15.4 Input Capture

As presented previously, a timer can be used for triggering an output at a specified time to general output signals (PWM output, comparator output). This section discusses the usage of timers as input capture.

The objective of input capture is to find the time span between the rising or falling transition events of an external signal, as shown in Figure 15-12. Captured events can be (1) either rising edges or falling edges, (2) only falling edges, or (3) only rising edges.

Input capture timers are useful in many applications such as measuring motor speed and position, calculating the time-of-flight for ultrasonic distance sensors, and communication between two remote devices.

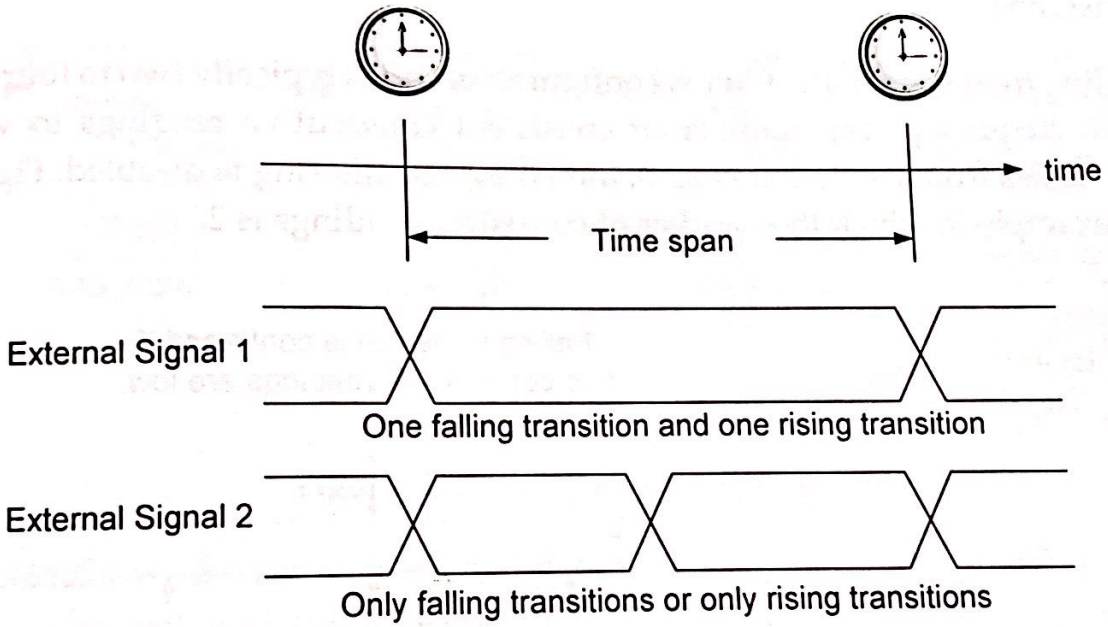

Figure 15-12. Input capture measures the time span between two consecutive events of an external signal. Timer can capture events (1) either rising or falling edges, (2) only rising edges, or (3) only falling edges, depending on the application's need. The time span is the product of the clock counter period and the difference between two consecutively captured values of the timer counter.

When a transition of an input signal occurs, the timer hardware records this time instant by copying the value of the free-run counter to the compare and capture register. When a long time span is measured, software must take care of the overflow of the free-running counter and use a variable to record the number of overflows.

Each input capture channel has an input pin, a filter, a counter, and an edge detector.

- The edge detector can detect only falling edges, only rising edges, or both.

- The timer counts according to a clock scaled by a prescaler constant, which can be any value between 1 and 65536.
- The filter specifies the number of events need to validate a transition on the input. If we need to capture each valid transition, software should disable the external trigger filtering by setting the External Trigger Filter (ETF) to zero.

Each input channel also has a simple digital input filter to remove noise pulses in an input signal. For example, if the input signal of a push button takes 10 internal clock cycles to become stable, then we can make the filter duration last longer than 10 clock cycles. We can validate the transition by repeatedly sampling the inputs. If a sequence of consecutive samples remain unchanged and stay at the same level, then this level is considered as a stable input. If a noise spike occurs causing the input to change during any of the consecutive sampling points, the input is unstable and can be ignored, thus filtering out noisy transitions.

The sampling frequency of the filter is configurable, and is typically two to four times the timer clock frequency. The number of consistent consecutive readings to validate a transition ranges from 0 to 8. If it is zero, then the noise filtering is disabled. Figure 15-13 shows an example in which the number of consistent readings is 2.

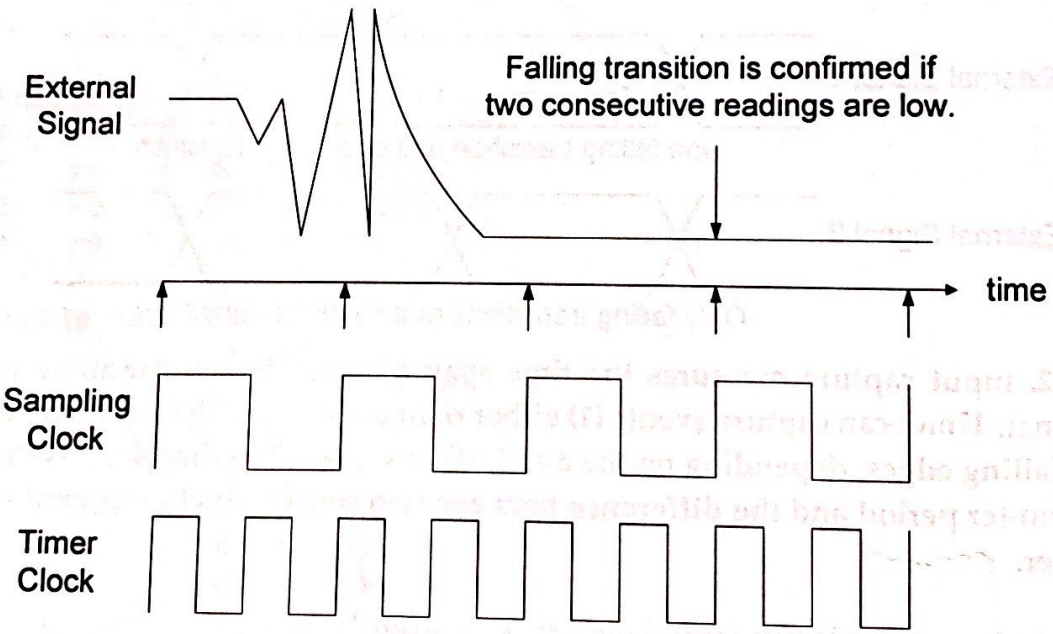

Figure 15-13. An example of filtering noise in the external signal. The falling transition is confirmed if two consecutive sample readings are low. The sampling frequency is 1/8 of the timer frequency at which the timer counter is incremented or decremented.

The difference between two consecutive transitions measures an elapsed time span. If the input signal is periodic, the difference of the counters captured at two consecutive rising edge measures the period of the waveform. Similarly, the difference between a rising

edge and a falling edge measures the pulse width. In this way, we can calculate the frequency and duty cycle of the input waveform.

15.4.1 Input Capture Timer Diagram

Software can select internal clocks, external clocks, or internal trigger input (such as another timer) as the clock of the 16-bit free-running counter. The clock frequency can be divided by a 16-bit clock prescaler (PSC) to reduce the clock frequency for the free-running counter.

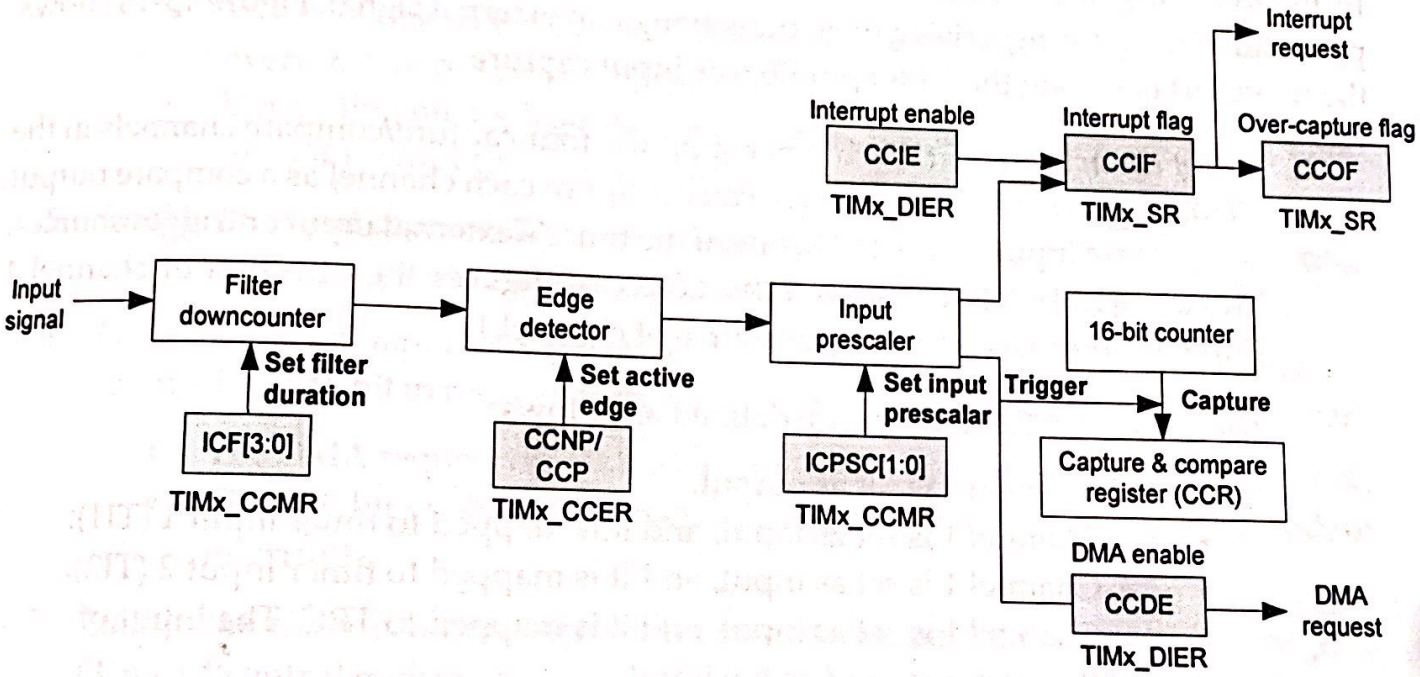

Figure 15-14. Diagram of input capture

Figure 15-14 shows the timer diagram for input capture. When the number of transitions on an input channel reaches the threshold defined by the *input capture prescaler* bits (ICPSC) in the capture/compare mode register (TIMx_CCMR), a capture occurs and the processor automatically performs the following operations.

- It latches the value of the free-running counter to the capture/compare register (TIMx_CCR) corresponding to that channel. There is a 16-bit capture/compare for each capture channel (TIMx_CCR1 for channel 1, TIMx_CCR2 for channel 2, TIMx_CCR3 for channel 3, and TIMx_CCR4 for channel 4).
- In the status register (TIMx_SR), the capture/compare interrupt flag (CCIF) corresponding to that channel is set by hardware on a capture. If the CCIF flag has already been set, then hardware sets the corresponding capture/compare over-capture flag (CCOF) in the status register (TIMx_SR). Software clears the CCIF flag by writing it to 0, or hardware automatically clears it if software reads the corresponding CCR register. Only software can clear the CCOF flag.

- If software enables the interrupt by setting the capture/compare interrupt enable bit (CCIE) of the DMA interrupt enable register (TIMx_DIER), an interrupt corresponding to that channel is then generated.
- If software enables DMA by setting the capture/compare DMA request enable bit (CCDE) of the TIMx_IER register, a DMA request is then generated.

15.4.2 Configuring Input Capture

In the following, we use the channel 1 of timer 4 as an example to illustrate the basic procedure for capturing a rising edge transition of an external signal. Figure 15-15 shows the flowchart of setting the GPIO pin PB.6 as input capture.

1. ***Select the active input.*** Each timer supports four capture/compare channels in the STM32L processor, and software can configure each channel as a compare output or a capture input connected to one of the timer's external input or trigger sources. The CC1S[1:0] bits in register TIM4_CCMR1 configures the direction of channel 1 (input or output) and the input source of channel 1.

   The usage of CC1S[1:0] bits are defined as follows:

   - 00: Channel 1 is set as output.
   - 01: Channel 1 is set as input, and it is mapped to timer input 1 (TI1).
   - 10: Channel 1 is set as input, and it is mapped to timer input 2 (TI2).
   - 11: Channel 1 is set as input, and it is mapped to TRC. The input of another timer is used as the input.

   To link to the timer input 1 (TI1), we need to set CC1S bits to 01. Note, if CC1S bits are 00, then the channel is configured as a compare output. When a channel is set up as capture input, its corresponding CCR register becomes read-only.

2. ***Set the input filter.*** The IC1F bits in register TIM4_CCMR1 define the frequency used to sample TI1 and the length of digital filtering applied to TI1. The digital filtering is to check whether a number of consecutive readings of an input remain the same. If yes, the input is stable and the input transition is valid.

   - For example, if IC1F is 0000, the digital filtering is disabled.
   - If IC1F is 0010, the sampling frequency is the counter frequency, and if four consecutive samplings remain the same, then the transition is valid.

3. ***Set the active edge.*** The CC1P bit and CC1NP bit in TIM4_CCER (Timer 4 capture/compare enable register) collectively determines the edge of an active transition on the TI1 channel.

- If CC1NP = 0 and CC1P = 0, then the edge detection is configured to capture only rising edges.
- If CC1NP = 0 and CC1P = 1, only falling edges are captured.
- If CC1NP = 1 and CC1P = 1, then both falling and rising edges are captured.

4. ***Set the input prescaler.*** If we want to capture the event each time an edge is detected in channel 1, the prescaler, defined by the IC1PSC bits in register TIM4_CCMR1 (Timer 4 capture/compare mode register 1), should be set to 0.
   - When IC1PSC is 01, 10, and 11, the input capture is then performed once every 2, 4, and 8 events, respectively.
   - Depending on applications' needs, such as noise filtering, we might want to wait for multiple valid transitions before the counter is latched.
5. ***Enable the input capture.*** The input capture of channel 1 can be enabled by setting the CC1E bit in register TIM4_CCER (Timer 4 capture/compare enable register).
6. ***Enable interrupt and DMA if needed.*** The related interrupt request is enabled by setting the CC1E bit in register TIM4_DIER (Timer 4 DMA/interrupt enable register).
   - The DMA request is enabled by setting the CC1DE bit in register TIM4_DIER.
   - The TIE bit enables trigger interrupt, and the UIE bit enables the update interrupt.
7. ***Enable the timer counter.*** Sets the counter enable bit (CEN) in register TIM4_CR1 (Timer 4 control register 1).

Assuming the timer is counting up, Example 15-5 shows the implementation of the timer interrupt service routine, which calculates the time interval between two consecutive capture events.

This example code does not take care of the counter overflow and, therefore, the time interval may be negative occasionally. A more robust code should check the update interrupt flag (TIM_SR_UIF) of the timer status register (TIM4->SR). If the update event occurs (*i.e.*, overflow during counting up or underflow during counting down), the time interval should be calculated differently.

```
void TIM4_IRQHandler() {
   if(TIM4->SR & TIM_SR_CC1IF != 0) {// If update flag is set
      current_value = TIM4->CCR1; // Reading CCR1 clears CC1IF interrupt flag
      time_interval = current_value - old_value; // if counting up
      old_value = current_value;
    }
}
```

Example 15-5. Timer interrupt handler in C. Assume the timer counts up.

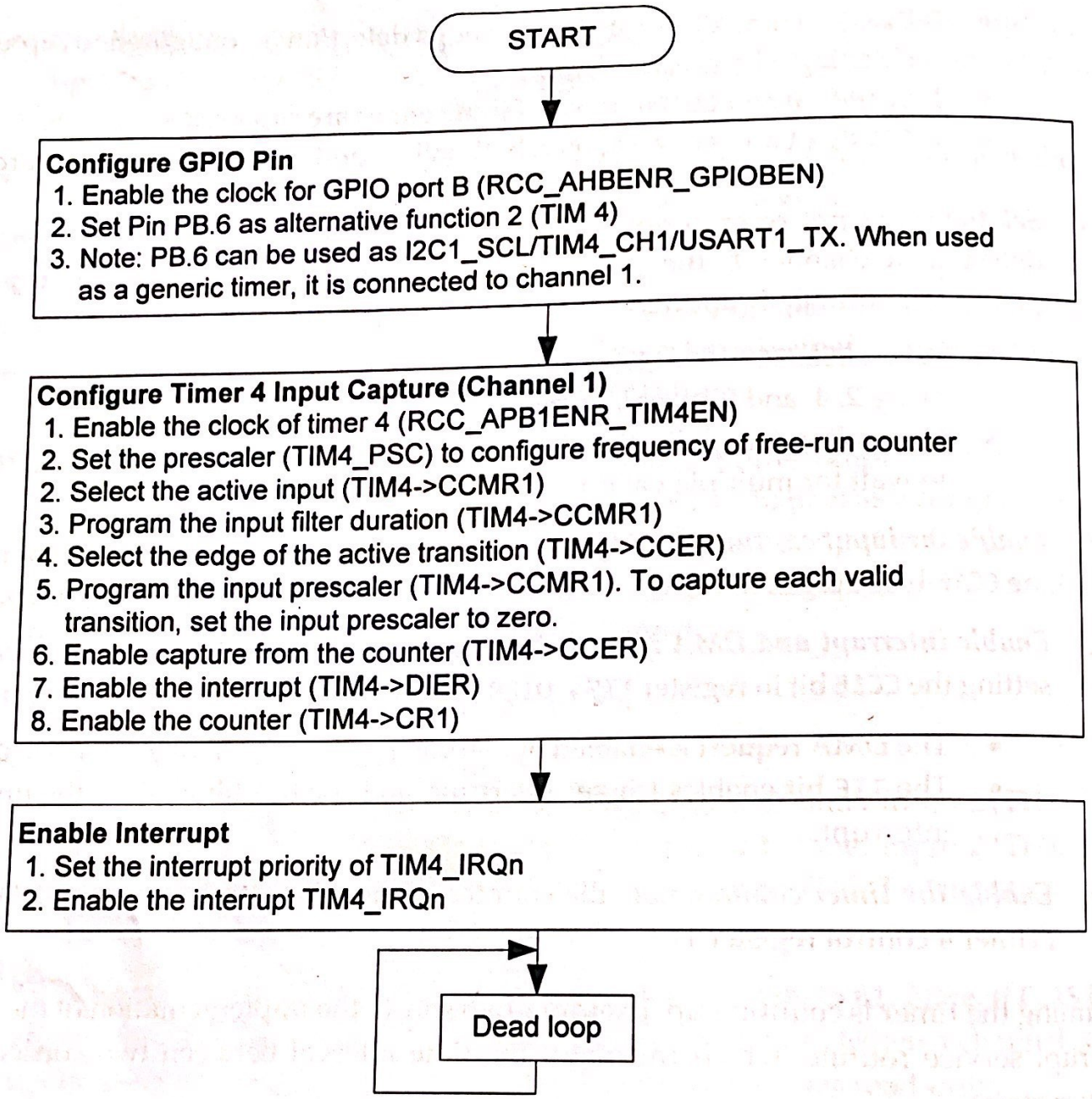

Figure 15-15. Flowchart to set the GPIO pin PB.6 as input capture

Table 15-5 shows the configuration of pin PB.6 as timer function. The pin is set as no pull-up and no pull-down, and alternative function mode.

GPIO Pin	Connection	Mode	AF	Pull-up/Pull-down	Clock
Port B pin 6	Blue LED	AF	TIM4_CH1	No pull-up, No pull-down	40 MHz

Table 15-5. Configuration of PB.6 as timer function

The following gives an example code of setting pin PB.6 as input capture.

```
#include <stdint.h>
#include "stm32l1xx.h"
```

```
int value = 0;

int main() {
   // Enable GPIOB clock
   RCC->AHBENR  |=  RCC_AHBENR_GPIOBEN;

   // Configure GPIO port B pin 6 as AF function (i.e. TIM4_CH1)
   // Alternation function of PB6 include I2C1_SCL, TIM4_CH1, or USART1_TX.
   // with no pull-up/pull-down, and enable the clock of GPIO port B

   // Add your code for GPIO pin and configuration here
   ...

   // Configure channel x to perform input capture
   // (1) CCRx stores the value of the counter after a transition detected
   //     by corresponding ICx signal.
   // (2) When a capture occurs, CCxIF flag of TIMx_SR is set.
   //     If CCxIF is already set, then the over-capture flag CCxOF of
   //     TIMx_SR is set.
   // (3) CCxIF is cleared by software by writing it to 0 or
   //     by reading TIMx_CCRx.
   // (4) CCxOF is cleared by software by writing it to 0.

   // Enable the clock of timer 4
   RCC->APB1ENR |= RCC_APB1ENR_TIM4EN;

   // Set up an appropriate prescaler to slow down the clock of timer counter
   TIM4->PSC = 127;

   // Set the direction as input and select the active input
   TIM4->CCMR1  &= ~TIM_CCMR1_CC1S;
   TIM4->CCMR1  |= 0x1; // CC1S[1:0] for channel 1:
                        // 00 = output
                        // 01 = input, CC1 is mapped on timer Input 1
                        // 10 = input, CC1 is mapped on timer Input 2
                        // 11 = input, CC1 is mapped on slave timer

   // Program the input filter duration
   // Disable digital filtering by clearing IC1F[3:0] bits
   // because we want to capture every event
   TIM4->CCMR1  &= ~TIM_CCMR1_IC1F;

   // Select the edge of the active transition
   // Detect both rising and falling edges in this example
   // CC1NP:CC1P bits
   //   00 = rising edge,
   //   01 = falling edge,
   //   10 = reserved,
   //   11 = both edges
   TIM4->CCER  |= (1<<1 | 1<<3);       // Both rising and falling edge
```

```
    // Program the input prescaler
    // To capture each valid transition, set the input prescaler to zero;
    // IC1PSC[1:0] bits (input capture 1 prescaler)
    TIM4->CCMR1 &= ~(TIM_CCMR1_IC1PSC);

    // Enable capture of the counter
    // CC1E: Capture/compare output enable for channel 1
    // 0 = capture disabled;  1 = capture enabled
    TIM4->CCER  |= TIM_CCER_CC1E;

    // Enable related interrupts
    // CC1IE: Capture/compare interrupt enable for channel 1
    TIM4->DIER  |= TIM_DIER_CC1IE;

    // CC1DE: Capture/compare DMA request enable for channel 1
    TIM4->DIER  |= TIM_DIER_CC1DE;

    // Enable the counter
    TIM4->CR1   |= TIM_CR1_CEN;

    // Set priority to 1
    NVIC_SetPriority(TIM4_IRQn, 1);

    // Enable TIM4 interrupt in NVIC
    NVIC_EnableIRQ(TIM4_IRQn);

    while(1);
}
```

Example 15-6. Implementation of the initialization of a timer for input capture

15.4.3 Interfacing to Ultrasonic Distance Sensor

An ultrasonic distance sensor has one transmitter and one receiver. The transmitter generates short bursts of high-frequency ultrasonic waves. The receiver detects any wave reflected back from the target, as shown in Figure 15-16.

Without contacting with the target physically, it measures the difference in time between sending waves and receiving reflected waves. The distance is then calculated as follows:

$$Distance = \frac{Round\ Trip\ Time \times Speed\ of\ Sound}{2}$$

One example application is an automatic door opener, which opens a door when a person approaches. Compared with optical distance sensors, ultrasonic distance sensors are low cost but less accurate.

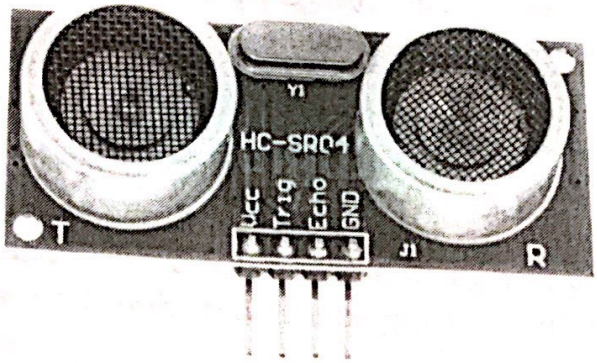

Figure 15-16. Ultrasonic distance sensor of HC-SR 04
(VCC = +5V DC, Trig = Trigger input to sensor, Echo = Echo time output of sensor)

To start a distance measurement, the processor should send a high pulse signal (≥3.2V) with a width of 10 μs to the trig pin. The ultrasonic transmitter then sends out 8 cycles of 40-KHz ultrasonic waves (*i.e.* for 200 μs), which is greater than the upper limit of human hearing range (typically 20 KHz).

When the ultrasonic receiver detects any waves reflected back within a predefined time window, it generates a high pulse (5V) on the echo pin. The pulse width is linearly proportional to the distance of the nearest objects.

Specifically, if the width of the pulse on the echo pin is in microseconds (μs), the distance, measured in centimeters, is calculated as follows:

$$Distance\ (cm) = \frac{Pulse\ Width\ (\mu s)}{58}$$

The sensor can measure a distance of 2 cm – 400 cm, with a resolution of 0.3 cm, and the corresponding echo pulse width is 150 μs – 25 ms. When the sensor detects no object, the echo pulse width is 38 ms.

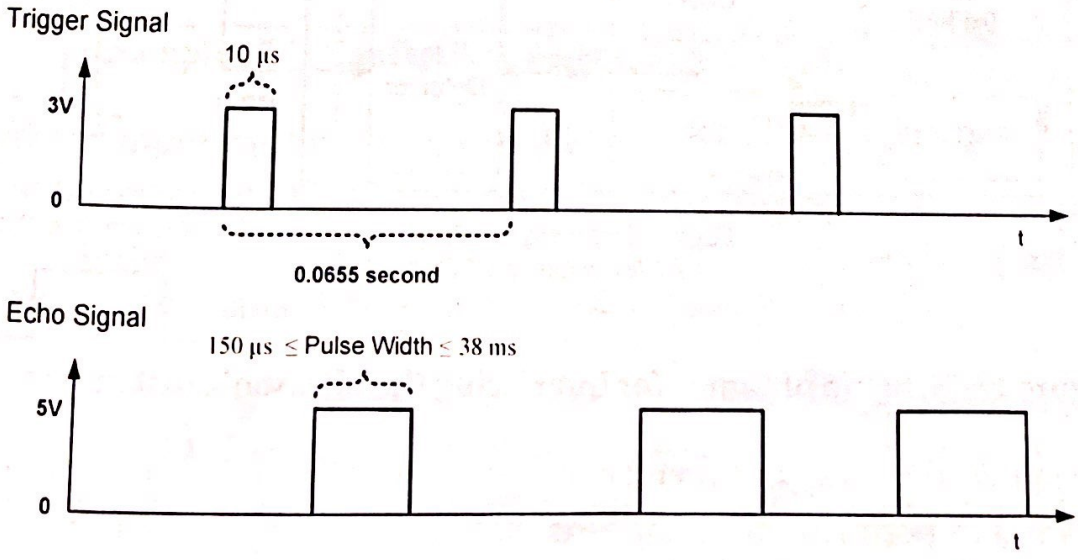

Figure 15-17. Trigger and echo signals

Suppose we want to measure the distance every 0.0655 seconds, as shown in Figure 15-17. We can use two timers to interface the ultrasonic distance sensor.

- The first timer, which is set as output compare and connected to the trigger pin of the sensor, generates a pulse signal with a width of 10 μs.
- The second timer, which is set as input capture and connected to the echo pin of the sensor, measures the pulse width of the echo signal. The second timer should be able to detect both the rising and falling edge of the echo signal.

Note the distance measured is only in one particular direction with limited angular resolution. We can use multiple ultrasonic sensors to detect objects in several directions.

It is also possible that an object reflects ultrasonic waves away if the target surface is oriented at unfavorable angles, resulting in the object being undetected. Also, objects with a soft or irregular surface might not reflect enough ultrasonic waves back, and accordingly, ultrasonic sensors fail to detect them. Moreover, sounds travel slower in colder air, and thus the program should perform some calibration to achieve better accuracy.

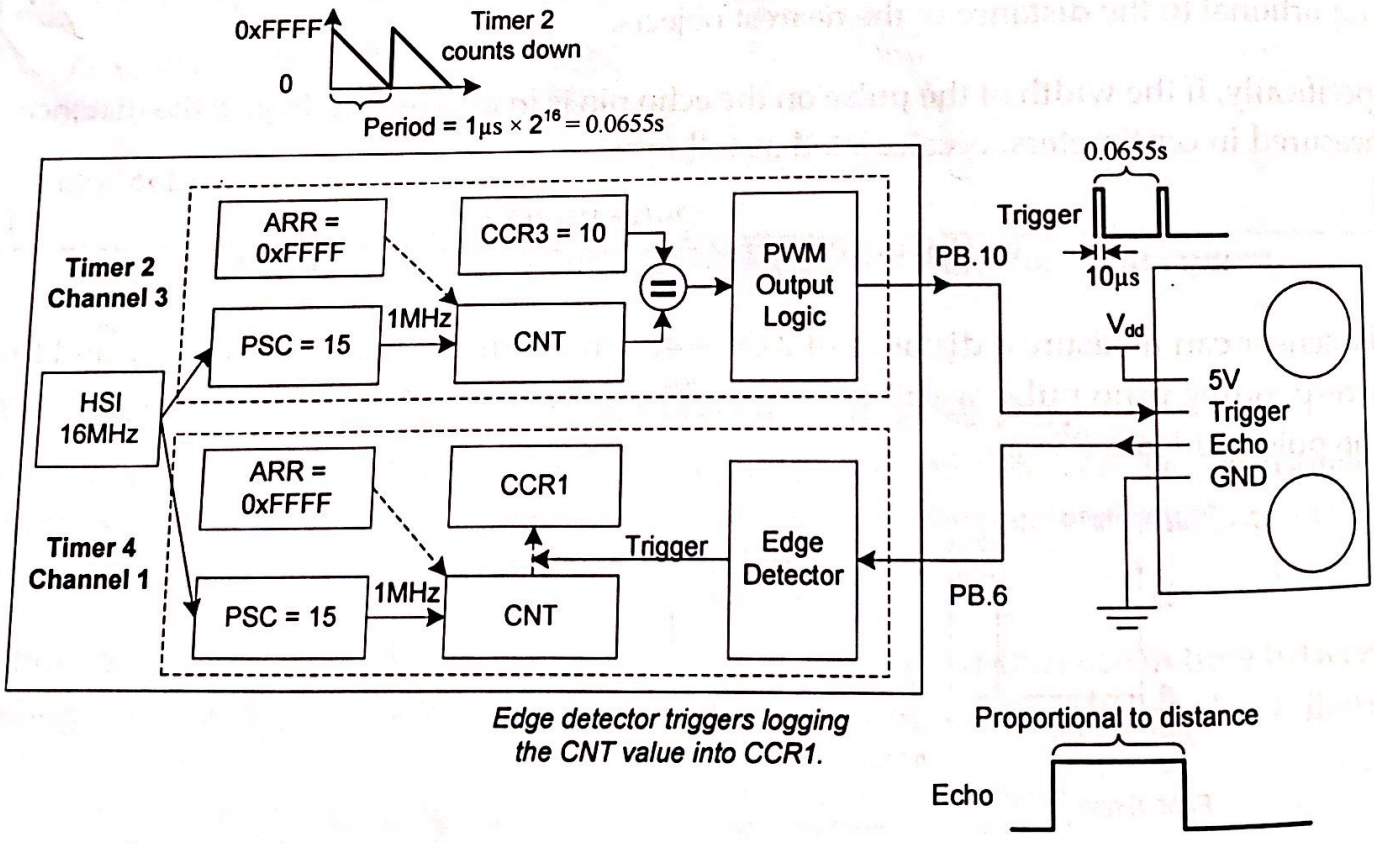

Figure 15-18. Setup of timers for interfacing the ultrasonic distance sensor.

Figure 15-18 shows the setup of timer 2 and timer 4. On the STM32L discovery board, we can use pin PB.6 to perform input capture. Because PB.6 connects to the blue LED, we can use the brightness of the LED to observe the distance measured. We can also use pin

PB.5 to generate the trigger signal periodically. The STM32L processor allows pin PB.6 has the alternative functions of I2C1_SCL, TIM4_CH1, and USART1_TX, and pin PB.10 has the alternative functions of I2C2_SCL, USART3_TX, TIM2_CH3, and LCD_SEG10. We set PB.6 to be the input capture on channel 1 of timer 4 (TIM4_CH1), and PB.10 to be the output of channel 3 of timer 2 (TIM2_CH3).

Suppose the processor need to use 16-MHz HSI (high-speed internal) oscillator as the system clock. Because the default system clock is MSI (multi-speed internal), we have to set up HSI, as shown in the following code.

```
; Select HSI (high-speed internal, 16 MHz, 1% accuracy) as the system clock.
HSI_init  PROC
          EXPORT HSI_init
          LDR  r0, =RCC_BASE

          ; Turn on HSI oscillator
          LDR  r1, [r0, #RCC_CR]
          ORR  r1, r1, #RCC_CR_HSION
          STR  r1, [r0, #RCC_CR]

          ; Select HSI as system clock
          LDR  r1, [r0, #RCC_CFGR]
          BIC  r1, r1, #RCC_CFGR_SW
          ORR  r1, r1, #RCC_CFGR_SW_HSI
          STR  r1, [r0, #RCC_CFGR]

          ; Wait for HSI stable
WaitHSI   LDR  r1, [r0, #RCC_CR]
          AND  r1, r1, #RCC_CR_HSIRDY
          CMP  r1, #0
          BEQ  WaitHSI

          BX   LR
          ENDP
```

Example 15-7. Configuring high-speed internal clock (HSI)

Timer 4 works as input capture to record the time instant when a rising or falling edge takes place in the external ECHO signal, as shown in Figure 15-18. Software should set the update interrupt enable flag (`UIE`) and the capture event flag (`CCxIE` for channel `x`) in the DMA/interrupt enable register (`DIER`) to allow these interrupts.

Timer 4 is set as counting upward. A counter overflow occurs when the 16-bit timer counter reaches `0xFFFF`. Because the counter increments every 1μs, an overflow occurs every `0.0655s` in this example. When an overflow occurs, the timer generates an interrupt and sets up the `UIF` flag bit in the interrupt status register (`SR`).

It is possible that the counter can overflow when measuring the pulse width of the ECHO signal. Thus, we need a variable (named *overflow* in the following code) that counts the number of overflows within one pulse. The time span can be calculated as follows:

$$Time\ Span\ (\mu s) = (Current\ Counter\ Value - Last\ Counter\ Value) + 65536 \times Number\ of\ Overflows$$

After calculating the time span, the program should use the current counter value to update the last counter value to prepare the next calculation.

The following assembly code shows the implementation of the interrupt handler for timer 4.

```
timespan     DCD  0   ; the pulse width
lastCounter  DCD  0   ; the timer counter value of last capture event
overflow     DCD  0   ; Counter the number of overflows

TIM4_IRQHandler PROC
             EXPORT TIM4_IRQHandler
             PUSH {r4, r6, r10, lr}

             LDR r0, =TIM4_BASE
             LDR r2, [r0, #TIM_SR]
             AND r3, r2, #TIM_SR_UIF      ; Check update event flag
             CBZ r3, check_CC

             LDR r3, =overflow
             LDR r1, [r3]
             ADD r1, r1, #1               ; Increment overflow counter
             STR r1, [r3]

             BIC r2, r2, #TIM_SR_UIF      ; Clear update event flag
             STR r2, [r0, #TIM_SR]

check_CC     AND r2, r2, #TIM_SR_CC1IF    ; Check capture event flag
             CBZ     r2, exit

             ; Fetch the current CCR value
             LDR r0, =TIM4_BASE
             LDR r1, [r0, #TIM_CCR1]      ; Read the capture value

             LDR r2, =lastCounter
             LDR r0, [r2]                 ; load the last counter value
             STR r1, [r2]                 ; save the new counter value
             CBZ r0, clearOverflow

             LDR r3, =overflow
```

```
            LDR r4, [r3]                ; load the overflow value
            LSL r4, r4, #16             ; x 65536
            ADD r6, r1, r4
            SUB r10, r6, r0             ; r10 = timer counter difference
            LDR r2, =timespan
            STR r10, [r2]

clearOverflow
            MOV r0, #0
            LDR r3, =overflow
            STR r0,  [r3]               ; clear overflow counter

exit        POP {r4, r6, r10, pc}
            ENDP
```

In the above code, we use "`LSL r4, r4, #16`" instruction to replace multiplication with 65536 (*i.e.* 2^{16}). A shift instruction runs much faster than a multiplication instruction. Additionally, the *lastCounter* variable is initialized to `0`.

The code above calculates the time span between a rising edge and a falling edge. We can improve the code by only calculating the time span from a rising edge to a falling edge, and ignore the time span from a falling edge to a rising edge.

One common mistake is that programmers often forget to clear the update event flag in the interrupt status register (SR). In the above code, the hardware sets up two flags in SR. It sets the update flag when a counter overflow (or a counter underflow if the counter counts down) occurs. It also sets the capture flag when a rising edge or a falling edge of an external signal is detected.

Hardware automatically clears the capture flag after software reads the timer data register (DR). However, the software has to clear the update flag in the interrupt handler of the timer. If it were not cleared, the timer interrupt handler would be called repeatedly, making the processor have no time to run other codes with a lower priority.

Software must clear the update flag.

15.5 Exercises

1. Suppose we want to generate a PWM signal with a duty cycle of 0.25. The processor clock is 32 MHz. We want the PWM signal to have a fixed frequency of 320 Hz. How would you design the prescaler (PSC), auto-reload register (ARR), and compare and capture (CCR)? Show your calculation. The timer output mode

is set as follows: the PWM output is high if the counter is larger than or equal to the content of CCR.

2. Use an oscilloscope to measure duty cycles. Suppose the default MSI clock is used, which is 2.097 MHz.
   a. What is the relationship between the system clock, the counter clock CK_CNT, the prescaler TIM4_PSC, and the pulse period measured?
   b. We want to keep TIM4_ARR fixed, but set TIM4_CCR1 to three different values, as listed below. How would you set up the ARR, PSC, and CCR register values? Calculate the duty measured and verify the correctness.
      - Case 1: TIM4_CCR1 = 1/6 * (TIM4_ARR + 1)
      - Case 2: TIM4_CCR1 = 1/3 * (TIM4_ARR + 1)
      - Case 3: TIM4_CCR1 = 1/2 * (TIM4_ARR + 1)

Case	TIM4_CCR1	Pulse Width	Pulse Period
#1			
#2			
#3			

3. Suppose the HSE (high-speed external clock) of 16 MHz is selected as the clock of the timer. In order to generate a 1Hz square wave with a duty cycle of 50%, how would you set up the timer? Indicate your counting mode and show the value of ARR, CCR, and PSC registers.
4. Write an assembly program that uses the output compare function of a timer to toggle an LED every second.
5. Write an assembly program that generates a PWM output signal to dim an LED periodically.
6. Write an assembly program that uses PWM to generate a square wave signal with a frequency of 440 Hz and a duty cycle of 50% (*i.e.* musical tone A).
7. Write an assembly program that uses PWM to control a stepper motor via micro stepping.
8. Write an assembly program that uses the input capture function to measure the frequency of an external signal. Use a function generator to generate a 1HZ square wave. Send the square wave to the STM32L board to verify the correctness of your assembly program.
9. Write an assembly program that uses the HC-SR04 ultrasonic distance sensor presented in Chapter 15.4.3 to measure the distance to an object.