ECSE 444: Microprocessors
Lab 2: GPIO and DAC

Abstract

In this lab you will (a) learn how to take and react to a digital GPIO input, in the form of a button press, (b) generate output for a digital to analog converter, and (c) how to use the debugging interface to plot variables as they change with time.

Deliverables for demonstration

- C implementation of LED lighting on button press.
- C implementation of triangle, saw, and sine signals.
- Visual confirmation of triangle, saw, and sine signals.
- Audible confirmation of triangle, saw, and sine signals.

Grading

- LED lighting on button press
  - 10%
- C implementation of signals
  - 10% triangle
  - 10% saw
  - 10% sine
- Visualization of signals
  - 10% triangle
  - 10% saw
  - 10% sine
- Audible confirmation of signals
  - 10% triangle
  - 10% saw
  - 10% sine

Changelog

- 29-Jul-2020 Initial revision.

Overview

In this lab, we'll take a more in-depth look at GPIO, and introduce digital to analog conversion. Unlike the previous two labs, this time you won't be walked through each step of the process, but instead be directed to reference material. First, we'll use an on-board button to control an LED. Second, we'll configure and drive an on-chip digital to analog converter (DAC) with a periodic signal using trigonometric functions available from CMSIS-DSP.

Resources

ARM Cortex-M4 Programming Manual
B-L475E-IOT01A User Manual
HAL Driver User Manual
STM32L475VG Datasheet
STM32L47xxx Reference Manual

Part 1: GPIO in Digital Mode, with Buttons and LEDs

Configuring the Board

Initialization

As before, start a new project in STM32CubeMX, and initialize its configuration, reviewing instructions in Lab 0 and Lab 1 if necessary. These basic steps (as well as some additional ones) will be repeated at the beginning of each of the rest of the labs.

1. Check the clock configuration to ensure that HCLK is 80 MHz. (See Lab 0.)
2. Clear the pinout. (See Lab 0.)
3. Set up PB3 and PA13 to support use of the ITM for debugging. (See Lab 1.)

LED and Push-button Configuration

For the first part of this lab, we'll use the push-button and LED. Start by configuring the LED. (See Lab 0.)

Next, to configure the button we need to first determine which pin is associated with the blue button on the development board; the black button is always configured to reset the processor. Looking at the table of contents of the B-L475E-IOT01A User Manual, observe that LEDs and buttons are covered in Section 7.14. There, in Table 2, we observe that the blue button is referred to as B2; however, unlike for the LEDs, no pinout information is provided.

Referring once again to the table of contents, observe that schematics are available in Appendix B. At the beginning of the appendix, we observe that peripherals are covered in Figure 31. Figure 31 indicates the name of the signal that the blue button is ultimately connected to, BUTTON_EXTI13.

Finally, referring to Appendix A (I/O assignment), we can search Table 11 for this signal (*Signal or Label*), and thereby determine the pin associated with this signal (*Pin Name*), as well as the option to select when configuring this pin (*Feature / Comment*).

Once you've identified the appropriate pin, return to STM32CubeMX and configure it. As in Lab 0, it is recommended that you add labels for the pins used for the LED and push-button, as this

makes it easier to refer to them in your source code. Generate your code and open the project in STM32CubeIDE.

Lighting the LED when the Button is Pressed

Now, write code that turns on the LED when the button is pressed. The simplest code to implement this is a while loop that continuously checks (or *polls*) the status of the button. If the button is pressed, the LED is set to on. Otherwise, the LED is set to off. Section 31.2.4 of the HAL Driver User Manual lists the functions that you will need.

Note: recall that if you've labeled pins in MX, you can use these names rather than those detailed in the specifications for these functions. You can confirm the names of the pins, and see how MX generates code to set them up, by inspecting the function `MX_GPIO_Init(...)` in `main.c`.

Note: you may notice that when you run your code that the LED defaults to on, but turns off when the button is pressed. If this occurs, refer back to Figure 31 to understand why this is happening and correct it.

Further reading on GPIO pins may provide additional useful context; see Chapter 8, pp. 294 to 312 in the STM32L47xxx Reference Manual for more information.

Part 2: GPIO in Analog Mode, with DAC

Configuring the Board

The second part of this lab requires that we return to MX and configure a few more GPIO pins, this time for analog output. DAC converts digital register values (i.e., integers) into analog values (i.e., voltages), e.g., to drive a speaker with an oscillating signal.

We're going to 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 is through manuals. Unlike for the push-button, the signals we're looking for don't appear in the figures of Appendix A. This time, we need Chapter 4 of the STM32L47xxx Datasheet, which describes the pinout of the chip. Looking through this manual, we observe that the chip has a single DAC with two channels, DAC1_OUT1 and DAC1_OUT2. Table 16, starting on page 60, lists all of the pins for the device, and therefore, those which correspond to these two outputs. Select each of these pins in MX, and choose DAC1_OUT1 and DAC1_OUT2 as their corresponding mode.

The second path is using 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.

In order 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)

Re-generate your code and return to IDE. Hopefully you remembered to write your code within `USER CODE BEGIN` and `USER CODE END`, and it's all still there!

Making Signals

Previously, we have read the state of a button, and written the state of an LED. Now we need to initialize and write the state of the DAC to generate signals in an audible frequency range (so we can verify the system with a small speaker).

Implement code that manually generates two signals: a triangle wave, and saw wave. Without use of interrupts (more on this later!), it is difficult to precisely time signals. However, do your best to generate oscillating signals with a period of ~15 ms (corresponding to 65 Hz, or C2 for musicians).

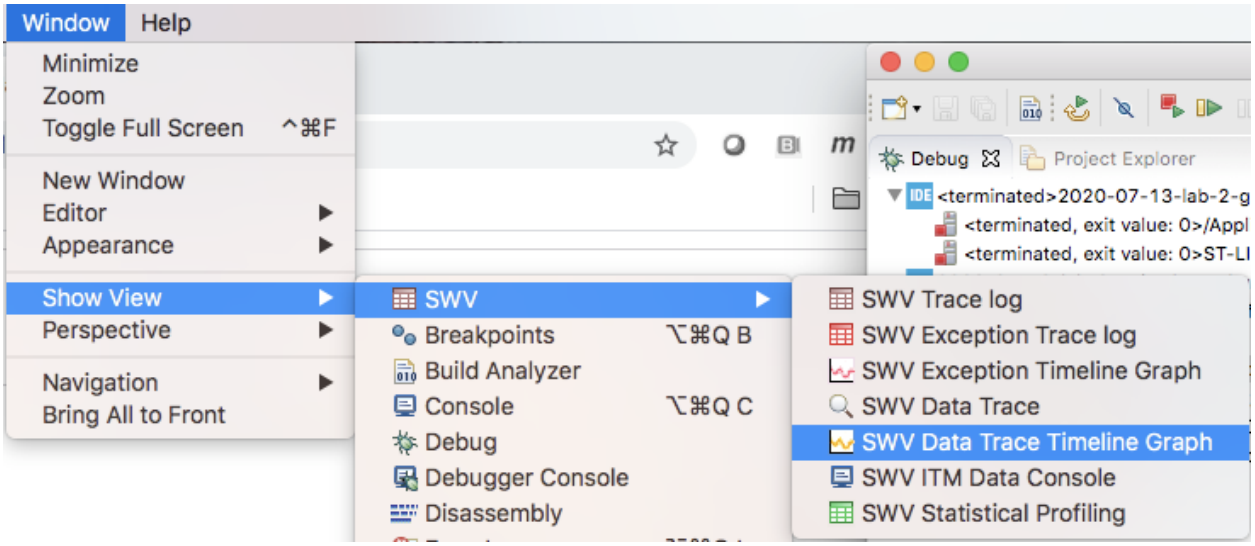
Assign each signal to a different DAC output channel. To initialize the DAC and write data to it, you'll need more HAL functions. Sections 16.2.3 and 16.2.4 of the HAL Driver User Manual list the functions you will need; they are detailed in Section 16.2.7.

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 may simplify your implementation.

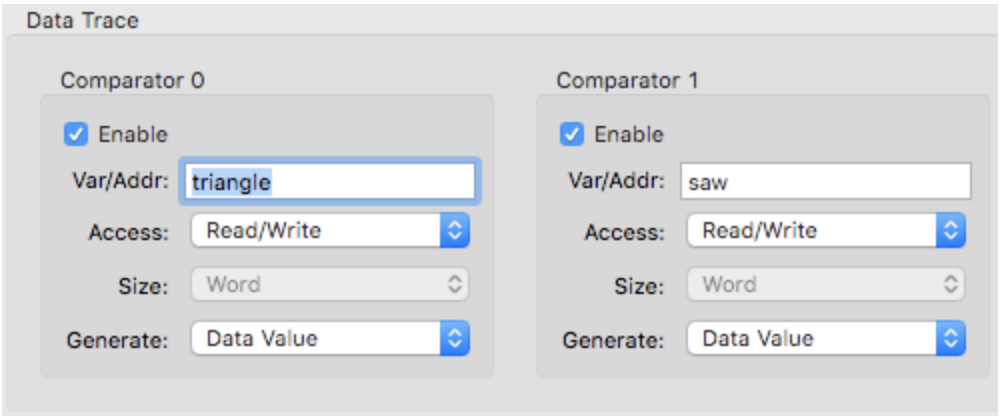
Also note that `HAL_Delay(...)` can be used to insert a delay between operations in your code. The details of its usage can be found in the HAL Driver User Manual.

Before you test your code with a speaker, it is worthwhile to use the ITM interface to verify that it is working as intended. Ensure the *Serial Wire Viewer* (SWV) is enabled and configured appropriately in the debugger configuration. (See Lab 1.) Since we'll use the ITM's data trace functionality this time, no code modifications are required (e.g., to timestamp events as in Lab

1). 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.



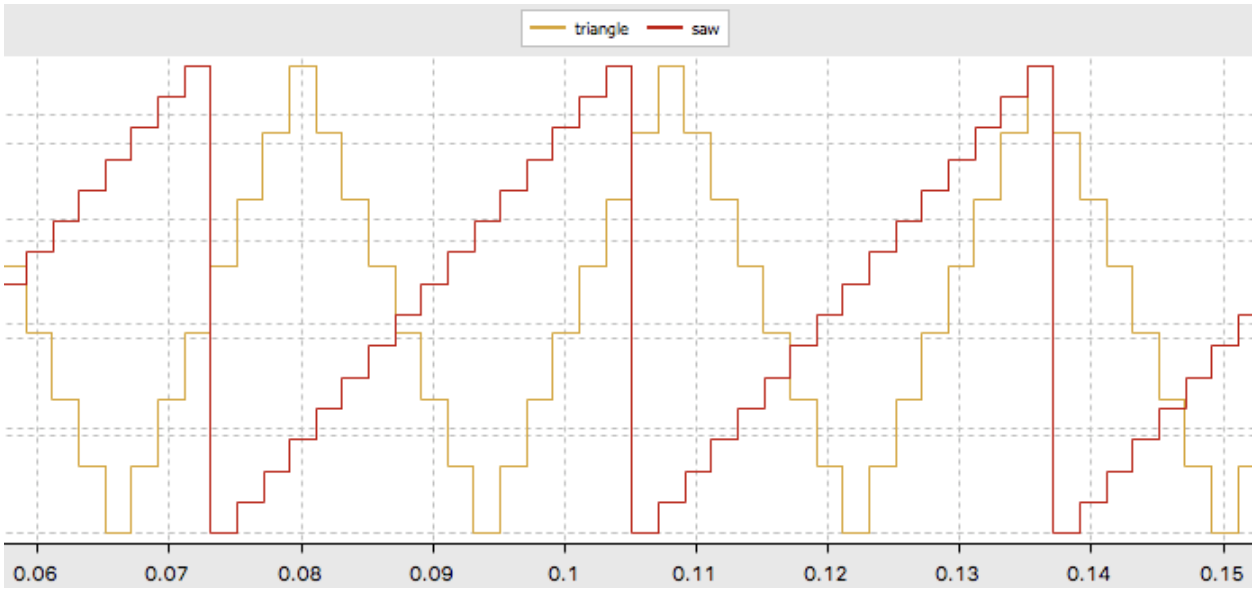
Before resuming execution, we need to configure (wrench) and then start recording (red button), just like in Lab 1. However, this time the configuration of the Serial Wire Viewer looks a little different. Enable Comparator 0 and Comparator 1, and write the names of the variables you wish to monitor in *Var/Addr*. In my case, the variables that hold the current signal values are `triangle` and `saw`.



If you try to specify variables when 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

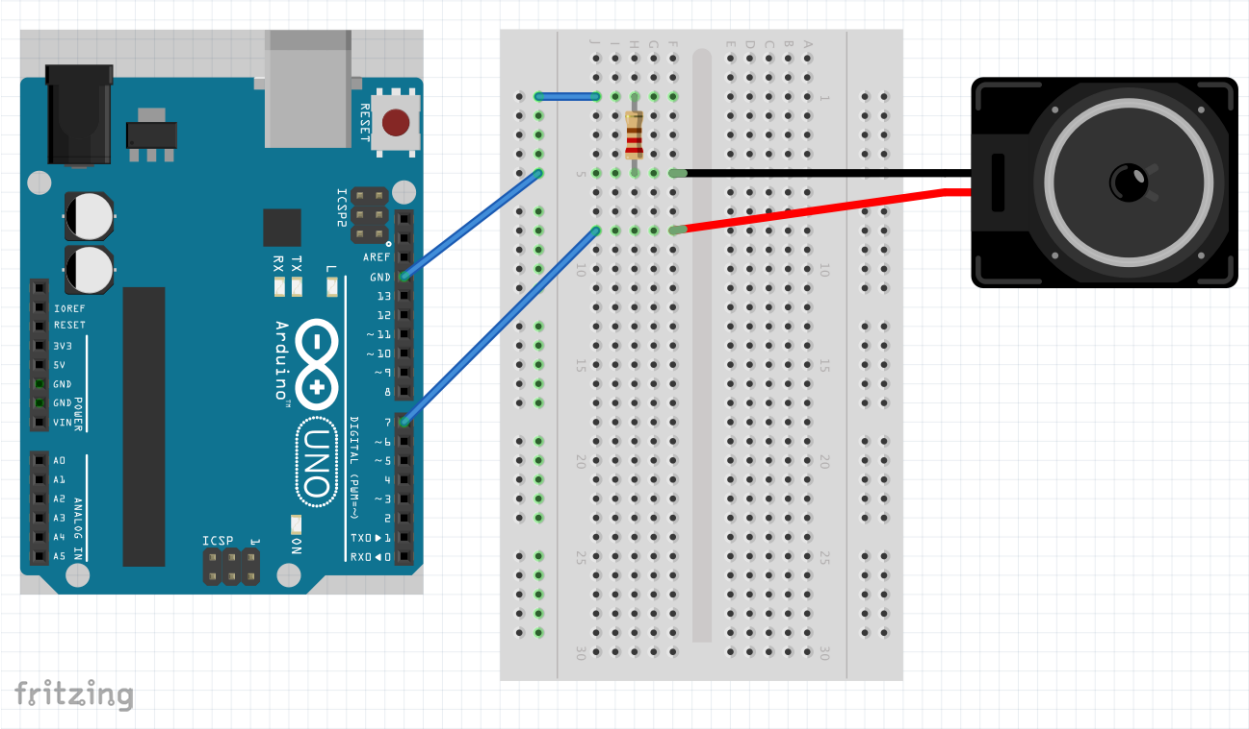
zoom in a bit in order to distinguish your triangle and saw waves. In my case, you will observe that the period of the two signals is not exactly the same. We'll achieve more precise timing in later labs when we use interrupts.



You will notice that LED1 blinks when this part of your project is running. Can you figure out why?

Making Sounds

Now that we've verified that the waveforms look about right, it's time to wire a speaker. Using the components available to you, wire your speaker. Note that (1) an Arduino Uno is pictured. Your board and the Arduino Uno have the same external interface (A0-A5 and D0-D15, plus assorted other pins). (2) The speaker that is pictured is different from yours; yours will, however, fit perfectly into the breadboard with the indicated spacing. (3) The resistor is between ground and the speaker in order to both limit the current at the GPIO, as well as the power at the speaker, in order to protect both devices.



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. (See Lab 1.) As before, trace the values before driving the speaker.