ECSE 444: Microprocessors
Lab 3: Timers, Interrupts, DMA, and DFSDM

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 interrupts
- C implementation of DAC using timer and global interrupt
- C implementation of DAC using timer and DMA
- C implementation of processing and rescaling of DFSDM output for DAC
- C implementation of appropriate interrupts for push-button recording and playback
- Final application

Grading

- 10% Push-button interrupt handler toggles LED
- 20% Timer interrupt handler drives DAC
- 20% Timer triggers DAC DMA
- 30% Push-button trigger of DFSDM mic recording and continuous playback
- 20% Final application

Changelog

- 16-Oct-2023 Slight revision of grading scheme to reduce weight of final application.
- 6-Oct-2023 Initial revision.

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, and then build an application based on more complex input (a mic) and output (multiplexing between different sources).

We'll use a timer to interrupt normal execution to execute a special function, an interrupt handler. Interrupts tend to be more efficient than *polling* or using `HAL_Delay(...)`, and give 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.

We'll then build on this with a new peripheral, an on-board mic connected to an on-chip digital filter for sigma-delta modulator (DFSDM). The DFSDM converts the pulse density modulated output (i.e., digital, with duty cycle in proportion to input frequency) of the microphone into a 24-bit signed integer. We'll then take this value and transform it for output to our DAC.

The final application will record using the mic, and then playback a combination of notes and the recorded sample; state changes will be facilitated by the button, and indicated by the LED.

Resources

HAL Driver User Manual
Interfacing PDM digital microphones Application Note
MP34DT01-M Datasheet

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 and initialize the configuration, reviewing instructions in earlier labs if necessary.

LED

As in Lab 0, configure the LED. We'll use the LED to test our push-button interrupt handling.

Push button

As in Lab 2, configure the push button. 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.

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 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 70.2 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 the configuration of your processor. 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: DFSDM Microphone

Now we'll configure a new input source, a microphone that uses a digital filter for sigma-delta modulator (DFSDM) to produce a digital signal. Your board comes with two (thereby supporting stereo input); we'll use one. For information on the basic mechanics of this process, see Interfacing PDM digital microphones Application Note and MP34DT01-M Datasheet.

Configuration

The DFSDM input channels are wired to different pins in order to interface with different peripherals. Therefore, as we have in the past, we must start by figuring out which pins are wired to the on-board microphones. The microphone interface consists of two signals: one for data, and another for clock. Identify the channel that connects to the correct data signal, and enable it (*PDM/SPI input from chX and internal clock*). Check the CKOUT to enable the output clock signal that will be used to synchronize with the microphone.

Note: historically, the default configuration has not assigned the clock to the correct pin even when you've identified the appropriate channel; check, and if necessary, select the appropriate pin, and choose the clock signal, as this disables the assignment of the signal to any other pin.

Once you've ensured that the DFSDM channel that you've enabled is connected to the correct pins, it's time to configure the DFSDM itself. We need to set up: (a) a filter to convert the PDM input into a digital output, (b) the clock to the microphone, and (c) DMA to transfer from the DFSDM to memory.

Choose a filter; it doesn't matter which one, as any channel may be connected to any filter. Enable the appropriate channel as a "regular channel." Ensure that *Continuous Mode*, and *Software trigger* are selected. Enable *Fast Mode* and *Dma Mode*.

The other filter parameters must be set with the Output Clock settings in mind, because they work together to determine the sampling rate. We're using the system clock to drive the microphone. DFSDM *Output Clock* configuration gives an option to configure a clock divider. Choose an appropriate value, given (a) the operating frequency of your system (*Clock Configuration*), and the requirements of the microphone (MP34DT01-M Datasheet). Under filter configuration, *Fors* specifies the oversampling rate of the filter. (Leave Iosr as 1.) The clock rate to the microphone, divided by the oversampling rate, determines the true sampling rate. Not all oversampling rates are supported by all Sinc filter types; a higher order generally achieves better resolution. (In testing I used Sinc 3, but did not experiment with other options.)

A trade-off emerges: a higher sampling rate (e.g., 44.1kHz) will allow us to produce a clearer output. However, this requires more storage for the same recording interval (e.g., 1 second). A lower sampling rate (e.g., 8kHz) will produce less clear output, but requiring less storage, and enabling us to record for longer and still fit the data in SRAM.

Choose a target sampling frequency (which you will need to reproduce at the DAC for correct playback), and select system clock, output clock divider, and oversampling frequency, accordingly.

Finally, we need to configure the DMA. Note that given our current setup (a single channel and microphone), the DFSDM will produce a 24-bit output, padded in the least significant 8 bits with channel information. (Resulting in a complete 32-bit word.) Also note that "normal" or "circular" mode are reasonable selections, depending on how you ultimately structure your program.

DAC and Timer

Now reconfigure DMA transfer to the DAC. Timer parameters should be chosen such that the output sampling rate matches the input sampling rate of the microphone above.

Implementation

Now implement a program that records microphone input and plays it back on the speaker. Your program must

- begin recording when the button is pressed; the LED should blink when recording;
- begin playback, without user intervention, once the recording buffer is full and has been processed; the LED should stay on when playing;
- continue playback until the button is pressed, at which point it records again, etc.

Before anything is recorded, what should your program do? That's up to you!

You should be able to record for a couple of seconds; you will get a linking error if the arrays you allocate are too large to be stored in SRAM.

The new functions you will need to control the DFSDM can be found in Section 19.2 of HAL Driver User Manual.

Notes

- Be mindful of data types. The DFSDM will produce 32-bits per sample; the most significant 24 is *signed* data, the lower 8 is information about the channel (and can be discarded). The data must be rescaled for the (*unsigned*) output range of the DAC.
- *Normal* DMA will perform a transfer once and stop; starting it again will repeat this process. *Circular* DMA, once started, will run until it is explicitly stopped by function call.
- In normal mode, DFSDM DMA transfer will generate a *CpltCallback* interrupt when it is done with the array. Like the `HAL_GPIO_EXTI_Callback(...)` function that we overwrote earlier, there are corresponding functions for other peripherals. Note that we are using regular conversion, not injected conversion.
- In circular mode, DFSDM DMA transfer will additionally generate a *HalfCpltCallback* interrupt when it is done with half of the array. This makes it possible to process the first half of an array while the second half is being filled. This is necessary in circular mode, because otherwise, e.g., the DFSDM will continue to write, overwriting the array before you've processed it.
- Carefully test your setup to ensure that there's a match between input sampling and output production. The microphone and speaker are both suitable for reproducing speech when configured properly. (Don't forget to start your timer! I did.)
- Note that you need to be quite close to the microphone for it to register at full dynamic range.

Part 4: Putting it all together

Now combine the operation of all of the above in a single application. Your program must

- Record a fixed-length sample on button press;
- On subsequent button press, play, in any order, at least six notes, and the recorded sample (e.g., six notes then the sample; or, three notes, sample, three notes);
- Re-record on subsequent button press, etc;
- Use the LED to indicate the state of the program.

Not all six notes need be at distinct frequencies.

Notes

- Be mindful of data types.
- Be mindful of sampling rates.

Deliverables

Your demo is limited to 10 minutes. Be sure to highlight top-level software structure and program flow. When applicable, it is useful to highlight that your software computes correct partial and final values.

Your demo will be graded by assessing, for each part above, the correctness of the observed behavior, and the correctness of your description of that behavior.

Grading

The breakdown of grading is as follows:

- Push-button interrupt handler toggles LED
  - 10%
- Timer interrupt handler drives DAC
  - 20%
- Timer triggers DAC DMA
  - 20%
- Push-button trigger of DFSDM mic recording and continuous playback
  - 30%
- Final application
  - 20%

Each part of the demo will be graded for (a) clarity, (b) technical content, and (c) correctness:

- 1pt *clarity*: the demo is clear and easy to follow
- 1pt *technical content*: correct terms are used to describe your software
- 3pt *correctness*: given an input, the correct output is clearly demonstrated

Submission

Please submit, on MyCourses, your:

- Source code used to demo (only files you modified, including IOC file).