Sre Krishna Subrahamanian, Michael Savo

LAB 10

TIMER LAB EXERCISE:

SIGNAL GENERATOR WITH PRECISION TIMING AND BUFFERING

Contents

1 Introduction **1**

1.1 Lab overview 1

2 Learning Objectives **1**

3 Requirements **1**

4 Details **2**

4.1 Hardware 2

4.1.1 Connections 2

4.2 Software 2

5 Procedure **4**

5.1 Set up THE LOGIC ANALYZER 4

5.2 Evaluate Busy-Wait Playback Performance 4

5.3 Evaluate Interrupt-driven Playback 5

5.4 Add Queue-Monitoring LED Code 6

1 Introduction

1.1 Lab overview

In this project you will periodic interrupt timer and GPIOs (since this particular platform does not offer DAC) to generate signals which can be viewed on a logic analyzer or an oscilloscope. The timing accuracy will be improved through the use of a timer. You will then investigate the impact of delaying your task code, and how to buffer output data.

2 Learning Objectives

- Modify C program functions to generate a 25Hz sine wave.
- Evaluate the systems timing performance for Busy-Wait Playback and Interrupt-driven Playback for the 25 Hz wave form generator.
- Write C code to monitor the queue state of the buffer and update the LEDs state based on the following: full, empty and between empty and full.

3 Requirements

In this lab, we will be using the following hardware and software:

- **Keil µVision5 MDK IDE**
  - Please see the included Getting Started with Keil guide on how to download and install Keil.
- **STM32 Nucleo-L552ZE-Q**
  - For more information, click here.
- **Logic Analyzer or Oscilloscope**

4 Details

4.1 Hardware

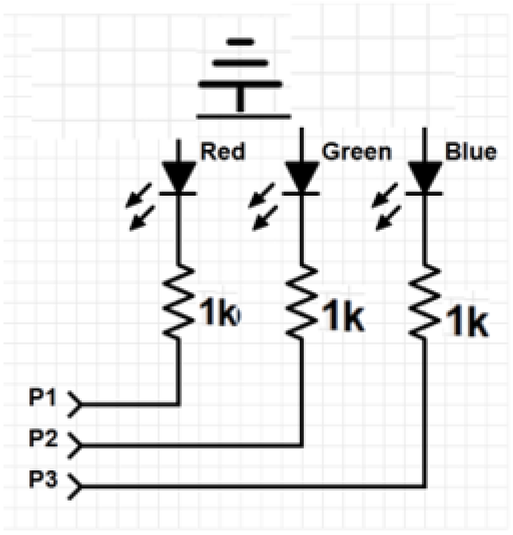
Please see the included Nucleo-L552ZE-Q pins legend (NUCLEO_L552ZE_pins.docx) for the pinout of the Arduino-included Zio connectors for CN7, CN8, CN9 and CN10.

4.1.1 Connections

Connect the logic analyzer to the signals SAMPLE, PERIOD and SW1 on the MCU board as shown in table below. Connect the logic analyzer ground to the ground on the MCU board.

As before, your button circuit should consist of 5V through a 100k resistor along with the Input to the MCU on one side, and ground on the other side.

Also as before:



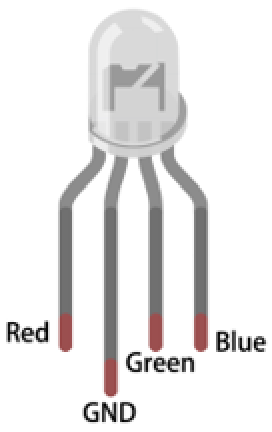


Table 1. Signals and connections

Signal Name	Description	Direction	MCU
SAMPLE	Digital	Output from MCU	PB_3
PERIOD	Digital	Output from MCU	PB_5
SW1	Button	Input to MCU	PD_15
GND	Ground	Power	
P_LED_R (P1)	Output to Red LED	Output from MCU	PA_5
P_LED_G (P2)	Output to Green LED	Output from MCU	PA_6
P_LED_B (P3)	Output to Blue LED	Output from MCU	PA_7

4.2 Software

We can use interrupt timer to toggle the SAMPLE digital signal at regular intervals, and PERIOD signal twice per period, eliminating timing jitter. The ISR (the IRQ handler) operates asynchronously from the main program, so we need to coordinate the two parts of the program. One approach is to have the main program do some calculations, for example, generating an output value, and then let the ISR to toggle SAMPLE and PERIOD signals indicating that the output value was generated. For example, the ISR could run every 20 microseconds. The main program needs to wait until the ISR has loaded it (perhaps indicated by a shared flag) and then generate the next value for the ISR.

One major problem with this approach is that it requires very precise timing control of the main program in order to work properly. In this particular lab, we will use a waveform generation as the output value. This code needs to run between each output sample, so we need to ensure that the main output value generation code runs every 20 microseconds for long enough to generate the new output value. This is easy if there is no other processing (whether main code or other ISRs), but as soon as more processing is added, we need to schedule that processing to ensure that we meet our “enough-time-every-20 microseconds” requirement.

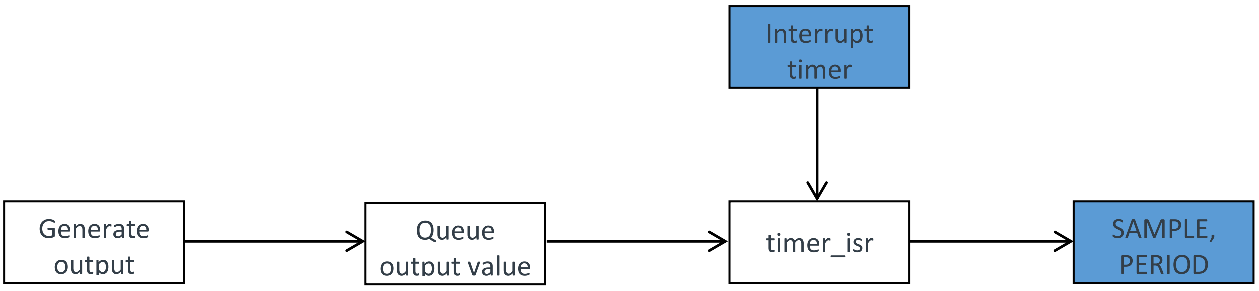


Figure 1. Communication diagram for buffered version of signal generator. Software is white, peripheral hardware is blue.

To loosen this timing requirement we will use a queue to buffer data generated by the main program and used by the ISR. The source code is in queue.c and queue.h. The main code will enqueue output values for the ISR, and the ISR will dequeue one value at a time. The buffer size determines how much looser our timing requirements become. For example, if we create a buffer with 64 samples, then our system can tolerate output value function not running for about 64 * 20 microseconds = 1280 microseconds. This will make the system design much easier.

We need to check for both overflow and underflow conditions.

- The output value generator needs to ensure there is space in the queue before enqueuing any data. If there is no space in the queue, then the function can yield the processor for other processing. No output samples will be missed if the function runs again within 1280 microseconds.
- The ISR needs to ensure the queue is not empty before attempting to dequeue data and toggling SAMPLE and PERIOD signals. If the queue is empty, then we have an “underflow” situation and we have run out of data. This is an error condition indicating that the buffer is too small given the timing characteristics of the scheduling approach used for the output value generation function. If this occurs, we need to increase the buffer size, improve the

scheduling approach, reduce the output data rate, or some combination of all three approaches.

5 Procedure

5.1 Set up THE LOGIC ANALYZER

Use the logic analyzer to view all of SAMPLE, PERIOD, and SW1 signals. PERIOD signal will help you to identify the period of the generated SAMPLE signal, and SW1 will help you identify when button presses are registered. Include waveform(s) of each step in your answer.

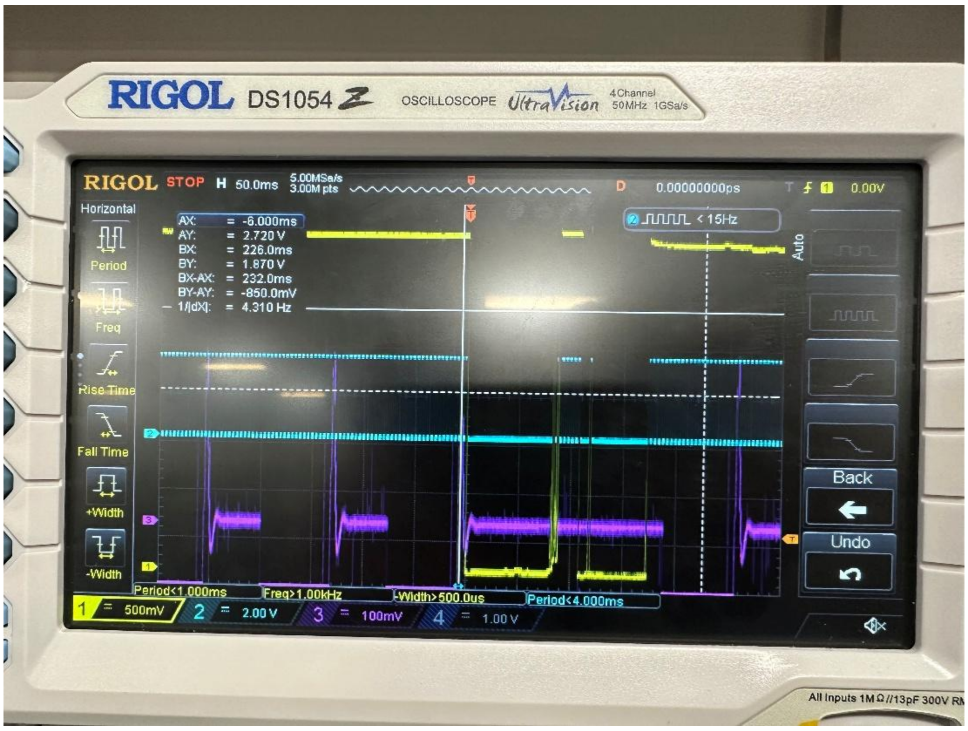
5.2 Evaluate Busy-Wait Playback Performance

Configure the code in main.c (main function) to generate a 25Hz square wave (period = 40000 us) with 1000 cycles using the function tone_play_with_busy_waiting. Make sure you locate tone_play_with_busy_waiting in the tone.c to check what each field of the function is responsible for.

1. Compile and run the code. Verify that the PERIOD output signal has the expected frequency and take a screenshot

Then trigger the scope on the falling edge of the button input. Use normal trigger sweep mode on the scope.

2. After using the trigger to freeze the screen, take a screenshot showing that you can successfully capture all three of your signals including their high and low values. SAMPLE and PERIOD should constantly oscillate except when you press the button. Once you press the button, you should see all three signals at constant values.



Here you can see that the PERIOD is 40000us and the number of cycles is 1000 shown as 1.000 KHz. This screenshot is after pressing the switch and all of the high and low values are shown.

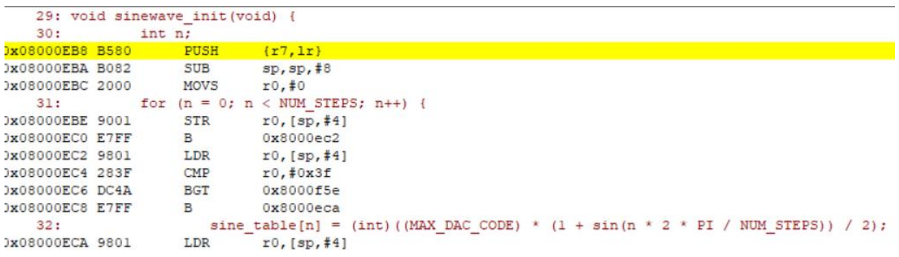
5.3 Evaluate Interrupt-driven Playback

Please read the code comments carefully! You should be able to create the interrupt code in the appropriate spot while still keeping your busy wait code, but you must do what the code comment on line 8 says for this to work properly!

Configure the code to generate a 25Hz square wave (period = 40000 us) with 1000 cycles using the function tone_play_with_interrupt. Set the buffer size (QUEUE_SIZE in queue.h) to 64 samples. Again, make sure you locate tone_play_with_interrupt in the tone.c to check what each field of the function is responsible for.

Compile and run the code. Then trigger the scope on the falling edge of the button input. Use normal trigger sweep mode on the scope (instead of “Single”). Then press and hold the button.

3. How long after pressing the button does the SAMPLE output stop changing? Does this match what you expect? Why or why not?
   a. While pressing the button, it does not change the sample output but when pressing the button it continues to stop changing for around 480us.
4. While you’re in tone.c, view the disassembly for the sinewave_init function. Comment on the context saved/restored by that function. Explain the assembly instructions associated with just the context, creation of int n, and for loop of the function (NOT the sine_table part).

a. 

```
    29: void sinewave_init(void) {
    30:         int n;
)x08000EB8 B580      PUSH     {r7,lr}
)x08000EBA B082      SUB      sp,sp,#8
)x08000EBC 2000      MOVS     r0,#0
    31:         for (n = 0; n < NUM_STEPS; n++) {
)x08000EBE 9001      STR      r0,[sp,#4]
)x08000EC0 E7FF      B        0x8000ec2
)x08000EC2 9801      LDR      r0,[sp,#4]
)x08000EC4 283F      CMP      r0,#0x3f
)x08000EC6 DC4A      BGT      0x8000f5e
)x08000EC8 E7FF      B        0x8000eca
    32:                 sine_table[n] = (int)((MAX_DAC_CODE) * (1 + sin(n * 2 * PI / NUM_STEPS)) / 2);
)x08000ECA 9801      LDR      r0,[sp,#4]
```

b. The assembly code underneath the function is initiating a stack where it returns an address on the stack. The SUB instruction is allocating stack space by clearing out the SP register by 8. Finally, the MOVS instruction is initializing the int n as shown in the C code. Now the assembly code is representing the for loop. It stores the int n with an offset of 4. The branch is there to check if it meets the conditions to stay within the loop. LDR loads the int n and compares it with 63 to check if it meets the buffer size and if n is larger than 63, it exits the loop.

5.4 Add Queue-Monitoring LED Code

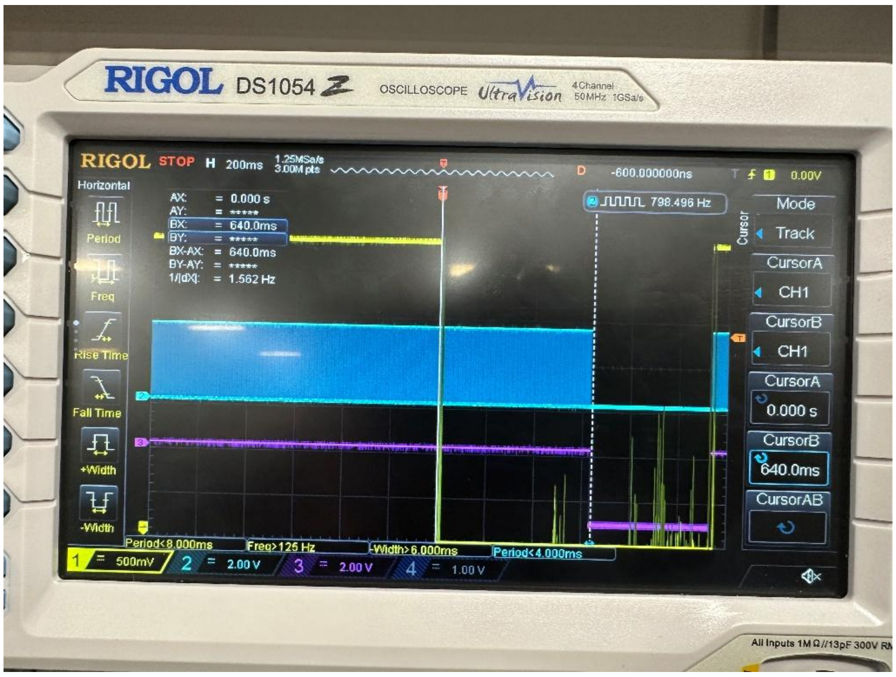
Add code to set the light of the LED according to queue state:

- Green: full (set in tone_play_with_interrupt)
- Blue: between empty and full (set in both timer_callback_isr when the queue is not empty, and in tone_play_with_interrupt when the queue is not full)
- Red: empty (set in timer_callback_isr)

Note that "tone.c" is where the functions you need to modify are located. "leds.c" has the leds_set function that you have used previously to change the color appropriately. Finally, the "queue.c" file has some nice helper functions to check on whether the queue is full or empty. The input to those functions should be &play_queue

Compile and run the code. Test to verify that it works then measure response times.

5. With the default values in place, it is very difficult to observe the situation where only the blue LED is on. Modify buffer size, period, num_step, or other variables to have a large enough buffer/queue or other situation such that the blue LED is the only one lit for a visible length of time. Comment on your process including what did or did not work.
    a. After implementing the code that should have been correct, it seemed like it wasn't working since it would transition from green to red very quickly. Trying to change the code around did not help. Adding else if instead of just else was also not helping since the code should be working. The source of this issue came from the buffer size only being 64. Increasing this to 1000 allows it to be easier to visualize the length of time it takes for the queue to empty.
6. With your modifications from 5, how long does the buffer take to empty – how long after first pressing and holding the button does the LED turn red? Does this match what you would expect based off of the buffer/queue size (visible in queue.h)? Use the logic analyzer to monitor the red LED signal (you can remove the probe that was measuring PERIOD to do this) to help you determine this and take another picture of the oscilloscope.
    a. Using the probe to measure the time it takes for the red led signal to turn on, it takes around 640ms after pressing the button. It lines up almost exactly with the SAMPLE signal. With the queue size being 1000, it is faster than I was expecting to empty. I thought it would be around a second.

b. 

7. How long does the buffer take to fill – How long after releasing the button does the LED turn green? Does this match what you would expect? How much time is needed to load each sample (in terms of time per queue size)? Use the logic analyzer to monitor the green LED signal (you can remove the probe that was measuring the red signal to do this) to help you determine this and take another picture of the oscilloscope.
   a. After releasing the button it takes about 123ms for the led to turn green. For the buffer to fill again. This does match what I expect because once it starts sampling at 40KHz it should fill up faster than a second. Since it takes 0.123s to fill up the queue, and the size is 1000, 0.123/1000 tells you the time to load each sample. Roughly 0.000123s.