EE-156 Lab 5

Emmett Roberts & Griffin Faecher

11/3/2025

Driving a DAC from DMA using a hardware timer

Professor Joel Grodstein

Tufts University

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? (It had a very important purpose for me when I wrote and debugged this lab). (15 points)

   The counter is what determines when the TrgO output occurs and that TrgO output is necessary to drive the DAC. So only the TrgO output is necessary. The channel 1 output is purely for debugging visual purposes. It allows the channel 1 output to drive a pin to show the counter reset frequency. The DAC only needs the TrgO output not the channel 1 output.

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!). The nicest feature for this (at least IMHO) is in the RM0394 reference manual, but not in the class slides. (15 points)

   If you think the DMA is working fine and the counter is not sending triggers to the DAC, you could look at the DAC software trigger. However, this is not very relevant time wise as the timer is going faster than the 1ms interrupt clock so you couldn't see updated values of the TrgO output in the register.

   Instead DAC has a triangle wave generation feature that essentially manually creates a triangle wave based on realtime TrgO inputs to the DAC. It will increment/decrement once every trigger input varying from a DC signal creating a triangle wave. This triangle wave's period is 2 trigger inputs. Therefore, you can tell if the DAC is receiving trigger inputs in real time based on the triangle wave being generated by the DAC.

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). (15 points)

   We initially set the clock to run at 80Mhz. When we initialize the timer 2 channel, we set the prescaler to 100, meaning the clock driving the DAC trigger to 800khz. We also initialize the reload value to 100, meaning that every 100 cycles, the counter reloads and a trigger is sent out. So, this trigger is sent every 8 khz. Every time this trigger goes out, the dac switches which value it outputs from the array of values we have that create the triangle wave. Because there are 14 values in the array, that means that it takes 14 triggers to go through one period of the wave. This means that the frequency of the wave should be around 8000 / 14 = 571.429 hz. On our scope, we get that the frequency is $\frac{1}{.00177} = 564.97hz$, which is very close to our calculated value.

```
// Now set up timer 2, channel 1. We'll use this to trigger the DAC.
   setup_TIM2_channel (100, 100, 1);    // prescaler, reload #, channel
```

Figure 1: Code Setting up the timer with a prescaler and reload value of 100

```
static void setup_DMA (uint8_t *DMA_mem_addr,
                      volatile uint32_t *DMA_periph_addr,
                      unsigned int DMA_data_size, int circular) {
   // The DMA controller is chapter 11 of the RM0394 reference manual.

   // Turn on the clocks to the DMA controller.
   RCC->AHB1ENR |= RCC_AHB1ENR_DMA1EN;

   // DMA ISR (interrupt status reg) needs no programming.

   // Configuration register for DMA channel x (DMA_CCRx); one CSR for each
   // channel.
   // PSize is pretty much irrelevant (at least for 8b DAC writes). E.g., 2B
   // MSize and 1B PSize just seems to mean losing the 8 most-significant bits
   // of each memory read, rather than very efficiently having 1 mem read ->
   // 2 DAC writes.
   DMA1_Channel3->CCR &= ~DMA_CCR_EN;
   DMA1_Channel3->CCR |= (DMA_CCR_CIRC | DMA_CCR_DIR | DMA_CCR_MINC);

   DMA1_Channel3->CCR  &= ~(DMA_CCR_PINC | DMA_CCR_MEM2MEM | DMA_CCR_PSIZE |
DMA_CCR_MSIZE);

   //DMA_CCR_PINC and memtomem and DMA_CCR_MEM2MEM

   // Number of data to transfer register for "DMA channel x" (DMA_CNDTRx). One
   // CSR for each DMA channel; it contails the number of data to transfer.
   // Set to the appropriate number. And we can read it (after disabling the
   // channel).
   DMA1_Channel3->CNDTR &= ~DMA_CNDTR_NDT;
   DMA1_Channel3->CNDTR |= DMA_data_size;

   // Peripheral address register for "DMA channel x" (DMA_CPARx)
   DMA1_Channel3->CPAR &= ~DMA_CPAR_PA;
   DMA1_Channel3->CPAR |= (uint32_t)DMA_periph_addr;

   // Memory address register for "DMA channel x" (DMA_CMARx)
   DMA1_Channel3->CMAR &= ~DMA_CMAR_MA;
   DMA1_Channel3->CMAR |= (uint32_t)DMA_mem_addr;

   // DMA channel selection register (DMA_CSELR)
   // This is one CSR for all seven channels of one DMA controller.
   // Bits for request mapping: so,
```

```
    DMA1_CSELR->CSELR &= ~DMA_CSELR_C3S;
    DMA1_CSELR->CSELR |= (0b011000000000);

    // Finally: enable the DMA channel.
    DMA1_Channel3->CCR |=  DMA_CCR_EN;
}
```
Figure 2: Code for our DMA setup