ECSE444 Microprocessors

Winter 2021

Lab 3: DAC, Timers, Interupts, DMA and Analog Interfacing

This exercise will extend the use of GPIOs and will employ the DAC and the buzzer (speaker) to generate observable audio outputs. You will also practice the use of a timer (TIM), interrupts and finally the DMA. The lab will be done in two parts, with the second part building on the success of the first one.

This exercise relies on the previous laboratory exercises, the classes and tutorials, and it will focus on the use of basic hardware blocks within the processor. In addition to consulting the class notes, you should consult the processor documentation to complete this exercise. Some specific hints will be given in the tutorial and the lectures leading to this lab exercise.

Background

Timer (TIM)
STM32L4+ processors have multiple timers, as described in detail in Sec. 38 of the STM32L4+ Reference Manual, available on MyCourses. These timers can generate a variety of signals and interrupts, and they are able to start DMA.

Lab 3 – Part 1

You will need to use Cube MX to configure a few GPIO pins for analog output. Since DAC converts digital register values (i.e., integers) into analog values (i.e., voltages), we will use that signal to drive a speaker with an oscillating signal.

You will drive two different signals on two different DAC output channels: a saw wave and a triangle wave, with as similar a frequency as possible.

There are two basic paths to discovering which pins must be configured for the on-board DAC. The first, cumbersome way is through manuals, so we will ignore it here. The better path is to use MX. In the *Pinout & Configuration* tab, MX summarizes many of the features of the chip on the left-hand side, under categories such as *System Core*, *Analog*, *Timers*, etc. Under *Analog*, choose *DAC1*. Enabling OUT1 and OUT2 will automatically enable the correct pins in the appropriate mode.

To configure the DAC, we need to find *DAC1* under *Analog* in the list of features under *Pinout & Configuration*. If you haven't already, in *DAC1 Mode and Configuration*, enable OUT1 and OUT2 in *Connected to external pin only* mode. Then, verify the *DAC Out1 Settings* and *DAC Out2 Settings*:

- Output Buffer (Enable)
- Trigger (None)
- User Trimming (Factory trimming)

- Sample And Hold (Sampleandhold Disable)

Step 1: Making Signals

In Lab 2, you have read the state of a button and written to a LED (besides using ADC). In this lab, you will initialize and write to the DAC to generate signals in an audible frequency range, such that we can observe the system operation with a small speaker. Button and a LED will be used a bit differently.

You should at first implement the code that manually generates two signals: a triangle wave, and a saw(tooth) wave. Without use of interrupts, it is difficult to precisely time these signals. However, do your best to generate oscillating signals with a period of ~15 ms (corresponding to 65 Hz, or note C2). You will be shown below how to observe the generated signals by a debugger before sending them to the DAC.

Next, assign each signal to a different DAC output channel. To initialize the DAC and write data to it, you'll need more HAL functions. Sections 16.2.3 and 16.2.4 of the HAL Driver User Manual list the functions you will need; they are detailed in Section 16.2.7.

Note that the DAC can operate with either 8-bit (0 to 255) or 12-bit (0 to 4095) precision. You make this choice with parameters passed to the HAL driver. Recall that 8- and 16-bit integer data types are available (uint8_t and uint16_t) and using them may simplify your implementation.

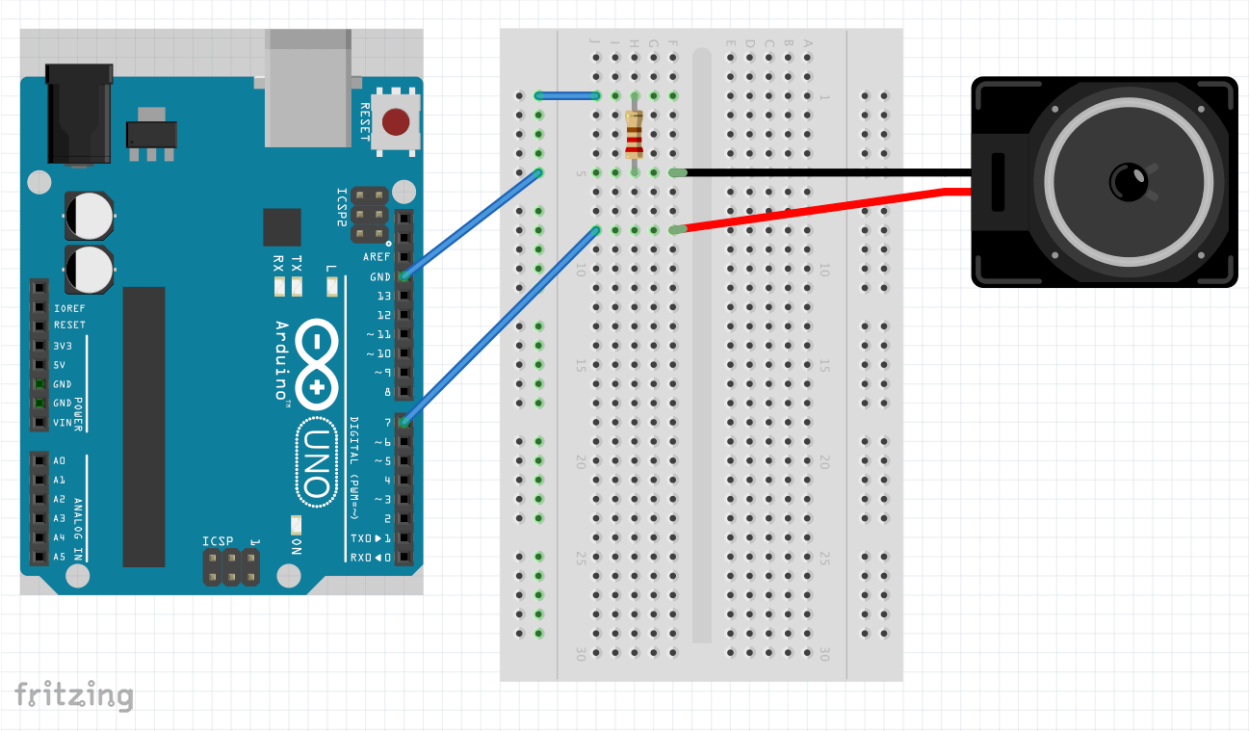
Further, note that HAL_Delay(...) can be used to insert a delay between operations in your code. As a reminder, the details of its usage can be found in the HAL Driver User Manual.

If you generate your code and return to IDE, the DAC1 should be configured. (Hopefully you will still remember to write your own code within USER CODE BEGIN and USER CODE END, and it's all still there!).

Please take notice whether any light (LED1?) of the board blinks when this part of your project is running. Can you explain why?

Step 2: Making Sounds

When the waveforms look right, wire a speaker using the components available to you. Note that:
(1) Your board has the same external interface as Arduino (A0-A5 and D0-D15, plus others).
(2) Your speaker is different, but it fits into the breadboard with the indicated spacing.
(3) The resistor is placed in series to speaker to limit the current and protect both devices.



Step 3: Making Better Sounds

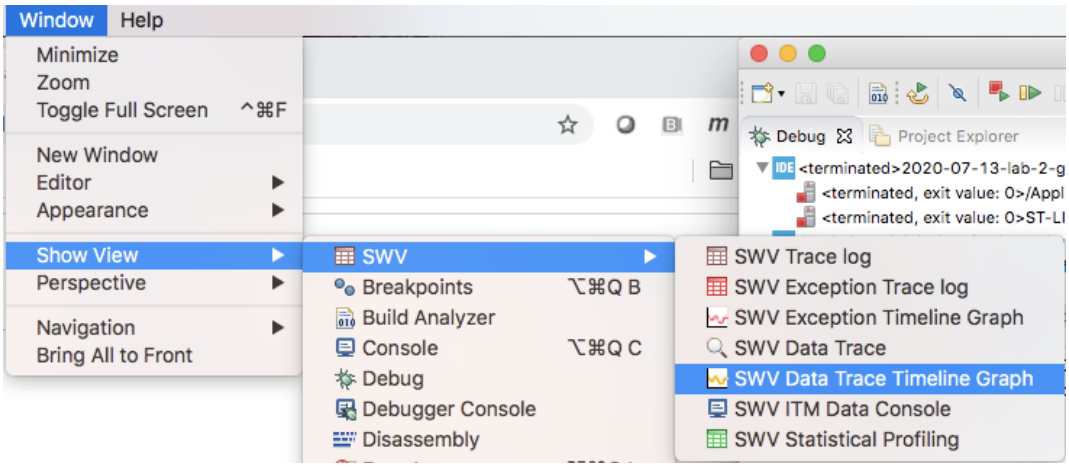
How do the triangle and saw waves sound? Not great. Do they have the desired frequency? Not really, though we can't really fix this without using timers and interrupts (later!).

Next, generate a signal with approximately the same period as above but using the arm_sin_f32() function in the DSP library. (similar to Lab 1.) As before, trace the values before driving the speaker.

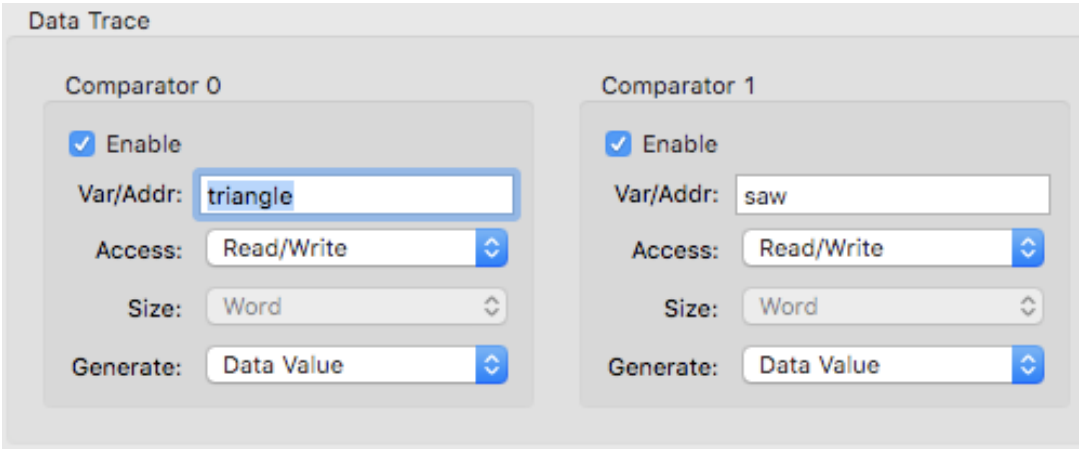
Useful Notes

The Debugger use in Step 1 Above

While developing your code, you will spend substantial time using the debugger. Before you test your code with a speaker, use the ITM interface to verify that it is working as intended. Ensure that the *Serial Wire Viewer* (SWV) is enabled and configured appropriately in the debugger configuration. Since we'll use the ITM's data trace functionality this time, no code modifications are required (e.g., to timestamp events). Start the debugger. Once it pauses execution at the first line of main, ensure that the SWV Data Trace Timeline Graph is visible; find it under the *Window > Show View > SWV pull-down* menu.

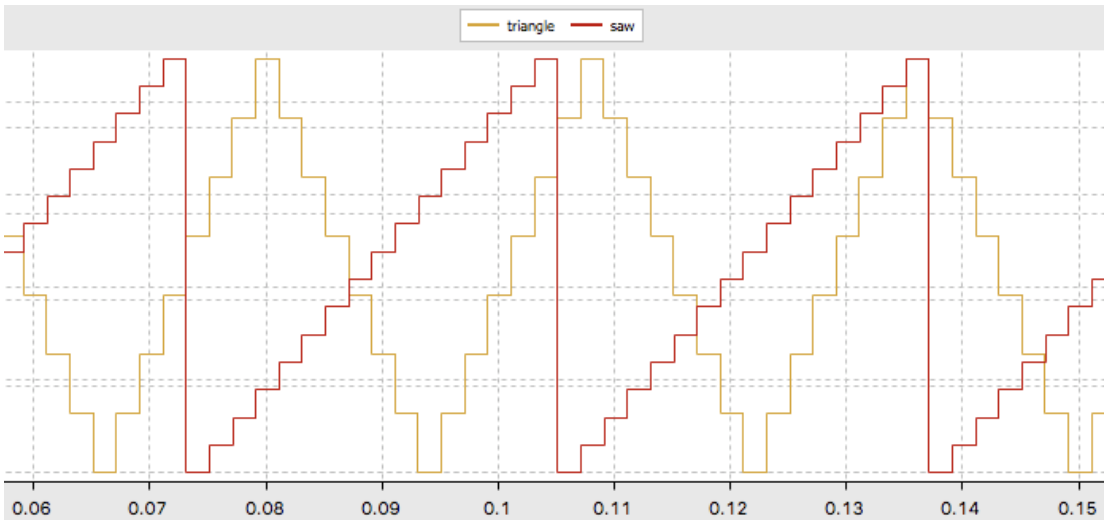


Before resuming execution, you need to configure (wrench) and then start recording (red button). Configure the Serial Wire Viewer to enable Comparator 0 and Comparator 1, and write the names of the variables you wish to monitor in *Var/Addr*. In my case, the variables that hold the current signal values are triangle and saw.



If you try to specify variables for tracing when they are out of scope (e.g., you pause and the code stops inside a library), you may get a warning indicating *Variable not found!* Tracing will not work properly unless you configure the comparators while the variables are in scope.

When you resume execution (don't forget to record), if everything is working properly, the data trace will rapidly fill with oscillating signals. Note that at our target frequency you may have to zoom in a bit in order to distinguish your triangle and saw waves. In my case, you will observe that the period of the two signals is not exactly the same. We'll achieve more precise timing in later labs when we use interrupts.



Lab 3 – Part 2

In this part, you will improve upon the quality of the output by using the timer to control the rate of writing to the DAC. The timer can generate interrupt to execute a special function, an interrupt handler. Interrupts tend to be more efficient than *polling* (which is how we've interacted with the button), or using HAL_Delay(...) (which is how we've interacted with the DAC), and gives us greater control over timing, which is essential for a wide variety of applications. We'll first use an interrupt to detect when the button has been pressed. We'll then use a timer, and its periodic interrupt, to determine when to write new data to the DAC. Then we'll use the timer, and direct memory access (DMA) to write the DAC; in this last case, sending values to the DAC will be handled almost entirely by hardware, leaving the processor free for other tasks.

Push button

Configure the push button to enable external interrupts. In the *Pinout & Configuration* tab, on the left, select *GPIO*. Under *Configuration*, select *NVIC*, and enable *EXTI line[15:10] interrupts*. This means that an interrupt will be generated whenever there is a signal change on the external interrupt lines; you button should wired to one interrupt, and the corresponding code action will be written for that interrupt to affect the LED, as explained on the next page.

DAC

For Part 2, modify DAC to use only one channel. (Remember to check the schematic; you'll be using your speaker again, and it will not work when wired to the wrong output.)

Timer

The timer will help to update the DAC output at regular intervals. What's an appropriate interval? CD-quality audio is sampled (and reproduced) at 44.1kHz. Voice call audio can be sampled at lower rates.

Choose a sampling rate (e.g., 44.1 kHz). Given your system clock frequency (e.g., 80 MHz),

calculate the counter period (the maximum value of the counter) to achieve this sampling rate. In this lab, the timer pre-scaler is not necessary. Finally, under *Parameter Settings*, set the *Trigger Event Selection TRGO* to *Update Event*, and under *NVIC Settings*, enable *TIM2 global interrupt*. Together, these settings ensure that (a) when the timer elapses, execution in main() is interrupted; and, (b) the callback function (defined below) is executed.

Step 1: Implementing Push-button Interrupts

An interrupt is a signal (internal or external) that prompts the processor to stop normal execution (e.g., in main()), and begin executing an interrupt service routine (ISR) handler, a function responding to the interrupt event. In Lab 2, the code *polled* (checked over and over and over again) for changes in the push button signal; in this lab, you will write a function that is executed whenever the push button interrupt occurs, such that the LED shows the value of the button (1/0).

Section 31.2 of the HAL Driver User Manual details the functions used to interact with GPIO. What we're interested in, in particular, is HAL_GPIO_EXTI_Callback(...). This function is called by the GPIO external interrupt handler, and we can control what it does in main.c by simply writing a new definition; our new function is automatically used instead of the weakly defined original.

Write this function in main.c (be sure to respect the function prototype defined in the HAL manual) so that it toggles the LED. This function takes as an argument the pin that caused the interrupt; it's good programming practice to verify that the interrupt was caused by the pin we think it was. This isn't essential for our lab, since there are no other external interrupts, but is necessary when a single callback function may need to handle various interrupt sources.

Note: remember again to put your code in a USER CODE region so that it doesn't disappear when we go back to MX to modify our configuration. We'll be using the push button again later.

Step 2: Implementing Timer-driven DAC Output

Now, write a callback function for the timer. Section 72.2 of the HAL Driver User Manual details the functions used to interact with timers. You are particularly interested in two sets of functions: the TIM Base functions, and TIM Callback functions. You want to start the timer WITH global interrupts enabled, i.e., in interrupt mode. Read the function definitions carefully, so that you start your timer in the correct mode. (Yes, you need to call a function to start the timer; don't forget to do so, as this is an otherwise very frustrating problem to debug.)

Just like for the button, HAL_TIM_PeriodElapsedCallback(...) is called by the TIM interrupt handler. Write a new definition for it in main.c. Again, it's good programming practice to verify that the timer causing the interrupt (an argument passed to your function) is actually the one you want to respond to.

In this function, you'll send a new value to the DAC (see Part 1 and Section 16.2 of the HAL Driver User Manual). You cannot pass it as an argument, because you don't call this function; it is called asynchronously in an entirely hardware-controlled process, and the only argument is the timer that caused the interrupt.

What you can do, however, is put the DAC values in a global variable (defined outside of any function, like other variables in main.c). You don't have control over when the timer will elapse and the callback is called; you need to prepare *all* the DAC values, and save them in global variables that can be accessed by the callback function.

In main(...), write code to populate an array with a sine wave (use the ARM math library). You can "play" this wave on our speaker (using the same circuit as in Lab 2). To get the best possible results:

- Pick a wave frequency in the 1-2 kHz range (~C6-C7 in music parlance). Lower frequencies are harder to hear; higher frequencies too, depending on your hearing abilities.
- Note that the timer frequency is (and must be) higher than that of the signal you want to drive; how do you ensure that your desired frequency is realized? (Nyquist to the rescue)
- Note that the number of saved samples matters; if you save samples for anything other than $2n\pi$ radians, you will have a discontinuity from the end to the beginning of the array, causing distortion.
- Scale your DAC values so they vary over about 2/3 of the possible dynamic range. The chip will dynamically clamp GPIO outputs to prevent damage, limiting their current to 20 mA. If you attempt to use the full range of DAC output, the signal will look fine under high impedance (e.g., with a voltmeter or pocket oscilloscope), but will clip when connected to the speaker, causing distortion.

Using a global array defining the values to be sent to the DAC, write your implementation of the timer callback so that it sends a new value from this array to the DAC each time it is called.

Part 2: Driving DAC with Timer and DMA

Next, change our code to use direct memory access (DMA). DMA uses an on-chip peripheral that can be programmed to perform memory accesses. In this case, DMA will read our array of sine values and write to the DAC for us. This means that we no longer need to execute code in the timer interrupt callback, saving CPU cycles for other tasks (if we had any) or reducing power.

To use DMA, you need to reconfigure the DAC. Instead of using our timer to trigger a callback that sets the DAC value, we'll use our timer to trigger the DAC itself. Return to MX. The first thing we need to change, then, is to select the appropriate trigger in *Parameter Settings*. Under *Trigger*, choose the trigger out event corresponding to your timer.

Next, we need to set up DMA. Go to *DMA Settings*, and add a DMA request. Choose *Circular* mode; this means the DAC will repeatedly read from the array, starting over from the beginning when the end is reached. *Normal* mode implies that the array would be read and transferred once. Choose the appropriate data width for your software; e.g., I've used 8-bit resolution for my DAC, and a uint8_t array for my sine waves, and therefore want DMA to transfer bytes.

Now regenerate your code. Comment out or otherwise disable your timer callback; it is no longer needed. In fact, the global interrupt for your timer isn't necessary at all, and can be disabled (though it won't hurt anything). The last thing to do is change how you start the DAC, to start it in DMA mode (Section 16.2 of the HAL Driver User Manual).

Step 4: Putting it All Together and Multiple Tone Generation

Finally, combine functionality into something more sophisticated. Expand your code so that when the button is pushed, the tone played on the speaker changes. Select at least three different tones; an *arpeggio* (e.g., C6, E6, G6) would suit the purpose well, but anything else is fine, too. Use interrupts and DMA.

Experimental Results to Demo

You are asked to reach the following milestones.

Grading

- C implementation of signals (triangle, saw, sine) in Part 1
  - 10%
- Visualization of signals (triangle, saw, sine) in Part 1
  - 10%
- Audible confirmation of signals (triangle, saw, sine) in Part 1
  - 10%
- Pushbutton interrupt
  - 10%
- Timer interrupt for driving DAC
  - 10%
- DMA driving the data
  - 20%
- Multiple tone audio generation
  - 10%
- Working demo organization and success
  - 20%

Final Report

Once you have all the parts working, include all the relevant data to your report. The report should concisely explain your solution to the problem given, including the final code. You should use the established 2-column IEEE format. Please capture the screen shots and relevant code snippets, and include them in the Appendix. All code should be well documented. Any performance evaluation and correctness validation should be apparent from your written report.

Due Dates

The first two labs will be completed in several phases, over the three weeks. First, you should take time to understand the lab and ask any questions in regular lab sessions or through discussion groups.

There will be the first lab demonstration on

Mar. 9-11th,

by which time you should solve Part 1, and be able to demo and explain how you approach the exercise. Please note that you will be asked to cycle through 3 different signal shapes.

The final demonstration in which all the parts (interrupts, timer, DMA) are put together will be on

Mar. 14th and 16th

and will include showing your source code and demonstrating a working program for all test cases.

The final report will be due on

Friday, Mar. 17th