LillyGo TAudio Effects Software With Neopixels and SD Card

John Talbert - January 2023

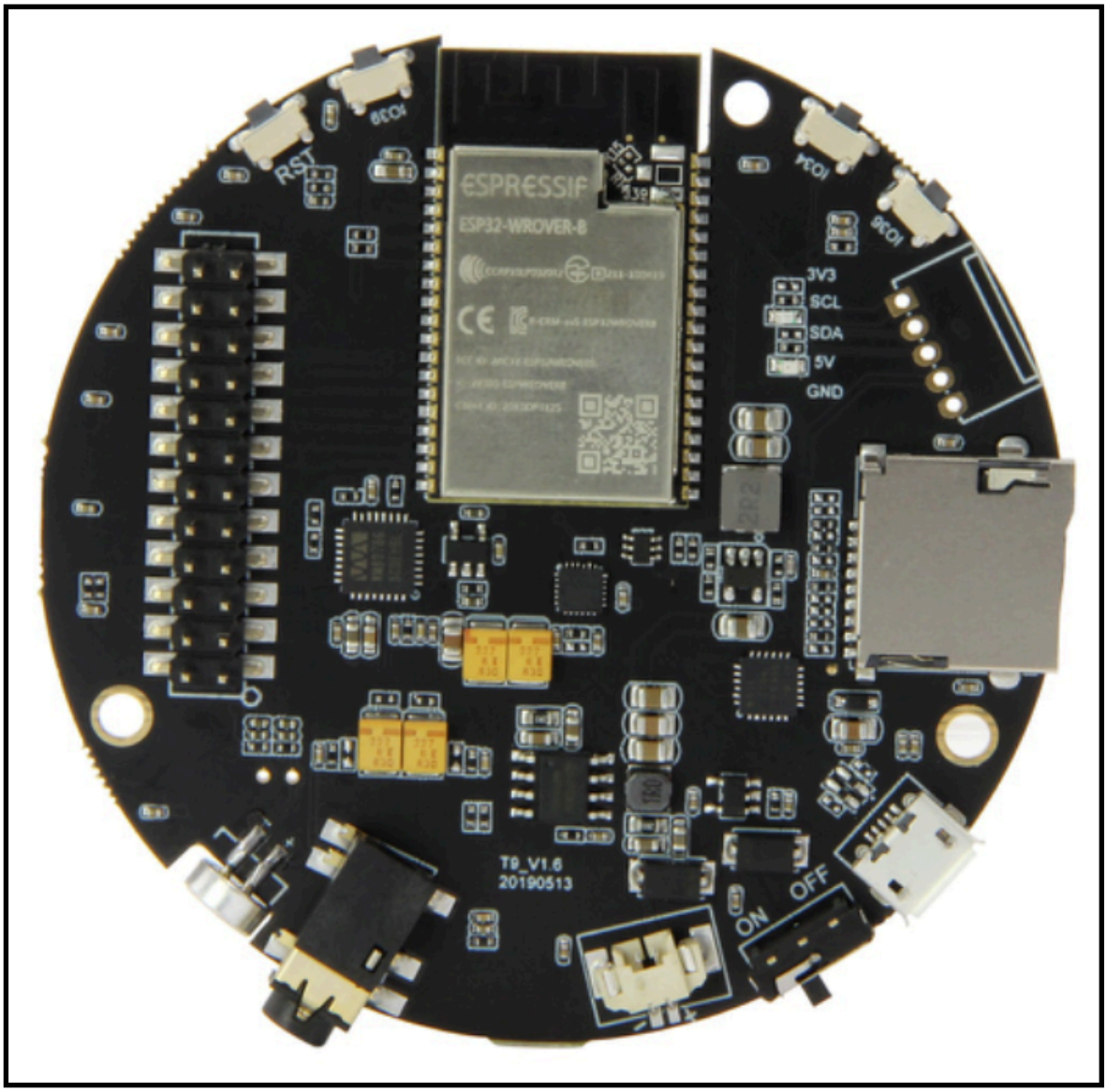


Table of Contents

Codec Effects Software

What is offered here is a complete Effects Programming Software package for the LillyGo TAudio DSP audio development board. It is basically the same software package found on my https://www.jtalbert.xyz/ESP32/ website with only a few changes specific to this board, namely the incorporation of NeoPixels and an SD Card playback.

Please read the tutorial PDF "**Codec Software**" for a complete, more detailed description of the codec effects software package and the Effects example used here.

The code was written on the PlatformIO IDE with an Arduino Framework, all from Visual Studio Code (VSC) as shown in the **platformio.ini** file:

```
[env:esp32dev]
platform = espressif32@5.2.0
board = esp32dev
framework = arduino
monitor_speed = 115200
lib_deps =
adafruit/Adafruit NeoPixel@^1.10.7
SPI
Wire
SD
```

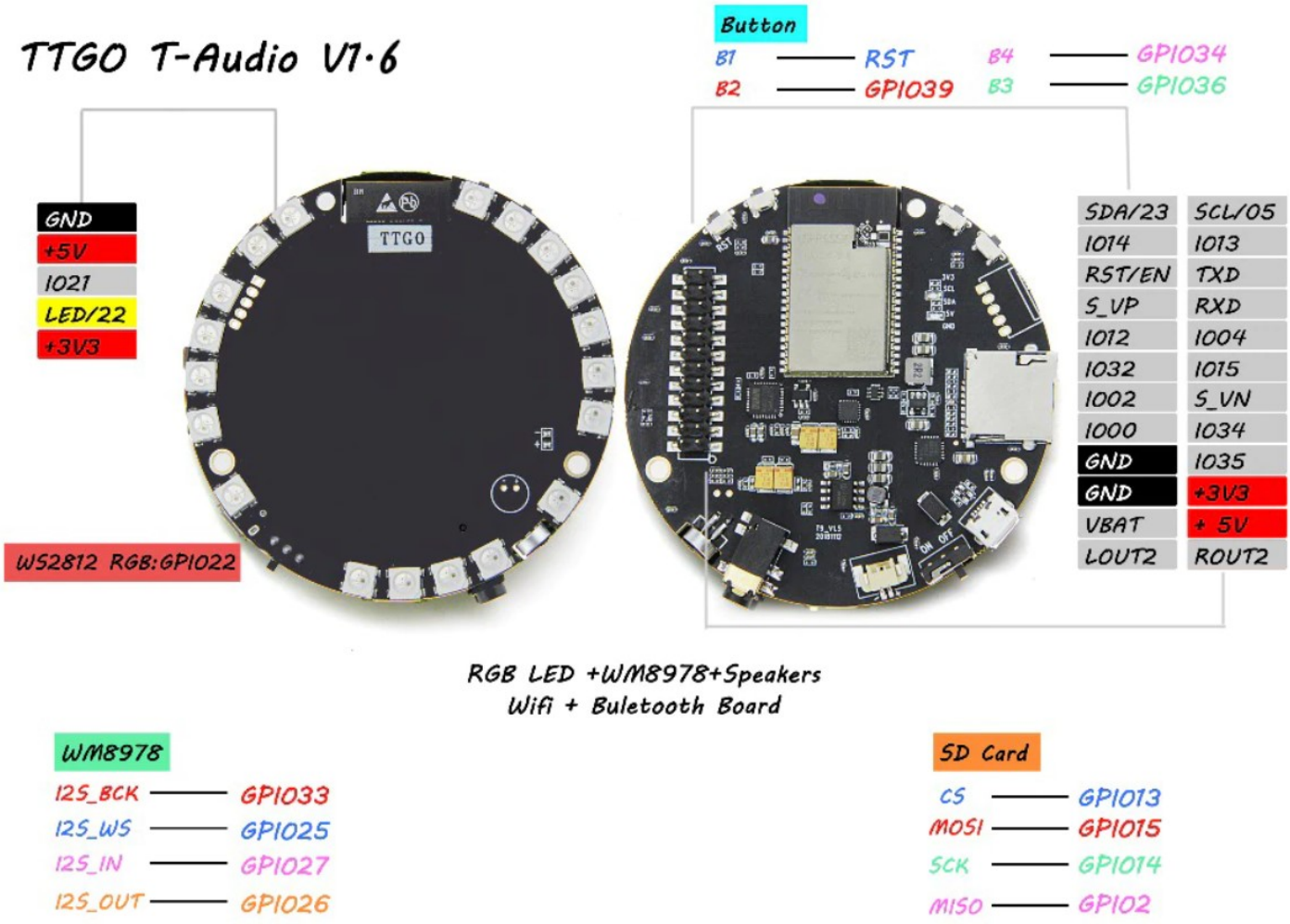
This board uses the WM8978 Codec, the same one used on the PÚCA. The codec driver, used in the codec files -- **codec.h**, **codec.cpp**, **set_codec.h**, and **set_codec.cpp**, was readily available from the PÚCA GitHub.

The LillyGo TAudio DSP

The TTGO TAudio board by LillyGo is an open-source, Arduino-compatible ESP32 development board for audio and digital signal processing (DSP) applications (http://www.lilygo.cn/prod_view.aspx?TypeId=50063&Id=1171). It is no longer on the LillyGo product list and is hard to find. Here we will mainly be using it to demonstrate the addition of NeoPixels and SD Card audio playback to our Codec Software Package.

Here is a list of its features:

Espressif official ESP32-WROVER module
WIFI
bluetooth
4MB Flash
4MB PSRAM
Lithium battery interface, 500mA Max charging current
MicroSD Card Slot, support SD and SPI mode
Audio WM8978 Codec
12Bits WS2812B RGB, 19 NeoPixel LEDs
MPU9250 Gyroscope Compass 9-Axis Sensor



TAudio Controller Hardware

The figure above illustrates some of the hardware features of the TTGO TAudio board along with GPIO pin connections. Here is a list of those features with short descriptions.

1. A ESP32 Wrover3 processor by Espressif.

2. A Wolfrom WM8978 Audio Codec with I2S and I2C interfaces.

   The four I2S interface connections are shown in the figure. The Master Clock **MCLK** pin used is **GPIO 0**. The I2C pin connections are **I2C_SDA GPIO-19** and **I2C_SCL GPIO-18**.

   There are two stereo DAC outputs readily available. **Lout1** and **Rout1** are each connected to a stereo Miniphone Headphone jack after a series resistor and capacitor. **Lout2** and **Rout2** are available straight from the WM8978 pins to the convenient 24-pin header. **Out3** and **Out4** are left unconnected.

   The only readily available ADC input is one electret microphone which is connected to both **L1** (plus and negative) and **R1** (plus and negative). The plus and minus mic connections also run to two circuit board hole pads marked + and - and also marked with a white circle on the rear of the board. These may be useful for an external microphone. The other Codec audio inputs - **L2, R2, AuxL** and **AuxR** - unfortunately, are left unconnected and thus pretty much unaccessible.

3. **MPU9250** 9 DOF 16 Bit Gyroscope Acceleration Magnetic Sensor 9-Axis Attitude +Gyro+Accelerator+Magnetometer Sensor Module

   This motion sensor shares the same I2C interface with the Codec on pins **I2C_SDA GPIO-19** and **I2C_SCL GPIO-18**, both with pullup resistors. Its chip select **AD0/SD0** is tied high.

   There are several online libraries available for this sensor.

4. 19 RGB NeoPixel LEDs on the back of the board.

   These are controlled from **GPIO-22**. A green LED is also connected to this pin. Pads for a header, including **GPIO-22** and **21** and power, are shown on the figure above. An on-board slide switch can turn of or on the power to the NeoPixels.

5. An SD Card Slot

   With connections for an SPI Interface are shown on the figure above.

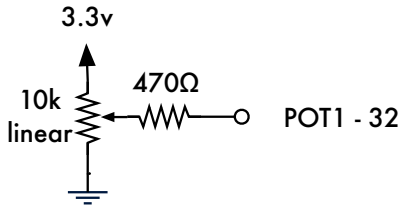
6. Three pushbutton switches along with Reset.

   3 normally-open pushbutton switches are connected to GPIO pins 34, 36(VP) and 39(VN). These pins are are also available on the 24-pin header. They can be used for analog controllers such as potentiometers but be sure to include a small series resistor, such as 470Ω, to protect against the 3.3v pot connection shorting to GND when the pushbutton is pressed.

7. 24-pin Header

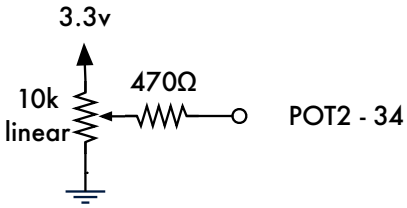
   This convenient header is detailed in the figure above. It has connections for codec audio outputs, a second I2S interface, power connections, the reset pin, and several GPIO pins available for external controllers.

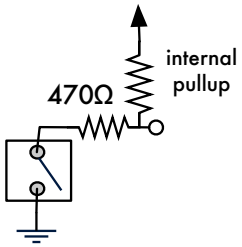
   The figure on the next page shows some simple circuits needed for hooking up potentiometers, switches, LEDs, and light sensors to the ESP32 GPIO pins. The pin connections shown are from another project, but they could easily be switched to any number of the GPIO pins available from the LillyGo 24-pin header.



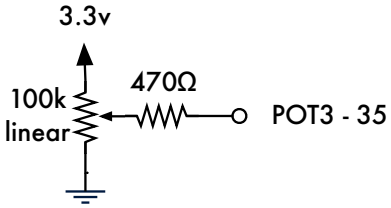
ESP32 / PUCA
Sensor – Controller Circuits

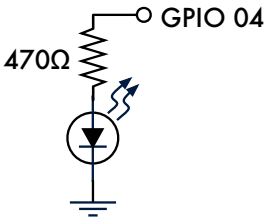
John Talbert 2023

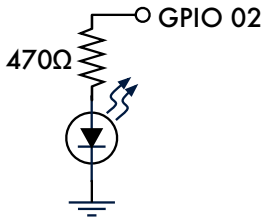


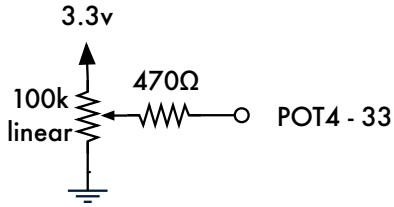


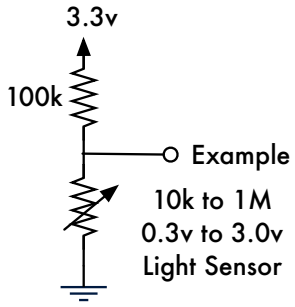
KEY1 GPIO 14
KEY2 GPIO 13
KEY3 GPIO 15
KEY4 GPIO 21











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

Codec Effects Software Package

There are a few GitHub repositories for the LillyGo TTGO TAudio board at https://github.com/LilyGO/TTGO-TAudio and https://github.com/goofy2k/TTGO-TAudio.

On my website at https://www.jtalbert.xyz/ESP32/ I've worked to create an audio effects software package for most any audio codec ESP32 board that is easier to use than the usual ADF/IDF libraries available and has full documentation and tutorials. This software package is available for download on this site along with this PDF. It is not yet available on a GitHub repository as I have not yet taken the time to set up an account and learn how to use it.

The package was created 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**.

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.

What follows is a short description of the files that make up this Codec Software Package for the ESP32.

1. **codec** -- The driver for a specific codec.

2. **controller_mod** -- A base class container for all possible 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. A setup function to place and start up the tasks.

4. **bsdsp** -- Digital Signal Processing class tools for the effects.

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 with the Effect Processing code.

The LillyGo TTGO TAudio board is one of several ESP32/Codec projects that have successfully used this Effects Software Package. For a more complete description of the Codec Effects Software Package please read the documentation PDFs for these other Codec projects at https://www.jtalbert.xyz/ESP32/, especially the "**Codec Software**" pdf.

The remainder of the paper will describe how to extend this Codec Software to include NeoPixel functionality and Audio Playback from an SD Card Reader, both special features of the LillyGo TAudio board.

NeoPixel LEDs

Description

> The WS2812 Integrated Light Source — or NeoPixel in Adafruit parlance — is the latest advance in the quest for a simple, scalable and affordable full-color LED. Red, green and blue LEDs are integrated alongside a driver chip into a tiny surface-mount package controlled through a single wire. They can be used individually, chained into longer strings or assembled into still more interesting form-factors.
>
> Not all addressable LEDs are NeoPixels. "NeoPixel" is Adafruit's brand for individually-addressable RGB color pixels and strips based on the WS2812, WS2811 and SK6812 LED/drivers, using a single-wire control protocol.

The above description is from the Adafruit website at https://learn.adafruit.com/adafruit-neopixel-uberguide. There are basically three methods for driving a string of NeoPixels from a single digital line. The three methods are described in this web page, https://blog.ja-ke.tech/2019/06/02/neopixel-performance.html.

NeoPixels on TAudio

The TAudio has 19 NeoPixels mounted in a circle on the back of the circuit board. They are all driven from GPIO pin 22. Interestingly, this pin also controls a Green LED on the front of the board. Since loading all 19 LEDs with a color from GPIO 22 takes less than a millisecond, loading a steady high or low in between NeoPixel loads will work fine for setting the Green LED.

If more NeoPixels are needed, the 5-pin header with power, pin22 and pin21 can be used. Be sure to first consult the Arduino web tutorial for power requirements.

NeoPixel Library

The NeoPixels require a library to operate. Here I've chosen "**Adafruit_NeoPixel**". Loading a library into your software project is easy within **VSC** (Studio Visual Code). Click on the "Cricket" sidebar icon. Scroll through the menu **QuickAccess/ PIO_Home/Open**. The PlatformIO Window that comes up with the large Cricket icon has a sidebar with a "Libraries" icon. Clicking that will bring up a search window. Enter "Adafruit NeoPixels" to get a number of available libraries. Now you can explore everything about the library you select. Make sure the list of platforms for the library includes "**Espressif 32**" which means it will work with the ESP32 on your board.

Once you are sure about the library you want, click "Add to Project" and select the project name from a list of all your software projects. The installer will then download the library to the **.pio/libdeps** folder in your project and it will make a note of the install in the **platformio.ini** file with the line

```
lib_deps =
    adafruit/Adafruit NeoPixel@^1.10.7
```

Note the version number for the library. Sometimes project code can break with a new version of a library. To prevent that from occurring, the actual library, at a specific version, is stored with your project in the **.pio** folder.

Conveniently, all the library files can be viewed from the VSC project menu. Here I chose the library Example file "**strandtest**" to incorporate into my own project files.

The Set_Settings File

The first step in incorporating NeoPixel functionality into our Software Package is to define any constants such as GPIO pin connections among other things. This is done in the set_settings.h file.

```
#ifndef SETTINGS_H_
#define SETTINGS_H_

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

  #define SAMPLE_RATE     (32000)
  #define BITS_PER_SAMPLE (16)
  #define CHANNEL_COUNT 2

  //LILLYGO T_AUDIO PIN ASSIGNMENTS
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~

  #define POT1 32
  #define POT2 34
  #define POT3 35
  #define POT4 36
  #define POT5 39

  #define LED1 22

  #define KEY1 39
  #define KEY2 34
  #define KEY3 36
  #define KEY4 23

  #define TOUCH_THRESHOLD 30

//ESP32-Codec PIN SETUP
#define I2S_NUM               (0)
#define IS2_MCLK_PIN (0)
#define I2S_BCLK    (33)
#define I2S_LRC     (25)
#define I2S_DIN     (27)
#define I2S_DOUT    (26)

#define Codec_SDA         19
#define Codec_SCK         18
#define I2C_MASTER_SCL_IO 18
#define I2C_MASTER_SDA_IO 19
#define I2C_SDA           19
#define I2C_SCL           18

#define Codec_ADDR    0x1A  //WM8978
#define WM8978_ADDR   0X1A  //WM8978
```

```
#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 / SPI interface
#define SD1 04
#define SD2 12
#define SD3 13
#define SDA 23  //??

#define SD_CARD_CS 13
#define SD_CARD_MISO 02
#define SD_CARD_MOSI 15
#define SD_CARD_CLK  14

#define SAMPLES_BUFFER_SIZE 1024

//NEO PIXEL SETTINGS
#define PIN 22
#define NUM_LEDS 19
#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
```

In this file pin labels are assigned to any possible controllers such as pots and switches, LEDs such as LED1 assigned to pin 22, the I2S pins used by the Codec, the I2C pins used by the Codec and Motion Sensor, the SPI interface pins used by the SD Card, DMA memory settings used by the Codec, and Codec audio sample settings.

Specific to the NeoPixels, **PIN** is the label for the serial pixel data communications line, the number of LEDs on the TAudio board is labeled as **NUM_LEDS**, a general LED brightness value from 1 to 10 is given the label **BRIGHTNESS**.

```
//NEO PIXEL SETTINGS
#define PIN 22
#define NUM_LEDS 19
#define BRIGHTNESS 5
```

The Task File

The package **task** file contains several tasks set up as simple functions: a task to poll all analog controller values called **controltask( )**, a task to poll all the digital controller values called **buttontask( )**, a task to print to the monitor running controller values and CPU loads. These tasks must all appear to be running simultaneously even though they compete for the same processor time - a job for **FreeRTOS**.

Each task function is slowed down a bit with the **RTOS** tool **VTaskDelay(wait time)**. This is a non-blocking delay that allows other task threads to get processor time while another task is in a **VTaskDelay** wait. The wait time for polling controller values can be set as high as 20 ms without any perceived slowing of response times for manual controllers.

The task file function **taskSetup( )** is used to activate all these task functions with the **RTOS** tool **xTaskCreatePinnedToCore( )** as shown here:

```
//the main audio processing task is placed in the main loop() of main.cpp
(core1)

//decoding button presses
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);
```

Each task is assigned one of the two ESP32 processor cores, zero in this case, as indicated by the final zero element in the **xTask** function. Processor core 1 is reserved for the more time intensive and time sensitive audio DSP task.

Notice the **neopixeltask** line among the four activated tasks. The **task** file is the perfect place to insert NeoPixel functionality in our Software Package.

The NeoPixel Task

One **Example** file from the Adafruit NeoPixel Library was selected to serve as the basis of a **neopixeltask( )**. As usual, in c++ programming code placement is one of the more difficult problems to overcome. Coding the **neopixeltask( )** function was not as simple as placing the entire **Example** file into a function. It must be carefully spread out within the **task.cpp** file. Here is the task file code that applies to the neopixel task. First, in the **task.h** header file:

```
#include "set_settings.h"
#include "set_module.h"
#include <Adafruit_NeoPixel.h>

void controltask(void* arg);  //task for pots
void buttontask(void* arg);   //task for switches
void neopixeltask(void* arg); //task for NeoPixel LEDs
void acceltask(void* arg);  //task for MPU9250 accel
```

Note the **#include** line for the Adafruit NeoPixel Library, and the **neopixeltask( )** function declaration line.

Then, in the task.cpp code more detailed definition file:

```
// Parameter 1 = number of pixels in strip
// Parameter 2 = Arduino pin number (most are valid)
// Parameter 3 = pixel type flags, add together as needed:
// NEO_KHZ800  800 KHz bitstream (most NeoPixel products w/WS2812 LEDs)
// NEO_KHZ400  400 KHz (classic 'v1' (not v2) FLORA pixels, WS2811 drivers)
// NEO_GRB   Pixels are wired for GRB bitstream (most NeoPixel products)
// NEO_RGB   Pixels are wired for RGB bitstream (v1 FLORA pixels, not v2)
// NEO_RGBW  Pixels are wired for RGBW bitstream (NeoPixel RGBW products)

//  Create an instance object of Adafruit_NeoPixel called strip
Adafruit_NeoPixel strip = Adafruit_NeoPixel(NUM_LEDS, PIN, NEO_GRB +
NEO_KHZ800);

  //~~~~~~~~~~~NeoPixel Functions~~~~~~~~~~~~~~~~
  //~~~~~~~~~~used in neopixeltask( )~~~~~~~~~~~~
  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  // Input a value 0 to 255 to get a color value.
  // The colours are a transition r - g - b - back to r.
  uint32_t Wheel(byte WheelPos) {
    WheelPos = 255 - WheelPos;
    if(WheelPos < 85) {
      return strip.Color(255 - WheelPos * 3, 0, WheelPos * 3);
    }
    if(WheelPos < 170) {
      WheelPos -= 85;
      return strip.Color(0, WheelPos * 3, 255 - WheelPos * 3);
    }
    WheelPos -= 170;
    return strip.Color(WheelPos * 3, 255 - WheelPos * 3, 0);
  }

  // Fill the dots one after the other with a color
```

```
  void colorWipe(uint32_t c, uint8_t wait) {
    for(uint16_t i=0; i<strip.numPixels(); i++) {
      strip.setPixelColor(i, c);
      strip.show();
      vTaskDelay(wait);
    }
  }

  void rainbow(uint8_t wait) {
    uint16_t i, j;

    for(j=0; j<256; j++) {
      for(i=0; i<strip.numPixels(); i++) {
        strip.setPixelColor(i, Wheel((i+j) & 255));
      }
      strip.show();
      vTaskDelay(wait);
    }
  }

//Slightly different, this makes rainbow equally distributed throughout
  void rainbowCycle(uint8_t wait) {
    uint16_t i, j;

    for(j=0; j<256*5; j++) { // 5 cycles of all colors on wheel
      for(i=0; i< strip.numPixels(); i++) {
        strip.setPixelColor(i, Wheel(((i * 256 / strip.numPixels()) + j) &
255));
      }
      strip.show();
      vTaskDelay(wait);
    }
  }

  //Theatre-style crawling lights.
  void theaterChase(uint32_t c, uint8_t wait) {
    for (int j=0; j<10; j++) {  //do 10 cycles of chasing
      for (int q=0; q < 3; q++) {
        for (uint16_t i=0; i < strip.numPixels(); i=i+3) {
          strip.setPixelColor(i+q, c);    //turn every third pixel on
        }
        strip.show();

        vTaskDelay(wait);

        for (uint16_t i=0; i < strip.numPixels(); i=i+3) {
          strip.setPixelColor(i+q, 0);        //turn every third pixel off
        }
      }
    }
  }

  //Theatre-style crawling lights with rainbow effect
  void theaterChaseRainbow(uint8_t wait) {
    for (int j=0; j < 256; j++) {     // cycle all 256 colors in the wheel
      for (int q=0; q < 3; q++) {
        for (uint16_t i=0; i < strip.numPixels(); i=i+3) {
          strip.setPixelColor(i+q, Wheel( (i+j) % 255));    //turn every
third pixel on
        }
```

```
          strip.show();

          vTaskDelay(wait);

          for (uint16_t i=0; i < strip.numPixels(); i=i+3) {
            strip.setPixelColor(i+q, 0);        //turn every third pixel off
          }
        }
      }
    }


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~NEOPIXEL LED TASK~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void neopixeltask(void* arg)
{
  strip.setBrightness(BRIGHTNESS);
  strip.begin();
  strip.show(); // Initialize all pixels to 'off'

while(true)  //NeoPixel functions below defined at start of task.cpp
  {

    // Some example procedures showing how to display to the pixels:
    //colorWipe(strip.Color(255, 0, 0), 50); // Red
    //colorWipe(strip.Color(0, 255, 0), 50); // Green
    //colorWipe(strip.Color(0, 0, 255), 50); // Blue
    //colorWipe(strip.Color(0, 0, 0, 255), 50); // White RGBW

    int pixcolor = (int8_t) (myPedal->gain * 255); //control color

    // Send a theater pixel chase, "gain" controls red to blue
    theaterChase(strip.Color(pixcolor, 0, (255 - pixcolor)), 20);
    //theaterChase(strip.Color(127, 0, 0), 50); // Red
    //theaterChase(strip.Color(0, 0, 127), 50); // Blue

    //rainbow(20);
    //rainbowCycle(20);
    //theaterChaseRainbow(50);
  }
}
```

The first requirement is to create an instance object of the **Adafruit_NeoPixel** Library Class and call it "**strip**". This must happen at the start of the **task.cpp** file if we are to be granted access to the Adafruit Class methods from within the **neopixeltask( )** function as well as from within the specialized Pixel effects functions to follow.

```
Adafruit_NeoPixel strip = Adafruit_NeoPixel(NUM_LEDS, PIN, NEO_GRB + NEO_KHZ800);
```

This line will also trigger the Class **Constructor** Method which needs some information to set up the NeoPixel communication. Information such as the number of LED Pixels on the TAudio NeoPixel "**strip**", the GPIO pin to use for the serial data, and the NeoPixel type used. **NUM_LEDS** and **PIN** are labels defined in the **set_settings.h** file. **NEO_GRB** and **NEOKHZ800** are labels defined in the Adafruit

NeoPixel library and described here in the code comments.

Next to define are six short functions from the library **Example** file, each of which creates a specific NeoPixel effect. Consult the **Adafruit_NeoPixel** library to find out the details of how these work. Each pixel effects function originally included the Arduino **delay( )** to time the display effects. Those **delay( )** functions have all been replaced with the **RTOS VTaskDelay( )**.

All this preparation results in a super simple **neopixeltask( )**. It starts off with three **Adafruit_NeoPixel** library methods: **setBrightness**, **begin** and **show**. The details of these three methods can be found in the library. The bulk of the **neopixeltask( )** is then an infinite **while(1)** loop that continually repeats the pixel effects lined up within the loop. The **Example** script runs through quite a showcase of pixel effects using the pixel functions defined above, though here I've commented out all but one for the following demonstration.

An important advantage to placing the **neopixeltask( )** within the task.cpp file is that it has access to virtually all the controller parameters and variables set up in the **set_module** files. Here the "**gain**" variable will be used to affect both the NeoPixels and the audio playback volume.

The software effects package presented here and available for download on **jtalbert.xyz** is set up with a simple audio effect. Playback volume of a wav file from the SD Card is controlled from two board pushbuttons. The code in **set_module.cpp** creates the variable "**gain**" with a value from zero to one, controlled by the two pushbuttons, one for increasing the gain towards one, and another for decreasing the gain toward zero. This gain variable is multiplied with the audio samples coming from the SD wav file before they are presented to the Codec DAC resulting in output volume control.

```
int pixcolor = (int8_t)(myPedal->gain * 255);
// Send a theater pixel chase, "gain" controls red to blue
theaterChase(strip.Color(pixcolor, 0, (255 - pixcolor)), 20);
```

In the code above the **neopixeltask( )** function uses the same "**gain"** variable, accessed with "**myPedal->gain**", to affect the color of the NeoPixel ring. It is applied to the Red and Blue pixels of the RGB (red, green, blue) LEDs resulting in red pixels for full volume output, blue pixels for zero volume output, and all variations in between. This is just a simple example. Any pixel control you can imagine can be set up in the **set_module.cpp** file and **neopixeltask( )** function. See more detailed descriptions and examples of controller setups in the other PDF files on the **jtalbert.xyz** website.

SD Card Playback with Effects

The LillyGo TTGO TAudio board includes an SD Card Reader. The LyraT and A1S Audio Kit boards, described on my website https://jtalbert.xyz/ESP32/, also include SD Card Readers so this section applies to them as well. Alternatively, you can buy an external SD Card Module for about $10 with as few as 6 connections for the ESP32.

Here we will add SD Card audio playback with Effects to the "**Codec Effects Software Package**" with the creation of two new files -- **sd_play.h** and **sd_play.cpp**.

Acknowledgments are due to the excellent XTronical tutorials at https://www.xtronical.com/i2s-ep1/ upon which I based my SD Card Library. Also very helpful were some tutorials at Random Nerd Tutorials -- https://randomnerdtutorials.com/esp32-microsd-card-arduino/ and https://randomnerdtutorials.com/esp32-spi-communication-arduino/. https://microcontrollerslab.com/microsd-card-esp32-arduino-ide/

SPI Protocol

The microSD card slot communicates using **SPI** communication protocol and for this you need four GPIO lines:

- **MISO**: Master In Slave Out, SPI Output from the SD Card
- **MOSI**: Master Out Slave In, SPI Input to the SD Card
- **SCK**: Serial Clock
- **CS /SS**: Chip Select

By default ESP32 has two SPI communication channels **VSPI** and **HSPI**. Each has a default pin settings for **MOSI, MISO, SCK** and **CS**. **VSPI** is 23, 19, 18, 5 and **HSPI** is 13, 12, 14, 15. But we can also map these pins to other GPIO pins using the line

```
SPI.begin(SD_CARD_CLK, SD_CARD_MISO, SD_CARD_MOSI, SD_CARD_CS);
```

The LillyGo TAudio board uses unconventional GPIO pins for the SD SPI interface so the line above must be called to initialize the SPI protocol. The pins the TAudio uses for the SD SPI are set up in the file **set_settings.h**:

```
#define SD_CARD_CS   13
#define SD_CARD_MISO 02
#define SD_CARD_MOSI 15
```

```
#define SD_CARD_CLK  14
```

Libraries for the SD

The plan for adding SD audio playback to our Software Package is to create an **SDplay** Class with two new files, **sd_play.h** and **sd_play.cpp**. Three external libraries must be included in **sd_play.h**: **FS.h** to handle files, **SD.h** to interface with the microSD card and **SPI.h** to use SPI communication protocol.

These three libraries are part of the Arduino core for the ESP32. The Arduino core, **Espressif/Arduino-esp32**, should be automatically available when you designate **framework=arduino** during the Project setup; however, I've found that the compiler will not automatically recognize code from these libraries. You must use **#include** statements in **sd_play.h** for each of the libraries and/or designate them in the **platformio.ini** file with the lines:

```
lib_deps =
adafruit/Adafruit NeoPixel@^1.10.7
SPI
Wire
SD
```

Each compile build of Software Package will then confirm this with the following lines:

```
Dependency Graph
|-- Adafruit NeoPixel @ 1.10.7
|-- SPI @ 2.0.0
|-- Wire @ 2.0.0
|-- SD @ 2.0.0
|   |-- FS @ 2.0.0
|   |-- SPI @ 2.0.0
```

From this Dependency Graph it looks like **FS** and **SPI** libraries are included within the **SD** library so you may only need `#include <SD.h>` in the file **sd_play.h**.

Before examining **sd_play.h** and **sd_play.cpp** it might be useful to first take a look at the **SD** library methods available for our new **sd_play** files. The Arduino Reference page at https://www.arduino.cc/reference/en/libraries/sd/ contains a full description of the **SD** Library. From this site page you can download the latest **SD** Library zip, check out some example code, and explore all the library methods. As expected, the **SD** Library includes the File Library **FS** with methods listed for each:

SD class

- **begin()**
- exists()
- exists()
- mkdir()
- **open()**
- remove()
- rmdir()

File class

- name()
- available()
- close()
- flush()
- peek()
- position()
- print()
- println()
- **seek()**
- size()
- **read()**
- write()
- isDirectory()
- openNextFile()
- rewindDirectory()

The four methods I ended up using are marked in the above list. Other methods such as **openNextFile()**, **available()**, **exists()** look interesting for future revisions. Here are descriptions of the four methods:

SD - begin()

Initializes the SD library and card. This begins use of the SPI bus and the chip select pin. Note that even if you use a different chip select pin, the hardware SS pin must be kept as an output or the SD library functions will not work.

Syntax
SD.begin()
SD.begin(cspin)

Parameters
cspin (optional): the pin connected to the chip select line of the SD card; defaults to the hardware SS line of the SPI bus.
Returns
1 on success, 0 on failure.

SD - open()

Opens a file on the SD card. If the file is opened for writing, it will be created if it doesn't already exist (but the directory containing it must already exist).

Syntax
SD.open(filepath)
SD.open(filepath, mode)
Parameters
filepath: the name of the file to open, which can include directories (delimited by forward-slashes, /).
mode (optional): the mode in which to open the file. Mode can be FILE_READ (open the file for reading, starting at the beginning of the file) or FILE_WRITE (open the file for reading and writing, starting at the end of the file).

FS - read()

Read from the file. read() inherits from the Stream utility class.

Syntax
file.read()
file.read(buf, len)
Parameters
file: an instance of the File class (returned by SD.open()).
buf: an array of characters or bytes.
len: the number of elements in buf.
Returns
The next byte (or character), or -1 if none is available.

FS - seek()

Seek to a new position in the file, which must be between 0 and the size of the file (inclusive).

Syntax
file.seek(pos)
Parameters
file: an instance of the File class (returned by SD.open()).
pos: the position to which to seek (unsigned long).
Returns
1 on success, 0 on failure.

The SD_PLAY.h File

```
/*    -------------------------------------------------------

Title: SD Card Wav Player

Description:
   Simple example to demonstrate the fundementals of playing WAV
files(digitised sound) from an SD Card via the I2S interface of the ESP32.
Plays WAV file from SD card. To keep this simple the WAV must be stereo and
16bit samples. The Samples Per second can be anything. On the SD Card the
wav file must be in root and called wavfile.wav  Libraries are available to
play WAV's on ESP32, this code does not use these so that we can see what
is happening.  This is part 3 in a tutorial series on using I2S on ESP32.
See the accompanying web page (which will also include a tutorial video).

Boring copyright/usage information:
(c) XTronical, www.xtronical.com
Use as you wish for personal or monatary gain, or to rule the world.
However you use it, no warrenty is provided etc. etc. It is not listed as
fit for any purpose you perceive  It may damage your house, steal your
lover, drink your beers and more.

http://www.xtronical.com/i2s-ep3

    ---------------------------------------------------------------
*/

#ifndef __SDPLAY_H
#define __SDPLAY_H

#include <Arduino.h>
#include "SD.h"   // SD Card library, usually part of the standard install
#include "driver/i2s.h" // Library of I2S routines, ESP32 standard install
#include "set_settings.h"

class SDplay
{
 public:

    File WavFile;

    static byte Samples[];          // Array for frame data from WAV
    static uint32_t BytesReadSoFar;// Number of bytes read from file so far
    static uint16_t BufferIdx;      // Current pos of buffer to output next

    struct WavHeader_Struct
      {
          //   RIFF Section
          char RIFFSectionID[4];      // Letters "RIFF"
          uint32_t Size;              // Size of entire file less 8
          char RiffFormat[4];         // Letters "WAVE"

          //   Format Section
          char FormatSectionID[4];    // letters "fmt"
          uint32_t FormatSize;        // Size of format section less 8
          uint16_t FormatID;          // 1=uncompressed PCM
          uint16_t NumChannels;       // 1=mono,2=stereo
```

```
            uint32_t SampleRate;        // 44100, 16000, 8000 etc.
            uint32_t ByteRate; // =SampleRate * Channels * (BitsPerSample/8)
            uint16_t BlockAlign;        // =Channels * (BitsPerSample/8)
            uint16_t BitsPerSample;     // 8,16,24 or 32

            // Data Section
            char DataSectionID[4];      // The letters "data"
            uint32_t DataSize;          // Size of the data that follows
          }WavHeader;

 private:

    size_t BytesWritten;         // Returned by the I2S write routine,
    uint16_t BytesToRead;        // Number of bytes to read from the file
    uint8_t* DataPtr;            // Point to next data to send to I2S
    uint16_t BytesToSend;        // Number of bytes to send to I2S
    bool done = false;

 public:
    void ReadFile(byte* Samples);      // Read WAV file from SD
    void FillI2SBuffer(byte* Samples); // Write SD Buffer to DMA tx buffer
    void SDCardInit();
    void OpenWaveFile();
    bool ValidWavData(WavHeader_Struct* Wav); //Check for valid types
    void PrintWAVHeader(WavHeader_Struct* Wav); //Print WAV Header
    void PrintData(const char* Data,uint8_t NumBytes); // Print char bytes

};

#endif
```

This is the "**header**" file for our new Class **SDplay**. It lists or "declares" all the class elements which are to be "defined" in detail in the **sd_play.cpp** file. It first declares all the "attributes" (variables) used in the class and then the "methods" (functions) used. Let's look at a few of the attributes.

WaveFile

```
File WaveFile
```

"**File**" is the class created in the **FS** Library. Here "**WaveFile**" is declared as an instance object of the class "**File**". As such it can access any of the **FS** Library methods listed in the previous section such as **WaveFile.read( )** and **WaveFile.seek( )**. However, it has yet to be empowered by associating it with an actual file. That will happen within the method **OpenWaveFile( )** as shown below from the **sd_play.cpp** file.

```
void SDplay::OpenWaveFile()
{
    WavFile = SD.open("/wavfile.wav"); // Open wavfile.wav at root of SD
    if(WavFile==false)
      Serial.println("Could not open 'wavfile.wav'");
```

```
    else
    {
      // Read in the WAV header, which is first 44 bytes of the file.
      // We have to typecast to bytes for the "read" function
      WavFile.read((byte *) &WavHeader,44);

      // Print the header data to serial, optional!
      PrintWAVHeader(&WavHeader);

      // optional if you're sure the WAV file will be valid.
      if(ValidWavData(&WavHeader))

      Serial.println("Wav file is valid'");
    }
}
```

The line `WavFile = SD.open("/wavfile.wav");` uses the **open( )** method from the **SD** Class. "**SD**" has already been defined as an instance object of the **SD** class. The **File** object **WavFile** is now associated with an actual audio wav file "**wavfile.wav**" on the SD card. It is expected that an audio file has already been loaded onto the SD card and it is called "wavfile.wav". If not, the user will sadly be informed that it "Could not open 'wavfile.wav' " printed out on their screen Monitor.

Samples[ ]

Samples[ ] is an array of bytes as declared in the **sd_play.h** file with the line `static byte Samples[];` . The "**static**" keyword means that only one **Samples[ ]** array will exist to be shared by all instance objects of the **SDplay** Class and within all the Class methods.

The **Samples** buffer array will be filled with audio samples from **wavfile.wav** on the SD Card using the command `WavFile.read(Samples,BytesToRead);` Memory for the Samples array is allocated in the file **sd_play.cpp** with this line:

```
byte SDplay::Samples[SAMPLES_BUFFER_SIZE];
```

The label **SAMPLES_BUFFER_SIZE** is defined, along with other **#define** Labels, in the file **set_settings.h**. Currently it is set at 1024 bytes. This value is also referred to as the sample **Framesize**. A **Frame** is a chunk of audio samples that will be downloaded from the **wavfile** for processing before being dumped to the **DAC DMA** memory buffers.

As a rule, **DSP** (Digital Signal Processing) can only happen piecemeal, on a small **frame** of samples with the following steps: first, access to the SD Card is set up, a frame of samples are read from the wavfile, the frame of samples are processed for the desired audio effect, assess to the DMA buffers is set up, the frame of samples are

loaded into the DMA buffers to be sent to the DACs one at a time at the set SampleRate.

With the **DMA** buffers constantly pulling out samples to feed the **DAC** you run the risk of the DMA DAC buffer running out of samples if the **DSP** takes too long on a **frame** of data that's too large. For this reason the signal processing is always done on smaller framesize chunks of samples instead of the entire wavfile. Similarly, if the framesize is too small, the increased overhead time used in repeatedly setting up access to the SD Card and DMA also runs the risk of not getting the samples out in a timely manner. Framesize and DMA buffer sizes can both be adjusted in **set_settings.h** to avoid such problems.

On another point, a notable property of all C++ arrays is that the array name, without the brackets, acts as a pointer to the memory address of the start of the array data. To keep track of where you are in the array you can create a pointer into the array with the line:

```
DataPtr=Samples+BufferIdx;
```

The Samples[ ] array is used in two **SDplay** Class methods, **ReadFile( )** and **FillI2SBuffer( )** to be described later.

WaveHeader Struct

"**WaveHeader**" is a Class attribute **struct**. A "**struct**" is a catch-all collection of different data types. It essentially operates like a **Class** definition. Elements of the **struct** are accessed with a **dot** operator like `WavHeader.DataSize` "**WavHeader_Struct**" is a name given to the "**type**" of **struct**. It is useful for defining the input parameters of functions or methods.

```
bool ValidWavData(WavHeader_Struct* Wav); // Check WAV Header for valid types
void PrintWAVHeader(WavHeader_Struct* Wav); // Print WAV Header to Monitor
```

In the above method declarations "**Wav**" is declared as the input parameter to these methods. When the method is called, a "**WavHeader_Struct**" type of **struct** is required as the input "Wav" ("**WaveHeader**", for example). The asterisk (*) indicates that **Wav** is actually a pointer to a **WavHeader_Struct** type **struct**. Looking at the details of how these two methods are defined in the file **sd_play.cpp** you can see that pointers to a **struct** don't access elements of the **struct** with the **dot** operator. Instead they use the **->** arrow operator like `Wav->DataSize`

The method **OpenWaveFile( )** shows what these two methods look like when called.

```
// Print the header data to serial, optional!
PrintWAVHeader(&WavHeader);

// optional if you're sure the WAV file will be valid.
if(ValidWavData(&WavHeader)) Serial.println("Wav file is valid'");
```

The input parameter to each of these methods will be **WavHeader**, an obvious **WavHeader_Struct** type of **struct**; however, since the methods require a pointer to a **struct**, a "**&**" is attached making it instead, a pointer to **WavHeader**.

So what is the data contained in **WavHeader**? WAV or Waveform Audio File Format was developed jointly by Microsoft and IBM as an audio file standard for storing digital audio on PC. The bulk of a wav file is uncompressed audio which is easily edited and manipulated. The standard, most common wav file has a 44 byte header at the start of the file before any audio data. The **WavHeader struct** reveals how this header is structured. In particular, it contains information about the audio samples:

```
 // Format Section
          char FormatSectionID[4];     // letters "fmt"
          uint32_t FormatSize;         // Size of format section less 8
          uint16_t FormatID;           // 1=uncompressed PCM
          uint16_t NumChannels;        // 1=mono,2=stereo
          uint32_t SampleRate;         // 44100, 16000, 8000 etc.
          uint32_t ByteRate;           // =SampleRate*Channels*(BitsPerSample/8)
          uint16_t BlockAlign;         // =Channels * (BitsPerSample/8)
          uint16_t BitsPerSample;      // 8,16,24 or 32
// Data Section
          char DataSectionID[4];       // The letters "data"
          uint32_t DataSize;           // Size of the data that follows
```

One important element, **WavHeader.DataSize,** is used to detect the end of the audio data. The two methods discussed above were built to use this header data. **PrintWAVHeader( )** prints out the header data to the screen monitor. **ValidWavData( )** will check on the validity of the wavfile to see if it is a **RIFF** Wave file, the most commonly used format. Other formats cannot be read with the code written here. Both methods are run from the **OpenWaveFile( )** method at the start of the Program Package code.

The SD_PLAY.cpp File

```
#include "sd_play.h"

File WaveFile;
// Memory allocated for frame data from WAV
byte SDplay::Samples[SAMPLES_BUFFER_SIZE];
// Number of bytes read from file so far
uint32_t SDplay::BytesReadSoFar = 0;
```

```
// Current pos of buffer to output next
uint16_t SDplay::BufferIdx = 0;

void SDplay::SDCardInit()
{
    pinMode(SD_CARD_CS, OUTPUT);
    digitalWrite(SD_CARD_CS, HIGH); // SD card chip select set high
    SPI.begin(SD_CARD_CLK, SD_CARD_MISO, SD_CARD_MOSI, SD_CARD_CS);

    if(!SD.begin(SD_CARD_CS))         //setup SD
    {
        Serial.println("Unable to talk to SD card!");
        while(true);                  // end program
    }
    else
        Serial.println("SD.begin(), Talking to SD card!");
}


void SDplay::OpenWaveFile()
{
    WavFile = SD.open("/wavfile.wav"); // Open wavfile.wav at root of SD

    if(WavFile==false)
      Serial.println("Could not open 'wavfile.wav'");

    else
    {
      // Read in the WAV header, which is first 44 bytes of the file.
      // We have to typecast to bytes for the "read" function
      WavFile.read((byte *) &WavHeader,44);

      // Print the header data to serial, optional!
      PrintWAVHeader(&WavHeader);

      // optional if you're sure the WAV file will be valid.
      if(ValidWavData(&WavHeader))
        Serial.println("Wav file is valid'");
    }
}


void SDplay::ReadFile(byte* Samples)
{
    // Always fills the the Samples[] buffer with SAMPLES_BUFFER_SIZE bytes

    //check for end of wavfile
    if(BytesReadSoFar + SAMPLES_BUFFER_SIZE > WavHeader.DataSize)
    {
      //what's left at end of wav file
      BytesToRead=WavHeader.DataSize-BytesReadSoFar;

      //partial fill of Samples[] buffer
      WavFile.read(Samples,BytesToRead);

      //pack remaining bytes with silent samples
      for(int i=BytesToRead; i<SAMPLES_BUFFER_SIZE; i++) Samples[i]=0;

      WavFile.seek(44);  //reset to start of WavFile
      BytesReadSoFar=0;
```

```
    }

    else
    {
      //fill Samples[] buffer with SAMPLES_BUFFER_SIZE bytes from WavFile
      WavFile.read(Samples,SAMPLES_BUFFER_SIZE);

      BytesReadSoFar+=SAMPLES_BUFFER_SIZE; //adjust index into WavFile
    }
}


void SDplay::FillI2SBuffer(byte* Samples)
{
  // Writes SAMPLES_BUFFER_SIZE bytes to DAC DMA buffers.
  // Repeat while (!done) until you know they've all been written,
  // then you can re-fill Samples[] using ReadFile()

  done = false;  //initial condition to start an i2s_write

  while(!done)
  {
    // Set address to next byte in Samples[] buffer to send out
    DataPtr=Samples+BufferIdx;

    // This is amount to send (total = less what we've already sent)
    BytesToSend=SAMPLES_BUFFER_SIZE-BufferIdx;

    // Send to DAC DMA, 1 RTOS tick to complete
    i2s_write(i2s_port_t I2S_NUM,DataPtr,BytesToSend,&BytesWritten,1);

    // increase Samples index by number of bytes actually written
    BufferIdx+=BytesWritten;

    if(BufferIdx>=SAMPLES_BUFFER_SIZE)
    {
      // finished sending out all SAMPLES_BUFFER_SIZE bytes in Samples[],
      // reset index and set done to indicate this
      BufferIdx=0;
      done=true;
    }
    else
      done=false; // DAC DMA was too full, Still more data to send
  }
}

bool SDplay::ValidWavData(WavHeader_Struct* Wav)
{
  if(memcmp(Wav->RIFFSectionID,"RIFF",4)!=0)
  {
    Serial.print("Invalid data - Not RIFF format");
    return false;
  }
  if(memcmp(Wav->RiffFormat,"WAVE",4)!=0)
  {
    Serial.print("Invalid data - Not Wave file");
    return false;
  }
  if(memcmp(Wav->FormatSectionID,"fmt",3)!=0)
  {
```

```
    Serial.print("Invalid data - No format section found");
    return false;
  }
  if(memcmp(Wav->DataSectionID,"data",4)!=0)
  {
    Serial.print("Invalid data - data section not found");
    return false;
  }
  if(Wav->FormatID!=1)
  {
    Serial.print("Invalid data - format Id must be 1");
    return false;
  }
  if(Wav->FormatSize!=16)
  {
    Serial.print("Invalid data - format section size must be 16.");
    return false;
  }
  if((Wav->NumChannels!=1)&(Wav->NumChannels!=2))
  {
    Serial.print("Invalid data - only mono or stereo permitted.");
    return false;
  }
  if(Wav->SampleRate>48000)
  {
    Serial.print("Invalid data - Sample rate cannot be greater than
48000");
    return false;
  }
  if((Wav->BitsPerSample!=8)& (Wav->BitsPerSample!=16))
  {
    Serial.print("Invalid data - Only 8 or 16 bits per sample permitted.");
    return false;
  }
  return true;
}

void SDplay::PrintData(const char* Data,uint8_t NumBytes)
{
    for(uint8_t i=0;i<NumBytes;i++)
      Serial.print(Data[i]);
      Serial.println();
}

void SDplay::PrintWAVHeader(WavHeader_Struct* Wav)
{
  if(memcmp(Wav->RIFFSectionID,"RIFF",4)!=0)
  {
    Serial.print("Not a RIFF format file - ");
    PrintData(Wav->RIFFSectionID,4);
    return;
  }
  if(memcmp(Wav->RiffFormat,"WAVE",4)!=0)
  {
    Serial.print("Not a WAVE file - ");
    PrintData(Wav->RiffFormat,4);
    return;
  }
  if(memcmp(Wav->FormatSectionID,"fmt",3)!=0)
  {
```

```
    Serial.print("fmt ID not present - ");
    PrintData(Wav->FormatSectionID,3);
    return;
  }
  if(memcmp(Wav->DataSectionID,"data",4)!=0)
  {
    Serial.print("data ID not present - ");
    PrintData(Wav->DataSectionID,4);
    return;
  }
  // All looks good, dump the data
  Serial.print("Total size :");Serial.println(Wav->Size);
  Serial.print("Format section size :");Serial.println(Wav->FormatSize);
  Serial.print("Wave format :");Serial.println(Wav->FormatID);
  Serial.print("Channels :");Serial.println(Wav->NumChannels);
  Serial.print("Sample Rate :");Serial.println(Wav->SampleRate);
  Serial.print("Byte Rate :");Serial.println(Wav->ByteRate);
  Serial.print("Block Align :");Serial.println(Wav->BlockAlign);
  Serial.print("Bits Per Sample :");Serial.println(Wav->BitsPerSample);
  Serial.print("Data Size :");Serial.println(Wav->DataSize);
}
```

Setup( ) in main.cpp

Two of the methods defined above, **SDCardInit( )** and **OpenWaveFile( )**, are executed at the start of the **setup( )** section in **main.cpp**. This is the program entry point for the entire Effects Software Package.

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

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

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

  //~~I2S & SD & SD WaveFile initialized. See set_settings.cpp for I2S~~
  //~~Make sure SAMPLE_RATE = that of the wavfile on SD~~

        I2S_init();
        mySDplay.SDCardInit();
        mySDplay.OpenWaveFile();

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

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

} //Setup End
```

Setup( ) is where all interfaces, including **I2S** and **I2C**, are started up and initialized.

SDCardInit( ) starts up the SPI interface used by the SD Card with

```
SPI.begin(SD_CARD_CLK, SD_CARD_MISO, SD_CARD_MOSI, SD_CARD_CS);
```

and then attempts to detect the presence of an SD Card with

```
SD.begin(SD_CARD_CS)
```

OpenWaveFile( ) opens the **wavfile.wav** file on the SD Card with

```
WavFile = SD.open("/wavfile.wav");
```

and then reads the header info on the wavefile with

```
WavFile.read((byte *) &WavHeader,44);
```

Loop( ) in main.cpp

After **setup( )** in **main.cpp** the main **loop( )** takes over. This is where all the audio sample manipulation happens and where the I2S interface is engaged to move the processed samples to the DMA buffers and then on to the DAC output.

```
void loop()
{
  //Test of LillyGo Taudio SD Card playback
  //leave the main loop dedicated only to the I2S audio task

  byte txbuf[SAMPLES_BUFFER_SIZE]; //transmit buffer
  float rxl, rxr, txl, txr;  //left/right samples, processed as floats

  myPedal->init();
  taskSetup();

  while(1)  //signal processing loop
  {
      //gather 1024 input samples into Samples buffer from 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

        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        //~~~~~~~~~~ SIGNAL PROCESSING  ~~~~~~~~~~~~~~~~
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

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

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

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

The actual sample processing happens within an inner **while(1)** loop. Before that, several variables used in the processing are declared -- a transmit buffer array and some receive/transmit left/right sample **float**s. External controllers are then set up by **init( )** from the file **set_module.cpp** with `myPedal->init( )`, and controller polling tasks are started up with `taskSetup( )` from the file **task.cpp**.

The sample processing **while(1)** loop has five steps that are repeated:

1. `mySDplay.ReadFile(mySDplay.Samples);`

   This function from the SDplay Class will collect 1024 (**SAMPLES_BUFFER_SIZE**) bytes from the SD Card and store them in the **Samples[ ]** array.

2. ```
   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
  ```

   A **for( )** loop will now begin pulling out audio samples one at a time for processing. In particular, it will pull out one 16-bit left channel audio sample and one 16-bit right channel audio sample from the **Samples[ ]** buffer. However, the sample data from the SD Card is not ready-made for processing. It is sequenced in a particular format: left channel low byte, left channel high byte, right channel low byte, right channel high byte.

   The above code converts the two left channel bytes into a 16-bit sample by
  ```

shifting the high byte by 8-bits and OR'ing it with the low byte. The result is then typecast to a **float**. The same is done for the right channel. After pulling these four bytes from **Samples[ ]**, the **for( )** loop index is incremented by 4 to prepare for getting the next stereo pair of samples, but first, this current pair must be processed.

3.
```
txl = myPedal->gain * myPedal->gainRange * rxl;
txr = myPedal->gain * myPedal->gainRange * rxr;
```

The processing shown here is simple volume control. The variable "**gain**" varies between 0 and 1, set by one "up volume" pushbutton and one "down volume" pushbutton. The variable "**gainRange**" is set at 2. These two variables are created in the file **set_module.cpp**. Such a simple effects example is mainly for demonstration. Many more interesting effects can certainly be applied to the SD playback signal and this is where it happens.

4.
```
txbuf[i]   = ((int16_t) txl) & 0xff ;
txbuf[i+1] = ((int16_t) txl) >> 8;
txbuf[i+2] = ((int16_t) txr) & 0xff ;
txbuf[i+3] = ((int16_t) txr) >> 8;
```

The two processed left/right float samples must be converted back into 4 bytes and loaded, in the proper sequence, into a transmit buffer defined to hold byte-sized elements. This is accomplished with some binary processing (shifting and AND'ing) on the 16-bit left and right samples after typecasting the floats back into 16-bit samples.

The **for( )** loop then repeats steps 1 through 4 for the next 4 bytes of sample data, until all 1024 (**SAMPLES_BUFFER_SIZE**) bytes have been processed and the **txbuf[ ]** is full.

5.
```
mySDplay.FillI2SBuffer(txbuf);
```

This is another **SDplay** Class function. It loads the 1024 bytes from the **txbuf** buffer into the DMA buffers that feed the DAC outputs, using a special **i2s_write( )** function.

Processing these 1024 byte "frames" of sample data continues within the **while( )** loop until the entire wavfile has been played.

ReadFile( )

```
void SDplay::ReadFile(byte* Samples)
//Always fills the the Samples[] buffer with SAMPLES_BUFFER_SIZE bytes
{
      //check for end of wavfile
    if(BytesReadSoFar + SAMPLES_BUFFER_SIZE > WavHeader.DataSize)
    {
            //what's left at end of wav file
            BytesToRead=WavHeader.DataSize-BytesReadSoFar;

            //partial fill of Samples[] buffer
             WavFile.read(Samples,BytesToRead);

            //pack remaining bytes with silent samples
            for(int i=BytesToRead; i<SAMPLES_BUFFER_SIZE; i++)
            Samples[i]=0;

            WavFile.seek(44);    //reset to start of WavFile
            BytesReadSoFar=0;
    }
    else
    {
      //fill Samples[] buffer with SAMPLES_BUFFER_SIZE bytes from WavFile
      WavFile.read(Samples,SAMPLES_BUFFER_SIZE);

      //adjust index into WavFile
      BytesReadSoFar+=SAMPLES_BUFFER_SIZE;
    }
}
```

ReadFile( ) is an **SDplay** Class method placed at the start of the signal processing **while(1)** loop. Its job is to collect a "frame" of samples to process. Basically it fills the **Samples[ ]** buffer with the next 1024 bytes from the wavfile on the CD Card using the **FS** read function `WavFile.read(Samples,BytesToRead);` while also updating the index variable `BytesReadSoFar` to keep track of the playback position in the wavfile.

This is complicated a bit by what happens at the end of the wavfile. More than likely there won't be a full 1024 bytes left at the end of the file. In that case we read what bytes are there and then fill in the remaining 1024 bytes with zeroes. This will result in moment of silence at the end of the wavfile that will last no more than about 10 msec, an imperceptible amount of time.

Notice also that at the end of the file we just rewind and repeat the playback. Other FS functions such as **seek( )** and **openNextFile( )** suggest possible alternatives to just repeating the playback.

FillI2SBuffer( )

```
void SDplay::FillI2SBuffer(byte* Samples)
{
  // Writes SAMPLES_BUFFER_SIZE bytes to DAC DMA buffers.
  // Repeat until you know they've all been written.
```

```
  done = false;  //initial condition to start an i2s_write

  while(!done)
  {
    // Set address to next byte in buffer to send out
    DataPtr=Samples+BufferIdx;

    // This is amount to send (total less what we've already sent)
    BytesToSend=SAMPLES_BUFFER_SIZE-BufferIdx;

    // Send to DAC DMA, 1 RTOS tick time to complete
    i2s_write(i2s_port_t I2S_NUM,DataPtr,BytesToSend,&BytesWritten,1);

    // increase by number of bytes actually written
    BufferIdx+=BytesWritten;

    if(BufferIdx>=SAMPLES_BUFFER_SIZE)
    {
      // Sent out all SAMPLES_BUFFER_SIZE bytes in Samples[].
      // Reset index and set done to indicate this.

      BufferIdx=0;
      done=true;
    }
    else
      done=false; // DAC DMA was full with more data to send
  }
}
```

FillI2SBuffer( ) is an **SDplay** Class method placed at the end of the signal processing **while(1)** loop. Its job is to load a frame of samples from the transmit buffer to the I2S DMA buffers.

In the case of processing signals from the **Codec Input,** only the **i2s_write( )** function is needed here because samples coming in from the ADCs are going out to the DACs at the same sampling rate. Here, however, the input samples are probably coming from the SD Card at a much higher rate than the sample rate driving the DACs, risking a pile up of samples at the DMA buffers. Luckily, the **i2s_write( )** function within **FillI2SBuffer( )** can detect when the DMA buffer is full, stop the transfer, and indicate with **BytesWritten** how many bytes were successfully written to the DMA buffer. The **FillI2SBuffer( )** was coded to deal with full DMA buffers. It will keep coming back to write more into the buffer until all 1024 bytes have been loaded.

The Set_Codec.cpp File

```
#include "set_codec.h"

// declaration of codec, an instance of Codec
                Codec codec;

//Some functions built to set Codec registers

 void codec_sets()       //to be executed in main.cpp
 {

  codec.addaCfg(1,0);     // Enable adc and dac (DAC 1/0, ADC 1,0)
  codec.inputCfg(0,0,0); // Input config, (MIC 1/0, LINE 1/0, AUX 1/0)
  codec.outputCfg(1,0);  // Output MIXER config (DAC 1/0, INPUT BYPASS 1/0)
  codec.sampleRate();     // SAMPLE_RATE khz = 48, 32, 24, 16, 12, 8
  codec.hpVolSet(40,40); // Headphone volume 0 TO 63, (LEFT, RIGHT)
  codec.i2sCfg(2,0);      // I2S format MSB, 16Bit
  codec.loopback(0);      // Input to Output when 1, no Bypass when 0

  };
```

This file's **set_codec.h** function selects several codec register "set" functions from **codec.cpp** to be executed from **setup( )** in **main.cpp**. Our application here uses audio input from the SD Card Reader, not the ADC inputs on the Codec, therefore we want to disable any ADC inputs using the above **addaCfg( )** and **inputCfg( )** codec set functions. The comments explain how this is done. Zero to disable, 1 to enable a parameter.

Misc Problems

After uploading the program the gain on the wavfile will probably be too high. You will hear the audio breaking up on peaks. Press the "gain" down pushbutton several times to see the NeoPixels turn from red to blue and the audio "crackling" go away. A more permanent solution is to set the "gainRange" lower.

With the SD Card audio running, program uploads can fail. The error is described as:

```
A fatal error occurred: Serial data stream stopped: Possible serial noise
or corruption.

Failed to communicate with the flash chip, read/write operations will fail.
Try checking the chip connections or removing any other hardware connected
to IOs.
```

I suspect that the problem is the LillyGo TAudio's use of GPIO 02 in the SD SPI interface. GPIO 02 is one of several ESP32 pins that does double duty with some type of startup job. The upload does succeed after several attempts.