***Efficient Embedded Course***

**LAB 7**

**TIMER LAB EXERCISE:**

**SIGNAL GENERATOR WITH PRECISION TIMING AND BUFFERING**

Note. The figures shown in solutions may vary subject to different experimental environments

**Issue 1.0**

Contents

[1 Introduction 1](#_Toc78890859)

[1.1 Lab overview 1](#_Toc78890860)

[2 Requirements 1](#_Toc78890861)

[3 Details 2](#_Toc78890862)

[3.1 Hardware 2](#_Toc78890863)

[3.1.1 Connections 2](#_Toc78890864)

[3.2 Software 2](#_Toc78890865)

[4 Procedure 4](#_Toc78890866)

[4.1 Set up oscilloscope 4](#_Toc78890867)

[4.2 Evaluate Busy-Wait Playback Performance 4](#_Toc78890868)

[4.3 Evaluate Interrupt-driven Playback 5](#_Toc78890869)

[4.4 Add Queue-Monitoring LED Code 6](#_Toc78890870)

# Introduction

## 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.

# Requirements

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

* **KEIL µVision5 MDK IDE**
  + Please check the Getting Started with KEIL guide on how to download and install it.
* **STM32F407G-DISC1**
  + For more information, click [here](https://www.st.com/en/evaluation-tools/stm32f4discovery.html).
* **Logic Analyzer or Oscilloscope**

# Details

## Hardware

A picture containing text, electronics, circuit

Description automatically generated

Figure 1. DiscoveryF4 pinout.

### 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.

Table 1. Switch signals and connections

|  |  |  |  |
| --- | --- | --- | --- |
| Signal Name | Description | Direction | MCU |
| DAC\_OUT | Analog | Output from MCU | PA5 |
| SW1 | Switch | Input to MCU | PA0 |
| GND | Ground | Power |  |

## 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.

Generate output values

Queue output value

timer\_isr

SAMPLE,

PERIOD

Interrupt timer

Figure . 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.

# Procedure

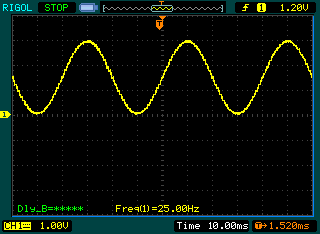
## Set up oscilloscope

Use dual channel oscilloscope to view both the switch input and the DAC output. Initially trigger the scope on the DAC output with automatic sweep.

## Evaluate Busy-Wait Playback Performance

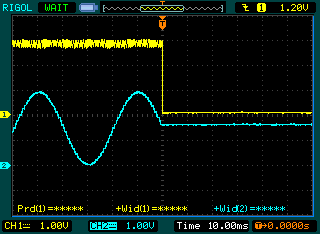
Configure the code in main.c (main function) to generate a 25 Hz sine wave (period = 40000 us) using the function tone\_play\_with\_busy\_waiting.

Compile and run the code. Verify that the DAC output generates a sine wave with the expected frequency.

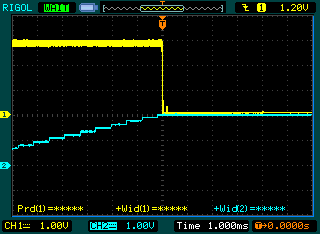


Then trigger the scope on the falling edge of the switch input. Use normal trigger sweep mode on the scope.

1. How long after pressing the switch does the DAC output stop changing? Does this match what you expect? Why or why not?



The output stops changing as soon as the switch is pressed.



A detailed view of another output stop.

## Evaluate Interrupt-driven Playback

Configure the code to generate a 25 Hz sine wave (period = 40000 us) using the function tone\_play\_with\_interrupt. Set the buffer size to 64 samples.

Compile and run the code. Verify that the DAC output generates a sine wave with the expected frequency.

1. How long after pressing the switch does the DAC output stop changing? Does this match what you expect? Why or why not?

Graphical user interface

Description automatically generated

The screenshot shows the output stops changing roughly 160 ms after the switch is pressed.

A new DAC output value is generated F\_sine\_wave\*NUM\_STEPS times per second, or at 25 Hz\*64 = 1600 Hz in this case. 1/1600 Hz = 625 us per sample. The buffer holds 256 samples, so we would expect the buffer to empty in 256 \* 625 us = 160 ms, matching the measured value.

## 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 timer\_callback\_isr)
* Red: empty (set in timer\_callback\_isr)

Do not stare at the LED for long periods of time as it is very bright. It is helpful to cover the LED with a small piece of paper to diffuse the light. You could also light the LEDs less brightly by using pulse-width modulation (extra credit!).

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

1. How long does the buffer take to empty – how long after pressing the switch does the LED turn red? Does this match what you would expect? Use the oscilloscope to monitor the red LED signal.

Graphical user interface

Description automatically generated

Background pattern

Description automatically generated

This confirms the calculations performed above, and is a different way to measure the same thing.

1. How long does the buffer take to fill – How long after releasing the switch does the LED turn green? Does this match what you would expect? Use the oscilloscope to monitor the green LED drive signal. How much time is needed to load each sample? How many CPU clock cycles does this correspond to given a 168 MHz clock rate?

A picture containing chart

Description automatically generated

The buffer takes about 162 us to fill. It takes 162 us/256 = 633 ns to load each new entry. This corresponds to about 105 clock cycles. If needed, we could reduce this with some code optimization. Note that the supplied project file may NOT have compiler optimization enabled (is at optimization level 0).

1. Why is queue sometimes not full after it has been filled?

The periodic non-full state of the queue comes from the ISR dequeueing one sample, making the queue not full. You can see that this happens roughly every 620 us, as expected. This is also why the blue led appears to be on while the queue is full.