ECSE 444: Microprocessors
Lab 4: Timers, Interrupts, and DMA

Abstract

In this lab you will learn how to use timers, interrupts, and direct memory access (DMA) to better control the DAC and produce a wider range of frequencies for output on your speaker.

Deliverables for demonstration

- C implementation of push-button and LED using interrupt
- C implementation of DAC using timer and global interrupt
- C implementation of DAC using timer and DMA
- Audible confirmation of sine signals, with at least three different frequencies, changing on button push

Grading

- 10% Push-button interrupt handler toggles LED
- 40% Timer interrupt handler drives DAC
- 30% Timer triggers DAC DMA
- 20% At least three different DAC output frequencies, changed on button push

Changelog

- 12-Aug-2020 Initial revision.
- 18-Sep-2022 Updated to account for new development kits

Overview

In this lab, we'll revisit the basic mechanics of Lab 2, but improve upon the quality of the output by carefully controlling the rate at which the DAC is written. We'll do this by exploring a bit more a peripheral introduced in Lab 0, the timer. The timer can be used to interrupt normal execution to execute a special function, an interrupt handler. Interrupts tend to be more efficient than *polling* (which is how we've interacted with the button) or using `HAL_Delay(...)` (which is how we've interacted with the DAC), and gives us greater control over timing, which is essential for a wide variety of applications. We'll first use an interrupt to detect when the button has been pressed. We'll then use a timer, and its periodic interrupt, to determine when to write new data to the DAC. Then we'll use the timer, and direct memory access (DMA) to write the DAC; in this last case, sending values to the DAC will be handled almost entirely by hardware, leaving the processor free for other tasks.

Resources

ARM Cortex-M4 Programming Manual
B-L475E-IOT01A User Manual
B-L4S5I-IOT01A User Manual
HAL Driver User Manual
STM32L475VG Datasheet for B-L475E-IOT01A
STM32L4S5VI Datasheet for B-L4S5I-IOT01A
STM32L47xxx Reference Manual

Part 1: Driving DAC with Timer and Global Interrupt

First, we'll set up the LED, push-button, timer, and DAC, very similarly to how we have in the past. The goal is to use the timer to periodically send new values to the DAC, and play a nice, clear sine-wave tone on our speaker.

Configuration

Initialization

As before, start a new project in STM32CubeMX and initialize the configuration, reviewing instructions in earlier labs if necessary.

LED

As in Lab 0, configure the LED. Refer to the B-L475E-IOT01A User Manual or B-L4S5I-IOT01A User Manual for further information. (Remember to check the schematic!) We'll use the LED to test our push-button interrupt handling.

Push button

As in Lab 2, configure the push button. Refer to the B-L475E-IOT01A User Manual or B-L4S5I-IOT01A User Manual for further information. (Remember to check the schematic!)

Additionally, we need to enable external interrupts. In the *Pinout & Configuration* tab, on the left, select *GPIO*. Under *Configuration*, select *NVIC*, and enable *EXTI line[15:10] interrupts*. This means that an interrupt will be generated whenever there is a signal change on the external interrupt lines; our button is wired to one of them.

DAC

For Part 1, set up the DAC the same way as you did in Lab 2; this time, however, we only need one channel. (Remember to check the schematic; you'll be using your speaker again, and I can confirm that it doesn't work when wired to the wrong output.)

Timer

We're going to be driving our speaker again in this lab, and the timer will be helping us to update the DAC output at regular intervals. What's an appropriate interval? CD-quality audio is sampled (and reproduced) at 44.1kHz (it turns out that's not the highest sampling rate in audio). Voice call audio is sampled at lower rates.

Choose a sampling rate (e.g., 44.1 kHz). Given your system clock frequency (e.g., 80 MHz), calculate the counter period (the maximum value of the counter) to achieve this sampling rate. In Lab 0, we used the prescaler; in this lab, this isn't necessary. Finally, under *Parameter Settings*, set the *Trigger Event Selection TRGO* to *Update Event*, and under *NVIC Settings*, enable *TIM2 global interrupt*. Together, these settings ensure that (a) when the timer elapses, execution in `main()` is interrupted; and, (b) the callback function we'll define is executed.

Implementing Push-button Interrupts

An interrupt is a signal (internal or external) that prompts the processor to stop normal execution (e.g., in `main()`), and begin executing an interrupt handler, a special function we write to respond to the event that caused the interrupt signal. We'll get into this more in lectures later in the semester. In Lab 0, we wrote code that *polled* (checked over and over and over again) for changes in the push button signal; in this lab, we'll simply write a function that is executed whenever the push button interrupt occurs.

Section 31.2 of the HAL Driver User Manual details the functions used to interact with GPIO. What we're interested in, in particular, is `HAL_GPIO_EXTI_Callback(...)`. This function is called by the GPIO external interrupt handler, and we can control what it does in `main.c` by simply writing a new definition; our new function is automatically used instead of the weakly defined original.

Write this function in `main.c` (be sure to respect the function prototype defined in the HAL manual) so that it toggles the LED. This function takes as an argument the pin that caused the interrupt; it's good programming practice to verify that the interrupt was caused by the pin we think it was. This isn't essential for our lab, since there are no other external interrupts, but is necessary when a single callback function may need to handle various interrupt sources.

Note: remember to put your code in a USER CODE region so that it doesn't disappear when we go back to MX to modify our configuration. We'll be using the push button again later.

Implementing Timer-driven DAC Output

Now it's time to write a callback function for our timer. Section 72.2 of the HAL Driver User Manual details the functions used to interact with timers. We're particularly interested in two sets of functions: the TIM Base functions, and TIM Callback functions. When we start our timer, we want to start it WITH global interrupts enabled, i.e., in interrupt mode; read the function definitions carefully so that you start your timer in the correct mode. (Yes, you need to call a function to start the timer; don't forget to do so, as this is an otherwise very frustrating problem to debug.)

Just like for the button, `HAL_TIM_PeriodElapsedCallback(...)` is called by the TIM interrupt handler. Write a new definition for it in `main.c`. Again, it's good programming practice to verify that the timer causing the interrupt (an argument passed to your function) is actually the one you want to respond to.

In this function, you'll send a new value to the DAC (see Lab 2 and Section 16.2 of the HAL Driver User Manual). But what value? You can't pass it as an argument, because you don't call this function; it is called in an entirely hardware-controlled process, and the only argument is the timer that caused the interrupt.

What we can do, however, is put the DAC values in a global variable (defined outside of any function, like other private variables in `main.c`). We don't have control over when the timer will elapse and the callback is called; what we need to do, then, is prepare *all* the DAC values we may need, and save them in global variables that can be accessed by the callback function.

In `main(...)`, write code to populate an array with a sine wave (see Lab 2 for how to use the ARM math library). We'll "play" this wave on our speaker (using the same circuit as in Lab 2). To get the best possible results:

- Pick a wave frequency in the 1-2 kHz range (~C6-C7, for the musically inclined). Lower frequencies are harder to hear; I haven't personally tested with higher frequencies.
- Note that the timer frequency is (and must be) higher than that of the signal you want to drive; how do you ensure that your desired frequency is realized?
- Also note that the number of samples you save matters; if you save samples for anything other than $2n\pi$ radians, you will have a discontinuity from the end to the beginning of the array, causing distortion.
- Scale your DAC values so they vary over about 2/3 of the possible dynamic range. The chip will dynamically clamp GPIO outputs to prevent damage, limiting their current to 20 mA. If you attempt to use the full range of DAC output, the signal will look fine under high impedance (e.g., with a voltmeter or pocket oscilloscope), but will clip when connected to the speaker, causing distortion.

Now that you have a global array defining the values to be sent to the DAC, write your implementation of the timer callback so that it sends a new value from this array to the DAC each time it is called.

Part 2: Driving DAC with Timer and DMA

Now we'll change our code to use direct memory access (DMA). DMA uses an on-chip peripheral that can be programmed to perform memory accesses. In this case, DMA will read our array of sine values and write to the DAC for us. This means that we no longer need to execute code in the timer interrupt callback, saving CPU cycles for other tasks (if we had any) or reducing power.

In order to use DMA, we need to reconfigure the DAC. Instead of using our timer to trigger a callback that sets the DAC value, we'll use our timer to trigger the DAC itself. Return to MX. The first thing we need to change, then, is to select the appropriate trigger in *Parameter Settings*. Under *Trigger*, choose the trigger out event corresponding to your timer.

Next, we need to set up DMA. Go to *DMA Settings* and add a DMA request. Choose *Circular* mode; this means the DAC will repeatedly read from the array, starting over from the beginning when the end is reached. *Normal* mode implies that the array would be read and transferred once. Choose the appropriate data width for your software, e.g., I've used 8-bit resolution for my DAC, and a `uint8_t` array for my sine waves, and therefore want DMA to transfer bytes.

Now regenerate your code. Comment out or otherwise disable your timer callback; it is no longer needed. In fact, the global interrupt for your timer isn't necessary at all and can be disabled (though it won't hurt anything). The last thing to do is change how you start the DAC, to start it in DMA mode (Section 16.2 of the HAL Driver User Manual).

Part 3: Putting it All Together

Finally, we'll combine our various pieces of functionality into something more sophisticated. Expand your code so that when the button is pushed, the tone played on the speaker changes. Select at least three different tones; an *arpeggio* (e.g., C6, E6, G6) would suit the purpose well, but anything else is fine, too. Use interrupts, and DMA.