PUCA DSP Effects Software

John Talbert - January 2023

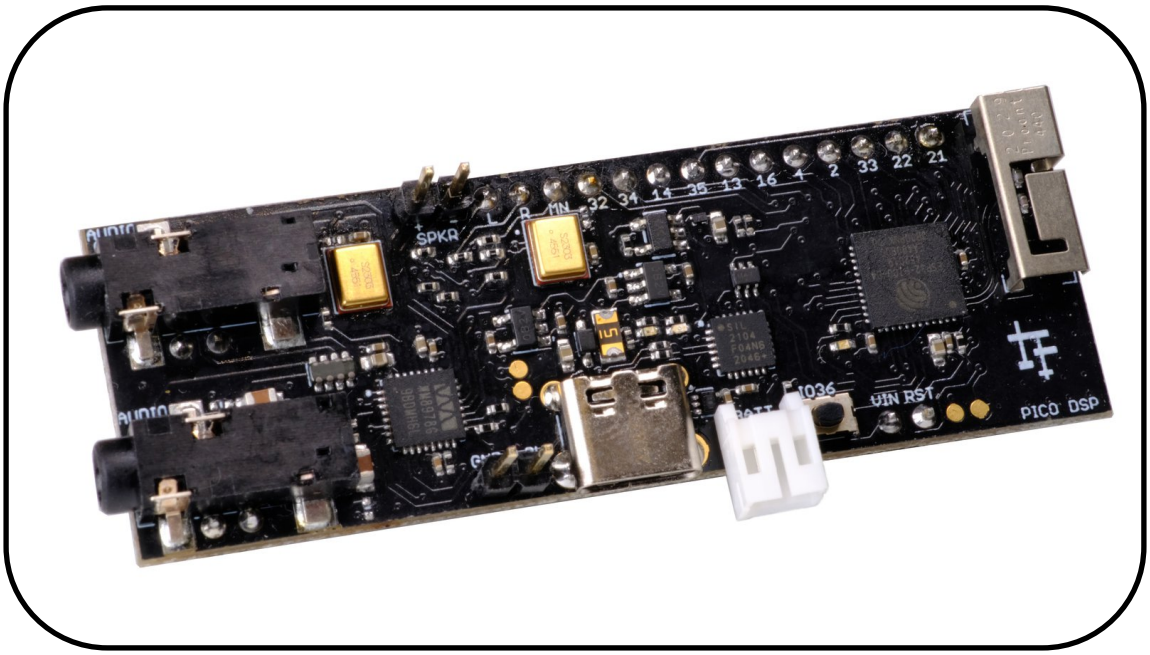


Table of Contents

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 to Andy Wilson (andy-wi) and David Swarbrick for their PÚCA DSP development work on the GitHub https://github.com/ohmic-net/puca_dsp. Thanks to the NuovotonDuino project: https://github.com/DFRobot/NuvotonDuino on which the I2C Configuration for a Wolfson Codec Audio Codec was based.

Codec Effects Software

What is offered here is a complete Effects Programming Software package for the PÚCA 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 the board. Please read the tutorial PDF "**Codec Software**" for a complete, more detailed description of the codec effects software package and the two Effects examples 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:tinypico]
platform = espressif32@5.2.0
board = tinypico
framework = arduino

; set cpu frequency - 80, 160 or 240MHz
board_build.f_cpu = 240000000L

; Serial Monitor options
monitor_speed = 115200

;upload_port = /dev/tty*

board_build.partitions = default.csv

build_flags =
    -DBOARD_HAS_PSRAM
```

```
-mfix-esp32-psram-cache-issue
-DCORE_DEBUG_LEVEL=5
;-Wl,-Map,output.map
```

This board uses the WM8978 Codec. My projects up to now have been for the ES8388 codec so the codec files -- **codec.h**, **codec.cpp**, **set_codec.h**, and **set_codec.cpp** will need to be changed to support this new codec. The codec driver was readily available from the PÚCA GitHub.

The PÚCA DSP

PÚCA DSP is an open-source, Arduino-compatible ESP32 development board for audio and digital signal processing (DSP) applications, available from https://www.crowdsupply.com/ohmic/puca-dsp. It offers an expansive audio-processing feature set on a small-format, breadboard-friendly device that provides audio inputs, audio outputs, a low-noise microphone array, an integrated test-speaker option, additional memory, battery-charge management, and ESD protection all on one tiny PCB.

Here is a list of its features as provided on the website:

Processor & Memory

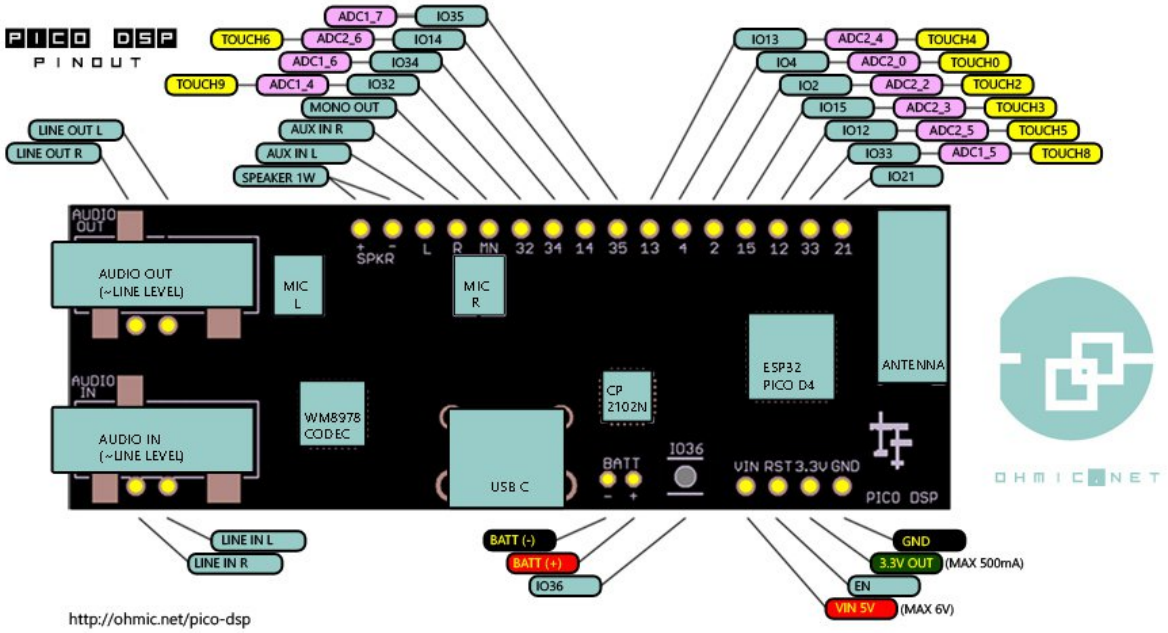
- Espressif ESP32 PICO D4 processor
- 32-bit dual-core 80 MHz / 160 MHz / 240 MHz
- Choice of either 4 MB SPI Flash
- 8 MB additional PSRAM or 16 MB External SPI Flash
- 2.4 GHz Wi-Fi 802.11 b/g/n
- Bluetooth BLE 4.2
- 3D antenna

Audio

- Wolfson WM8978 Stereo Audio Codec
- Audio Line In on a stereo 3.5 mm connector
- Audio Headphone / Line Out on stereo 3.5 mm connector
- Stereo Aux Line In, Audio Mono Out routed to GPIO Header
- 2 x Knowles SPM0687LR5H-1 MEMS microphones
- ESD protection on all audio inputs and outputs
- 8, 11.025, 12, 16, 22.05, 24, 32, 44.1, 48 kHz sample rates
- 1 W Speaker Driver routed to GPIO Header

PÚCA Controller Hardware

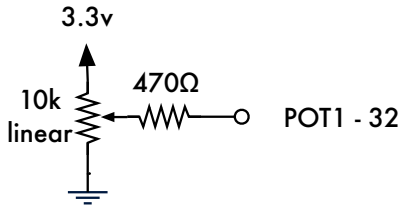
The circuit board has a 16 pin header with 11 ESP32 pins available for controllers. There is also an on-board pushbutton connected to GPIO36. A four pin header carries power lines and the Reset pin.



Controller Circuits

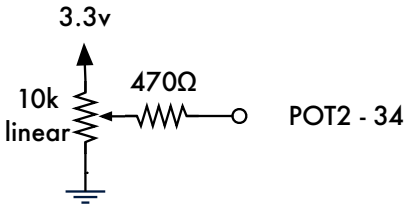
In this project 4 of the pins are used for potentiometers, 4 for pushbutton switches, and 2 for LEDs. The circuit wiring for these controllers is shown in the figure below. The series 470Ω resistors in the pot and pushbutton circuits are included as a safety feature preventing possible short circuits if the pins are mistakingly defined as outputs. For the 2 LEDs, the 470Ω resistor sets the LED brightness. An example circuit for a cadmium cell light sensor was included in the figure, but not implemented.

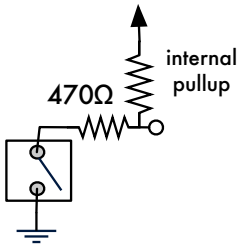
Note that the potentiometers can be any value between about 5k and 100k but must be linear taper.



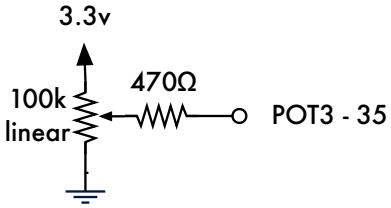
ESP32 / PUCA
Sensor – Controller Circuits

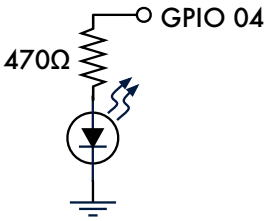
John Talbert 2023

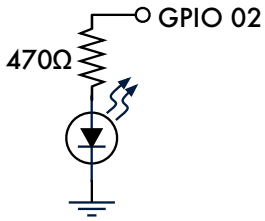


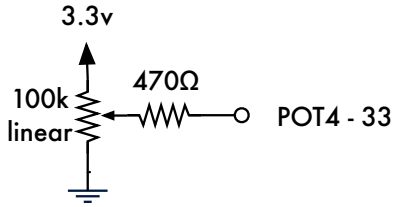


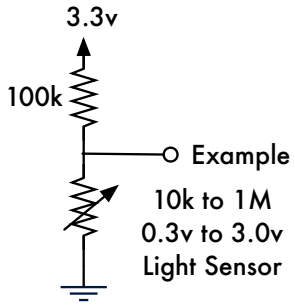
KEY1 GPIO 14
KEY2 GPIO 13
KEY3 GPIO 15
KEY4 GPIO 21







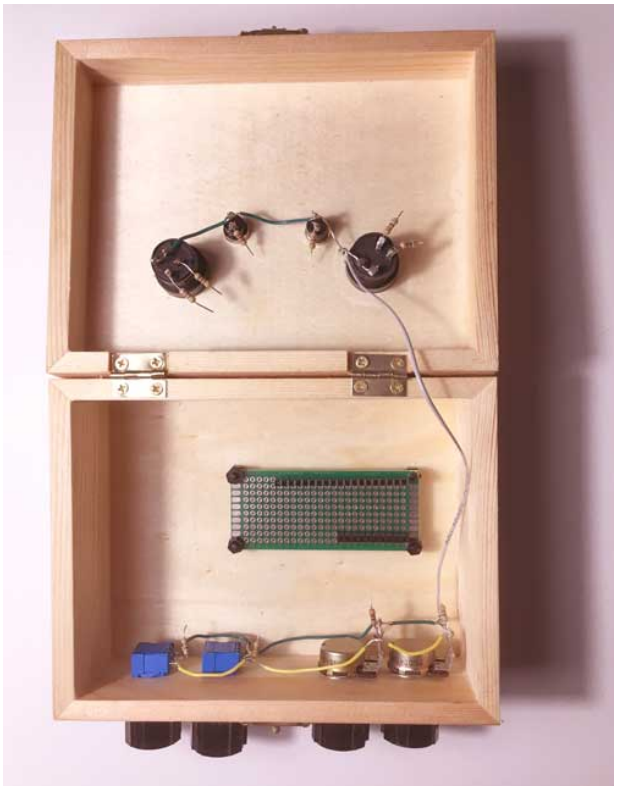


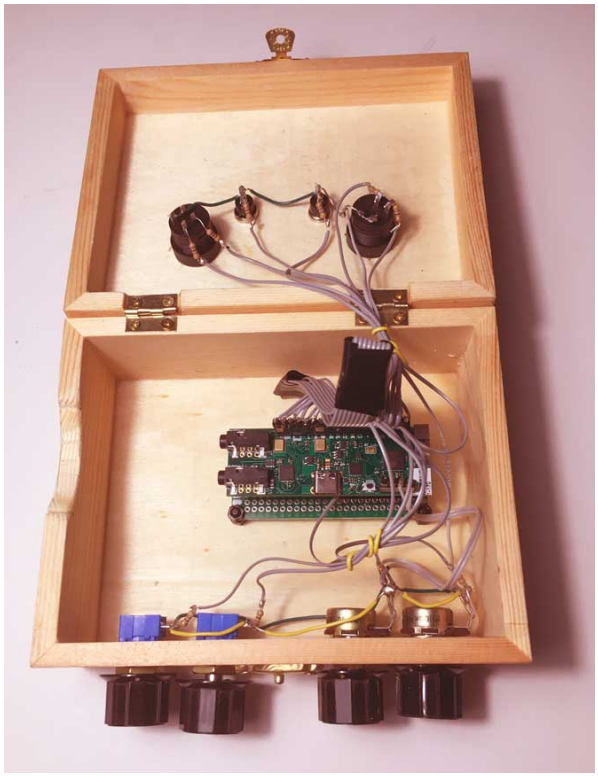


470Ω's for pots and switches are for safety,
in case pin is defined as an output

Board Mounting

The PÚCA circuit board has no mounting holes. Extended male headers were used to sit the PÚCA board into female headers on a secondary mounted proto board.





Reset and Boot Problems

Several pins on the ESP32 perform extra duties during boot time. Of all the pins available on the PÚCA header pin 12 may cause reset/boot to fail if pulled high. If GPIO pin 12 is used for a potentiometer the solution is to turn down the pot before any reset.

Here are some other pins that may cause boot problems depending on how you are using them.

The ESP32 chip has the following strapping pins:

- GPIO 0 (must be LOW to enter boot mode)
- GPIO 2 (must be floating or LOW during boot)
- GPIO 4
- GPIO 5 (must be HIGH during boot)
- GPIO 12 (must be LOW during boot)
- GPIO 15 (must be HIGH during boot)

Random Nerd Tutorials provides an excellent review of ESP32 pin functions at https:/

/randomnerdtutorials.com/esp32-pinout-reference-gpios/. Here is what is suggested in this review.

> These pins above are used to put the ESP32 into bootloader or flashing mode. On most development boards with built-in USB/Serial, you don't need to worry about the state of these pins. The board puts the pins in the right state for flashing or boot mode. More information on the ESP32 Boot Mode Selection can be found here.
>
> However, if you have peripherals connected to those pins, you may have trouble trying to upload new code, flashing the ESP32 with new firmware, or resetting the board. If you have some peripherals connected to the strapping pins and you are getting trouble uploading code or flashing the ESP32, it may be because those peripherals are preventing the ESP32 from entering the right mode. Read the Boot Mode Selection documentation to guide you in the right direction. After resetting, flashing, or booting, those pins work as expected.

The PÚCA board exhibited another boot problem. Occasionally it would timeout while trying to upload new programs, complaining that it was unable to get into boot/flash mode. The usual solution to this problem is to utilize the "boot" pushbutton (connected to GPIO 0), however, the PÚCA board has no "boot" button. The problem seemed to go away for a while if the the board power was disconnected for a couple minutes.

My final solution was to connect a 10µF capacitor between the RST (reset) line and GND (ground) on the 4 pin board header. The negative side of the cap is connected to GND. This solution is suggested in the website https://randomnerdtutorials.com/solved-failed-to-connect-to-esp32-timed-out-waiting-for-packet-header/

PÚCA Software Package

This ESP32 Codec software package is 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.

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

1. The **set_settings.h** and **.cpp** files 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. These files also set 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 controller **init( )**, 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. The PÚCA board uses a Wolfson WM8978 codec. Changing to a different codec driver in these files will not likely affect the content of the other files in the package.

2. The **controller_mod.h** and **.cpp** files will hardly ever need changes. Any needed changes are applied 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. The LyraT PDF does show some simple changes made to the button polling task to accommodate Touch sensors. In general, though, these files will not need any changes. They will satisfy most any controller used in your 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.

The Codec Files

The **codec.h** and **.cpp** files are used to configure the Wolfson WM8978 codec. All codecs have the same basic components:

- audio inputs to Analog to Digital Converters (ADCs),
- audio outputs from Digital to Analog Converters (DACs),
- a large bank of registers to set the codec operation parameters,

- an I2C serial interface to access those registers,
- an I2S serial interface to move data in and out of the ADCs and DACs,
- a master clock input to time the I2S data movement.

So codecs use two interface protocols. A Two-wire I2C interface is used to configure the chip, and the I2S is used to move the audio data. In the WM8978 codec the I2C interface is used to load and read 58 user programmable 9-bit registers that set up I/O connections, sampling rate, sample format, sample size, volume, filters, effects, etc.

codec.h and **.cpp** files were readily available for the WM8978 from the PÚCA GitHub development software at https://github.com/hamuro80/blackstomp. They included all the components common to codec drivers:

1. Creation and initialization of an I2C interface. The initialization requires two GPIO pin assignments and a speed value. It uses an `#include "driver/i2c.h"` library. The **initI2C( )** method is placed at the start of the codec **init( )** method. This is a little different from I2C initialization for the LyraT and E1S boards which used the **Wire.h** library and required some extra I2S lines in the **set_codec.h** file and the **setup( )** section of **main.cpp**.

2. The WM8978 has 58 9-bit registers used to configure the codec via the I2C interface. The next required methods are **writeReg( )** and **readReg( )** to access these registers.

3. Next is an **init( )** method used to initialize many of the 58 codec registers. This will be called in the **setup( )** section of **main.cpp**.

4. Finally, several short methods are built for convenient access to specific codec registers such as input and output connections, output volume, mute, bypass, sample rate, etc. Several of these are included within the **codec_sets( )** function defined in **set_codec.cpp** and called in the **setup( )** section of **main.cpp**.

To demonstrate how easy it is to work with the codec.cpp "set" methods, here is simple revision to the sampleRate( ) register set:

```
//BEFORE
// Set Sample Rate
// srate: 0~5 , 48kHz, 32kHz, 24kHz, 16kHz, 12kHz, 8kHz
void Codec::sampleRate(uint8_t srate)
{
    uint16_t regval = 0;            // 48kHz, default
```

```
    if(srate==1) regval = 0x2;
    if(srate==2) regval = 0x4;
    if(srate==3) regval = 0x6;
    if(srate==4) regval = 0x8;
    if(srate==5) regval = 0xA;
    writeReg(7,regval);             // R7, Additional Ctrl
}

//AFTER
// Set Sample Rate with SAMPLE_RATE set in set_settings.h
// 48kHz, 32kHz, 24kHz, 16kHz, 12kHz, 8kHz
void Codec::sampleRate(void)
{
    uint16_t regval = 0;            // 48kHz, default
    if(SAMPLE_RATE==32000) regval = 0x2;
    if(SAMPLE_RATE==24000) regval = 0x4;
    if(SAMPLE_RATE==16000) regval = 0x6;
    if(SAMPLE_RATE==12000) regval = 0x8;
    if(SAMPLE_RATE==8000)  regval = 0xA;
    writeReg(7,regval);             // R7, Additional Ctrl
}
```

With this simple change the user only needs to set the sample rate at the **#define SAMPLE_RATE** in the file **set_settings.h**. They no longer have to make sure that the codec **sampleRate( )** method agrees with the **SAMPLE_RATE** setting.

The complete documentation for the Wolfson WM8978 Codec and all its 58 registers can be downloaded from https://www.mouser.com/datasheet/2/76/WM8978_v4.5-1141768.pdf

The Controller_Mod Files

The **controller_mod.h** and **.cpp** file pair builds a base class, called **controllerModule**, that will facilitate the use of ESP32 connected potentiometers, switches, and other sensors to control the parameters of an audio effects program. Two arrays are created as class attribute members, one for up to 6 potentiometers or other analog sensors called **control[ ]**, and another one for up to 6 switches or other digital sensors called **button[ ]**. Each array element in turn has several properties such as name, GPIO pin, mode of operation, and value.

The Task Files

There is a task function for the **button[ ]** array elements and one for the **control[ ]** array elements. These tasks will continuously poll all the physical controllers attached to the ESP32. Using the array element **pin** parameter, the button task will perform a **digitalRead(pin)** and the control task will perform an **analogRead(pin)**. The data is then manipulated according to the **mode** parameter and the result stored in the **value**

parameter. This is done for each enabled button and control, and then repeated in an infinite loop.

The task files also includes a system monitor task function which prints out the above **button** and **control** values along with some system information about once per second.

The DSP Files

Effects programming can take advantage of the three **Blackstomp** Digital Signal Processing files **-- bsdsp.h**, **bsdsp.cpp**, and **dsptable.h**, which contain the following effect classes and tables:

dsptable.h contains a 256 element sine-wave table. **sine_table[ ]** covers a full wavelength in floating point fractional values ranging from +1 to -1. Also included for filters is a **hann_table[ ]**, 256 floating point fractional values ranging from 0 to +1.

The following DSP classes are defined in the **bsdsp** files:

biquadFilter -- A direct-form-2 biquad iir filter.

oscillator -- Creates an oscillator from a 256 element table of one waveform cycle. It can use any built waveform table including the **sine_table[ ]** from **dsptable.h**

fractionalDelay -- Implements a delayed output by building a circular sample buffer sized for a given **maxDelayInMs**. You can then request a sample read at any delay value up to the **maxDelay**.

waveShaper -- Applies a transfer function from **transferFunctionTable[ ]** to the input stream. This is a default exponential function but any 256 element array could be used.

rcHighPass -- A simple RC High Pass Filter with variable cutoff frequency.

rcLowPass -- A simple RC Low Pass Filter with variable cutoff frequency.

simpleTone -- A simple Bandpass Filter using **rcHighPass** and **rcLowPass**.

noiseGate -- A Noise Gate with Envelope and Threshold controls.

lookupLinear( ) -- A function used whenever a table lookup is implemented in any of the above dsp classes. It can interpolate a fractional index into a table. Value = table[integer part of index] + (fractional part of index) * (table[index + 1] - table[index]).

The Set_Codec Files

The **set_code**c files deal exclusively with the Codec class created in the **codec.h/.cpp** files. **set_codec.h** starts off declaring an instance of the WM8978 **Codec** class called **codec**.

```
extern Codec codec;
```

Using the **dot** operation on the **codec** object, the user can then call any of the convenient register "set" functions built in **codec.cpp** to configure specific Codec settings. This is exactly what **codec_sets( )** does in the **set_codec.cpp** file. It runs seven Codec "set" functions and is executed in the **setup( )** section of **main.cpp**. Check the comments in the code for an easy reference to the register setup possibilities.

```
//Some functions built to configure Codec registers

void codec_sets()         /to be executed in main.cpp
{
 codec.addaCfg(1,1);     //enable adc and dac (DAC 1/0, ADC 1,0)
 codec.inputCfg(0,1,0); //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();     //48kHz, 32kHz, 24kHz, 16kHz, 12kHz, 8kHz
 codec.hpVolSet(40,40); //headphone volume 0 TO 63, (LEFT, RIGHT)
 codec.i2sCfg(2,0);      // I2S format MSB, 16Bit
 codec.loopback(0);      //Bypass, Input to Output when 1, no Bypass when 0
 };
```

The Set_Settings Files

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

Note that some of the original WM8978 **codec.h** constants such as I2C pin assignments were moved to this file in order to keep all the main program settings in one place.

set_settings.h also declares the **I2S_init( )** function. If you remember, I2S is the codec interface used to move the audio data into and out of the codec. This function is defined in detail in **set_settings.cpp** and executed in the **setup( )** of **main.cpp**. Many of the settings in **I2S_init( )** use the constant labels defined in **set_settings.h**.

Here is the **set_settings.h** file specific to the PÚCA board and the attached pots, pushbuttons and LEDs described in this project:

```
#ifndef SETTINGS_H_
#define SETTINGS_H_

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

  #define SAMPLE_RATE     (48000)
  #define BITS_PER_SAMPLE (16)
```

```
  #define CHANNEL_COUNT 2

  //ESP32 PUCA PIN ASSIGNMENTS
  //~~~~~~~~~~~~~~~~~~~~~~~~~~

  #define POT1 32
  #define POT2 34
  #define POT3 35
  #define POT4 33

  #define LED1 4
  #define LED2 2

  #define KEY1 14  //T6
  #define KEY2 13  //T4
  #define KEY3 15  //T3
  #define KEY4 21
  #define KEY_BOARD 36

  #define TOUCH_THRESHOLD 30

//ESP32-Codec PIN SETUP

#define I2S_NUM               (0)
#define IS2_MCLK_PIN (0)
#define I2S_BCLK    (23)
#define I2S_LRC     (25)
#define I2S_DIN     (27)
#define I2S_DOUT    (26)

#define Codec_SDA          19   //SDA
#define Codec_SCK          18   //SCL
#define I2C_MASTER_SCL_IO  18
#define I2C_MASTER_SDA_IO  19
#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

#define FRAMELENGTH    256
#define AUDIO_PROCESS_PRIORITY  10

#define DMABUFFERLENGTH 64
#define DMABUFFERCOUNT  4

// processor timing variables for system monitor in task.cpp
  extern unsigned int runningTicks;
  extern unsigned int usedticks;
  extern unsigned int availableticks;
  extern unsigned int availableticks_start;
  extern unsigned int availableticks_end;
  extern unsigned int usedticks_start;
  extern unsigned int usedticks_end;
  extern unsigned int processedframe;
  extern unsigned int audiofps;

 void I2S_init(void);
```

```
#endif
```

Effects Programming

Programming Effects will mainly involve the **set_module.h**, **set_module.cpp**, and **main.cpp files**. Two examples will be given for the PÚCA board, first a simple volume control effect called gainDoubler, and then a more involved Stereo Chorus effect that uses some of the DSP class tools from the files **bsdsp.h** and **.cpp**.

The Set_Module Files for gainDoubler

Set Module Header

With the following lines the **set_module.h** file makes two important code declarations:

```
      class controller_module:public controllerModule
{
public:
        float gain;
        float gainRange;
        void init();
        void onButtonChange(int buttonIndex);
        void onControlChange(int controlIndex);
};
      //controller_module myPedal declaration with extern
      extern controller_module *myPedal ;
```

It first creates a child class called **controller_module** derived from the base/parent controller class, **controllerModule**, which was built in the **controller_mod** files. Within the class brackets are those base **controllerModule** methods that will be needed for your particular effect application. Here you must also include any additional class attribute variables needed in the effect, most often to hold the values read off the pots and switches such as the **gain** and **gainRange**. A child class inherits all the elements of the base class but you can also expand it with elements of its own such as these new effects attributes.

In the final line an instance object called **myPedal** is created from the child class controller_module. In actuality, **myPedal** is declared as a pointer to the instance object. This will make the code a bit easier to read. The access operator for **myPedal** as a pointer will be "**->**" while the simple dot is the access operator for the different properties of **control[ ]** and **button[ ]**. As an example, accessing a pot value will look like `myPedal->control[3].value`

Set Module CPP

The file **set_module.cpp** will flesh out the details of this new child class, **controller_module**, but first it defines **myPedal**, which was only declared above in the header file. This will immediately trigger the **controllerModule** Constructor method from **controller_mod.cpp** which initializes all the members of **control[ ]** and **button[ ]**. Specifically, all the member mode's are set to **DISABLED**.

```
controller_module *myPedal = new controller_module();
```

Set Module Init( )

The first **controller_module** class method defined is **init( )**. Here we can start off giving our effect an actual name. Our example effect here is "Gain Doubler" which will provide simple signal amplitude control. Next, the **init( )** is a convenient place to configure any ESP32 pins connected to switches as inputs with pullup resistors, and any ESP32 pins connected to LEDs as digital outputs.

```
  name = "gainDoubler"

  pinMode(KEY1, INPUT_PULLUP);
  pinMode(KEY2, INPUT_PULLUP);
  pinMode(KEY3, INPUT_PULLUP);
  pinMode(KEY4, INPUT_PULLUP);

  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
```

Next, within **init( )**, the properties of the class attributes **control[ ]** and **button[ ]** are defined. If you remember, the class Constructor method initializes all these properties making all the controls and buttons inactive. That means we only have to enable and set up two buttons and two controls since that is all the controllers used in this effect pedal.

```
//setting up the buttons

  button[0].name = "KEY1";
  button[0].mode = BM_MOMENTARY;
  button[0].pin = KEY1;

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

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

  //add range control
  control[1].name = "Range";
  control[1].mode = CM_SELECTOR;
```

```
control[1].levelCount = 3;
control[1].pin = POT2;

gain = 1.0;
gainRange = 1.0;
```

Note that many of the **control[ ]** and **button[ ]** properties are not implemented either here or in the **controltask( )** and **buttontask( )** functions. They are available for future use. The **control[ ]** and **button[ ]** properties set up in **init( )** are directly used by **buttontask( )** and **controltask( )** in the file **task.cpp**. In fact, this **init( )** method is executed in **main.cpp** right before **taskSetup( )** which starts up the button and control tasks.

Note also that the two extra class attributes **gain** and **gainRange** are defined within **init( )** and given initial values.

Set Module Event Handlers

After adjusting the pot and button values according to their respective **mode** and **levelCount** properties, the button and control tasks send out their final controller value to the methods **onButtonChange( )** and **onControlChange( )**. What happens inside these two methods is defined next within the file **set_module.cpp**.

```
void controller_module::onButtonChange(int buttonIndex)
{
  switch(buttonIndex)
  {
    case 0: //"KEY1" button state has changed
    {
      if(button[0].value) //if effect is activated
      {
        //codec.loopback(0);
        codec.hpVolSet(40, 40);
        digitalWrite(LED1, HIGH);
      }
      else //if effect is bypassed
      {
        //codec.loopback(1);
        codec.hpVolSet(0, 0);
        digitalWrite(LED1, LOW);
      }
      break;
    }

    case 1: // "KEY2" button[1] state has changed
    {
      if(button[1].value) // just test LED and Switch
      {digitalWrite(LED2, HIGH);}
      else \
      {digitalWrite(LED2, LOW);}
      break;
    }
  }
}
```

In the above **onButtonChange( )** method the switch/case steps through each of the hardware pushbuttons used in this effects box (only two in this case) and sets an action for each of them. The one switch will use the codec "set" method **hpVolSet( )** to act as a Mute and set an LED as an indicator light. The other switch turns on and off the other LED in toggle mode as a simple test.

```
 void controller_module::onControlChange(int controlIndex)
{
  switch(controlIndex)
  {
    case 0:
    {
      gain = (float)control[0].value/127.0;
      break;
    }
    case 1:
    {
      if(control[1].value==0)
        gainRange = 1;
      else if(control[1].value==1)
        gainRange = 2;
      else gainRange = 3;
      break;
    }
  }
}
```

In the above **onControlChange( )** method the value from the "**gain**" pot is converted to a float and then divided by 127 resulting in a fractional value between 0 and 1. The other pot is used for "**gainRange**". In **init( )** it was configured in **CM_SELECTOR** mode with 3 levels. As such it will act like a 3 position selector switch, with the **gainRange** attribute values of 1, 2, or 3. These two object variables, **myPedal->gain** and **myPedal->gainRange**, will then be used in the actual signal processing loop within **main.cpp** to affect the signal volume.

To summarize, the **init( )** method is defined mainly to set up the **control[ ]** and **button[ ]** properties used in **buttontask( )** and **controltask( )** to adjust the values read from the ESP32 controller pins. The two event handlers, **onButtonChange( )** and **onControlChange( )**, use the final **button[ ].value** and **control[ ].value** to affect the signal processing either directly through codec methods or indirectly through created processing variables.

The Main.cpp File for gainDoubler

The **main.cpp** file stands apart from all other files. It takes the place of the **main( )** function found in most C/C++ programs. As such, it is considered by the compiler to be the programming entry point, the first method that will get executed by the compiler. All the file content described up to this point serves only as input support to **main.cpp**. All previously defined methods await execution only from within **main( )**. In general, no function can be called from outside of **main.cpp** except the class Constructor methods which is called when a class instance object is created.

Since we are programming in an **Arduino Framework** from the **PlatformIO IDE** the **main.cpp** file has the same format as an Arduino sketch file (labeled with the **.ino** extension in the Arduino IDE but with the **.cpp** extension here). It includes two main functions, one called **setup( )** and another called **loop( )**. The first is called one time only at the start of the program. The second is called repeatedly in an infinite loop as the program continues.

Includes

Preceding **setup( )** within the **main.cpp** file are a number of **#includes**:

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

`#include <Arduino.h>` is needed because the Arduino IDE is not being used. Without it the compiler would not recognize various Arduino constant labels like **HIGH**, **LOW**, **INPUT_PULLUP** and such. Note that all three "set" files described above are in the **#include** list along with the task file.

Setup( )

The **setup( )** function within **main.cpp** is the official program start point. Within **setup( )** several functions are executed that initialize and startup the codec and its two interfaces -- I2C, used to load the codec registers, and I2S, used to control the audio data flow. The file **set_codec.cpp** built the function **codec_sets( )** which includes several codec register "set" methods to configure the codec. That is executed here in **setup( )**. Finally, the Serial Monitor is turned on with **Serial.begin(115200)** so that the System Monitor can be run if the user wants some continuous visual feedback on the effects controllers and system loads.

```
void setup()
```

```
{
  //~~~~~~~~~~~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("Init success!!");

  //~~~~~~I2S is initialized. See set_settings.cpp~~~~~~~~~

      I2S_init();

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

      Serial.begin(115200);
      delay(1000);

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

} //Setup End
```

Loop( )

The **loop( )** function within **main.cpp** is the heart of the effects pedal program. The signal processing code for the effect is contained in **loop( )** within an inner loop bound by the brackets of **while(1) { ... }**. Code inside the **loop( )** function is automatically assigned by the compiler to Processor Core 1.

Before the inner loop is entered, all buffers and floating point variables used in the signal processing code are defined. Then **myPedal init( )** (see **set_module.cpp**) is called along with **taskSetup( )** (see **task.cpp**) before the inner loop starts.

Below is the complete inner **while(1)** loop containing the code for the simple gainDoubler audio effect.

```
while(1)
{
  i2s_read(I2S_NUM_0, rxbuf, FRAMELENGTH*2, &readsize, 20);
  for (int i=0; i<(FRAMELENGTH); i+=2)
   {
    rxl = (float) (rxbuf[i]) ;   //convert samples 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 samples back to integer
    txbuf[i+1] = ((int16_t) txr) ;
   }
  i2s_write(I2S_NUM_0, txbuf, FRAMELENGTH*2, &readsize, 20);
}
```

This is simple amplitude control of a 16-bit audio signal. It starts off with **i2s_read( )** loading **rxbuf** with **FRAMELENGTH** number of samples from a DMA receive Buffer filled by the ADC under DMA control. Since the samples are 16 bits or 2 bytes (**BITS_PER_SAMPLE** = 16) and **i2s_read( )** reads a byte at a time, it must read a total of **FRAMELENGTH*2** bytes. The signal is stereo (**CHANNEL_COUNT** = 2) so the incoming samples alternate between left and right channels.

A "**for**" loop will step through the frame of samples one at a time. The sample is first converted to a floating point number and placed in a temporary holding float variable, **rxl** for left channel and **rxr** for right channel. Floating point math usually has a severe negative effect on CPU performance time but, amazingly, the ESP32 has a built-in **FPU** (Floating Point Unit) which provides acceleration on single precision floating point arithmetic.

The middle two lines are the actual signal processing code:

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

The variables **gain** and **gainRange** are derived from two pot values, as defined in the file **set_module.cpp**. The variable "**gain**" has been adjusted to result in a floating fractional value between 0 and 1. The "**gainRange**", also defined in **set_module.cpp**, comes from a 3-selector switch pot with values 1, 2, and 3. These are all multiplied with the left and right signal samples for a super simple amplitude control.

The processed right and left float values are then converted back to 16-bit integers (**int16_t**) and loaded into the **txbuf**. When the "**for**" loop is finished processing the entire frame's collection of samples, the **i2s_write** function is ready to load them all into a DMA transmit buffer to be fed to the DAC output at the **SAMPLE_RATE** under DMA control.

Timing is critical here. The **i2s_write( )** function must happen before the transmit buffers go empty from the DMA constantly feeding the output DAC; and the **i2s_read( )** function must happen before the receive buffers are completely filled from the DMA constantly feeding them input ADC samples. An overflowing receive buffer or an empty transmit buffer can be prevented by careful settings of **DMABUFFERLENGTH**, **DMABUFFERCOUNT**, and **FRAMELENGTH**, and, of course, by keeping the signal processing time as short as possible.

The Set Module Files for Chorus

Set Module Header

The **set_module.h** file first declares two instances of the **fractionalDelay** DSP class and two instances of the DSP **oscillator** class, both found in the **bsdsp** files. It then sets up the child class **controller_module** and outlines all the class elements needed for the Chorus Effect including several new attributes - **depth**, **freq**, **beatFrequency**, **phaseDiff**, **asynch**, and **stereo**.

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

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

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

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

Set Module CPP

The **set_module.cpp** file completes the definition of DSP instance objects **delay1**, **delay2**, **lfo1**, **lfo2** and initializes the delay buffer sizes with the **fractionalDelay** class method **init( )** found in the file **dpdsp.cpp**.

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

Set Module Init( )

It then sets up all the necessary **pinModes** and the **button[ ]** and **control[ ]** properties for the physical switches and potentiometers to be used to control the chorus effect parameters. Notice also that initial values are set for all the new chorus attributes declared above, even making use of an **oscillator** class method, s**etFrequency( )** to initialize the two **oscillator** instances with frequency values.

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

// Define the ControllerModule functions declared above
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
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
  // For mode details, see control_task() and button_task() in task.cpp
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(KEY1, INPUT_PULLUP);
  pinMode(KEY2, INPUT_PULLUP);
  pinMode(KEY3, INPUT_PULLUP);
  pinMode(KEY4, INPUT_PULLUP);

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

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

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

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

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

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


  freq=5;
  depth=0.5;
  beatFrequency=2.5;
  stereo = 1;
  asynch = 1;
  lfo1.setFrequency(freq);
  lfo2.setFrequency(freq+beatFrequency);
 }
```

Set Module Event Handlers

Finally, the button and control event handlers are defined.

```
void controller_module::onButtonChange(int buttonIndex)
{
  switch(buttonIndex)
  {
    case 0: //main button state has changed
    {
      if(button[0].value) //if effect is activated
      {
        codec.loopback(0);
        //codec.hpVolSet(40, 40);
        digitalWrite(LED1, HIGH);
      }
      else //if effect is bypassed
      {
        codec.loopback(1);
        //codec.hpVolSet(0, 0);
        digitalWrite(LED1, LOW);
      }
      break;
    }
    case 1: //the button[1] state has changed
    {
      if(button[1].value) // just test LED and Switch
      {digitalWrite(LED2, HIGH);}
      else
      {digitalWrite(LED2, LOW);}
      break;
    }
  }
}
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void controller_module::onControlChange(int controlIndex)
{
  switch(controlIndex)
```

```
  {
    case 0: //rate
    {
      freq = 0.5 + 10 * (float)control[0].value/127.0;
      lfo1.setFrequency(freq);
      lfo2.setFrequency(freq + beatFrequency);
      break;
    }
    case 1:  //depth
    {
      depth = 1.49 * (float)control[1].value/127.0;
      break;
    }
    case 2:  //phase or frequency difference
    {
      beatFrequency = 5 * (float)control[2].value/127.0;
      phaseDiff = (float)control[2].value;
      lfo2.setFrequency(freq + beatFrequency);
      break;
    }
    case 3:  //stereo
    {
      stereo = (bool)control[3].value;
      break;
    }
  }
```

The Main.cpp File for the Chorus Effect

Most of the **main.cpp** file will remain the same as described for the gainDoubler. This will be the case for any programmed effect. The only change is to the sample processing part of the **while(1)** loop. Here those lines are framed by the "stereoChorus Processing" comment lines (shown in bold).

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

      setDebugVars(myPedal->depth, myPedal->freq, 0, 0);

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

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

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

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

     lfo1.update();
```

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

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

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

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

Here is a short description of the signal processing going on.

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

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