ECE271: Microcomputer Architecture and Applications — University of Maine

Lab #11: Digital to Analog Converter (DAC)

Week of 18 April 2022

Goals

1. Understand the basic concepts of Digital to Analog Conversion (DAC).
2. Configure DAC resolution and sample rate.
3. Use a timer to trigger DAC conversion.
4. Perform fixed-point operations.
5. Use table lookup for fast calculation of a complex function.

Pre-lab

1. Complete the pre-lab before attending lab. The pre-lab is in a separate pdf file, found on the website.

Lab Procedure

This lab involves configuring the Digital/Analog Converter (DAC) to generate a 440Hz sine wave on pin PA5.

Part A – Initial Setup

1. You can use a previous lab as a template, but in general we aren't really re-using much code from previous labs.

2. If you're using Linux, I've posted an updated template to the website that has some more definitions in the header file that will be helpful with this lab.

Part B – Set up the DAC to be software triggered

1. First make a `DAC_Channel2_Init()` function, like in Example 21-4 in the textbook.

   (a) Enable the DAC clock by setting the `RCC_APB1ENR_DAC1EN` bit in `RCC->APB1ENR1`

   (b) Disable the DACs (they have to be disabled for configuration to take effect) by clearing `DAC_CR_EN1` and `DAC_CR_EN2` in `DAC->CR`

   (c) Set the DAC channel 2 mode to be external pin with the buffer enabled (by setting the `DAC_MCR_MODE2` bits in `DAC->MCR` to `0b000`.

   (d) Select the software trigger by setting `DAC_CR_TSEL2` in `DAC->CR`

   (e) Enable DAC Channel 2 by setting `DAC_CR_EN2` in `DAC->CR`

(f) The two DAC channels output to pins PA4 and PA5 respectively. You can note that pin PA4 does not have a connector pin on the STM32L4 board, which is why we are using channel 2. (Channel 1 PA4 connects through the internal opamp, which outputs on PA3, but for simplicity we will just use Channel 2 on pin PA5).

(g) Enable the clock for GPIOA

(h) Set pin PA5 to be analog mode (so set `GPIOA->MODER` for PA5 to be analog (`0b11`)).

2. Set up a manual value on the DAC and make sure it works.

   (a) Some similar code is in Textbook Example 21-5.

   (b) Be sure to call `DAC_Channel2_Init()` from `main()`

   (c) In your while loop, have a loop that waits until `DAC_SR_BWST2` in `DAC->SR` is 0, indicating the hardware is done updating the DAC.

   (d) Next, set a 12-bit right-aligned value into the DAC `DAC->DHR12R2`. For testing, you can set this to 2048.

   (e) Next, trigger an update that will move the value to the DAC. set `DAC_SWTRIGR_SWTRIG2` in `DAC->SWTRIGR`

   (f) After doing this, a half-value output (1.5V) should appear on PA5 which you can test with a volt-meter or oscilloscope.

   (g) The example in the book does a fancier sawtooth wave output. You can test that if you want too, but it's not really necessary for the rest of the lab.

Part D – Set up an auto-triggered 440Hz Sine Wave

1. First, be sure the system timer is using the HSI 16MHz clock (see your code from the last lab)

2. Next, configure TIM4 to generate a 44.1kHz clock. Put this in a function `TIM4_Init()` which is called from `main()`

   (a) Enable the TIM4 clock by setting the `RCC_APB1ENR1_TIM4EN` bit in `RCC->APB1ENR1`

   (b) Set edge-aligned mode by clearing `TIM_CR1_CMS` in `TIM4->CR1`

   (c) Set the output to be trigger TRG0 (which we can connect to the DAC later) by first clearing the `TIM_CR2_MMS` bits in `TIM4->CR2` and then setting it to `0b100`.

   (d) Enable the trigger and update interrupts for TIM4 by setting `TIM_DIER_TIE` and `TIM_DIER_UIE` in `TIM4->DIER`.

   (e) Set PWM mode 1 output by setting the `TIM_CCMR1_OC1M` fields of `TIM4->CCMR1` to `0b0110`.

   (f) Set the `TIM4->PSC` and `TIM4->ARR` fields to give you a 44.1kHz output (you calculated this in the prelab).

   (g) Set the `TIM4->CCR1` field to give a 50% duty cycle.

   (h) Set the `TIM_CCER_CC1E` bit in `TIM4->CCER`

   (i) Finally, enable the timer by setting `TIM_CR1_CEN` in `TIM4->CR1`

(j) Also remember to enable the TIM4 interrupt in the interrupt handler with `NVIC_SetPriority()` and `NVIC_EnableIRQ()`

3. Change the DAC code to be triggered by TIM4 rather than manually

   (a) Modify your existing `DAC_Channel2_Init()`

   (b) Change the trigger setting bits in `DAC->CR` `DAC_CR_TSEL2` to be `0b101` (Timer4 TRG0) rather than `0b111` (software trigger)

4. Set up the `sin()` routine.

   (a) Create `sine_table.h` containing the sine lookup table from the prelab.

   (b) Include this file into your program using `#include "sine_table.h"`

   (c) Create a function called `int lookup_sine(int angle)`

   (d) The contents should look something like Example 21-2 in the textbook.

   (e) Our sine table is only 1/4 of the sine (0 to 90 degrees) so this function creates the other values by using symmetry and mirroring things.

   (f) Also note that a mathematical sine output goes from -1 to 1, but our values go from 0 to 4096.

5. Set up the `TIM4_IRQHandler()` to output a 440Hz sine wave.

   (a) You can base this on Example 21-8 from the textbook.

   (b) If the `TIM_SR_CC1IF` flag is set, it means the interrupt was triggered by PWM and we want to handle the DAC update.

   (c) Have the current angle stored in a global variable.

   (d) We will need to have a fractional angle. We will use X.3 decimal fixed point for this (meaning the values we use are multiplied by 1000, we are using milli-degrees).

   (e) Set `DAC->DHR12R2` to the looked up sine value. Pass `angle/1000` (to convert from fixed point to the integer part) to the `sin()` routine in the last section.

   (f) Next, add in the angle delta value to set up the angle for the next time things are called. The value for this was calculated in the prelab.

   (g) If the angle value is greater than 360 degrees (360,000 milli-degrees) make sure you wrap around to 0.

   (h) Clear the `TIM_SR_CC1IF` flag in `TIM4->SR` to ACK the interrupt.

   (i) Also ACK the `TIM_SR_U1F` flag at the end

6. Test your result

   (a) Hook up PA5 to the oscilloscope. If you have done things right you should have a 440Hz sine wave.

   (b) Take a picture/capture of your wave and include it with your README into gitlab.

   (c) Prof. Weaver has set up some old speakers in the lab if you'd like to listen to your results as well.

Part F – Something Cool

Do something cool! You can come up with something on your own, but here is a list of ideas you can use.

1. Generate a wave with a different frequency
2. Generate a Triangle wave instead of a Sine wave
3. Enable the noise or triangle features of the DAC
4. (Advanced) play a song using the DAC.
   (a) First you will have to create a list of notes to play. The textbook has some (in assembly language) in Chapter 21.8. In C this will look something like

```
#define NOTES   42

int twinkle_freq[NOTES]={
        262,262,392,392,440,440,392,
        349,349,330,330,294,294,262,
        392,392,349,349,330,330,294,
        392,392,349,349,330,330,294,
        262,262,392,392,440,440,392,
        349,349,330,330,294,294,262,
};

int twinkle_len[NOTES]={
        1,1,1,1,1,1,2,
        1,1,1,1,1,1,2,
        1,1,1,1,1,1,2,
        1,1,1,1,1,1,2,
        1,1,1,1,1,1,2,
        1,1,1,1,1,1,2,
};
```

   (b) To play the notes, in the interrupt handler first check to see if the note is done playing. The length of the note can be found in the "len" array. If you assume 120 beats per minute this means a note length of 1 would be 1/2 second (or 22,100 cycles if you are using a 44,200Hz clock). So start a counter at 22100 and decrement it each interrupt, and then you hit 0 reload the count and increment your offset into the arrays (to update the length and the frequency).
   (c) To play the proper frequency, you can calculate the stepsize to use with something like

```
f=44300000/twinkle_freq[which];
    stepsize=360*1000000/f;
```

   (d) With a few small changes to your TIM4 IRQ handler like above you can play simple songs.
   (e) To play other songs you will need some sheet music, the ability to translate it to notes, and then use a frequency table similar to Textbook Table 21-1.

Lab Demo

1. Submit your code
    - Complete a README with the post-lab answers.
    - Make sure the code is properly commented.
      This includes a header at the top of your main.c with your name and a brief summary of the lab.
    - Check your code and README into your gitlab tree.
2. Demo your implementation to your lab TA.
    (a) Demonstrate the 440Hz sine wave on the oscilloscope.

Post-Lab

- Place your answers to the question in a file Readme.md
- Submit with your code via the gitlab server.
- Questions:
    1. If we want to play the note "D" instead of "A" we would want the frequency to be 293.665Hz. To play this note, how much should the angle variable increment each timer interrupt (assuming TIM4 is set up at 44.1kHZ)

    2. In C when you include/link against the math library, you get access to a function`double sin(double x)` that calculates the sine. What difficulties might we find if we tried to use this inside of an interrupt handler?