Laboratory 3: DAC, Timers, Interrupts, DMA, and Analog Interfacing

Karim Elgammal
Computer Engineering, McGill University
260920556

Rafid Saif
Electrical Engineering, McGill University
260903802

***Abstract*—The laboratory at hand involves the use of GPIOs, the Digital to Analog Converter (DAC) and a speaker to generate an observable audio output. Throughout the laboratory, we examine different methods to relay the sound data from some function to the DAC which then outputs an electric voltage to the circuit involved with the speaker. Further, we are familiarized through the investigation with the concepts of initialization and timing synchronizations of different components throughout the STM32L4S5; these include the DMA and the timer. Finally, a final program which plays a set of distinguishable audio outputs was made to realize the full functionalities of interrupts between the components. It is noted the lab involves the use of a breadboard, a resistor and a speaker to realize the discussed circuit.**

***Index Terms*—GPIOs, DAC, speaker, audio output, DMA, timer, interrupts, STM32L4S5, STMCubeMX, STMCubeIDE**

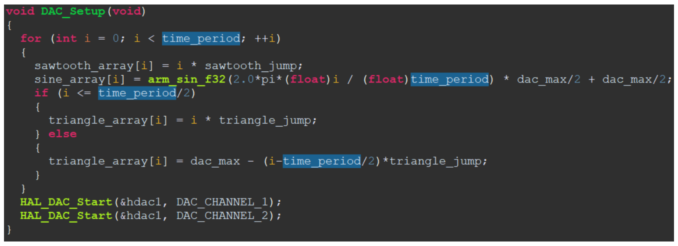
I. Introduction

This laboratory takes on two distinctive parts; firstly, creating and outputting an observable audio output through the DAC and secondly, configuring and controlling that output by way of interrupts. The first part utilizes the use of the DAC to output two signals, a saw wave, and a triangle wave. These two waves are visualized using the Serial Wire Viewer (SWV) and outputted through a simple speaker-resistor circuit. The second part involves scaling our sound data to avoid distortion of the sound and then driving the DAC through two methods; Timer-driven DAC and DMA-driven DAC. Finally, we realize a program which outputs sine sound waves with varying frequencies by initializing interrupts and timing configurations between the DAC, DMA and Push Button.

II. Methodology and Analysis

A. Part 1: Creating Signals for the Speaker Circuit

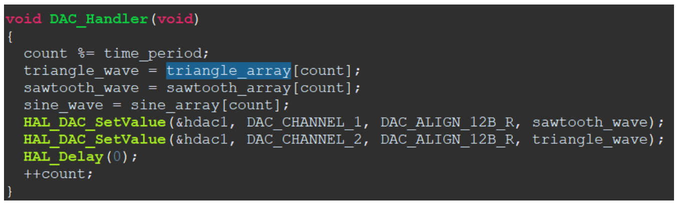
The first part of this laboratory is to implement three waveforms: triangular, sawtooth and sinusoidal; each with a time period of 15 ms using HAL_Delay(). To do this, we first create three arrays to store the values that the waveforms will take on. Given that we can configure HAL_Delay() for a minimum of 1 millisecond, we create arrays of size 15 to store the values for every millisecond. We set up the two DAC channels to output any of these two waves. At each iteration of the while loop, the DAC value is set to the next value in the array. The creation of the arrays is done in the DAC_Setup() function as seen in figure 1.



```
void DAC_Setup(void)
{
  for (int i = 0; i < time_period; ++i)
  {
    sawtooth_array[i] = i * sawtooth_jump;
    sine_array[i] = arm_sin_f32(2.0*pi*(float)i / (float)time_period) * dac_max/2 + dac_max/2;
    if (i <= time_period/2)
    {
      triangle_array[i] = i * triangle_jump;
    } else
    {
      triangle_array[i] = dac_max - (i-time_period/2)*triangle_jump;
    }
  }
  HAL_DAC_Start(&hdac1, DAC_CHANNEL_1);
  HAL_DAC_Start(&hdac1, DAC_CHANNEL_2);
}
```

Fig. 1. DAC_Setup Function

In the while loop, we then update the value of the arrays and use HAL_DAC_SetValue to update the DAC value. This is done within the function DAC_Handler() which simply runs in the while loop. Note, within this function, we neeeded to call HAL_Delay(0) to get the actual delay of 1 ms. Calling HAL_Delay(1) seemed to delay for 2ms based on the SWV Trace. The DAC_Handler() function and the graph from the trace viewer is seen in the figures below.



```
void DAC_Handler(void)
{
  count %= time_period;
  triangle_wave = triangle_array[count];
  sawtooth_wave = sawtooth_array[count];
  sine_wave = sine_array[count];
  HAL_DAC_SetValue(&hdac1, DAC_CHANNEL_1, DAC_ALIGN_12B_R, sawtooth_wave);
  HAL_DAC_SetValue(&hdac1, DAC_CHANNEL_2, DAC_ALIGN_12B_R, triangle_wave);
  HAL_Delay(0);
  ++count;
}
```

Fig. 2. DAC_Handler Function

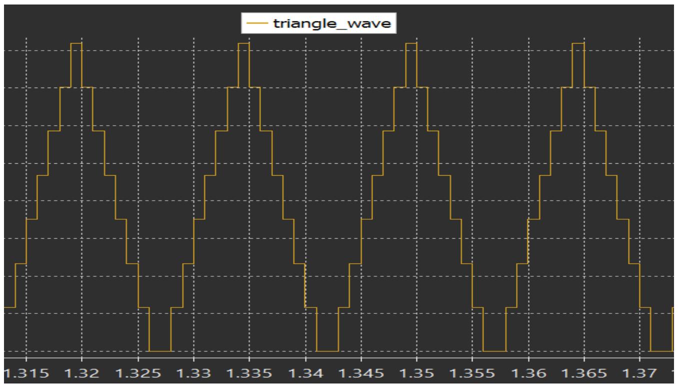


Fig. 3. Generated Triangle Wave

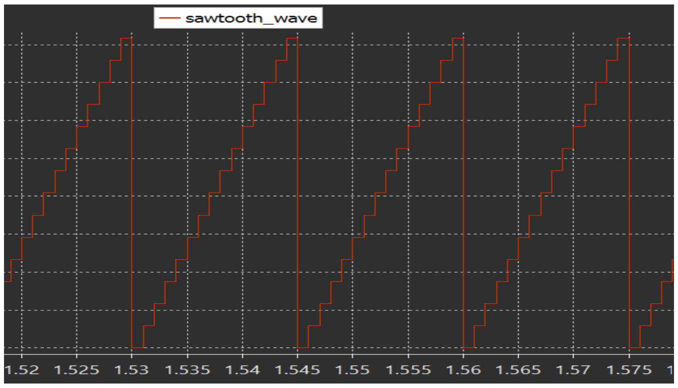


Fig. 4. Generated Sawtooth Wave

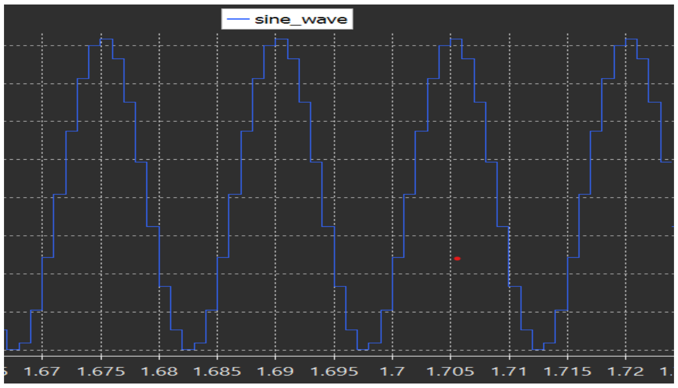


Fig. 5. Generated Sine wave

B. Part 2: DMA-Driven and Timer-Driven DAC Outputs

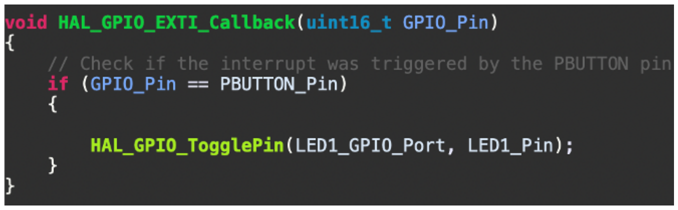
Part 2 of this laboratory was split into four distinctive steps; implementing push-button interrupts, implementing timer-driven DAC output, driving DAC with timer and DMA, and the final program with multiple tone generation. During previous labs we have opted to use polling when interacting with I/O devices, within this lab we utilize the functionalities of interrupts to handle interactions between the processor and I/O devices. This allows us to 'free up' the processor and allow the device to interrupt the processor, saving us computational time and energy.

Step 1: Implementing Push-button Interrupts

The first step of this part asks to implement the HAL_GPIO_EXTI_Callback() function which serves as the Interrupt Service Routine (ISR) for the push button at pin PC13. Before doing so, we configured the push-button on STM32CubeMX, by enabling interrupts from a signal change on the external interrupt lines. We then implemented the ISR as seen in figure 6, allowing us to toggle the state of LED 1 through the push-button. This is done by the HAL_GPIO_TogglePin() function, after a verification that the device that caused the interrupt was indeed the push-button.

Step 2: Implementing Timer-driven DAC Output

We then enabled global interrupts from TIM2, as it is a 32-bit general purpose timer. Further, we configured the timer

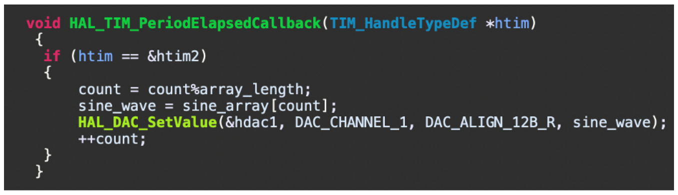


```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    // Check if the interrupt was triggered by the PBUTTON pin
    if (GPIO_Pin == PBUTTON_Pin)
    {

        HAL_GPIO_TogglePin(LED1_GPIO_Port, LED1_Pin);
    }
}
```

Fig. 6. Push-Button ISR, 1

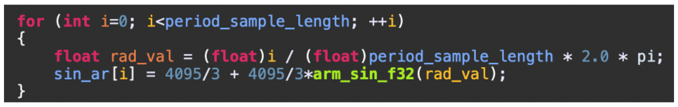
with a counter period of 7499, this is to achieve a running frequency of 16 kHz on the 120 MHz chip. We chose a running frequency of 16 kHz as it is the frequency of speech signals. As before, once we regenerated the code, we implemented the HAL_TIM_PeriodElapsedCallback() function, which serves as the ISR for the timer. This function is called once the timer has counted 7499 counts and performs the snippet of code seen in figure 7. As seen in figure 7, the function simply relays values to the DAC by calling the HAL_DAC_SetValue() function, from a previously defined array called sine_array. This array is populated with values obtained from the code snippet seen in figure 8. As seen from the figure these values are generated by iterating through each sample in the sine wave, calculating its corresponding radian value using the formula (i / period_sample_length) * 2 * pi, where "i" is the current sample index and "pi" is the mathematical constant pi. It then calculates the sine of this radian value using the arm_sin_f32 function, which is a sine function optimized for ARM processors.



```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
 {
  if (htim == &htim2)
  {
      count = count%array_length;
      sine_wave = sine_array[count];
      HAL_DAC_SetValue(&hdac1, DAC_CHANNEL_1, DAC_ALIGN_12B_R, sine_wave);
      ++count;
  }
 }
```

Fig. 7. Timer ISR

By dividing 4095 by 3, the code snippet scales the sine wave values to a range of [-1365, 2730], which is approximately centered around the midpoint of the DAC range, ultimately avoiding distortion. These values are then stored in an array and used by the ISR to relay values to the DAC. This was then tested on the circuit realized previously and successfully generated a sine wave sound.



```
for (int i=0; i<period_sample_length; ++i)
{
    float rad_val = (float)i / (float)period_sample_length * 2.0 * pi;
    sin_ar[i] = 4095/3 + 4095/3*arm_sin_f32(rad_val);
}
```

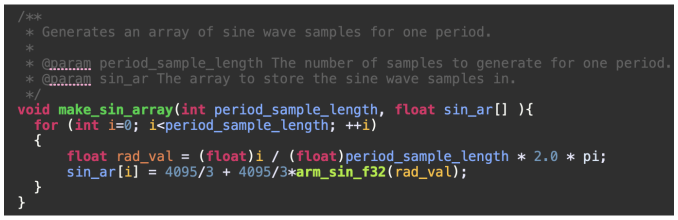
Fig. 8. Sine Array Creation

Step 3: Driving DAC with Timer and DMA

For this part of the lab, we realized a program that drives the DAC using the DMA. To do this, we had to reconfigure the DAC using STMCubeMX so that it gets triggered by the timer, instead of the ISR we implemented earlier. And so, we changed the trigger within the configuration settings of the DAC. Next we set up the DMA (direct memory access), in circular mode, specifying that the type of data expected from memory should be a half word for the memory data that the DMA will be handling, the sine values for the specified frequency. Finally, we had to change the way we start the DAC, to start in DMA mode. To achieve this, we used the HAL_DAC_Start_DMA() which takes a pointer to the sine array we created in figure 8, to start relaying that data in a circular fashion to the DAC. This program was then tested and indeed worked to supply an audible output which matches that of step 3.

Step 4: Multiple Tone Generation

Finally, we were asked to create a functioning program, that uses the DMA-driven DAC to output 3 different tones corresponding to the musical parlance of C6, E6 and G6, under control of the push-button. And so, to achieve this, we first had to create three different arrays for the varying tones we decided to output. It is noted that since our machine runs at 120 MHz and, we have set the timer to run at a sampling rate of 16 kHz, the frequency of our sine wave is achieved by the length of the array we define. That is, the array covers an entire period of the sine wave and the length is calculated as sampling_rate/sine_frequency, in order to achieve the correct array length.

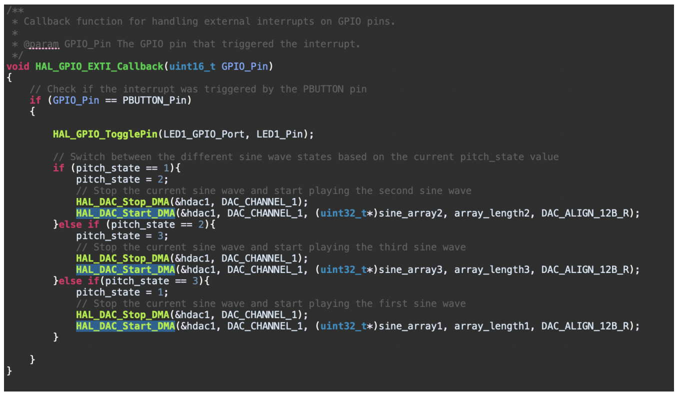


```
/**
 * Generates an array of sine wave samples for one period.
 *
 * @param period_sample_length The number of samples to generate for one period.
 * @param sin_ar The array to store the sine wave samples in.
 */
void make_sin_array(int period_sample_length, float sin_ar[] ){
  for (int i=0; i<period_sample_length; ++i)
  {
      float rad_val = (float)i / (float)period_sample_length * 2.0 * pi;
      sin_ar[i] = 4095/3 + 4095/3*arm_sin_f32(rad_val);
  }
}
```

Fig. 9. make_sin_array Function

And so, to produce tones of C6, E6 and G6, we gathered the respective frequencies of each, and used the value as sine_frequency to obtain the array size for each of the tones. It is noted that C6, E6 and G6 correspond to frequencies 1027.47 Hz, 1294.54 Hz and 1539.47 Hz respectively [1]. Once each of the array lengths was calculated, we encompassed the sine_array creation into a function make_sin_array(), seen in figure 9. This function takes the length and the array object as parameters and fills the array with the corresponding samples. We decided to create the 3 arrays in order to avoid dynamically changing the length of the array. This is because while it might save us memory space, it will increase the computational time of the program by causing it to iterate over the make_sin_array() function, every time the tone is changed. This classical time-memory trade-off presents us with two solutions, but we have opted to go forward with prioritizing limiting the computational time.

And so, in order to introduce the functionality of the push-button changing the tone being played; we had to alter the ISR we wrote previously in HAL_GPIO_EXTI_Callback().



```
/**
 * Callback function for handling external interrupts on GPIO pins.
 *
 * @param GPIO_Pin The GPIO pin that triggered the interrupt.
 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    // Check if the interrupt was triggered by the PBUTTON pin
    if (GPIO_Pin == PBUTTON_Pin)
    {

        HAL_GPIO_TogglePin(LED1_GPIO_Port, LED1_Pin);

        // Switch between the different sine wave states based on the current pitch_state value
        if (pitch_state == 1){
            pitch_state = 2;
            // Stop the current sine wave and start playing the second sine wave
            HAL_DAC_Stop_DMA(&hdac1, DAC_CHANNEL_1);
            HAL_DAC_Start_DMA(&hdac1, DAC_CHANNEL_1, (uint32_t*)sine_array2, array_length2, DAC_ALIGN_12B_R);
        }else if (pitch_state == 2){
            pitch_state = 3;
            // Stop the current sine wave and start playing the third sine wave
            HAL_DAC_Stop_DMA(&hdac1, DAC_CHANNEL_1);
            HAL_DAC_Start_DMA(&hdac1, DAC_CHANNEL_1, (uint32_t*)sine_array3, array_length3, DAC_ALIGN_12B_R);
        }else if(pitch_state == 3){
            pitch_state = 1;
            // Stop the current sine wave and start playing the first sine wave
            HAL_DAC_Stop_DMA(&hdac1, DAC_CHANNEL_1);
            HAL_DAC_Start_DMA(&hdac1, DAC_CHANNEL_1, (uint32_t*)sine_array1, array_length1, DAC_ALIGN_12B_R);
        }

    }
}
```

Fig. 10. Push-Button ISR, 2

As seen in figure 10, the ISR retains the LED toggling to validate that we enter the ISR. Then, we have introduced a global variable pitch_state, which holds a value of 1, 2 or 3, depending on which tone is being played. Using this variable, we created a sort of finite state machine, which simply moves between states using the pitch_state variable, then uses HAL_DAC_Stop_DMA() to stop the tone being played and starts the new state's tone by calling HAL_DAC_Start_DMA() with the relevant sine array to output. Each state corresponds respectively to the tones described earlier. Once the push-button is pressed, the machine starts executing the ISR, and the respective tones are accessed by pressing the button, which increments the state until it reaches state . Once in state 3, the next push of the button sends the program back to playing pitch_state = 1, C6.

III. Conclusion

In conclusion, throughout this lab we explored various configurations and their effects on the Digital to analog converter of the STM32. Starting with simply using HAL_Delay() demonstrated its downside of inaccuracy. Moving on to using the timer, the generated waves were significantly smoother, and we were able to achieve much higher sampling rates. Finally, using DMA, we were able to directly trigger the DAC when a timer event occurred and used this to implement a tone-changing program on our speaker.

IV. Works Cited

[1] J. L. Suits, ”Note Frequency Calculator and Player,” Michigan Technological University, Houghton, MI, USA, 2010. [Online]. Available: https://pages.mtu.edu/ suits/notefreq432.html. [Accessed: Mar. 15, 2023].