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, (c) how to use the debugging interface to plot variables as they change with time, and (d) how to take input from an analog to digital converter.

Deliverables for demonstration

- C implementation of LED lighting on button press.
- C implementation of triangle, saw, and sine signals.
- C implementation scaling ADC outputs to calculate CPU temperature in degree C.
- C implementation of temperature-dependent output to the speaker.
- C implementation of a final application that integrates the above deliverables.

Grading

- LED lighting on button press
  - 10%
- C implementation of signals
  - 10% triangle
  - 10% saw
  - 10% sine
- C implementation scaling ADC outputs to calculate CPU temperature in degrees C
  - 20%
- C implementation of temperature-dependent output to the speaker
  - 20%
- C implementation of final application
  - 20%

Changelog

- 24-Sep-2023 Changed the target output frequency to 1-2kHz from 65 Hz; this should make the output easier to hear. Also, be sure to check your DAC output with an oscilloscope before you connect your speaker!
- 18-Sep-2023 Initial revision.

Overview

In this lab, we'll take a more in-depth look at GPIO, and introduce both digital to analog conversion and 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. Then, we'll configure ADCs to read the internal reference voltage used for analog to digital conversion, and the internal temperature sensor, and subsequently use these values to determine the temperature of the processor in degrees Celsius. Finally, we'll use the push-button to control an application that can switch between two modes: one where a fixed output is sent to the speaker, and another where the output depends on CPU temperature.

Resources

ARM Cortex-M4 Programming Manual
B-L475E-IOT01A User Manual / B-L4S5I-IOT01A User Manual
HAL Driver User Manual
STM32L475VG Datasheet / STM32L4S5VI Datasheet
STM32L47xxx Reference Manual / STM32L4+ Reference Manual

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

Configuring the Board

Initialization

As before, start a new project, 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. (*For the L4S5I variant, this information is in Section 6.12, Table 4 of the manual.*)

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. (*Interestingly, there is no such schematic for the L4S5I variant, but the signal name is the same. How would you figure this out on your own? Who knows?!*)

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, 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.

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, 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 the code generated 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 in the STM32L47xxx Reference Manual (*same chapter for the L4S5I variant*) for more information.

Part 2: GPIO in Analog Mode, with DAC

Configuring the Board

The second part of this lab requires that we 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. (*This is also found in Chapter 4 for the STM32L4Sxx Datasheet.*) Your chip is using the LQFP100 package.

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 (*Table 15, page 77*), lists all of the pins for the device, and therefore, those which correspond to these two outputs. Select each of these pins (according to their *Pin name*) in STM32CubeIDE, and choose DAC1_OUT1 and DAC1_OUT2 as their corresponding mode.

The second path is looking in the *Pinout & Configuration* tab, which 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).

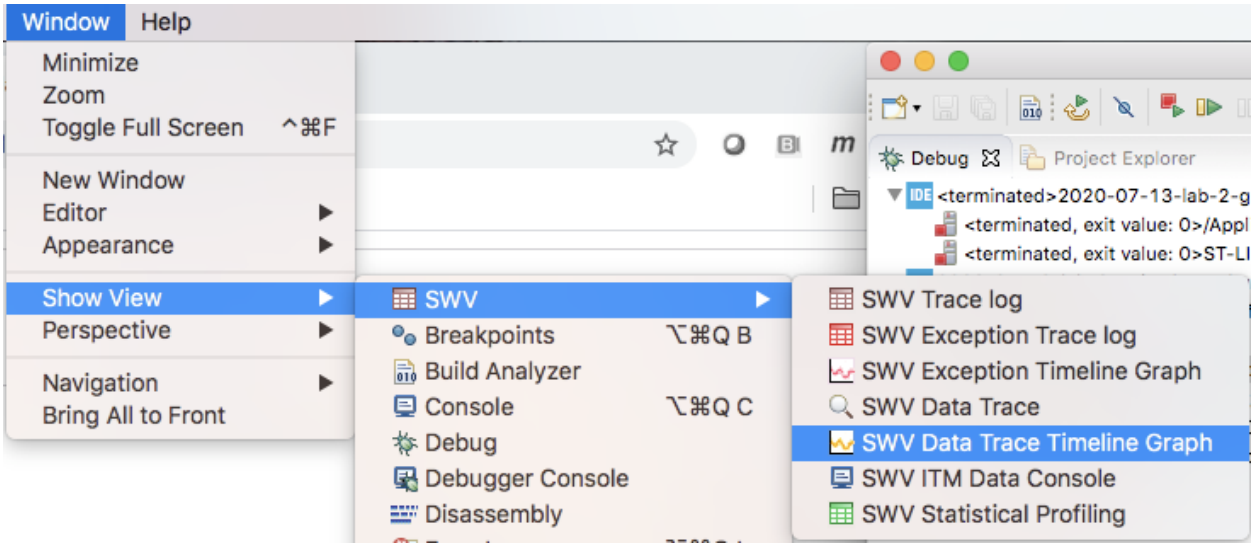
Implement code that manually generates two signals: a triangle wave, and saw wave. Without use of interrupts (more on this later!), it is difficult to precisely time signals. However, do your best to generate oscillating signals with a period of ~500 µs - 1 ms (1-2kHz, or C5 to C6). *You may find it difficult to achieve this, do the best you can.*

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.

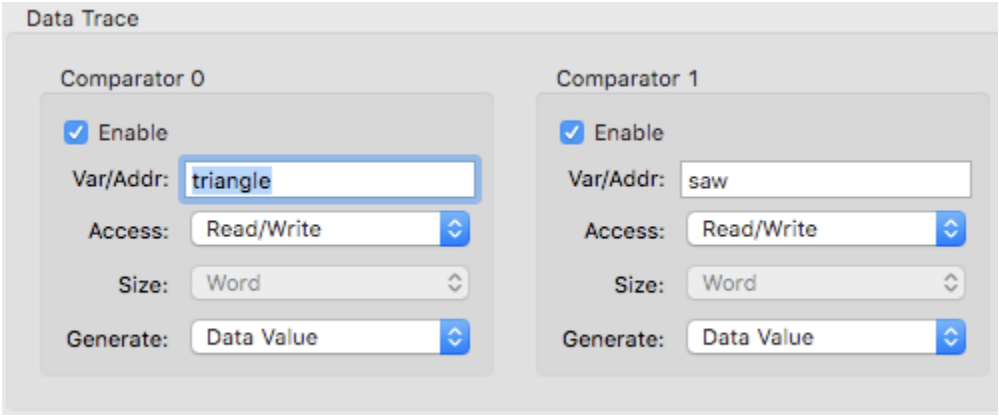
Note that the DAC can operate with either 8-bit (0 to 255) or 12-bit (0 to 4095) precision. You make this choice with parameters passed to the HAL driver. Recall that 8- and 16-bit integer data types are available (`uint8_t` and `uint16_t`), and may simplify your implementation.

Also note that `HAL_Delay(...)` or a `for` loop can be used to insert a delay between operations in your code. The details of the usage of `HAL_Delay` 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.

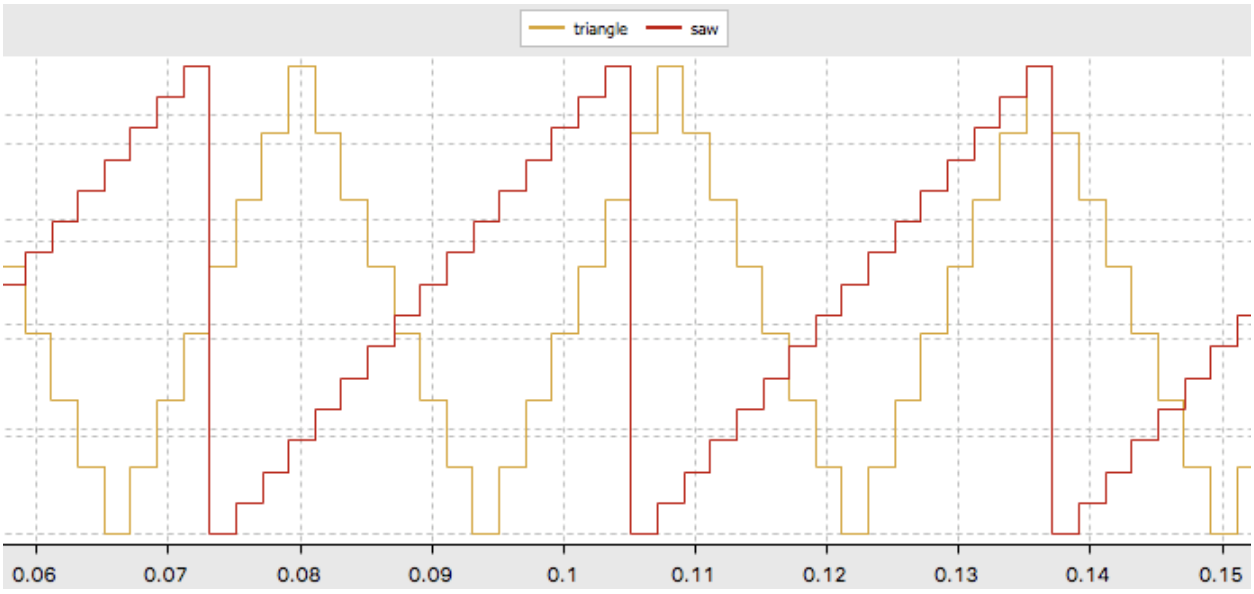


Before resuming execution, we need to configure (wrench) and then start recording (red button), just like in Lab 1. However, this time the configuration of the Serial Wire Viewer looks a little different. Enable Comparator 0 and Comparator 1, and write the names of the variables you wish to monitor in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



If you try to specify variables 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.

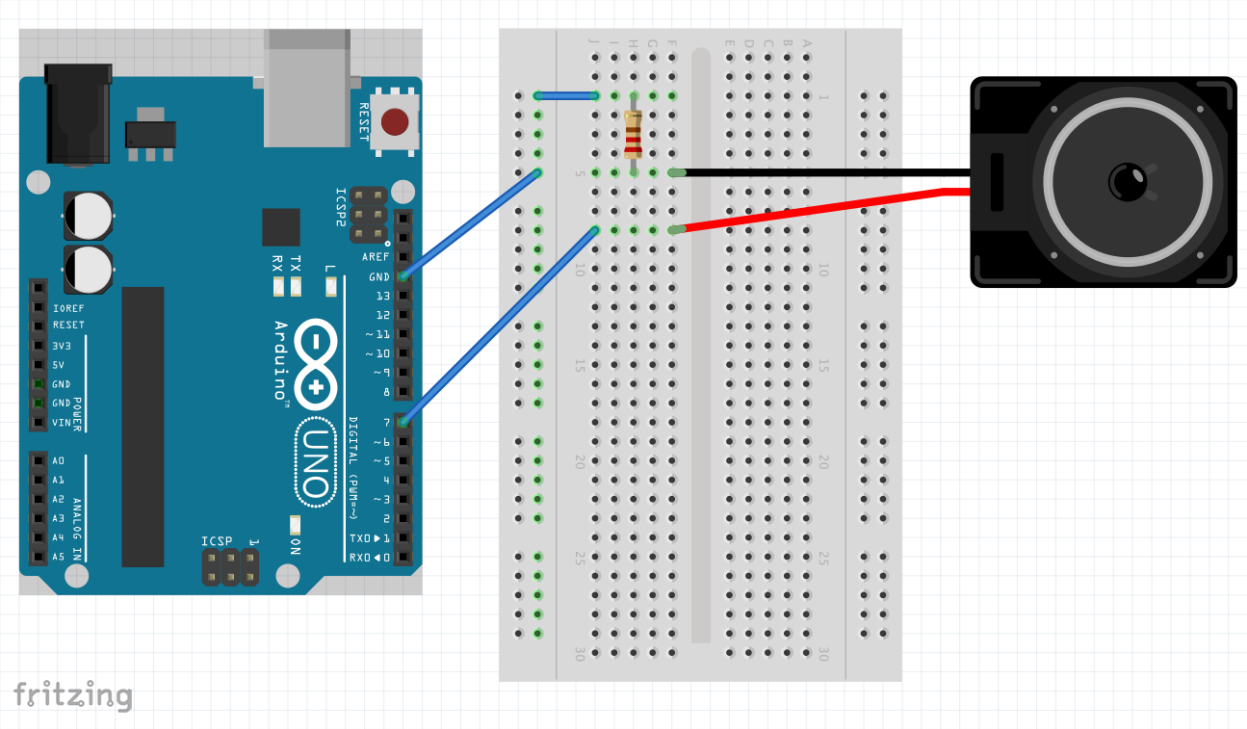
When you resume execution (don't forget to record), if everything is working properly, the data trace will rapidly fill with oscillating signals. Note that at our target frequency you may have to zoom in a bit in order to distinguish your triangle and saw waves. In my case, you will observe that the 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 verify the signal with an oscilloscope. If the signal appears as expected, wire the speaker to the DAC. 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 with SWV and check with an oscilloscope before driving the speaker.

Part 3: Analog to Digital Conversion

Before we configure the board for this part, it is important to establish a bit of context. The code for this lab is straightforward; configuring the ADC properly so it produces valid output, and manipulating that output to recover the real value it represents, is not.

ADC Sample Timing

Analog to digital conversion takes a sample (in time) of a signal (usually a voltage), and converts it into a binary representation. The ADCs available on the STM32L475VG (and STM32L4S5VI) can produce 6- to 12-bit representations. More bits means more *precision* (a smaller voltage range is represented by a single digital value); in general, higher precision conversion requires more *time*. That's because the ADCs are *successive approximation* ADCs, which refine their conversion over a series of steps. We'll cover that in greater detail in a lecture later; the key constraint for this lab is that the ADCs *must be given sufficient time to do their work*. It's up to us to figure out (a) how long the ADC needs, and (b) how to configure the board accordingly.

To get started, for an overview of how the internal temperature works, read Section 18.4.32 in the STM32L47xxx Reference Manual. (*Section 21.4.32 in the STM32L4+ Reference Manual.*) Note (2) under *Reading the temperature*, and find the appropriate sampling time for reading the temperature sensor in the STM32L475VG Datasheet (*STM32L4S5VI Datasheet.*)

Now read Section 18.4.16 in the STM32L47xxx Reference Manual. (*Section 21.4.16 in the STM32L4+ Reference Manual.*) Note the assumptions about (a) sampling time, and (b) clock frequency. When configuring the board, we'll have to pick a clock frequency for the processor and for the ADC, and choose the sampling cycle for the ADC. For an example of how ADC sampling cycle relates to sampling time, see Table 67 in the STM32L475VG Datasheet. (*Table 73 in the STM32L4S5VI Datasheet.*)

Between the sampling time requirements for the sensor, and configuration options for the ADC, you should now have enough information to configure your board properly.

ADC Output Scaling

ADCs convert voltages into a digital representation by comparing the analog input with an internal reference voltage. Sensors have a range of analog output; under ideal circumstances, the sensor and ADC are calibrated such that this range maps onto the full range of the ADC's digital output. This calibration occurs under an assumption about the internal reference voltage used by the ADC; understanding calibration conditions, and how they differ from operating conditions, is essential for writing software that correctly interprets sensor values (binary numbers scaled according to what is sensed relative to an internal reference voltage) as physical quantities (e.g., temperature).

For example, the processor temperature sensor in your development board has been factory calibrated at two temperatures. This means that the behavior of your sensor has been characterized under controlled conditions: if the reference voltage is X, and the temperature is Y, the ADC outputs Z. Given the ADC output at these temperatures, and the assumption that sensor output voltage varies linearly with temperature, we can interpolate to determine operating temperature. Note that these ADC outputs are taken assuming an ADC resolution of 12 bits.

In order to properly scale the ADC output for the temperature sensor, we need (a) these constants, and (b) an equation to relate them. The equation you've seen; it's in Section 18.4.32 (*Section 21.4.32*) of the STM32L47xxx Reference Manual (*STM32L4+ Reference Manual*), noted above. The equation also lists the various names of constants that you'll need. Note: the calibration output ADC values *are stored in the memory of your processor*. The STM32L475VG Datasheet (*STM32L4S5VI Datasheet*) has more information.

Additional notes:

- The constants in the reference manuals may have different names in source code and headers used by STM32CubeIDE; you'll have to do some detective work to figure it out. (Hint: the names in the reference manuals do appear in the source and headers, even if they aren't used to identify the constants and addresses we're looking for.)
- You'll need to use the internal voltage reference sensor, too, which has also been calibrated as described above; repeat the above process for the internal reference voltage. Section 18.4.34 in the STM32L47xxx Reference Manual (*Section 21.4.34 in the STM32L4+ Reference Manual*) is a good place to start.

With information on calibration conditions, and where to find calibrated ADC outputs, you should now have the information you need to write code to scale the ADC output for the temperature sensor.

ADC Configuration

You'll need to configure two ADCs: one to read the internal temperature sensor, and another to read the internal voltage reference. You'll find the ADCs on the left side of the *Pinout & Configuration* tab, under *Analog*; the STM32L47xxx variant has three, ADC1-ADC3. Each ADC has a number of channels that are available (under *Mode*); IN1-INX should be disabled in each case. Enable the temperature sensor channel in one ADC, and the Vrefint channel in another.

The STM32L4+ variant only has two ADCs, and Vts and Vrefint are both connected to ADC1. Fortunately, there is built-in support for the ADC to iteratively sample each on successive calls to the appropriate functions. To get the ADC configured for this mode of operation, first enable both the temperature sensor and internal VREF channels for ADC1 (under *Mode*). Then, set the number of conversions to 2 (under *Configuration* and *Parameters Settings*). Now enable *Discontinuous Conversion* and set the number of discontinuous conversions to 1. Set the number of conversions to 2, and confirm that one channel (Vrefint or Vts) is associated with

each rank. Finally, the *End of Conversion Selection* should be set to *End of single conversion.* (With thanks to Jacoby Roy.)

Under *Configuration*, and *Parameter Settings*, you'll see that an ADC is the most complex peripheral we've configured to date. Under *ADC_Settings*, make note of the options for *Resolution*. Under *ADC_Regular_ConversionMode* > *Rank* you'll find options for *Sampling Time*. You need to choose a sampling time for each ADC that satisfies sampling time guidelines for the sensors and takes the ADC clock frequency into account. The ADC clock frequency can be changed under the *Clock Configuration* tab.

Note: you can empirically test if your ADC has too short a sampling time. If you increase the sampling time, and the sampled value increases significantly, then you need to give the sensor's output more time to converge.

Reading ADC Output

The HAL Driver User Manual describes how to make use of the ADC API in Section 7.2. Note that code generation takes care of configuring the ADC; all we need to do is follow the instructions for how to perform ADC conversion using *polling*. *Polling* means waiting for the ADC to finish its conversion; alternatively, interrupts can be used to notify user code when a conversion has been completed. This is the subject of the next lab.

If you are using the STM32L4+ variant and discontinuous conversion, note that the first sequence of conversion function calls will use Rank 1 and the second will use Rank 2. Additional conversions will alternate between the two. (With thanks to Jacoby Roy.)

Write code to poll each ADC approximately every 200 ms. Verify your choice of ADC sampling time; observe the effect of changing ADC resolution.

Scaling ADC Output

Now extend your code to calculate temperature from the temperature sensor's ADC output.

Notes:

- You need to read the calibrated ADC outputs from memory; the calibration temperatures are defined as constants, but can also be found in documentation.
- You need to scale the calibrated ADC outputs based on how the internal reference voltage differs from calibration conditions. Temperature sensor output is linear with respect to reference voltage.
- You need to scale the ADC output for the temperature sensor whenever you use an output resolution other than 12-bit to account for the difference in output range.
- And be careful with operator order, data type casting, etc!
- You can check your implementation using the ADC calibration output values.

The scaled temperature value is expected to be close to room temperature. A blow dryer will be available in the lab to (carefully) heat the board and observe that internal temperature readings change.

Part 4: Putting It All Together

Now combine the operation of all of the above in a single application.

- By default, the application should play a fixed sound of your choice (triangle, saw, sine).
- On button press,
    - The LED should turn on, and
    - The sound should change to be a function of the temperature sensor.
- On subsequent button press,
    - The LED should turn off, and
    - The sound should change back to a fixed sound. The fixed sound should rotate among triangle, saw, and sine.

At all times, the output to the DAC should be displayed in the data trace timeline graph; you will also be expected to verify the output with an oscilloscope during your demo.

Deliverables

Your demo is limited to 10 minutes. Be sure to highlight top-level software structure and program flow. When applicable, it is useful to highlight that your software computes correct partial and final values.

Your demo will be graded by assessing, for each part above, the correctness of the observed behavior, and the correctness of your description of that behavior.

Grading

The breakdown of grading is as follows:

- LED lighting on button press
  - 10%
- C implementation of signals
  - 10% triangle
  - 10% saw
  - 10% sine
- C implementation scaling ADC outputs to calculate CPU temperature in degrees C
  - 20%
- C implementation of temperature-dependent output to speaker
  - 20%
- C implementation of final application
  - 20%

Each part of the demo will be graded for (a) clarity, (b) technical content, and (c) correctness:

- 1pt *clarity*: the demo is clear and easy to follow
- 1pt *technical content*: correct terms are used to describe your software
- 3pt *correctness*: given an input, the correct output is clearly demonstrated

Submission

Please submit, on MyCourses, your:

- Source code used to demo (only files you modified, including IOC file).