ECE342, Lab 3, 2024
University of Toronto

Numerical Formats and Writing Efficient Embedded Software

Fabián Torres Álvarez, Karthik Ganesan, Juan Camilo Vega, Prof. Bruno Korst

Introduction

This lab will introduce you to several new concepts, such as :

1. using the timer peripheral to measure execution time,
2. different ways of producing sinusoidal and square waves and
3. using fixed-point numbers for real values.

We will start with the timer peripheral which can be used to measure the execution time for different segments of your code.

At the end of this lab document, you will find a table that you will need to fill out. You must show your TA the completed table when you are being marked.

1. Measuring execution time

The timer is a dedicated hardware counter that you use to how many cycles an operation takes. The timer is configured in a similar manner to the other peripherals you have used so far. You can use the `HAL_TIM_Base_Start()` function to enable the timer. You can get the current timer value using the `__HAL_TIM_GET_COUNTER()` function.

To measure execution time, save the timer value before the operation starts. Then, subtract the saved value from the the timer value once the operation ends. By default, the timer peripheral runs on the same clock as your processor and increments every clock cycle. However, the timer only uses a 16-bit register, so you can only count up to 65,536 cycles before the counter rolls over.

To increase the number of cycles you can count, the timer allows you to use a clock divider to slow the input clock to the timer. This is called the `prescalar` and can be set to any value from 0 to $65,536$. The input clock to the timer is then divided by $prescalar + 1$. By default, the *prescalar* is set to 0, which means the clock is divided by 1 and therefore unchanged. You should adjust the prescalar when needed to count operations that take more than $65,536$ cycles.

Use the timer to measure the execution time for different segments of code. For example, write a `for` loop to sum the numbers from 0 to 99. How long does this take? Vary the values and see if you get a linear relationship between the loop bounds and the execution time.

2. Producing a sine wave

In Lab 2, you used the *Wave Gen* capability of the oscilloscope to produce a sine wave. You then used an ADC to read that sine wave and output it using the DAC. You must now create your own sine wave, without using an ADC. You will do this in two different ways in this lab:

1. using the `sin()` function in C, and
2. using a look up table.

2.1 Using the sin() function

First, you must make a sine wave using the `sin()` function from the `<math.h>` include file. Inside your `while(1)` loop, iterate over one full period of the sine function and output each point using the DAC. Keep in mind that the `sin()` function accepts inputs in radians. As a baseline, calculate one output for every 0.01 radians. You should start by determining how many points will cover a full period if the interval is to be 0.01 radians.

Also, keep in mind that the DAC has a resolution of 12-bits and only outputs positive values. You must appropriately scale the output of `sin()` so that you can use the full range of the DAC output.

In the results page, note down the frequency of the produced sine wave. Next, use the timer to measure the number of cycles taken to generate one period of the sine wave and enter that value in the table.

For this lab, the clock frequency of the CPU is set to $100MHz$. What is the relationship between the number of points you calculate and the output frequency of the sine wave?

Try it yourself: Try varying the number of points you output for one period and see how this impacts the output quality.

Try it yourself: What happens to the frequency of the sine wave if you use delays between subsequent points? Can you find a formula for the delay you need to get a sine wave of a desired frequency?

2.2 Using a lookup table

In the case above, you called the `sin()` function to produce each point within a period of the sine wave. However, calling the `sin()` function is computationally expensive, and we are calling it for every we calculate. It would be much more efficient to compute the points once, store them in a buffer (that is, an array in memory) and repeatedly output the stored points one by one.

Write code to generate points for one full period of the sine wave and store it in an array. Such an array is referred to as a 'lookup table' or LUT. You can then simply loop through this LUT to produce one period of the sine wave. You can use the same number of points you used earlier to make your results comparable.

The `sin()` function works for any value, not just values within one period. Think about how you can use the periodic nature of the sine function to do the same for your code. Once you are done, write down the frequency and the execution time for your LUT code in the results page.

3. Generating arbitrary waveforms

You know from your studies of the Fourier Series that sine waves can be used to produce (or synthesize) many other waveforms, including square waves, sawtooth waves and triangular waves.

The Fourier series equations for these three waveforms in the range 0 to 1 is provided below. You will need to scale them.

$$Square : f(x) = 0.5 + \frac{2}{\pi}\sum_{n=0}^{\infty}\frac{1}{2n+1}\sin\left((2n+1)x\right)$$

$$Sawtooth : f(x) = \frac{1}{2} - \frac{1}{\pi}\sum_{n=1}^{\infty}\frac{1}{n}\sin(nx)$$

$$Triangular : f(x) = 0.5 + \frac{4}{\pi^2}\sum_{n=0}^{\infty}\frac{(-1)^n}{(2n+1)^2}\sin\left((2n+1)x\right)$$

As you can see, we can produce all three waveforms (and many others) by summing up an infinite number of sine waves. In practice, we cannot sum an infinite number of them, but we can sum the fundamental frequency to a few of its harmonics and see that we can approximate the waveform we are trying to synthesize.

As an example, we will be generating square waves. Begin by writing code to produce a square wave using the `sin()` function. Since you cannot add infinite sine waves, you must vary the number of terms you add and observe the effect on the output.

You must analyze the impact of varying the number of terms added. How many terms do you need to add for the wave to “look square”? What is the impact on the frequency domain for each additional term you add? You must write down the frequencies while varying the number of terms on the results page. To avoid having to re-program the board each time, you can configure the pushbutton to add an additional term each time. You should be able to modify the interrupt code you used for Lab 1 to do this.

Next, you will use the look-up table approach to synthesize a square wave, but you will store a sine wave on the LUT. You do not want to store a different table for each value of $sin(x)$ as this can use up a lot of memory. Instead, you should only store values corresponding to the lowest frequency (i.e., $sin(x)$ in the equation above) in the LUT. You must then compute the other terms (i.e., $sin(3x), sin(5x)$ etc) from the LUT. What is the relationship between $sin(nx)$ and $sin(x)$ that you can use to do this? You will also need to make sure you are accessing the right values in your stored array. Write down the frequencies you get for different number of terms for this version as well.

Repeat the above with the other waveforms for which you are given the formulae.

4. Numerical representations

Thus far, we have used floating point representations for all of our calculations. However, these require dedicated floating point hardware which many embedded systems may not have. We will now explore using fixed point for our calculations, which only requires integer hardware.

You are provided with a `fixedpt.h` file which contains the code you will need to implement basic fixed point operations. You must fill in the missing code to compute basic arithmetic operations. By default, the provided code implements an $S3.28$ fixed point format.

Start by writing some simple operations using fixed point and compare the results to performing the same operations using floating point. For example, take some numbers in floating point, convert them to fixed point before doing calculations. Then convert the results back to floating point and compute the error between the original and converted values. You should try a range of values to make sure you understand the behaviour and limitations of using the fixed point format. What is the biggest number you can represent using the $S3.28$ format? See what happens if you try to store a number bigger than this in the `fixedpt` datatype.

Finally, vary the fixed point format to see the difference in the output quality. Try other representations which use a shorter datatype, such as $S3.12$ (which uses 16-bit ‘short int’) and $S3.4$ (which uses 8-bit ‘char’). What is the impact on error when switching to these smaller datatypes?

Try it yourself: Measure the difference in cycle count between $S3.28$ and $S3.12$. Can you think of a use case where $S3.12$ would be preferred over $S3.28$

4.1 Generating fixed point waves

Calculate points for a sine wave and store them using a fixed point format. Think carefully about how the stored points correspond to the values you should output to the 12-bit DAC? It is highly recommended that you print out a few values, hand-calculate what the value written to the DAC should be before coding this part. Once you get a proper sine wave, write down the frequency and execution time you get on the results page.

Next, repeat the experiments you did for generating a square wave, now using fixed point numbers. Once again, record the frequency your fixed point code while varying the number of terms being added. What do you think is the cause of the difference in frequency compared to your earlier floating point version?

5. In-lab demo

During your lab, you must show your TA the completed results page below. You must also demo your code outputting a square wave using both floating point and $S3.28$ fixed point formats. You must also be able to produce a sawtooth and triangular wave using floating point.

RESULTS

Generating sine waves

	Floating point		Fixed point (S3.28)
	sin()	LUT	LUT
Frequency			
Execution cycles			

Generating square waves

Number of terms	1	2	3	4	5	6
Floating point						
S3.28 fixed point						

Generating other waves using floating point

Number of terms	1	2	3	4	5	6
Sawtooth						
Triangular						