ECE 153B – Winter 2024

Sensor and Peripheral Interface Design

Lab 3 – SysTick, PWM, Servo Motor, and Ultrasonic Sensor

Deadline: **Feb 8, 2024, 10:00 PM**

Objectives

1. Understand the basic concept of the system timer (SysTick)
   - Use SysTick to create time delay functions
2. Understand the concept of Pulse Width Modulation (PWM)
   - Learn how to configure and start timers
   - Use PWM to control LED brightness
   - Use PWM to control Servo Motor
3. Understand the basic concept of the timer input capture function
   - Handle different events in the interrupt service routine
   - Handle timer counter overflow and underflow
   - Use a timer to measure the timestamp of a signal edge external to the microprocessor

Grading

Part	Weight
Part A	20 %
Part B	30 %
Part C	20 %
Checkoff Questions	30 %

You must submit your code and answers to the questions to the submission link on Gradescope by the specified deadline. In the week following the submission on Gradescope, you will demo your lab to the TA.

Please note that we take the Honor Code very seriously, do not copy the code from others.

Contents

1 Necessary Supplies

- STM32L4 Nucleo Board
- Type A Male to Mini B USB Cable
- HC-SR04 Ultrasonic Distance Sensor
- Breadboard
- Jumper Wires
- SG90 Micro Servo Motor

2 Lab Overview

A. Use SysTick to generate an interrupt every 1 ms. Implement a function that toggles the green LED every second by implementing function `delay()`.

B. Configure the timer to generate PWM outputs. Create a periodic dimming light by modifying the duty cycle of the generated PWM output.

C. Understand how to use timers for input capture. Interface with the HC-SR04 Ultrasonic Distance Sensor and use input capture to get measurements from the sensor.

3 Part A – SysTick

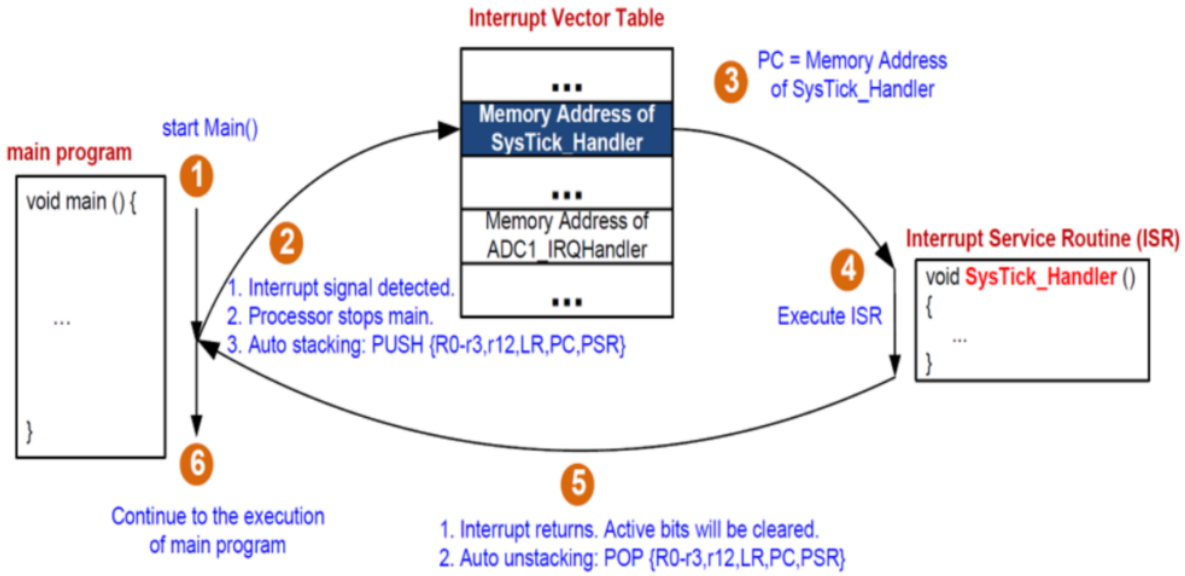


Figure 1

In this part of the lab, you will set up the system timer (SysTick) to generate interrupts (see Figure 1), with a period of 1 ms. The `CLKSOURCE` bit of the `SysTick_CTRL` register indicates the clock source for SysTick.

- If `CLKSOURCE` is `0`, the external clock is used. The frequency of SysTick is the frequency of the AHB clock divided by 8.
- If `CLKSOURCE` is `1`, the processor clock is used.

For this lab, *you are required to configure the AHB clock frequency to be 8 MHz by using MSI.* The following steps outline how to change the clock frequency.

1. Select the MSI clock range by configuring the `MSIRANGE` bits of `RCC_CR` (RCC Clock Control Register). Check "STM32L4 Reference Manual" and `stm32l476xx.h` to find out what values the `MSIRANGE` bits should be.

   `MSIRANGE` = _____

   Note that

   - `MSIRANGE` can be modified when MSI is OFF (i.e. `MSION` = `0`) or when MSI is ready (i.e. `MSIRDY` = `1`). `MSIRANGE` must *not* be modified when MSI is ON and not ready (i.e. `MSION` = `1` and `MSIRDY` = `0`).

- The MSIRGSEL bit in RCC_CR selects which MSIRANGE bits will be used.
  - If MSIRGSEL = 0, the MSIRANGE bits in RCC_CSR are used to select the MSI clock range. This is the default. MSIRANGE can be written only when MSIRGSEL = 1. However, changing the MSIRANGE in RCC_CSR does not change the current MSI frequency. The clock will be changed after a standby.
  - If MSIRGSEL = 1, the MSIRANGE bits in RCC_CR are used.

2. Set MSION (MSI Clock Enable) and wait for MSIRDY (MSI Clock Ready Flag) in RCC_CR.
   - We use CLKSOURCE = 0, what is the value of the SysTick Reload Value Register that will generate an interrupt every 1 ms?

     SysTick_LOAD = _____

The frequency of internal clocks (RC oscillators) may vary from one chip to another due to manufacturing process variations. In addition, the operating temperature has an impact on the accuracy of the RC oscillators. At 25 °C, the HSI and MSI oscillators typically have an accuracy of ±1.5 %, however the accuracy decreases in extreme temperatures in either direction (around −40 °C or 105 °C).

In SysTimer.c, you will see a variable defined with the volatile keyword. This keyword should be used in the variable definition if the variable's value can be changed unexpectedly (e.g. when the value is modified within an interrupt service routine). When the volatile keyword is used, the compiler takes this fact into consideration when generating the executable binary and performing optimizations to the code.

Write a simple program that toggles the green LED every second. First, complete the implementation of function delay(). You can then perform an action every second by calling delay(1000).

4 Part B – Pulse Width Modulation

In this part of the lab, you will control the brightness of the green LED and angle of a server motor using PWM. To do this, you will use the general purpose timers on the STM32 Nucleo board. For this part of the lab, we will just use the default 4 MHz system clock.

4.1 Introduction - LED

From the previous lab, we know that **PA5** (GPIO Port A Pin 5) is connected to the green LED. However, each GPIO pin can also be configured to perform an alternative function by configuring the GPIO to be used as an alternative function and specifying the desired alternative function number. For **PA5**, the available alternative functions are `TIM2_CH1`, `TIM2_ETR`, `TIM8_CH1N`, `SPI1_SCK`, `LPTIM2_ETR` and `EVENTOUT`. The alternative function that we are interested in is `TIM2_CH1` (the timer). Figure 2 shows a diagram of the control circuit for channel 1 of timer x and Table **??** shows how the control bits affect the output of the timer control circuit.

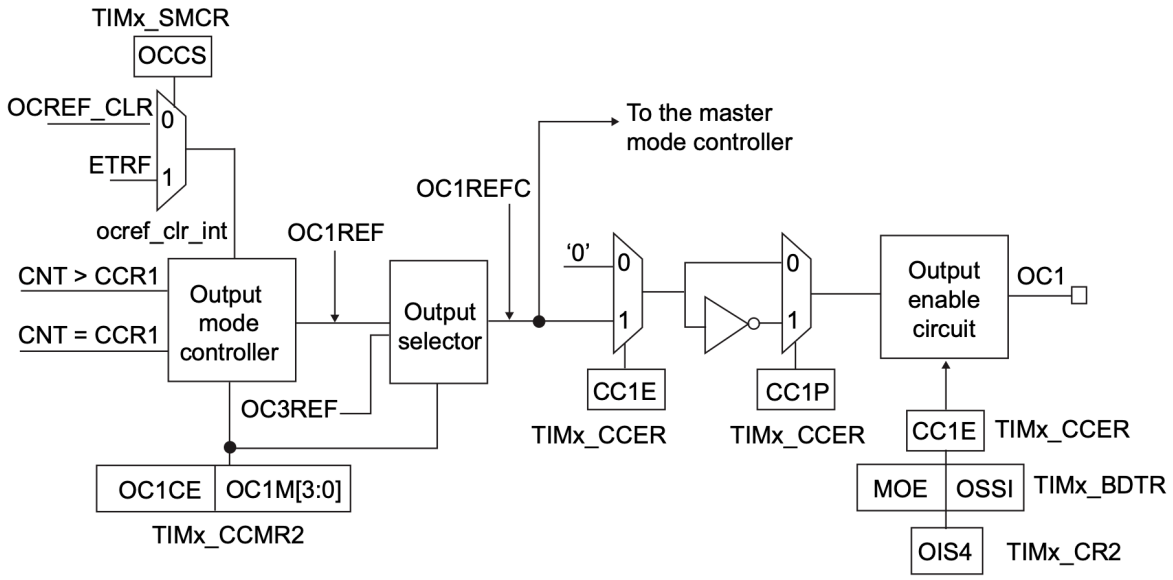


Figure 2: Timer x Control for Channel 1

We can change the brightness of an LED by rapidly switching the LED on/off with pulses of varying widths. Depending on the pulse width (i.e. the amount of time in which the signal is high during one clock cycle) of the signal, this will create the effect of the LED being "dimmer" (lower duty cycle) or "brighter" (higher duty cycle). The timers on the STM32L4 Nucleo board can be configured to produce PWM outputs so we can modify the brightness of the LED.

4.2 Lab Exercise - LED

Use the following steps to help you determine the masks and values for the registers that need to be set for configuring PWM using the alternative function of **PA5**. Note that even when using the alternative function of the GPIO pin, we will be getting an output from the alternative function, allowing the green LED to update based on the output of the alternative function.

1. **Enable the Clock of GPIO Port A in RCC**

2. **Enable the Clock of Timer 2 in RCC_APB1ENR1**

3. **Configure PA5 to Alternative Function Mode**
   GPIO Mode: Input (`00`), Output (`01`), Alternative Function (`10`), Analog (`11` – default)

4. **Configure PA5 to Very High Output Speed**
   GPIO Output Speed: Low (`00`), Medium (`01`), High (`10`), Very High (`11`)

Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`OSPEEDR`	`OSPEED15[1:0]`		`OSPEED14[1:0]`		`OSPEED13[1:0]`		`OSPEED12[1:0]`		`OSPEED11[1:0]`		`OSPEED10[1:0]`		`OSPEED9[1:0]`		`OSPEED8[1:0]`		`OSPEED7[1:0]`		`OSPEED6[1:0]`		`OSPEED5[1:0]`		`OSPEED4[1:0]`		`OSPEED3[1:0]`		`OSPEED2[1:0]`		`OSPEED1[1:0]`		`OSPEED0[1:0]`	
Value																																

5. **Configure PA5 to No Pull-Up, No Pull-Down**
   GPIO PUPD: No Pull-Up, Pull-Down (`00`), Pull-Up (`01`), Pull-Down (`10`), Reserved (`11`)

6. **Configure and Select the Alternative Function for PA5**
   See the *STM32L476 Pins + Functions* document to find the alternative functions that are available for **PA5**. Then, find the alternative function number for Timer 2 Channel 1 and enter the binary representation of the alternative function into the correct register.

Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`AFR[0]`	`AFRL7[3:0]`				`AFRL6[3:0]`				`AFRL5[3:0]`				`AFRL4[3:0]`				`AFRL3[3:0]`				`AFRL2[3:0]`				`AFRL1[3:0]`				`AFRL0[3:0]`			
Value																																
`AFR[1]`	`AFRH15[3:0]`				`AFRH14[3:0]`				`AFRH13[3:0]`				`AFRH12[3:0]`				`AFRH11[3:0]`				`AFRH10[3:0]`				`AFRH9[3:0]`				`AFRH8[3:0]`			
Value																																

7. **Configure the PWM output for Timer 2 Channel 1**
   Note: The alternative function that we are using is `CH1`, not `CH1N`. So, make sure to enable the correct channel in the `TIM2_CCER` register.

   The timer is going to be used in an upcounting configuration. The timer will start from 0 and will continue to increment its counter every clock cycle until it sees the ARR, which will reset the count to 0. Then, the ARR determines the frequency of the PWM signal. The CCR determines when the output signal will change polarity. Then, the CCR determines the duty cycle of the PWM signal. Refer to Section 27.2 in *STM32L4x6 Reference Manual* for a more detailed description of the purpose of these two values and for example timing diagrams.

   We need to set the frequency of the clock that the timer is going to use for counting ticks by scaling the system clock. With knowledge of the tick period, we can set values for CCR and ARR that will effectively make the timer count the time in which the output signal is high. In addition, we must ensure that the correct output channel is enabled. The following is a list of things you have to configure. We have provided a register map for `TIM2`, but you should also refer to Section 27.4 in *STM32L4x6 Reference Manual* for more detailed information about each register and the bits in each register.

   (a) [`TIM2_CR1`] Set the direction such that the timer counts up.

   (b) Set the prescaler value.

   (c) Set the auto-reload value.

   (d) [`TIM2_CCMR1`] Configure the channel to be used in output compare mode. We will use output compare 1.

   - Clear the output compare mode bits.
   - Set the output compare mode bits to PWM mode 1.
   - Enable output preload.

   (e) [`TIM2_CCER`] Set the output polarity for compare 1 to active high.

   (f) [`TIM2_CCER`] Enable the channel 1 output.

   (g) [`TIM2_CCRx`] Set the capture/compare value. For now, set it such that the duty cycle of the PWM output is 50%. You will be modifying this value later to change the brightness of the green LED.

   (h) [`TIM2_CR1`] Enable the counter.

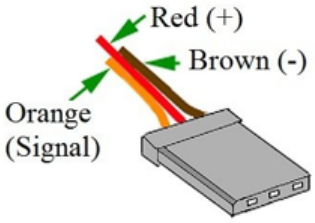
Now that everything is set up, you will now be able to play around with different PWM outputs. Write code that will make the green LED change its brightness periodically. Demonstrate the periodic dimming to get checked off for this part of the lab. The LED should change its brightness periodically and smoothly, going from minimum to maximum and back to minimum. You might need an extra variable to keep track of current direction (up or down) for changing the CCR.

4.3 Introduction - Servo Motor

In this part of the lab, you will control the position of the SG90 micro servo motor through various PWM duty cycles.

The key difference between a servo motor and a stepper motor is that the servo motor is controlled by a closed-loop system that uses position feedback to perform self-adjusting. A servo motor has an internal position sensor, which continuously provides position feedback. On the contrary, a stepper motor is typically an open-loop system with no built-in position sensor.

A servo motor is preferred in applications that require higher speed. A stepper motor is often used in applications that need higher holding torque.



In this lab, we will use the SG90 servo motor. It can rotate approximately 180 degrees, with 90 degrees in each direction. The operating speed is 0.12sec/60 degrees (4.8V, no load). The motor has three wires, red wire for +5V, brown for ground, and orange for the PWM control signal.

The position is controlled by the pulse width, as shown in Figure 3 below.

- A pulse of 1.5 ms makes the motor return to the middle (0 degree)
- A pulse of 1 ms makes it rotate all the way to the left (-90 degrees)
- A pulse of 2 ms makes it rotate all the way to the right (90 degree)

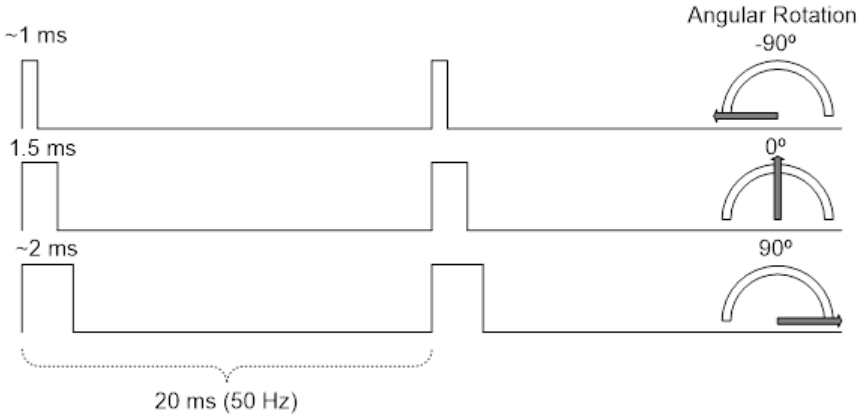


Figure 3: Position of servo motor for different duty cycles

A timer can generate an output or capture an input. The above diagram (Figure **??**) shows the control of generating an output. Read the STM32L4 Reference Manual to find the details

of these timer registers.

The stepper motor is connected to **PA 0**. The following table shows the Alternate functions available for this pin. In this lab, `TIM5_CH1` is selected for **PA 0**.

Peripheral	Pin	Available Alternate Functions
Servo Motor	PA 0	TIM2_CH1 / **TIM5_CH1** / TIM8_ETR / USART2_CTS / UART4_TX / SAI1_EXTCLK / TIM2_ETR / EVENTOUT

Table 1: Alternate function options for PA 0

Refer to Appendix B to control the outputs OCx and OCxN.

4.4 Lab Exercise - Servo Motor

In this lab we will set up a timer and use it to generate Pulse-width modulated (PWM) signals for the servo motor, similar to the Green LED in previous lab. The servo motor is connected to **PA 0**, which can function as Channel 1 of Timer 5 (`TIM5_CH1`). The requirements for this part of the lab are as follows.

1. Initialize TIM5 so the ARR period happens at 50Hz (20ms)
2. Calculate the values you need in the CCR1 register to end up with a duty cycle of 1ms, 1.5ms, and 2ms.
3. Modify your code from previous part in Part B, make your servo motor rotate along with the LED light: when light is fully off, turn all the way clockwise. When light is fully on, turn all the way counterclockwise. It should have a smooth rotation, as in previous part.

Note: You may find that for your servo 1ms / 2ms might not be the exact right values to rotate 90 degrees. Adjust the values until you get a good 90-degree rotation.

Use the following steps to configure the alternate function on PA 0

1. Configure `RCC_AHB2ENR` to enable the clock of GPIO Port A
2. Configure PA 0 (Servo Motor) as Alternate Function Mode
3. Configure and Select the Alternative Function for PA0
4. Configure PA 0 to No Pull-Up, No Pull-Down
5. Configure the PWM output for Channel 1 of Timer 5

Setting up the TIM5 timer should be the same as TIM2 timer. This time, you need to configure ARR to generate 50Hz period

5 Part C – Timer Input Capture and Ultrasonic Sensor

5.1 Introduction

The free-run counter (`CNT`) of timers used in this lab are limited to 16 bits. There are two special events that can occur while counting:

- While up-counting, `CNT` restarts from `0` after reaching `0xFFFF`. This event is called *counter overflow*.
- While down-counting, `CNT` restarts from `0xFFFF` after reaching `0`. This event is called *counter underflow*.

When an overflow or underflow occurs, the timer can generate a time interrupt if the **Update Interrupt Enable** (`UIE`) bit is set in the timer **DMA/Interrupt Enable Register** (`TIM_DIER`).

In the interrupt service routine of the corresponding timer, you can check the timer **Status Register** (`TIM_SR`) to find out what events have generated the timer interrupt.

- If a counter overflow or underflow occurs, the **Update Interrupt Flag** (`UIF`) is set in `TIM_SR`. This flag is set by the hardware.
- If a channel is configured to input mode, a valid transition of an external signal can trigger the timer interrupt. Take channel 1 as an example. Say channel 1 is configured as input (capture) and the **Channel 1 Interrupt Flag** (`CC1IF`) in `TIM_SR` is set. Then, when the capture of channel 1 has been triggered, the counter value is copied to the `CCR1` register. `CCxIF` is set by the hardware.

The timer interrupt service routine must clear these flags in `TIM_SR` to prevent it from being immediately called again by the processor. `CCxIF` is automatically cleared when the `TIM_CCRx` register is read from. However, `UIF` must be explicitly cleared.

5.2 Lab Exercise

In this part of the lab, you will interface with the ultrasonic distance sensor and store your measurements in the Stack or monitor them in the watch window. The following are the main points that you need to know when interfacing with the ultrasonic distance sensor (see the *HCSR04 Ultrasonic Sensor* datasheet for the full documentation). Figure 4 shows the connections between the STM32L4 Nucleo board and the sensor.

- The sensor is powered with a 5 V source. Connect the `Vcc` pin on the ultrasonic sensor to the `5V` pin on the STM32L4 Nucleo board and connect the `GND` pins together.
- While the sensor runs at 5 V, it can be triggered with a 3.3 V pulse. The sensor outputs a 5 V signal, but many of the inputs on the STM32L4 Nucleo board are 5 V tolerant and can handle a 5 V input.
- As described in the documentation, to activate the sensor, a high pulse of at least 10 µs must be sent to the `Trig` input. An ultrasonic 40 kHz burst will be emitted, and the sensor will output (to the `Echo` pin) a square wave with a pulse width proportional to the distance to the nearest object.
- The resulting square wave can have a pulse width ranging from 150 µs to 25 ms (the pulse width will be 38 ms if there is no object within range). To convert this value to a distance in inches or centimeters, you can use the following formulas.

$$d = \frac{\text{pulse width (µs)}}{58} \text{ cm} \qquad d = \frac{\text{pulse width (µs)}}{148} \text{ in}$$

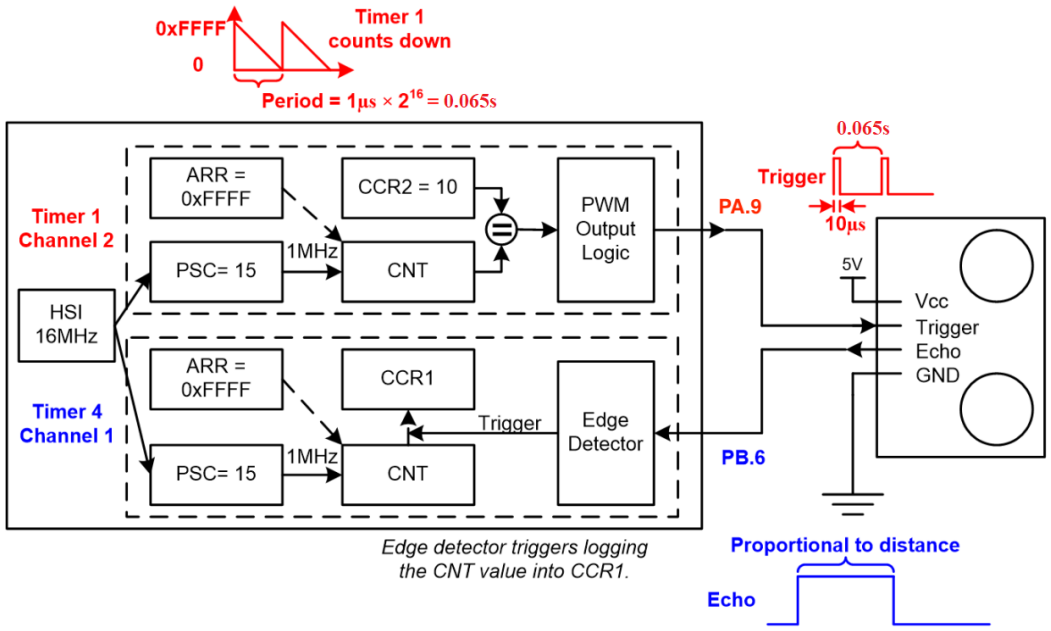


Figure 4: Connection and Configuration for Sensor Interfacing

For this part of the lab, use **PA9** to create a PWM signal to activate the distance sensor and use **PB6** to capture the resulting square wave.

	Pin	**Alternative Function**
Trigger	**PA9**	`TIM1_CH2`
Echo	**PB6**	`TIM4_CH1`

For this part of the lab, the clock frequency is 16 MHz. Set the prescaler value to 15. Then

$$f_{\text{Timer Clock}} = \frac{f_{\text{HSI}}}{1 + \texttt{PSC}} = \frac{16 \text{ MHz}}{1 + 15} = 1 \text{ MHz}$$

Use the following steps to create a PWM signal that will trigger the ultrasonic sensor. Refer to Sections 8.4 (RCC), 9.4 (GPIO), and 26.4 (Timer 1) in *STM32L4x6 Reference Manual* for details about registers and their bits.

1. Set up **PA9**.
   (a) Enable the clock for GPIO Port A.
   (b) Configure **PA9** to be used as alternative function `TIM1_CH2`.
   (c) Set **PA9** to no pull-up, no pull-down.
   (d) Set the output type of **PA9** to push-pull.
   (e) Set **PA9** to very high output speed.
2. Enable Timer 1 in `RCC_APB2ENR`.
3. Set the prescaler to 15.
4. Enable auto reload preload in the control register and set the auto reload value to its maximum value.
5. Set the CCR value that will trigger the sensor. (*Hint*: What is the timer clock frequency and what kind of signal activates the sensor?)

6. In the capture/compare mode register, set the output compare mode such that the timer operates in PWM Mode 1 and enable output compare preload.
7. Enable the output in the capture/compare enable register.
8. In the break and dead-time register, set the bits (at the same time) for main output enable, off-state selection for run mode, and off-state selection for idle mode.
9. Enable update generation in the event generation register.
10. Enable update interrupt in the DMA/Interrupt enable register and clear the update interrupt flag in the status register.
11. Set the direction of the counter and enable the counter in the control register.

Next, we need to set up input capture. Input capture is used to find the time span between two transitions in a signal (rising, falling, or both). When a transition is detected, the timer hardware generates an interrupt and copies the free run counter value into the capture/compare register. By keeping track of the CCR values from two consecutive interrupts, we can compute the amount of time that has passed between the two transitions. For example, let's say that an interrupt is triggered on both rising and falling edges. Then, taking the difference between the two CCR values will give us the pulse width. See Figure 5 for a visualization of this process.

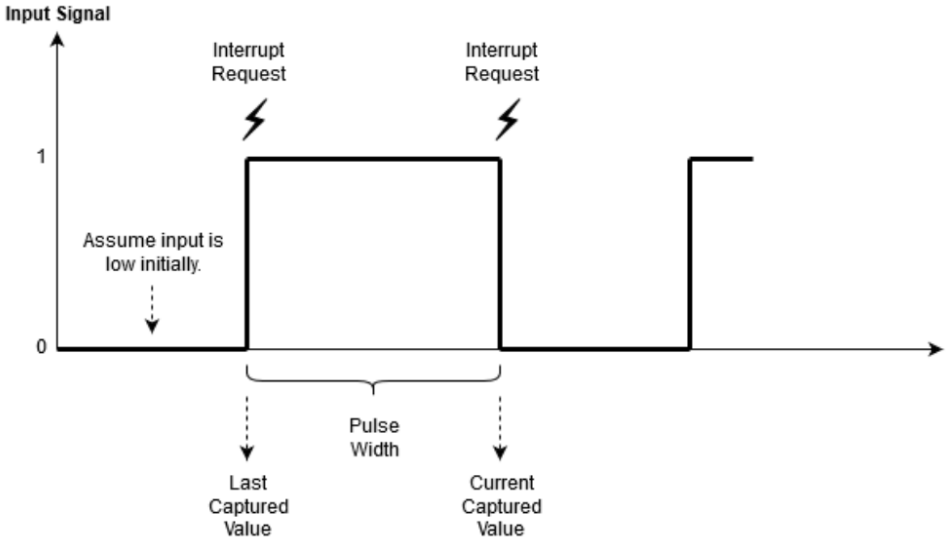


Figure 5: Computing Pulse Width Using Input Capture

Use the following steps to set up input capture for measuring the pulse width of the ultrasonic sensor's output. Refer to Sections 8.4 (RCC), 9.4 (GPIO), and 27.4 (Timer 4) in *STM32L4x6 Reference Manual* for details about registers and their bits.

1. Set up **PB6**.
   (a) Enable the clock for GPIO Port B.
   (b) Configure **PB6** to be used as alternative function `TIM4_CH1`.
   (c) Set **PB6** to no pull-up, no pull-down.
2. Enable Timer 4 in `RCC_APB1ENRx`.
3. Set the prescaler to 15.

4. Enable auto reload preload in the control register and set the auto reload value to its maximum value.
5. In the capture/compare mode register, set the input capture mode bits such that the input capture is mapped to timer input 1.
6. In the capture/compare enable register, set bits to capture both rising/falling edges and enable capturing.
7. In the DMA/Interrupt enable register, enable both interrupt and DMA requests. In addition, enable the update interrupt.
8. Enable update generation in the event generation register.
9. Clear the update interrupt flag.
10. Set the direction of the counter and enable the counter in the control register.
11. Enable the interrupt (`TIM4_IRQn`) in the NVIC and set its priority to 2.

You will have to implement the interrupt handling function `TIM4_IRQHandler()`, which should take care of computing the difference between two consecutive CCR values (the first from an interrupt on the rising edge and the second from an interrupt on the falling edge) to compute the pulse width. (You should not be computing the values between every two consecutive edges.) Remember to clear necessary flags. **Note**: If a counter overflow/underflow occurs, the difference of two consecutive CCR values may not correctly measure the time interval. Why do you think this is the case and how would you fix this issue? (*Hint*: What event occurs when the timer keeps counting, but reaches the maximum value?)

Within the `while` loop, write code that converts the sensor measurements to a distance measurements in *centimeters*. This value (just the number is fine) should be stored on the Stack (Hint: assign value to local variables) or monitor them in the watch window . If there is no object in range, the value is `0x00`.

6 Checkoff Requirement

1. Part A - LED toggles every second
2. Part B - LED brightness changes periodically and smoothly
3. Part B - Servo Motor turns along with the brightness of LED, and is also smooth
4. Part C - Distance sensor can measure distance from 5cm to 50cm. Display the distance measured on debug mode watch window.

7 References

[1] STM32L4x6 Advanced ARM-based 32-bit MCUs Reference Manual

[2] Yifeng Zhu, "Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C", ISBN: 0982692633

[3] HC-SR04 Ultrasonic Sensor User Manual

Appendix A - Timer Register Map

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	CR1	Reserved																				UIFREMAP		CKD[1:0]		ARPE	CMS[1:0]		DIR	OPM	URS	UDIS	CEN
	Value																																
0x04	CR2	Reserved																								TI1S	MMS[2:0]			CCDS			
	Value																																
0x08	SCMR	Reserved															SMS[3]	ETP	ECE	ETPS[1:0]		ETF[3:0]				MSM	TS[2:0]			OCCS		SMS[2:0]		
	Value																																
0x0C	DIER	Reserved																TDE	COMDE	CC4DE	CC3DE	CC2DE	CC1DE	UDE		TIE		CC4IE	CC3IE	CC2IE	CC1IE	UIE	
	Value																																
0x10	SR	Reserved																		CC4OF	CC3OF	CC2OF	CC1OF			TIF		CC4IF	CC3IF	CC2IF	CC1IF	UIF	
	Value																																
0x14	EGR	Reserved																								TG		CC4G	CC3G	CC2G	CC1G	UG	
	Value																																
0x18	CCMR1 Output Compare Mode	Reserved							OC2M[3]	Reserved							OC1M[3]	OC2CE	OC2M[2:0]			OC2PE	OC2FE	CC2S[1:0]		OC1CE	OC1M[2:0]			OC1PE	OC1FE	CC1S[1:0]		
	Value																																
	CCMR1 Input Capture Mode	Reserved																IC2F[3:0]				IC2PSC[1:0]		CC2S[1:0]		IC1F[3:0]				IC1PSC[1:0]		CC1S[1:0]		
	Value																																

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1C	CCMR2 Output Compare Mode	Reserved							OC4M[3]	Reserved							OC3M[3]	OC2CE	OC4M[2:0]			OC4PE	OC4FE	CC4S[1:0]		OC3CE	OC3M[2:0]			OC3PE	OC3FE	CC3S[1:0]	
	Value																																
	CCMR2 Input Capture Mode	Reserved																IC4F[3:0]				IC4PSC[1:0]		CC4S[1:0]		IC3F[3:0]				IC3PSC[1:0]		CC3S[1:0]	
	Value																																
0x20	CCER	Reserved																CC4NP		CC4P	CC4E	CC3NP		CC3P	CC3E	CC2NP		CC2P	CC2E	CC1NP		CC1P	CC1E
	Value																																
0x24	CNT	UIFCPY	CNT[30:16]															CNT[15:0]															
	Value																																
0x28	PSC	Reserved																PSC[15:0]															
	Value																																
0x2C	ARR	ARR[31:16]																ARR[15:0]															
	Value																																
0x34	CCR1	CCR1[31:16]																CCR1[15:0]															
	Value																																
0x38	CCR2	CCR2[31:16]																CCR2[15:0]															
	Value																																
0x3C	CCR3	CCR3[31:16]																CCR3[15:0]															
	Value																																
0x40	CCR4	CCR4[31:16]																CCR4[15:0]															
	Value																																
0x48	DCR	Reserved																				DBL[4:0]								DBA[4:0]			
	Value																																
0x4C	DMAR	Reserved																DMAB[15:0]															
	Value																																
0x50	OR1	Reserved																												TI1_RMP		ETR13_RMP	ITR1_RMP
	Value																																
0x60	OR2	Reserved																ETRSEL[2:0]				Reserved											
	Value																																

Appendix B - Timer Output Control

Control bits					Output states[1]	
MOE bit	**OSSI bit**	**OSSR bit**	**CCxE bit**	**CCxNE bit**	**OCx output state**	**OCxN output state**
1	X	X	0	0	Output disabled (not driven by the timer: Hi-Z) OCx=0, OCxN=0	
		0	0	1	Output disabled (not driven by the timer: Hi-Z) OCx=0	OCxREF + Polarity OCxN = OCxREF xor CCxNP
		0	1	0	OCxREF + Polarity OCx=OCxREF xor CCxP	Output Disabled (not driven by the timer: Hi-Z) OCxN=0
		X	1	1	OCREF + Polarity + dead-time	Complementary to OCREF (not OCREF) + Polarity + dead-time
		1	0	1	Off-State (output enabled with inactive state) OCx=CCxP	OCxREF + Polarity OCxN = OCxREF x or CCxNP
		1	1	0	OCxREF + Polarity OCx=OCxREF xor CCxP	Off-State (output enabled with inactive state) OCxN=CCxNP
0	0	X	X	X	Output disabled (not driven by the timer anymore). The output state is defined by the GPIO controller and can be High, Low or Hi-Z.	
	1		0	0		
			0	1	Off-State (output enabled with inactive state) Asynchronously: OCx=CCxP, OCxN=CCxNP (if BRK or BRK2 is triggered). Then (this is valid only if BRK is triggered), if the clock is present: OCx=OISx and OCxN=OISxN after a dead-time, assuming that OISx and OISxN do not correspond to OCX and OCxN both in active state (may cause a short circuit when driving switches in half-bridge configuration). **Note**: BRK2 can only be used if OSSI = OSSR = 1.	
			1	0		
			1	1		

1. When both outputs of a channel are not used (control taken over by GPIO), the OISx, OISxN, CCxP and CCxNP bits must be kept cleared.