LyraT & A1S Effects Software

John Talbert - January 2023

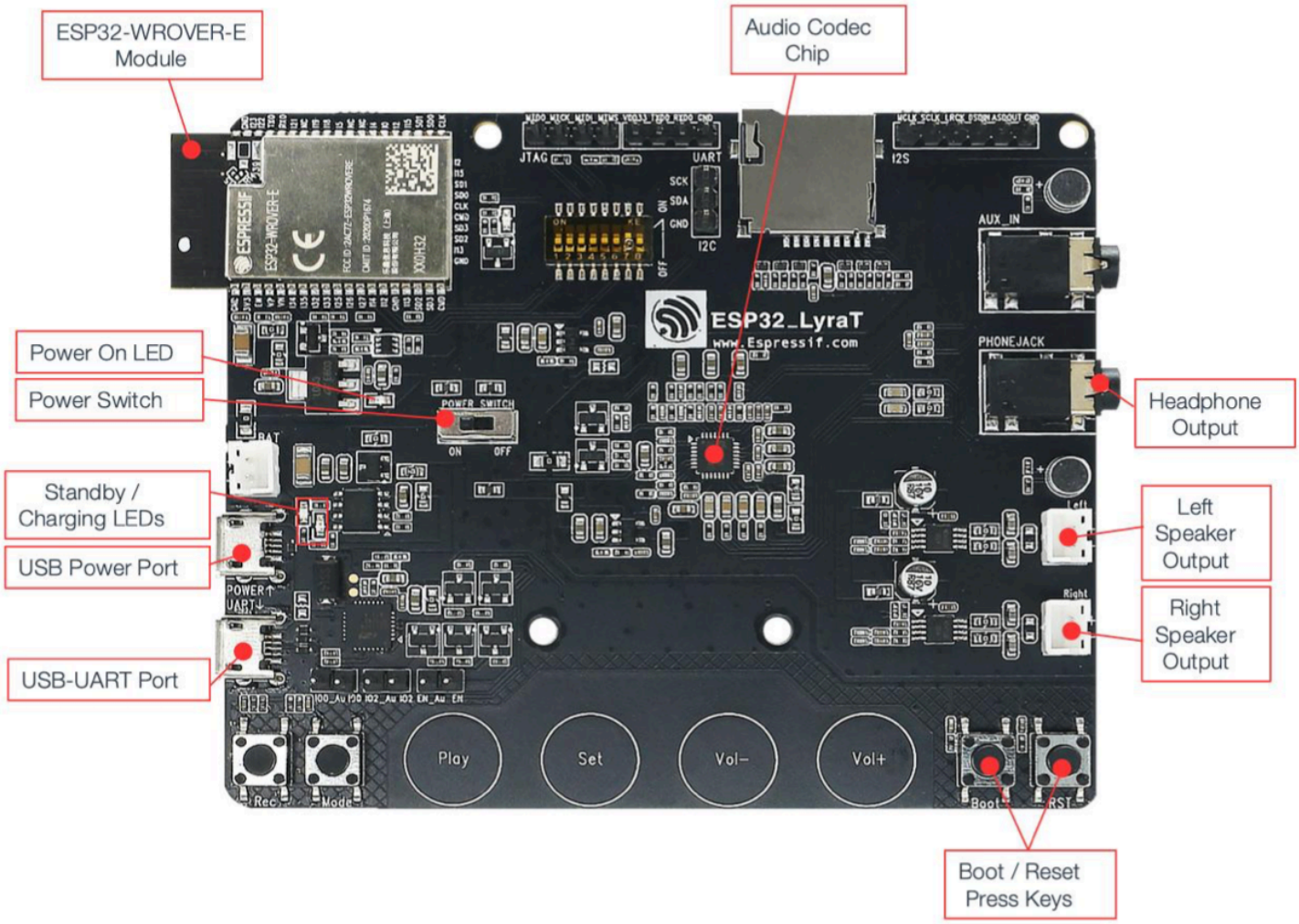


Table of Contents

Acknowledgements

Many thanks to Hasan Murod who created the original software package upon which this project is based. It was written for the Blackstomp Effect Pedal project (https://www.deeptronic.com/blackstomp/) which is a quick development platform for an ESP32 based audio effects module. The original software package can be found at https://github.com/hamuro80/blackstomp

Thanks also to Arif Darmawan for his ES8388Arduino software at https://github.com/vanbwodonk/es8388arduino. This is probably the absolute minimum software needed to get the ES8388 codec working on an ESP32. This coding project started from this simple working package. Elements from the Blackstomp software were incrementally added until I had a full, though scaled back, working version of Blackstomp.

ES8388 Codec Effects Software

What is offered here is a complete Effects Programming Software package for the LyraT and A1S audio development boards. It is basically the same software package found on my https://www.jtalbert.xyz/ESP32/ website with only a few changes specific to each board. Please read the tutorial PDF "**Codec Software**" for a complete description of the codec effects software package.

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:

```
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and examples
; https://docs.platformio.org/page/projectconf.html

[env:esp32dev]
platform = espressif32@5.2.0
board = esp32dev
framework = arduino

monitor_speed = 115200
```

Both the LyraT and A1S boards use an ES8388 Codec. The same codec was used in the original package, so there are no changes needed to the codec files -- **codec.h**, **codec.cpp**, **set_codec.h**, and **set_codec.cpp**. There are also no changes to the Digital Signal Processing tools -- **bsdsp.h**, **bsdsp.cpp**, and **dsptable.h**. The remaining files have minor changes which are explained in detail and in programming sequence in the next several sections.

The LyraT

The ESP32-LyraT V4.3 is an audio development board produced by Espressif built around ESP32. It is intended for audio applications, by providing hardware for audio processing and additional RAM on top of what is already onboard the ESP32 chip. The specific hardware includes:

ESP32-WROVER-E Processor Module
ES8388 Audio Codec Chip
Dual Microphones on board
Headphone output
2 x 3-watt Speaker output
Dual Auxiliary Input
MicroSD Card slot (1 line or 4 lines)
Six buttons (2 physical buttons and 4 touch buttons)
JTAG header
Integrated USB-UART Bridge Chip
Li-ion Battery-Charge Management

https://www.espressif.com/en/products/devkits/esp32-lyrat

LyraT Controllers

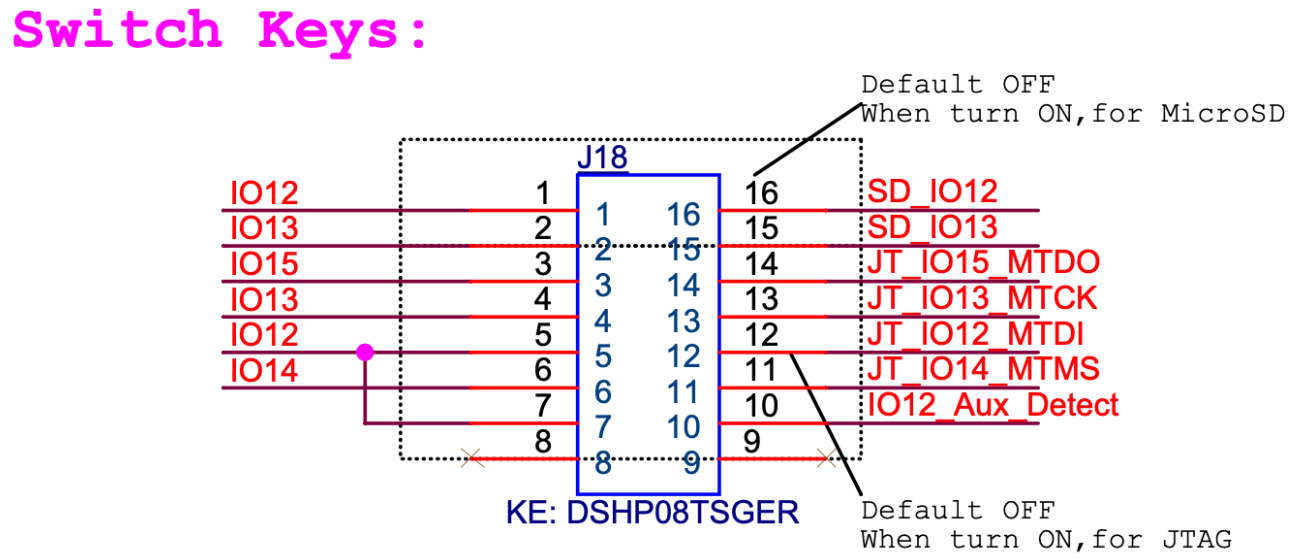
The LyraT board has six mounted switches available as user controllers. Two of them, **Rec** and **Mode**, are momentary pushbuttons. Four are capacitive touch switches -- **Set**, **Play**, **Volume+**, **Volume-**. There are also three sensors that detect inserts to the Headphone Jack, Aux Input Jack, and the SD micro Card Reader. One Green LED is available on board.

The file **set_settings.h** assigns ESP32 pins and Labels for each of the above controllers

and sensors as well as Codec pin assignments for the **I2S** and **I2C** interfaces.

```
//ESP32 LyraT PIN ASSIGNMENTS
  //~~~~~~~~~~~~~~~~~~~~~~~~~~

  // DIP Switch Settings for MicroSD 4-wire Mode:
  // 1-2 ON, 3-7 OFF
  // TOUCH_VOL_M not available unless 2 -> OFF
  // AUX_INSERT not available unless 7 -> ON

  #define POT2 39          //possible wired hack ?
  #define POT1 36          //possible wired hack ?

  #define LED_GRN     22
  #define PA_ENABLE   21

  #define KEY_REC     36    //pushbutton, or Pot Hack?
  #define KEY_MODE    39    //pushbutton, or Pot Hack?
  #define TOUCH_SET   32    //TOUCH9
  #define TOUCH_PLAY  33    //TOUCH8
  #define TOUCH_VOL_M 13    //TOUCH4, when DIP SWTCH 2&4 OFF
  #define TOUCH_VOL_P 27    //TOUCH7

  #define TOUCH_THRESHOLD 30

  #define PA_ENABLE   21    //PA Enable Output
  #define HDPHN_INSERT 19   //Headphone jack insert detect in
  #define SD_INSERT   34    //MicroSD insert detect in
  #define AUX_INSERT  12    //when DIP Swtch 5off, 7on
                            //keep DIP Switch 7off when powering on
                            //or unplug AUX input jack

//pins NOT available for OLED Display or other I2C device
#define SCK_PIN       22    //Green LED out
#define SDA_PIN       21    //PA Enable out

//ESP32-Codec Codec PIN SETUP
#define I2S_NUM               (0)
#define IS2_MCLK_PIN (0)
#define I2S_BCLK    (5)
#define I2S_LRC     (25)
#define I2S_DIN     (35)
#define I2S_DOUT    (26)

#define Codec_SDA       18    //SDA
#define Codec_SCK       23    //SCL
#define Codec_ADDR      0x10

  // SD Card Reader SPI
  #define SD_CARD_D0  02
  #define SD_CARD_D1  04
  #define SD_CARD_D2  12
  #define SD_CARD_D3  13
  #define SD_CARD_CMD 15
  #define SD_CARD_CLK 14
```

Note that several of the controller pins are shared with other board functions such as the **JTAG** output header and the micro **SD Card Reader**. Eight onboard **DIP** Switches are set by the user to configure the pin functions as explained in some of the above file comments and as shown here in a circuit diagram:



There are no extra ESP32 pins available on the LyraT board for other uses such as potentiometers or more LED indicators. The usefulness of some pin functions are questionable and might be co-opted, such as the Headphone and Aux Insert indicator pins, but changes to the micro circuitry of the board would be difficult. Later I will suggest an easier circuit hack which replaces the two pushbuttons with potentiometers.

Touch Switches

Many ESP32 pins can act as Capacitive Touch Switches. Here, pins IO32, IO33, IO27 and IO13 have been assigned to capacitive touch switches on the LyraT board. Touch pins do not act as digital outputs. They put out an analog type value that changes with the proximity of your finger to the touch pad. When touched the four LyraT touch switch values hover around 15 to 21, and when not touched the value is 45 to 67.

The Arduino function **digitalRead( )** can't be used for Touch Pads. Instead you use

touchRead(pin) from the ESP32 code library. The following code for **main.cpp** can be used to monitor the values from all the LyraT sensors including the four Touch Sensors.

```
#include <Arduino.h>
#include "set_settings.h"


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

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


// initialize Switches, some with pullup resistor
pinMode(KEY_REC, INPUT);
pinMode(KEY_MODE, INPUT);
pinMode(TOUCH_SET, INPUT);
pinMode(TOUCH_PLAY, INPUT);
pinMode(TOUCH_VOL_M, INPUT);
pinMode(TOUCH_VOL_P, INPUT);

pinMode(HDPHN_INSERT, INPUT);
pinMode(SD_INSERT, INPUT);
pinMode(AUX_INSERT, INPUT_PULLUP);

pinMode(LED_GRN, OUTPUT);

delay(2000);

} //Setup End


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

void loop()
{
  //Test of LyraT Switches


  digitalWrite(LED_GRN, HIGH);
  delay(300);
  digitalWrite(LED_GRN, LOW);
  delay(400);

  Serial.print(digitalRead(KEY_REC));
  //  Serial.print(analogRead(POT1));
  Serial.print(" ");
  Serial.print(digitalRead(KEY_MODE));
  //  Serial.print(analogRead(POT2));
  Serial.print("      ");

  //Print Raw Touch Switch values to determine threshold
  Serial.print(touchRead(TOUCH_PLAY));
```

```
  Serial.print(" ");
  Serial.print(touchRead(TOUCH_SET));
  Serial.print(" ");
  Serial.print(touchRead(TOUCH_VOL_M));
  Serial.print(" ");
  Serial.print(touchRead(TOUCH_VOL_P));

  Serial.print("     ");
  Serial.print(digitalRead(HDPHN_INSERT));
  Serial.print(" ");
  Serial.print(digitalRead(AUX_INSERT));
  Serial.print(" ");
  Serial.println(digitalRead(SD_INSERT));



} // End of Main Loop
```

Ideally, we would like the Touch Pads to act like digital pushbuttons. This can be accomplished by comparing the **touchRead( )** readings with a fixed threshold value. This threshold value must lie someplace between the "touched" and "un-touched" results from the above **touchRead( )** readings. Then the following simple code can transform the readings to a digital one or zero.

```
if(touchRead(pin) <= TOUCH_THRESHOLD)
      { touch_digital_value = 0; }
else  { touch_digital_value = 1; }
```

The threshold chosen was 30, stored in the constant, **TOUCH_THRESHOLD**, in the file **set_settings.h**.

Touch Code

The original codec effects package had no provisions for dealing with Touch Sensors so it has to be added. First, a boolean "**touch**" property is added to the **BUTTON struct**, along with a String **name** to facilitate button recognition in the **SytemMonitor**. Within the **controllerModule** class, in the file **controller_mod.h**, a six element **BUTTON** array was created to accommodate all six button sensors on the LyraT board.

```
struct BUTTON
{
  BUTTON_MODE mode;
  String name;
  bool inverted;
  bool touch;
  int min;
  int max;
```

```
  int value;
  int pin;
};

~~~~~~~~~~~~~~~~~~~~~~~
  CONTROL control[6];
  BUTTON button[6];
```

The **Constructor** method defined in the file **controller_mod.cpp** initializes all 6 button elements, setting the **touch** property to **false** and the mode to **BM_DISABLED**.

```
for(int i=0;i<6;i++)
   {
    button[i].inverted = false;
    button[i].touch = false;
    button[i].value = 0;
    button[i].min = 0;
    button[i].max = 1;
    button[i].mode = BM_DISABLED;
   }
```

The button properties can then be reconfigured as needed by the user in the **init( )** method within the file **set_module.cpp**.

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::init()  //effect module class initialization
 {
  name = "LYRAT";
  inputMode = IM_LR;   // IM_LR or IM_LMIC

  // Set up pin Modes for the switches and LEDs
  // Some need a pullup resistor.
  // Most have hardware pullup resistors.

  pinMode(KEY_REC, INPUT);
  pinMode(KEY_MODE, INPUT);
  pinMode(TOUCH_SET, INPUT);
  pinMode(TOUCH_PLAY, INPUT);
  pinMode(TOUCH_VOL_M, INPUT);
  pinMode(TOUCH_VOL_P, INPUT);

  pinMode(HDPHN_INSERT, INPUT);
  pinMode(SD_INSERT, INPUT);
  pinMode(AUX_INSERT, INPUT_PULLUP);

  pinMode(LED_GRN, OUTPUT);


  //setting up the buttons

  button[0].name = "Rec";
  button[0].mode = BM_MOMENTARY;
  button[0].touch = false;
  button[0].pin = KEY_REC;

  button[1].name = "Mode";
  button[1].mode = BM_MOMENTARY;
```

```
  button[1].touch = false;
  button[1].pin = KEY_MODE;

  button[2].name = "Play";
  button[2].mode = BM_TOGGLE;
  button[2].touch = true;
  button[2].pin = TOUCH_PLAY;

  button[3].name = "Set";
  button[3].mode = BM_TOGGLE;
  button[3].touch = true;
  button[3].pin = TOUCH_SET;

  button[4].name = "Vol-";
  button[4].mode = BM_TOGGLE;
  button[4].touch = true;
  button[4].pin = TOUCH_VOL_M;

  button[5].name = "Vol+";
  button[5].mode = BM_TOGGLE;
  button[5].touch = true;
  button[5].pin = TOUCH_VOL_P;

  // Only Button Control, No Pots
```

These properties for each of the six LyraT buttons are then passed to the **buttontask( )** within **task.cpp**. This task continuously polls the button pin values and manipulates those values to create a **MOMENTARY** or **TOGGLE** type switch action (specified in the **button[ ].mode**) from the polled values.

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//~~~~~~~~~~~~~~~BUTTON CONTROL TASK~~~~~~~~~~~~
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void buttontask(void* arg)
{
  //state variables
  int bstate[6];
  int bstatecounter[6];


  for(int i=0;i<6;i++)
  {
    bstate[i]=0;
    bstatecounter[i] = 0;
  }

  while(true)
  {
    vTaskDelay(1);

    for(int i=0;i<6;i++)  //service each of 6 possible switches
    {

      if(myPedal->button[i].mode == BM_TOGGLE)
      {
        int temp = 0;
        if(myPedal->button[i].touch){
```

```
      if(touchRead(myPedal->button[i].pin) >= TOUCH_THRESHOLD)
           { temp = 0; }
      else { temp = 1; }
  }
  else {
        temp = !digitalRead(myPedal->button[i].pin);
  }

  if(myPedal->button[i].inverted)
  temp = !temp;

  if(temp!= bstate[i])
  {
    bstatecounter[i]++;
    if(bstatecounter[i]>9) //debouncing
    {
      bstate[i] = temp;
      bstatecounter[i]=0;
      if(temp)
      {
        myPedal->onButtonPress(i);
        if(myPedal->button[i].value == 1)
          myPedal->button[i].value = 0;
        else
          myPedal->button[i].value = 1;

        myPedal->onButtonChange(i);
      }
      else
      {
        myPedal->onButtonRelease(i);
      }
    }
  }
  else bstatecounter[i]=0;
}
else if(myPedal->button[i].mode == BM_MOMENTARY)
{
  int temp = 0;
  if(myPedal->button[i].touch){
      if(touchRead(myPedal->button[i].pin) >= TOUCH_THRESHOLD)
           { temp = 0; }
      else { temp = 1; }
  }
  else {
        temp = !digitalRead(myPedal->button[i].pin);
  }
  if(myPedal->button[i].inverted)
  temp = !temp;

  if(temp != myPedal->button[i].value)
  {
    bstatecounter[i]++;
    if(bstatecounter[i]>9) //debouncing
    {
      bstatecounter[i]=0;
      if(temp==0)
      {
        myPedal->onButtonRelease(i);
        myPedal->button[i].value = 0;
```

```
                myPedal->onButtonChange(i);
              }
              else //temp=1
              {
               myPedal->onButtonPress(i);
                myPedal->button[i].value = 1;
                myPedal->onButtonChange(i);
              }
            }
          }
          else //temp = myPedal->button[i].value
          {
            bstatecounter[i]=0;
          }
        }

      }
    }
  } //End Button task
```

Notice in the above code that if the button is designated as a "**touch**" sensor, a **touchRead( )** is executed and the reading is compared with the threshold value to convert it to a digital high or low. If not, the regular **digitalRead( )** function is used instead. The final button values are then updated and the controller method **onButtonChange( )** is called if the button value changed. This task is continually repeated within an infinite while(1) loop.

Controller Event Handlers

The specific effect illustrated in this package is simple amplitude control of an audio input signal using the LyraT button controllers. The programmer will define an "event handler" in the file **set_module.cpp** for each of the 6 LyraT button sensors in the form of the controller class method **onButtonchange( )**.

```
void controller_module::onButtonChange(int buttonIndex)
{
  switch(buttonIndex)
  {
    case 3: //"Set" button state has changed
    {
      if(button[3].value) //if effect is activated
      {
        //codec.analogBypass(false);
        codec.DACmute(0);
        //digitalWrite(LED_GRN, HIGH);
      }
      else //if effect is bypassed
      {
        //codec.analogBypass(true);
        codec.DACmute(1);
        //digitalWrite(LED_GRN, LOW);
```

```
        }
        break;
      }

      case 2: // "Play" button[2] state has changed
      {
        if(button[2].value) // just test LED and Switch
        {digitalWrite(LED_GRN, HIGH);}
        else
        {digitalWrite(LED_GRN, LOW);}
        break;
      }

      case 4: // "Vol-" button[4] state has changed
      {
        gain = gain - 0.2;
        if(gain < 0) {gain=0;}
        break;
      }

      case 5: // "Vol+" button[5] state has changed
      {
        gain = gain + 0.2;
        if(gain > 1) {gain=1;}
        break;
      }

      case 0: // "Rec" button[0] state has changed
      {
        if(button[0].value)
        {gainRange = 2;} //boost volume
        else
        {gainRange = 1;}
        break;
      }
    }
  }
```

Here is an explanation of what the code above accomplishes for each button:

1. **Rec** This is a momentary action pushbutton. If pressed the audio signal is boosted by 2, using the **int** variable **gainRange**.

2. **Mode** This is a momentary action pushbutton. No action is defined, but it will show up in the system monitor along with all the others.

3. **Play** This is a toggle action touch pad. It will toggle the green LED off and on.

4. **Set** This is a toggle action touch pad. It toggle the codec audio mute.

5. **Volume -** This is a toggle action touch pad. It will decrease the audio signal volume by 1/5 each time pressed until zero is reached, using the **float** variable "**gain**".

6. **Volume +** This is a toggle action touch pad. It will increase the audio signal volume by 1/5 each time pressed until one is reached, using the **float** variable "**gain**".

LyraT gainDoubler Effect

Finally, the effect processing code is found in the l**oop( )** of the **main.cpp** file.

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

      setDebugVars(myPedal->gain, myPedal->gainRange, 0, 0);

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

//process samples one at a time from buffers
for (int i=0; i<(FRAMELENGTH); i+=2)
{
    rxl = (float) (rxbuf[i]) ;   //convert sample to float
    rxr = (float) (rxbuf[i+1]) ;

    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 txbuf by loading transmit buffer into DAC DMA memory
  i2s_write(I2S_NUM_0, txbuf, FRAMELENGTH*2, &readsize, 20);
```

The actual signal processing happens in the two middle code lines. The left and right channel audio samples are multiplied by two variables, **gain** and **gainRange**, which were manipulated by 2 touch pads and one pushbutton in the event handlers defined above.

Effects Programming Hints

When loading your programs onto the LyraT board both the **Boot** and **RST** buttons must be used in a specific sequence. As soon as you see "**Connecting .....**" hold down

the **Boot** button and then press and release the **RST** button. Then release **Boot**. The program will load but the Monitor won't engage until you Reset with the **RST** button once again.

~~~~~~

One very useful programming tool in this software package is the **SystemMonitor** Task. It can be started up from **setup( )** in main.cpp with this command:

```
runSystemMonitor( );
```

Along with **CPU** data, it will print out all current Controller values, updating around once per second. A sample of the Monitor output is shown below:

```
running ticks: 36453
CPU Usage: nan %
BUTTON-0 Rec: 0
BUTTON-1 Mode: 0
BUTTON-2 Play: 1
BUTTON-3 Set: 1
BUTTON-4 Vol-: 1
BUTTON-5 Vol+: 1
Debug String: None
Debug Variables: 0.4, 1, 0, 0
```

~~~~~~

Note the "Debug Variables" line. This is enabled with the command **setDebugVars( )**. It will display the current value of any four program variables you care to select. Here it was placed within the signal processing **while(1)** loop to provide the programmer with a current check of the **gain** and **gainRange** variables.

```
setDebugVars(myPedal->gain, myPedal->gainRange, 0, 0);
```

~~~~~~

One other change to the function **sysmon_task( )** in the file **task.cpp** was needed to take advantage of the added "**name**" property in the **BUTTON struct**.

```
if(myPedal->button[i].mode != 0)  // not CM_DISABLED
            Serial.printf("BUTTON-%d %s: %d\n",  i,
myPedal->button[i].name.c_str(),   myPedal->button[i].value);
```
~~~~~~

LyraT Hardware Hacks

Unfortunately, there are no ESP32 pins left available on the LyraT board for adding potentiometers, LEDs, or other Sensors. Some board features will need to be removed to free up pins for other uses.

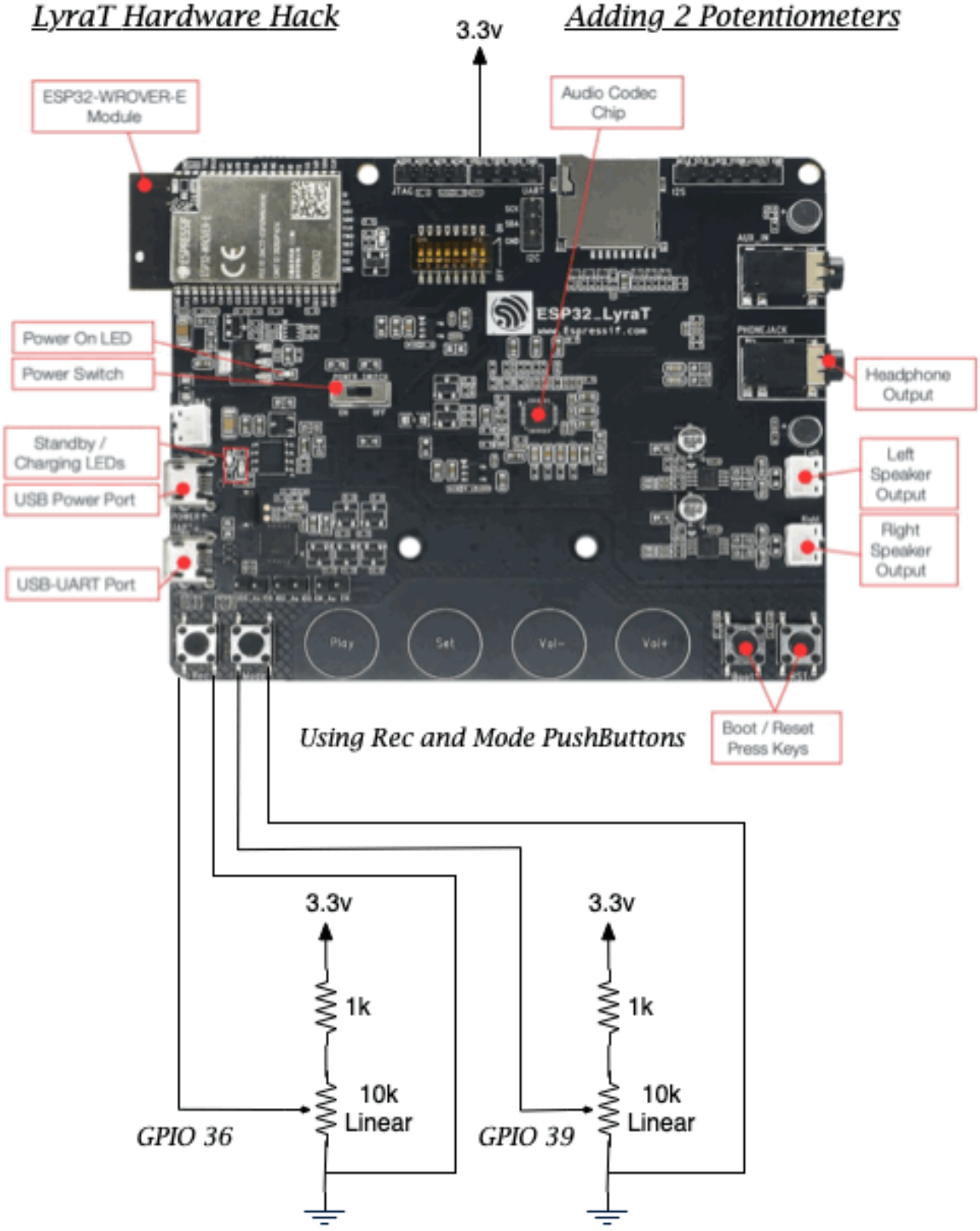
The **HDPHN_INSERT** and **AUX_INSERT** pin functions are of questionable usefulness but disconnecting their tiny traces on the circuit board would be difficult.

Four ESP32 pins on the **JTAG** header are readily accessible by setting the DIP switches. However, these pins are also used by the **SD Card Reader**. There is no circuitry inside the actual SD Slot, instead, it is inside the SD Card itself. That means that the four **JTAG** pins are free to be wired to other circuits as long as no SD Card is inserted in the SD Slot. To use the **SD Card Reader**, then, one would first have to disconnect any external circuits to those **JTAG** header pins, which may be as simple as pulling the **JTAG** header connection. It is useful here to note that GND and 3.3volts are readily available on the header next to the **JTAG**.

The solution chosen here was to take over the two pushbuttons **Rec** and **Mode** as shown in the figure below and leave the SD Card Reader always available.

LyraT Hardware Hack

Adding 2 Potentiometers



Rec and Mode can still be used as PushButtons if Pot Wipers are positioned midway.

These two wired potentiometers can now be used simultaneously with the **SD Card Reader** functions, unlike the **JTAG** pin solution discussed above. The two pushbuttons, **Rec** and **Mode**, have large metallic feet which provide a easy soldering site for connecting wires to our pots (be careful to limit the soldering time to 2 seconds or less). In addition, the pushbutton feature is still available if the pot wiper is positioned at least half way, leaving the user a choice between analog or digital controllers at **GPIO** pins **36** and **39**.

The **1k** series resistor is required to guard against a short circuit between 3.3volts and Ground when the pot wiper is turned all the way up and the user presses the pushbutton, which connects the pot wiper to Ground or zero volts. The supply voltage 3.3volts was taken from the far left pin of the 4 pin header next to the **JTAG** header.

Chorus Effect

Much more interesting effects can be created using the DSP tool files and it only involves writing code for the **while(1)** loop in **main.cpp** and setting up the controllers in the two **set_module** files. All the other files in the effect software package can be left as is including most of the changes described above.

Here is the 16-bit Stereo Chorus processing code to be placed in the **loop( )** of **main.ccp**

```
while(1)
{

setDebugVars(myPedal->depth, myPedal->freq,
             myPedal->stereo, myPedal->asynch);

  i2s_read(I2S_NUM_0, rxbuf, FRAMELENGTH*2, &readsize, 20);

  for (int i=0; i<(FRAMELENGTH); i+=2)
  {
    rxl = (float) (rxbuf[i]) ;   //convert sample to float
    rxr = (float) (rxbuf[i+1]) ;

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

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

```
    else  //synchronous
      dt2 = (1 + lfo1.getOutput(myPedal->phaseDiff))* myPedal->depth;

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

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

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

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

}
```

A general description of the stereoChorus Effect code is given here:

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

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

The effect controllers are set up in the **set_module.h** and **set_module.cpp** files. First, here is the **set_module.h**:

```
#ifndef MODULE_H_
#define MODULE_H_

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

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

 extern fractionalDelay delay1;
 extern fractionalDelay delay2;
 extern oscillator lfo1;
 extern oscillator lfo2;

//Create a child class derived from controllerModule
```

```
//controller_module sets all Pot, Switch, and LED pin, mode, and actions

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

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

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

#endif
```

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

Next is the **set_module.cpp** file where all the elements declared above are defined in detail, mostly inside the **init( )** method. Here we'll use the two pots wired as described in the circuit hack section above.

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

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

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

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

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

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


void controller_module::init()  //effect module class initialization
 {
  name = "Stereo Chorus";
  inputMode = IM_LR;   // IM_LR or IM_LMIC

  // Set up pin Modes for the switches and LEDs
  pinMode(LED_GRN, OUTPUT);
  pinMode(TOUCH_SET, INPUT);
  pinMode(TOUCH_PLAY, INPUT);
  pinMode(TOUCH_VOL_M, INPUT);
  pinMode(TOUCH_VOL_P, INPUT);

  //setting up the 2 pots
  control[0].name = "Rate";
  control[0].mode = CM_POT;
  control[0].levelCount = 128;
  control[0].pin = POT1;

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

  //setting up the 4 touch switches
  button[2].name = "F/P Diff";
  button[2].mode = BM_TOGGLE;
  button[2].touch = true;
  button[2].pin = TOUCH_PLAY;

  button[3].name = "Stereo";
  button[3].mode = BM_TOGGLE;
  button[3].touch = true;
  button[3].pin = TOUCH_SET;

  button[4].name = "Sync";
  button[4].mode = BM_TOGGLE;
  button[4].touch = true;
  button[4].pin = TOUCH_VOL_M;

  button[5].name = "Bypass";
  button[5].mode = BM_TOGGLE;
  button[5].touch = true;
  button[5].pin = TOUCH_VOL_P;

  //set initial values for the chorus variables
  freq=5;
  depth=0.5;
  beatFrequency=2.5;
  stereo = 1;
  asynch = 1;
  lfo1.setFrequency(freq);
  lfo2.setFrequency(freq+beatFrequency);
 }
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::onButtonChange(int buttonIndex)
{
  switch(buttonIndex)
```

```
  {
    case 5: // button[5] Bypass state has changed
    {
      if(button[5].value) //if effect is activated
      {
        codec.analogBypass(false);
        //codec.DACmute(0);
        digitalWrite(LED_GRN, HIGH);
      }
      else //if effect is bypassed
      {
        codec.analogBypass(true);
        //codec.DACmute(1);
        digitalWrite(LED_GRN, LOW);
      }
      break;
    }
    case 4: //the button[4] Sync state has changed
    {
      asynch = (bool)button[4].value;
      break;
    }
    case 3: //the button[3] Stereo state has changed
    {
      stereo = (bool)button[3].value;
      break;
    }
     case 2: //the button[2] FP/Diff state has changed
    {
      beatFrequency = 5 * (float)button[2].value;
      phaseDiff = (float)button[2].value * 127;
      lfo2.setFrequency(freq + beatFrequency);
      break;
    }
  }
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::onControlChange(int controlIndex)
{
  switch(controlIndex)
  {
    case 0: //rate
    {
      freq = 0.5 + 10 * (float)control[0].value/127.0;
      lfo1.setFrequency(freq);
      lfo2.setFrequency(freq + beatFrequency);
      break;
    }
    case 1:  //depth
    {
      depth = 1.49 * (float)control[1].value/127.0;
      break;
    }
 }
}
```

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

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

Next, the controller child **init( )** method is defined, to be executed later in the main loop of **main.cpp** with the line `myPedal->init( );`. Here **pinMode**s are set up for 4 switches and one LED. Then the control properties required for 4 switches and 2 pots are configured. Finally, all the new variables used to hold the pot and switch values are defined and given initial values. At this point we can also assign some of these variables to the inputs of some DSP methods. The **setFrequency( )** method of the **lfo1 oscillator** class object will get its frequency from the variable **freq**. The other low frequency **oscillator, lfo2**, will get a slightly higher frequency, **freq+beatFrequency**.

One pushbutton is set up in the controller method **onButtonChange( )** to either enable the Chorus Effect or bypass it and indicate which with an LED. The other switches change the Effect.

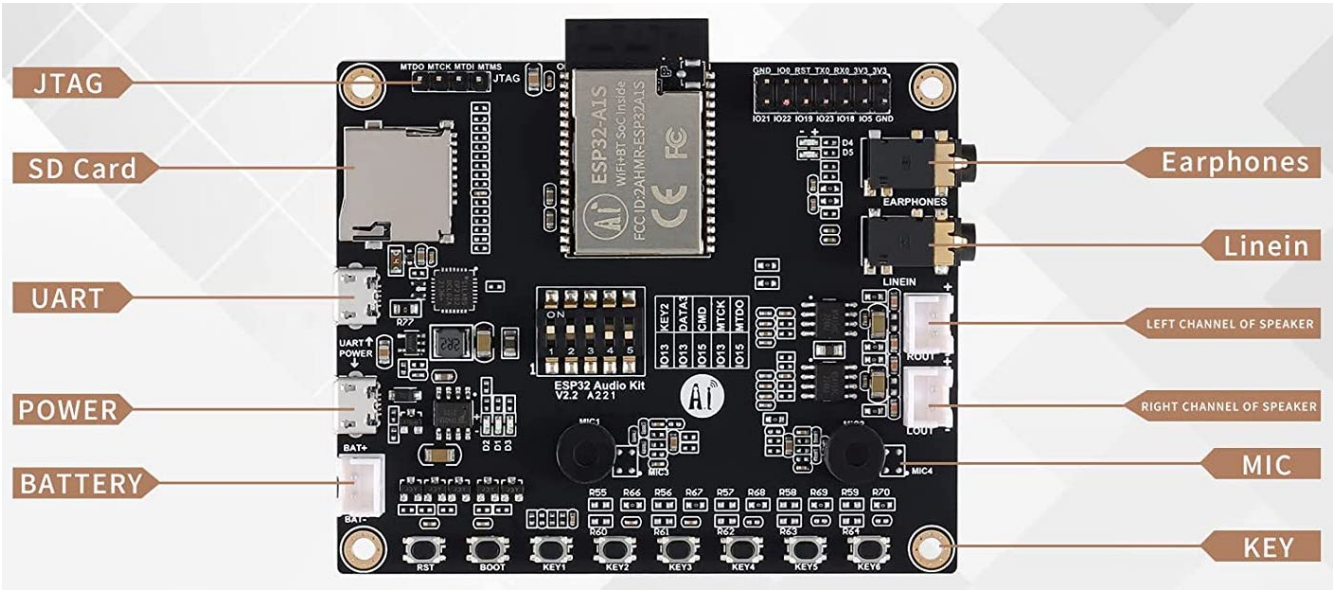
The action of 2 pots are configured in the controller method **onControlChange( )**. One pot value is assigned to the variable **depth** after a bit of mathematical adjustments. It will be used later in the signal processing loop. The other pot value is used right away to set the frequency of the low frequency oscillator objects using the **oscillator** class **setFrequency( )** method.

A1S ESP32 Audio Kit

The ESP Audio Kit board is an audio development board with the same features as the LyraT.

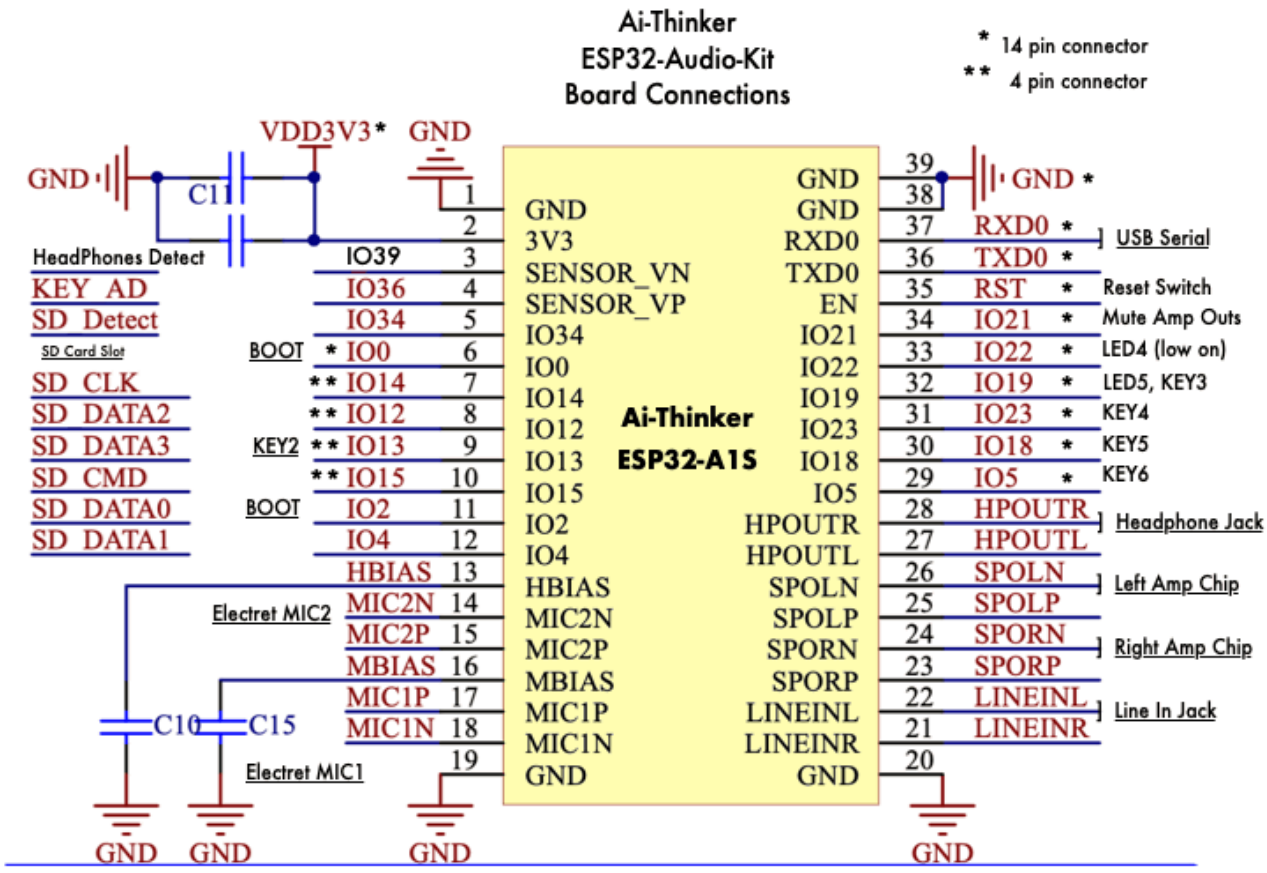
ESP32-WROVER-E Processor Module
ES8388 Audio Codec Chip
Dual Microphones on board
Headphone output
2 x 3-watt Speaker output
Stereo Auxiliary Input
MicroSD Card slot (1 line or 4 lines)
Six pushbuttons
JTAG header
Integrated USB-UART Bridge Chip
Li-ion Battery-Charge Management
Built-in 520 KB SRAM, 448 KB ROM, 16KB RTC SRAM, 4MB SPI Flash

The main difference lies in its use of the A1S Processor by AI Thinker. This is an ESP32 processor with an integrated codec. Earlier versions used the AC101 codec and later versions (v2.2 and above) used the ES8388 codec. https://docs.ai-thinker.com/_media/esp32-a1s_v2.3_specification.pdf



Pin/Labels

The following diagram shows the A1S pin assignments, which can be used to generate a list of pin Labels for the file **set_settings.h**



Include these pin labels in the set_settings.h file:

```
//A1S ESP32 Audio Kit PIN ASSIGNMENTS
 //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  #define POT2      39    //possible wired hack ?
  #define POT1      36    //possible wired hack ?

  #define LED1      22    //Low On
  #define LED2      19    //shared with Key3

  #define KEY1      36    //pushbutton, Pot Hack?
  #define KEY2      13    //pushbutton, shared with SD Data
  #define KEY3      19    //pushbutton, shared with LED2
  #define KEY4      23    //pushbutton
  #define KEY5      18    //pushbutton
```

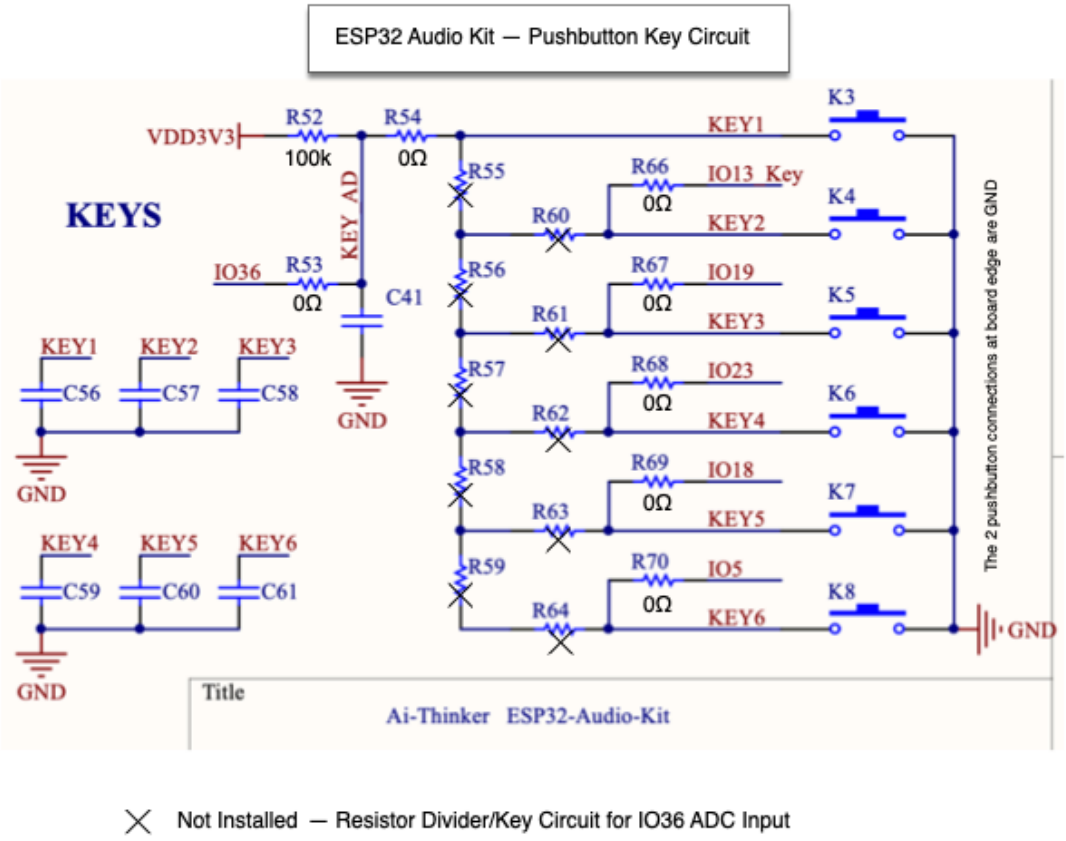
```
  #define KEY6     05    //pushbutton

  #define PA_ENABLE    21    //Mute Audio Output Amps
  #define HDPHN_INSERT 39    //Headphone jack insert detect, Pot Hack
  #define SD_INSERT    34     //MicroSD insert detect in

  //pins NOT available for OLED Display or other I2C device
  #define SCK_PIN       22    // LED1 out
  #define SDA_PIN       21    // PA Mute

  //ESP32-Codec Codec PIN SETUP
#define I2S_NUM               (0)
#define IS2_MCLK_PIN (0)
#define I2S_BCLK    (5)
#define I2S_LRC     (25)
#define I2S_DIN     (35)
#define I2S_DOUT    (26)

  //ESP32-Codec I2C PIN SETUP
  #define Codec_SDA       33   //SDA
  #define Codec_SCK       32   //SCL
  #define Codec_ADDR      0x10


  //DATA3 and CMD on-board switches in the ON position
  // SD Card Reader SPI
  #define SD_CARD_CS   13
  #define SD_CARD_MISO 2
  #define SD_CARD_MOSI 15
  #define SD_CARD_CLK  14

  //SPI.begin(PIN_AUDIO_KIT_SD_CARD_CLK, PIN_AUDIO_KIT_SD_CARD_MISO,
  //          PIN_AUDIO_KIT_SD_CARD_MOSI, PIN_AUDIO_KIT_SD_CARD_CS);

  //when you set up the SD library you need to indicate the
  //PIN_AUDIO_KIT_SD_CARD_CS as CS pin
```

A1S Hardware Hacks

Unlike the LyraT, the A1S board has no Touch Switches. Instead, it uses six regular pushbutton switches labeled **KEY1** through **KEY6**, along with **Reset** and **Boot**. These six Pushbuttons are connected to ESP32 pins 36, 13, 19, 23, 18, and 05. Five of them are available on a 14-pin board header, convenient for wiring up 5 external switches in parallel with the 5 on-board switches, **KEY2** through **KEY6**. Make sure that any external pushbuttons are "normally open" like the on-board pushbuttons.

KEY1 is wired to pin 36. This GPIO pin is the only one among the 6 pushbuttons that has an ADC Input capability along with its Digital Input function. A look at the board's pushbutton circuit below reveals an interesting resistor divider network made up of resistors R55 through R59. These resistors along with R60 through R64 are not installed. If installed, GPIO36, set up as an ADC Input, would receive a different

voltage depending on which pushbutton is pressed. All 6 on-board pushbuttons would then affect the one ADC input. Short-circuit resistors R66 through R70 could then be removed cutting the connections between KEY2 - KEY6 with their GPIO digital input pins. These 5 GPIO pins(13, 19, 23, 18, and 05) could then be used for other purposes such as external switches using the convenient 14-pin header connections.

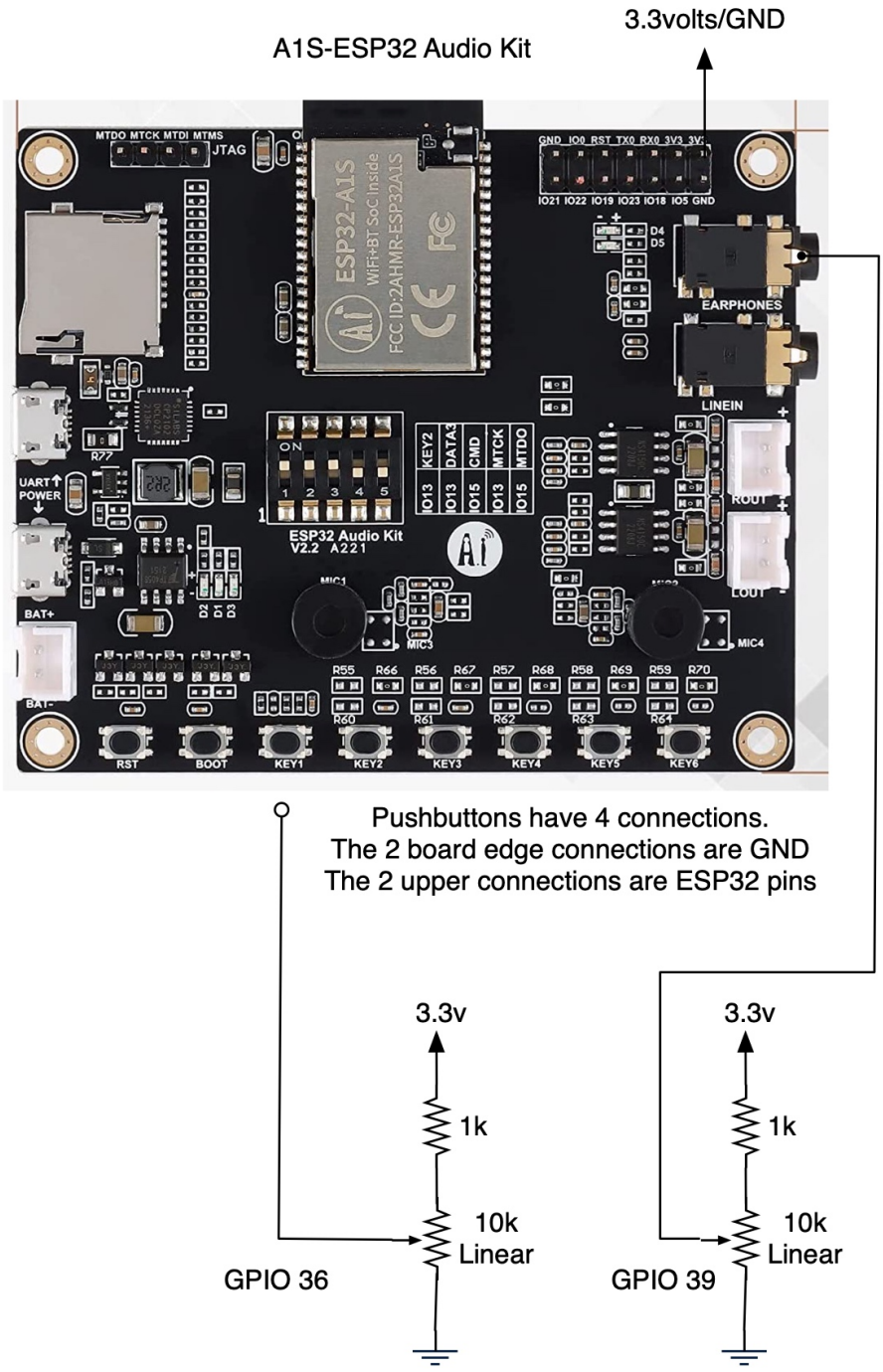


Not Installed — Resistor Divider/Key Circuit for IO36 ADC Input

Unfortunately, like the LyraT, there are no GPIO pins free on the Audio Kit board, especially with ADC capabilities for use with external potentiometers. There is a 4-pin **JTAG** header, but those pins are also used by the **SD Card Reader** as set by 5-pin **DIP** switches. When **DIP** switches 1, 4, 5 are set to "**ON**" the **JTAG** header carries IO15, IO13, IO12, and IO14. When set to "**OFF**", the **SD Card Reader** is connected. Note that pin IO13 of KEY2 is also shared with the **SD Card Reader** and **JTAG**.

The circuit hack below illustrates one way to connect two external potentiometers to the ESP32 Audio Kit board while keeping the **SD Card Reader** available.

GPIO36, from onboard **KEY1**, is co-opted for one of the external pots. Ground and 3.3volts are available from the right end of the 14-pin header. The two **KEY1**



pushbutton leads closest to the board edge are connected to **GND**. The other two are your connections to GPIO36. These pushbutton leads are tiny and difficult to solder. Use thin, single strand, connection wire and only a half second or so of solder application time. Hot glue on the wire insulation can keep the connection strong. Since **KEY1** is a "normally open" switch, it doesn't have to be removed. However, a 1k

series resistor is used to protect against 3.3v being shorted to ground when the pushbutton is pressed while the pot is dialed up.

The second pot connection is a unique hack. The ESP32-Audio-Kit uses pin IO39 to detect when headphones are connected to the mounted headphones jack. IO39 is connected to the sleeve tab of the headphones jack. It is also connected to a 100k pull-up resistor to 3.3v making it normally high at 3.3v. When a plug is inserted into the jack the sleeve tab makes a connection with the plug sleeve ground and IO39 will go low to zero volts.

If IO39 is to be used for a pot controller its **Headphone Detect** function must be disabled. This is easily accomplished by simply bending up the Sleeve Tab on the top of the board's headphone jack so that it will no longer make contact with the sleeve of an inserted headphone plug. The Sleeve Tab is then a convenient place to solder a wire going to the pot wiper.

Note that the LyraT board also has a **Headphone Detect** but it is not wired up in exactly the same way to the Headphones Jack.

Programming the Audio Kit

Effects programs for the A1S Audio Kit (version 2.2 or later with the ES8388 Codec) will be almost the same as shown for the LyraT. The only changes needed are the new pin/label assignments to be placed in the file **set_settings.h**, as shown above, and incorporating those new labels into **set_module.cpp**. Be sure to set the **button[ ]** properties "**touch**" to "**false**" since this board does not use touch switches.

Programming Summary

Most of the effects programming will involve just three files in the package.

1. The **set_settings.h** file holds all the program settings in the form of constants such as sample rate, bits per sample, number of channels, DMA memory sizes and, of course, the ESP32 pin assignments for the physical controllers, LEDs, I2S interface and I2C interface. This file also sets up the I2S_init( ) function.

2. The **set_module.h** and **.cpp** files hold all the setups for physical

controllers such as the **button[ ]** and **control[ ]** parameters, the **init( )** method, and the event handler methods. These methods are also used to control any LEDs and set up any **DSP** tools and variables. The **set_module** files will be a major focus point in effects programming.

3. Finally, the **while(1)** infinite loop within **main.cpp** will hold the actual effects processing routines down at the audio sample level.

The remaining files in the package will rarely need changes.

1. The **codec.h** and **.cpp** files along with **set_codec.h** and **.cpp** need to be changed only with a different codec. In both the LyraT and A1S Audio Kit boards the codec used was an ES8388. Both use the exact same codec files. With a different codec these files will need to be completely changed but that change will likely not affect the content of the other files in the package.

2. The **controller_mod.h** and **.cpp** files will never need changes. Any needed changes to the Parent Class defined here are designed to happen only in the "child" class **controller_module** constructed in the **set_module** files.

3. The **task.h** and **.cpp** files define task functions for polling the buttons and controllers along with some system monitor tasks. In the above LyraT example some simple changes were made to the button polling task to accommodate Touch sensors and an added String button[ ] name for display in the system monitor. Other unique sensors may require simple program changes to these files, a rotary encoder for example. In general, though, these files will not need any changes. They will satisfy most any effects program you design.

4. Finally, the **DSP** (Digital Signal Processing) tool files -- **bsdsp.h**, **bsdsp.cpp**, and **dsptable.h**, can always be expanded with new class tools, but currently they hold quite a wide variety of tools ripe for exploration. One useful tool not yet available might be an **SD Card Reader** class.

This ESP32 Codec software package was designed to simplify what can be an intimidating programming task by encapsulating each of many program tasks into separate program files, each with its own clearly defined job.