



Centrale Nantes

**MAC : 5<sup>th</sup> Lab's Report**  
**Interrupts — Ultrasonic sensor**

1<sup>st</sup> year Embedded Systems Engineering

**By**

EL KHAYDER Zakaria

SAOUTI Rayan

**Professor**

BRIDAY Mikael

December 24, 2023

Session

2023-2024

Made with L<sup>A</sup>T<sub>E</sub>X

# Contents

<b>Contents</b>	<b>I</b>
<b>Listings</b>	<b>II</b>
<b>1 Trigger Signal</b>	<b>1</b>
1.1 Timer . . . . .	1
1.2 Interrupt Configuration . . . . .	1
1.3 Interrupt Handler: Trigger signal generation . . . . .	1
<b>2 Echo signal</b>	<b>2</b>
2.1 Theory . . . . .	2
2.2 Setup . . . . .	3
2.2.1 Timer . . . . .	3
2.2.2 External Interrupt . . . . .	3
2.3 Interrupt handler . . . . .	4
<b>3 Interface</b>	<b>4</b>
<b>4 Robustness</b>	<b>5</b>
<b>Resources</b>	<b>6</b>

## Listings

1	TIM6 Configuration . . . . .	1
2	TIM6 Interrupt enable . . . . .	1
3	TIM6 Interrupt handler: Trig signal generation . . . . .	2
4	TIM7 Config . . . . .	3
5	Echo interrupt setup . . . . .	3
6	TIM6 Interrupt handler — Updated . . . . .	3
7	Echo interrupt handler . . . . .	4
8	Main loop . . . . .	4
9	Sensor availability checker . . . . .	5

# 1 Trigger Signal

## 1.1 Timer

We should set up one of the internal STM32 timers as an internal interrupt. We chose to use TIM6 for that.

```
1 void setup()
2 {
3     // input clock = 64MHz.
4     RCC->APB1ENR |= RCC_APB1ENR_TIM6EN | RCC_APB1ENR_TIM7EN;
5     // reset peripheral (mandatory!)
6     RCC->APB1RSTR |= RCC_APB1RSTR_TIM6RST | RCC_APB1RSTR_TIM7RST;
7     RCC->APB1RSTR &= ~(RCC_APB1RSTR_TIM6RST | RCC_APB1RSTR_TIM7RST);
8
9     // Configure timer
10    TIM6->CNT = 0; // Reset counter
11    TIM6->SR = 0; // Reset status register
12    TIM6->PSC = 64000 - 1; // Prescaler: 64K => F = 1K => T = 1ms
13    TIM6->ARR = 100 - 1; // 100ms
14    TIM6->CR1 |= TIM_CR1_CEN; // Timer enable
15 }
```

Listing 1: TIM6 Configuration

## 1.2 Interrupt Configuration

```
1 void setup() {
2     ...
3
4     // enable interrupt
5     TIM6->DIER |= TIM_DIER_UIE;
6     NVIC_EnableIRQ(TIM6_DAC1_IRQn);
7 }
```

Listing 2: TIM6 Interrupt enable

## 1.3 Interrupt Handler: Trigger signal generation

Whenever we get a TIM6 interrupt (each 100ms), we should send a  $10\mu\text{s}$  HIGH signal in the TRIG\_ECHO pin. Since the TRIG\_ECHO serves both as an input and an output for our ultrasonic sensor, we should set the pin to output at the beginning of our interrupt handler, and then flip it back to output so we can read the echo signal from the sensor.

To handle our TIM6 interrupt, we should declare a function with exactly TIM6\_DAC1\_IRQHandler as a

name, but given we are using C++ as our programming language and not C, we have to be careful of name mangling, hence declaring our interrupt handlers as external C functions using `extern "C"`

```

1 extern "C" void TIM6_DAC1_IRQHandler(void)
2 {
3     pinMode(TRIG_ECHO, OUTPUT); // Set pin as OUTPUT
4
5     digitalWrite(TRIG_ECHO, 1); // Set HIGH
6     for (volatile int i = 0; i < 50; i++); // A short delay (we measured around 12us)
7     digitalWrite(TRIG_ECHO, 0); // Set LOW
8
9     pinMode(TRIG_ECHO, INPUT); // Set the pin back to input
10
11     TIM6->SR &= ~TIM_SR_UIF; // acknowledge
12 }

```

Listing 3: TIM6 Interrupt handler: Trig signal generation

## 2 Echo signal

### 2.1 Theory

Once the ultrasonic sensor has emitted the ultrasonic burst, it will set the **Echo** signal to **HIGH** until it receives back the signal it generated, and then it sets **Echo** signal back to **LOW**. The measured time between the Rising and the Falling edges is how long the ultrasonic burst took to travel back and forth ( $2d$ ).

Given the ultrasonic burst travels at the speed of sound  $s \approx 343m/s$  (at 20 °C), we can pretty accurately estimate the distance between the object and the sensor up to  $\pm 2cm$ .

We will be using TIM7 with a resolution of  $1 \mu s$

$$\begin{aligned}
 s_{(m/s)} = \frac{2d_{(m)}}{t_{(s)}} &\Rightarrow d_{(m)} = \frac{s_{(m)} \cdot t_{(s)}}{2} \\
 &= \frac{343_{(m/s)} \cdot t}{2} \\
 &= 171.5_{(m/s)} \cdot t_{(s)} \\
 \Rightarrow d_{(cm)} &= 171.5_{(m/s)} \cdot \frac{10^2_{(cm/m)}}{10^6_{(\mu s/s)}} \cdot t_{(\mu s)} \\
 &= 0.01715_{(cm/\mu s)} \cdot t_{(\mu s)} \\
 &\approx \frac{t_{(\mu s)}}{58_{(\mu s/cm)}}
 \end{aligned}$$

## 2.2 Setup

### 2.2.1 Timer

```
1 void setup() {
2     ...
3
4     // Configure timer
5     TIM7->CNT = 0; // Reset Counter
6     TIM7->SR = 0; // Reset status register
7     TIM7->PSC = 64 - 1; // Prescaler: 64 => F = 1Mhz => T = 1us
8     TIM7->ARR = 50000 - 1; // 50ms (will come in handy later)
9     TIM7->CR1 |= TIM_CR1_CEN;
10 }
```

Listing 4: TIM7 Config

### 2.2.2 External Interrupt

Thanks to the provided function helpers, configuring external interrupts is easy.

We attach an external interrupt to TRIG\_ECHO pin on each CHANGE (rising and falling edges)

```
1 void setup() {
2     ...
3
4     EXTI->IMR |= EXTI_IMR_MR10;
5     attachInterrupt(TRIG_ECHO, CHANGE);
6 }
```

Listing 5: Echo interrupt setup

Since we are using the same pin for both Trig and Echo, and we are listening to changes on this pin now, this means that whenever we manipulate the pin for the Trig signal, we will get an interrupt because we change its state. We need to update the previous Trig signal generation code to disable the External interrupts on this pin before sending the signal, and then re-enabling it.

```
1 extern "C" void TIM6_DAC1_IRQHandler(void)
2 {
3     EXTI->IMR &= ~EXTI_IMR_MR10; // Disable pin 10 external interrupt
4
5     ...
6
7     EXTI->IMR |= EXTI_IMR_MR10; // Re-enable pin 10 external interrupts
8
9     TIM6->SR &= ~TIM_SR_UIF; // acknowledge
10 }
```

Listing 6: TIM6 Interrupt handler — Updated

## 2.3 Interrupt handler

Whenever we get an external interrupt from Echo pin, we have to determine whether it is a rising or falling edge:

- Rising: We reset the TIM7 counter
- Falling: We save the current TIM7 counter value in a global variable that we can access later on to calculate the distance using the previous equation.

```
1 volatile uint32_t dt_us = 0; // Let's not forget the volatile keyword so that the compiler
  does not do anything funny
2
3 extern "C" void EXTI15_10_IRQHandler(void)
4 {
5     if (digitalRead(TRIG_ECH0) == 1) // Rising edge
6         TIM7->CNT = 0; // Reset
7     else // Falling edge
8         dt_us = TIM7->CNT; // Save
9
10    EXTI->PR |= EXTI_PR_PR15; // acknowledge
11 }
```

Listing 7: Echo interrupt handler

## 3 Interface

Now that we have all the parts that we need, maybe we should display it to the user, you know... the purpose of this device?

We can use the functions already provided by `tft.h` to manipulate the display and draw the bar graph.

```
1 int main(void)
2 {
3     setup();
4     Tft.setup();
5
6     while(1) {
7         Tft.erase();
8
9         uint32_t distance = dt_us / 58; // cm
10
11         const int16_t MAX_HEIGHT = Tft.height() - 20,
12                     MIN_HEIGHT = 4,
13                     MAX_DISTANCE = 100,
14                     MIN_DISTANCE = 2;
```

```

15
16     float per = (float)distance / (float)(MAX_DISTANCE - MIN_DISTANCE);
17
18     if (per > 1)
19         per = 1;
20
21     int16_t height = MIN_HEIGHT + (float)(MAX_HEIGHT - MIN_HEIGHT) * per;
22
23     Tft.fillRect(
24         Tft.width() / 2 - 25,
25         MAX_HEIGHT - height,
26         50,
27         height,
28         ST7735_BLUE * (1 - per) + ST7735_RED * per);
29
30     Tft.setCursor(3, 80);
31     Tft.print("Distance: ");
32     Tft.print(distance);
33     Tft.println(" cm");
34
35     for (volatile int i = 0; i < 2000000; i++); // Delay
36 }
37 }

```

Listing 8: Main loop

## 4 Robustness

The last part is adding a mechanism to detect if the sensor is present or not. For this, we will start a 50ms timer whenever we send a **Trig** signal. If the duration elapses without receiving any response, we assume that the sensor is unavailable (disconnected or broken).

We can use the TIM7 that we already use to get the travel time. If you can remember, we have already configured its ARR register to 50ms, the only thing left now is to enable an interrupt on it and handle it.

Whenever we send a **Trig** signal, we should enable the timer, and disable it on the first **Echo** pin state change. The code will be something like this.

```

1 volatile bool unavailable = false;
2
3 extern "C" void TIM6_DAC1_IRQHandler(void) {
4     ...
5
6     // Enable 7 interrupt
7     TIM7->DIER |= TIM_DIER_UIE;
8     NVIC_EnableIRQ(TIM7_DAC2_IRQn);
9     TIM7->CNT = 0;

```



```

10 }
11
12 extern "C" void EXTI15_10_IRQHandler(void)
13 {
14     TIM7->DIER &= ~TIM_DIER_UIE;
15     NVIC_DisableIRQ(TIM7_DAC2_IRQn);
16     unavailable = false;
17
18     ...
19 }
20
21 extern "C" void TIM7_DAC2_IRQHandler(void)
22 {
23     unavailable = true;
24     TIM7->CNT = 0;
25
26     // acknowledge
27     TIM7->SR &= ~TIM_SR_UIF;
28 }
29
30 int main(void) {
31     ...
32
33     while(1) {
34         Tft.erase();
35
36         if (unavailable) {
37             Tft.setCursor(1, 1);
38             Tft.println("Sensor unavailable");
39         } else {
40             ...
41         }
42     }
43 }

```

Listing 9: Sensor availability checker

## Resources

The code files, and this report's source code, are available on this GitHub repository: [elkhayder/sec1-tp-mac](#)