ECSE 444 Lab 3 Report

Bowen Cui, David Gul

I. Introduction

In this lab, we experimented with the DAC hardware of the STM32L4 microcontroller. With the HAL library, different techniques such as timer, interrupts and DMA functions were used to generate different waveforms to output audio to a buzzer that is connected to the DAC output.

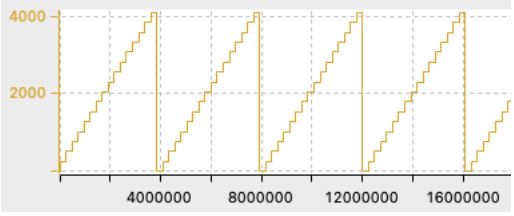
To build the circuit, one terminal of the buzzer is connected to pin 7 of the development kit, which is later configured to the output 1 of the DAC. The other end of the buzzer is connected to ground through a resistor to limit current on the GPIO pin.

II. Basic Usages of the DAC

In the first part of the lab, we have experimented on the basic usages of the DAC. First, the output 1 of the DAC is enabled in the STM32CubeMX utility. After code generation, we can see that initialisation code for the DAC is there, and there is a handle for DAC defined as "hdac1".

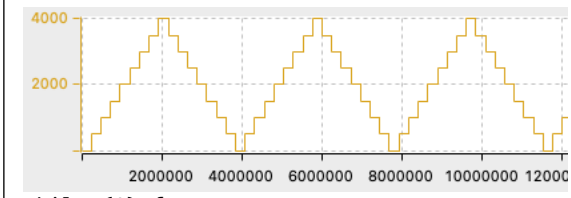
To make sounds from the DAC, first, it will have to be started. In this exercise, the normal mode is used, which means that the output of the DAC changes according to the value in the corresponding register. With the HAL library, it's done by calling the HAL_DAC_SetValue function.

First, a waveform of sawtooth wave is to be generated to be fed to the DAC. A loop is written so that in each millisecond period, the output value of the DAC is increased by 256. If the value exceeds the threshold of 4095, which is the largest number accepted by the DAC in 12 bits, the value will be reset to 0. The pseudocode is included below. This essentially creates a very crude and low-resolution sawtooth wave. Since the quality of the wave created in this way is too low, the resulting sound is very noisy compared to true sawtooth waves.



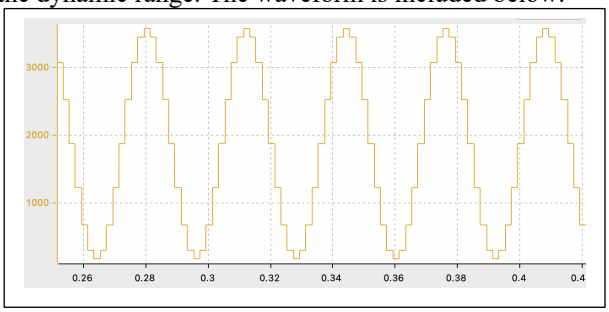

```
while (1) {
        if (DAC_value > 4095) {
                DAC_value = 0;
        } else {
                DAC_value += 256;
        }
        Delay 1ms;
}
```

Then, a waveform of triangle wave is to be generated to be fed to the DAC. Here, a loop is also used that increases the value of the DAC by 500 every millisecond. After the DAC value reaches the predetermined threshold, the DAC value will start to be decreased by 500 every millisecond. After the value reaches 0 again, the whole sequence is repeated. The pseudocode is included below. The low resolution made the result sounded noisy.




```
while (1) {
        while (DAC_value < 4000) {
                DAC_value += 500;
                Delay 1ms;
        }
        while (DAC_value > 0) {
                DAC_value -= 500;
                Delay 1ms;
        }
}
```

Finally, a sine wave with 16 levels is generated. For each period, the sine function from CMSIS-DSP library is called to get a sine value, then it's interpolated by shifting it above for 1.1, then it's timed by 1700 to interpolate the wave to around ¾ of the dynamic range. The waveform is included below.



III. Interrupts

To improve the quality sound, we need to improve the quality of the waveform used to drive the DAC. To do that, we can improve the rate which the processor accesses the DAC to use waveforms with more detail. One method will be using a timer to interrupt the processor to update the DAC value, which can achieve a much higher refresh rate than the delay function.

A. Button Interrupt

To learn and practice using the interrupts, first, a button interrupts is implemented. There are three elements in a interrupt setup: configure for the NVIC, configure for the IRQ source and the callback routine. The configure for the NVIC enables a specific IRQ source, the configure of the source determines when and how the interrupts will be invoked, and the callback routine contains the routines that is executed by the processor after each interrupt is invoked.

For button interrupt, first, the pin connected to the user button is configured to "GPIO_EXTI13", which means that the GPIO will act as the 13th external interrupt source. Then, in NVIC mode and configuration, the item "EXTI line[15:10] interrupts" is ticked, enabling interrupts for the EXTI13 source. After code is generated for that, in "main.c", a "HAL_GPIO_EXTI_Callback" function is added to overwrite the original function. In the callback function, a routine is written so that the LED connected to PB14 will be toggled if the interrupt source is the user button.

After compiling and uploading the program to the development kit, the user button is pressed for a few times, and the LED is toggled each time when the button is pressed, thus proving that interrupt is configured correctly.

B. Timer Interrupt

To enable audio playback of high quality, the sampling rate of the audio should be increased. According to the rule of Nyquist frequency, the sampling rate should be at least twice the signal that is reproduced. For the human hearing range of 20 kHz, the frequency should be at least 40 kHz. In this application, I have chosen the frequency of 48 kHz, which is a commonly used sampling rate in applications like digital TV and DVD.

To update the DAC in 48 kHz, the timer interrupt must be used. In the STM32CubeMX utility, timer 2 is enabled and configured to use a counter period of 2500, which when used to divide the system IO clock frequency of 120 MHz, results in interrupts of 48 kHz, which is exactly the frequency that we need. After the timer is configured, the item "TIM2 global interrupt" is ticked in NVIC settings.

For the DAC to output a perfect sine wave of 1.5 kHz, the program is designed to calculate a lookup table for the sine wave for one period, then the interrupt callback routine will write the next sample to the DAC at a stable frequency of 48 kHz, while the main routine does nothing after generating the lookup table. By dividing the period of 1.5 kHz by the period of 48 kHz sampling rate, the required buffer size is determined to be 32 samples. The pseudocode for the program is shown below. The resulting audio output is much smoother that sounded like a proper sine wave.

```
int main(void) {
        …
        for (i from 0 to 31) {
                // generate audio buffer
                buffer[i] = sin(2*PI*(i/32);
        }

        Start DAC and timer interrupt;
        while (1) {do nothing for ever;}
}

// Override timer interrupt callback function
void HAL_TIM_PeriodElapsedCallback(Timer) {
        if (Timer==TIM2) {
                if (counter >= 32) {
                        counter = 0;
                } else {
                        counter++;
                }

                Write buffer[counter] to DAC;
        }
}
```

IV. Driving DAC with Timer and DMA

Interrupt driven DAC is still inefficient since it uses CPU cycles. If higher sampling rate is needed for higher performance audio tasks, the tasks on the main thread will be significantly slower since more CPU time will be used to process the interrupt service routine. To solve this problem, DMA is used. For DMA driven DAC operation, the CPU only initiates the transfer, then the DMA controller will transfer data from the memory to the DAC at the interval set by the timer without any help from the CPU, saving CPU cycles.

To enable DMA for DAC operation, the DMA module is first enabled in the STM32CubeMX utility. In the DMA mode and configuration page, the DMA request of "DAC1_CH1" is added. The direction is set to "Memory to Peripheral", and the mode is set to "Circular". Also, since each sample is a 12-bit data stored in a 16-bit unsigned integer, the data width of the memory is set to "Half Word", with "Increment Address" set to true for memory. After DMA is enabled, the trigger for DAC1 is set to "Timer 2 Trigger Out Event", which means the DAC is triggered to fetch one halfbyte from the DMA controller at each timer update, which happens at 48 kHz as it's set up before.

The code logic for initialising transfer is trivial. First, the timer will have to be enabled in the default mode, where it does not generate interrupt. After the buffer is populated with a waveform, the DAC is started in DMA mode by calling "HAL_DAC_Start_DMA" and passing the pointer to the buffer and size of the buffer as arguments. After that, the DAC should start to play the sine wave stored in the buffer on itself, while the processor is free to perform other tasks.

V. Multiple Tone Generation

To generate multiple tones in a sequence, i.e., an arpeggio, the processor will have to modify the contents of the buffer to use another waveform to replace the buffer that the DMA controller is reading. To update the buffer, a function is written

to take a frequency in floating point number, and a pointer to the buffer. It will generate one period of sine wave with a frequency very close to the frequency argument. Then, the size of the complete period is returned. The pseudocode is included below. The function is very simple and it takes as little buffer as possible, but the frequency of the resulting wave is not accurate due to the simple calculation and rounding performed.

```
uint32_t generate_sine(frequency, buffer) {
        size = (1/frequency) / (1/48kHz);
        if (size > buffer) {
                return 0;
        } else {
                generate sine wave;
                write waveform to buffer;
        }

        return size;
}
```

As the processor is not doing anything during the DMA operation, we are free to generate the sine wave and update the buffer with it. Therefore, in the while loop part of the main function, first, the buffer is updated with a sine wave of C6 tone, with the size of the buffer saved, then the DMA is restarted with the new size to play the C6 tone. Finally, the processor will wait for some time with “HAL_Delay” function, and the routine is repeated with the next notes, E6 and G6.