ECE271: Microcomputer Architecture and Applications — University of Maine

Lab #7: System Timer and Interrupts

Week of 21 March 2022

Goals

1. Understand the clock tree of the STM32L4.
2. Understand functionality of the system timer.
3. Understand the basics of interrupt handlers.

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

The end goal of this lab is to have the red LED toggle once per second, with a delay function driven by the SysTick timer interrupt.

Part A – Initial Setup

1. Disconnect the stepper motor, as it can get hot if sitting connected and un-driven. Also, the GPIOB2 pin for the RED LED is used by the motor and can cause an over-current condition if we output to it for long periods without actually stepping the rest of the pins.

2. Make a copy of one of your existing C labs to use as a base. Lab5 is probably good. Strip out everything in `main()`.

3. 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 8MHz Multi-Speed Internal (MSI) Clock

1. As described in the pre-lab, your board by default starts up using a 4MHz MSI clock. We want to change this to 8MHz.

2. Create a function called `System_Clock_Init()` and call it at the start of your `main()` function. (Note, on Linux this function already exists; replace the contents with the code as described below).

3. The registers being set here are described in Section 6.4 of the **"STM32L4x5 and STM32L4x6 advanced Arm-based 32-bit MCUs"** manual which you can find on the course website.

4. Before we can change the clock frequency, the manual says that either the MSI clock must be off (the `MSION` field in `RCC->CR`) or else MSI clock can be on but the `MSIRDY` field must be 1.

   The easiest thing to do here is to disable `MSION` at the beginning of your function.

5. Next, set the `MSIRANGE` field in the `RCC->CR` register to the value you looked up in the pre-lab.

6. Next, we have to tell the MSI clock to use this new value. You do this by setting the `MSIRGSEL` bit in `RCC->CR` to one.

7. Now re-enable the MSI clock by setting the `MSION` bit back to 1.

8. Finally, wait for the MSI clock to be ready by waiting for the `MSIRDY` bit in `RCC->CR` to be 1. The most straightforward way to do this is to poll, so read it in a tight loop until it changes.

9. In theory we should also set the the MSI clock to be the system clock via the `RCC->CFGR` register, but this is the reset default so we will assume it will be properly set for us.

Part B – SysTick Initialization

1. Now make a `SysTick_Initialize(int ticks)` function

2. Section 11.7 of the textbook has some info on how to do this.

3. Note that SysTick counter is an ARM Cortex-M interface, not specific to the STM32L4 SoC. So documentation on it lives in a different document, the "**Cortex-M4 Devices Generic User Guide**" in Section 4.4.

4. First disable the SysTick counter. One quick way is to write the `SysTick->CTRL` value to 0 which will disable all of the bits.

5. Next set the value in the reload register, `SysTick->LOAD`. The textbook suggests setting this to "ticks-1"; if you do that make sure when you call this function you take into account the "-1".

6. Set the interrupt priority, as done in the sample code in the textbook. This involves setting up the `SCB->SHP` register. The textbook has you use the `NVIC_SetPrioity()` function which is smart enough to set `SCB->SHP` for negative (system) interrupts.

7. Set `SysTick->VAL` to zero, which resets the count.

8. Set `SysTick->CTRL` to use the internal processor clock rather than the external clock.

9. Set `SysTick->CTRL` to enable interrupts (the `TICKINT` field).

10. Set `SysTick->CTRL` to enable the timer via the `ENABLE` field.

11. Finally, now that you have the Initialize function, call it from `main()` with the value you calculated in the pre-lab (making sure to remember that the routine you wrote is subtracting 1 from the ticks value).

Part C – Write the Interrupt Handler

1. Write the interrupt handler. On Cortex-M processors this looks just like a normal function. Call it `void SysTick_Handler(void)`

2. On Keil this function you write will automatically override the version declared "WEAK" in the initialization routine. If you are doing things on Linux instead find the existing `SystTick_Handler()` and replace the code.

3. The handler is simple. Declare a global volatile variable called `TimeDelay`. The handler should check if this variable is greater than 0. If so, decrement it.

Part D – Write the Delay Routine

1. Give this function the name `void Delay(uint32_t nTime)`

2. The first thing it does is set the value of the global `TimeDelay` from the interrupt handler to the value passed in as `nTime`.

3. Next it should loop forever waiting for `TimeDelay` to hit zero. Each time the interrupt handler is called the value will decrement, so this `Delay` routine will busy wait until the interrupt handler has been called `nTime` times.

Part E – Setup the LED Code

1. Set up the Red LED on GPIOB2.

2. Have an infinite loop in `main()` that has the LED on for 1s (`Delay(1000);`) then off for 1s (`Delay(1000);`) and repeat.

Part F – Activate Global Interrupts

1. Even though we have enabled timer interrupts and set up the timer, our handler might not be called yet. This is because it is possible to globally disable (or "mask") processor-wide receiving of interrupts. To ensure we get interrupts we need to unmask/enable interrupt handling at the CPU level.

2. The way to do this is the `CPSIE I` assembly language instruction. Can we do this in C? We'll have to use inline assembly.

   (a) On Keil, you can do this:

   ```
   __asm("CPSIE i");
   ```

   (b) On Linux it will look like:

   ```
   asm volatile ( "cpsie i" );
   ```

3. Put this code in `main()` after the timer init.

4. Compile your code, load it on the board, and you should having the LED toggling once per second.

Part G – Something Cool

Do something cool! You can come up with something on your own, but here is a list of ideas you can use. (Note, flashing the green LED is not enough).

1. Use the delay to display a clock on the LCD.
2. Use the delay to send a Morse code message on the LED.
3. Use the delay function to drive the stepper motor.
4. Use one of the other system clocks (HSE or HSI) to drive the system timer.
5. Program the time delay on the LED via the keypad.

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) Display the LED toggling once a second.
   (b) Use an oscilloscope to measure the exact period of the LED signal on GPIOB2.
      i. How accurate is the timing? (How close is it to 1s?)
      ii. What is the percent error?
      iii. Take a picture of the oscilloscope screen and commit it into git with your README.

Post-Lab

- Place your answers to the question in a file Readme.md
- Submit with your code via the gitlab server.
- Questions:
  1. What is the address of the interrupt vector that points to the SysTick handler?
  2. Suppose a 16MHz clock is used to drive the timer. What is the maximum interval between two interrupts that we could obtain?