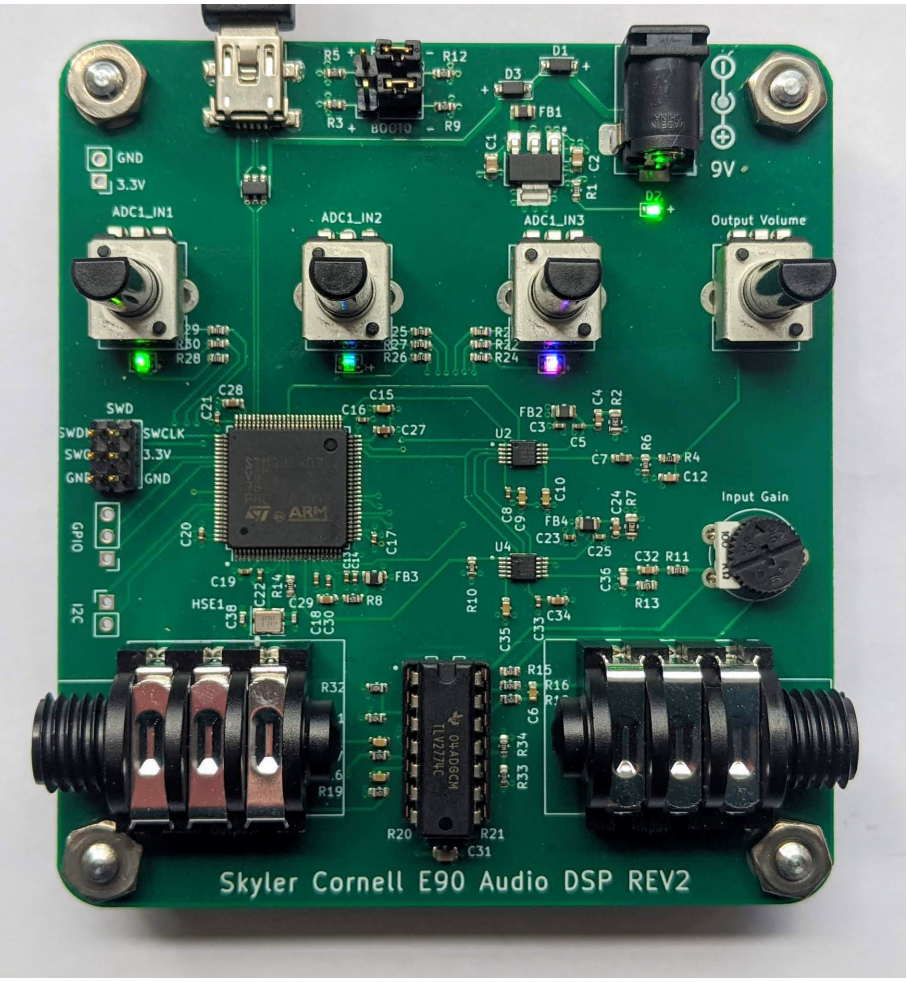
Audio DSP Development Board

Skyler Cornell

Advised by Prof. Erik Cheever

May 2021



1 Overview

This paper presents the design principles and practical use of the audio DSP development board designed by Skyler Cornell. The platform is designed for the development of DSP algorithms and provides the hardware and software infrastructure to get started writing programs manipulating real time audio signals in the C programming language.

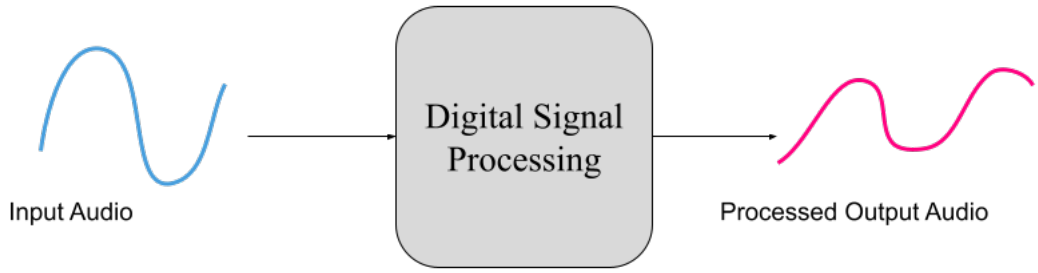


Figure 1: System Flow

The board manages the analog audio input and output circuits, microcontroller peripherals, and the data flow, allowing a programmer to focus their efforts on designing DSP. The platform consists of a hardware interface that electric instruments can plug directly into as well as an efficient underlying software framework to handle the high volume of audio data flow. The platform was designed for those with DSP or programming experience to develop real-time audio applications on, or for musicians to simply plug their instruments into and run pre-compiled effects. The eventual goal of the project is to curate a growing library of pre-compiled firmware written by a community of developers that anyone can download and easily upload to the board to try out with their instruments.

Rather than a designated DSP chip which tend to be more expensive and have limited peripheral interfaces, this development board instead uses a high performance microcontroller. The microcontroller on this board is from the STM32F4 Microcontroller family which can run the CPU at a maximum frequency of 168MHz, has a number of standard peripherals for project expansions, and plenty of information and vast open source community support online.

The board can run standard audio DSP applications such as digital filters, FFTs, and audio effects at professional audio standards (24 bit samples at up to 96 KHz sampling rate). While powerful enough for DSP applications, the board is also versatile and can leverage the numerous microcontroller

peripherals to support a variety of potential projects. The centerpiece of the development board, the STM32F407 microcontroller has multiple GPIO ports, peripherals for standard hardware protocols (I^2C,I^2S, SPI, CAN), support for dozens of external and internal interrupts, ADCs, DACs, and even peripherals designed for driving LCD displays. The board is equipped with potentiometers for users to control digital parameters, standard quarter inch mono audio connectors, RGB LEDs, and a USB Mini connection for power, firmware upgrades, and general purposedata transfer.

2 System Design

This section explores the platform's hardware and software topology. From left to right in the block diagram of figure 2, an analog audio signal passes through an amplifier which applies a gain to the signal. The amplified signal is sampled at 48 kHz by the ADC (Analog to Digital Converter) which converts the it to a digital value and transmits it serially to the microcontroller over an I^2S hardware protocol connection.

The microcontroller is configured to generate an interrupt when new samples are available on the I^2S line, at which point samples are processed by a programmers DSP algorithm. The processed sample data are placed in an output buffer in the program and on the next sample period (approx. 26 μS for 48 kHz sampling) the data is sent out to the DAC (digital to analog converter) over another I^2S connection.

The DAC reconstructs the signal into a continuous analog voltage. The last step in the signal flow is the post processing stage which contains an analog volume control circuit and a buffer which outputs the processed signal through the output audio jack.

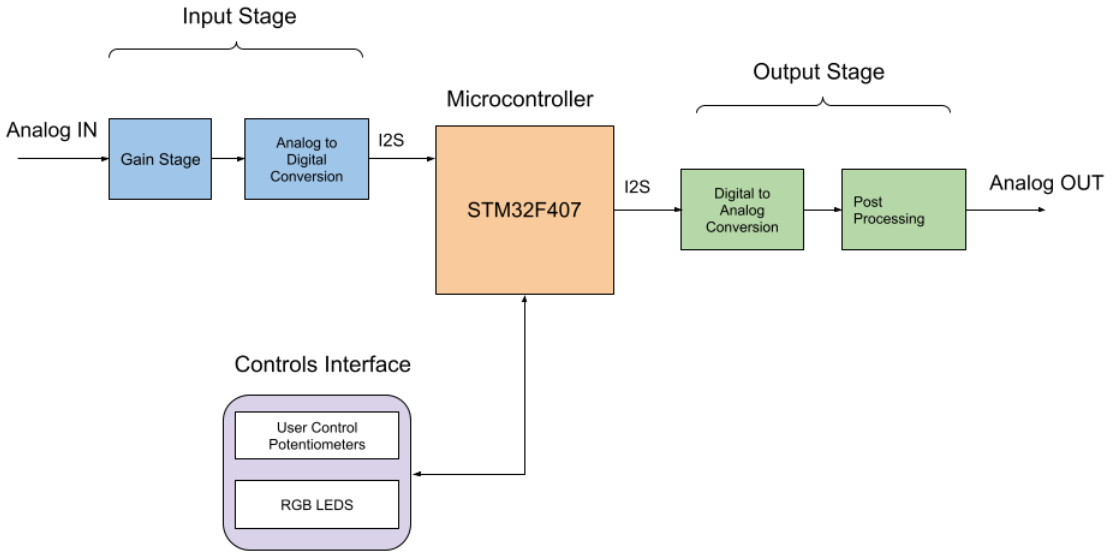


Figure 2: Hardware Topology: High Level

The four primary design components, the input stage, output stage, user controls interface, and the software data processing scheme are discussed in detail in the following sub sections.

2.1 Input Stage

The purpose of the input stage is to take the raw input audio voltage signal and convert it to a high fidelity digital signal. The output quality of the whole system is limited by the resolution of the input, so it is the purpose of the gain stage to expand the dynamic range of the input in order for the ADC to sample over its full 24 bit range. The input gain stage consists of an op amp in the inverting configuration with a gain varying from unity to x10, controllable by a potentiometer. Why not use a fixed gain? Standard line level audio voltages typically produce a maximum of 2V peak to peak, while electric guitars may output anywhere from 10's of milivolts peak to peak all the way to nearly a volt, depending on the make and model of the pickups. The variable gain allows a user to tune the board so that it best suits their signal source whether it be an electric organ, guitar, or a line out.

After the gain stage, the signal is sampled by the onboard CS5434 ADC at 48 kHz producing 24 bit signed integer samples. 24 bit samples have more than 16 million possible quantization levels, so an analog signal that fills the nearly the full dynamic range of the ADC is represented digitally by integers ranging from -8,000,000 to +8,000,000. After conversion, these digitized samples are transmitted to the microcontroller over I^2S where they are processed by a programmer's DSP routine.

2.2 Output Stage

The output stage consists of a CS4344 DAC chip followed by a volume control circuit and buffer. After processing the samples, the microcontroller transmits them over another I^2S line to the DAC, which reconstructs the analog signal. The reconstructed signal is then fed into the post processing circuit which controls the output volume by attenuating the analog signal through a potentiometer dependent voltage divider. The signal is then fed through a voltage follower op amp circuit which ensures the signal is produced with minimal output impedance with no DC offset. It must be noted that the output op amp is not sufficient to drive the necessary current for low impedance loads like speakers or studio headphones. The E90 board outputs should only be connected to other audio devices like amplifiers or recording equipment.

2.3 User Controls Interface

The user controls interface consists of 3 potentiometers and 3 RGB LEDs, all of which were designed to be arbitrarily configured by a program. The utility of the on-board potentiometers is their ability to control separate digital parameters in an application. In a digital filter implementation for example, pot 1 could control the center frequency and pot 2 could control the filter Q.

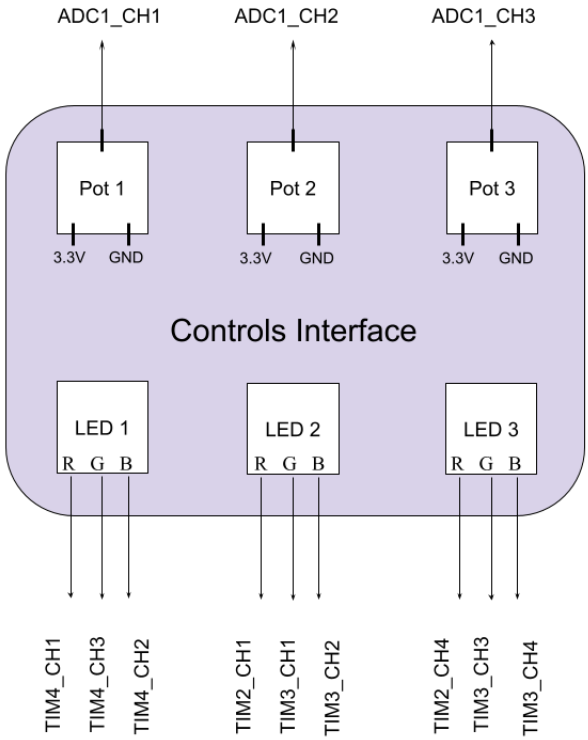


Figure 3: User Control Interfaces

The potentiometers have their first and third pins connected to 3.3V and GND and their center wiper pin connected to an ADC channel of the microcontroller. This is a standard embedded potentiometer configuration which forms a variable voltage divider that can sweep a range of digital values to the ADC with the turn of a knob. The potentiometer connections to the microcontroller are show in figure 4. The way the starter code is set up, ADC1 Channels 1-3 are configured in 8 bit mode providing 256 values the potentiometer can control, however the internal ADCs can be reconfigured to sample at resolutions up to 16 bits.

	Microcontroller Connection
Pot 1	ADC1_CH1
Pot 2	ADC1_CH2
Pot 3	ADC1_CH3

Figure 4: Potentiometer MCU Connections

The RGB LEDs are each placed beneath the 3 potentiometers on the board. The R,G, and B pins

of each LED are connected to microcontroller timer PWM channels shown in figure 4, which allows the programming of each LED to adopt a full range of possible colors. Note the LEDs are common anode so a low signal must be applied in order to turn them on.

		Microcontroller Connection
LED 1	R	TIM4_CH1
	G	TIM4_CH3
	B	TIM4_CH2
LED 2	R	TIM2_CH1
	G	TIM3_CH1
	B	TIM3_CH2
LED 3	R	TIM2_CH4
	G	TIM3_CH3
	B	TIM3_CH4

Figure 5: LED Connections

Because the LEDs are located near the potentiometers on the board, they could be configured in an application to visually indicate if the corresponding potentiometers actually map to a digital parameter or not. If a program uses pots 1 and 2 but not pot 3 for example, then LEDs 1 and 2 would be on, and LED 3 off. Another use of the LEDs could be for improving the safety of the input gain circuit by visually indicating signal clipping. The LEDs could easily be programmed to illuminate or flash red when the input exceeds a digital threshold, telling the user to decrease the input gain, preventing signal degradation due to clipping.

2.4 Software Data Stream Design

The software topology was designed to hold the occurrence of a new sample as the highest priority system event. When a new input sample is available (occurring 48 thousands times per second for a 48 kHz sampling rate), the CPU jumps into an interrupt routine to process the new sample. Through the help of the DMA (direct memory access) peripheral, new samples are automatically received and processed, while old ones are transmitted out, every sampling period. For more details on the data stream implementation see the following sub section "Double Buffer Data Stream."

Figure 6 outlines the three primary software functions. The high priority interrupt routine (shown in red) occurs every sample period and makes function calls to process() (shown in yellow) which contains the actual DSP code. The main loop, (shown in blue) is much lower priority and code there only executes using leftover CPU cycles, when it's not executing the high priority DSP. The main loop is where LED values may be updated and potentiometers values can be polled, which are fine to occur on a much slower time scale than the DSP.

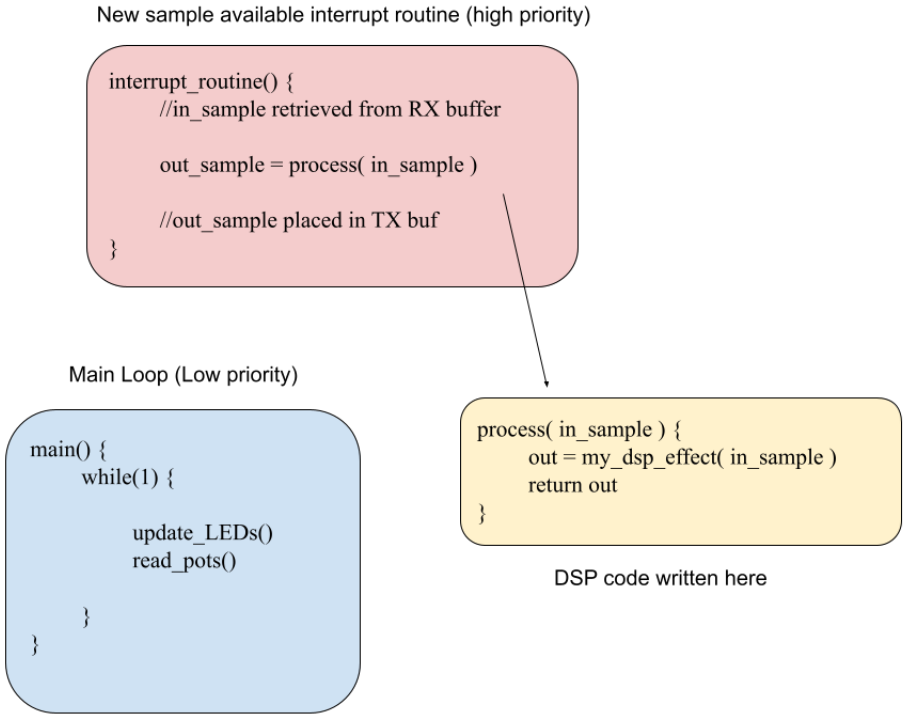


Figure 6: Software Topology

2.4.1 Double Buffer Data Stream

There is a constant flow of incoming and outgoing data samples in a digital signal processing application, so implementing an efficient way of handling the data stream is essential. The audio data stream for the E90 Board is implemented using what is known as a double buffer scheme. The double buffer data implementation takes advantage of the DMA (Direct Memory Access) STM32 peripheral to continuously transfer incoming samples into a RX (receive) buffer, while simultaneously transferring processed samples out of a separate TX (transmit) buffer. The DMA simply carts data to and from data memory and I^2S peripheral registers completely independently of the CPU, leaving it free to compute the DSP routine.

Two arrays are needed for a double buffer as the name implies, an rx_buf for storing incoming samples, and a tx_buf for storing outgoing processed samples. Each array hold twice the number of samples as a single block. For sample by sample processing which is the method discussed in this manual, a block is 1 sample, so the aforementioned buffers need only be large enough for 2 samples. The double buffer scheme is expressed in figure 7.

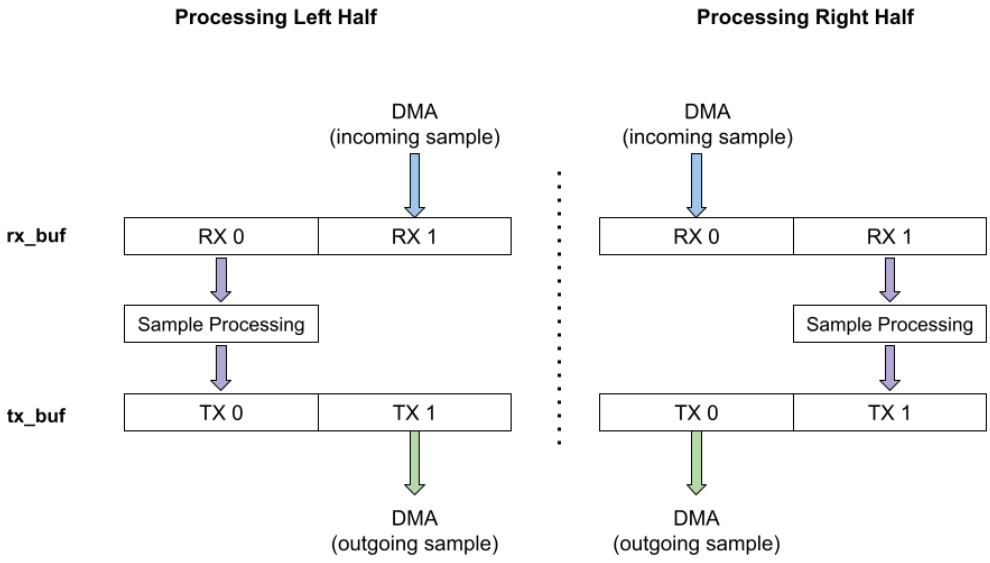


Figure 7: Double Buffer Audio Scheme

Processing Left Half:

As shown in Figure 7, the CPU processes the data contents of the left half of rx_buf (RX 0) and stores the processed value in the left half of tx_buf (TX 0). Simultaneously, the DMA is filling the right half of rx_buf (RX 1) as another DMA instance transfers the processed sample from the previous period (TX 1) to the I^2S output registers and eventually the DAC.

Processing Right Half:

This cycle alternates every sample period, repeating the same process on the other half of the buffer. The sample that was previously incoming and stored in RX 1 is processed and stored in TX 1 as a new sample is transferred in to RX 0 and old samples are transferred out of TX 0 by the DMA. This configuration is sometimes called a "ping-pong buffer" because the core alternates between processing the left and right halves of the double buffers.

In the time the DMA takes to fill up one half of the receive buffer with incoming samples from the I^2S unit, the CPU is free to process away. This of course means that for sample by sample processing, the CPU must complete its DSP routine within 1 sample period. The rx/tx buffers are twice the size of a single block so that the DMA can be filling up/overwriting one half of the buffer while still having safe access to the other half. The larger the block size the more time your processor has to process the samples before the next block comes in.

Implementation Details As walked through in detail in sections 4, the DMA generates a "transfer half complete interrupt" after filling up half of the receive buffer and generates a "transfer complete interrupt" after the DMA finishes transferring into the last index of the receive buffer array. The interrupts being for the half complete or complete transfer just distinguishes which half of the buffers are being processed. These two interrupt routine fire every other sample period, alternating. Hence, the DSP routine must be called in the ISR of both the "transfer half complete" interrupt and the "transfer complete" interrupt routines for the DSP to occur each sample period.

2.5 Additional System Features

For extra project versatility and hack-ability, one I^2C port and three GPIO pins have been broken out (left side of board) for general use by a designer for their project.

The board may be powered using the DC barrel jack connector with 9-12V center ground supplies, or the 5V from a USB mini connection. It must be noted that the board should not be used with unregulated power supplies, which supply much higher than nominal voltage when little current is

drawn, as is the case for the E90 board generally. 12 volts is the absolute maximum the onboard voltage regulator can step down before getting brûléd.

The board also houses a USB Mini connector to be used for programming the board with the STM32CubeProgrammer desktop software tool by STMicroelectronics. The USB connection can be used for general purpose data transfer to/from a host computer or simply to be used as a power supply. Future applications using the USB connection for general purpose data transfer could include a desktop application for graphically designing DSP block connections that can transfer the data describing those connections to the board over USB.

3 Getting Started (No Code)

This tutorial section describes the procedure for getting started uploading effects to the E90 board without writing code or using external debugging hardware.

First, you must open STM32CubeProgrammer, the desktop application designed by ST for debugging and programming STM32 Microcontrollers. First time users can download it for free from ST's site https://www.st.com/en/development-tools/stm32cubeprog.html/ and install it.

Next, you will of course need to find the compiled binary file for the effect you would like to upload. A good place to start browsing for compiled effects to run on the E90 board is this project's github repository at https://github.com/Skyler-Cornell/E90 in the "Effect Binaries" directory which will continue to be under development as new features and audio DSP effects are added.

After acquiring the compiled binary you must put the microcontroller into bootloader mode which allows it to be programmed over USB. Put it into booloader mode by disconnecting the board from power and putting the BOOT0 jumper to the + position (left) and the BOOT1 jumper to the - position (right). These boot pins are located next to the USB connector near the top of the board. Refer to figure 8 for visual confirmation.

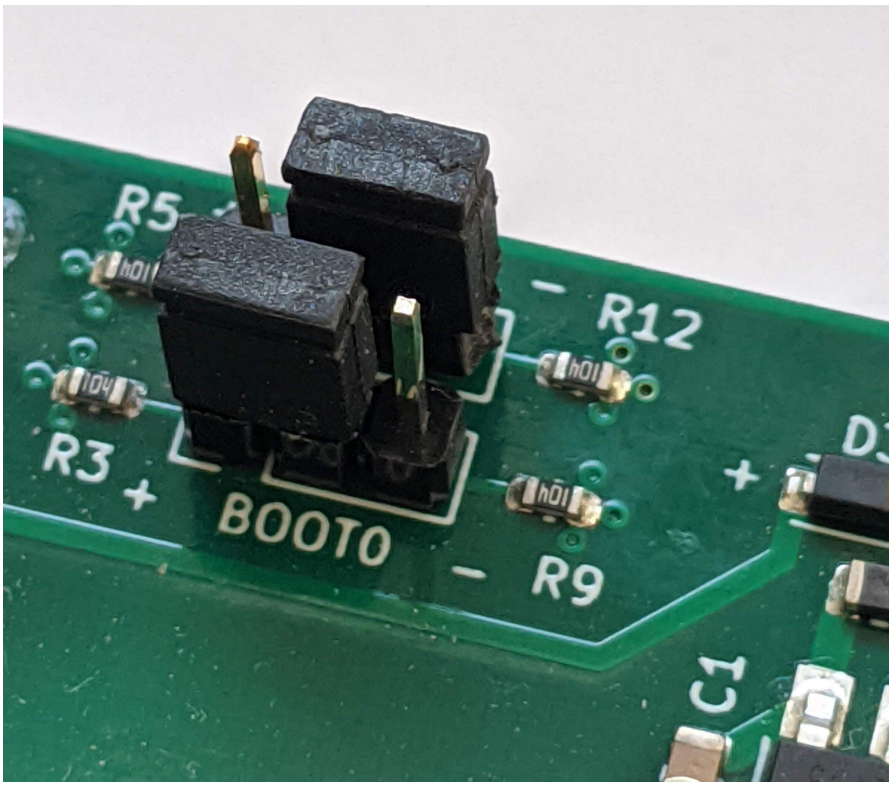


Figure 8: Boot Pin Configuration for USB Programming

Pulling BOOT0 high and BOOT1 low with the jumpers signals the microcontroller to run the built in bootloader program on the next power up cycle, which allows the microcontroller to be programmed over several available protocols, in this case over USB.

After placing the board in bootloader mode with the jumpers, connect the board to your computer with a USB mini cable and open the STM32CubeProgrammer application. You must establish the connection to the board by selecting USB as the hardware protocol in the upper right hand panel of STM32CubeProgrammer and then select the USB device (figure 9). The blue drop down option next to the green "Connect" button should be set to "USB," and the "Port" option should be set to the USB connection to the E90 board, in this case it shows up as "USB1". Click the refresh button if Port does not show a device.

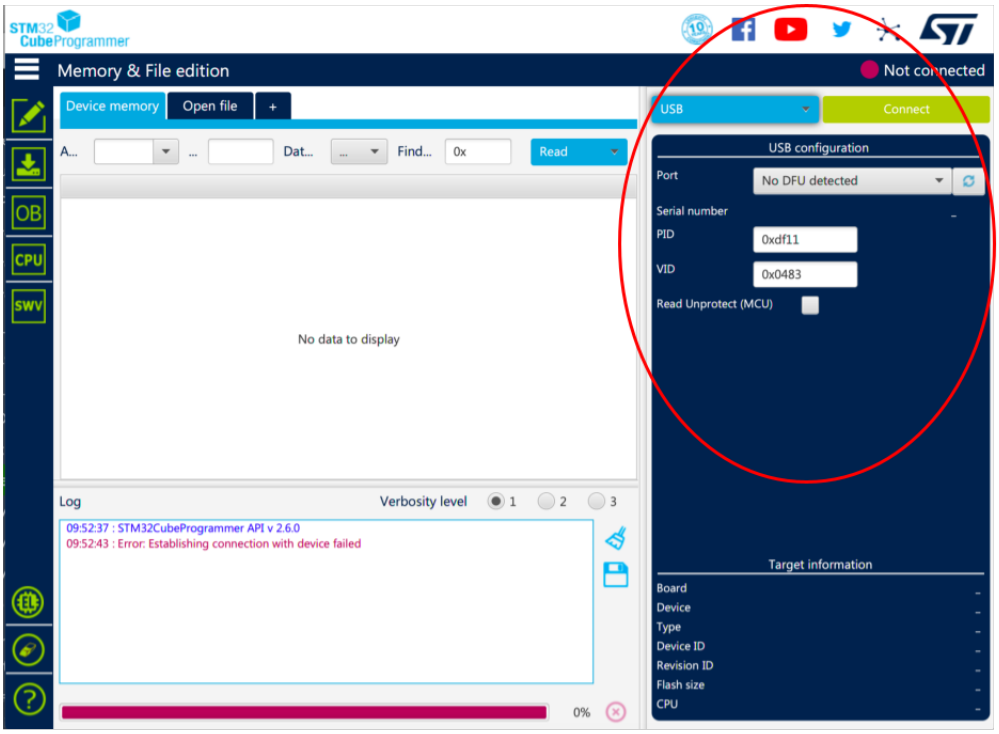


Figure 9: USB Programming Setting

Next, click the green Connect button which should establish the USB connection from the Programmer to the board and the window should appear similar to figure 10.

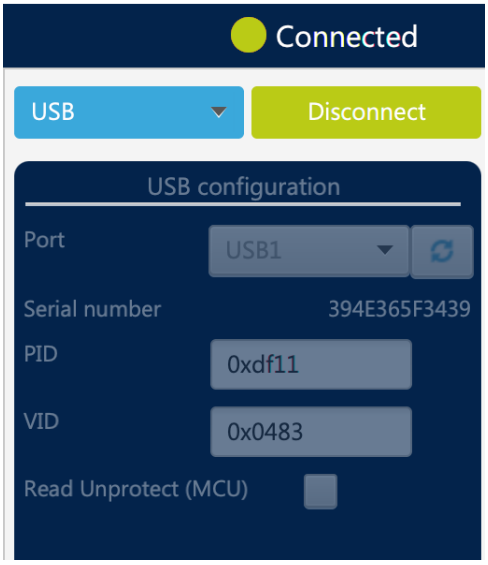


Figure 10: USB Connected!

Next, click "Open File" towards the top of the "Memory & File edition" window and select the program binary you want to program the board with.

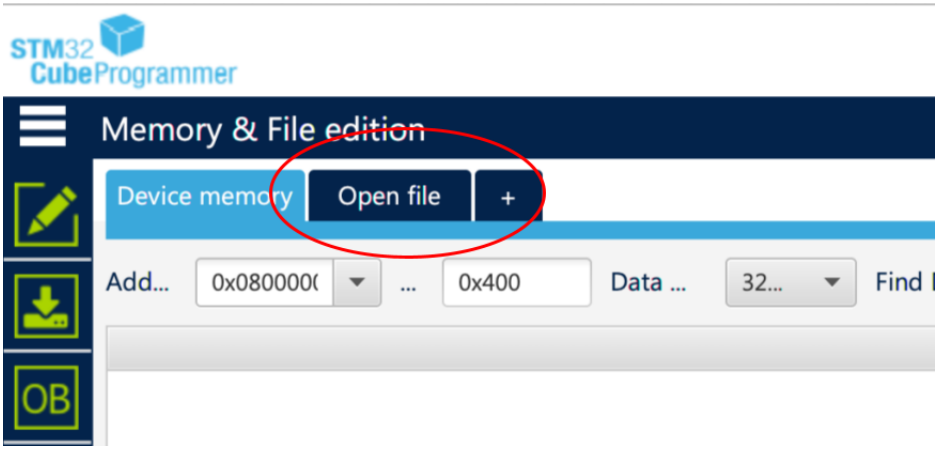


Figure 11: Open Binary File

Next navigate to the "Erasing & Programming" window using the left panel, and click the green "Start Programming" button as in figure 12.

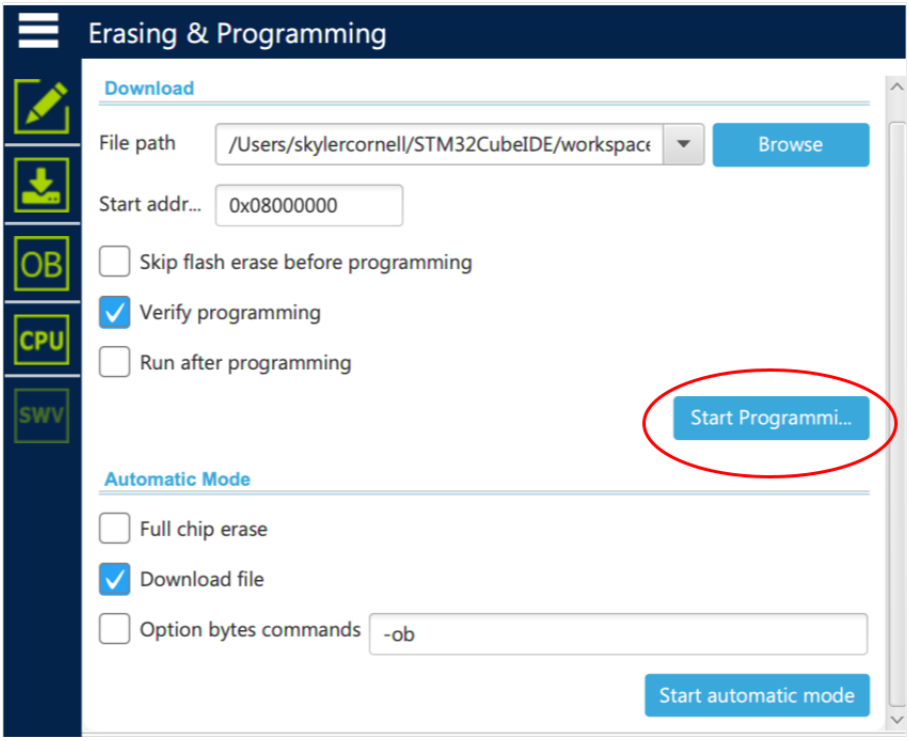


Figure 12: Program the Board

If all goes well you should receive a pop-up window confirming that the program download was successful. The next step is get the microcontroller out of bootloader mode, by disconnecting it from power and restoring the boot pin jumpers so that both BOOT0 and BOOT1 are pulled low (positioned to the right). When the board is powered up again, the microcontroller will run the new program.

4 Writing Your First Program

Before getting started it is necessary to download and install STM32CubeIDE, the code development studio for working with STM32 microcontrollers. The download can be found at

https://www.st.com/en/development-tools/stm32cubeide.html . If you simply want to start writing your audio DSP application then you can download the starter boiler plate template from the project github repository and follow along from "4.2 Your first DSP application." Otherwise you may continue along from "4.1 Setting up a Starter Code Template" to learn how to properly set up the embedded system from scratch.

4.1 Setting up a Starter Code Template

By the end of this tutorial you will have set up a code template that can be the starter code you use for most if not all of your future DSP applications.

1. Make a new STM32CubeIDE project, select board as STM32F407VG and name your project.

2. Configure I2S peripheral

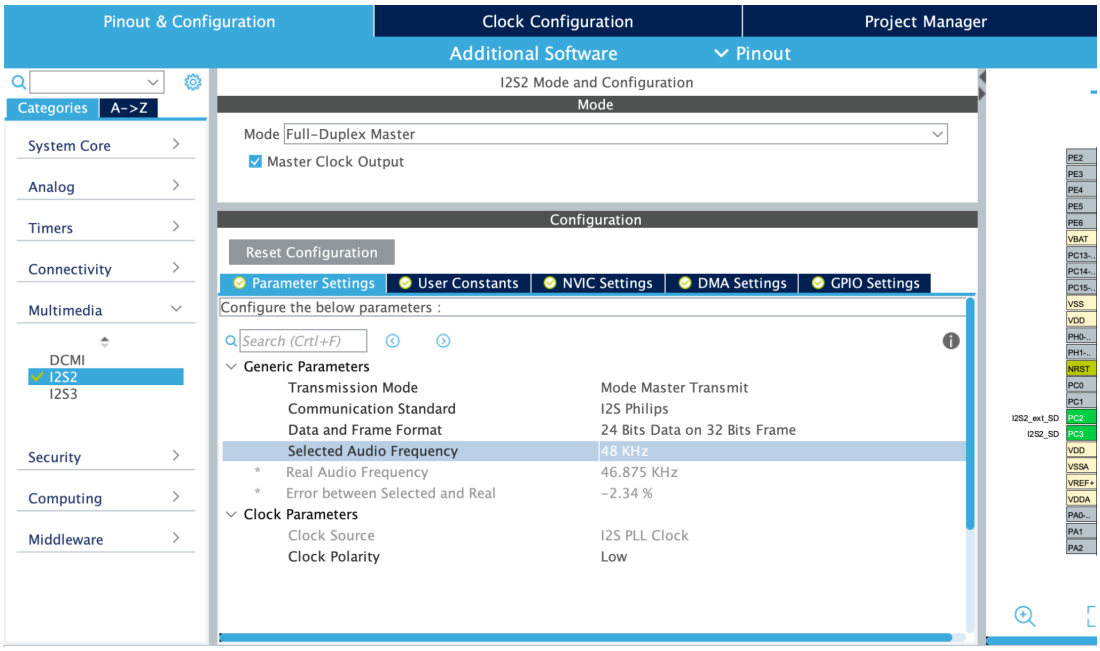


Figure 13: Step 2: Configure I2S Parameter Settings

In the Multimedia panel select the I2S2 unit, select Mode: Full-Duplex Master and enable Master Clock Output. The Master clock output is provided to the onboard audio conversion

chips which operate optimally with a master clock. Under Generic Parameters select 24 bits data on 32 bits frame. Select your desired sampling frequency, 48kHz is typical.

3. Configure I2S Direct Memory Access (DMA)

   Under DMA Settings of the I2S panel have the following configurations set for both I2S_EXT_RX and SPI2_TX. Circular mode is essential for the double buffer audio stream. Data Width must be selected as half word so that the DMA transfers 16 bits at a time to/from the I2S data registers. Increment Address for memory should be checked and peripheral increment unchecked. This enables the DMA to automatically transfer the next data incoming/outgoing half word to/from the next address of the global memory arrays in main.c.

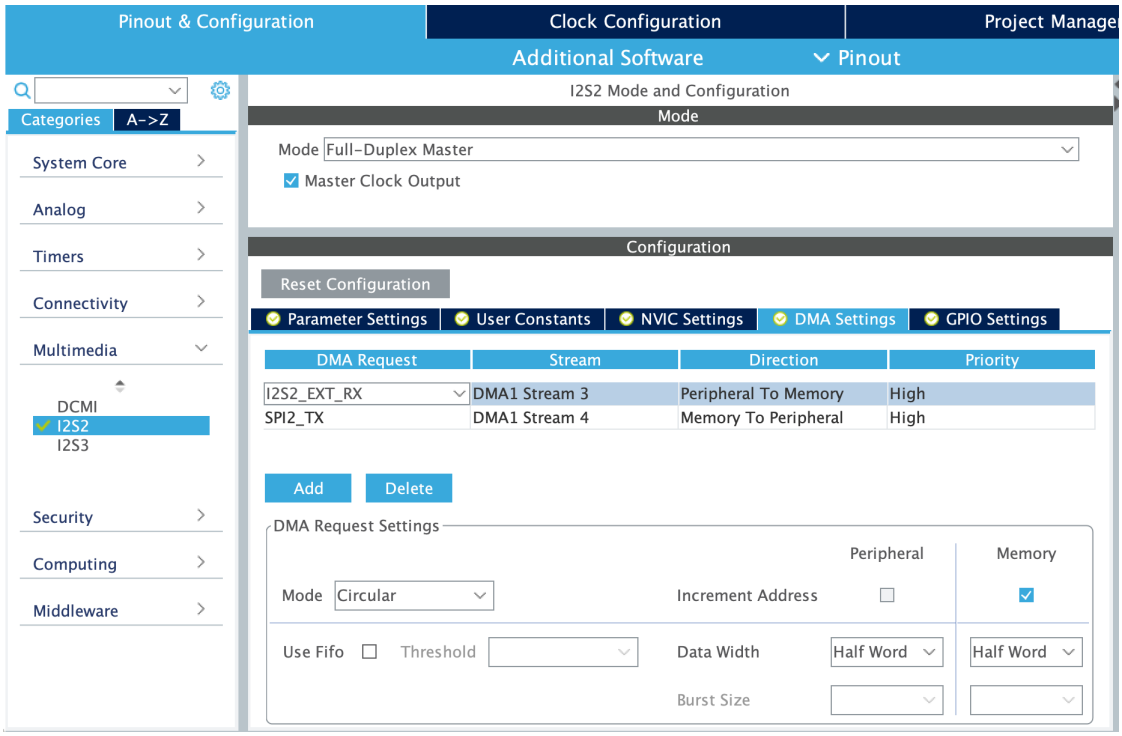


Figure 14: Step 3: Configure I2S DMA

This DMA config can be done separately in the the System Core - DMA panel but it's convenient and equivalent to do it directly in the I2S configuration panel.

4. Enable external crystal and configure peripheral clocks

   The board was designed with a 16MHz external crystal oscillator which can provide a stable clock source from which to derive the other system clocks. In the System Core panel select RCC

to enable the crystal oscillator. Select the Crystal/Ceramic Resonator option for High Speed Clock (HSE)

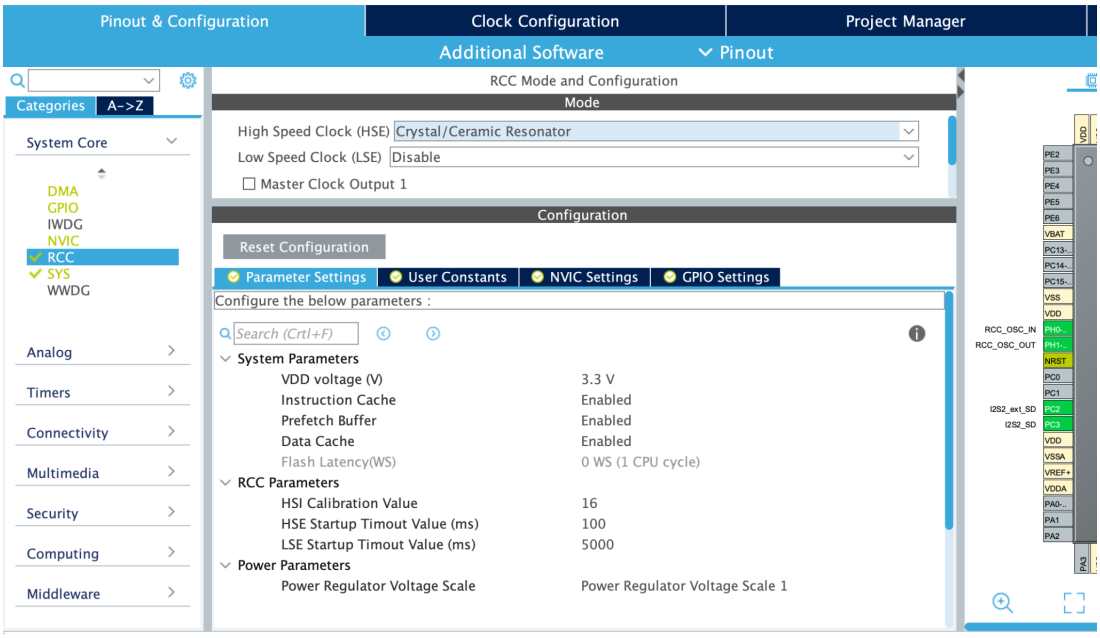


Figure 15: Step 4a: Enable HSE external crystal

Under Clock Configuration, set the clock tree to the following:

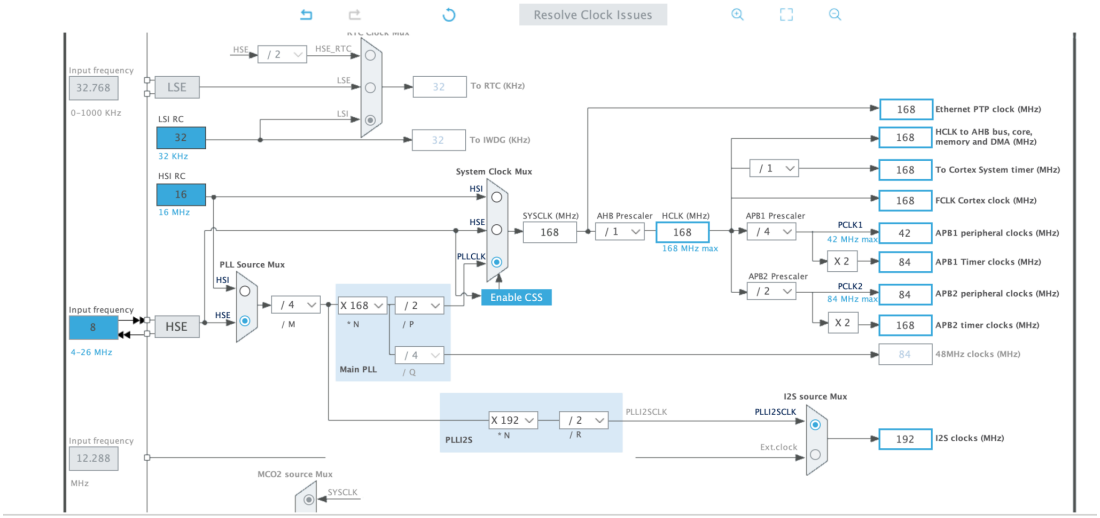


Figure 16: Step 4b: Configure the clock tree

This derives the main system clock (SYSCLK) from the 16MHz external crystal oscillator configures the internal PLL's to step up the frequency so the microcontroller can operate at its

maximum of 168 MHz, and provides a clock source of 192 MHz to the I2S peripheral.

5. Click on the gear to generate the configuration code we just set up, and then open up main.c

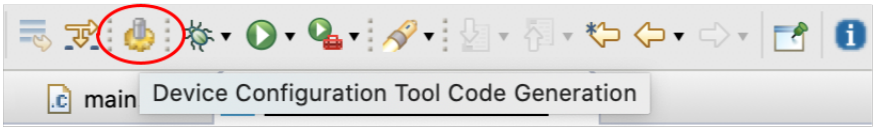


Figure 17: Step 5: Generate the Code

At this point, we have leveraged the use of the CubeMX tool, to easily configure all of the microcontroller's peripherals without even writing any code yet.

6. In main.c comment out DMA_Stream4 Interrupt enable lines

```
195 /**
196   * Enable DMA controller clock
197   */
198 static void MX_DMA_Init(void)
199 {
200
201   /* DMA controller clock enable */
202   __HAL_RCC_DMA1_CLK_ENABLE();
203
204   /* DMA interrupt init */
205   /* DMA1_Stream3_IRQn interrupt configuration */
206   HAL_NVIC_SetPriority(DMA1_Stream3_IRQn, 0, 0);
207   HAL_NVIC_EnableIRQ(DMA1_Stream3_IRQn);
208
209   /* DMA1_Stream4_IRQn interrupt configuration */
210  // HAL_NVIC_SetPriority(DMA1_Stream4_IRQn, 0, 0);
211  // HAL_NVIC_EnableIRQ(DMA1_Stream4_IRQn);
212
213 }
```

Figure 18: Step 6: Disable Interrupts for TX DMA line

Comment out or delete the generated code for NVIC enable calls for the DMA stream associated with the TX DMA line, in the MX_DMA_Init() function. We only want the CPU to jump into an ISR when receiving new data, and not when we have completed sending out processed data.

7. Declare double buffer global variables

Declare global unsigned 16 bit int arrays for the double buffer scheme, rx_buf[4] and tx_buf[4]. The samples are transferred by the DMA by half-word (16 bits), so each 24 bit sample is formed

as a composite of two 16 bit integers.

Also globally declare the two integers to store in_sample (variable for unprocessed sampled), and out_sample (variable to place the processed sample).

```
59  /* USER CODE BEGIN PV */
60
61
62  uint16_t rx_buf[4];
63  uint16_t tx_buf[4];
64  int in_sample, out_sample;
65
```

Figure 19: Step 7: Declare double buffers

8. Establish DMA and I2S Data Transfers

   Add the function call HAL_I2SEx_TransmitReceive_DMA(&hi2s2, tx_buf, rx_buf, 2); above the while(1) loop. This is one of the many functions defined by ST's HAL (hardware abstraction layer). The role of this function call is to establish the full duplex data transfer between the I2S unit and the user defined tx_buf and rx_buf. The data transfers (see "2.4.1 Double Buffer Data Stream" for details) commence each time the I2S unit generates an event signal each sampling period.

```
132   MX_TIM2_Init();
133   MX_TIM3_Init();
134   MX_TIM4_Init();
135   MX_USB_OTG_FS_PCD_Init();
136   /* USER CODE BEGIN 2 */
137
138   HAL_I2SEx_TransmitReceive_DMA(&hi2s2, tx_buf, rx_buf, 2);
139
140
141   /* USER CODE END 2 */
142
143   /* Infinite loop */
144   /* USER CODE BEGIN WHILE */
145   while (1)
146   {
147     /* USER CODE END WHILE */
148
149
150     /* USER CODE BEGIN 3 */
151   }
152   /* USER CODE END 3 */
153 }
```

Figure 20: Step 8: Establish DMA and I2S Full Duplex Data Transfer

9. Add extra line in HAL's I2SEx_TxRxDMACplt()

   Unfortunately there is a small bug in the ST HAL library that prevents the DMA transfer complete interrupt from firing when the DMA is configured in circular mode so it is necessary to add a line of code to I2SEx_TxRxDMACplt() to get around it. Under the project explorer tab in the IDE, navigate to

   Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_i2s_ex.c and finally to the I2SEx_TxRxDMACplt() function definition. At the very bottom of the function add a call to HAL_I2SEx_TxRxCpltCallback(hi2s); and save the file. The bottom of function should look like:

```
930         /* Call user TxRx complete callback */
931 #if (USE_HAL_I2S_REGISTER_CALLBACKS == 1U)
932         hi2s->TxRxCpltCallback(hi2s);
933 #else
934         HAL_I2SEx_TxRxCpltCallback(hi2s);
935 #endif /* USE_HAL_I2S_REGISTER_CALLBACKS */
936       }
937     }
938   }
939   HAL_I2SEx_TxRxCpltCallback(hi2s);
940 }
941
```

Figure 21: Step 9: Fix a bug in the HAL library

10. Define the new sample interrupt service routine

    Now, define the DMA transfer half-complete/complete interrupt routines HAL_I2SEx_TxRxHalfCpltCallback() and HAL_I2SEx_TxRxCpltCallback() in main.c. In these function definitions you can write DSP to process new samples, or better yet make function calls to a separate DSP routine, for the sake of software abstraction. These two callback functions are weakly defined and declared in the HAL library, which is why you can simply define them here in main.c without needing make a function declaration at the top of the file.

    The DMA unit transmits the 24 bit samples on 16 bit frames, so in these interrupt routines you must form the full sample by bit wise masking the separate frames together. in_sample is shifted right by 8 in order to compensate for storing a 24 bit value in a 32 bit integer. Within the routines, write a function call to process() which we have not defined yet. Process() should take a pointer to the input sample and a pointer to the output sample and return void. x

```
635 //I2S RX line DMA transfer half complete callback
636 void HAL_I2SEx_TxRxHalfCpltCallback(I2S_HandleTypeDef *hi2s)
637 {
638   //retrieve sample from rx_buf
639   in_sample  = (((int)rx_buf[0]<<16)|rx_buf[1])>>8;
640
641   // do audio processing
642   process(&in_sample, &out_sample);
643
644   //place processed sample into tx_buf
645   tx_buf[0] = (out_sample>>8) & 0xFFFF;
646   tx_buf[1] = out_sample & 0xFFFF;
647
648
649 }
650
651 //I2S RX line DMA transfer complete callback
652 void HAL_I2SEx_TxRxCpltCallback(I2S_HandleTypeDef *hi2s)
653 {
654   //retrieve sample from rx_buf
655   in_sample  = (((int)rx_buf[2]<<16)|rx_buf[3])>>8;
656
657   // does audio processing
658   process(&in_sample, &out_sample);
659
660   //place processed sample into tx_buf
661   tx_buf[2] = (left_out>>8) & 0xFFFF;
662   tx_buf[3] = left_out & 0xFFFF;
663
664 }
665
```

Figure 22: Step 10: Define Interrupt Service Routines

11. Create DSP Abstraction Files

Under the project explorer tab of the IDE, create a new source file "audio_process.c" in Core/Src and a header file "audio_process.h" in Core/Inc. Define two function in audio_process.c, void process_init() and void process(int* in_sample, int* out_sample). The role of process_init() is to execute once before entering the while(1) loop. This is synonymous with the setup() function in Arduino. process() is where the DSP action happens, it processes the incoming sample and places the result in the out_sample variable by way of pointer de-referencing.

Make sure to declare void process(int* in_sample, int* out_sample) and void process_init() in the header file. Also be sure to put a #include "audio_process.h" at the top of main.c so that the functions in audio_process.c are visible to main.c.

That's it! You have now created a fully functional boiler plate audio DSP project. The next step is to program DSP in the process() function. Writing you first real time DSP application is covered in the quite brief next section.

4.2 Your first DSP application

The following steps are designed to walk you through setting up your minimal DSP application, we'll call it "passthru" which as the name suggests will simply set the output signal to be same value as the input signal. After establishing the boilerplate and testing with passthru, you can start programming more involved DSP applications.

The code for passthru is as simple as *out_sample = *in_sample;

```
/*
 * audio_process.c
 *
 */
#include "audio_process.h"


/*
 * Declare global variables and structures units here
 */


void process_init()
{

}


void process(int *in_sample, int* out_sample)
{

  //Passthru
  *out_sample = *in_sample;


}

```

Figure 23: Passthru Example

The beauty of abstraction is that after setting up the starter template code, future applications can operate without any need for peripheral configuration. In fact, with the exception of using the potentiometers or LEDs, your DSP programs will be entirely written outside of main.c.

4.3 Programming the Board via SWD Debugger

It is recommended for developers programming applications on the E90 board to upload programs using a designated external Serial Wire Debugger, which provides a host of necessary debug tools in the STM32CubeIDE development environment. SWD (Serial Wire Debug) is a slow speed 2 wire hardware protocol that transfers data directly between a host computer and the internal memories of an STM32 micrcontroler. SWD is a proprietary STMicroelectronics hardware protocol which was designed to fill the role of the traditional J-Tag external debuggers.

A SWD Debugger such as the ST-Link V2 augments the DSP development experience by adding the ability to add breakpoints and view internal registers and program variables in real time.

For users of the E90 board who simply want to upload a new audio effect to the board, they may

do so by placing the board into bootloader mode by changing the boot mode jumpers on the board to Boot0 = + and Boot1 = - and using the STM32CubeProgrammer tool to transfer the compiled binary (.bin file) to the board over the USB connection. This process is outlined in "3 Getting Started (No Code)" in detail. This method provides no access to the system in the development environment and should be thought of as a firmware upgrade.

While the SWD protocol is 2 wire in the sense that there is a clock and data line, when wiring the debugger up to the board it is necessary to connect their ground wires as well as provide the debugger a 3.3V line from the board. The E90 board must be externally powered when debugging because the debugger does not power the E90 board with the 3.3V line but instead reads that voltage in order to set the proper logic levels of the data and clock signals it generates.

There is an optional SWO (Single Wire Output) wire on the Serial Wire Debugger connection which enables you to write standard I/0 commands in C like printf(), however this feature is not explored in this manual.

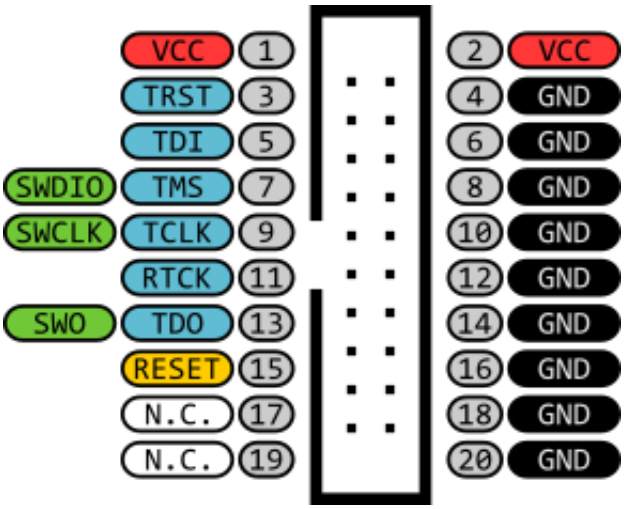


Figure 24: ST-Link Debugger Pinout

Connect the board's programming pins (SWDIO, SWCLK, VCC, GND) which are labelled on the board's silkscreen to the corresponding pins of the debugger. In this example, I am using the ST-Link debugger which has a pinout according to figure 24. and highlights the SWD pins in green.

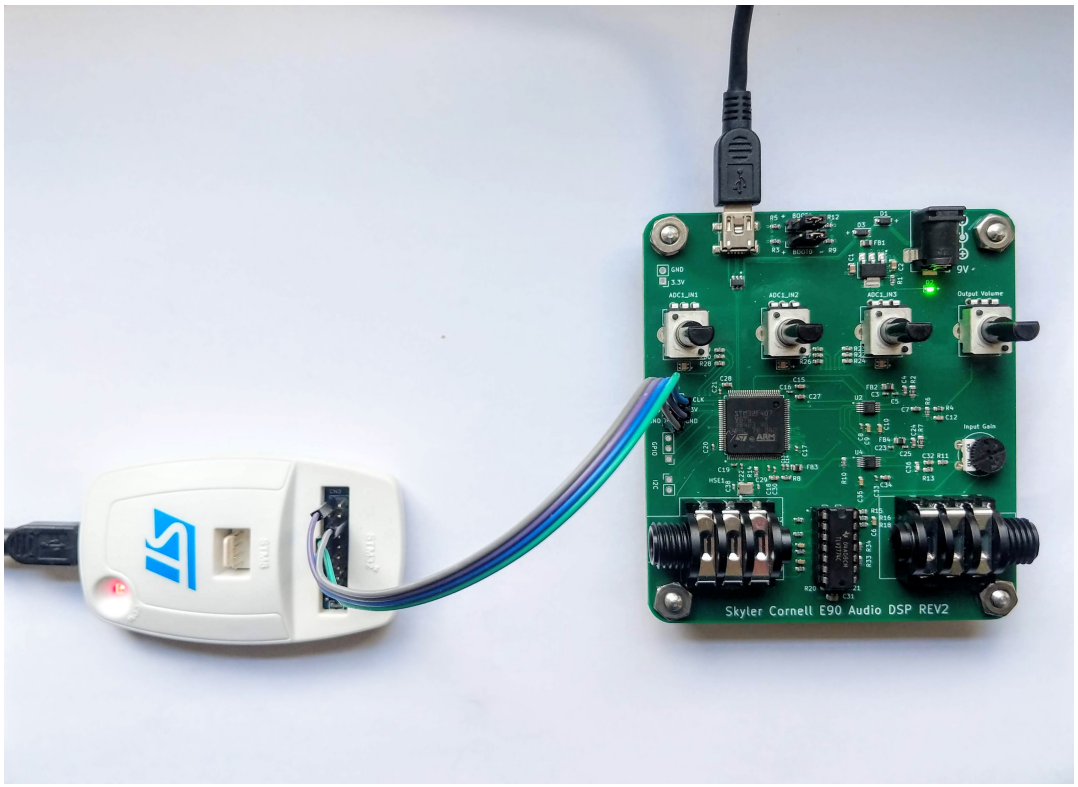


Figure 25: Programming the Board via SWD

5 Digital Signal Processing Software Framework

The software framework has been designed so that if the programmer only wants to program DSP, they need only work in "audio_process.c" in the audio_process() function. Of course they are free to create new source and header files for their applications, but process() is where the action happens. The process() function has been set up in main.c to be called every time a new sample is available to process. process(int *in_sample, int *out_sample) is provided with pointers to the input samples and the output sample where the programmer should de-reference and store the processed sample. The double buffer data stream is already implemented in main.c when startig from the starter template code, so a programmer's DSP need only be written in audio_process.c.

A more robust DSP software framework could be developed to include a library of utility functions and pre-tested audio effects modules to be used in a signal chain. The following figures shows the power of code abstraction, in that the DSP code in process() could be designed entirely by signal chaining pre-written effects defined in other source files.

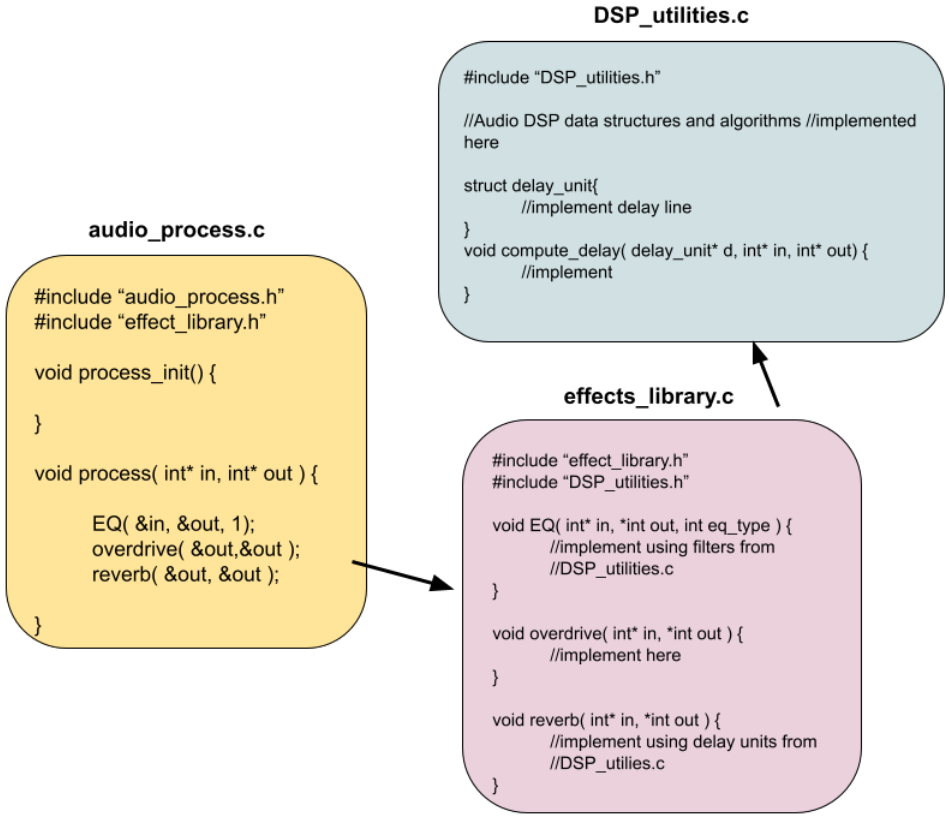


Figure 26: DSP Framework

In the example above, pointers to the out sample are passed into and out of the function calls forming a series signal chain. The resulting signal chain looks like:

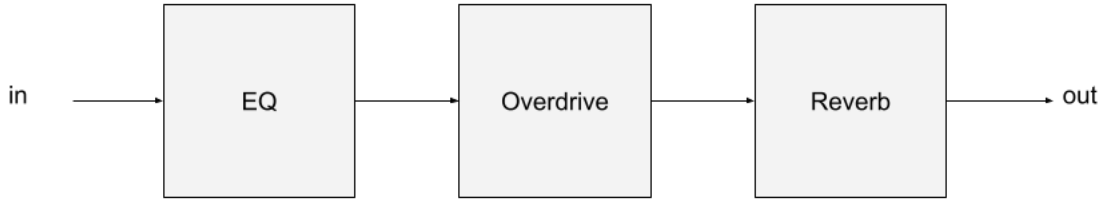


Figure 27: Signal Chain

It is easy to see how abstracting lower level DSP functions, and even audio effects allows the possibility of designing complex signal chains through the use of reusable code.

6 Board Schematic and PCB

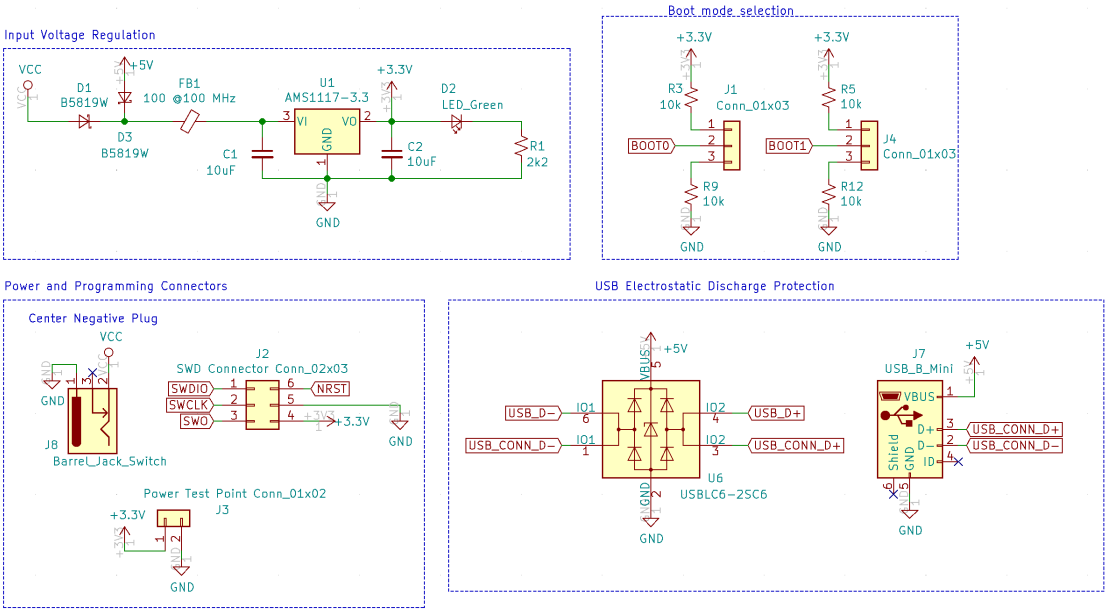


Figure 28: Power and Programming

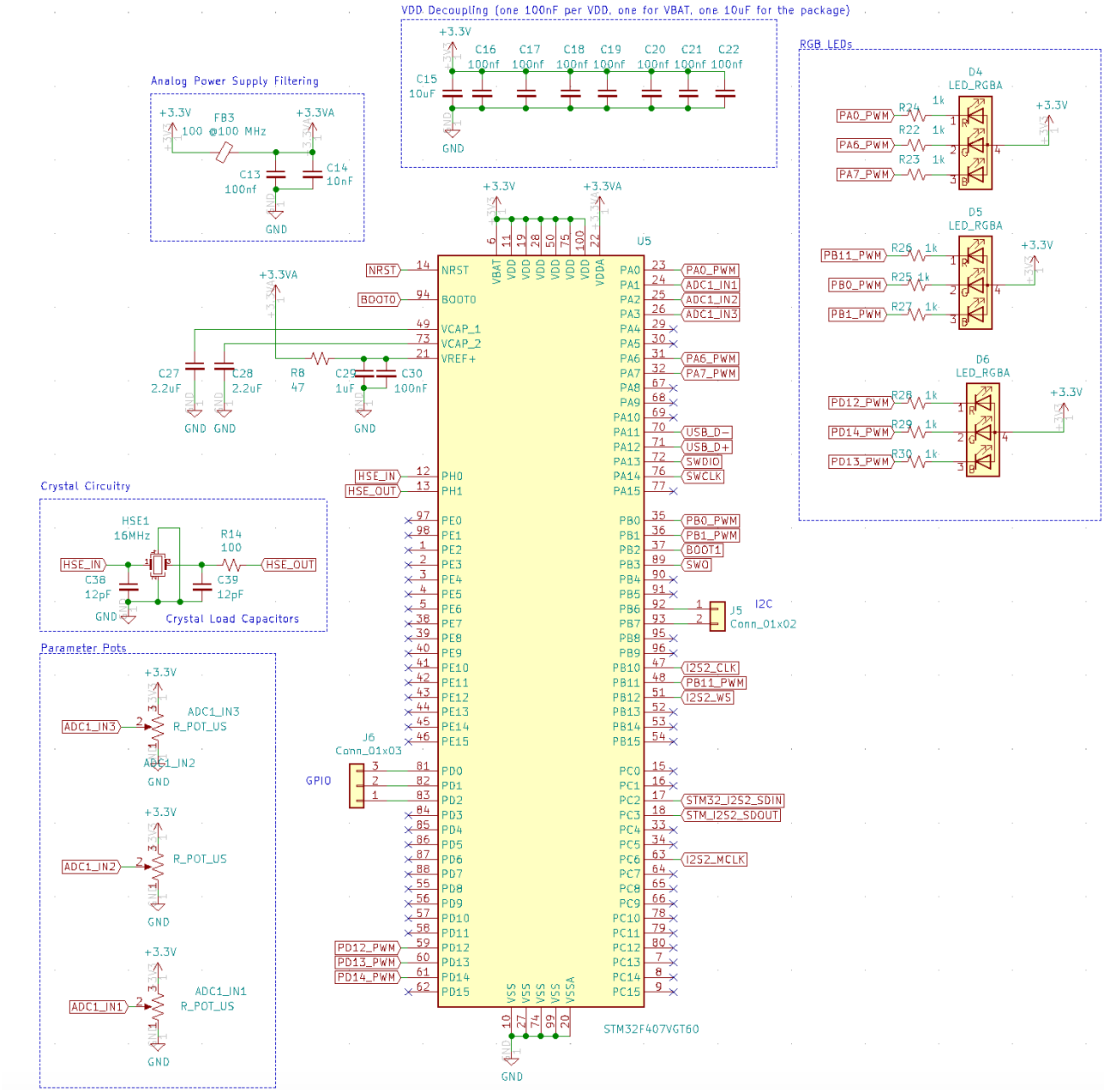


Figure 29: STM32 Microcontroller

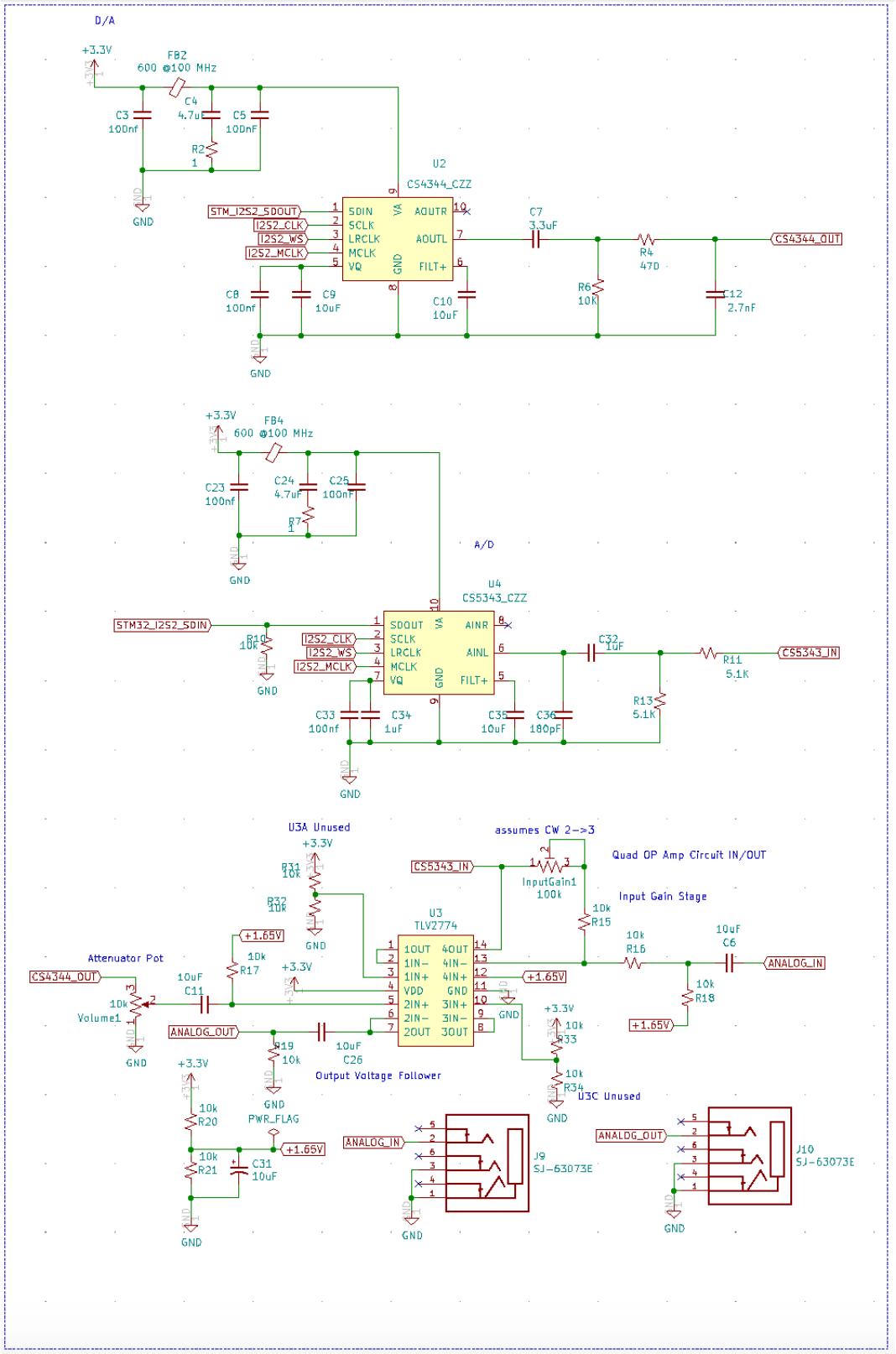


Figure 30: Audio In/Out

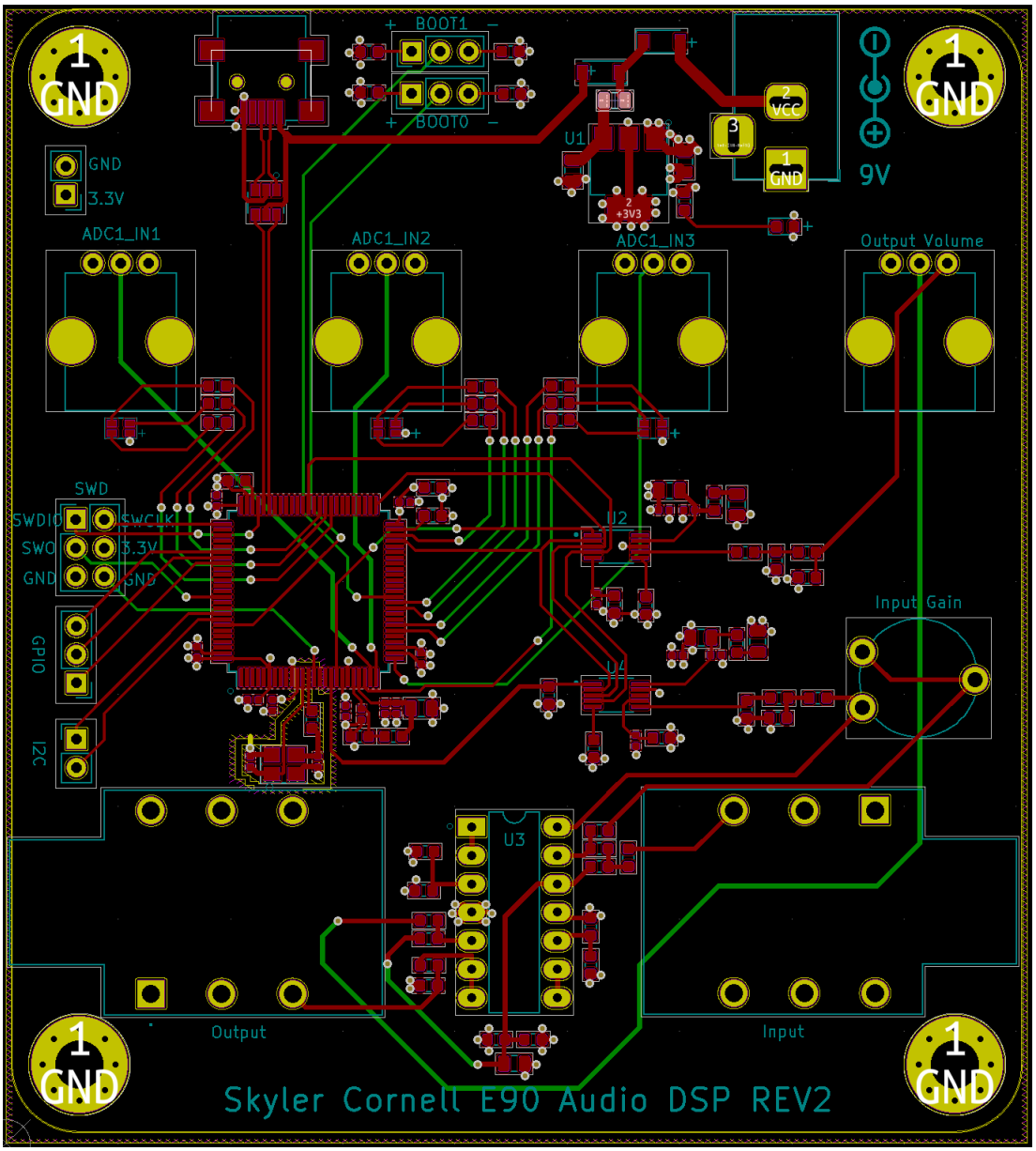


Figure 31: Revision 2 PCB Layout

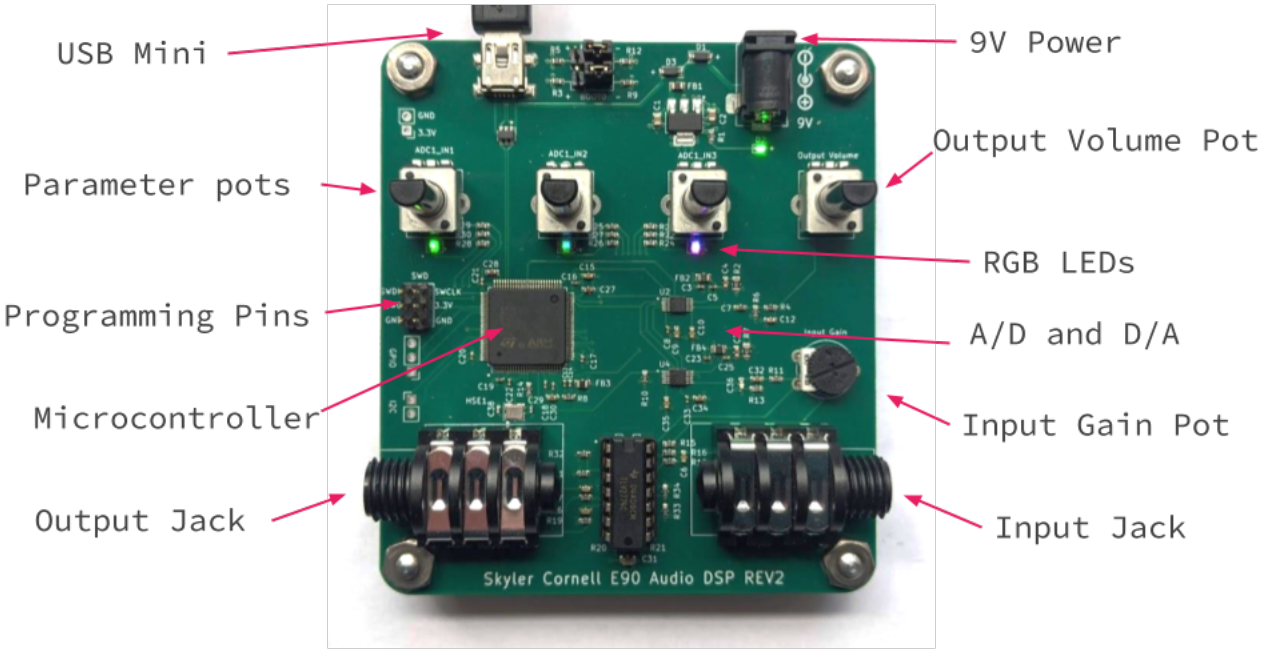


Figure 32: Revision 2 Board Layout

7 Example Program: IIR Biquad Bandpass Filter

To help orient new developers to the E90 board framework we will implements a fixed frequency IIR bandpass filter from a biquad coefficient cookbook. The coefficient calculation formulas used are taken from the legendary online audio biquad cookbook written by Robert Bristow-Johnson, which can be found at https://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.html . The cookbook approach to filters gives us the equations to compute the filter coefficients for given parameters such as center frequency ω_0 and quality Q. According to the cookbook site, this particular bandpass filter's coefficients are given by figure 33.

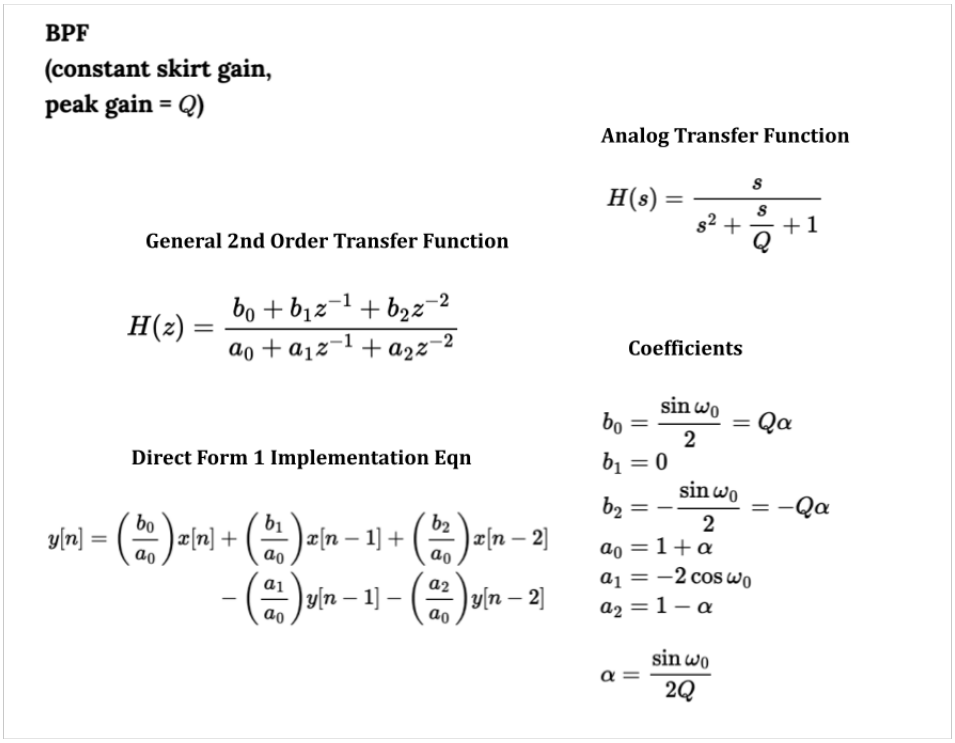


Figure 33: Bandpass Filter Biquad Cookbook Formula

For this filter implementation we will need to define a C struct for maintaining biquad values such as the previous input and output samples and the filter's coefficients. We will also need to define two functions: one for computing the biquad coefficients from the cookbook equations, we'll call it compute_BPF_coeff(), and another to apply the filter to an input sample which we will call compute_biquad().

Define the biquad struct to maintain 2 previous inputs and 2 previous outputs as well as the 6

biquad coefficients b0,b1,b2,a0,a1,a2. Declare the function pointers in audio_process.h, which should look something like this:

```
/*
audio_process.h
*/
typedef struct {

  float xn1, xn2;  //previous inputs
  float yn1, yn2;  //prev outputs

  //coefficients
  float b0,b1,b2,a0,a1,a2;

}biquad_t;

void process_init();
void process(int *in_sample, int *out_sample);

void compute_BPF_coeff(biquad_t *bq, float w0, float Q);
void compute_biquad(biquad_t *bq, float *in, float *out);
```

Next we must define the compute_BPF_coeff() and compute_biquad() functions in audio_process.c. void compute_BPF_coeff(biquad_t *bq, float w0, float Q) takes a biquad_t struct pointer, and the filter parameters ω_0 and Q to compute the biquad coefficients and save them into the biquad_t coefficient struct fields.

We must define the global biquad struct after the includes in audio_process.c which will be used and updated each sample period. We must also add a function call to compute_BPF_coeff(&BPF_biquad, w0, Q) in process_init() with the ω_0 and Q values set to our specifications. In this example I set a Q of 6 and a center frequency of 1 kHz. Because this example is for a fixed frequency filter, compute_BPF_coeff() need only be called once at the beginning of the program, hence its place in process_init(). audio_process.c should look something like:

```
/*
 * audio_process.c
 */

#include "audio_process.h"
#include <math.h>

/*
 * Declare global variables and structures units here
 */
biquad_t BPF_biquad;

float F0;
float Fs = 48000;
float w0;
```

```
float Q;

/*
 * Executes once before entering program loop
 */
void process_init()
{
  //reset biquad history
  BPF_biquad.yn2 = 0;
  BPF_biquad.yn1 = 0;
  BPF_biquad.xn2 = 0;
  BPF_biquad.xn1 = 0;

  F0 = 1000; // 1 kHz center freq;
  w0 = 2*3.141592*(F0/46875); //compute normalized freq
  Q = 6;

  //compute the fixed set of biquad coefficients
  compute_BPF_coeff(&BPF_biquad, w0, Q);
}

void process(int *in_sample, int *out_sample)
{
  float in = (float)*in_sample;
  float out = 0;

  //produce filtered output sample
  compute_biquad(&BPF_biquad, &in, &out);

  //cast float to int
  *out_sample = (int)out;
}

void compute_BPF_coeff(biquad_t *bq, float w0, float Q){
  float sinw0_over2 = sin(w0)/2;
  float alpha = sinw0_over2/Q;;

  bq->b0 = sinw0_over2;
  bq->b1 = 0;
  bq->b2 = -sinw0_over2;

  bq->a0 = 1+alpha;
  bq->a1 = -2*cos(w0);
  bq->a2 = 1-alpha;

}

void compute_biquad(biquad_t *bq, float *in, float *out){

  float b0 = bq->b0;
  float b1 = bq->b1;
  float b2 = bq->b2;

  float a0 = bq->a0;
  float a1 = bq->a1;
  float a2 = bq->a2;
```

```
  // Direct Form 1 Biquad Implementation
  *out = (*in)*(b0/a0);
  *out += bq->xn1*(b1/a0);
  *out += bq->xn2*(b2/a0);
  *out -= bq->yn1*(a1/a0);
  *out -= bq->yn2*(a2/a0);

  //update filter history
  bq->yn2 = bq->yn1;
  bq->yn1 = *out;
  bq->xn2 = bq->xn1;
  bq->xn1 = *in;
}
```

The next thing to do is upload to the board and measure the output. The results of this simple filter implementation are encouraging as the measured and theoretical frequency response curves are close.

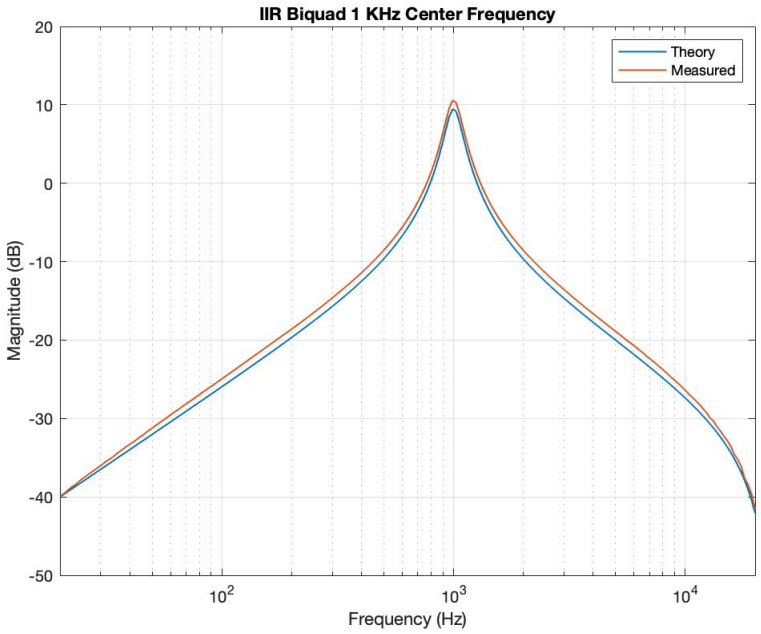


Figure 34: Biquad Frequency Response

8 Sanity Checks

When getting started programming applications on the E90 board, it's useful to know some simple checks to verify everything is running as it should. Conversely, if your applications aren't working properly, it might be wise to do these sanity checks to try to pinpoint why they are failing. All of the code for the following sanity checks as well as their compiled binaries can be found on the E90 project github repository.

1. Power on LED

   The first sanity check should always be the green ON LED just beneath the DC power connector which indicates that the board is successfully powered.

2. Blink program

   The next check should be a program to blink one of the LEDs. Passing the blink program check indicates that the board is in fact programmable.

3. Audio passthru

   An audio pass through test indicates that all of the analog circuity on the board is working, as well as the underlying audio data flow software.

4. Oscillator (no input needed)

   The oscillator program generates a pure sin wave without any need for an input signal. If something is wrong with your application, or the board is failing passthru tests, generating and measuring a pure sine wave output would indicate that the microcontroller and output circuitry are fine, but something is wrong with the input processing circuit.

9 Sources

1. STM32F4 Discovery User Manual

   https://www.st.com/resource/en/user_manual/dm00039084-discovery-kit-with-stm32f407vg-mcu-stmicroelectronics.pdf

2. STM32F407 Datasheet

   https://www.st.com/resource/en/datasheet/stm32f405rg.pdf

3. STM32F407 User Manual

   https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405-415-stm32f407-417-stm32f427-437-and-stm32f429-439-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf

4. SWD Protocol

   https://research.kudelskisecurity.com/2019/05/16/swd-arms-alternative-to-jtag/

5. GPIO Modes Info

   https://www.google.com/url?q=http://fastbitlab.com/output-configuration-gpio-pin-open-drain/&sa=D&source=editors &ust=1621488066075000&usg=AOvVaw2pYEhc0uOwS2vyX7tJ2fgx

6. Guide to STM32

   https://predictabledesigns.com/STM32_UltimateGuide.pdf

7. ST HAL Library User Manual

   https://www.st.com/resource/en/user_manual/dm00105879-description-of-stm32f4-hal-and-ll-drivers-stmicroelectronics.pdf

8. GPIO Alternal Functions Info

   http://www.se.rit.edu/ swen-563/resources/STM32L476/GPIO

9. Owl Digital Pedal

   https://github.com/pingdynasty/OwlWare

10. Youtube Channel: Phil's Lab

    https://www.youtube.com/channel/UCVryWqJ4cSlbTSETBHpBUWw

11. Youtube Channel: YetAnotherElectronicsChannel

https://www.youtube.com/channel/UCwOkALY6oQbOL1zU7ApaPHg