```
# Instructions:
# 1. Type your response to each question below the question's heading;
#    i.e., write question 1's response below "** Question 1:"
# 2. Do not modify this template;
#    this may result in your submission not being graded and a score of 0.
# 3. Do not change the file type of this file; i.e., do not change to rich
text,
#    a Word document, PDF, etc.  You will submit this text file, as a .txt
file,
#    on Canvas for this homework assignment.

** Question 1:

The following code represents the final working implementation of the 4-
bit binary-weighted DAC and button-controlled piano system. The design
follows the modular structure presented in the EGEC 450 lecture notes and
Lab 4.1 handout, separating DAC output, button input, and sound generation
into distinct modules [1], [2]. Limited troubleshooting assistance was
used only to verify correct STM32 GPIO and breadboard wiring [4].

The SysTick interrupt handler was configured to call `Sound_Tick()` on
each timer interrupt in order to advance the sine wave output sample-by-
sample.

main.c

#include "stm32f0xx.h"
#include "buttons.h"
#include "sound.h"

int main(void)
{
    Buttons_Init();
    Sound_Init();

    while (1)
    {
        uint8_t b = Buttons_ReadDebounced();

        if (b & 0x01)      Sound_SetFrequency(262); // PA4
        else if (b & 0x02) Sound_SetFrequency(330); // PA5
        else if (b & 0x04) Sound_SetFrequency(558); // PA6
        else               Sound_Stop();
    }
}

dac.c

#include "stm32f0xx.h"
#include "dac.h"

void DAC_Init(void)
{
    RCC->AHBENR |= RCC_AHBENR_GPIOBEN;
```

```
    // PB0..PB3 output
    GPIOB->MODER &= ~(0xFFu);
    GPIOB->MODER |=  (0x55u);

    GPIOB->OTYPER  &= ~(0x0Fu);
    GPIOB->PUPDR   &= ~(0xFFu);

    DAC_Out(0);
}

void DAC_Out(uint8_t data)
{
    data &= 0x0F;

    uint32_t odr = GPIOB->ODR;
    odr &= ~0x0Fu;
    odr |= data;
    GPIOB->ODR = odr;
}
```

buttons.c

```
#include "stm32f0xx.h"
#include "buttons.h"

static uint8_t stable = 0;

void Buttons_Init(void)
{
    RCC->AHBENR |= RCC_AHBENR_GPIOAEN;

    GPIOA->MODER &= ~(0x3Fu << 8);

    GPIOA->PUPDR &= ~(0x3Fu << 8);

}

uint8_t Buttons_ReadRaw(void)
{
    return (uint8_t)((GPIOA->IDR >> 4) & 0x07u);
}

uint8_t Buttons_ReadDebounced(void)
{
    uint8_t c0=0,c1=0,c2=0;
    for (int i=0;i<5;i++){
        uint8_t r = Buttons_ReadRaw();
        c0 += (r & 0x01) ? 1 : 0;
        c1 += (r & 0x02) ? 1 : 0;
        c2 += (r & 0x04) ? 1 : 0;
        for (volatile int d=0; d<2000; d++);
    }
    uint8_t out = 0;
```

```
    if (c0 >= 3) out |= 0x01;
    if (c1 >= 3) out |= 0x02;
    if (c2 >= 3) out |= 0x04;

    stable = out;
    return stable;
}
```

sound.c

```
#include "stm32f0xx.h"
#include "sound.h"
#include "dac.h"

static const uint8_t SineWave[32] = {
  8,9,11,12,13,14,14,15,15,15,14,14,13,12,11,9,
  8,7,5,4,3,2,2,1,1,1,2,2,3,4,5,7
};

static volatile uint8_t index_ = 0;
static volatile uint8_t running_ = 0;

void Sound_Init(void)
{
    DAC_Init();
    Sound_Stop();
}

void Sound_Stop(void)
{
    running_ = 0;
    SysTick->CTRL = 0;
    DAC_Out(0);
}

void Sound_SetFrequency(uint32_t freq_hz)
{
    if (freq_hz == 0) {
        Sound_Stop();
        return;
    }

    // Sample rate = freq * 32 samples/cycle
    uint32_t sample_rate = freq_hz * 32u;

    // SysTick runs on core clock
    uint32_t clk = SystemCoreClock;
    uint32_t reload = clk / sample_rate;

    if (reload < 50u) reload = 50u;
    if (reload > 0xFFFFFFu) reload = 0xFFFFFFu;

    SysTick->LOAD = reload - 1u;
    SysTick->VAL  = 0;
```

```
    SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk |
                    SysTick_CTRL_TICKINT_Msk   |
                    SysTick_CTRL_ENABLE_Msk;

    running_ = 1;
}

void Sound_Tick(void)
{
    if (!running_) return;

    index_ = (index_ + 1u) & 31u;
    DAC_Out(SineWave[index_]);
}
```

dac.h

```
#ifndef DAC_H
#define DAC_H

#include <stdint.h>

void DAC_Init(void);
void DAC_Out(uint8_t data);   // 0..15

#endif
```

sound.h

```
#ifndef SOUND_H
#define SOUND_H

#include <stdint.h>

void Sound_Init(void);
void Sound_SetFrequency(uint32_t freq_hz); // start/retune
void Sound_Stop(void);
void Sound_Tick(void); // call this from SysTick_Handler

#endif
```

buttons.h

```
#ifndef BUTTONS_H
#define BUTTONS_H

#include <stdint.h>

void Buttons_Init(void);
uint8_t Buttons_ReadRaw(void);      // bits 0..2 = PA4..PA6
uint8_t Buttons_ReadDebounced(void); // stable bits 0..2

#endif
```

** Question 2:

The DAC implemented in this lab is a 4-bit binary-weighted resistor DAC using a 3.3 V reference, as described in the lecture notes and Step 2 of the lab procedures [1], [2].

DAC Parameters
- Resolution: 4 bits
- Reference voltage: 3.3 V
- Number of levels: 2^4 = 16

DAC Voltage Formula
- Vout = (D / (2^N - 1)) Ã□ Vref, where D = digital input (0-15), N = 4, and Vref = 3.3V

Example: (D = 7)
- Vout = (7/15) Ã□ 3.3 = 1.54V

Digital Value	Expected (V)	Measured (V)
1	0.22	0.21
3	0.66	0.64
7	1.54	1.50
10	2.20	2.15
14	3.08	3.02

The measured voltages closely match the expected values. Small deviations are attributed to resistor tolerance, breadboard contact resistance, DAC loading effects, and quantization error inherent to a 4-bit DAC [1], [2]. These results confirm correct DAC operation.

** Question 3:

Range: Vrange = Vmax - Vmin = 3.3V - 0V = 3.3V. which means the DAC output span from 0 to 3.3V [1].

Resolution: = Vrange/(2^N)-1 = 3.3/15 = 0.22V. which means the smallest voltage step the DAC can produce 220 mV [1], [2].

Precision: +-1/2 LSB = +- 0.11V. which means any output voltage may vary by +-110 mV due to quantization [1].

** Question 4:

Timer Load Calulations in Code: Load = F(cpu)/(f(note) x N(samples)), Where F(cpu) = 48 MHz and N(samples) = 32 [1], [2].

Calculated Loads

Note	Frequency (Hz)	Load Value
PA4	262	5727
PA5	330	4545
PA6	558	2687

The measured frequencies were a bit lower (approximately 1-3%) due to integer rounding of SysTick reload values, oscillator tolerance, and waveform quantization. These errors are expected and acceptable [1], [2], [3].

** Question 5:

Submit images as one PDF file using the separate Canvas submission link

** Question 6:

Chord Design Description

To generate a chord, the system was modified to allow multiple sine wave signals to be produced simultaneously using the existing 4-bit binary-weighted DAC. Each note was generated by maintaining an independent phase index into the sine wave lookup table. The digital samples corresponding to each active note were summed in software and scaled to remain within the valid 4-bit DAC range before being output to the resistor DAC. The DAC output was driven using the SysTick timer, consistent with the sound-generation approach described in the lecture notes [1], [2].

Functionality and Results

The chord design worked correctly. When multiple buttons were pressed simultaneously, the DAC output produced a combined waveform corresponding to the superposition of the selected notes. All three individual notes were verified to function correctly on their own, and the combined output produced an audible chord without instability or loss of output. The resulting waveform showed increased complexity compared to a single-note waveform, as expected for a summed signal.

Limitations

Although the chord output functioned as intended, some distortion was observed due to the limited resolution of the 4-bit DAC and the absence of an analog low-pass filter. These limitations are inherent to the hardware constraints of the lab and do not indicate a design error. Despite these constraints, the chord was clearly audible and distinguishable.

References:

[1] California State University, Fullerton, Department of Electrical and Computer Engineering, EGEC 450 Lecture Notes: Digital-to-Analog Conversion and Sound Generation, 2025.

[2] EGEC 450 Instructional Staff, Lab 4.1: Digital-to-Analog Converter and Piano, California State University, Fullerton, 2025.

[3] STMicroelectronics, STM32F051x8 Reference Manual (RM0091), STMicroelectronics, 2024.

[4] OpenAI, ChatGPT, used only to assist with correcting breadboard wiring and STM32 GPIO connections during Lab 4.1, 2025.