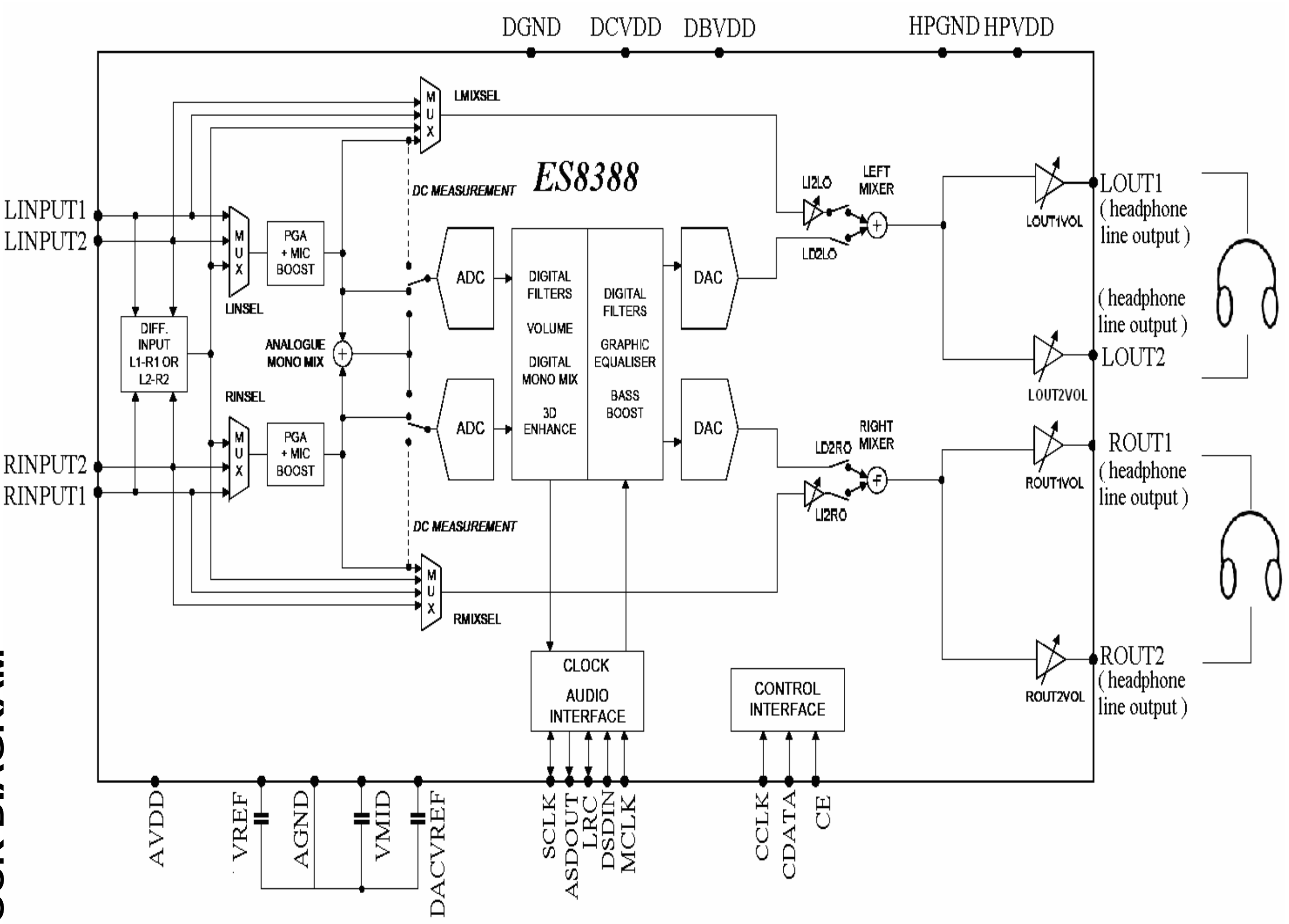
Codec Effects Software for the ESP32

John Talbert - December 2022



BLOCK DIAGRAM

Table of Contents

Acknowledgements

Many thanks to Hasan Murod who created the 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.

Project Goals

The main goal of this project was a simple software platform for exploring audio effects on an ES8388, or other codec, connected to the ESP32 microprocessor. All parts of the package should be straightforward and accessible. Though this is a scaled back version of the Blackstomp package, its best attributes were retained:

- A robust codec driver.
- A versatile controller interface providing the user full setup and use of devices such as potentiometers, switches, LEDs and sensors.
- Deployment of the ESP32 dual processors and FreeRTOS for superior program efficiency.
- A system and error monitoring tool.
- Full access to Blackstomps' collection of dsp audio effects

There is much to learn here about c++ programming, how codecs work, and about some surprising features built into the ESP32. This paper should serve somewhat as a tutorial on the following topics with concrete examples in the package code:

Object Oriented Programming
- The IDE Platform
- OOP terminology such as class, attribute, method, instance, etc.
- Header/CPP file pairs
- C++ structures such as #define, enum, enum typedef, struct

Codec Structure
- I2C Interface for registers
- I2S Interface for ADC and DAC
- Parameter Registers
- Register Methods

Controllers
- Digital - buttons, switches
- Analog - pots, sensors
- Controller arrays with member typedef enum parameters
- Controller value manipulation
- Controller Event Handlers
- Controller OOP base class, child class, instance object

ESP32 Special Features
- Dual Processor Cores and Tasks
- FreeRTOS (Real Time Operating System)
- DMA (Direct Memory Access)
- FPU (Floating Point Unit)
- I2C Serial Interface
- I2S Audio Serial Interface

Audio Effects Coding
- ADC and DAC with DMA
- DMA memory
- Integrating Physical Controllers
- Integrating DSP classes (Digital Signal Processing)

Program Monitoring and Error Detect

C++ Software Structure

The IDE Platform

This software package was programmed in an Arduino Framework built over top of C++ and as such it should be compatible with the Arduino IDE (Integrated Development Environment). However, the IDE actually used was the PlatformIO within Visual Studio Code. Instructions for installing and using PlatformIO can be found at https://randomnerdtutorials.com/vs-code-platformio-ide-esp32-esp8266-arduino/ It has the following advantages:

• Advanced Editor with color coding and auto-complete code entry suggestions
• Detailed Error highlighting and descriptions
• Multiple open file tabs
• Code search and navigation over multiple files
• Project files loaded with specified versions of dependent libraries
• Auto detection of your COM port
• Included Terminal window
• Included GIT tool for saving different stages of your work

OOP Terminology

Object Oriented Programming is used throughout this project. OOP has its own special structure and terminology which will be summarized here to give you a better understanding of the software coding.

The central component of OOP is the **Class**. This is a kind of container for related variables called **attributes**, and functions called **methods**. Here is an example of a class declaration:

```
class   className
{
  private:
      int x;
      int y;

  public:
      int attribute1;
      bool attribute2;
      ATTRIBUTE3 my_TypedefEnum;

  className( );    // Class Constructor
  ~className( );  // Class Destructor
  bool method1( );
  bool method2( );
```

```
        int getX( );
        bool setX(int x_value);
      };
```

The class declaration starts off with a list of class **attributes** (variables) and ends with a number of class **methods** (functions). Notice the "access specifiers" **private** and **public**. **Public attributes** can be accessed outside the class. **Private attributes** can only be accessed through special class methods. Some of these special methods may include functions generically called "getters" and "setters". Notice the getter method **getX( )** used to return the value of the private integer x attribute. Likewise the **setX( )** method sets the value of the private x attribute and returns a true if successful.

The **className( )** method in the above example is a special function called the class **Constructor**. This unique method has the same name as the class name and is executed only once when a version, or instance, of the class is first created. Usually it is used to initialize the class attributes. Similarly, the **Destructor** is a special method used to clean up when the class is no longer needed.

Notice how sparse the above class declaration is. The attributes are not given values here and the methods reveal only input/output structure but no details of what it actually does. This is called a class **"declaration"** and is placed in an **.h**, or header file (somefilename.h). A paired **.cpp** file (somefilename.cpp) is then used to fill out the details of the class, referred to as the class **"definition"**. The form a method definition takes is the following using the double colon operator.

```
int className::getX( ) { method details }
```

An OOP Class acts only as a kind of template. Before actually using it in your code you must first create one or more **"instances"** of the class (or instantiate the class) like so:

```
className  myClassname;
```

myClassname is then an **"object"** of the class **className**. Elements of this new class object can be accessed using a **dot** operator like `myClassname.parameter1`, or `myClassname.method1( )`.

Alternatively, the new class object can be instantiated as a **pointer**:

```
className *myClassnamePointer;
```

The elements of this pointer to a new class object must be accessed using an **arrow operator** like `myClassnamePointer->parameter1` or `myClassnamePointer->method1( )`.

Another useful alternative is to use OOP Inheritance to create a new **"child"** class from "className", add your own extra features to the child class, and then instantiate this expanded child class. The child class is said to be **"derived"** from a **"base"** or "**parent**" class and it inherits all the attributes and methods of its base class. You can then add more attributes and methods, expanding on those of the base class.

```
class child_of_className:public className {    };
child_of_className  myChild;
```

Header and CPP Files

One of the more difficult aspects of C++ programming in a complex multi-file OOP project is code organization and placement. Compile errors can easily crop up complaining of "undefined" variables or methods "within this scope" even though they were defined elsewhere. Then the opposite error of "multiple definitions" can happen even though they were defined only once, but were #included on several file levels (including a file that includes a file with definitions).

The best way to mostly avoid such problems is to adhere to the common program structure of paired **.h** header **declaration** files with **.cpp definition** files. The header file can be "included" at the start of any file that requires its content with this line: `#include "somefile.h"` Note that, at the very least, this **#include** line must always be added to the top of its paired file.cpp.

In summary, each class or group of utility functions will place their definitions in a separate **.cpp** file that **defines** everything in detail what was simply **declared** in a corresponding **.h** file. The **.h** file can then be #included by the many different parts of the program that need those classes or functions. Never name a **.cpp** file in an **#include**. In general, **#include** names only **.h** files.

In this way each **.cpp** file is separately compiled only once, and then linked, using the **.h** file **#include** directives, with the output of other **.cpp** files to form the complete program. If there are pieces of code that need to be #included in more than one other file of the program, you may need to separate the code out as smaller functional unit **.h** /**.cpp** file pairs to avoid the "multiple definitions" error.

A **.cpp** file can sometimes include code **declarations** but only when it doesn’t need to share them with other parts of the program. However, keep in mind that the primary purpose of a **.cpp** file is to contain **definitions** that must only be compiled once.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

It's important to understand the distinction between "declaring" and "defining".

Declaring a variable (in a **header** file **.h**) simply declares the existence of the variable to the program. It tells the compiler that a variable of a certain type exists somewhere in the code. You declare a **float** variable as follows: `float x;`
At this point, the variable doesn't have any memory allocated to it. The compiler only knows that a **float** variable named x exists somewhere in the code. Defining the variable (usually in the paired **.ccp** file), on the other hand, means declaring the existence of the variable as well as allocating the necessary memory for it. You define a variable as follows: `float x = 10.14;`

You can declare a variable as many times as you want, but you can define a variable only once. This is because you cannot allocate memory to the same variable multiple times (multiple definitions error).

To guarantee against the dreaded "multiple definitions" error use the **"extern"** keyword in front of all variable, function, and class **declarations** in the **.h** header file as follows: `extern float x;` Then, in the paired .ccp file **define** the variable as usual in detail: `float x = 0;`

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

There are two forms of **#include**, **#include <somefile>** and **#include "somefile.h"**. The **#include < >** instructs the processor to look for the file in a standard C directory or external library. The **#include " "** instructs the processor to first look for the file in the current directory of user programmed files, before checking the external C directory.

Constant Labels

This software package makes liberal use of names or labels associated with integer constants. These are often indicated by all-cap labels and are used to make the code more readable. They come in the several different forms.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#define is a preprocessor directive. It is not a program statement. Before any compilation, the preprocessor runs through the code replacing any #defined **LABEL** with its defined value. Note that **#define** is different from a "**const**" declaration which has a scope. Also note that there is no comma or semicolon at the end of the declaration.

```
        #define Codec_DACCONTROL30 0x34
```

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

An "**enum**" is similar to **#define** in that it gives a set of integers individual labels. The computer uses the number while the programmer can use the more descriptive label. If an actual integer value is not designated behind an equal sign within the enum, the integer values will start with 0 in the list and increment by one.

```
enum
{
  DATA_FORMAT_I2S   = 0x00,
  DATA_FORMAT_LEFT  = 0x01,
  DATA_FORMAT_RIGHT = 0x02,
  DATA_FORMAT_DSP   = 0x03,
};
```

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

An **enum** can be combined with "**typedef**" to give a "type" name to the data in the **enum** list. When functions need very specific input argument values, a **typedef** in the function declaration and **typedef** labels in the function definition can nail down exactly what function input is required from the user. One useful convention it to put a "**_t**" at the end of the **typedef** name to identify the name as a **typedef**.

```
typedef enum
{
  NORTH,
  SOUTH,
  EAST,
  WEST,
} DIRECTIONS_t;

// NORTH=0, SOUTH=1, EAST=2, WEST=3  An enum assigns numbers to Labels.
// The computer uses the numbers while you can use the labels
// With typedef you define your own type, not the usual int, bool, etc.

// declare "directions" as type "DIRECTIONS_t" defined above

DIRECTIONS_t directions;
directions = EAST;          // set directions = 2
directions = 3;             //set directions = WEST
```

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

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.

```
struct WIND //a struct is essentially a Class definition with various types
{
   DIRECTIONS_t wind_direction;
   int speed;
```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
    int min;
    int max;
    bool rain;
    int temperature;
};

WIND monday_wind;  //declare a Monday wind weather object
monday_wind.wind_direction = NORTH;
monday_wind.speed = 35;
monday_wind.rain = true;

WIND wind_week[7]; //declare a Monday through Sunday weather array
wind_week[0].wind_direction = NORTH; //set Monday's wind direction
wind_week[1].speed = 35;             //set Tuesday's wind speed
```

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

The Codec Files

Codec Basics

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 ES8388 codec the I2C interface is used to load and read 53 user programmable 8-bit registers that set up I/O connections, sampling rate, sample format, sample size, volume, filters, effects, etc.

A block diagram of the ES8388 codec by Everest Semiconductor is shown on the cover sheet of this paper. The PDF data sheet for the ES8388 can be found at http://www.everest-semi.com/pdf/ES8388%20DS.pdf

The software driver for a codec may at first seem complex but is actually very straightforward. Any seeming complexity comes from the large number of setup registers (53 for the ES8388). The bulk of the ES8388 datasheet is concerned with identifying these 53 registers. Only a few of these registers are of real interest to the user and these are provided get and set functions in the codec class. Most of the registers can be set in the **init( )** function and left at their default settings. The codec datasheet will suggest "default" values for most of the registers.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

The **codec.cpp** file starts off assigning LABEL Addresses for each of these 53 ES8388 8-bit registers. Before accessing the registers, however, a few things must be configured.

I2C Interface

First, the I2C interface, used to access the codec registers, must be set up. In **codec.h** the external library **Wire.h** is included. Then, inside the **codec.h** class declaration, the line `TwoWire  i2c = TwoWire(0)` uses code from the **Wire.h** library to declare "**i2c**" as an instance of the **TwoWire** class (the instance name can be anything you want).

Back in **codec.cpp** the **Codec** class Constructor method, **Codec(uint8_t _sda, uint8_t _scl, uint32_t _speed)**{ ... } is defined to run the **TwoWire** method **i2c.begin** after getting the two i2c interface pin numbers connected to the ESP32 and its serial data speed. This constructor function will be executed when an instance of the **Codec** class is declared in **main.cpp**.

The "**i2c**" **TwoWire** interface is now available for communicating with the codec registers. The next two methods to be defined are, of course, those that read from and write to the registers, `write_reg(register address, register data)` and `read_reg(register address)`. After that, a massive codec initialization method can be defined, **init( )**. Here all 53 ES8388 8-bit registers are initialized with 53 **write_reg( )** executions using each of the register address labels defined at the start of the **.cpp** file along with very specific 8-bit input register values. This is your cue to pull out the ES8388 data sheet.

Get and Set Methods

In effect, our Codec code is complete here. The codec is ready to use; however, it may be useful to create a few "set" and "get" methods to allow easier access to some of the more useful codec register settings. These methods are all short and easy involving only **write_reg( )** or **read_reg( )** and a few "**if/else**" decisions. Here are the ones that might be useful:

- outputSelect(the typedef outsel_t   outsel)
- inputSelect(the typedef insel_t   insel)
- DACmute(bool mute)
- setBitsPerSample( )
- setOutputVolume(uint8_t vol)
- getOutputVolume( )

- setInputGain(uint8_t gain)
- getInputGain( )
- setALCmode(the typedef alcmodesel_t   alc)
- mixerSourceSelect(the typedef mixsel_t LEFT, the typedef mixsel_t RIGHT)
- mixerSourceControl(the typedef mixercontrol_t mix)
- analogBypass(bool bypass)
- analogSoftBypass(bool bypass, the typedef bypass_mode_t bm)
- optimizeConversion(int range)
- setMicGain(uint8_t gain)
- getMicGain( )
- setMicNoiseGate(int gate)
- getMicNoiseGate( )

Several **typedef enum**s and/or **enum**s are created in **codec.h** or **codec.cpp** to make it easier to see what data bytes are appropriate to load into what registers. Several of the "set" methods defined in **codec.cpp** spell out the specific **typdef enum** to use for data input. Look for the specification ending in "**_t**" in the definition and then look up its label list from the **typedef enums** declared at the start of the **codec.h** file. This helps prevent mistakes. Those "set" methods with specified data **typedefs** will only accept labels from that specific **typedef enum**.

Other Codecs

This particular Codec Software package was built to work with the ES8388 Codec Circuit Board found at https://www.jtalbert.xyz/ESP32/. The original Blackstomp Effect Software built a "codec" base class and two derived child classes, one for the ES8388, like this one, and another for the AC101 codec. They support two versions of ESP32-A1S modules sold by Deeptronic (https://www.deeptronic.com/store/). The ESP32-A1S is an ESP32 module by AI_Thinker with one of two built-in codecs, the ES8388 or the AC101.

I have provided untested **codec.h** and **codec.cpp** files for the AC101Codec in a separate folder. At the top of the **codec.cpp** file are some commented out **#define** Labels for ESP32 pin numbers that should work with the Deeptronics ESP32-A1S AC101 module. These would replace those in the file **set_settings.h**.

For this Codec software package to work with other Codecs the 2 files, **codec.h** and **codec.cpp**, would need to be changed to fit the different parameters of a different codec. Often times, driver files for a specific codec can be found on the internet in project GIT repositories. In any case, the general format of any **codec.h** and **.cpp** file should be familiar to you since they all include the same general content:

1. Construction of a **Codec** Class.
2. A `TwoWire  i2c = TwoWire(0)` line to instantiate an **i2c** type object
3. A `Codec(uint8_t _sda, uint8_t _scl, uint32_t _speed){ ... }` method to begin the i2c interface. Here it is placed as the Codec Constructor method.
4. A long list of **#define** CodecRegisterAddress Labels
5. Two **private write_reg( )**, **read_reg( )** methods used to access the codec registers
6. One **public init( )** method to initialize all the Codec Registers. The codec tech sheet should have some default, suggested settings.
7. For convenience, a set of "get" and "set" methods for some of the more used registers.
8. A collection of **enum** labels and **typedef enum** labels to help with the "get" and "set" methods as well as the **init( )** method.

A different codec board will have different ESP32 pin connections for the codec i2s and i2c interfaces and the physical controllers. These are all set in the **set_settings.h** file. The **escodec( )** function defined in **set_codec.cpp** executes a number of "set" codec methods at the start of the program. You can enter your own list of "set" methods here with your own input values. The set_settings and set_codec files also specify operational parameters such as sample rate and bits per sample and i2s settings. Make sure that they are all correct, and agree with what is being set up in the codec.

Any codec could work with this Software Package for the ESP32 if the above considerations are met.

The Controller Module Files

The **controller_mod.h** and **.cpp** file pair sets up one of the most useful features of the Blackstomp Audio Effects package. This 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 4 switches or other digital sensors called **button[ ]**. There is no reason these array sizes can't be expanded to allow for any number of controllers if desired.

Control Array

The **control[ ]** array specifies 8 properties for each of its 6 elements as defined in a **CONTROL struct**:

- **String name;** //a user name to describe what it controls
- **CONTROL_MODE mode;** //modes of operation listed in a typedef enum
- **bool inverted;** //specify a rising or falling controller action direction
- **int min;** //set a minimum value for the controller
- **int max;** //set a maximum value for the controller
- **int levelCount;** //range of pot values is zero to this level (2 to 255)
- **int value;** //container for the current value of the controller
- **bool slowSpeed;** //slows down the controller changes
- **int pin;** // the ESP32 pin connection of the controller

These properties are accessed with a dot operation such as: `control[2].min = 2;`
or `control[4].value = analogRead(control[4].pin);`

One of the controller properties, its mode, has 5 possible modes of operation as spelled out in a **typedef enum**, **CONTROL_MODE**.

- **CM_DISABLED,** //controller not used
- **CM_POT,** //simple potentiometer action
- **CM_SELECTOR,** //selector switch action, with specified number of levels
- **CM_TOGGLE,** //pot acts like a push button toggle
- **CM_MOMENTARY,** //pot acts like a momentary push button

Button Array

The **button[ ]** array specifies 6 properties for each of its 4 elements as defined in a **BUTTON struct**:

- **BUTTON_MODE mode;** //modes of operation listed in a typedef enum
- **bool inverted;** //choose between on/off and off/on for the switch action
- **int min;** //sets a minimum value for the switch, usually 0
- **int max;** //sets a maximum value for the switch, usually 1
- **int value;** //container for the current switch value
- **int pin;** // the ESP32 pin connection of the switch

Like the **control[ ]**, the **button[ ]** properties are accessed with a **dot** operation.

One of the button properties, its mode, has 3 possible modes of operation as spelled out in a **typedef enum**, **BUTTON_MODE**.

• **BM_DISABLED,**	//button not used
• **BM_MOMENTARY,**	//button up = 0, button down = 1
• **BM_TOGGLE,**	//changes state when pressed and holds it

Several other class data elements are declared. "String name" can be used to give a name to your effect program. **INPUT_MODE**, **ENCODER_MODE**, and **BLETERMINAL** were used in the original package but not here.

Methods

Several controller methods, referred to as "event handlers", are declared:

onControllerChange(controlIndex)
onButtonChange(buttonIndex)
onButtonPress(buttonIndex)
onButtonRelease(buttonIndex)
onBleTerminalRequest( ) (Bluetooth capabilities left out of this package)

The keyword "**virtual**" is attached to each of these "event handler" methods which means that the function will be defined in detail only by the user in a descendant child class, not here in the base class where it is just declared. As you may expect from this, there is no mention of these methods in the **controller_mod.cpp** file. An **init( )** method is also one that must be defined by the user in the descendant class, not here. These methods will be defined later in the file **set_module.cpp**.

The only function defined here in **controller_mod.cpp** is the class constructor which is run later, when an instance object of a **controllerModule** class descendant is declared. Its job is to initialize all the button and control properties for that instance object, setting them up as **DISABLED**.

Controllers

As you may suspect by now, the **controller_mod** files are only a preamble to the whole controller story. There are 3 other controller components in 3 other files. These are programmed by the user and are constructed to achieve whatever audio effect the user is trying to create. They will be changed for different programmed effects.

1. All the physical controllers used will be set up in detail in the

set_module.h and **.cpp** files which also define a **controller_mod init( )** function and button/control "event handler" methods.

2. Task functions defined in **task.cpp** will continuously watch, or poll, all the controller connected ESP32 pins and keep their **value** updated and adjusted to the requested mode of operation and other controller properties.

3. Finally, in **main.cpp**, the data emerging from all the above controller processes is applied to the parameters of some amazing audio effect.

The controller devices will operate at three levels in three files as summarized here:

1. As described here, a base level controller class called **controllerModule** was built in the file **controller_mod.cpp** The class attributes include two arrays: **control[ ]** and **button[ ]**. Pots and other analog sensors are represented in the six element array **control[ ]**. Each **control[ ]** element is given 8 properties. Switches and other digital sensors are represented in a four element array **button[ ]**. Each **button[ ]** element is given 6 properties. Four **virtual** event handler methods and an **init( )** method are declared in the class. A class constructor method is defined to initialize all the **control[ ]** and **button[ ]** properties.

2. A descendant child class of **controllerModule** called **controller_module** is declared in the file **set_module.h** with the line `class controller_module:public controllerModule{ }`. The file **set_module.cpp** then defines in detail this child class' attributes and methods. Attributes such as the **button[ ]** and **control[ ]** element properties, and methods such as **init( )** and event handlers.

3. An actual instance of the **controller_module** child class called **myPedal** is created in the file **set_module.cpp** with the line `controller_module *myPedal = new controller_module();` This myPedal object is defined as a pointer so that its elements are accessed using the **->** arrow operator. **myPedal->** is used in the **main.cpp** effect process and in the file **task.cpp**'s button and control tasks.

More details in the sections to follow.

The Task File

The **task.cpp** file builds several functions labeled as "tasks" and one all-important **taskSetup( )** function that starts them all up.

ControlTask

The **controltask( )** function continuously polls all the enabled physical controllers (pots) in **control[i]** performing the following operations:

- executes `analogRead(myPedal->control[i].pin)`
- adjusts the read controller value according to the specified mode of operation, `myPedal->control[i].mode`
- loads the adjusted value into `myPedal->control[i].value`
- finally calls the event handler function `myPedal->onControlChange(i)`

The ESP32 analog input ADC pins are 12-bit with values that range from 0 to 4095. The **controltask( )** in the file **task.cpp** will knock that range down to whatever is specified by the user in **control[ ].level**, usually 127 or 255. The **taskSetup( )** function will limit the level to 255, no matter what the user specifies, though that can easily be changed. When **control[ ].mode** specifies a selector switch or pushbutton type of pot action, the user can set a drastically lower "level" such as 2 or 3. This level cut back from the 12-bit 4095 helps to stabilize the control value. That along with some hysteresis adjustments within **controltask( )** will remove most annoying jitters in the control **value** when the pot is left sitting in one position. Note that the **max**, **min** and **slowSpeed** parameters are not currently employed in the **controltask( )**.

ButtonTask

The **buttontask( )** function performs a similar continuous polling of all the enabled switches in **button[i]**:

- executes a `digitalRead(myPedal->button[i].pin)`
- adjusts the read controller value according to the specified mode of operation, `myPedal->button[i].mode` along with some debouncing
- loads the adjusted value into `myPedal->button[i].value`
- finally calls the event handler function `myPedal->onButtonChange(i)`

The **buttontask( )** will make adjustments to the polled digital pin value. The **button[ ].mode** parameter can specify a momentary or toggle switch type action. All physical switches exhibit a short burst of noise when making a transition, bouncing on and off the switch contact before settling down. The **buttontask( )** thankfully includes a "debouncing" routine to get rid of this noise. Note that **buttontask( )** does not currently utilize the **max** and **min** parameters. It also does not include a convenient "name" parameter for display by the system monitor. This can easily be changed in the code.

System Monitor Task

A System Monitor (**sysmon_task( )**) task will display real-time changes in the **button[ ]** and **control[ ]** values at an update rate of around one second. It can also display audio processing and CPU load info. CPU load info employs a collection of nine "tick" variables which are declared in the file **set_settings.h** and defined at the start of this **task.cpp** file. To engage this processing and CPU data these "tick" variables must be carefully arranged inside the audio processing code in **main.cpp**. An example of this will be provided.

The "tick" variables also illustrate a coding method that avoids some common compile errors. The variables are "declared" in the **set_settings.h** file which is #included in the **main.cpp** file that uses them. The "**extern**" keyword must be attached to the start of each variable declaration in that **.h** file. The **task.cpp** file then "defines" the same list by repeating it without "**extern**". The list is defined at the start of the **task.cpp** file prior to the task functions that use them.

The **sysmon_task( )** also offers a useful debug feature. The functions **setDebugStr( )** and **setDebugVars( )** can be used to load up to 4 program variables or a message from anywhere within the program code. The results are displayed using **sysmon_task( )**.

Another Monitor task, **runScope( )** can display a signal on the Arduino IDE's serial plotter.

FreeRTOS

The **controltask( )** and **buttontask( )** are set up to run continuously, always watching for user controller actions programmed to alter the processed audio streams in some way. To create this continuous action both functions set up an infinite loop with "`while(true) { }`". However, they must be programmed somehow for real-time response while running at the same time. The user must feel that any button press or knob turn has an immediate effect in spite of these two and other tasks all competing for processor time. The main programming tool for creating this real-time response is **FreeRTOS** (Free Real Time Operating System). The ESP32 development board comes with **FreeRTOS** firmware already installed. The Arduino IDE supports this as well. **FreeRTOS** is a Real-time Operating System used to run multiple tasks individually in sequence while making them appear to run simultaneously. This firmware allows the ESP32 board to multitask. Both tasks described above use **FreeRTOS** commands to improve responses to user input.

This is how it's accomplished. Since the human reaction time is relatively slow, around 150 ms, the tasks can be slowed down to allow other tasks to do their thing without affecting the perceived user response time. This is done with an **RTOS** Delay command at the start of each task's infinite **while(true)** loop. Both **controltask( )** and **buttontask( )** use the **RTOS** command "**vTaskDelay(1)**", a delay of about 1ms, at the start of their respective loops. Unlike the Arduino **delay( )** command, **vTaskDelay()** is a non-blocking delay; it lets other tasks continue working while the one task is idling.

Another great illustration of **vTaskDelay( )** is in the simple **framecounter_task( )** shown below. This is another infinite loop that runs concurrently with **buttontask( )** and **controltask( )**. Each time the audio processing loop (to be described later) finishes processing a bank of audio samples, called a frame, the processedframe counter will be incremented. The **framecounter_task**'s job is to capture this counter value, save it to the variable **audiofps** (audio frames per second) for display by the **sysmon_task( )**, and then reset the **processedframe** counter back to zero. This must happen only once every 1000 msec. (1 second -- for frames per second) and that is easily accomplished with the **RTOS** delay command **vTaskDelay(1000)**.

```
void framecounter_task(void* arg)  //Only useful for the System Monitor
{
  while(true)
  {
    audiofps = processedframe;
    processedframe = 0;
    vTaskDelay(1000);
  }
}
```

The original Blackstomp Effect package included a **blinktask( )** as part of an ledindicator class, which were dropped in this version. These made even more extensive use of **RTOS**. They used the interesting **Semaphore** construct. An **RTOS Semaphore** is a type of shared permit to access the processor. A class method must check on its "availability" before it can "Take" it, perform whatever it needs to do, and then "Give" it back.

taskSetup

The final function to discuss in the **task.ccp** file is the one that pulls everything together, **taskSetup( )**. It starts off with a simple check of controller pot properties in **control[i]**. It will correct any unreasonable settings made by the user in **set_module.cpp**. It then starts up the three tasks discussed above with the following lines:

```
//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);
```

These program lines require a bit of explanation:

The ESP32 microprocessor has one very useful feature. It actually has two processor cores that allows two program threads to run simultaneously. The **xTaskCreatePinnedToCore( )** is an **RTOS** Library function used to set up specific tasks (functions) to run in one of the two ESP32 cores. The last argument "0" in the xTask functions above indicates that these three tasks are assigned to the ESP32 Core 0 processor. Several other tasks defined in **task.cpp** are also assigned to Core 0, such as **sysmon_task( )** and **scope_task( )**.

The only task assigned to ESP32 Core1 is the one which manages the audio data streams to and from the codec. The processing of the audio data streams is extremely processor intensive and time sensitive. In this way the user controller functions, on the other core, will never interrupt the audio into and out of the codec ADCs and DACs.

The original Blackstomp Effect package set up an **i2s_task( )** to manage the audio data stream to and from the codec from within a controller class **process( )** method, and used **xTaskCreatePinnedToCore( )** to assign it to the Core 1 processor.

In this pared down version of the Blackstomp package the audio processing code is simply put directly within the main **loop** of **main.cpp**. This file's main **loop** code is automatically assigned to the Core 1 processor by the compiler.

The Set Files

The program files covered up to this point can, in general, be set in stone. They will not need any changes, even as different audio effects are explored by the programmer. However, the program files to be explored now are central to generating different code for different effects. These files are as follows. Note that the convention of creating file pairs is followed with **.h** file declarations and **.cpp** file definitions.

set_codec.h
set_codec.cpp
set_settings.h
set_settings.cpp
set_module.h
set_module.cpp
main.cpp

Set Codec

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 **Codec** class called **codec**.

```
extern Codec codec;
```

Note the "**extern**" keyword at the beginning of the **codec** object declaration used to avoid "multiple definition" compile errors. The **codec_sets( )** function is also declared. The **set_codec.cpp** file reveals exactly what that function does; it basically executes a number of codec "set" functions.

set_codec.cpp then defines the **codec** object with the line:

```
Codec codec(ES8388_SDA, ES8388_SCK, 400000);
```

This is actually the **Codec** Class Constructor method defined in **codec.cpp**. It will execute the **i2c.begin( )** function. **i2c** is the instance of the **TwoWire** class declared in **codec.h**. "**begin( )**" is a method found in the TwoWire library **Wire.h**, #included in **codec.h** Before running, it needs the two ESP32 pin numbers for the i2c serial interface and the serial interface speed.

Now that the **codec** and **i2c** objects are up and running, the user can call any of the convenient functions built in **codec.cpp** to change specific Codec settings. This is exactly what **codec_sets( )** does. It runs ten Codec "set" functions and is executed in the **setup( )** section of **main.cpp**. Check the comments included in the definition for an easy reference to the codec setup possibilities.

Codec set functions can easily be used within the effects code. For example, a toggle pushbutton can be set up to either mute or bypass an audio effect with the simple code here:

```
if(button[0].value)
      {
        //codec.analogBypass(true);
        codec.DACmute(1);
        digitalWrite(LED1, HIGH);
      }
```

Set Settings

The **set_settings.h** file starts off attaching all-cap labels to several integer constants using **#define**. These are preprocessor directives. All **#define** label references in the program will be replaced with the assigned integer. Many important code values are set here such as sample-rate, bits-per-sample, number of audio channels, ESP32 pin numbers for all the pot and switch connections, ESP32 pin numbers for the i2c interface and the i2s codec interface, audio processing settings for DMA size and Framesize.

Next, the "tick" variables used by **sys_mon( )** to display CPU and processing loads are all declared here, to be defined later at the start of the file **task.cpp**.

Finally, **set_settings.h** declares the **I2S_init( )** function. This function is defined in detail in **set_settings.cpp** and executed in the **setup( )** of **main.cpp**. Here are its main components:

1. **i2s_config** This **typedef enum** specifies 11 i2s interface settings. Some of these are automatically set using **#define** labels such as **SAMPLE_RATE**, **BITS_PER_SAMPLE**, **DMA_BUFFERCOUNT**, and **DMA_BUFFERLENGTH**. The other settings can be left as is, unless the user wants to dig into the i2s library to find out what they actually do.

2. **pin_config** This typedef enum uses **#define** labels to configure the ESP32 pins connected to the codec's i2s interface connections.

3. **PIN_FUNC ...** I2S configure, cryptic commands mainly to set the I2S master clock to GPIO 0

4. **i2s_driver_install( )** I2S driver initialized using **i2s_config**

5. **i2s_set_pin( )**   I2S pin connections set using **pin_config**

6. **i2s_set_clk( )**   Bit clock settings using constants found in
   set_settings.h

A warning is appropriate at this point. The two **set_settings** files are critical. Even though they won't need to be changed very often, even a single wrong setting can prevent the whole effects package from working. Be forewarned.

Set Control

We are now finally at the level where you can start coding your own special effects. The **set_module** files will make the connections between your signal processor and all external pots, switches and other sensors.

Set Control 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 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. 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 Control 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 Control 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.

```
pinMode(LED1, OUTPUT);
 pinMode(LED2, OUTPUT);
 pinMode(KEY1, INPUT_PULLUP);  //internal pullup
 pinMode(KEY2, INPUT_PULLUP);
```

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 one button and two controls since that is all the controllers used in this effect pedal.

```
//setting up the buttons
  button[0].mode = BM_TOGGLE;
  button[0].pin = KEY1;

  //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 **button( )** 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( )**.

Set Control 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: //main button state has changed
    {
      if(button[0].value) //effect activated
      {
        //codec.analogBypass(false);
        codec.DACmute(0);
        digitalWrite(LED1, HIGH);
      }
      else //effect muted
      {
        //codec.analogBypass(true);
        codec.DACmute(1);
        digitalWrite(LED1, LOW);
      }
      break;
    }
  }
}
```

In the above **onButtonChange( )** method the switch/case steps through each of the hardware pushbuttons used in this effects box (only one in this case) and sets an action for each of them. The one switch will turn off and on the codec Mute mode (or

Bypass) and set an LED as an indicator light.

```
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

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**.

One exception to this rule is the Class Constructor. This special function is recognizable by having the same name as the Class. The Constructor method is called immediately upon instantiation of a Class Object. It is usually used to set up all the variables and other arguments of an instance object upon its creation.

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. Some of this work, however, has already been initiated within various Class Constructor methods. Here is a list of startup function calls and the location from which they were called:

1. **TwoWire i2c = TwoWire(0);**   **Codec.h/Class Codec**
   Instance of **TwoWire** created, **i2c**. Constructor of **TwoWire** is run from included **Wire( )** library?

2. **Codec codec**   **set_codec.h**

Instance of **Codec** created, **codec**. No Constructor method without input parameters is defined. An "overload" constructor with input parameters runs elsewhere (see #3).

3. **Codec codec(...)   set_codec.cpp**
   Overloaded Constructor is called, **codec(ES8388_SDA, ES8388_SCK, 400000)**. Defined in **Codec.cpp**. The Constructor, in turn, runs **TwoWire i2c.begin( )**.

4. **codec.init( )   setup( )/main.cpp**
   Method found in **Codec.cpp**. Initializes all 53 ES8388 registers

5. **codec_sets( )   setup( )/main.cpp**
   Method found in **set_codec.cpp**. Some special Codec registers are overloaded using methods defined in **Codec.cpp**.

6. **I2S_init( )   setup( )/main.cpp**
   The function that initializes the i2s interface defined in **set_settings.cpp**

7. **controller_module *myPedal   set_module.h, .cpp**
   Instance of child class **controller_module** is created as the pointer "**myPedal**". Constructor method found in **controller_mod.cpp** is run initializing **control[6]** and **button[4]** arrays.

8. **myPedal->init( )   loop( )/main.cpp**
   User created **init( )** called, defined in **set_module.cpp**. Sets up the needed **control[ ]** and **button[ ]** parameters.

9. **taskSetup( )   loop( )/main.cpp**
   Defined in **task.cpp**. Performs some **control[ ]** parameter checks and then starts up **controltask**, **buttontask( )** and **framecounter_task( )** in processor core 0.

The **setup( )** function also starts up the Serial Monitor with "**Serial.begin(115200);**" During startup several messages are sent out to the monitor to let the user know everything is starting off well. If desired, the **setup( )** function can run the System Monitor task with **runSystemMonitor( )**. It will start up **sysmon_task( )** placing it in processor core 0 along with the button, control and framecounter tasks.

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. It is important to make sure that the bit settings **int16_t** defined here for the two buffers agree with the same settings as **BITS_PER_SAMPLE** set in the **set_settings.h** file (except for the 24 bits setting which is a special case described later). Also, note that the **myPedal init( )** method (see **set_module.cpp**) is called once along with the **taskSetup( )** function (see **task.cpp**) before the inner loop starts.

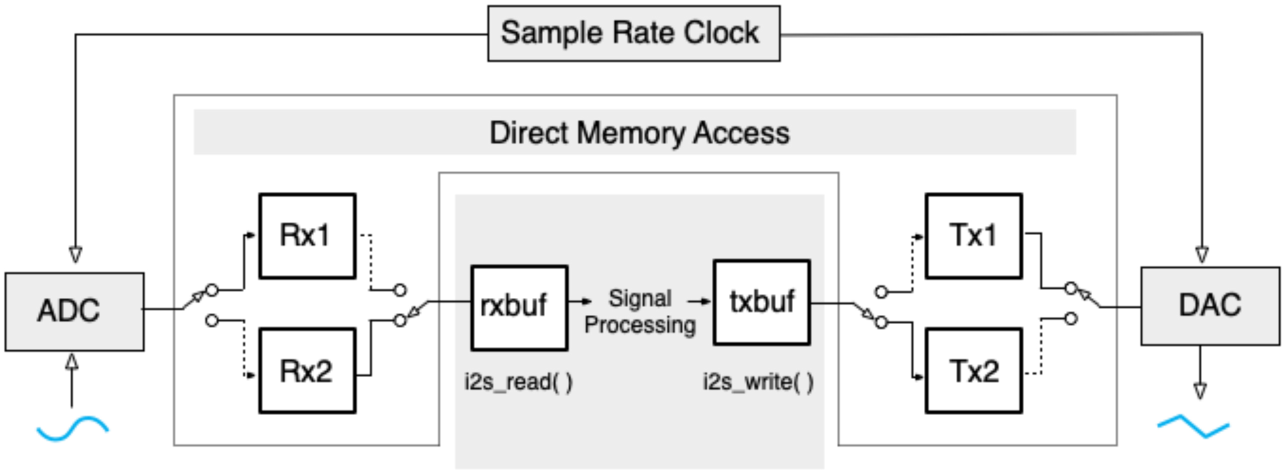
Direct Memory Access (DMA)

Signal processing within the inner **while(1)** loop is based upon an I2S interface working in conjuction with a DMA (Direct Memory Access) processor built into the ESP32. Though much of the complex details of this action is hidden behind the scenes, a basic understanding is useful.

DMA allows the codec DACs and ADCs to access a block of system memory without involving the ESP32 processor. A master clock will time the loading of audio samples one at a time at the designated **SAMPLE_RATE** into a DMA Rx block of memory from the ADC. At the other end, samples waiting in another DMA Tx block of memory will be loaded to the DAC one sample at a time at the same designated **SAMPLE_RATE**. These transfers between the memory and the converters are supposedly initiated by the processor but the actual transfer is all taken care of by the DMA, leaving the processor free.

In-between the DMA Rx holding ADC samples and the DMA Tx holding DAC samples lies the user's signal processing. The function **i2s_read( )** collects a "frame" of samples from the DMA Rx storing them in the user buffer **rxbuf[ ]**. The **#define** label **FRAMELENGTH** sets the number of samples collected in a frame. Signal processing code then works on the samples one sample at a time, moving the processed sample to user buffer **txbuf[ ]** when done. Finally, after processing all the samples in the frame, the function **i2s_write( )** moves the data in the **txbuf[ ]** to the DMA Tx.

Here is a block diagram of the process.



Both the ADC and DAC are being clocked at the sample rate. The figure shows the ADC samples being routed to the DMA buffer Rx1, while the samples feeding the DAC come from DMA buffer TX1. The DMA allows peripherals like the ADC and DAC to directly access system memory without involving the CPU while it also acts as a kind of memory traffic control.

When the Rx1 buffer is full the DMA will flip the two switches at the ADC end of the figure to now send ADC samples to the Rx2 buffer and connect the user **rxbuf** to the full Rx1. The CPU will supposedly receive an interrupt to initiate transfer of the Rx1 buffer to the user **rxbuf** using the function **i2s_read( )**. In a similar fashion, when Tx1 is empty the DMA will flip the two switches at the DAC end of the figure. The DAC will now receive samples from the Tx2 buffer while a CPU interrupt initiates emptying the **txbuf** into Tx1 using the function **i2s_write( )**.

After an **i2s_read( )** operation the audio samples in **rxbuf[ ]** are processed for a specific audio effect, one at a time, and then loaded into **txbuf[ ]**. Finally, when the entire "frame" of samples have been processed and loaded into **txbuf[ ]**, the operation **i2s_write( )** can be executed to load the processed samples into a DMA Tx buffer.

The user sets up the DMA buffers using two **#define** constants, **DMABUFFERLENGTH** and **DMABUFFERCOUNT**. These values are configured by the user in s**et_settings.h** and used by **i2s_config** in the file **set_settings.cpp**. The buffer length is the size in bytes of each DMA buffer. The buffer count is the number of these buffers available to the DMA. The figure shows 2 buffers at the ADC and 2 at the DAC. From the operation described above you can see why at least two are required at each end, though more could be assigned by the user. The total DMA memory used is given by the following formulas:

DMABUFFERLENGTH * DMABUFFERCOUNT * BITS_PER_SAMPLE/8 * CHANNEL_COUNT
Total memory must not exceed 4092

Actual DMA buffersize is
DMABUFFERLENGTH * BITS_PER_SAMPLE/8 * CHANNEL_COUNT

Set a lower size for low audio signal latency between input and output
Set a higher size for less CPU interrupt involvement.

DMABUFFERLENGTH must be a value between 8 and 1024 in bytes.
DMABUFFERCOUNT must be at least 2.

A third **#define** constant, **FRAMELENGTH**, defines the number of audio samples collected, called a "frame", for signal processing. This value is used to set the size of the user buffers **rxbuf[ ]** and **txbuf[ ]**. It must be equal or greater than **DMABUFFERLENGTH * CHANNEL_COUNT**.

In practice, there seems to be a lot of leeway in the above value settings. I've had luck with **DMABUFFERLENGTH** = 64, **DMABUFFERCOUNT** = 2, **FRAMELENGTH** = 256. There is still a lot of uncertainty in the tech sources. **CHANNEL_COUNT** is actually called **slot_num**. Does that stand for ADC plus DAC channels so that a stereo in and out would require **DMABUFFERCOUNT** = 4? The spec sheet also says this: "The receiving buffer that is offered by user in i2s_read should be able to take all the data in all dma buffers, that means it should be bigger than the total size of all the dma buffers" That seems excessive given what actually works in practice.

For a more detailed examination of I2S for the ESP32 go to the Espressif site at https://docs.espressif.com/projects/esp-idf/en/v4.4.3/esp32/api-reference/peripherals/i2s.html

The DMA description above was gleaned from many fragmented, incomplete sources. My hope is that it is mostly correct in spite of some fuzziness around the edges. For example, it is not clear to me when exactly the CPU/DMA interrupts occur and exactly what the interrupt routines do.

The Effect Programs

gainDoubler 16-bit

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 Buffer Rx 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. There are reports of some restriction when using **FreeRTOS** but only the **Core 0** processor is using **RTOS**, not **Core 1** in this main loop.

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 Tx 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 Tx buffers go empty from the DMA constantly feeding the output DAC; and the **i2s_read( )** function must happen before the Rx buffers are completely filled from the DMA constantly feeding them input ADC samples. An overflowing Rx buffer or an empty Tx 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.

gainDoubler 24-bits

The above effect example uses 16-bit samples at a sample rate of 44,100KHz. This is the standard CD quality format. For a higher quality formats 24-bit sampling is often used. 24-bit samples can be accommodated in the software with one caveat, c++ has no **int24_t** integer type. Here is how you get around this problem.

1. The **rxbuf[ ]** and **txbuf[ ]** must be set to **int32_t** (4 byte samples) in the files **main.cpp** and **set_settings.h**.

2. In **set_settings.h**, set **BITS_PER_SAMPLE (24)**. The function **codec.setBitsPerSample( )** uses **BITS_PER_SAMPLE** to automatically send the codec the command `write_reg(Codec_ADCCONTROL4, 0X20).`

3. Set up **i2s_read** and **i2s_write** to handle 4 byte samples with the input parameter **FRAMELENGTH*4**. This will result in an extra zeroed byte in the least significant byte position.

4. After **i2s_read**, shift out this zero lower byte to convert to a 24-bit sample. Before **i2s_write**, shift 8 zero bits back into the lower byte to convert back to a 32-bit sample.

Here then is the **main.cpp while(1)** loop code for a 24-bit gainDoubler effect:

```
while(1)
{
  i2s_read(I2S_NUM_0, rxbuf, FRAMELENGTH*4, &readsize, 20);

  for (int i=0; i<(FRAMELENGTH); i+=2)
   {
    rxl = (float) (rxbuf[i] >> 8) ;
    rxr = (float) (rxbuf[i+1] >> 8) ;

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

    txbuf[i]   = ((int32_t) txl) << 8 ;
    txbuf[i+1] = ((int32_t) txr) << 8 ;
   }

  i2s_write(I2S_NUM_0, txbuf, FRAMELENGTH*4, &readsize, 20);

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

This simple effect gainDoubler was used to better reveal the basic components required for any effect program. Physical controllers are configured in the **set_module** files, and the actual effect code is found in **main.cpp** at the center of the **while(1)** loop. The code before and after the effect processing code, code that includes **i2s_read**, **i2s_write**, **txbuf[ ]** and **rxbuf[ ]**, can remain basically the same no matter what effect is coded.

Note that amplitude control doesn't have to happen at the sample level. An easier method of gain control would be to use the codec function **codec.setOutputVolume(gain)** within **onControlChange( ).** Make "**gain**" vary between 0 and 33 with `control[0].levelCount = 33;` Both settings are made within **set_module.cpp**.

BSDSP DSP Files

More involved effects can take advantage of the three **Blackstomp** Digital Signal Processing files -- **bsdsp.h**, **bsdsp.cpp**, and **dsptable.h**.

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]).

Stereo Chorus 24-bit

What follows is an example of a "Stereo Chorus" effect that uses several of the **bsdsp** file's DSP effect classes. To start off, here is the **set_module.h** file:

```
#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.

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

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

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

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

```
  oscillator lfo1;
  oscillator lfo2;

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

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

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

  //setting up the buttons
  button[0].mode = BM_MOMENTARY;
  button[0].pin = KEY1;
  button[2].mode = BM_MOMENTARY;
  button[2].pin = KEY2;

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

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

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

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

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


  freq=5;
  depth=0.5;
  beatFrequency=2.5;
  stereo = 1;
  asynch = 1;
```

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

```
          stereo = (bool)control[3].value;
          break;
        }
        case 4:  //depth
        {
          asynch = (bool)control[4].value;
          break;
        }
   }
  }
```

The first thing accomplished in the **set_module.cpp** file is the creation of instances for all the Classes used in the Effect. A pointer to **myPedal** is created, the main instance object of the **controller_module** child class. **delay1** and **delay2** are instance objects of the **fractionalDelay** DSP class. **lfo1** and **lfo2** (low frequency oscillators) are instance objects of the **oscillator** DSP class. Both delay instances are initialized with **delay1.init(3)** and **delay2.init(3)**. This will create buffers that hold 3 milliseconds of samples given the defined **SAMPLE_RATE**. The number of samples in buffer = (samples per second) * (0.003 seconds)). These **init( )** methods return boolean **true** if the buffer build was successful.

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

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

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

The only Effect code left to discuss now is found in the **while( )** loop within **main.cpp**:

```
while(1)
{
  i2s_read(I2S_NUM_0, rxbuf, FRAMELENGTH*4, &readsize, 20);
```

```
        for (int i=0; i<(FRAMELENGTH); i+=2)
        {
          rxl = (float) (rxbuf[i] >> 8) ;
          rxr = (float) (rxbuf[i+1] >> 8) ;

          //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
          //~~~~~~~~~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]   = ((int32_t) txl) << 8 ;
          txbuf[i+1] = ((int32_t) txr) << 8 ;
        }

        i2s_write(I2S_NUM_0, txbuf, FRAMELENGTH*4, &readsize, 20);

      }
```

The code framing the middle stereoChorus Processing should be familiar. It is exactly the same code used in the 24-bit sampling gainDoubler. For 16-bit sampling copy the code found in the 16-bit gainDoubler. Just be sure to set **BITS_PER_SAMPLE** = 16, and set up the **rxbuf** and **txbuf** with **int16_t** elements in both places where they are declared and defined.

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

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**".

Stereo Chorus with System Monitor

When working with more involved signal processing it is sometimes useful to check on the CPU load using the System Monitor. CPU load is tracked using the group of "tick" variables declared in **set_settings.h** and defined in **task.cpp**. These must be carefully arranged around the main loop in **main.cpp**. An example of this applied to the 24-bit sampled stereoChorus Effect is shown below.

```
void loop()
{
  //leave the main loop dedicated only to the I2S audio task

  size_t readsize = 0
  int32_t rxbuf[FRAMELENGTH], txbuf[FRAMELENGTH];
  float rxl, rxr, txl, txr;

  myPedal->init();
  taskSetup();

  //################ get ticks ################################
  usedticks_start = xthal_get_ccount();
  availableticks_start = xthal_get_ccount();

  while(1){   //signal processing loop

  //############### initialize ticks for sysmon ###############
  availableticks_end = xthal_get_ccount();
  availableticks = availableticks_end - availableticks_start;
  availableticks_start = availableticks_end;

  i2s_read(I2S_NUM_0, rxbuf, FRAMELENGTH*4, &readsize, 20);

  //########### used-tick counter starting point ##############
  usedticks_start = xthal_get_ccount();

  for (int i=0; i<(FRAMELENGTH); i+=2) {

    rxl = (float) (rxbuf[i] >> 8) ;
    rxr = (float) (rxbuf[i+1] >> 8) ;

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    //~~~~~~~~~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]   = ((int32_t) txl) << 8 ;
        txbuf[i+1] = ((int32_t) txr) << 8 ;
      }

      //############### used-tick counter end point ####################
      usedticks_end = xthal_get_ccount();
      usedticks = usedticks_end - usedticks_start;
      processedframe++;

      i2s_write(I2S_NUM_0, txbuf, FRAMELENGTH*4, &readsize, 20);

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

Be sure to uncomment `runSystemMonitor( );` in the **setup( )** section of **main.cpp**. Surprisingly enough the System Monitor does not seem to affect the audio output stream. The **sysmon_task( )** is placed in **Core 1** along with the **controltask( )** and **buttontask( )** and is given the lowest priority, 0 or idle. It seems to update only around once per second.

Summary

To complete this description of the Codec Software Package for the ESP32 here is a quick review of its files :

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 startup 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.