

Violin Tutor Device

By Victor Tran

Introduction

For this project, I am using the STM32L476RG Nucleo Board along with FREERTOS to create an embedded violin instrument. The goal of this project is to teach user's how to play or practice on a violin without going through the hassle of buying one. The system application runs on two modes: Human Mode and Computer Mode. During the Human Mode, notes can be played by the user by putting a finger on a particular location of a SoftPot and holding down a push button. During Computer Mode, the embedded device itself will play a note of a song at a precise pitch. The goal is for the human to attempt to replicate that note. The system relays back and forth between Human Mode and Computer Mode and loops through the song. The synchronization is done using semaphores.

I used FreeRTOS and the Nucleo Board to create a time-critical application, which requires audio processing. While I am not sampling input audio on this system, I am sampling ADC values that need to be filtered and I am outputting audio samples with an DAC. In order to achieve high-quality sounds, with little-to-no jitter, and at the correct frequency up to a few thousand Hz, I need to use an RTOS. This is especially true because outputting audio is only one portion of the system. The CPU will be occupied with outputting feedback to the user using a console, transferred with UART. The CPU will also be occupied with button presses, lighting up external LEDs, sampling the ADC, converting samples to the target Hz of a violin, and so on. Although FreeRTOS handles the software scheduling of these tasks, I also learned how to use the Direct Memory Access (DMA), which saves CPU processing power for more important tasks.

With this project, I wanted to get a hands-on approach with an enterprise operating system used in the industry. This complements what we have done in class because I am taking the next step from building my own rudimentary RTOS, to using a more sophisticated one. As I completed this project, I still noticed many basic fundamental similarities between FreeRTOS and my custom RTOS coded in EGCP 460. For instance, the scheduler may block tasks that are waiting on a mutex or variable. Critical regions are used to make multiple lines of code atomic. In addition, the scheduler can put tasks to sleep mode for a fixed amount of time. This is something we have not implemented in class. While there are other real-time operating systems to choose from such as VxWorks and Zephyr, I chose FreeRTOS because it is free and has a smaller initial learning curve. This project shows how important an RTOS is for embedded devices, especially because chips are compact and have limited memory. The board I bought cost approximately \$15 dollars, and the majority of that amount is from the Cortex M4 processor. However, I learned that I could still make a sophisticated system for that small amount of cost and limited space. RTOS are more application-specific and cheaper to use with hardware as opposed to a general-purpose operating system.

Design Process

Building this project required me to quickly learn two new objectives: interfacing with the Nucleo Board and coding with FreeRTOS. To do this, I followed several tutorials from Shawn Hymek and the Maker.io Staff of Digikey [1]. This part was not too difficult as they explained how to boot up the board using a USB A to USB mini-b cable and how to download the STM32Cube IDE. Watching these videos and following along took me approximately 3 hours. I completed watching the videos on May 1st. However, I only followed a subset of the tutorial to the point where I can simply blink an on-board LED using a FreeRTOS task. However, that is enough as I was confident from the knowledge gained from the class that I can use this barebones application as a springboard for my embedded violin application.

STM32CUBE IDE

The STM32Cube IDE provides a powerful user-friendly interface called the Device Configuration Tool (DCT). When creating a STM32CubeIDE project, I specified what type of board I was using. Based on that, the DCT's Pinout and Configuration Tab provides a visual and interactive pinout schematic of my board. To set a certain pin to a particular GPIO or special register, I just have to click on an unused pin and specify its mode. Then the STM32Cube IDE will automatically generate the initialization of the ports and pins for me. All I have to do is write the application code without needing to mull through datasheets about the hardware of this board. Of course, some of the pins are already pre-set such as one on-board LED, one on-board push button GPIO, one on-board reset button, an internal clock, and the Serial Wire Debug (SWO). The SWO is a two-wire protocol that allows me to flash the board with new code and use the debugger to set breakpoints, step through code, analyze specific registers, read watch expressions, and more. The DTC also enables me to configure the 10 usable timers with options such as prescaling and the auto reload register. These options help create periodic interrupts, which are used for debouncing buttons and calling the DMA particularly for this project. This IDE saved me a lot of time from writing custom code to configure the many pins and ports of the STM32L476RG microcontroller.

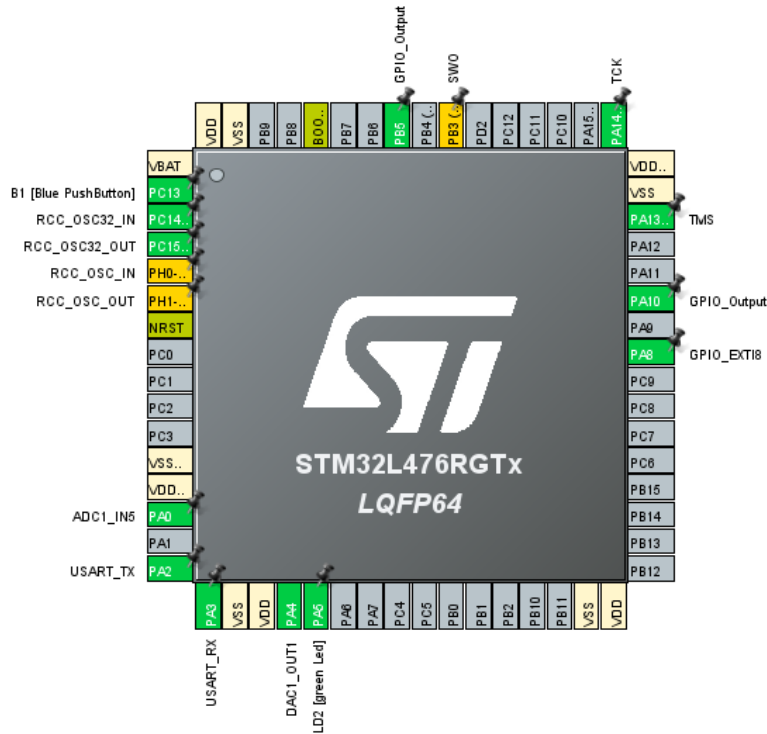


Figure 1: DCT Pinout and Configuration User-Interface

FREERTOS and CMSIS

Finally, the DCT provides the FreeRTOS middleware base code for the Nucleo Board. FreeRTOS is a real-time operating system that is designed to be light-weight enough to run on microcontrollers for embedded applications [2]. Real-time operating systems such as FreeRTOS are dedicated to creating applications that are guaranteed to respond to a certain event with a deterministic deadline. Since the Nucleo Board uses the ARM controller, FreeRTOS is abstracted in a software layer called the Common Microcontroller Software Interface Standard (CMSIS). This layer allows the same application code to be ported to other microcontrollers and boards by ARM. It includes a library of FreeRTOS functions and tools that are wrapped in CMSIS functions instead. The DCT provides an interface to create Tasks, Mutexes, Semaphores, and other FreeRTOS objects. For instance, creating a Task is as simple as specifying its name, its pre-defined priority levels, its stack size in words, and an entry function name. The code is automatically generated like the code for pin and port initialization, which further saved me time and headache. While the code is automatically generated, I still see similar parallels between the task structure of FreeRTOS and the thread structure designed in EGCP 460.

The Embedded Violin Application

The embedded violin application has three user-tasks and three interrupt request callbacks. When designing this application, I decided to separate the process to the three user-tasks. The first user-task is the Heartbeat Task. The Heartbeat Task is the lowest-priority task, which simply toggles the onboard LED every 100ms. This is done by simply calling a toggle pin function given by the Hardware Abstraction Layer (HAL) library and specified with the Port and Pin number. Then, I called a delay function that does not busy-wait, but rather suspends the tasks to sleep for 100ms. The concept of putting a Task to sleep is similar to blocking a task. The task will be put into a sleep state by the scheduler. It will “wake up” and be put in the ready state once 100ms of time slices has expired. Meanwhile, other tasks are free to be scheduled from the ready state to the active state. This task was easy to do because it was basically shown in the Digikey tutorial. I decided to keep this task as a visual verification that my application is not stalling. If the on-board LED was not blinking at a period of 100ms, then I immediately know that a task or interrupt is bottlenecking the system. I finished this part on May 1st.

The second user-task is the HumanPlay Task. This is a medium-priority task that waits on a PlayHumanSemaphore before it can proceed. The PlayHumanSemaphore is given/released by the third task, the Computer Play Task. This semaphore is essentially a mutex for setting the mode of the system. The system can be in two modes. The first mode is Human mode, where the human has control of playing the instrument. The second mode is Computer mode, where the computer takes control and plays preprogrammed notes. The HumanPlayTask switches the application to Human mode and allows the user to play note(s) until it switches to Computer mode. Also, this task prints the word “Play” to the computer console to instruct the user to play a note. This is sent through a UART function call. UART stands for Universal, Asynchronous Receiver-Transmitter. This is a two-wire protocol that allows messages in bits to be sent between two systems. The two systems must first agree on a baud rate to send, receive, and interpret the messages. The baud-rate I selected was 115200 bytes per second. I used Putty to connect to the board’s UART system in order to read the messages sent. This was also helpful in verifying that the task is currently running whenever I see the word “Play” in the console. One minor difficulty was seeing gibberish on the Putty console at first. This is due to me specifying an inconsistent baud rate in the Putty console. Once I changed the rate to 115200 to match the baud rate of the Nucleo Board’s UART, Putty received the correct messages. This was finished on May 5th.

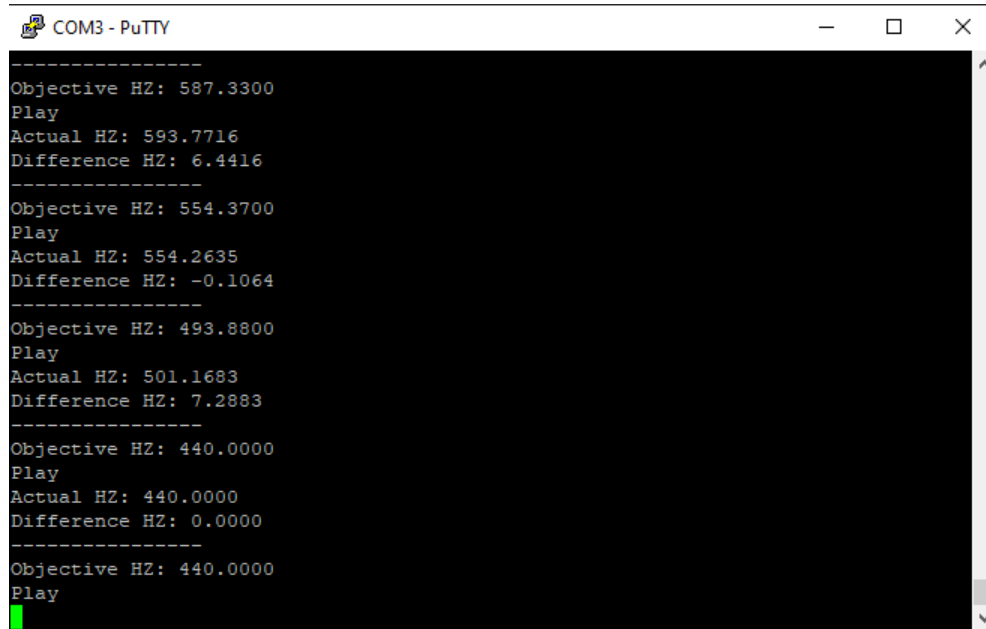
A screenshot of a PuTTY terminal window titled 'COM3 - PuTTY'. The terminal displays four sets of frequency measurements, each separated by a dashed line. Each set includes an 'Objective HZ' value, a 'Play' command, an 'Actual HZ' value, and a 'Difference HZ' value. The measurements are: 1) Objective HZ: 587.3300, Actual HZ: 593.7716, Difference HZ: 6.4416; 2) Objective HZ: 554.3700, Actual HZ: 554.2635, Difference HZ: -0.1064; 3) Objective HZ: 493.8800, Actual HZ: 501.1683, Difference HZ: 7.2883; 4) Objective HZ: 440.0000, Actual HZ: 440.0000, Difference HZ: 0.0000. The terminal ends with 'Objective HZ: 440.0000' and 'Play' on a new line with a green cursor.

Figure 2: I used the Putty console to provide a visual interface to the embedded system.

Also, the HumanPlay Task initially starts the ADC in DMA mode. The DMA unit is an essential unit for real-time embedded systems because it allows an input or output device to send or receive chunks of data in main memory without much CPU intervention [3]. I am using a SoftPot to act as a linear potentiometer. Its output voltage is calibrated by where I place my finger on the ribbon [4]. The ribbon is akin to a string on the violin. The output of the SoftPot is sampled by an ADC. However, this linear potentiometer creates noisy output. I could reduce the noise by adding an analog filter; however, I do not have the particular capacitor and resistor values to filter the high-frequency noise. Therefore, I created a basic digital filter using a simple averaging algorithm. This algorithm averages the sum of the 50 consecutively sampled ADC values. The DMA is needed here because I do not want the CPU to be occupied by transferring 50 samples to a buffer, one at a time. Instead, the 50 samples are automatically loaded consecutively to a pre-defined buffer in memory. Although gathering 50 samples does not use significant CPU processing power for this simple application, using the DMA allows this application to be scaled with more features and tasks in the future. Once the buffer is filled, the DMA unit sends an interrupt signal to the CPU, to signify that the ADC has completed its 50 samples.

In the ADC Callback function, I looked through the buffer and averaged the adc sample. Without the filtering, the pitches of the violin would change drastically even though my finger is seemingly stationary at a particular position in the ribbon. However, with the digital filter, there is reduced noise. The ADC Callback function is also responsible for setting the 80MHz timer of the DAC to control the pitch of the violin. The DAC sends a sine wave that is represented by a look-up-table (LUT) with 256 samples for a single period. This table holds the adc values that map to a sine-wave. The Nucleo Board has a 12-bit ADC, so we are mapping from 0 to 4095. To send a 1000Hz signal, which is roughly a C6 note, the sine wave must have a period of 1/1000 seconds. Since the LUT has 256 samples, each sample must be sent every $1/(256*1000)$

seconds. The DAC sends a signal for every Timer 4 overflow. The overflow is captured when the counter of the timer wraps over the auto-reload register (ARR) value and is set back to 0. Since the timer is not pre-scaled, it increments at a frequency of 80MHz. Depending on the target frequency, I need to adjust the ARR value. The formula for this is given in Equation 1 below.

$$ARR = \frac{f_{DAC}}{f_{Target} * LUT_LENGTH}$$

Equation 1: The auto reload register is determined by the frequency of output by the DAC. This is divided by the product of the target frequency and the LUT length.

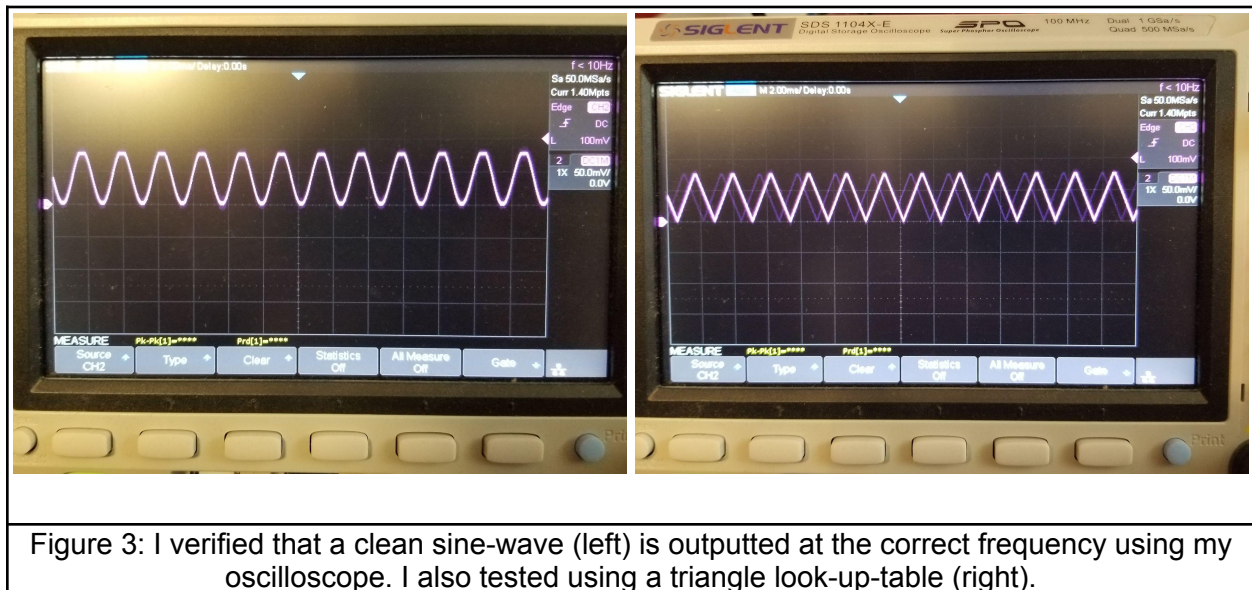


Figure 3: I verified that a clean sine-wave (left) is outputted at the correct frequency using my oscilloscope. I also tested using a triangle look-up-table (right).

Finally, based on the mode of the system, the DAC can either change its timer's ARR value to match the pitch of what the user is playing or to match the pitch of a given song encoded in the application. Learning how to use the DMA was difficult at first, but when I abstracted it to a simple callback function when the buffer is full, it became easy to use. To debug this, I had to flash the board in debugging mode and add a watch expression to the ADC sampling array. I had to verify that 50 samples were indeed collected when the ADC callback function was called and paused with a breakpoint. Also, during testing, I was not confident in trusting the SoftPot to give accurate sampling rates initially. I had never used a SoftPot before, so I did not know how precise it was. Instead, I tested with a simple twist-knob potentiometer. I printed the average ADC sample readings, updating at a rate of 100ms, to verify that the digital filter works. Once I was confident in my results, I switched out the twist-knob potentiometer with the SoftPot. I finished this section on May 10th.

The third user-task is the ComputerPlay Task. This is a high-priority task that also waits on a semaphore, the PlayComputerSemaphore because it can proceed. The PlayComputerSemaphore is given in the GPIO callback after the user lets go of the play button.

When it is the user's turn, he or she can place his or her finger on the SoftPot to adjust the pitch of a note. However, this alone will not play a sound. There is an external button using a pull-up resistor that needs to be pressed and held to register a note. Holding down the button is akin to bowing a violin string. The button is configured to register a GPIO callback for a falling or rising edge detection. However, the hardware button does not have a jitter filter or a schmitt trigger. Therefore, many spurious edge triggers when the button is pressed and released occurs. To deal with this, I did not create a delay within the callback function for debouncing. The callback/interrupt handlers are meant to execute as soon as possible. Instead, I set up a periodic timer interrupt using Timer 15. This interrupt occurs every 50ms and starts the TimerPeriodElapsed Callback. The time of 50ms is enough for all spurious edge triggers to end. Within this callback, the button state is switched from an intermediary to a final state safe from jitters. This required adding a volatile `uint8_t` `button_state` variable. Adding the volatile directive ensures that the compiler does not optimize the variable, as it could change at any moment via a hardware interrupt rather than the application code's control flow. Also note that in FREERTOS, interrupts can interrupt other lower priority interrupts. Therefore, I wrapped this interrupt procedure with a Critical Section to ensure that the button press states are changed atomically [5]. As you can see, I needed to revise and refine how the HumanPlay Task works to better interact with the ComputerPlay Task. This process took approximately 2 days to complete and finished on May 12th.

When the play button is pressed by the user, the DAC DMA starts to output a pitch determined by the user's placement of his or her finger. When the button is unpressed, that signifies that the human has finished his or her playing turn. In this GPIO callback, the `PlayComputerSemaphore` is released so that the Computer has its turn to play the next target note. Before that happens, the final pitch calculated in the ADC callback is printed to the console as well as the difference from the target pitch. This is feedback to how well the user has played the target note. If the note is too sharp or too flat by 5.0Hz, an external red LED is illuminated. If the note is within the 5.0Hz range, the green LED is illuminated. The DAC DMA is also paused.

Once the ComputerPlay Task grabs its semaphore, it immediately sleeps for 500ms to allow the human to anticipate that a target note will be played. The system mode is then switched to 1 (computer mode), and the DAC DMA starts again. This time, instead of playing the user's note, the system will play its own note given by a codified song. The song in this case is "Twinkle Twinkle Little Star." The song is codified as a simple global variable of a generic `Song` data-structure with three elements. The first element is a `uint32_t` size, which represents how many notes are in the song. The second element is a `uint32_t` `current_note_index`, which represents the index of the note to be played. The final element is an array of 100 floating-point values. The floating-point values represent the notes in frequency. The ComputerPlay Task sets iterates to the next note to be played by incrementing the `current_note_index` value. Then it uses the next note's value (in Hz) to play said pitch through the DAC. The task also prints to the console through UART the target pitch value in Hz to the user. The note is played for 1000 milliseconds using another sleep call. Then the task stops the DAC and releases the `HumanPlaySemaphore` to switch to the user's turn. I completed this part on May 15th. Therefore, the entire application was successfully completed in 2 weeks.

Overall, the two semaphores, the HumanPlaySemaphore and the ComputerPlaySemaphore, synchronize the HumanPlay Task, ComputerPlay Task, and the interrupts handlers. The HumanPlay and ComputerPlay Task could also be synchronized with just one “Play Semaphore,” but I used two in case I want to add more tasks that could run at the same time with these tasks. Running at the same time in this context means that the switcher can switch really quickly between tasks, giving the illusion that tasks are running in parallel. In actuality, the Nucleo Board has only one processor, so parallel programming is not possible.

Conclusion

The result of the project is that I was able to build a system that mimics a violin using the SoftPot as a replacement for the string and a button as a replacement for the bow. I only used one SoftPot to replicate one string on the violin because using four SoftPot was too expensive for me. However, the goal of my project was to use just one SoftPot as a proof of concept, which I completed successfully. This project can be extended to use four or more SoftPots as strings. The project did work as anticipated. This is further shown through my demonstration video attached on Canvas.

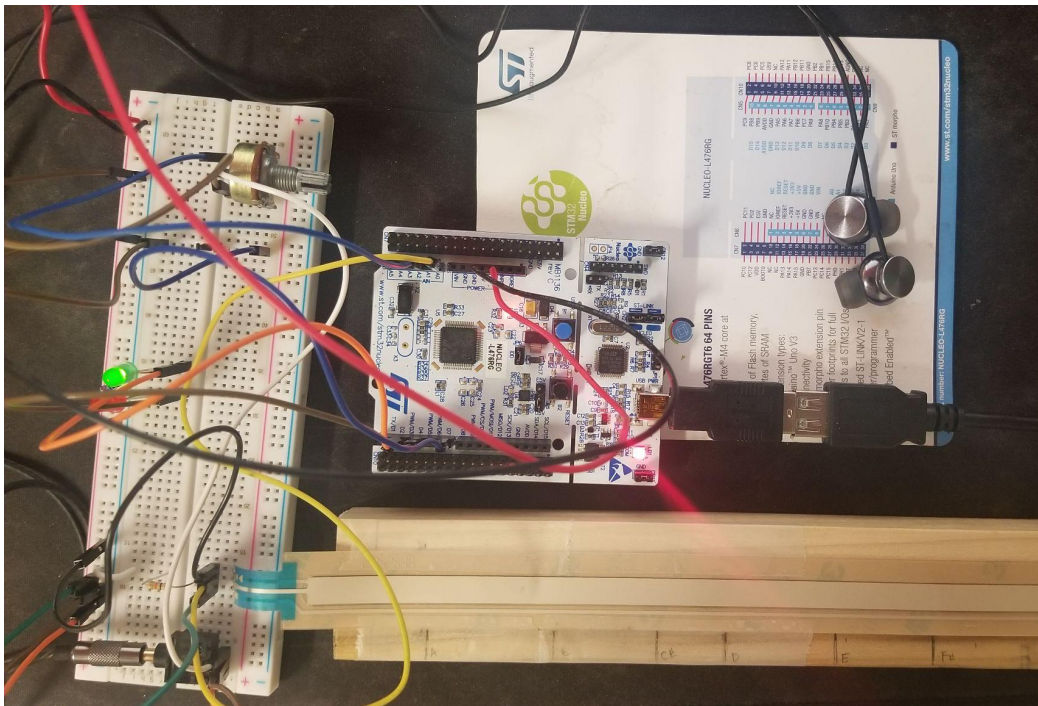


Figure 4: The embedded violin system.

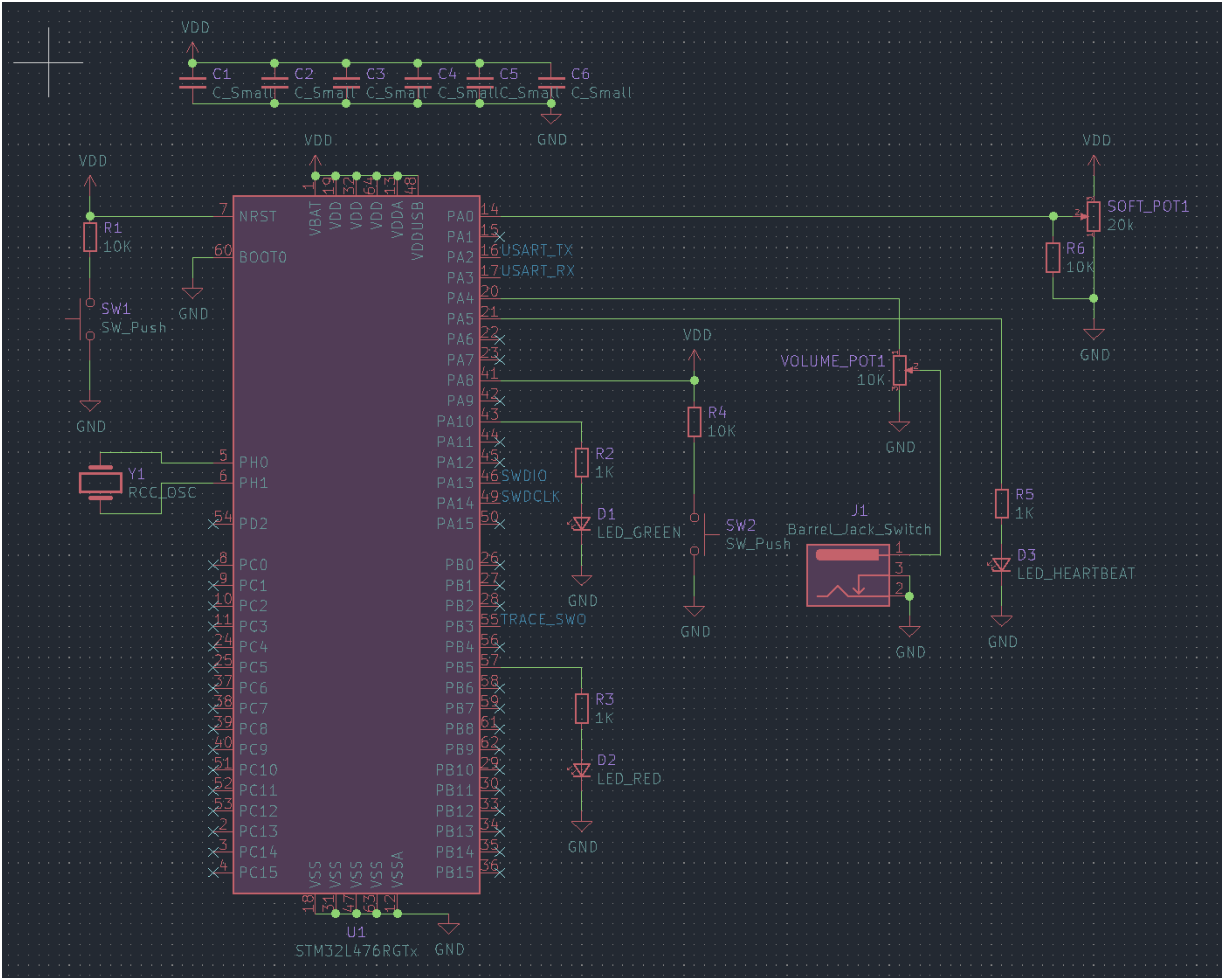


Figure 5: The schematic of the embedded system on KiCAD. Note that there is a barrel jack for the audio output of the DAC. There is also a twist-knob potentiometer to control the volume.

If I were to redo this project, I would definitely integrate an LCD as a visual interface instead of using UART and the Putty console. This would make my system more physically portable and allow users to learn or practice anywhere without a computer connection for feedback. Also, I would add an analog filter via a Schmitt trigger to the button to solve the issue with spurious interrupts. This is a more simple method than coding a debouncer, and allows the button presses to be more responsible in the magnitude of tens of milliseconds. I would also change the name of some variables to make the code more user-friendly to read and interpret. Finally, I would definitely prototype this system directly on a protoboard with solder instead of the breadboard. This is because certain wires keep getting disconnected as I move the device around, causing me to lose time on debugging. Embedded systems are tricky to debug because there is no easy traceback to the issue on most problems. The error could be a software error or a hardware error. Had I soldered this simple circuit on a protoboard, I would have saved much time pinpointing my errors to software.

If I were to expand further on this project, I would add more SoftPots to mimic more strings. I would also generalize the embedded device to not only mimic the violin, but also the cello, viola, erhu, and other stringed instruments. The goal would be to build this device on a

printed circuit board (PCB) and have different attachments of fingerboards based on which instrument the user wants to play. Based on those attachments, the system would configure itself to the correct pitch range and tone of the instrument. I would also add neopixels, which are a linked chain of small LEDs that can be lit up in many different configurations using pulse-width modulation. I want these lights to be mapped on the notes of the fingerboard. Then, when a user is asked to play an objective note, the light can turn on at that finger position, indicating where the user should place his or her finger.

When I proposed this project, I expected to learn how the basic GPIOs, ADC, and DAC of the Nucleo Board works. I accomplished learning enough about these devices to implement my project. Though, it required me to additionally learn how to use the DMA for better efficiency. I also expected to learn a small part of FreeRTOS: (1) initializing tasks with certain priority, stack size, and entry function, (2) synchronizing tasks with semaphores/mutexes, and (3) putting tasks to sleep for a fixed amount of time. Of course, there are many more complex features of FreeRTOS that would definitely make my system more efficient, scalable, and coder-friendly. However, my objective was not to master FreeRTOS, but to extrapolate upon the fundamental topics I learned in EGCP 460 and apply it to an RTOS used commonly in industry. Therefore, what I expected to learn is what I did learn. Although much of the FreeRTOS code was wrapped with STM32's CMSIS, so to better understand FreeRTOS, I would need to look even further into the code.

One question I would have is if FreeRTOS and CMSIS have a library that allows me to interface with a more sophisticated Audio Codec instead of a DAC. I could not find any libraries or resources given by the STM32 vendor. However, I know that Keil has its own CMSIS for digital signal processing on its boards. If there is no CMSIS for an Audio Codec provided by STM32, I wonder if there is any open-source library that others have made. I could not find any up-to-date portable ones on github as of yet, but I am still searching. Also, I wonder how FreeRTOS works further under the hood rather than being abstracted by STM32's CMSIS layer. For instance, there are different nuances in FreeRTOS than the scheduler shown by Jonathan Valvano. For instance, there is a distinction between Mutex and Semaphore functions where only Mutex functions provide priority inversion. There is a separate sleep state for tasks. There are different functions to Acquire and Release Mutexes/Semaphores for when it is called within a task versus being called within an interrupt service routine/callback. I wonder why these distinctions are made and what edge-cases justify these implementations.

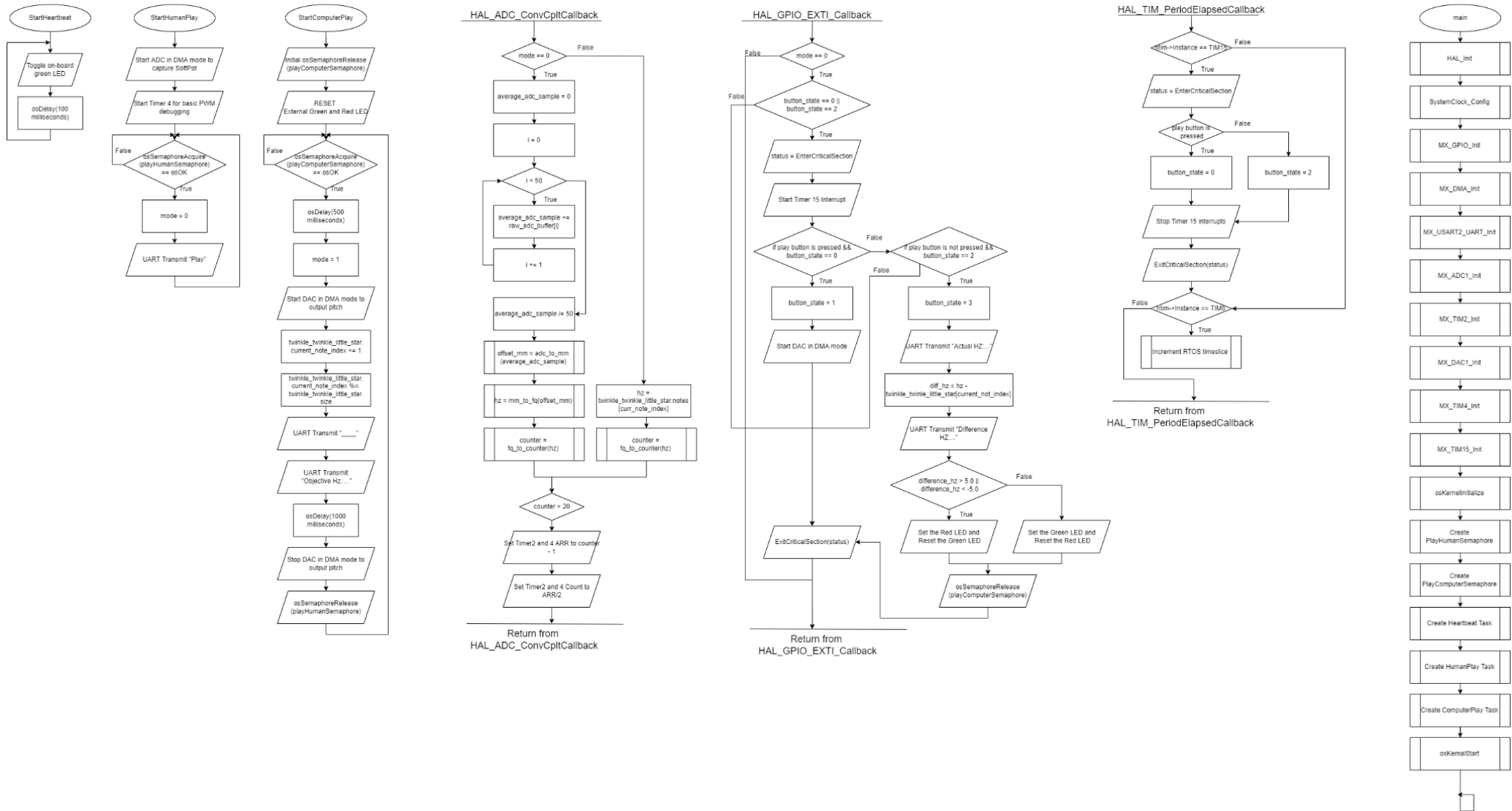
In my opinion, this project was a success. I was able to output a high-quality sine-wave tone with little to no jitters. The pitch of this tone was mapped based on the finger placement on the SoftPot ribbon. The spacing of the finger placements on the SoftPot also match that of a full size violin within a few millimeters of error. Also, the resolution provided by the SoftPot allows the pitch to be incremented at sub 1-Hz precision. I could even do and hear vibrato. The HumanPlay and ComputerPlay Task were synchronized successfully with the use of two semaphores. One task did not interrupt the other until necessary. I also implored calls to put tasks to sleep for a fixed time period. For the majority of the time, all the Tasks are sleeping, which allows FreeRTOS to switch the system to low-power mode. This would increase the lifespan of the system if it were to be battery-powered.

I would recommend this project to people interested in real-time digital signal processing, especially if they are an audiophile or a musician. The complexity of this system can be

expanded by having more complex audio signals and filters. Instead of a simple sine-wave or triangle wave, someone can design a sound wave that mimics the violin. This could be further filtered to have dynamics and color instead of a constant tone. Instead of a button switch, a pressure sensor could be used. The pressure placed on the sensor could mimic the pressure of the bow on the string. The violin sound wave could be further filtered to give a more dark, rustic sound. The project could be extended to all kinds of instruments.

Appendix

Flowchart



Code

The other drivers and libraries were not modified.

main.c

```
/* USER CODE BEGIN Header */
/**
 * *****
 * @file      : main.c
 * @brief     : Main program body
 * *****
 * @attention
 *
 * <h2><center>&copy; Copyright (c) 2022 STMicroelectronics.
 * All rights reserved.</center></h2>
 *
 * This software component is licensed by ST under BSD 3-Clause license,
 * the "License"; You may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 *             opensource.org/licenses/BSD-3-Clause
 *
 * *****
 */
/* USER CODE END Header */
/* Includes -----*/
#include "main.h"
#include "cmsis_os.h"

/* Private includes -----*/
/* USER CODE BEGIN Includes */
#include <string.h>
#include <stdio.h>
/* USER CODE END Includes */

/* Private typedef -----*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define -----*/
/* USER CODE BEGIN PD */
/* USER CODE END PD */

/* Private macro -----*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables -----*/
ADC_HandleTypeDef hadc1;
DMA_HandleTypeDef hdma_adc1;
```

```

DAC_HandleTypeDef hdac1;
DMA_HandleTypeDef hdma_dac_ch1;

TIM_HandleTypeDef htim2;
TIM_HandleTypeDef htim4;
TIM_HandleTypeDef htim15;

UART_HandleTypeDef huart2;

/* Definitions for heartbeat */
osThreadId_t heartbeatHandle;
const osThreadAttr_t heartbeat_attributes = {
    .name = "heartbeat",
    .stack_size = 128 * 4,
    .priority = (osPriority_t) osPriorityBelowNormal,
};
/* Definitions for humanPlay */
osThreadId_t humanPlayHandle;
const osThreadAttr_t humanPlay_attributes = {
    .name = "humanPlay",
    .stack_size = 128 * 4,
    .priority = (osPriority_t) osPriorityNormal,
};
/* Definitions for computerPlay */
osThreadId_t computerPlayHandle;
const osThreadAttr_t computerPlay_attributes = {
    .name = "computerPlay",
    .stack_size = 256 * 4,
    .priority = (osPriority_t) osPriorityAboveNormal,
};
/* Definitions for playHumanSem */
osSemaphoreId_t playHumanSemHandle;
const osSemaphoreAttr_t playHumanSem_attributes = {
    .name = "playHumanSem"
};
/* Definitions for playComputerSem */
osSemaphoreId_t playComputerSemHandle;
const osSemaphoreAttr_t playComputerSem_attributes = {
    .name = "playComputerSem"
};
/* USER CODE BEGIN PV */

/* USER CODE END PV */

/* Private function prototypes -----*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DMA_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_ADC1_Init(void);
static void MX_TIM2_Init(void);
static void MX_DAC1_Init(void);

```

```

static void MX_TIM4_Init(void);
static void MX_TIM15_Init(void);
void StartHeartbeat(void *argument);
void StartHumanPlay(void *argument);
void StartComputerPlay(void *argument);

/* USER CODE BEGIN PFP */

/* USER CODE END PFP */

/* Private user code -----*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
 * @brief The application entry point.
 * @retval int
 */
int main(void)
{
    /* USER CODE BEGIN 1 */

    /* USER CODE END 1 */

    /* MCU Configuration-----*/

    /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
    HAL_Init();

    /* USER CODE BEGIN Init */

    /* USER CODE END Init */

    /* Configure the system clock */
    SystemClock_Config();

    /* USER CODE BEGIN SysInit */

    /* USER CODE END SysInit */

    /* Initialize all configured peripherals */
    MX_GPIO_Init();
    MX_DMA_Init();
    MX_USART2_UART_Init();
    MX_ADC1_Init();
    MX_TIM2_Init();
    MX_DAC1_Init();
    MX_TIM4_Init();
    MX_TIM15_Init();
    /* USER CODE BEGIN 2 */

    /* USER CODE END 2 */

```



```

/* Init scheduler */
osKernelInitialize();

/* USER CODE BEGIN RTOS_MUTEX */
/* add mutexes, ... */
/* USER CODE END RTOS_MUTEX */

/* Create the semaphores(s) */
/* creation of playHumanSem */
playHumanSemHandle = osSemaphoreNew(1, 1, &playHumanSem_attributes);

/* creation of playComputerSem */
playComputerSemHandle = osSemaphoreNew(1, 1, &playComputerSem_attributes);

/* USER CODE BEGIN RTOS_SEMAPHORES */
/* add semaphores, ... */
/* USER CODE END RTOS_SEMAPHORES */

/* USER CODE BEGIN RTOS_TIMERS */
/* start timers, add new ones, ... */
/* USER CODE END RTOS_TIMERS */

/* USER CODE BEGIN RTOS_QUEUES */
/* add queues, ... */
/* USER CODE END RTOS_QUEUES */

/* Create the thread(s) */
/* creation of heartbeat */
heartbeatHandle = osThreadNew(StartHeartbeat, NULL, &heartbeat_attributes);

/* creation of humanPlay */
humanPlayHandle = osThreadNew(StartHumanPlay, NULL, &humanPlay_attributes);

/* creation of computerPlay */
computerPlayHandle = osThreadNew(StartComputerPlay, NULL,
&computerPlay_attributes);

/* USER CODE BEGIN RTOS_THREADS */
/* add threads, ... */
/* USER CODE END RTOS_THREADS */

/* USER CODE BEGIN RTOS_EVENTS */
/* add events, ... */
/* USER CODE END RTOS_EVENTS */

/* Start scheduler */
osKernelStart();

/* We should never get here as control is now taken by the scheduler */
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)

```

```

{
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
}

/**
 * @brief System Clock Configuration
 * @retval None
 */
void SystemClock_Config(void)
{
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
    RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

    /** Initializes the RCC Oscillators according to the specified parameters
     * in the RCC_OscInitTypeDef structure.
     */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
    RCC_OscInitStruct.HSIState = RCC_HSI_ON;
    RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
    RCC_OscInitStruct.PLL.PLLM = 1;
    RCC_OscInitStruct.PLL.PLLN = 10;
    RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
    RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
    RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
    {
        Error_Handler();
    }
    /** Initializes the CPU, AHB and APB buses clocks
     */
    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
        |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
    {
        Error_Handler();
    }
    PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_USART2|RCC_PERIPHCLK_ADC;
    PeriphClkInit.Usart2ClockSelection = RCC_USART2CLKSOURCE_PCLK1;
    PeriphClkInit.AdcClockSelection = RCC_ADCCLKSOURCE_PLLSAI1;
    PeriphClkInit.PLLSAI1.PLLSAI1Source = RCC_PLLSOURCE_HSI;
    PeriphClkInit.PLLSAI1.PLLSAI1M = 1;

```

```

PeriphClkInit.PLLSAI1.PLLSAI1N = 8;
PeriphClkInit.PLLSAI1.PLLSAI1P = RCC_PLLP_DIV7;
PeriphClkInit.PLLSAI1.PLLSAI1Q = RCC_PLLQ_DIV2;
PeriphClkInit.PLLSAI1.PLLSAI1R = RCC_PLLR_DIV2;
PeriphClkInit.PLLSAI1.PLLSAI1ClockOut = RCC_PLLSAI1_ADC1CLK;
if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
{
    Error_Handler();
}
/** Configure the main internal regulator output voltage
*/
if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
{
    Error_Handler();
}
}

/**
 * @brief ADC1 Initialization Function
 * @param None
 * @retval None
 */
static void MX_ADC1_Init(void)
{
    /* USER CODE BEGIN ADC1_Init 0 */

    /* USER CODE END ADC1_Init 0 */

    ADC_MultiModeTypeDef multimode = {0};
    ADC_ChannelConfTypeDef sConfig = {0};

    /* USER CODE BEGIN ADC1_Init 1 */

    /* USER CODE END ADC1_Init 1 */
    /** Common config
    */
    hadc1.Instance = ADC1;
    hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
    hadc1.Init.Resolution = ADC_RESOLUTION_12B;
    hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
    hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
    hadc1.Init.LowPowerAutoWait = DISABLE;
    hadc1.Init.ContinuousConvMode = ENABLE;
    hadc1.Init.NbrOfConversion = 1;
    hadc1.Init.DiscontinuousConvMode = DISABLE;
    hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
    hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
    hadc1.Init.DMAContinuousRequests = ENABLE;
    hadc1.Init.Overrun = ADC_OVR_DATA_PRESERVED;
    hadc1.Init.OversamplingMode = DISABLE;
    if (HAL_ADC_Init(&hadc1) != HAL_OK)

```

```

    {
        Error_Handler();
    }
    /** Configure the ADC multi-mode
    */
    multimode.Mode = ADC_MODE_INDEPENDENT;
    if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
    {
        Error_Handler();
    }
    /** Configure Regular Channel
    */
    sConfig.Channel = ADC_CHANNEL_5;
    sConfig.Rank = ADC_REGULAR_RANK_1;
    sConfig.SamplingTime = ADC_SAMPLETIME_640CYCLES_5;
    sConfig.SingleDiff = ADC_SINGLE_ENDED;
    sConfig.OffsetNumber = ADC_OFFSET_NONE;
    sConfig.Offset = 0;
    if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
    {
        Error_Handler();
    }
    /* USER CODE BEGIN ADC1_Init 2 */

    /* USER CODE END ADC1_Init 2 */

}

/**
 * @brief DAC1 Initialization Function
 * @param None
 * @retval None
 */
static void MX_DAC1_Init(void)
{
    /* USER CODE BEGIN DAC1_Init 0 */

    /* USER CODE END DAC1_Init 0 */

    DAC_ChannelConfTypeDef sConfig = {0};

    /* USER CODE BEGIN DAC1_Init 1 */

    /* USER CODE END DAC1_Init 1 */
    /** DAC Initialization
    */
    hdac1.Instance = DAC1;
    if (HAL_DAC_Init(&hdac1) != HAL_OK)
    {
        Error_Handler();
    }
    /** DAC channel OUT1 config

```

```

*/
sConfig.DAC_SampleAndHold = DAC_SAMPLEANDHOLD_DISABLE;
sConfig.DAC_Trigger = DAC_TRIGGER_T4_TRGO;
sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
sConfig.DAC_ConnectOnChipPeripheral = DAC_CHIPCONNECT_DISABLE;
sConfig.DAC_UserTrimming = DAC_TRIMMING_FACTORY;
if (HAL_DAC_ConfigChannel(&hdac1, &sConfig, DAC_CHANNEL_1) != HAL_OK)
{
    Error_Handler();
}
/* USER CODE BEGIN DAC1_Init 2 */

/* USER CODE END DAC1_Init 2 */
}

/**
 * @brief TIM2 Initialization Function
 * @param None
 * @retval None
 */
static void MX_TIM2_Init(void)
{
    /* USER CODE BEGIN TIM2_Init 0 */

    /* USER CODE END TIM2_Init 0 */

    TIM_ClockConfigTypeDef sClockSourceConfig = {0};
    TIM_MasterConfigTypeDef sMasterConfig = {0};

    /* USER CODE BEGIN TIM2_Init 1 */

    /* USER CODE END TIM2_Init 1 */
    htim2.Instance = TIM2;
    htim2.Init.Prescaler = 256-1;
    htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim2.Init.Period = 2272-1;
    htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
    {
        Error_Handler();
    }
    sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
    if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
    {
        Error_Handler();
    }
    sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
    {

```

```

    Error_Handler();
}
/* USER CODE BEGIN TIM2_Init 2 */

/* USER CODE END TIM2_Init 2 */
}

/**
 * @brief TIM4 Initialization Function
 * @param None
 * @retval None
 */
static void MX_TIM4_Init(void)
{
    /* USER CODE BEGIN TIM4_Init 0 */

    /* USER CODE END TIM4_Init 0 */

    TIM_ClockConfigTypeDef sClockSourceConfig = {0};
    TIM_MasterConfigTypeDef sMasterConfig = {0};

    /* USER CODE BEGIN TIM4_Init 1 */

    /* USER CODE END TIM4_Init 1 */
    htim4.Instance = TIM4;
    htim4.Init.Prescaler = 0;
    htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim4.Init.Period = 709-1;
    htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    htim4.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    if (HAL_TIM_Base_Init(&htim4) != HAL_OK)
    {
        Error_Handler();
    }
    sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
    if (HAL_TIM_ConfigClockSource(&htim4, &sClockSourceConfig) != HAL_OK)
    {
        Error_Handler();
    }
    sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    if (HAL_TIMEx_MasterConfigSynchronization(&htim4, &sMasterConfig) != HAL_OK)
    {
        Error_Handler();
    }
    /* USER CODE BEGIN TIM4_Init 2 */

    /* USER CODE END TIM4_Init 2 */
}

```

```

/**
 * @brief TIM15 Initialization Function
 * @param None
 * @retval None
 */
static void MX_TIM15_Init(void)
{
    /* USER CODE BEGIN TIM15_Init 0 */

    /* USER CODE END TIM15_Init 0 */

    TIM_ClockConfigTypeDef sClockSourceConfig = {0};
    TIM_MasterConfigTypeDef sMasterConfig = {0};

    /* USER CODE BEGIN TIM15_Init 1 */

    /* USER CODE END TIM15_Init 1 */
    htim15.Instance = TIM15;
    htim15.Init.Prescaler = 40000;
    htim15.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim15.Init.Period = 100;
    htim15.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    htim15.Init.RepetitionCounter = 0;
    htim15.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    if (HAL_TIM_Base_Init(&htim15) != HAL_OK)
    {
        Error_Handler();
    }
    sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
    if (HAL_TIM_ConfigClockSource(&htim15, &sClockSourceConfig) != HAL_OK)
    {
        Error_Handler();
    }
    sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    if (HAL_TIMEx_MasterConfigSynchronization(&htim15, &sMasterConfig) != HAL_OK)
    {
        Error_Handler();
    }
    /* USER CODE BEGIN TIM15_Init 2 */

    /* USER CODE END TIM15_Init 2 */

}

/**
 * @brief USART2 Initialization Function
 * @param None
 * @retval None
 */
static void MX_USART2_UART_Init(void)
{

```



```

/* USER CODE BEGIN USART2_Init 0 */

/* USER CODE END USART2_Init 0 */

/* USER CODE BEGIN USART2_Init 1 */

/* USER CODE END USART2_Init 1 */
huart2.Instance = USART2;
huart2.Init.BaudRate = 115200;
huart2.Init.WordLength = UART_WORDLENGTH_8B;
huart2.Init.StopBits = UART_STOPBITS_1;
huart2.Init.Parity = UART_PARITY_NONE;
huart2.Init.Mode = UART_MODE_TX_RX;
huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
huart2.Init.OverSampling = UART_OVERSAMPLING_16;
huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
if (HAL_UART_Init(&huart2) != HAL_OK)
{
    Error_Handler();
}
/* USER CODE BEGIN USART2_Init 2 */

/* USER CODE END USART2_Init 2 */
}

/**
 * Enable DMA controller clock
 */
static void MX_DMA_Init(void)
{
    /* DMA controller clock enable */
    __HAL_RCC_DMA1_CLK_ENABLE();

    /* DMA interrupt init */
    /* DMA1_Channel1_IRQn interrupt configuration */
    HAL_NVIC_SetPriority(DMA1_Channel1_IRQn, 5, 0);
    HAL_NVIC_EnableIRQ(DMA1_Channel1_IRQn);
    /* DMA1_Channel3_IRQn interrupt configuration */
    HAL_NVIC_SetPriority(DMA1_Channel3_IRQn, 5, 0);
    HAL_NVIC_EnableIRQ(DMA1_Channel3_IRQn);
}

/**
 * @brief GPIO Initialization Function
 * @param None
 * @retval None
 */
static void MX_GPIO_Init(void)

```

```

{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOA, LD2_Pin|GPIO_PIN_10, GPIO_PIN_RESET);

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, GPIO_PIN_RESET);

  /*Configure GPIO pin : B1_Pin */
  GPIO_InitStruct.Pin = B1_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);

  /*Configure GPIO pins : LD2_Pin PA10 */
  GPIO_InitStruct.Pin = LD2_Pin|GPIO_PIN_10;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  /*Configure GPIO pin : PA8 */
  GPIO_InitStruct.Pin = GPIO_PIN_8;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING_FALLING;
  GPIO_InitStruct.Pull = GPIO_PULLUP;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  /*Configure GPIO pin : PB5 */
  GPIO_InitStruct.Pin = GPIO_PIN_5;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

  /* EXTI interrupt init*/
  HAL_NVIC_SetPriority(EXTI9_5_IRQn, 5, 0);
  HAL_NVIC_EnableIRQ(EXTI9_5_IRQn);

  HAL_NVIC_SetPriority(EXTI15_10_IRQn, 5, 0);
  HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
}

/* USER CODE BEGIN 4 */
#include "math.h"

```

```

#define PI 3.1415926
#define DAC_RESOLUTION (0xFFF-1)

#define TRIANGLE_TABLE_LENGTH 256
uint16_t triangle_table[TRIANGLE_TABLE_LENGTH] = {
    32, 64, 96, 128, 160, 192, 224, 256,
    288, 320, 352, 384, 416, 448, 480, 512,
    544, 576, 608, 640, 672, 704, 736, 768,
    800, 832, 864, 896, 928, 960, 992, 1024,
    1056, 1088, 1120, 1152, 1184, 1216, 1248, 1280,
    1312, 1344, 1376, 1408, 1440, 1472, 1504, 1536,
    1568, 1600, 1632, 1664, 1696, 1728, 1760, 1792,
    1824, 1856, 1888, 1920, 1952, 1984, 2016, 2048,
    2079, 2111, 2143, 2175, 2207, 2239, 2271, 2303,
    2335, 2367, 2399, 2431, 2463, 2495, 2527, 2559,
    2591, 2623, 2655, 2687, 2719, 2751, 2783, 2815,
    2847, 2879, 2911, 2943, 2975, 3007, 3039, 3071,
    3103, 3135, 3167, 3199, 3231, 3263, 3295, 3327,
    3359, 3391, 3423, 3455, 3487, 3519, 3551, 3583,
    3615, 3647, 3679, 3711, 3743, 3775, 3807, 3839,
    3871, 3903, 3935, 3967, 3999, 4031, 4063, 4095,
    4063, 4031, 3999, 3967, 3935, 3903, 3871, 3839,
    3807, 3775, 3743, 3711, 3679, 3647, 3615, 3583,
    3551, 3519, 3487, 3455, 3423, 3391, 3359, 3327,
    3295, 3263, 3231, 3199, 3167, 3135, 3103, 3071,
    3039, 3007, 2975, 2943, 2911, 2879, 2847, 2815,
    2783, 2751, 2719, 2687, 2655, 2623, 2591, 2559,
    2527, 2495, 2463, 2431, 2399, 2367, 2335, 2303,
    2271, 2239, 2207, 2175, 2143, 2111, 2079, 2048,
    2016, 1984, 1952, 1920, 1888, 1856, 1824, 1792,
    1760, 1728, 1696, 1664, 1632, 1600, 1568, 1536,
    1504, 1472, 1440, 1408, 1376, 1344, 1312, 1280,
    1248, 1216, 1184, 1152, 1120, 1088, 1056, 1024,
    992, 960, 928, 896, 864, 832, 800, 768,
    736, 704, 672, 640, 608, 576, 544, 512,
    480, 448, 416, 384, 352, 320, 288, 256,
    224, 192, 160, 128, 96, 64, 32, 0,
};

#define SINE_TABLE_LEGNTH 256
uint16_t sine_table[SINE_TABLE_LEGNTH] = {
    2048, 2098, 2148, 2198, 2248, 2298, 2348, 2398,
    2447, 2496, 2545, 2594, 2642, 2690, 2737, 2784,
    2831, 2877, 2923, 2968, 3013, 3057, 3100, 3143,
    3185, 3226, 3267, 3307, 3346, 3385, 3423, 3459,
    3495, 3530, 3565, 3598, 3630, 3662, 3692, 3722,
    3750, 3777, 3804, 3829, 3853, 3876, 3898, 3919,
    3939, 3958, 3975, 3992, 4007, 4021, 4034, 4045,
    4056, 4065, 4073, 4080, 4085, 4089, 4093, 4094,
    4095, 4094, 4093, 4089, 4085, 4080, 4073, 4065,
    4056, 4045, 4034, 4021, 4007, 3992, 3975, 3958,
    3939, 3919, 3898, 3876, 3853, 3829, 3804, 3777,
    3750, 3722, 3692, 3662, 3630, 3598, 3565, 3530,

```

```

        3495, 3459, 3423, 3385, 3346, 3307, 3267, 3226,
        3185, 3143, 3100, 3057, 3013, 2968, 2923, 2877,
        2831, 2784, 2737, 2690, 2642, 2594, 2545, 2496,
        2447, 2398, 2348, 2298, 2248, 2198, 2148, 2098,
        2048, 1997, 1947, 1897, 1847, 1797, 1747, 1697,
        1648, 1599, 1550, 1501, 1453, 1405, 1358, 1311,
        1264, 1218, 1172, 1127, 1082, 1038, 995, 952,
        910, 869, 828, 788, 749, 710, 672, 636,
        600, 565, 530, 497, 465, 433, 403, 373,
        345, 318, 291, 266, 242, 219, 197, 176,
        156, 137, 120, 103, 88, 74, 61, 50,
        39, 30, 22, 15, 10, 6, 2, 1,
        0, 1, 2, 6, 10, 15, 22, 30,
        39, 50, 61, 74, 88, 103, 120, 137,
        156, 176, 197, 219, 242, 266, 291, 318,
        345, 373, 403, 433, 465, 497, 530, 565,
        600, 636, 672, 710, 749, 788, 828, 869,
        910, 952, 995, 1038, 1082, 1127, 1172, 1218,
        1264, 1311, 1358, 1405, 1453, 1501, 1550, 1599,
        1648, 1697, 1747, 1797, 1847, 1897, 1947, 1997
};

typedef struct Song {
    uint32_t size;
    uint32_t current_note_index;
    float notes[100];
} Song;

Song twinkle_twinkle_little_star = {
    .size = 8,
    .current_note_index = 0,
    .notes = {440.0, 659.25, 739.99, 659.25, 587.33, 554.37, 493.88, 440.0}
};

volatile uint8_t mode = 0; // 0 is user mode and 1 is computer mode

float adc_to_mm(float begin_adc, float end_adc, float string_length, float
target_adc) {
    // Returns the mm finger position given the adc sample from the softpot
    if (target_adc < begin_adc || target_adc > end_adc) return -1.0; // avoid out
of range
    return (target_adc - begin_adc)/(end_adc - begin_adc) * string_length;
}

float mm_to_fq(float base_fq, float string_length, float offset_mm) {
    // Returns the frequency given string base note, string length, and fingered
offset
    if (offset_mm == -1.0 || offset_mm == string_length) return base_fq; // avoid
divide by 0 and out of range

```

```

        return (string_length / (string_length - offset_mm)) * base_fq;
    }

uint32_t fq_to_counter(uint32_t lut_length, uint32_t fs, float fq) {
    // Returns counter based on lut_length, fs, and fq
    return (uint32_t)((float)fs/(fq*lut_length));
}

char debug_msg[30];

// #define ADC_DMA_LENGTH 10
#define ADC_DMA_LENGTH 50
uint16_t raw_adc_buffer[ADC_DMA_LENGTH];
uint32_t average_adc_sample = 0;
float offset_mm;
float hz;
float difference_hz;
uint32_t counter;

void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc) {
    if (mode == 0) {
        // Human mode
        average_adc_sample = 0;
        for (int i = 0; i < ADC_DMA_LENGTH; i++) {
            average_adc_sample += raw_adc_buffer[i];
        }
        average_adc_sample /= ADC_DMA_LENGTH;

        offset_mm = adc_to_mm(100, 2050, 328, average_adc_sample);
        hz = mm_to_fq(440, 328, offset_mm);
        counter = fq_to_counter(SINE_TABLE_LENGTH, 8000000, hz);
    } else {
        // Computer mode
        hz =
twinkle_twinkle_little_star.notes[twinkle_twinkle_little_star.current_note_index];
        counter = fq_to_counter(SINE_TABLE_LENGTH, 8000000, hz);
    }

    if (counter > 20) {
        TIM2->ARR = counter - 1;
        TIM2->CCR1 = TIM2->ARR/2;
        if (TIM2->CNT > TIM2->ARR) {
            TIM2->CNT = 0;
        }

        TIM4->ARR = counter - 1;
        TIM4->CCR1 = TIM4->ARR/2;
        if (TIM4->CNT > TIM4->ARR) {
            TIM4->CNT = 0;
        }
    }
}

```

```

HAL_DAC_ConvCpltCallbackCh1(ADC_HandleTypeDef* hadc) {
    // NOP
}

volatile uint8_t button_state = 0;

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
    if (mode == 0) {
        if (button_state == 0 || button_state == 2) {
            UBaseType_t status = taskENTER_CRITICAL_FROM_ISR();
            HAL_TIM_Base_Start_IT(&htim15);
            if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_8) == GPIO_PIN_RESET &&
button_state == 0) {
                button_state = 1;
                HAL_DAC_Start_DMA(&hdac1, DAC_CHANNEL_1, sine_table,
SINE_TABLE_LENGTH, DAC_ALIGN_12B_R);
            } else if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_8) == GPIO_PIN_SET
&& button_state == 2) {
                button_state = 3;
                sprintf(debug_msg, "Actual HZ: %.4f\r\n", hz);
                HAL_UART_Transmit(&huart2, (uint8_t*)debug_msg,
strlen(debug_msg), HAL_MAX_DELAY);

                difference_hz =
hz-twinkle_twinkle_little_star.notes[twinkle_twinkle_little_star.current_note_index]
;
                sprintf(debug_msg, "Difference HZ: %.4f\r\n",
difference_hz);
                HAL_UART_Transmit(&huart2, (uint8_t*)debug_msg,
strlen(debug_msg), HAL_MAX_DELAY);

                if ((difference_hz > 5.0) || (difference_hz < -5.0)) {
                    // Too much error
                    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_RESET);
                    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, GPIO_PIN_SET);
                } else {
                    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_SET);
                    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, GPIO_PIN_RESET);
                }

                HAL_DAC_Stop_DMA(&hdac1, DAC_CHANNEL_1);
                osSemaphoreRelease(playComputerSemHandle);
            }
            taskEXIT_CRITICAL_FROM_ISR(status);
        }
    }
}

```

```

/* USER CODE END 4 */

/* USER CODE BEGIN Header_StartHeartbeat */
/**
 * @brief Function implementing the heartbeat thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_StartHeartbeat */
void StartHeartbeat(void *argument)
{
    /* USER CODE BEGIN 5 */
    /* Infinite loop */
    for(;;)
    {
        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
        osDelay(100);
    }

    osThreadTerminate(NULL);
    /* USER CODE END 5 */
}

/* USER CODE BEGIN Header_StartHumanPlay */
/**
 * @brief Function implementing the humanPlay thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_StartHumanPlay */
void StartHumanPlay(void *argument)
{
    /* USER CODE BEGIN StartHumanPlay */
    HAL_ADC_Start_DMA(&hadc1, raw_adc_buffer, ADC_DMA_LENGTH);
    HAL_TIM_Base_Start(&htim4);

    /* Infinite loop */
    for(;;)
    {
        if (osSemaphoreAcquire(playHumanSemHandle, osWaitForever) == osOK) {
            mode = 0;
            sprintf(debug_msg, "Play\r\n");
            HAL_UART_Transmit(&huart2, (uint8_t*)debug_msg, strlen(debug_msg),
HAL_MAX_DELAY);
        }
    }

    osThreadTerminate(NULL);
    /* USER CODE END StartHumanPlay */
}

```



```

}

/* USER CODE BEGIN Header_StartComputerPlay */
/**
 * @brief Function implementing the computerPlay thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_StartComputerPlay */
void StartComputerPlay(void *argument)
{
    /* USER CODE BEGIN StartComputerPlay */
    osSemaphoreRelease(playComputerSemHandle);
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, GPIO_PIN_RESET);
    /* Infinite loop */
    for(;;)
    {
        if (osSemaphoreAcquire(playComputerSemHandle, osWaitForever) == osOK) {
            // Small delay to give human enough time to anticipate next computer
note
            osDelay(500);

            // Set Computer mode
            mode = 1;
            HAL_DAC_Start_DMA(&hdac1, DAC_CHANNEL_1, sine_table,
SINE_TABLE_LEGNTH, DAC_ALIGN_12B_R);

            // Move to next note
            twinkle_twinkle_little_star.current_note_index =
twinkle_twinkle_little_star.current_note_index + 1;
            twinkle_twinkle_little_star.current_note_index =
twinkle_twinkle_little_star.current_note_index % twinkle_twinkle_little_star.size;

            sprintf(debug_msg, "-----\r\n");
            HAL_UART_Transmit(&huart2, (uint8_t*)debug_msg,
strlen(debug_msg), HAL_MAX_DELAY);
            sprintf(debug_msg, "Objective HZ: %.4f\r\n",
twinkle_twinkle_little_star.notes[twinkle_twinkle_little_star.current_note_index]);
            HAL_UART_Transmit(&huart2, (uint8_t*)debug_msg,
strlen(debug_msg), HAL_MAX_DELAY);

            // Another delay to let computer play note for 1 second without
human interrupting
            osDelay(1000);
            HAL_DAC_Stop_DMA(&hdac1, DAC_CHANNEL_1);
            osSemaphoreRelease(playHumanSemHandle);
        }
    }

    osThreadTerminate(NULL);
    /* USER CODE END StartComputerPlay */
}

```

```

/**
 * @brief Period elapsed callback in non blocking mode
 * @note This function is called when TIM6 interrupt took place, inside
 * HAL_TIM_IRQHandler(). It makes a direct call to HAL_IncTick() to increment
 * a global variable "uwTick" used as application time base.
 * @param htim : TIM handle
 * @retval None
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    /* USER CODE BEGIN Callback 0 */

    // 50ms debouncer for button
    if (htim->Instance == TIM15) {
        UBaseType_t status = taskENTER_CRITICAL_FROM_ISR();
        if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_8) == GPIO_PIN_SET) {
            button_state = 0;
            HAL_TIM_Base_Stop_IT(&htim15);
        } else if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_8) == GPIO_PIN_RESET) {
            button_state = 2;
            HAL_TIM_Base_Stop_IT(&htim15);
        }
        taskEXIT_CRITICAL_FROM_ISR(status);
    }
    /* USER CODE END Callback 0 */
    if (htim->Instance == TIM6) {
        HAL_IncTick();
    }
    /* USER CODE BEGIN Callback 1 */

    /* USER CODE END Callback 1 */
}

/**
 * @brief This function is executed in case of error occurrence.
 * @retval None
 */
void Error_Handler(void)
{
    /* USER CODE BEGIN Error_Handler_Debug */
    /* User can add his own implementation to report the HAL error return state */
    __disable_irq();
    while (1)
    {
    }
    /* USER CODE END Error_Handler_Debug */
}

#ifdef USE_FULL_ASSERT
/**
 * @brief Reports the name of the source file and the source line number
 * where the assert_param error has occurred.
 */

```

```
* @param file: pointer to the source file name
* @param line: assert_param error line source number
* @retval None
*/
void assert_failed(uint8_t *file, uint32_t line)
{
    /* USER CODE BEGIN 6 */
    /* User can add his own implementation to report the file name and line number,
       ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
    /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */

/***** (C) COPYRIGHT STMicroelectronics *****/
```

References

- [1] M. Staff, "Getting started with STM32 and Nucleo part 3: How to run multiple threads with CMSIS-rtos interface," *Getting Started with STM32 and Nucleo Part 3-How to Run Multiple Threads with CMSIS-RTOS Interface*, 06-Sep-2019. [Online]. Available: <https://www.digikey.com/en/maker/videos/shawn-hymel/getting-started-with-stm32-and-nucleo-part-3-how-to-run-multiple-threads-with-cmsis-rtos-interface>. [Accessed: 03-May-2022].
- [2] "Why RTOS and what is RTOS?," *FreeRTOS*, 27-Nov-2019. [Online]. Available: <https://www.freertos.org/about-RTOS.html>. [Accessed: 03-May-2022].
- [3] "What is direct memory access (DMA)? - definition from Techopedia," *Techopedia.com*, 11-Aug-2020. [Online]. Available: <https://www.techopedia.com/definition/2767/direct-memory-access-dma>. [Accessed: 03-May-2022].
- [4] "Spectra symbol softpot: Soft membrane potentiometer," *Spectra Symbol -Innovation-Driven*, 02-Sep-2021. [Online]. Available: <https://www.spectrasymbol.com/product/softpot/>. [Accessed: 03-May-2022].
- [5] BurhanMuhyiddin and Instructables, "STM32CUBEMX button debounce with interrupt," *Instructables*, 25-Aug-2019. [Online]. Available: <https://www.instructables.com/STM32CubeMX-Button-Debounce-With-Interrupt/>. [Accessed: 03-May-2022].