2024

EEE3095/6S: PRACTICAL 2

1. OVERVIEW

A Digital-to-Analogue Converter (DAC) converts a digital code into an analog voltage or current, and works on the general premise that $V_{OUT} = k * DigitalInput$, where k is a proportionality factor and the digital input is a number in the range of 0 to $2^{bits} - 1$. DACs have a variety of uses, with the most important among them being the conversion of digital signals into analog audio in just about every single electronic audio system in the world!

Your STM dev board already (technically) has a built-in DAC, and setting it up is quite trivial — especially with there being plenty of resources and guides available online if you ever want to test it out. So, in this practical, we will instead be *making* our own 10-bit, simple, crude DAC using Pulse Width Modulation (PWM) and a low-pass filter. You will also be introduced to pushbutton interrupts!

2. OUTCOMES AND KNOWLEDGE AREAS

In this practical, you will be using C code (and the HAL libraries) to interface your microcontroller board with a low-pass filter circuit that you will build in the lab. You will also use look-up tables (LUTs) to represent a few different analogue waveforms, set up two timers on the dev board (one timer for generating the PWM signal and one for cycling through the LUT values periodically), and then use an oscilloscope to monitor the output from your filter. The aim of this is to use the LUT values to vary the CCR (Capture/Compare Register) value, effectively changing your duty cycle.

You will learn about the following aspects:

- DACs
- Filtering
- PWM
- Pushbutton interrupts

3. HARDWARE

You will require some external hardware (in addition to your STM dev board) to properly complete this practical:

- Breadboard

- Oscilloscope
- Signal generator
- Low-pass filter components

4. DELIVERABLES

For this practical, you must:

- Develop the code required to meet all objectives specified in the Tasks section
- Push your completed main.c code to a shared GitHub repository for you and your partner. Your folder and file structure should follow the format:
  STDNUM001_STDNUM002_EEE3096S/Prac2/main.c
- Demonstrate your working implementation to a tutor in the lab. You will be allowed to conduct your demo during any lab session before the practical submission deadline.
- Write a short report documenting your main.c code, GitHub repo link, and a brief description of the implementation of your solutions. This must be in PDF format and submitted on Amathuba/Gradescope with the naming convention:
  EEE3096S 2024 Practical 2 Hand-in STDNUM001 STDNUM002.pdf
  Note: Your code (and GitHub link) should be copy-pasted into your short report so that the text is fully highlightable and searchable; do **NOT** submit screenshots of your code (or repository link) or you will be penalised.
- Your practical mark will be based both on your demo to the tutor (i.e., completing the below tasks correctly) as well as your short report. Both you and your partner will receive the same mark.

5. GETTING STARTED

As before:

1. Clone or download the (updated) Git repository:
   $ git clone https://github.com/eee3096s/2024
2. The project folder that you will be using for this practical is /EEE3096S-2024/Practical2/Prac2_student
3. Open STMCubeIDE, then go to File --> Import --> Existing Code as Makefile Project -->> Next. Then Browse to the project folder above, select it, and select "MCU ARM GCC" as the Toolchain --> Finish.
   Note: This IDE provides a GUI to set up clocks and peripherals (GPIO, UART, SPI, etc.) and then automatically generates the code required to enable them in the main.c file. The setup for this is stored in an *.ioc* file, which we have provided in the project folder if you would ever like to see how the pins are configured. However, it is crucial that you do **NOT** make/save any changes to this *.ioc* file as

it would re-generate the code in your main.c file and may **delete** code that you have added.

4. In the IDE, navigate and open the main.c file under the Core/src folder, and then complete the Tasks below.
   Note: All code that you need to write/add in the main.c file is marked with a "TODO" comment; do not edit any part of the other code that is provided.

6. TASKS

Complete the following tasks using the main.c file in STM32CubeIDE with the HAL libraries, and then demonstrate the working execution of each task to a tutor:

1. Generate three LUTs — one each for a single cycle of a sinusoid, a sawtooth wave, and a triangular wave. Each LUT should have 128 values that range from 0 to a maximum of 1023. Plot your LUTs using MATLAB or Excel (or any other graphing software) to ensure that you have the correct waveforms, then copy the values into your main.c file.
2. Assign values to *NS*, *TIM2CLK*, and *Fsignal*. *NS* is the number of samples in a LUT, *TIM2CLK* should be set to 8 MHz (which you can see in the *.ioc* file), and *Fsignal* is the frequency that we want our analog signal to have; this should not exceed 1 kHz or the cut-off frequency of your filter — whichever is lower. TIM2 is used to cycle through the LUT values.
3. Develop an equation to calculate *TIM2_Ticks* — this is the number of clock cycles taken for a new LUT value to be written, and depends on *NS*, *TIM2CLK*, and *Fsignal*.
   Note: We will be using Direct Memory Access (DMA) to change the PWM duty cycle. DMA is used to provide high-speed data transfer between peripherals and memory (or memory to memory) without any CPU actions. Check the *.ioc* file if you want to see how the DMA is configured!
4. Start TIM3 in PWM mode on Channel 3 (corresponding to pin PB0 — and yes, there is *also* an LED connected there). Then start TIM2 in Output Compare (OC) mode on Channel 1.
   Note: PB0 will thus be our dev board's output signal that will be fed into the low-pass filter.
5. Start the DMA in Interrupt (IT) mode; the source address is the Sine LUTs you created earlier, and the destination address is the CCR3 register for TIM3_CH3. Then enable the DMA transfer.
6. An interrupt has been configured on the PA0 pushbutton. Write code that changes from one LUT waveform to the next (Sine --> Sawtooth --> Triangle --> Sine…) when the button is pressed. Implement **debouncing** to avoid problems with the pushbutton not always responding to presses.

Hint: Use the HAL library to disable DMA and abort IT mode to stop the DMA transfer *before* changing the source address; thereafter, restart IT mode and re-enable DMA. Also, the *HAL_GetTick* function may be useful for debouncing!

7. Build your low-pass filter on your breadboard and test it using a signal generator and oscilloscope. Confirm that your filter attenuates frequencies above your chosen cut-off frequency.
8. Connect PB0 (TIM3_CH3) to the input of your filter, making sure that the dev board and your filter **have a common Ground** (V_{SS} on your dev board; ch). Scope the output of your filter and test all three of your waveforms as you press PA0 to change LUTs.

   Note: Remember that this is a very crude DAC, and you will likely need to scale your oscilloscope output to properly see the different waveforms in a (somewhat) stable way!

7. QUESTIONS TO CONSIDER:

- *What is the fastest frequency you can generate with your development board? What are the trade-offs?*
- *What additional electronics can you add after your microcontroller to improve the signal integrity.*
- *What is the impedance of your output pin?*
- *Why low pass filter?*
- *Consider how you could use this to drive a pure sine wave inverter.*