digital controllers such as switches and potentiometers.

3. **task** -- Task functions for polling the analog and digital controllers. Task functions for System Monitoring. Task functions for special devices such as NeoPixels. A setup function to place and start up the tasks using **freeRTOS**.

4. **bsdsp** -- Digital Signal Processing (DSP) class tools for the audio effects.

5. **sd_play** -- Class tools for playing audio WAV files from an SD Card.

5. **set_settings**, **set_codec**, **set_module** -- Overall settings for the various package components.

6. **main** -- The main entry file that pulls it all together. Effect Processing on the audio samples.

The Codec Files

The Package **codec.h** and **codec.cpp** files are direct copies of the SparkFun WM8960 Arduino Library found at the GitHub repository https://github.com/sparkfun/SparkFun_WM8960_Arduino_Library. Many thanks to the good people at Sparkfun, especially Pete Lewis and Mike Grusin, for their great work.

The Library includes 15 **Example** Arduino sketches for different Codec setups. These are especially useful for including the codec register set functions necessary for each particular setup. For the demostration code presented here I needed a Codec setup that uses both ADC audio inputs and DAC outputs. **Example 8 - I2S Passthrough** was perfect for my application. It includes a **codec_setup( )** function with all the necessary codec register set functions. I simply copied them to my own **codec_sets( )** function in the package file **set_codec.cpp** as shown below:

```
#include "set_codec.h"

// declaration of codec, an instance of WM8960
               WM8960 codec;
```

```
void codec_sets()         //to be executed in main.cpp
{
 //Example_08 I2S passthrough
    // General setup needed
 codec.enableVREF();
 codec.enableVMID();

 // Setup signal flow to the ADC

 codec.enableLMIC();
 codec.enableRMIC();

 // Connect from INPUT1 to "n" (aka inverting) inputs of PGAs.
 codec.connectLMN1();
 codec.connectRMN1();

 // Disable mutes on PGA inputs (aka INTPUT1)
 codec.disableLINMUTE();
 codec.disableRINMUTE();

 // Set pga volumes
 codec.setLINVOLDB(0.00); //Valid options -17.25dB to +30dB (0.75dB steps)
 codec.setRINVOLDB(0.00); //Valid options -17.25dB to +30dB (0.75dB steps)

 // Set input boosts to get inputs 1 to the boost mixers
 codec.setLMICBOOST(WM8960_MIC_BOOST_GAIN_0DB);
 codec.setRMICBOOST(WM8960_MIC_BOOST_GAIN_0DB);

 // Connect from MIC inputs (aka pga output) to boost mixers
 codec.connectLMIC2B();
 codec.connectRMIC2B();

 // Enable boost mixers
 codec.enableAINL();
 codec.enableAINR();

 // Disconnect LB2LO (booster to output mixer (analog bypass)
 // For this example, we are going to pass audio throught the ADC and DAC
 codec.disableLB2LO();
 codec.disableRB2RO();

 // Connect from DAC outputs to output mixer
 codec.enableLD2LO();
 codec.enableRD2RO();

 // Set gainstage between booster mixer and output mixer
 // For this loopback example, keep these as low as they go
 codec.setLB2LOVOL(WM8960_OUTPUT_MIXER_GAIN_NEG_21DB);
 codec.setRB2ROVOL(WM8960_OUTPUT_MIXER_GAIN_NEG_21DB);

 // Enable output mixers
 codec.enableLOMIX();
 codec.enableROMIX();

 // CLOCK STUFF, settings for 44.1KHz sample rate, and class-d
 // freq 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 is 000 (what we need for 44.1KHz)
  //codec.setDACDIV(0); // Default is 000 (what we need for 44.1KHz)
  codec.setWL(WM8960_WL_16BIT);

  codec.enablePeripheralMode();
  //codec.enableMasterMode();
  //codec.setALRCGPIO(); // Should not be changed while ADC is enabled.

  // Enable ADCs and DACs
  codec.enableAdcLeft();
  codec.enableAdcRight();
  codec.enableDacLeft();
  codec.enableDacRight();
  codec.disableDacMute();

  //codec.enableLoopBack(); // Loopback sends ADC data directly into DAC
  codec.disableLoopBack();

  // Default is "soft mute" on, must disable mute to make channels active
  codec.disableDacMute();

  codec.enableHeadphones();
  codec.enableOUT3MIX(); // Provides VMID as buffer for headphone ground

  Serial.println("Volume set to +0dB");
  codec.setHeadphoneVolumeDB(0.00);

  Serial.println("Codec Setup complete. Listen to left/right INPUT1 on
Headphone outputs.");

 };
```

The above file first declares an instance object of the **WM8960** class found in **codec.h** and **codec.cpp**, calling it "**codec**". Now the "**codec**" object can access all the **WM9060** Class register set methods declared in **codec.h** and defined in **codec.cpp** using the simple **dot** operator, `codec.disableDacMute( )` for example. This **codec_sets( )** method is then executed at the start of **main.cpp** in its **setup( )** section to initialize the codec operation mode to enable ADC inputs and DAC outputs at a 44,100 Hz sample rate.

The Set_Settings File

The **set_settings.h** file is where all the important program settings are set and labeled such as sample-rate, bits-per-sample, number of audio channels, ESP32 pin numbers for all the physical pot and switch connections, ESP32 pin numbers for the i2c interface and the i2s codec interface, audio processing settings for DMA size and Framesize.

set_settings.h also declares the **I2S_init( )** function. If you remember, I2S is the codec interface used to move the audio data into and out of the codec. This function is defined in detail in **set_settings.cpp** and executed in the **setup( )** of **main.cpp**. It is the same I2S initialization function found the **Example 08** sketch. Many of the settings in **I2S_init( )** use labels defined in **set_settings.h** so that any needed changes can be applied to the labels while avoiding having to get inside **I2S_init( )**.

Here is the **set_settings.h** file specific to the Sparkfun Codec Thing PCB board including any possible external pots, pushbuttons and LEDs :

```
#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

//ESP32-Codec PIN SETUP
#define I2S_NUM               (0)
//#define IS2_MCLK_PIN (0)//onboard Osc Chip, MCLK of 24MHz
#define I2S_BCLK (4)   //BCLK, SCK, SCLK
#define I2S_LRC  (25)  //LRC, WS, ADCLRC, DACLRC, LRCLK -- Left/Right Chnl
#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
```

External Controllers

Many of the ESP32 Thing GPIO pins are assigned to specific tasks such as the I2C and I2S interfaces for the Codec and Qwiic, the SPI interface for the SD card, the MIDI serial interface, and a couple LEDS.

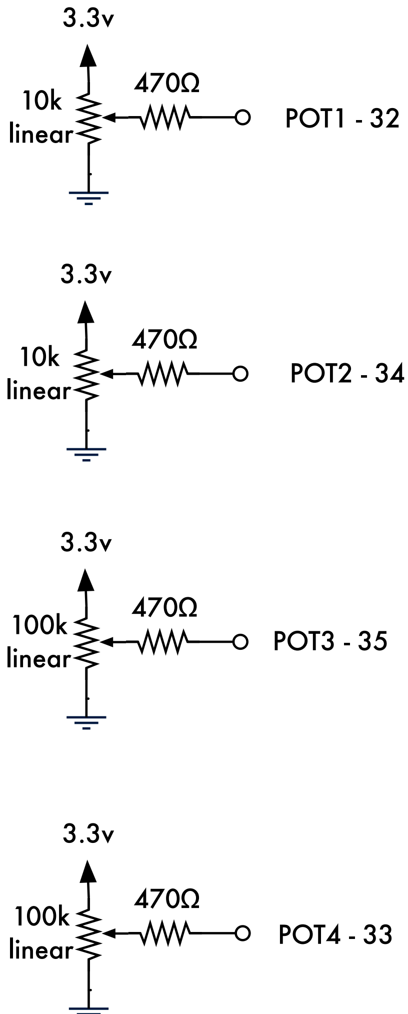
Believe it or not, there are pins left over and available for other tasks such as external controllers. These available GPIO pins are brought out to a 14-pin header for connecting to external controllers such as potentiometers and switches. Examples of controller circuits for these GPIO pins is shown in the figure below. The 470Ω resistors in series with the GPIO pins on the pot and pushbutton circuits are included as a safety feature preventing possible short circuits if the pins are mistakingly defined as outputs. For the LEDs, the 470Ω resistor sets the LED brightness. An example circuit for a cadmium cell light sensor was also included in the figure.

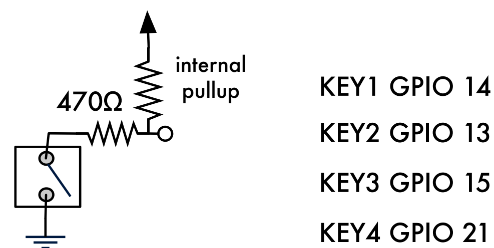
There are 14 "R" pads on the OSHPark PCB board for mounting 470Ω resistors for whatever controller circuits are needed. In effect, the 470Ω resistors will make the connection between a GPIO pin and an off-board pot, switch, or LED. In the case of the light sensor, a jumper wire on the "R" pad will suffice.

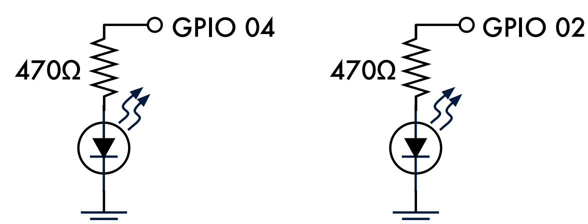
Note that the potentiometers can be any value between about 5k and 100k but must be linear taper.

ESP32 / PUCA
Sensor — Controller Circuits

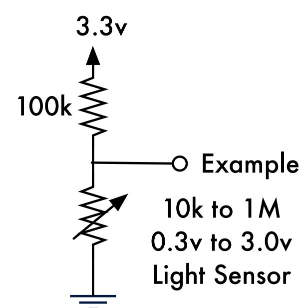
John Talbert 2023







470Ω's for pots and switches are for safety, in case pin is defined as an output



The Controller Module and Task Files

The **controller_mod.h** and **.cpp** file pair builds a base class, called **controllerModule**, that will define the basic data structures for all connected potentiometers, switches, and other sensors used to control the parameters of an audio effects program. Two arrays are created as class attribute members, one for up to 6 potentiometers or other analog sensors called **control[ ]**, and another one for up to 6 switches or other digital sensors called **button[ ]**. Each array element, in turn, has several properties such as name, GPIO pin, mode of operation, and value.

The **task.cpp** file creates a task function for the **button[ ]** array elements and one for the **control[ ]** array elements. These tasks will continuously poll all the physical controllers attached to the ESP32. Using the array element **pin** parameter, the buttontask( ) will perform a **digitalRead(pin)** and the controltask( ) will perform an **analogRead(pin)**. The data is then manipulated according to the controller's **mode** parameter and the result stored in the **value** parameter. This is done for each enabled button and control, and then repeated in an infinite loop.

The code in the **controller_mod** and **task** files described above is set and generally will not need any alterations. It's in the **set_module** files that all the controller details are worked out in a child class of **controllerModule** called **controller_module**.

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

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

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

void controller_module::init()  //effect module class initialization
 {
  name = "SparkfunGain";

  // Set up pin Modes for the switches and LEDs
  // Some need pullup resistor.
  pinMode(KEY1, INPUT_PULLUP);
  pinMode(LED1, OUTPUT);

  //setting up the buttons
  button[0].name = "Mute";
  button[0].mode = BM_TOGGLE;
  button[0].touch = false;
  button[0].pin = KEY1;  //label from set_settings.h


//add gain control
  control[0].name = "Gain";
```

```
  control[0].mode = CM_POT;
  control[0].levelCount = 128;
  control[0].pin = POT1;  //label from set_settings.h

  // special child class attributes
  gain = 1.0;
  gainRange = 2.0;
  mute = false;
 }
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::onButtonChange(int buttonIndex)
{
  switch(buttonIndex)
  {
    case 0: //pushbutton button[0] state has changed
    {
      if(button[0].value) //if effect is activated
      {
        codec.disableDacMute();
        digitalWrite(LED1, HIGH);
        mute = true;
      }
      else //if effect is bypassed
      {
        codec.enableDacMute();
        digitalWrite(LED1, LOW);
        mute = false;
      }
      break;
    }

  }
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::onControlChange(int controlIndex)
{
  switch(controlIndex)
  {
    case 0:  // potentiometer control[0] has changed
    {
      gain = (float)control[0].value/127.0;
      break;
    }
  }
}
```

In this simple effects demonstration three external controllers are needed - one pushbutton, one potentiometer and one LED. Most of the setup for these three devices happens in the **init( )** method shown above. First, the pushbutton and LED GPIO pins are set up with the usual **pinMode( )** function. Next the parameters of the pushbutton are defined in **button[0]** and the parameters of the pot are defined in **control[0]**.

The control and button tasks in the file **task.cpp** will use these parameters to continuously poll and store the pushbutton and pot values, waiting for any changes. When a change happens the tasks will call the methods **onButtonChange( )** or **onControlChange( )** which are defined next in **init( )**.

The pushbutton's **onButtonChange( )** method will act as an audio mute control directly calling the **codec.h** register set methods **enableDacMute( )** and **disableDacMute( )** with the LED acting as a mute indicator lamp.

The potentiometer's **onControlChange( )** method will take the collected pot value and turn it into a floating point "**gain**" variable that varies between zero and one.

This controller **init( )** method will be engaged in **main.cpp** with `myPedal->init( )` immediately before the task functions are set in motion with `taskSetup( )`.

The Main File

```
#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];
    float rxl, rxr, txl, txr;

    myPedal->init();
    taskSetup();

  while(1){   //signal processing loop

/*
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

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

    txl = myPedal->gain * myPedal->gainRange * rxl;
    txr = myPedal->gain * myPedal->gainRange * rxr;

    txbuf[i]   = ((int16_t) txl) ; //convert sample back to integer
    txbuf[i+1] = ((int16_t) txr) ;
  }

  // play processed 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 file **main.cpp** is the official program entry point. Here is a list of all its startup functions and from what files they originate.

`Serial.begin(115200);`	Arduino Library	Serial Monitor
`Wire.begin();`	Wire Library	start I2C 2-Wire
`codec.begin();`	**codec.cpp**	connect the I2C port
`codec_sets();`	**set_codec.cpp**	set codec registers
`I2S_init();`	**set_settings.cpp**	start I2S interface
`runSystemMonitor();`	**task.cpp**	print to screen controller values
`myPedal->init();`	**set_module.cpp**	set up controllers
`taskSetup();`	**task.cpp**	start up controller polling

After all the startup functions are initiated, an inner **while(1)** loop is entered. This infinite loop contains the code for digital signal processing on the audio samples. It starts off collecting a batch (frame) of audio signal samples from the DMA memory buffers holding data from the ADC converters, using the function **i2s_read( )**. These samples are then converted to floating point and processed one sample at a time. The frame of processed samples are then converted back to 16-bit integers and sent out to the DMA buffers serving the DAC converters, using the function **i2s_write( )**.

The only two lines that you really need to be concerned with are these:

```
txl = myPedal->gain * myPedal->gainRange * rxl;
txr = myPedal->gain * myPedal->gainRange * rxr;
```

This is the code that performs digital signal processing on the left and right channel audio samples. In this case the audio effect is simple amplitude control. The left and right samples are multiplied by both the 0 to 1 **gain** variable and the **gainRange** variable set up in the file **set_module.cpp**. If you remember, the **gain** value originates from an external potentiometer.

This is a rather trivial example and one that might be more easily accomplished by using a codec register set function such as `setDacLeftDigitalVolume(uint8_t volume)` inside the **set_module.cpp** file's **init( )** method, as done with the pushbutton **mute** control.

More importantly though, it is a simple demonstration of how to set up any effects processing on the samples of an audio signal. The bulk of the **main.cpp** file shown above can be left as is. Only the above two DSP code lines need to be changed to create a multitude of different effects. This is where the **bsdsp** file DSP tools will become indispensible.

For an example stereo chorus effect using the **bsdsp delay** and **oscillator** class tools, read the documentation for some of the other codec boards on the https://jtalbert.xyz/ESP32/ site.

The audio sample processing lines for the stereo chorus effect look like this:

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~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));
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

These lines take the place of the two amplitude control lines in our simple demonstration above. **delay1**, **delay2**, **lfo1**, and **lfo2** are instance objects of the **delay** class and **oscillator** class defined in the file **bsdsp.cpp**. These class objects along with the variables **dt1**, **dt2**, **depth**, **phaseDiff**, **stereo**, and **asynch** are all set up in the file **set_module.cpp** within the **init( )** method. For more details and a look at the stereo chorus **set_module.cpp,** read the PDF documents for the other codec boards.

Conclusion

For any new audio effect or different codec application the code must be rewritten in some of the software package files and left alone in others. This **Codec Effects Software Package** was designed to narrow down and clarify where those changes need to happen. Basically the coder only needs to examine four short files: **set_settings**, **set_codec**, **set_module**, and **main.cpp** (the "set" files are **.h/.cpp** pairs).

Each of these files was examined for the above simple amplitude control effect. For more details and tutorials please read the other codec PDF documents on the website.