**ECE271: Microcomputer Architecture and Applications — University of Maine**

# Lab #9: Input Capture and Distance Sensor

Week of 8 April 2019

## Goals

1. Understand input capture function of a timer.
2. Handle different events in the interrupt service routine.
3. Handle timer counter underflow and overflow.
4. Use a timer to measure the timestamp of a signal edge external to the microprocessor.

## 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 first part of this lab is using input capture with the TIM4 timer to measure a 1Hz external signal. The second part is using input capture to measure distance using an ultrasonic sensor.

### Part A – Initial Setup

1. This lab will probably be easiest if you use Lab#8 as a starting point, as you can adjust the existing timer code. (It should still be possible to do the lab even if you did not do Lab#8 yet).

2. 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 the board to use the 16MHz HSI clock

1. For this Lab we will use the 16MHz HSI clock on the board.

2. Modify the `System_Clock_Init();` function that gets called at the beginning of `main()`

   (a) Enable the `RCC_CR_HSION` field in the `RCC->CR` register.

   (b) Wait for the `RCC_CR_HSIRDY` field in `RCC->CR` to go high.

   (c) Select HSI as the system clock source by setting the `SW` field in the `RCC->CFGR` register to 01. Do this by clearing out the `RCC_CFGR_SW` field and making sure `RCC_CFGR_SW_HSI` is set.

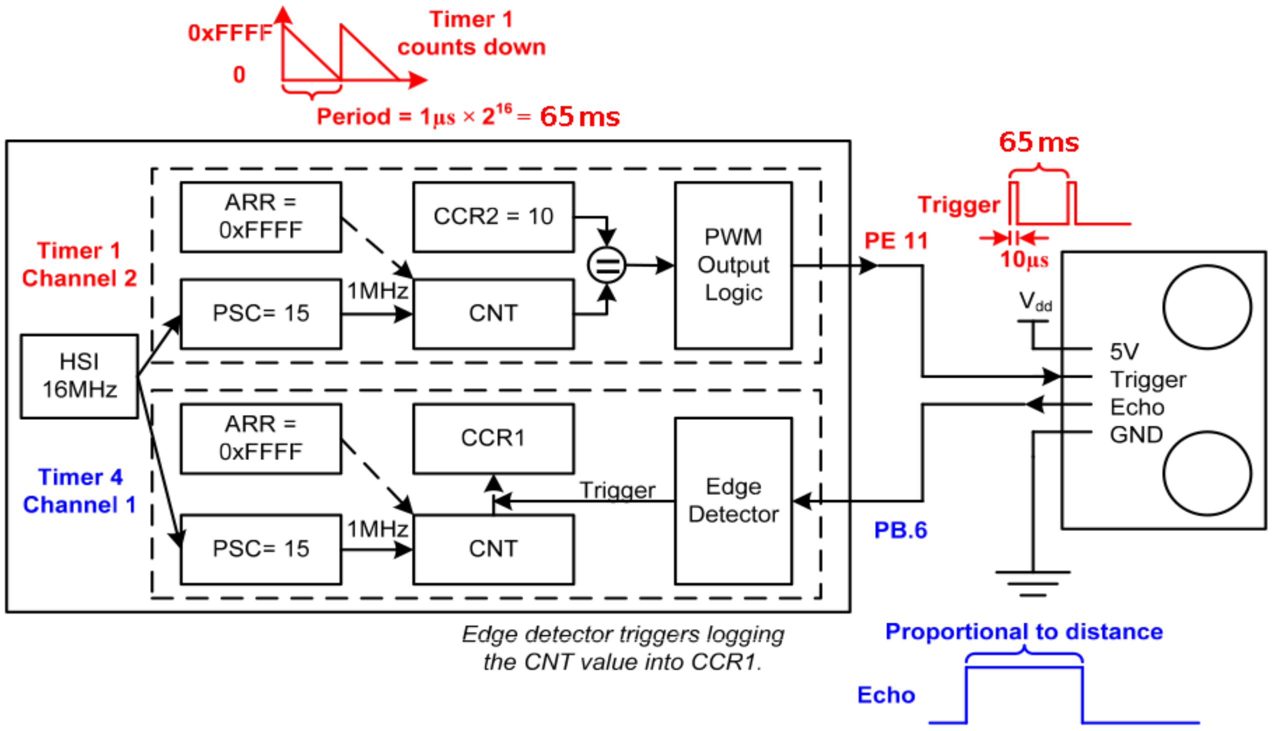

Figure 1: Timer setup for lab.

## Part C – Set up GPIO PB6 to use TIM4_CH1

1. Figure 15-23 in the textbook has the flowchart for the next few sections. Example 15-7 has example code.

2. First configure the GPIO PB6 pin

   (a) Make sure GPIO port B clock is enabled in `RCC_AHBENR_GPIOBEN`.

   (b) Set GPIOB pin 6 to be alternate function, function type `TIM4_CH1` (see textbook Appendix I to see which value this is).

## Part D – Configure TIM4_CH1 for Input Capture

1. Enable the clock of timer 4 (`RCC_APB1ENR1_TIM4EN`)

2. Set the prescaler register `TIM4->PSC` so that it divides the 16MHz clock down to 1MHz

3. Set the auto-reload register `TIM4->ARR` to the maximum 16-bit value

4. Set the direction of channel 1 as input, and set input to 1, so value 01 in the `CC1S` fields in `TIM4->CCMR1`

5. Set the input filter duration to 0 (bit `TIM_CCMR1_IC1F`) in `TIM4->CCMR1`

6. Set the capture to be on both rising and falling (so value 11) in `TIM4->CCER`. You do this by setting bits `TIM_CCER_CC1P` and `TIM_CCER_CC1NP`

7. Clear the input prescaler so that we capture each transition: `TIM4->CCMR1` bit `TIM_CCMR1_IC1PSC`

8. Enable capture for Channel 1: `TIM4->CCER` bit `TIM_CCER_CC1E`

9. Enable capture interrupt generation for TIM4 Channel 1: `TIM4->DIER` bit `TIM_DIER_CC1IE`

10. (This one is not in the textbook example) we will want to enable the overflow interrupt `TIM4->DIER` bit `TIM_DIER_UIE`

11. Enable timer4: `TIM4->CR1` bit `TIM_CR1_CEN`

12. Set the priority of the `TIM4_IRQn` interrupt to 0 (highest priority) using `NVIC_SetPriority()`

13. Enable the Timer4 interrupt in the interrupt controller using `NVIC_EnableIRQ()`

## Part E – Create a Timer Interrupt Handler for TIM4

1. See Example 15-6 in the textbook

2. Add (or on Linux, edit) the function `TIM4_IRQHandler()`

3. The code described above will generate interrupts whenever a rising/falling transition happens.

4. We set the `UIE` bit in the `TIM4->DIER` register, which also calls the interrupt handler when an overflow/underflow happens (in our case, after the count hits 65535us).

5. To find out what kind of event triggered the overflow, you can check the `TIM4->SR` register.

   (a) If `UIF` is set, then it was an overflow/underflow

   (b) If `CC1F` is set, it was an input capture event.

6. Before exiting the handler, you will need to acknowledge (ACK) the interrupt. Otherwise when you exit the interrupt will immediately trigger again. The CC1F bit is automatically cleared if you read `TIM_CCR1` but you must manually clear the `UIF` flag.

7. For this lab you will need to track the number of overflows that happen to properly measure the capture time.

8. The code in the textbook keeps track of the signal polarity, and calculates `pulse_width` by subtracting the time the signal went low from the time the signal went high.

9. The above will possibly be wrong if an overflow happens; to account for that keep track of the overflows in the overflow handler. Reset this to zero at the high transition, let it count on each overflow, then when you calculate `pulse_width` add in the overflow count times the time between overflows (in our case 0xffff micro seconds).

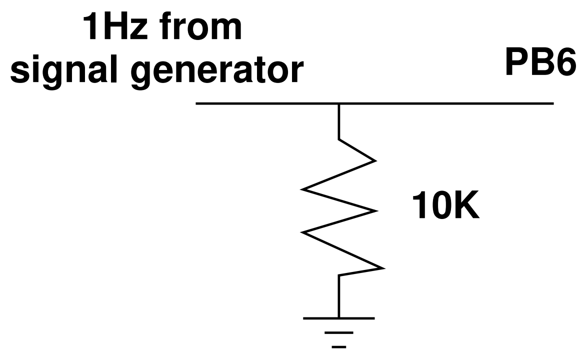

Figure 2: Current limiting resistor.

## Part F – Measure a 1Hz Clock Signal

1. Use the function generator in the lab to generate a 1Hz signal, then use input capture to capture it.
2. Hook the output of the generator to pin PB6. To protect the board you can use a 10K resistor to limit the current as shown in Figure 2. (Note, there has been some trouble with the resistor. Tou can try connecting the function generator directly to PB6 and your borad will probably be fine).
3. Make sure the signal has a max voltage of 3V
4. View the period of the signal. We can do this by looking at the "pulse_width" variable in memory. The pulse width should be the length in us of the high pulse, assuming a 1Hz signal with 50% duty cycle it should be roughly 500000. For the checkoff you will use the debugger to show the TA what value you measure.
5. Showing the value on Keil:
   (a) Enter the debugger
   (b) Add a "watch" for the pulse_width variable
   (c) You can do this by the "View" than "Watch Window"
   (d) The watch is added at the bottom of the screen. Double click the expression name and put "pulse_width" in there.
   (e) Now, as the code runs, it should update the value on the fly.
6. Showing the value using Linux:
   (a) Flash your program and leave openocd running (if you flash with "make flash" it will do this for you)
   (b) From another window type the following commands:
   ```
   gdb-multiarch ./lab9.elf
   target remote localhost:3333
   monitor reset halt continue
   ```
   (c) Now your code should be running. In the debugger window press control-C to temporarily halt things. Then you can print the current value by typing:
   ```
   print pulse_width
   ```
   and it should show the current value of the variable.

## Part G – Ultrasonic Distance Sensor

1. Hook up the ultrasonic distance sensor
   (a) Connect `Vcc` on the sensor to `EXT_5V`
   (b) Connect `GND` to a ground connector
   (c) Connect `Trigger` to PE11
   (d) Connect `Echo` to PB6
2. The sensor runs at 5V. It can be triggered by a 3.3V pulse, and although it's probably not the best for the STM board, it seems as though the input is 5V tolerant and can handle that as input.
3. Generate the 10us pulse on trigger on PE11 (you did these calculations in the prelab). This will trigger an ultrasonic burst of 40kHz audio.
   (a) Put the values from the prelab in to enable TIM1.
   (b) Be sure you are setting up Channel 2 for all the settings! The last lab we used Channel 1 so things are slightly different.
   (c) After the init be sure to enable the counter by setting `TIM_CR1_CEN` in `TIM1->CR1`.
   (d) If you want to verify the signal is working, hook it to the oscilloscope and make sure you are getting a 10us pulse every 65ms on PE11.
4. Conduct timer capture on PB6 and get the return signal. The result will be a square wave proportional to the distance to the nearest object.
   (a) The return on the ECHO pin will range from 150us to 25ms (38ms if nothing in range).
   (b) To calculate the distance:
   $distance(inches) = \frac{time(\mu s)}{148}$
   $distance(cm) = \frac{time(\mu s)}{58}$
   (c) Store the result in an integer variable.
   (d) For the checkoff you will show the TA the result in the debugger. Directions for this should be similar to those for measuring the 1Hz signal

## Part H – Something Cool

Do something cool! You can come up with something on your own, but here is a list of ideas you can use.

1. Print the distance to the LCD
2. Have the LEDs turn on/off at certain distance cutoffs (for example: put your hand closer than 3feet and the red LED comes on, closer than 1 foot and the green LED comes on)

## 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) Show your 1Hz result.
    (b) Demonstrate the ultrasonic sensor.

## 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 your accuracy when measuring the 1Hz square wave?
    2. Place an object at a 1 foot distance. How accurate is the result?