Lab #5 – driving a DAC from DMA using a hardware timer

Overview

In this lab, we'll

- implement a DMA driver. DMA is not something that is part of the Arduino API, but we'll see that it can be very useful.
- hook up the DMA driver to a DAC, with a hardware timer driving the DAC/DMA at a much higher frequency than we could by using just the FreeRTOS tick.

What you'll learn

- You'll code a DMA driver yourself; you'll reverse-engineer an existing general-purpose-timer driver.
- You'll understand how to interconnect multiple peripherals using a timer to control a DMA transfer to a DAC.

Lab setup – hardware

We'll be using our usual Nucleo-board-on-a-breadboard set up.

Big-picture view of the software

At a high level, here's how the software for today's lab works. A hardware timer is programmed to generate triggers at regular intervals. This trigger drives DAC #1 — every time the DAC receives a trigger, it converts a new value from its DAC_DOR1 CSR to an analog voltage to drive pin PA4 (Nano pin A3). The DAC also requests that DMA send it a new piece of data. Everything then waits until the timer generates the next trigger, at which point the cycle repeats indefinitely. You can fill the DMA's "from" memory locations with whatever data values you like, building any arbitrary wave shape. You then program the DMA to operate in circular-transfer mode to send out the same buffer repeatedly to the DAC, so that the waveform repeats indefinitely.

Note that this lab doesn't use FreeRTOS at all, and *lab5_main.c* thus doesntt create any tasks or turn on the scheduler. So if you want, you can create the directory structure for this lab by simply making a copy of lab #1 and then replacing *lab1_main.c* with *lab5_main.c*. Of course, you can also duplicate a lab that uses FreeRTOS; you'll just be compiling some unused files.

Write the DMA driver

Your first job is to write the DMA driver. You have the function skeleton *setup_DMA* (). Fill it in with the appropriate code based on what we learned in class, supplementing it as usual with the STM32L432 reference manual and *stm32l432xx.h*. Compile and build as usual.

What you don't have to do

While the lab relies on a general-purpose timer to trigger the DAC at every data point, that code is already written for you. The same is true for the DAC driver.

Generate scope pictures

Generate scope pictures for both the triangle and square wave data sets. You can switch data sets by commenting out all but the correct definition of *wave_mem*, and setting the parameter to *setup_DMA*() to the correct transfer size. In each case, generate your pictures showing two scope channels: not only the DAC output on Nano pin A3, but also the GPIO output driven by timer #2 on Nano pin A4. The GPIO output will help ensure that the timer is working, and also serve as a timing reference to better understand your DAC output.

Lab checkout

Show a TA your two screen screen shots (one for the triangle wave and one for the sine wave) to be sure they look reasonable.

Questions to write up and turn in:

1. The function *setup_TIM2_channel*() sets up timer #2 to drive a frequency-divided counter output on its channel 1, as well as driving its TrgO output. These are two separate pieces of counter functionality, and are programmed separately. Are both of them needed to trigger the DAC? If only one is needed for the DAC/DMA functionality, then what might be the purpose(s) of the other one in this lab?
2. If the DAC doesn't display the desired waveform, there are at least two explanations. It could be that the DAC is receiving triggers from counter #2 just fine, but you've set up the DMA incorrectly. Or perhaps the DMA is working fine, but the counter is not sending triggers to the DAC. Look through the features of the DAC – can you find any nice features that would help you to easily disprove this last possibility? (Of course, since the counter and DAC code are provided to you, it's hopefully unlikely that this would actually be the problem!).
3. Let's see how well you've understood how timers work. Given the main clock frequency, and the timer settings (which are described in comments in the code, so you don't have to dig into timer-CSR programming), can you predict the DAC triangle-wave frequency? Hopefully that matches what you see on the scope! (Note that the timer actually runs on what's called the APB1 clock, but in our case that's the same as the main 80MHz clock).