ESP32 DevKit
PCB Artists Codec

John Talbert - March 2022

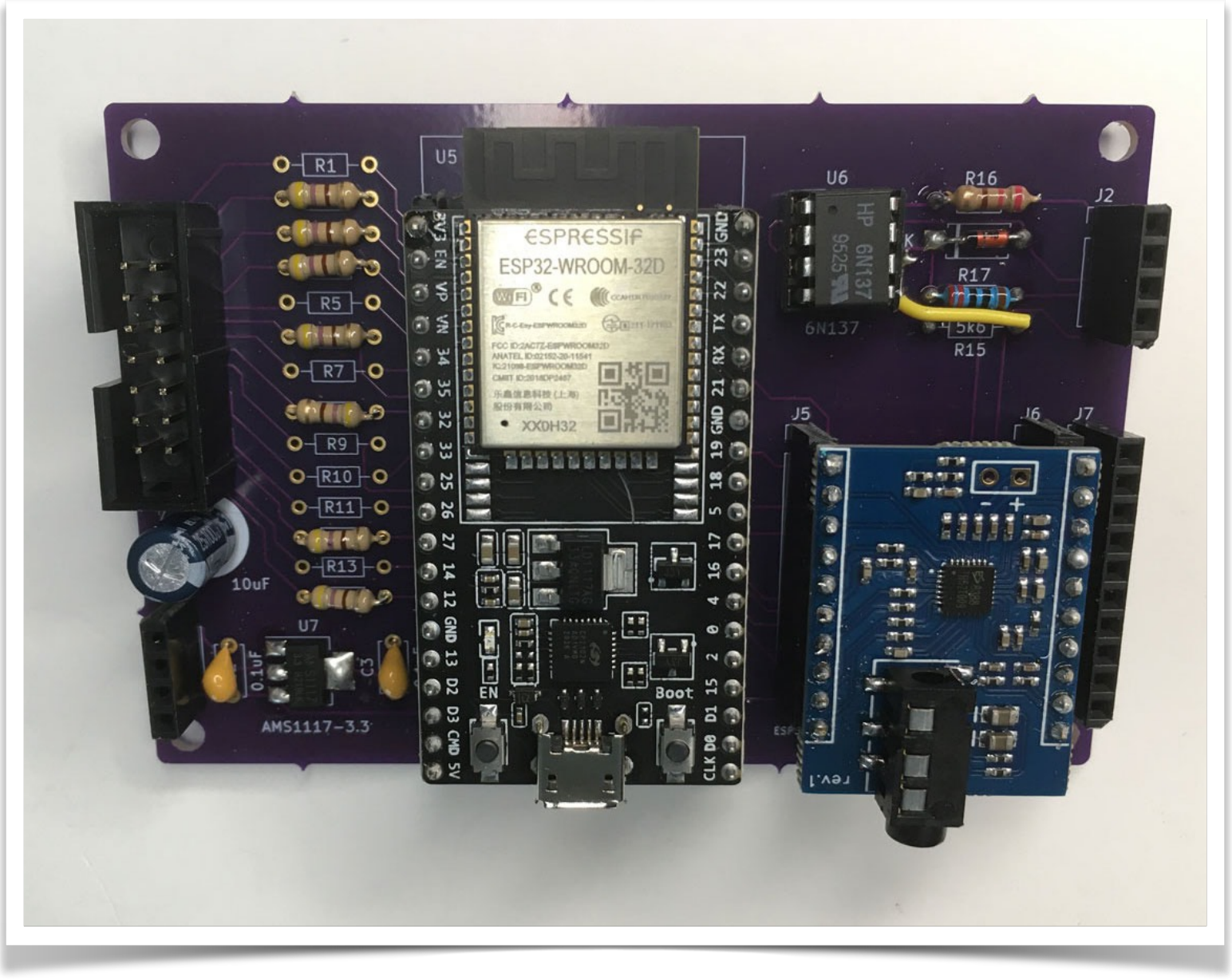


Table of Contents

Arduino Code

Platform IO Code

IDF / ADF Code

the ESP32

The ESP32 microprocessor is like a supercharged Arduino. It can be programmed with either the Espressif IDF or the Arduino IDE. Its expanded capabilities includes wireless WiFi, Bluetooth. It has a clock speed of 80MHz/240MHz compared with 48MHz for the Arduino MKR. Flash memory for user programs is 4MB/8MB compared to 0.256MB for the MKR. The SRAM memory for user variables is 520KB compared to 32KB for the MKR. The ESP32 processor is 32-bit, dual-core, enabling it to run two program threads simultaneously. ESP32 peripherals include up to 43 GPIO pins, 1 full-speed USB OTG interface, SPI, I2S, UART, I2C, LED PWM, LCD interface, camera interface, ADC, DAC, touch sensors.

Given its expanded capabilities, programming the ESP32 can be difficult, even within Arduino's familiar IDE environment. Here are several sources for tutorials and books on the ESP32.

Espressif https://www.espressif.com/en/products/modules ESP32 Developer

Tech Explorations https://techexplorations.com/pc/esp32/ Video Tutorials

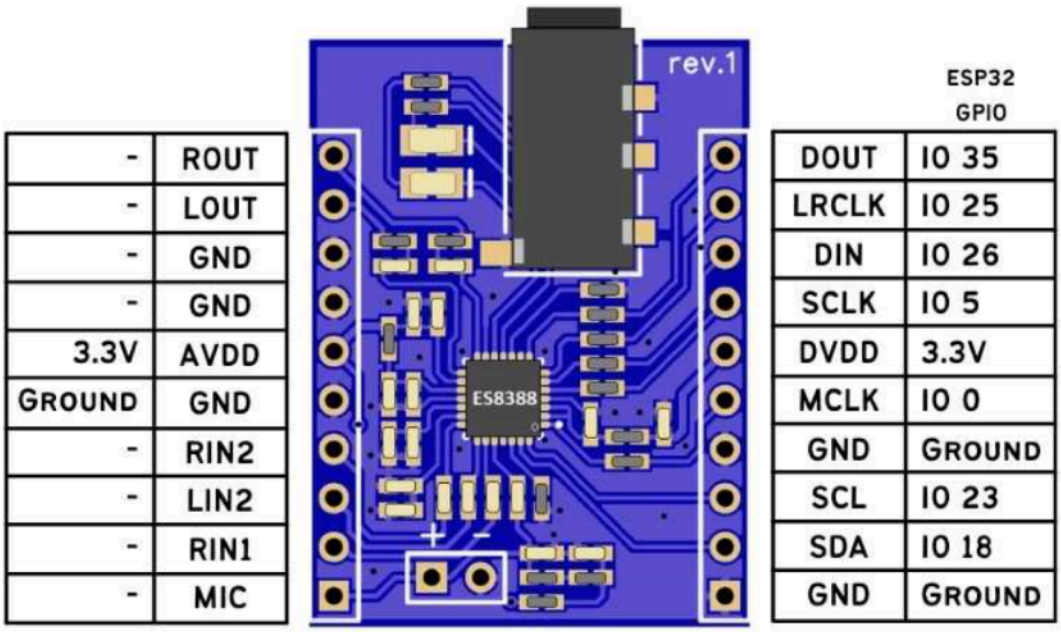
Random Nerd https://randomnerdtutorials.com/projects-esp32/ Tutorials, Books

Books https://bookauthority.org/books/best-esp32-books

the PCB Artists Codec

This is a breakout board for the ES8388 Audio Codec made by the electronic design company PCB Artists (pcbartists.com).

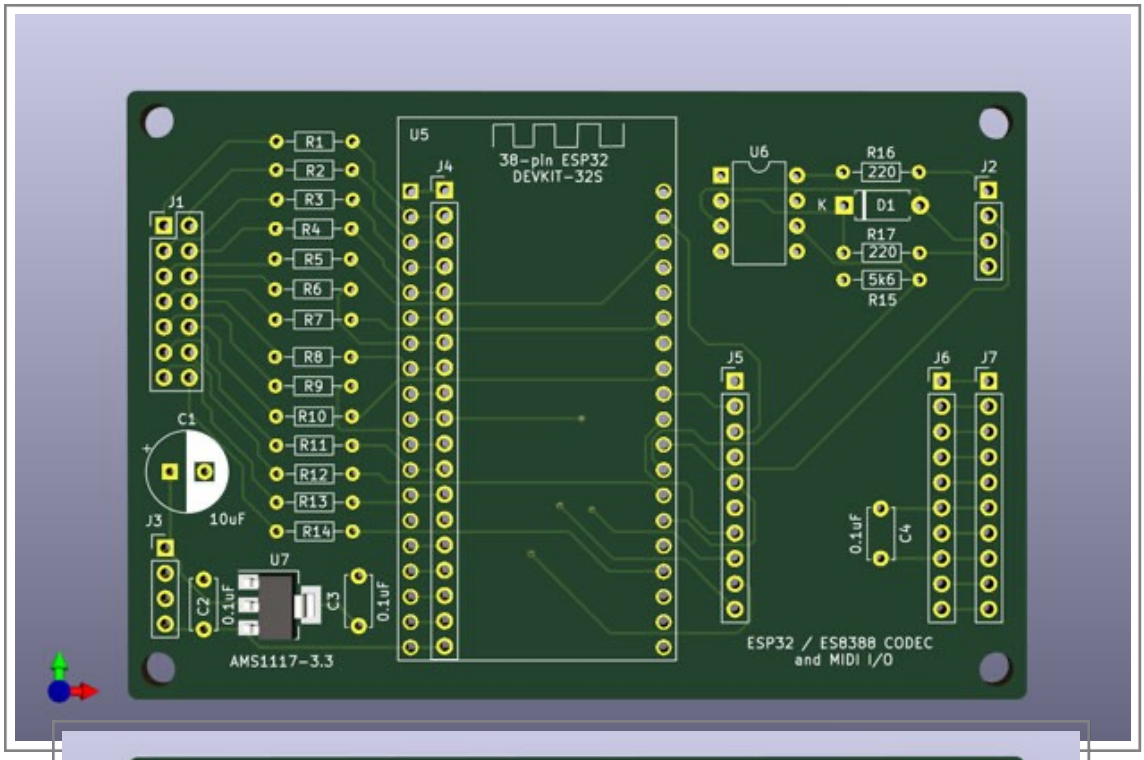
ESP32 ES8388 audio codec interfacing is a simple task. The PCB Artists ES8388 audio module or breakout board makes this task extremely easy. Just like any other audio codec that you would interface with the ESP32, the ES8388 has a basic I2C control port and an I2S audio port for writing and reading audio data. Other than these interfaces, there is an additional pin that accepts system clock from the ESP32 for all ES8288 internal operations. (Please note that in the image below DIN and DOUT refer to the Codec pins, not the ESP32 connections, which would be opposite).

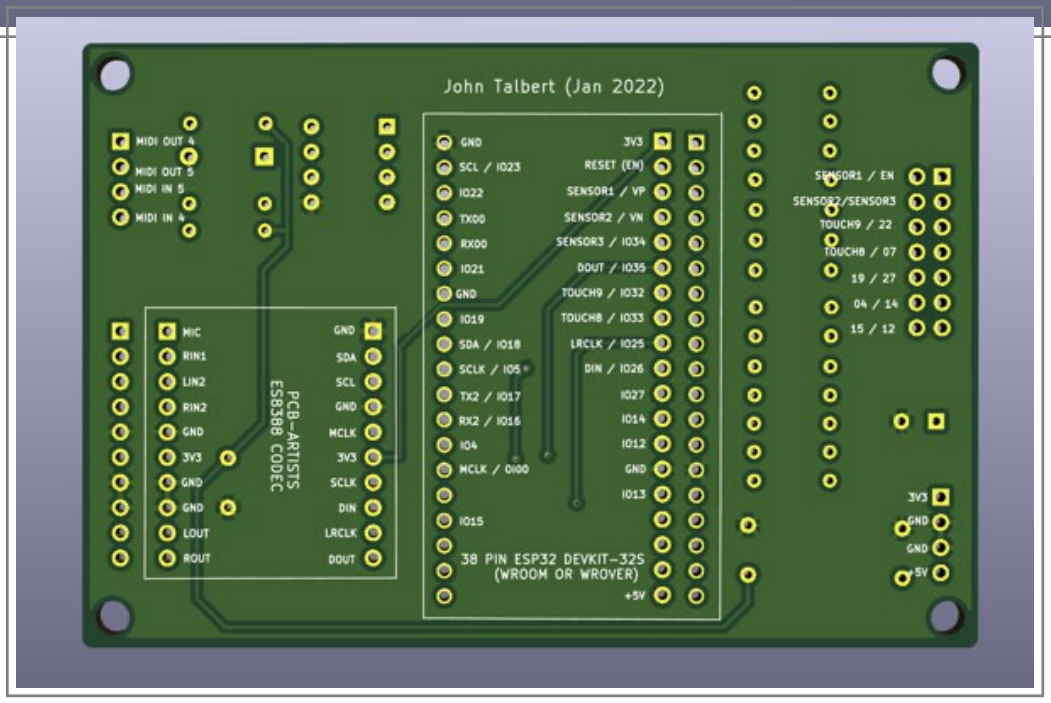


ESP32 ES8388 Audio Codec Module Connections

the Board

A PCB board, connecting the ESP32_DevKit and the PCB Artists Codec Module, was designed using the free, open source, KiCad app (www.kicad.org) and Dr. Peter Dalmaris' very useful book and tutorial "KiCad Like A Pro" (https://techexplorations.com).



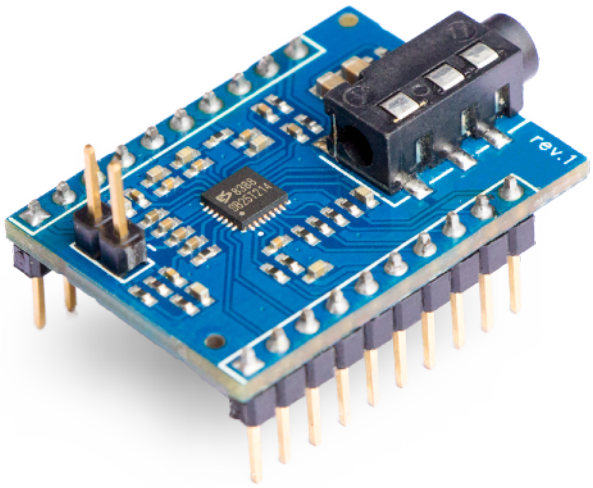


The central device on the PCB is the ESP32-DEVKITC-32D. This is a 39 pin module with a USB connection used to both program and power the ESP32 microprocessor. There are several versions of this board. The PCB board has an extra 19 pin header to accommodate two different sizes of the DEVKIT with pin widths of 22mm or 25mm.

A USB connector is mounted on one end of the ESP32-DevKit. This is used both to program the ESP32 and to provide power to the entire PCB board. With a USB cable connecting the ESP32 to an external computer, the Arduino IDE application can be used to program the board. Alternatively, the Espressif IDF application running from Visual Studio Code can also be used for program development. Both of these methods will be demonstrated later.

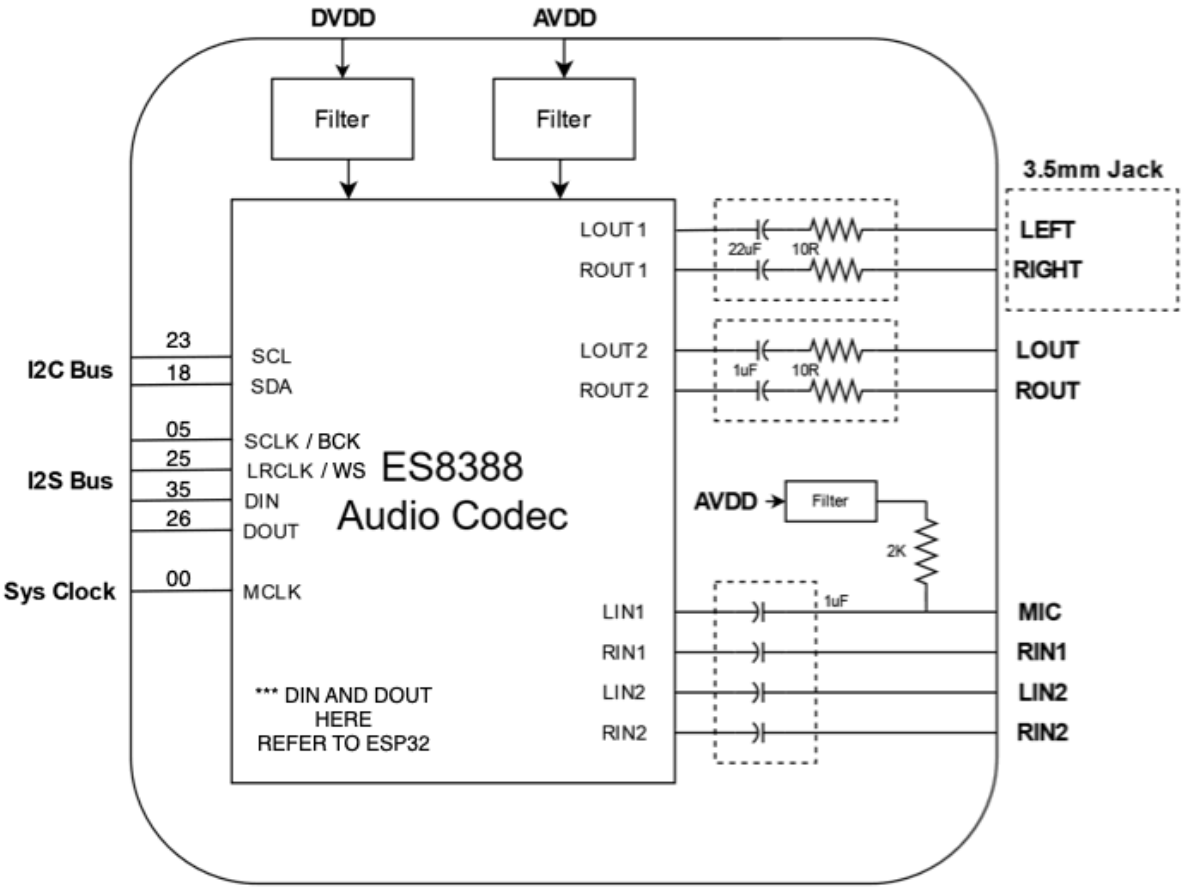
A MIDI Interface is provided on the PCB. It uses pins TX2 (IO17) and RX2 (IO16) for MIDI Input and MIDI Output. A 6N137 opto-isolator chip is used in the MIDI IN circuit. A 4-pin header provides connections for the standard 5-pin DIN MIDI Output and MIDI Input jacks. Note that the ESP32 WROVER version does not allow serial interface TX2 and RX2 functions on pins 16 and 17. Be sure to select the ESP32 WROOM DevKit if MIDI I/O is desired.

Two 10-pin headers are provided to mount the PCB Artists ES8388 Codec Module.



One side has all the digital connections to the ESP32 including the I2C Bus, the I2S Bus, and the System Clock. The other side has all the analog Audio connections, duplicated on a second 10-pin header. These include 2 audio outputs, 4 audio inputs, Circuit Ground, and 3.3volts. Two additional outputs are connected to a stereo headphone jack mounted on the Codec board.

The audio side of the Codec Module has its own 3.3 volt power line labeled AVDD which is separate from the digital signal side labeled DVDD. The PCB board includes a AMS1117 voltage regulator which derives the 3.3v AVDD voltage from a 5 volt ESP32 pin. In this way, hopefully, digital noise is kept out of the Audio lines. A 4-pin header on the PCB edge carries 5v, GND, GND, and the 3.3v AVDD for powering external devices such as controller pots, switches and LEDs.

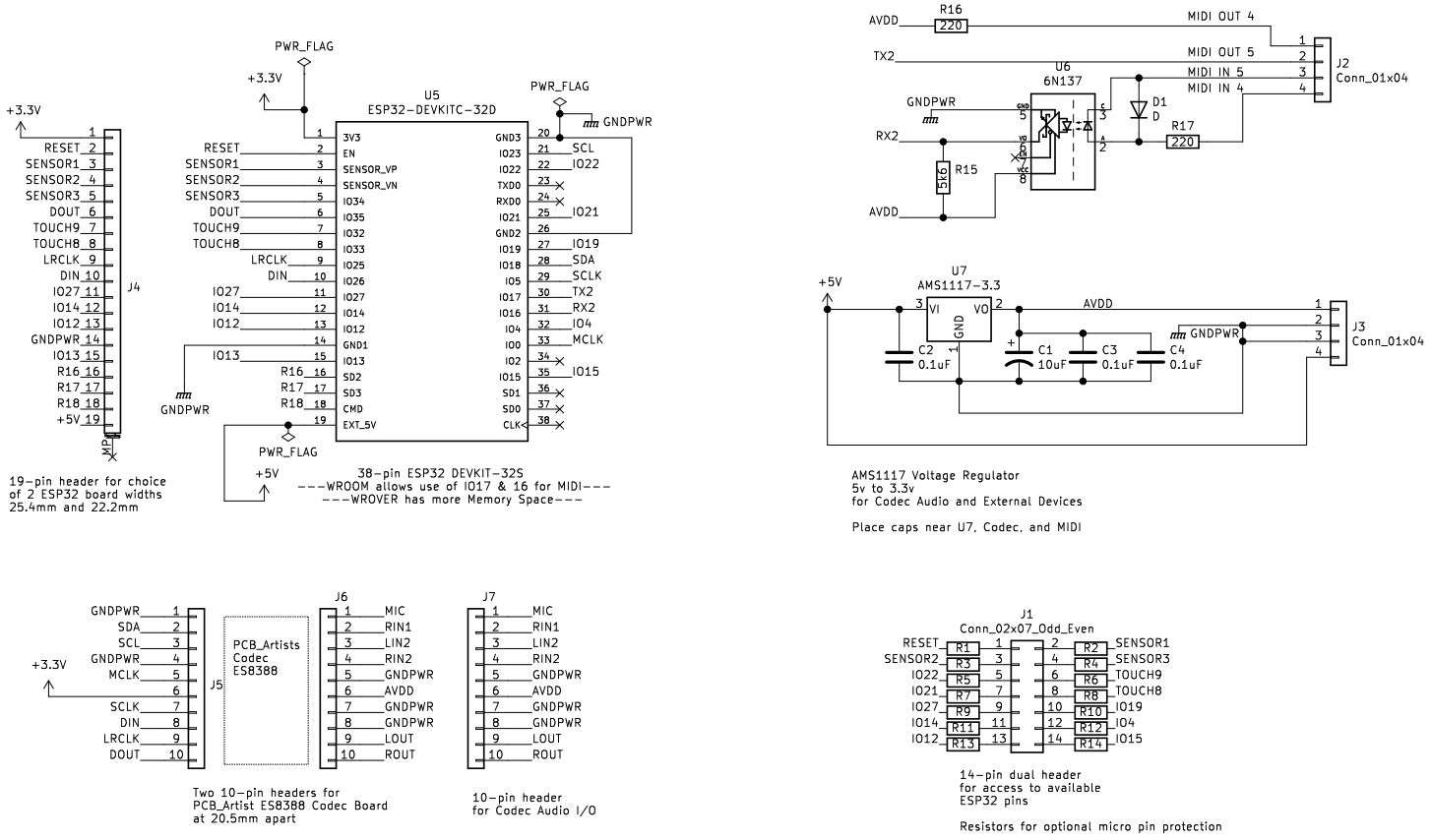


Once the PCB Artists Codec Module and the MIDI Interface have been connected to the ESP32, most of the left over, unused ESP32 pins are available for other uses such as potentiometer controls, sensor controls, LED lights, switches and even display modules.

These available pins are brought out to a 14-pin header connector. Connections to the 14 header pins are made via a convenient column of 14 resistor pads labeled R1 through R14.

R1	RESET / EN
R2	IO36 / VP (Input only)
R3	IO39 / VN (Input only)
R4	IO34
R5	IO22 / Display I2C SCL
R6	IO32 / TOUCH9
R7	IO21 / Display I2C SDA
R8	IO33 / TOUCH8
R9	IO27 / TOUCH7
R10	IO19
R11	IO14 / TOUCH6
R12	IO04 / TOUCH0
R13	IO12 / TOUCH5
R14	IO15 / TOUCH3

It is advised not to use GPIO pins 6, 7, 8, 9, 10 and 11 since they are employed by the flash memory. Even some of the pins made available above may have other functions at load, startup and reset. For example, pin IO12 connections can inhibit program loads if it is not connected in some way to Ground during program loading.



design by John Talbert

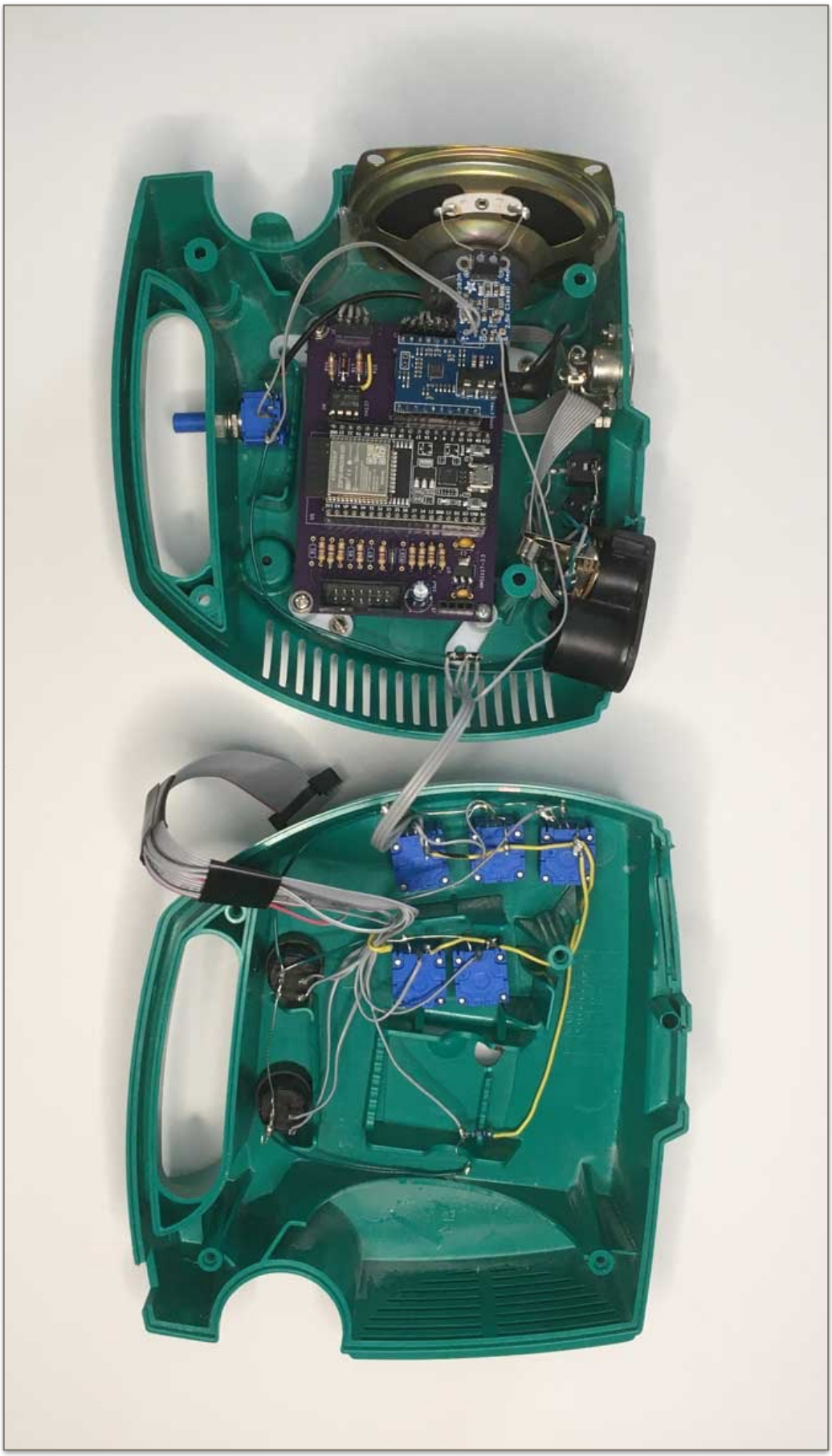
Sheet: /
File: ESP32_Codec.sch

Title: PCB_Artist Codec connected to ESP32 Dev Kit w/MIDI

Size: A4	Date: 2022-01-07	**Rev: v1**
KiCad E.D.A. kicad (5.1.10-1-10_14)		Id: 1/1

the Box

The PCB Codec board was mounted in the box shown above along with a number of controller devices and audio jacks connected by ribbon cable from the various board edge connectors.




Speaker Volume
IO04 LED
IO15 LED
IO32 Switch
IO33 Switch
IO36 Pot
IO12 Pot
IO39 Pot
IO14 Pot
IO34 Pot
Go Green
IO27 Light Sensor


LIN1 MIC
USB / HEADPHONE ACCESS
MIDI INPUT
MIDI OUTPUT
LOUT2
ROUT2
LIN2
RIN2

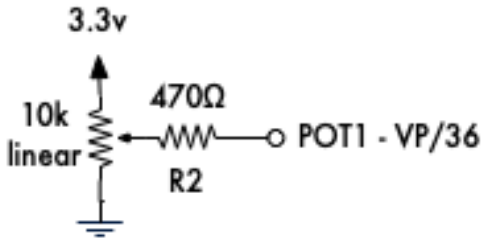
Controller devices can easily be added with a minimum of circuitry.

Rotary or slide potentiometers are simply wired between Ground and 3.3 volts. The wiper picks off a variable voltage between those two extremes and is connected to an ESP32 pin programed as an ADC input pin. The pots can be any value but need to have a linear taper. A small 470 ohm protection resistor is recommended to prevent short circuiting the ESP32 input pin in case the pin is accidentally defined as an output instead. One of the 14 "R" pads on the PCB is a convenient place to put this protection resistor, connecting the selected ESP32 pin to the board edge connector.

Pushbutton switches are easy. One side is connected to ground and the other side is connected to an ESP32 pin programed in software as a digital input with an internal pull-up resistor. Again, a 470 ohm protection resistor is recommended in case the pin is accidentally defined as an output instead. As with the pot, this resistor can be placed on the PCB in whatever "R" pad connects to the selected pin.

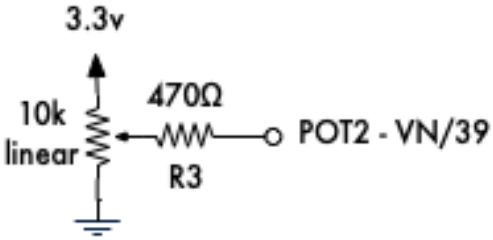
LEDs are easily set up with their cathode side connected to ground and the anode side connected through a current limiting resistor to an ESP32 pin defined as a digital output. A 470 ohm resistor results in an acceptable LED brightness and can also be placed in one of the PCB's "R" pads.

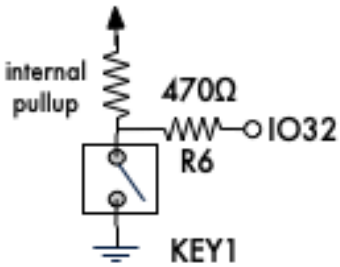
The Cadmium Cell light sensor is connected in series with another resistor with a value between the light sensor's maximum and minimum resistance values. The ESP32 pin is connected to the mid point between these two resistors and defined as an ADC input. No protection resistor is needed, so the selected pin's "R" pad is simply jumped with a wire.

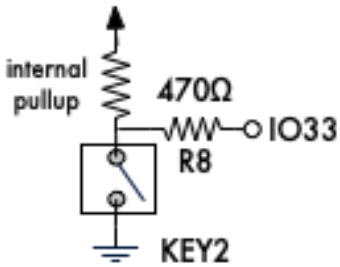


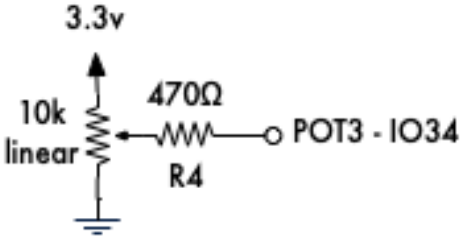
ESP32 / ES8388
Sensor — Controller Circuits

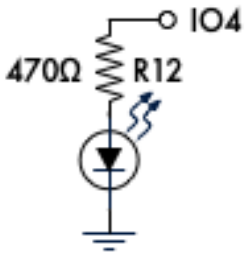
John Talbert 2022

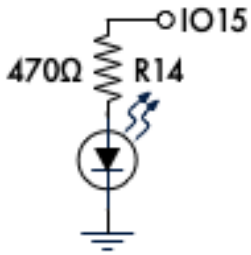


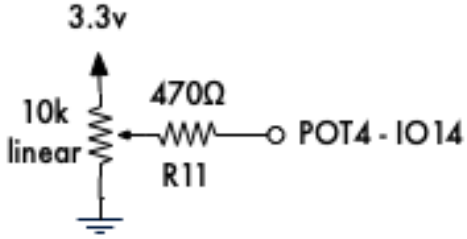




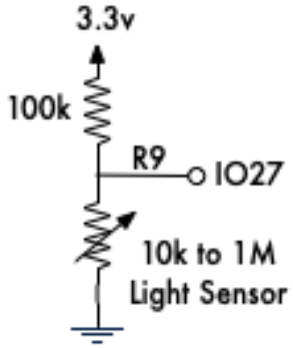




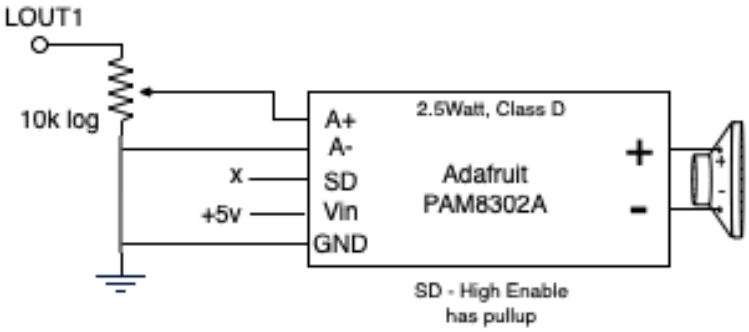


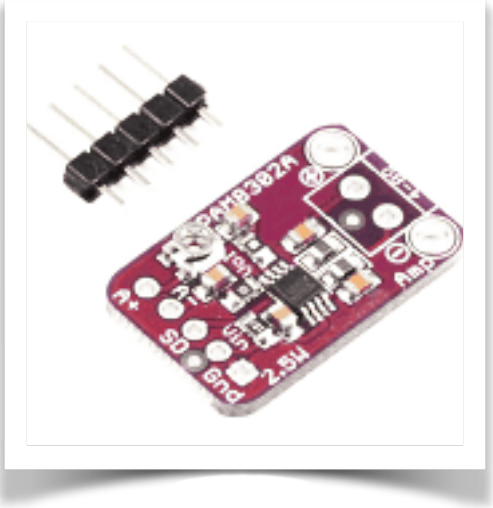


Warning: This Pot must be turn down during program LOADs



One side of the box featured a circular open grill (originally housing a fan) which, in this application, is perfect for a 3 inch Speaker. To accommodate this speaker, the left channel of the Headphone Jack mounted on the PCB Artists Codec Module was wired to a 10k log tapered volume pot and then on to a small 2.5 Watt Mono Amplifier circuit board by Adafruit Electronics (PAM8302). The circuit is illustrated below.



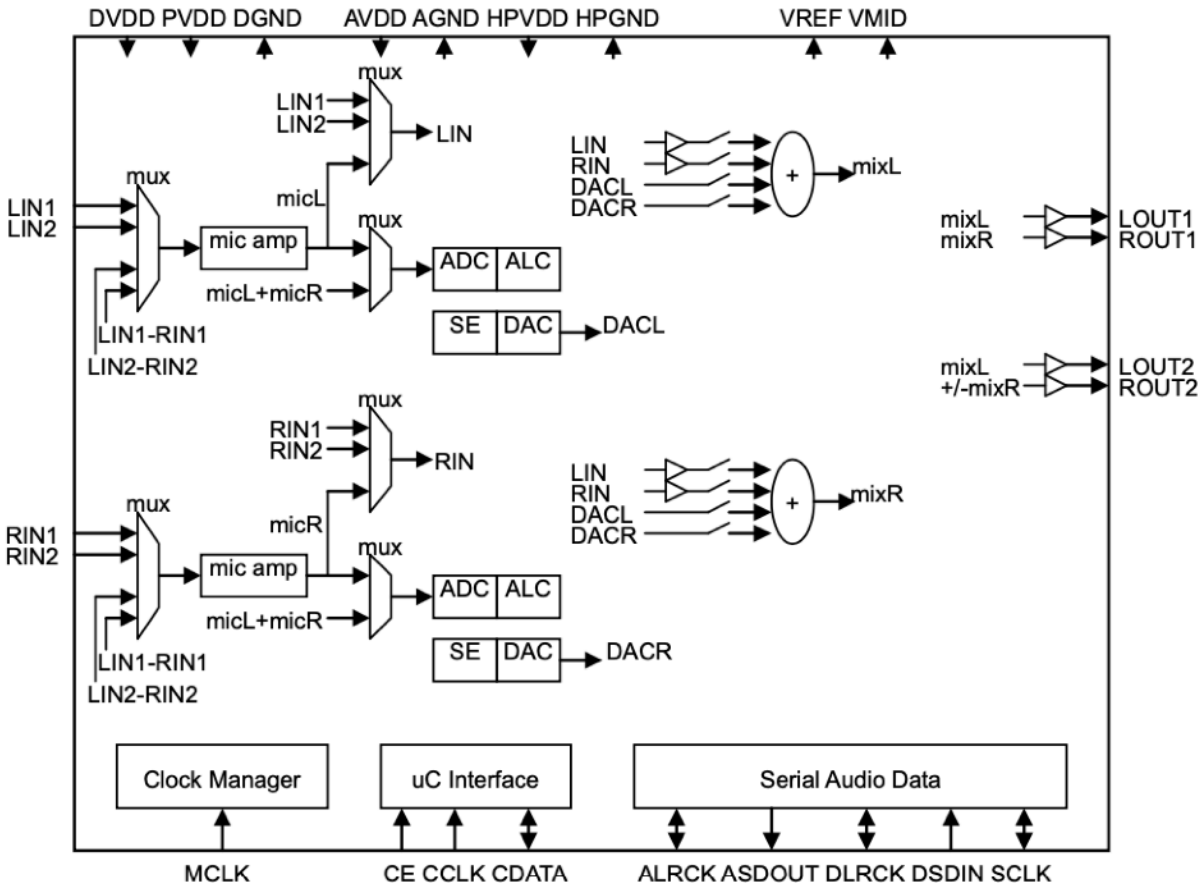


the ES8388 Codec

The ES8388 is the codec used in the PCB Artists Codec Module. It is a high performance, low power and low cost audio CODEC by Everest Semiconductors. It consists of 2-ch ADC, 2-ch DAC, microphone amplifier, headphone amplifier, digital sound effects, and analog mixing and gain functions.

Data Sheet:
http://www.everest-semi.com/pdf/ES8388%20DS.pdf

User Guide
https://dl.radxa.com/rock2/docs/hw/ds/ES8388%20user%20Guide.pdf



ADC

24-bit, 8 kHz to 96 kHz sampling frequency
95 dB dynamic range, 95 dB signal to noise ratio, -85 dB THD+N
Stereo or mono microphone interface with microphone amplifier
Auto level control and noise gate
2-to-1 analog input selection
Various analog input mixing and gains

DAC

24-bit, 8 kHz to 96 kHz sampling frequency
96 dB dynamic range, 96 dB signal to noise ratio, -83 dB THD+N
40 mW headphone amplifier, pop noise free
Headphone capless mode
Stereo enhancement
Bass and Treble
Various analog output mixing and gains

Low Power

1.8V to 3.3V operation
7 mW playback; 16 mW playback and record

System

I^2C or SPI uC interface
256Fs, 384Fs, USB 12 MHz or 24 MHz
Master or slave serial port
I^2S, Left Justified, DSP/PCM Mode

The Codec uses two interface protocols. The I2C interface is used to configure the chip, and the I2S is used to move the audio data.

The I2S interface loads and reads 53 user programmable 8-bit registers that set up I/O connections (see block diagram above), sampling rate, sample format, sample size, volume, filters, effects, etc. Only a few of these registers are of interest to the user. Most of them can be set and left in their default settings.

The I2S interface is used to move audio data between the ESP32 Microprocessor and the Codec, out of the Analog to Digital Converters and into the Digital to Analog Converters.

the Programming

Espressif Systems is the company that created and developed the ESP32 Microprocessor. They also provide several programming resources for the ESP32.

ESP-IDF IoT Development Framework

https://docs.espressif.com/projects/esp-idf/en/latest/esp32/
https://github.com/espressif/esp-idf

ESP-ADF Audio Development Framework

https://docs.espressif.com/projects/esp-adf/en/latest/
https://github.com/espressif/esp-adf

Arduino ESP Core

https://docs.espressif.com/projects/arduino-esp32/en/latest/
https://github.com/espressif/arduino-esp32

ESP-IDF is **Espressif's official IoT Development Framework for the ESP32, ESP32-S and ESP32-C series of SoCs**. It provides a self-sufficient SDK for any generic application development on these platforms, using programming languages such as C and C++.

ESP-ADF is an extension to IDF for audio applications. It was developed expressively for Espressif's audio development board, the LyraT.

https://docs.espressif.com/projects/esp-adf/en/latest/design-guide/dev-boards/get-started-esp32-lyrat.htm

The PCB Artists ES8388 breakout board is compatible with most examples within the ESP-ADF system. This is because the ESP-ADF contains a board support package for the ESP32-LyraT audio board, which is also based on the ES8388 audio codec chip. The PCB Artists web page describes ESP-ADF applications made for the LyraT that will also work with our board. The applications include a Bluetooth Speaker and an MP3 playback example.

https://pcbartists.com/products/es8388-module/esp32-es8388-audio-codec-interfacing/

The main advantage our board has over the LyraT is a MIDI interface and a header/resistor pad scheme that makes 13 pins readily available for external controller circuits. The LyraT has full stereo audio input and output capabilities along with an SD card interface, however, the only pins made available for external use are GPIO12, 13, 14, and 15 on a JTAG header.

The Arduino ESP Core is an add-on Library from Espressif for the Arduino IDE that allows you to program the ESP32 using the popular Arduino IDE and its programming language.

https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-windows-instructions/

The Arduino ESP Core includes many useful functions for integrating LEDs, PWM, Interrupts, Timers, Dual Core, EEProm, and so on into your applications. Many of these programmed features, now easily available on the ESP32, were only accessible on an Arduino board by hacking into its internal registers. It does not, however, include any of the ADF audio features or a driver for the ES8388 Codec. These can be added from other Github sources as covered in a later section.

the Espressif IDF/ADF

Espressif IDF is the official professional framework for programming the ESP32. Combined with ADF it offers a complete software development kit (SDK) for the ES8388 Codec attached to an ESP32 DevKit.

IDF text commands are entered from a Terminal window, an environment familiar to anyone who has worked with Unix. The actual programs are then written with any text editor. Instructions for installing IDF can be found here:

https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html#what-you-need

This IDF command terminal environment can be difficult for beginner and even intermediate programmers. After receiving several customer requests for an integrated solution to writing, managing and deploying ESP-IDF projects, Espressif started officially supporting not only one, but two such IDEs, Eclipse and Microsoft's Visual Studio (VS) Code. Here we will use Microsoft's free application Visual Studio Code.

VSC is basically a Code Editor but it can also be classified as an IDE (Integrated Development Environment). Like the Arduino IDE, you can both write and test your code from the same environment. Code is entered in a central editor window and then tested with Compile, Run, and Monitor buttons.

For larger programs, Visual Studio Code is a great aid. It supports multiple programming languages (c, cpp, python) and can work on Windows, Mac or Linux platforms. It error checks your code as you enter it and offers code completion pop-up suggestions. If your program involves multiple files and libraries, VSC will organize all of them in an easily accessible directory structure and then allow you to access any of them from editor window tabs. It also has a built in Terminal window and built in GIT resources.

Recent versions of Visual Studio Code allow the installation of IDF and ADF from within VSC, however, IDF requires several other utility tools that may need to be installed manually such as Python3, Git, CMake and Ninja.

What follows are step by step instructions for installing IDF and the required tools in Visual Studio Code. VSC will also provide its own instructions at each step.

IDF Install

1. Read through " https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/vscode-setup.html "

2. Watch the video at " https://github.com/espressif/vscode-esp-idf-extension " for an excellent installation guide that covers many of the steps below.

3. Install Microsoft's Visual Studio Code using the Download link at " https://code.visualstudio.com ".

4. Check that you have Python 3.5 or later installed on your computer by entering " python3 —version " in a Terminal. If not there, go to "https://www.python.org" for the installation download.

5. Install Git on your computer following the directions from " https://docs.github.com/en/get-started/quickstart/set-up-git ". You don't need the full blown GitHub membership.

6. Open Visual Studio Code. Click on the "Extension" button. Type in "esp-idf" and choose the Espressif IDF that shows up below. Click on the "Install" button. Install will take several minutes.

7. Next step is to install some tools used by IDF. In the top Menus choose View/Command Palette… Type and/or choose "ESP-IDF: configure ESP-IDF extension". Click "Start". It may ask for the Python3 path (usually at /user/local/bin/. Choose the IDF version (usually the latest). If already installed, give the location (bottom of page).

8. Or, you may get "First install prerequisites for MAC". In this case, go to the website brew.sh and download the install for "Homebrew", a terminal utility for package installs. Use it in a terminal to install cmake, ninja and others — " brew install cmake ninja dfu-util ccache". Quit VSC and restart.

9. Repeat the menu View/Command Palette… " ESP-IDF: configure ESP-IDF extension ". Choose "Express", "Advanced", or "Use Existing" setup mode and follow the instructions to load the other needed tools.

10. These instruction steps can change with newer versions or different platforms. If you get the button "Go to ESP-IDF Tools setup" Click on that for an easy automatic load and check of the tools needed.

IDF Example Program

At this point it may be instructive to try out a simple test program with the ESP32_DevKit board.

1. In the VSC Menus choose again View/Command Palette… Type in " ESP-IDF: Show Examples Projects ". Choose the "Blink" program and read the description and instructions provided.

2. Click on "Create project using example blink" and choose a location for the project files.

3. The main program is inside the "main" folder shown on the "explorer" sidebar. In this folder are several other setup files. You will find that this "simple" IDF program is much more involved than a simple Arduino LED blink program.

4. Instead of setting up the LED GPIO pin number and blink on/off times within the main program, this application uses a Python configuration tool to set these values and many other possible variables.

5. From the bottom set of buttons choose [>] to open a Terminal. Enter the command " idf.py menuconfig "

6. This opens a "DOS" looking configuration tool. Use your arrow keys and RETURN key to navigate it. Find the "example configuration" page to set the GPIO pin your LED is on and the blink time period. Another, perhaps easier, configuration tool is available with the "sprocket" button at the bottom of the VSC page.

7. Exit the menuconfig tool. Tap the button BUILD to compile the blink program and check for errors. Tap the FLASH button to Load the program onto your ESP32 and Run it. Tap the MONITOR button to get a readout of LED ON and OFF as your LED blinks.

8. If you get the obscurely worded Error message "failed to flash because of some unusual error. ERROR #1" it is probably because it can't find the USB connection your ESP32 is on. Hit the bottom left menu button for "Select Port". If your ESP32 port is not on the list disconnect the USB and reconnect. During program load (flashing) you may need to press the BOOT button on your ESP32 board.

ADF Install

ESP-IDF includes all that is needed for programming your ESP32 board, however, it knows nothing about the ES8388 Codec. For this you will need to add ESP-ADF which extends the IDF package to include Audio support.

1. In the VSC Menus choose again View/Command Palette… Type in " ESP-IDF: Install ESP-ADF " and follow the directions presented.

2. If it complains that it can't find GIT you will need to help it with the next step.

3. Step 5 of the IDF install deals with GIT. Check where your GIT install lives. On the Mac it is usually at /usr/bin/git. Under the VSC Preferences choose Open Settings. Go to Extensions/ESP-IDF/GIT PATH. Set the path for your Git installation (/usr/bin/git).

4. After ADF installs you can again enter " ESP-IDF: Show Examples Projects " from the menu View/Command Palette… It will now offer you a choice between IDF and ADF example projects.

5. ADF projects for LyraT will work with the PCB_Artist CS8388 Codec Module. The Codec to ESP32 pin connections I used are the same as the LyraT connections.

6. Please note that the DIN and DOUT designations in the PCB_Artists spec sheets are given from the Codec view. The Codec pin assignments set in the Codec software driver library is always given from the ESP32 view, which is opposite the Codec view.

7. You may also want to add "ESP-ADF: Add Arduino ESP32 as ESP-IDF Component" from the View/Command Palette… menu. It allows you to use Arduino code and Libraries within IDF.

Arduino Programming from VSC

Visual Studio Code with the PlatformIO plugin is an ideal IDE for your Arduino framework programming. It has several advantages over the Arduino IDE for larger programming projects:

VSC/Platform IO Advantages

- The Editor has an Auto Complete feature that pops up code completion suggestions as you type your program code.
- The Editor has Code Identification. A pop up box with code source information is triggered on any highlighted code element.
- Extensive Syntax Color Highlighting, user configurable.
- A Program Debugger with detailed error descriptions and solutions. “IntelliSense” highlights potential errors in the code the moment you make them.
- Integrates Version Control with Git used to store and track program versions and revisions.
- A Library Manager that can search and load specific library versions into your code, saving them in the project directory.
- A Board Manager that identifies the type of microprocessor board used in each project.
- A Directory Manager that organizes all the files of a project into a pull down directory list. Any number of project files can be placed on tabs in the Editor window for easy access.
- Buttons for Compile, Flash/Load, Monitor, and an integrated Terminal.

VSC/PlatformIO Install

1. Read this tutorial from Random Nerd Tutorials— https://randomnerdtutorials.com/vs-code-platformio-ide-esp32-esp8266-arduino/

2. Watch this Video from DroneBotWorkshop — https://www.youtube.com/watch?v=JmvMvIphMnY

3. Install Microsoft's Visual Studio Code using the Download link at " https://code.visualstudio.com ".

4. Check that you have Python 3.5 or later installed on your computer by entering " python3 —version " in a Terminal. If not there, go to "https://www.python.org" for the installation download.

5. Open Visual Studio Code. Click on the "Extension" button. Type in "platformio" and choose the PlatformIO that shows up in the list below. Click on the "Install" button. Install will take several minutes.

6. PlatformIO can be started up by clicking on the "bug" icon on the left or the "Home" button at the bottom left.

7. Start a new Project by clicking the New Project button. Enter a Project Name. Find the microcontroller board you are using from a list of over 900. Select "Arduino" as the Framework.

PlatformIO Differences

Here is a list of several programming procedures in PlatformIO that are different from what you may be use to from the Arduino IDE.

- The Project Directory is to the left of the Editor. The main program is under the “src” (source) directory and is labeled main.cpp (for c++ programming language) — not main.ino
- The main.cpp file must have the line “ #include <Arduino.h> “
- Functions must be defined before “Setup” or at least declared there (.h format) for the compiler to find them.
- Libraries are installed for each project using the convenient “Library” button in the PlatformIO side menu. They will be updated automatically if desired. They can also be loaded by hand into the Project “library” folder.
- Additional program files (like .h .cpp files) can be added to the src folder but the main.cpp file must reference them with #include <> statements for the compiler to find them, unlike in the Arduino IDE where they are automatically compiled in alphabetical order.
- A special project file called platform.ini sets up all the project parameters such as the Name, Board, and Framework entered for “New Project”. It also lists all the Libraries under “lib_deps =” (library dependencies) with version specifications if you need a specific library version. You can set the Serial Monitor baud rate with “monitor_speed =“. And there are many other possible project settings all editable by the user.
- One recommended tutorial for making the transition from the Arduino IDE to VSC PlatformIO is “ESP32 Unleashed” by Peter Dalmaris at TechExplorations.com

Arduino Code

Load "Arduino ESP Core" into the Arduino IDE

https://docs.espressif.com/projects/arduino-esp32/en/latest/
https://github.com/espressif/arduino-esp32

Sensor Test

Test Code for the ESP32_Codec external controllers. Five pots, two pushbutton switches, two LEDs and one Cadmium Cell Light sensor are tested by continuously printing their values to the Monitor. Can act as a template for future code that uses the controllers.

```
/*
 * Use Tools/SerialMonitor to print all pot and switch values
 * NOTE: POT5 (IO12) must be set near zero for program LOAD to work
 */
// ~~~~~~~~~~~~~~~~~~ CONSTANTS/VARIALBES ~~~~~~~~~~~~~~~~

const byte LED1 = 4;
const byte LED2 = 15;
const byte POT1 = 34;
const byte POT2 = 39;
const byte POT3 = 36;
const byte POT4 = 14;
const byte POT5 = 12;  //must be set near zero for program load to work
const byte LIGHT = 27;
const byte KEY1 = 32;
const byte KEY2 = 33;

short pot1;
short pot2;
short pot3;
short pot4;
short pot5;
short light;

bool key1;
bool key2;
int x=0;
```

```
// ~~~~~~~~~~~~~~~~~~~ FUNCTIONS ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loadSensors(){      // load all current sensor values
     pot1 =   analogRead(POT1) ;
     pot2 =   analogRead(POT2) ;
     pot3 =   analogRead(POT3) ;
     pot4 =   analogRead(POT4) ;
     pot5 =   analogRead(POT5) ;
     light =  analogRead(LIGHT) ;

     key1 = digitalRead(KEY1);
     key2 = digitalRead(KEY2);
}

int switchCombo(){
   int result = !key2 + (!key1 * 2);

   switch (result) {
     case 0:
          digitalWrite(LED2, LOW);
          digitalWrite(LED1, LOW);
          break;
     case 1:
          digitalWrite(LED2, HIGH);
          digitalWrite(LED1, LOW);
          break;
     case 2:
          digitalWrite(LED2, LOW);
          digitalWrite(LED1, HIGH);
          break;
     case 3:
          digitalWrite(LED2, HIGH);
          digitalWrite(LED1, HIGH);
          break;
   }
   return result;
}
```

```
// ~~~~~~~~~~~~~~~~~~~~ SETUP ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

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

  pinMode(KEY1, INPUT_PULLUP);  //internal pullup
  pinMode(KEY2, INPUT_PULLUP);
  Serial.begin(115200);
}

// ~~~~~~~~~~~~~~~~~~~~ LOOP ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void loop() {

  loadSensors();
  x = switchCombo();

   Serial.print("pot1 = ");
  Serial.print(pot1);
    Serial.print("  pot2 = ");
  Serial.print(pot2);
    Serial.print("   pot3 = ");
  Serial.print(pot3);
    Serial.print("   pot4 = ");
  Serial.print(pot4);
    Serial.print("   pot5 = ");
  Serial.print(pot5);
    Serial.print("   light = ");
  Serial.print(light);

 Serial.print("     ");

   Serial.print("  switches ");
  Serial.print(key1);
  Serial.print(key2);
  Serial.print("   ");
  Serial.println(x);

     delay(500);          // wait for a half second
}
```

MIDI I/O Test

MIDI Input and Output can be set up on the ESP32 RX2 and TX2 pins with the following program lines:

```
#include <MIDI.h>
MIDI_CREATE_INSTANCE(HardwareSerial, Serial2, MIDI);
```

Use the Arduino MIDI Library. Specify HardwareSerial and Serial2 which the ESP32 package will understand to be the RX2 and TX2 pins.

```
/*
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
            MIDI INPUT and OUTPUT tested with a MIDI input CallBack Function
            On each NOTE On message received,
            2 extra arpeggiated notes will be played.

            pot 1 sets the playback speed.
            pot 2 sets the note spread
            Switch 1 plays random notes at speed set by pot 4.
*/
//On ESP32 WROOM MIDI set up on Serial2 --> RX2 and TX2 on pins 16 and 17.

  #include <MIDI.h>  // version 4.x.x
  MIDI_CREATE_INSTANCE(HardwareSerial, Serial2, MIDI);

// ~~~~~~~~~~~~~~~~~~  CONSTANTS/VARIALBES  ~~~~~~~~~~~~~~~~
//                              GIOP Pin Assignments
const byte LED1 = 4;
const byte LED2 = 15;
const byte POT1 = 34;
const byte POT2 = 39;
const byte POT3 = 36;
const byte POT4 = 14;
const byte POT5 = 12;  //must be set near zero for program load to work
const byte LIGHT = 27;
const byte KEY1 = 32;
const byte KEY2 = 33;
```

```
short pot1;
short pot2;
short pot3;
short pot4;
short pot5;
short light;
bool key1;
bool key2;

// ~~~~~~~~~~~~~~~~~~~  FUNCTION ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loadSensors(){      // load all current sensor values
    pot1 =   analogRead(POT1) ;
    pot2 =   analogRead(POT2) ;
    pot3 =   analogRead(POT3) ;
    pot4 =   analogRead(POT4) ;
    pot5 =   analogRead(POT5) ;
    light =  analogRead(LIGHT) ;

    key1 = digitalRead(KEY1);
    key2 = digitalRead(KEY2);
}

//~~~~~~~~~~~~~~~~ Callback MIDI_In Test function~~~~~~~~~~~~~~~~~~~~

 void myHandleNoteOn(byte channel, byte note, byte velocity){

  //Serial.println(" Saw a NoteOn ");

  int x = (pot1 >> 4) + 20;     //pot1 sets arpeggio speed

  int y = pot2;                 //pot2 sets arpeggio pitch range
  y = map(y, 0, 4095, 0, 20);

  delay(x);
  MIDI.sendNoteOn(note + y, velocity, 1);
  delay(x);
  MIDI.sendNoteOn(note + y + y, velocity, 1);
  delay(x);

  MIDI.sendNoteOn(note, 0, 1);
  MIDI.sendNoteOn(note + y, 0, 1);
  MIDI.sendNoteOn(note + y + y, 0, 1);
}
```

```
// ~~~~~~~~~~~~~~~~~~~~~ SETUP()~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup() {

delay(1000);

  MIDI.setHandleNoteOn(myHandleNoteOn); //for Callback MIDI In Test
  MIDI.begin(MIDI_CHANNEL_OMNI);

  // initialize Switches with pullup resistor
  pinMode(KEY1, INPUT_PULLUP);
  pinMode(KEY2, INPUT_PULLUP);

  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  digitalWrite(LED1, HIGH);

  Serial.begin(115200);

} //End of Setup

// ~~~~~~~~~~~~~~~~~~~~~ MAIN LOOP()~~~~~~~~~~~~~~~~~~~~~~~~

void loop() {

loadSensors();

// On MIDI.read() MIDI class will call Callback functions.
// User created callback function myHandleNoteOn() in section before setup()
// and MIDI.setHandleNoteOn(myHandleNoteOn) in setup() section

MIDI.read();

if(!key1){    // test MIDI Output with Switch 1 and Pot 4
  int note = random(30, 90);
  MIDI.sendNoteOn(note, 64, 1);
  delay((pot4 >> 4) + 10);
  MIDI.sendNoteOn(note, 0, 1);
}

} //End of Main Loop
```

Phil Schatzmann Audio Libraries

Phil Schatzmann has created a library of Arduino-Audio-Tools-Main (https://github.com/pschatzmann/arduino-audio-tools) described in this Wiki (https://github.com/pschatzmann/arduino-audio-tools/wiki), and a second library, Arduino-Audiokit-Main (https://github.com/pschatzmann/arduino-audiokit/) with drivers to support a number of Audio Codec boards including a generic board based on the ES8388 audio chip.

In order to set up the AudioKit library to work with our ESP32 PCB_Artist Board the GPIO pins for I2S and I2C must be edited in the library file arduino-audiokit-main/src/generic_es8388/board_def.h to have the following pin assignments:

```
// I2S
#define PIN_I2S_AUDIO_KIT_MCLK 0
#define PIN_I2S_AUDIO_KIT_BCK 5
#define PIN_I2S_AUDIO_KIT_WS 25
#define PIN_I2S_AUDIO_KIT_DATA_OUT 26
#define PIN_I2S_AUDIO_KIT_DATA_IN 35

// I2C
#define I2C_MASTER_NUM I2C_NUM_0 /*!< I2C port number for master dev */
#define I2C_MASTER_SCL_IO 23
#define I2C_MASTER_SDA_IO 18
#define I2C_MASTER_ADDR 0x10
```

And the file arduino-audiokit-main/src/AudioKitSettings.h must have the line:

```
#define AUDIOKIT_BOARD 7
```

The sound effects implemented in the following Arduino Scripts can be found in arduino-audio-tools-main/src/AudioEffects/

Phil Schatzmann In to Out

```
/**
 * @library author Phil Schatzmann
 * @copyright Copyright (c) 2021
 *
 * ES8388 ADC Input routed to DAC output with Volume and Mute controls
 * Demonstrating full use of the AudioKit Library, but not Audio Tools Library
 *
 * "kit" is the declared instance of the Object AudioKit
 * "cfg" is the method for configuring "kit"
 * "read" is the method for filling a buffer with ES8388 ADC data
 * "write" is the method for sending out data from a buffer to the ES8388 DAC
 * read and write only works with buffer size between around 512 and 4096.
 */

 // ~~~~~~~~~~~~~~~~~~~  AUDIOKIT_HAL  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "AudioKitHAL.h"
AudioKit kit;

// ~~~~~~~~~~~~~~~~~  CONSTANTS/VARIALBES  ~~~~~~~~~~~~~~~

const byte LED1 = 4;
const byte LED2 = 15;

const byte POT1 = 34;
const byte POT2 = 39;
const byte POT3 = 36;
const byte POT4 = 14;
const byte POT5 = 12;  //must be set near zero for program load to work
const byte LIGHT = 27;

const byte KEY1 = 32;
const byte KEY2 = 33;

short pot1;
short pot2;
short pot3;
short pot4;
short pot5;
short light;

bool key1;
bool key2;
int x=0;

const int BUFFER_SIZE = 1024;
uint8_t buffer[BUFFER_SIZE];
```

```
// ~~~~~~~~~~~~~~~~~~  FUNCTIONS  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void loadSensors(){      // load all current sensor values
      pot1 =   analogRead(POT1) ;
      pot2 =   analogRead(POT2) ;
      pot3 =   analogRead(POT3) ;
      pot4 =   analogRead(POT4) ;
      pot5 =   analogRead(POT5) ;
      light =  analogRead(LIGHT) ;

      key1 = digitalRead(KEY1);
      key2 = digitalRead(KEY2);
}

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

void setup() {
  Serial.begin(115200);
  delay(2000);

  Serial.println("STARTING");

  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  digitalWrite(LED1, HIGH);
  digitalWrite(LED2, HIGH);

  pinMode(KEY1, INPUT_PULLUP);  //internal pullup
  pinMode(KEY2, INPUT_PULLUP);

  auto cfg = kit.defaultConfig(AudioInputOutput);
  LOGLEVEL_AUDIOKIT = AudioKitInfo;
  // Debug (lowest level) / Info / Warning / Error   (all levels)

/*
 * DEFAULT
 * adc_input = Line1/Left (mic)
 * dac_output = Line2/Left (left jack)
 * volume = 18 (0 to 100)
 * sample_rate = 44,000 samples per second
 * bits_per_sample = 16,
 * codec_mode = BOTH
 */

  cfg.sd_active = false;
  cfg.adc_input = AUDIO_HAL_ADC_INPUT_LINE2; //LINE1 (mic) / LINE2 / ALL / DIFFERENCE
  cfg.dac_output = AUDIO_HAL_DAC_OUTPUT_ALL; //LINE1 (jacks) / LINE2 (speaker) /ALL
  cfg.sample_rate = AUDIO_HAL_44K_SAMPLES;  //08K/11K/16K/22K/24K/32K/44K/48K
  cfg.bits_per_sample = AUDIO_HAL_BIT_LENGTH_16BITS;  // 16/24/32BITS
  cfg.codec_mode = AUDIO_HAL_CODEC_MODE_BOTH; //DECODE(dac)/ENCODE(adc)/BOTH/LINE_IN
  cfg.fmt = AUDIO_HAL_I2S_NORMAL;  // NORMAL / LEFT / RIGHT / DSP(PCM)
```

```
  kit.begin(cfg);

  x = kit.volume();
  Serial.print("volume = ");
  Serial.println(x);

  x = cfg.sampleRate();
  Serial.print("Sample Rate = ");
  Serial.println(x);

  x = cfg.bitsPerSample();
  Serial.print("Bits Per Sample = ");
  Serial.println(x);

  x = kit.pinSpiCs();
  //many other pins defined for Lyrat and AI-Thinker boards, not used here

  Serial.print("SD CS pin = ");
  Serial.println(x);

}

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

void loop() {

  kit.setVolume(analogRead(POT1) >> 5);  // 0 to 128, max of 100.
  kit.setMute(digitalRead(KEY2));

  size_t len = kit.read(buffer, BUFFER_SIZE);  // BUFFER_SIZE limitations 512 - 4096
               kit.write(buffer, len);
}
```

Phil Schatzmann Sinewave

```
#pragma once

#include <stdint.h>

class SineWaveGenerator {

    public:

        // the scale defines the max value which is generated
        SineWaveGenerator(float amplitude = 32767.0, float phase = 0.0){
            m_amplitude = amplitude;
            m_phase = phase;
        }

        /// Defines the frequency - after the processing has been started
        void setFrequency(uint16_t frequency) {
            this->m_frequency = frequency;
        }

        void setSampleRate(uint16_t sr){
            sample_rate = sr;
            this->m_deltaTime = 1.0 / sample_rate;
        }

        /// Provides a single sample
        int16_t readSample() {
            float angle = double_Pi * m_frequency * m_time + m_phase;
            int16_t result = m_amplitude * sin(angle);
            m_time += m_deltaTime;
            return result;
        }

        /// filles the data with 2 channels
        size_t read(uint8_t *buffer, size_t bytes){
            size_t result;
            int16_t *ptr = (int16_t*)buffer;
            for (int j=0;j<bytes/4;j++){
                int16_t sample = readSample();
                *ptr++ = sample;
                *ptr++ = sample;
                result+=4;
            }
            return result;
        }

    protected:
        int sample_rate;
        float m_frequency = 0;
        float m_time = 0.0;
        float m_amplitude = 1.0;
        float m_deltaTime = 0.0;
        float m_phase = 0.0;
        float double_Pi = PI * 2.0;

};
```

```
/**
 * @file output.ino
 * @author Phil Schatzmann
 * @brief Output of audio data to the AudioKit
 * @date 2021-12-10
 * @copyright Copyright (c) 2021
 */

#include "AudioKitHAL.h"
#include "SineWaveGenerator.h"

// ~~~~~~~~~~~~~~~~~~  CONSTANTS/VARIALBES  ~~~~~~~~~~~~~~~~

const byte POT1 = 34;
const byte KEY1 = 32;

AudioKit kit;
SineWaveGenerator wave;

const int BUFFER_SIZE = 1024;
uint8_t buffer[BUFFER_SIZE];

// ~~~~~~~~~~~~~~~~~~  SETUP  ~~~~~~~~~~~~~~~~
void setup() {

  pinMode(KEY1, INPUT_PULLUP);  //internal pullup

  auto cfg = kit.defaultConfig(AudioOutput);
  kit.begin(cfg);

  wave.setFrequency(1000);  //1k Hz
  wave.setSampleRate(cfg.sampleRate());
}

// ~~~~~~~~~~~~~~~~~~ LOOP  ~~~~~~~~~~~~~~~~
void loop() {

  if(!digitalRead(KEY1))  { wave.setFrequency(30 + analogRead(POT1));   }

  size_t l = wave.read(buffer, BUFFER_SIZE);
  kit.write(buffer, l);
}
```

Phil Schatzmann Sinewave Distort

```
/**
 * SineWave to Effects to I2S AudioKit output
 *
 * @AudioTools and AudioKit Libraries author Phil Schatzmann
 * @copyright GPLv3
 *
 * PCB_Artists ES8388 Module connected to ESP32_Dev_Kit with pots and switches.
 * Setup ESP32 pin# in AudioKit/src/generic_es8388/board_def.h
 * AudioTools Library used for generic ES8388 driver, and its audioEffects.h Classes
 * Effect settings read from analogRead() of several pots wired to the ESP32_Dev_Kit
 *
 * AnalogRead() in the Main Loop causes lots of noise on the output, so a momentary
 * pushbutton switch is used to momentarily enter new settings.
 */

 // ~~~~~~~~~~~~~~~~~~~  AUDIO TOOLS Library  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "AudioTools.h"
#include "AudioLibs/AudioKit.h"  // includes AudioKitHAL.h from AudioKit Library

// ~~~~~~~~~~~~~~~~~  CONSTANTS/VARIALBES  ~~~~~~~~~~~~~~~

const byte LED1 = 4;
const byte LED2 = 15;

const byte POT1 = 34;
const byte POT2 = 39;
const byte POT3 = 36;
const byte POT4 = 14;
const byte POT5 = 12;  //must be set near zero for program load to work
const byte LIGHT = 27;

const byte KEY1 = 32;
const byte KEY2 = 33;

short pot1;
short pot2;
short pot3;
short pot4;
short pot5;
short light;

bool key1;
bool key2;

int x=0;

// Initial Effects Control values
float volumeControl = 1.0;     //Boost volume fraction 0 to 1
int16_t clipThreashold = 4990; //Distortion clipThreshold 0 to maxInput of 6500
float fuzzEffectValue = 6.5;   //Fuzz 0 to 20 ??
int16_t tremoloDuration = 200; //Tremolo volume pulse time in milliseconds. Down to AM modulation.
int8_t  tremoloDepth = 80;     //Tremolo pulsing depth 0 to 100 percent
```

```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  AUDIO SETUP  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

const int sample_rate = 22050;
const int channels = 1;

SineWaveGenerator<int16_t> sine;                // "sine"-> subclass of SoundGenerator
//GeneratedSoundStream<int16_t> sound(sine);     // "sound"-> sine turned into a SoundStream, not used
AudioEffects<SineWaveGenerator<int16_t>> effects(sine);  // "effects"-> Instance of AudioEffects on sine
GeneratedSoundStream<int16_t> in(effects);              // "in"-> effects turned into a SoundStream

//from AudioEffect.h, see also Boost, Delay, ADSRGain.
Fuzz fuzz(fuzzEffectValue);                                 // "fuzz"-> instance of Fuzz Effect
Distortion distort(clipThreashold);                         // "distort"-> instance of Distortion Effect
Tremolo tremlo(tremoloDuration, tremoloDepth, sample_rate); // "tremlo"-> instance of Tremolo Effect

AudioKitStream out;                // "out"-> Instance of AudioKitStream for 8388 DAC out
StreamCopy copier(out, in);        // "copier"->Instance of StreamCopy, copy "in" to "out"

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

void setup(void) {
  // Open Serial
  Serial.begin(115200);
  delay(2000);
  Serial.println("STARTING");
  AudioLogger::instance().begin(Serial, AudioLogger::Warning); //other levels: Debug, Info, Error

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

  // setup effects, un-comment the ones you want to try out
  //effects.addEffect(new Boost(volumeControl));
  //effects.addEffect(fuzz);
 effects.addEffect(tremlo);
   effects.addEffect(distort);

  auto config = out.defaultConfig(TX_MODE);  //"config" method used to configure "out"
  config.sample_rate = sample_rate;
  config.channels = channels;
  config.bits_per_sample = 16;
  config.sd_active = false;

  out.begin(config);               // start AudioKitStream to ES8388 DAC out
  sine.begin(config, N_B4);        // start SineWave, initial frequency note N_84
  in.begin(config);                // start Effects Stream
}

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

void loop() {
```

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

void loop() {

  if(!digitalRead(KEY1)){       // pushbutton to enter new effects settings

    sine.setFrequency(100 + analogRead(POT1));             // 0 to 4095 Hz

    // effect parameters, uncomment the ones you want to try out

    //fuzz.setFuzzEffectValue(analogRead(POT2) >> 7) ;     // 0 to 31
    tremlo.setDepth(analogRead(POT2) >> 5);                // 0 to 127 percent
    tremlo.setDuration(analogRead(POT3) >> 3);             // 0 to 511 ms
    distort.setClipThreashold(analogRead(POT4) >> 1);      // 0 to 2047 peak amplitude
  }

  copier.copy();                                           // copy "in" to "out" with "copier"
}
```

Phil Schatzmann Sinewave Distort 2Core

```
/**
 * SineWave to Effects to I2S AudioKit output
 *
 * @AudioTools and AudioKit Libraries author Phil Schatzmann
 * @copyright GPLv3
 *
 * PCB_Artists ES8388 Module connected to ESP32_Dev_Kit with pots and switches.
 * Setup ESP32 pin# in AudioKit/src/generic_es8388/board_def.h
 * AudioTools Library used for generic ES8388 driver, and its audioEffects.h Classes
 * Effect settings read from analogRead() of several pots wired to the ESP32_Dev_Kit
 *
 * ESP32 has 2 processing cores that can run in parallel.
 *
 * Core 0 Task is set up to load effects parameters from pot values.
 * Taking these tasks out of Core 1 running the Main Loop
 * keeps them from slowing down and interrupting copier() which creates noise.
 *
 * Core 1 Task is the Main Loop that handles only the copier()
 */

 // ~~~~~~~~~~~~~~~~~~~  AUDIOTOOLS Library  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "AudioTools.h"
#include "AudioLibs/AudioKit.h"  // includes AudioKitHAL.h from AudioKit Library

TaskHandle_t Task1;  // Task1 will be assigned to ESP32 Core 0 processor

// ~~~~~~~~~~~~~~~~~~  CONSTANTS/VARIALBES  ~~~~~~~~~~~~~~~~

const byte LED1 = 4;
const byte LED2 = 15;

const byte POT1 = 34;
const byte POT2 = 39;
const byte POT3 = 36;
const byte POT4 = 14;
const byte POT5 = 12;  //WARNING!! Must be dialed to zero for program load to work
const byte LIGHT = 27;

const byte KEY1 = 32;
const byte KEY2 = 33;


// Initial Effects Control values
float volumeControl = 1.0;       //Boost volume fraction 0 to 1
int16_t clipThreashold = 4990;   //Distortion clipThreshold 0 to maxInput of 6500
float fuzzEffectValue = 6.5;     //Fuzz 0 to 20 ??
int16_t tremoloDuration = 200;   //Tremolo volume pulse time in milliseconds. Down to AM modulation.
int8_t  tremoloDepth = 80;       //Tremolo pulsing depth 0 to 100 percent

int16_t old_freq = 0;
int16_t new_freq = 0;
float old_vol = 0;
float new_vol = 0;
```

```
int16_t old_threash = 0;
int16_t new_threash = 0;
float old_fuzz = 0;
float new_fuzz = 0;
int16_t old_dur = 0;
int16_t new_dur = 0;
int8_t old_depth = 0;
int8_t new_depth = 0;

int8_t state = 4;

const int freqNumReadings = 10;
int freqReadings[freqNumReadings];
int freqTotal = 0;
int freqIndex = 0;
int freqRead = 0;
int freqAverage = 0;

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  AUDIO SETUP  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

const int sample_rate = 22050;
const int channels = 1;

SineWaveGenerator<int16_t> sine;                        // "sine"-> subclass of SoundGenerator
//GeneratedSoundStream<int16_t> sound(sine);              // "sound"-> sine turned into a SoundStream, not used
AudioEffects<SineWaveGenerator<int16_t>> effects(sine);     // "effects"-> Instance of AudioEffects on sine
GeneratedSoundStream<int16_t> in(effects);                // "in"-> effects turned into a SoundStream

//from AudioEffect.h, see also Boost, Delay, ADSRGain.
Fuzz fuzz(fuzzEffectValue);                             // "fuzz"-> instance of Fuzz Effect
Distortion distort(clipThreashold);                       // "distort"-> instance of Distortion Effect
Tremolo tremlo(tremoloDuration, tremoloDepth, sample_rate); // "tremlo"-> instance of Tremolo Effect
Boost vol(volumeControl);                               // "vol"-> instance of Boost Effect (volume)

AudioKitStream out;                                     // "out"-> Instance of AudioKitStream for 8388 DAC out
StreamCopy copier(out, in);                               // "copier"->Instance of StreamCopy, copy "in" to "out"

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

void setup(void) {
  // Open Serial
  Serial.begin(115200);
  delay(2000);
  Serial.println("STARTING");
  AudioLogger::instance().begin(Serial, AudioLogger::Warning); // other levels: Debug, Info, Error

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

  // setup effects, un-comment the ones you want to use.
  // effects are applied in a chain, in the "add" order set here.

   effects.addEffect(distort);
   effects.addEffect(tremlo);
   //effects.addEffect(fuzz);
   //effects.addEffect(vol);
```

```
  auto config = out.defaultConfig(TX_MODE);                  //"config" method used to configure "out"
  config.sample_rate = sample_rate;
  config.channels = channels;
  config.bits_per_sample = 16;
  config.sd_active = false;

  out.begin(config);                                  // start AudioKitStream to ES8388 DAC out
  sine.begin(config, N_B4);                               // start SineWave, initial frequency note N_84
  in.begin(config);                                  // start Effects Stream


  //create a task executed in Task1code() function, with priority 1 and executed on core 0
  xTaskCreatePinnedToCore(
                Task1code, /* Task function. */
                  "Task1", /* name of task (shown below). */
                  10000,  /* Stack size of task */
                  NULL,   /* parameter of the task */
                  1,      /* priority of the task */
                  &Task1, /* handle to keep track of task */
                  0       /* pin task to core0*/
  );

 delay(500);

} //End of Setup


// ~~~~~~~~~~~~~~~~~~~  Task1 Code  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  void Task1code( void * pvParameters ){
  Serial.print("Task1 running on core ");
  Serial.println(xPortGetCoreID());


for(;;){  //loop to continually update pot effect settings.

    delay(100); // limit how often effect parameters change


// Get Frequency by Averaging out the constant frequency readings from POT1
// set freqNumReadings higher for stabler freq.

  freqRead = analogRead(POT1) + 100;   // SineWave -> 100 to 4195 Hz, frequency
  freqTotal = freqTotal - freqReadings[freqIndex];
  freqReadings[freqIndex] = freqRead;
  freqTotal += freqRead;
  ++freqIndex;
  if (freqIndex >= freqNumReadings) { freqIndex = 0; }
  if (int(freqTotal/freqNumReadings) != freqAverage) { freqAverage = int(freqTotal/freqNumReadings); }

    new_freq = freqAverage ;
    //Serial.print("frequency = ");  Serial.println(new_freq);
    if (new_freq != old_freq) {
      old_freq = new_freq;
      sine.setFrequency(new_freq);
    }
```

```
 //Get Frequency from POT1 by pressing KEY1.  The Switch construct provides debouncing.
/*
   switch (state) {
  case 1:
   if (!digitalRead(KEY1)) { state = 5; }  //waiting for a key press
   break;
  case 2:
   if (digitalRead(KEY1)) { state = 1; }   //waiting for the key to be released
   break;
  case 3:                                  //debounce: key was pressed wait one delay cycle
   state = 4;
   break;
  case 4:                                  //after key pressed load new sine frequency from POT1
   sine.setFrequency(analogRead(POT1) + 100);
   state = 2;
   break;
  case 5:
   if (!digitalRead(KEY1)) { state = 4; }  //debounce: confirmed key is still pressed
   else { state = 1; }                     //debounce: false alarm, go back to waiting
   break;
  }
*/

/*
   new_vol = (float)(analogRead(POT5) / 4095.0) ;   // Boost -> 0 to 1, fractional amplitude
   //Serial.print("volume = ");  Serial.println(new_vol);
   if (abs(new_vol - old_vol) > 0.1 ) {
     old_vol = new_vol;
     vol.setVolume(new_vol);
   }
*/
   new_threash = (analogRead(POT4) << 3) ;         // Distortion -> 0 to 32k, peak amplitude
   //Serial.print("Distortion Threshold = ");  Serial.println(new_threash);
   if (new_threash != old_threash) {
     old_threash = new_threash;
     distort.setClipThreashold(new_threash);
   }
/*
   new_fuzz = (float)(analogRead(POT4) >> 7) ;   // Fuzz -> 0 to 31,  clipping amount
   //Serial.print("fuzz = ");  Serial.println(new_fuzz);
   if (new_fuzz != old_fuzz) {
     old_fuzz = new_fuzz;
     fuzz.setFuzzEffectValue(new_fuzz);
   }
*/
   new_dur = analogRead(POT2) >> 3 ;           // Tremolo Duration -> 0 to 511 ms, pulse cycle time
   //Serial.print("Tremolo Duration = ");  Serial.println(new_dur);
   if (new_dur != old_dur) {
     old_dur = new_dur;
     tremlo.setDuration(new_dur);
   }

   new_depth = analogRead(POT3) >> 5 ;         // Tremolo Depth -> 0 to 127 percent, pulse depth
   //Serial.print("Tremolo Depth = ");  Serial.println(new_depth);
   if (new_depth != old_depth) {
     old_depth = new_depth ;
     tremlo.setDepth(new_depth);
   }
```

```
} //End of for()
} //End of Task1code

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

void loop() {

  copier.copy();   // copy "in" to "out" with "copier"

}                 //End of Loop()


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

Phil Schatzmann Input Effects - 2 Core

```
/**
 * External AudioKit audio-in to Effects to I2S AudioKit output
 *
 * @AudioTools and AudioKit Libraries author Phil Schatzmann
 * @copyright GPLv3
 *
 * PCB_Artists ES8388 Module connected to ESP32_Dev_Kit with pots and switches.
 * Setup ESP32 pin# in AudioKit/src/generic_es8388/board_def.h
 * AudioTools Library used for generic ES8388 driver, and its audioEffects.h Classes
 * Effect settings read from analogRead() of several pots wired to the ESP32_Dev_Kit
 *
 * ESP32 has 2 processing cores that can run in parallel.
 *
 * Core 0 Task is set up to load effects parameters from pot values.
 * Taking these tasks out of Core 1 running the Main Loop
 * keeps them from slowing down and interrupting copier() which creates noise.
 *
 * Core 1 Task is the Main Loop that handles only the copier()
 */

 // ~~~~~~~~~~~~~~~~~~~  AUDIOTOOLS Library  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#include "AudioTools.h"
#include "AudioLibs/AudioKit.h"  // includes AudioKitHAL.h from AudioKit Library

TaskHandle_t Task1;  // Task1 will be assigned to ESP32 Core 0 processor

// ~~~~~~~~~~~~~~~~~~  CONSTANTS/VARIALBES  ~~~~~~~~~~~~~~~~

const byte LED1 = 4;
const byte LED2 = 15;

const byte POT1 = 34;
const byte POT2 = 39;
const byte POT3 = 36;
const byte POT4 = 14;
const byte POT5 = 12;  //WARNING!! Must be dialed to zero for program load to work
const byte LIGHT = 27;

const byte KEY1 = 32;
const byte KEY2 = 33;

// Initial Effects Control values
float volumeControl = 1.0;       //Boost volume fraction 0 to 1
int16_t clipThreashold = 4990;   //Distortion clipThreshold 0 to maxInput of 6500
float fuzzEffectValue = 6.5;     //Fuzz 0 to 20 ??
int16_t tremoloDuration = 200;   //Tremolo volume pulse time in milliseconds. Down to AM modulation.
int8_t  tremoloDepth = 80;       //Tremolo pulsing depth 0 to 100 percent
int16_t delayDuration = 1000;    //Delay time in milliseconds.
int8_t  delayDepth = 50;         //Delay mix depth 0 to 100 percent
```

```
float old_vol = 0;
float new_vol = 0;
int16_t old_threash = 0;
int16_t new_threash = 0;
float old_fuzz = 0;
float new_fuzz = 0;
int16_t old_dur = 0;
int16_t new_dur = 0;
int8_t old_depth = 0;
int8_t new_depth = 0;
int16_t old_durx = 0;
int16_t new_durx = 0;
int8_t old_depthx = 0;
int8_t new_depthx = 0;

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  AUDIO SETUP  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

const int sample_rate = 22050;  //22050;
const int channels = 1;

AudioKitStream kit;                                    // "kit"-> Instance of AudioKitStream, 8388 ADCin and DAC out
AudioEffects<GeneratorFromStream<effect_t>> effects(kit,channels); // "effects"-> Instance of AudioEffects on
"kit"
                                                       // Stream converted to a Generator
GeneratedSoundStream<int16_t> in(effects);                 // "in"-> effects turned into a SoundStream

//from AudioEffect.h

//Fuzz fuzz(fuzzEffectValue);                              // "fuzz"-> instance of Fuzz Effect
//Distortion distort(clipThreashold);                        // "distort"-> instance of Distortion Effect
Tremolo tremlo(tremoloDuration, tremoloDepth, sample_rate); // "tremlo"-> instance of Tremolo Effect
// Delay delayx(delayDuration, delayDepth, sample_rate);      // "delayx"-> instance of Delay Effect
//Boost vol(volumeControl);                                 // "vol"-> instance of Boost Effect (volume)

StreamCopy copier(kit, in);                              // "copier"->Instance of StreamCopy, copy "in" to "kit"
                                                   // (both from kit to effects or effects to kit are supported)
// ~~~~~~~~~~~~~~~~~~~~  SETUP  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

void setup(void) {
  // Open Serial
  Serial.begin(115200);
  delay(2000);
  Serial.println("STARTING");
  AudioLogger::instance().begin(Serial, AudioLogger::Warning); // other levels: Debug, Info, Error

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

  // setup effects, un-comment the ones you want to use.
  // effects are applied in a chain, in the "add" order set here.

   //effects.addEffect(distort);
   effects.addEffect(tremlo);
   //effects.addEffect(delayx);
   //effects.addEffect(fuzz);
   //effects.addEffect(vol);
```

```
  auto config = kit.defaultConfig(RXTX_MODE);               //"config" method used to configure "out"
  config.sample_rate = sample_rate;
  config.channels = channels;
  config.bits_per_sample = 16;
  config.sd_active = false;
  config.input_device = AUDIO_HAL_ADC_INPUT_LINE2;

  kit.begin(config);                                // start AudioKitStream to ES8388 DAC out
  in.begin(config);                                 // start Effects Stream


  //create a task executed in Task1code() function, with priority 1 and executed on core 0
  xTaskCreatePinnedToCore(
                Task1code, /* Task function. */
                  "Task1", /* name of task (shown below). */
                  20000,  /* Stack size of task */
                  NULL,   /* parameter of the task */
                  1,      /* priority of the task */
                  &Task1, /* handle to keep track of task */
                  0       /* pin task to core0*/
  );

 delay(500);

} //End of Setup


// ~~~~~~~~~~~~~~~~~~~  Task1 Code  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  void Task1code( void * pvParameters ){
  Serial.print("Task1 running on core ");
  Serial.println(xPortGetCoreID());


for(;;){  //loop to continually update pot effect settings.

    delay(200); // limit how often effect parameters change
/*
    new_vol = (float)(analogRead(POT5) / 4095.0) ;   // Boost -> 0 to 1, fractional amplitude
    //Serial.print("volume = ");  Serial.println(new_vol);
    if (abs(new_vol - old_vol) > 0.1 ) {
      old_vol = new_vol;
      vol.setVolume(new_vol);
    }
*/
/*
    new_threash = (analogRead(POT4) << 3) ;         // Distortion -> 0 to 32k, peak amplitude
    //Serial.print("Distortion Threshold = ");  Serial.println(new_threash);
    if (new_threash != old_threash) {
      old_threash = new_threash;
      distort.setClipThreashold(new_threash);
    }
*/
/*
    new_fuzz = (float)(analogRead(POT4) >> 7) ;   // Fuzz -> 0 to 31,  clipping amount
    //Serial.print("fuzz = ");  Serial.println(new_fuzz);
    if (new_fuzz != old_fuzz) {
      old_fuzz = new_fuzz;
      fuzz.setFuzzEffectValue(new_fuzz);
    }
*/
```

```
    new_dur = analogRead(POT2) >> 3 ;           // Tremolo Duration -> 0 to 511 ms, pulse cycle time
    //Serial.print("Tremolo Duration = ");  Serial.println(new_dur);
    if (new_dur != old_dur) {
      old_dur = new_dur;
      tremlo.setDuration(new_dur);
    }

    new_depth = analogRead(POT3) >> 5 ;         // Tremolo Depth -> 0 to 127 percent, pulse depth
    //Serial.print("Tremolo Depth = ");  Serial.println(new_depth);
    if (new_depth != old_depth) {
      old_depth = new_depth ;
      tremlo.setDepth(new_depth);
    }

/*
    new_durx = analogRead(POT2) >> 1 ;          // Delay Duration -> 0 to 1023 ms
    //Serial.print("Delay Duration = ");  Serial.println(new_durx);
    if (new_durx != old_durx) {
      old_durx = new_durx;
      delayx.setDuration(new_durx);
    }

    new_depthx = analogRead(POT3) >> 5 ;        // Delay Depth -> 0 to 127 percent, delay mix
    //Serial.print("Delay Depth = ");  Serial.println(new_depthx);
    if (new_depthx != old_depthx) {
      old_depthx = new_depthx ;
      delayx.setDepth(new_depthx);
    }
*/
} //End of for()
} //End of Task1code

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

void loop() {

  copier.copy();   // copy "in" to "out" with "copier"

}                 //End of Loop()


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

Blackstomp Pedal from Deeptronics

The Deeptronics group developed an Effects Pedal Box using the CS8388 Codec (https://www.deeptronic.com/blackstomp/). Their Arduino Library includes driver software for the CS8388, though they are also developing software for a number of other codecs. Their example Arduino sketches work on our ESP32 PCB_Artists board but with lots of output noise. Something in the software needs adjustments for our board to fix the noise problem. This software is currently still a work in progress.

In order to use the Blackstomp Library (https://github.com/hamuro80/blackstomp) for our board the following GPIO pin assignments must be edited in the Library file Blackstomp/scr/blackstomp.cpp Note all the sections marked “CHANGED”.

```
#include "blackstomp.h"
//#include "ac101.h"
#include "driver/i2s.h"
#include "esp_task_wdt.h"
#include "math.h"
#include "EEPROM.h"
#include "codec.h"

//CONTROL INPUT
//CHANGED
#define P1_PIN  34
#define P2_PIN  39
#define P3_PIN  36
#define P4_PIN  14
#define P5_PIN  12
#define P6_PIN  27

//ROTARY ENCODER
//CHANGED
#define RE_BUTTON_PIN 33
#define RE_PHASE0_PIN 33
#define RE_PHASE1_PIN 33

//FOOT SW PIN SETUP
//CHANGED
#define FS_PIN 32

//LED INDICATOR PIN SETUP
//CHANGED
#define MAINLED_PIN 4
#define AUXLED_PIN 15

//OLED Display PIN SETUP
#define SCK_PIN 22
#define SDA_PIN 21

//Digital input Generic Assignment
#define D1_PIN_AC101     FS_PIN
#define D2_PIN_AC101     RE_BUTTON_PIN
#define D3_PIN_AC101     RE_PHASE0_PIN
#define D4_PIN_AC101     RE_PHASE1_PIN

#define D1_PIN_ES8388    RE_BUTTON_PIN
#define D2_PIN_ES8388    SCK_PIN
#define D3_PIN_ES8388    SDA_PIN


//ESP32-A1S-AC101 PIN SETUP
#define I2S_NUM         (0)
#define I2S_MCLK        (GPI_NUM_0)
#define I2S_BCK_IO      (GPIO_NUM_27)
#define I2S_WS_IO       (GPIO_NUM_26)
#define I2S_DO_IO       (GPIO_NUM_25)
#define I2S_DI_IO       (GPIO_NUM_35)

#define AC101_SDA                   (GPIO_NUM_33)
#define AC101_SCK                   (GPIO_NUM_32)
#define AC101_ADDR                  0x1A
```

```
//ESP32-A1S-ES8388 PIN SETUP
//CHANGED
#define I2S_BCK_IO_ES    (GPIO_NUM_5)
#define I2S_WS_IO_ES    (GPIO_NUM_25)
#define I2S_DI_IO_ES    (GPIO_NUM_35)
#define I2S_DO_IO_ES     (GPIO_NUM_26)

#define ES8388_SDA              (GPIO_NUM_18)
#define ES8388_SCK              (GPIO_NUM_23)
#define ES8388_ADDR             0x10

//audio processing frame length in samples (L+R) 64 samples (32R+32L) 256 Bytes
#define FRAMELENGTH    64
//sample count per channel for each frame (32)re
#define SAMPLECOUNT   FRAMELENGTH/2
//channel count inside a frame (always stereo = 2)
#define CHANNELCOUNT  2
//frame size in bytes
#define FRAMESIZE   FRAMELENGTH*4
//audio processing priority
#define AUDIO_PROCESS_PRIORITY  10

//dma buffer length 32 bytes (8 samples: 4L+4R)
#define DMABUFFERLENGTH 32
//dma buffer count 20 (640 Bytes: 160 samples: 80L+80R)
#define DMABUFFERCOUNT  20

//CHANGED
//codec instance
//static AC101 _codec;
static codec* _acodec;
static bool _es8388Mode = true;
DEVICE_TYPE _deviceType = DT_ESP32_A1S_ES8388;
static uint8_t _codecAddress = 0;
static bool _muteLeftAdcIn = false;
static bool _muteRightAdcIn = false;
```

Platform IO Code

As discussed previously, building Arduino projects from Microsoft's Visual Studio Code (VSC) can have several advantages over the Arduino IDE platform like a more versatile Editor and better Debugger.

This section will take one of the Arduino IDE code examples from above and re-build it from within the Platform IO extension installed in Visual Studio Code.

Pschatzmann's Sinewave

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

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

lib_extra_dirs = ../lib
lib_ldf_mode = deep+
build_flags =  -DCORE_DEBUG_LEVEL=2 -DAUDIOKIT_BOARD=7
monitor_speed = 115200
monitor_filters = esp32_exception_decoder
```

Every Platform IO (PIO) project includes a platform.ini file as shown above. In the example above, platform, board, and framework values come from values entered by the user when first creating a New Project.

The monitor_speed is the baud rate for the program Monitor which displays any Serial.print() lines in the program.

The lib_extra_dirs = ../lib line directs the project compiler to the project directory location of a “lib” folder that contains any extras Libraries used by the project. It could also direct the compiler to a particular https web address that contains the library (lib_deps = https://github.com/pschatzmann/arduino-audiokit-hal).

Most common Libraries can be loaded into the project folder using the PIO “Library” menu button. This convenient feature can search for a library over the internet and also periodically load updates. In this case, however, the Pschatzmann Audio libraries were manually found from GitHub, edited with specific GPIO pin settings, and placed inside a “lib” folder, inside the directory that contains the “Sinewave” project folder. In this way, with the lib_extra_dirs directive, other projects can be directed to the same Library without have to duplicate it within their own project folders, in much the same way as Arduino sketches can share libraries from the Arduino “library” folder.

The build_flags directive defines any number of constants needed by the program. Each constant starts with “-D” followed by the constant’s name and value after the equals sign.

CORE_DEBUG_LEVEL, defined in a “Logger” library, sets how many program parameters will be printed to the Monitor while checking the progress of the program, useful when debugging the program.

AUDIOKIT_BOARD sets what Codec board drivers to use in the program. This is defined in the library file arduino-audiokit-main/src/ AudioKitSettings.h Here we set it to #7, a generic ES8388 board.

Check https://github.com/pschatzmann/arduino-audiokit/wiki/PlatformIO for other PlatformIO.ini suggestions.

```
/**
 * @file output.ino
 * @author Phil Schatzmann
 * @brief Output of audio data to the AudioKit
 * @date 2021-12-10
 *
 * @copyright Copyright (c) 2021
 *
 */
#include <Arduino.h>
#include "AudioKitHAL.h"
#include "SineWaveGenerator.h"

AudioKit kit;
SineWaveGenerator wave;
const int BUFFER_SIZE = 1024;
uint8_t buffer[BUFFER_SIZE];

void setup() {
  Serial.begin(115200);
  // open in write mode
  auto cfg = kit.defaultConfig(AudioOutput);
  kit.begin(cfg);

  // 1000 hz
  wave.setFrequency(500);
  wave.setSampleRate(cfg.sampleRate());
}

void loop() {
  size_t l = wave.read(buffer, BUFFER_SIZE);
  kit.write(buffer, l);
}
```

The above file is the Arduino PlatformIO main.cpp file for the Sinewave project located in the “src” folder. Note that this is denoted as a cpp (C++) instead of “.ino” used in the Arduino IDE. Note also the required “include <Arduino.h>” at the start of the program. Otherwise, the code follows the original Arduino code.

The included SineWaveGenerator.h file, not shown here, is located in the same src folder as main.cpp.

IDF/ADF Code

Espressif built an extensive IDF development environment for its ESP32 microcontrollers, and extended it with ADF tools for its ESP32 based Audio boards such as the LyraT. Both IDF and the ADF extension can be used from VSC (Visual Studio Code).

Since the LyraT uses the ES8388 Codec, our ESP32/PCB_Artists board can also take advantage of coding in IDE and ADF from VSC. However, most of the example ADF programs deal with recording to and playback from LyraT's SD card. There are few if any live signal processing examples.

What follows are a couple examples of ES8388 template libraries built for the IDF/ADF environment in C programming language, a good starting point for developing your own Codec projects.

Each template library includes the same codec_es8388.h file which sets up the constants, structures, and enums to use in loading the 53 8-bit parameter registers on the ES8388. A main.c file then builds all the functions needed to initialize the codec and manipulate some of its parameters. Included in the initialization is setting up the I2C interface used to load and read the 53 codec registers, and the I2S interface used to send audio data to the codec DACs and receive audio data from the codec ADCs.

Thaaraak Template

A template application to be used with Espressif IoT Development Framework.

https://github.com/thaaraak/ESP32-ES8388

main.c also includes a main loop for playing back a sine wave while manipulating its volume from the sine wave construct (not from the codec I2C).

Main.c has #includes for the following files to be found in Espressif's esp-idf/ components library which is installed in the VSC IDF programming environment (https://github.com/espressif/esp-idf).

```
#include "freertos/FreeRTOS.h"
#include "esp_system.h"
#include "esp_event.h"
#include "esp_event_loop.h"
#include "esp_log.h"
#include "driver/i2c.h"
#include "driver/gpio.h"
#include "driver/i2s.h"
#include "soc/gpio_sig_map.h"
#include "codec_es8388.h"
#include “math.h”
```

Any of these library files can be pulled up and viewed on a tab within the VSC Editor window.

Olimex Template

A template driver to be used with Espressif IoT Development Framework.

https://github.com/OLIMEX/ESP32-ADF

This library is more extensive with code for other boards and examples unrelated to the ES8388. The ES8388 template code can be found inside the library at :

ESP32-ADF/SOFTWARE/esp-va-sdk/board_support_pkgs/olimex_esp32_adf/
esp_codec/es8388/components/codec_es8388/