# Ultrasonic Theremin: A Digital Sound Synthesis Project

Exam Project Report: Digital Programmable Systems
Using STM32F401RE

Davide Marzolla

A.A. 2024/2025

## 1 Introduction

The exam project consists of the implementation of a Theremin using the STM32 F401RET6 microcontroller.

The Theremin is an electronic musical instrument, distinguished by the fact that it does not require physical contact from the performer. It was invented in 1919 by the physicist Lev Sergeyevich Termen and is based on oscillators that, operating in isofrequency, produce alterations in their characteristics as a result of the performer's hand movements within the wave field generated by two antennas.

The instrument is composed of two antennas placed, one on top and one on the side of a container that houses all the electronics: the upper antenna controls the pitch of the sound, while the side antenna controls the intensity of the sound.

The sound produced by the instrument results from the beat frequency between the waves emitted by the two antennas. Specifically, one antenna is connected to the oscillator's capacitor, and the movement of an obstacle near the antenna causes a variation in capacitance, thus altering the operating frequency.

This frequency *shift* generates the sound. It is worth noting that the capacitors have variable capacitance and are designed to produce a frequency *shift* between 20 and 20,000 Hz, which corresponds to the audible range of sound.

## 2 The Project

For my project, I used the following components:

- STM32 F401RET6 microcontroller
- Two breadboards as a base
- Jumper wires
- HC-SR04 ultrasonic distance sensor
- 8 Ohm 1 W 50 mm speaker

In this project, the pitch antenna of the traditional Theremin has been replaced by the ultrasonic distance sensor, while there is no volume control. The sound is produced by placing an object (or a hand) in front of the sensor, and the pitch of the sound changes depending on the distance between the sensor and the object (or hand).

## 3 Configuration

The steps for configuring the board were as follows:

- In **SYS** set **Serial Wire** inj the *Debug* field.
- Set Pin **PA10** as *GPIO_Output*, connected to the *Trig* port of the sensor.
- Set Pin **PA4** as *GPIO_Input*, connected to the *Echo* port of the sensor.
- Set Pin **PA1** as *TIM_CH2* connected to the speaker, from which the PWM signal is output.
- The internal timer clock is set to 72 MHz, and two timers were configured in the **Timers** section:
  - **TIM_1** with Clock Source : *Internal Clock* used for measuring the distance from the sensor.
  - **TIM_2** with Clock Source : *Internal Clock* and **PWM Generation CH2** on *Channel2*. This timer is specifically used for generating the PWM signal to be sent to the speaker and was configured as follows:
    - Prescaler = 260-1
    - Counter Mode = UP
    - ARR = 275
    - Internal Clock Division = No division
    - auto-reload preload = Enable

It is important to note that when auto-reload preload is enabled, the ARR value is updated only at the end of the current timer period. This means that any modifications to the ARR value will not take effect immediately but only upon the next counting cycle restart.

- In **Connectivity>USART_2** configured in **Asynchronous** mode at 115200 Bit/s.
- Pin **PA2** set as *USART2_TX* to transmit data via the serial interface. The distance and frequency data will be displayed in RealTerm.
- Pin **PA3** set as *USART_RX* to receive data from the serial interface.

# 4 Il Codice

The **read_distance()** function measures the distance using an ultrasonic sensor (such as the HC-SR04) and returns the calculated distance value in centimeters.

```
uint32_t read_distance(){

        // TRIG è ora settato high
        HAL_GPIO_WritePin(TRIG_PORT, TRIG_PIN, GPIO_PIN_SET);

        __HAL_TIM_SET_COUNTER(&htim1, 0);

        while (__HAL_TIM_GET_COUNTER (&htim1) < 10);  // Aspetta 10
        HAL_GPIO_WritePin(TRIG_PORT, TRIG_PIN, GPIO_PIN_RESET);

        pMillis = HAL_GetTick(); // Evitiamo un loop

        while (!(HAL_GPIO_ReadPin (ECHO_PORT, ECHO_PIN)) && pMillis + 10 >  HAL_GetTick());
        val1 = __HAL_TIM_GET_COUNTER (&htim1);

        pMillis = HAL_GetTick();
        while ((HAL_GPIO_ReadPin (ECHO_PORT, ECHO_PIN)) && pMillis + 50 > HAL_GetTick());
        val2 = __HAL_TIM_GET_COUNTER (&htim1);

        return (val2-val1)* 0.034/2;
        // sarebbe l'intervallo di tempo tra i due valori moltiplicato per la velocità del suono,
        // diviso 2 perchè il percorso viene fatto 2 volte
}
```

The ultrasonic sensor operates by sending a signal through the TRIG pin and measuring the time taken for the echo to return to the ECHO pin. The measured time is multiplied by the speed of sound (approximately 0.034 $cm/\mu s$) to obtain the distance. The value is then divided by 2 because the sound wave travels the distance twice (out and back).
The key steps in the code are:

- Set the TRIG pin to HIGH, signaling the sensor to start the measurement. Reset the htim1 timer counter, which will be used to measure the time.
- Wait 10 $\mu s$ while keeping TRIG HIGH (the sensor requires a pulse of at least 10 microseconds), then set TRIG to LOW to end the pulse.
- **pMillis = HAL_GetTick();** stores the current time (in milliseconds).
- Il while loop *waits for ECHO_PIN to become HIGH*, indicating that the return signal has been received.
- If the echo does not arrive within 10 ms, the loop breaks to prevent a freeze.
- **val1 = ___HAL_TIM_GET_COUNTER(&htim1);** stores the timer value when the echo starts.
- **pMillis = HAL_GetTick();** stores the current time.
- The while loop *waits for ECHO_PIN to go back to LOW*, marking the end of the echo.
- If the signal lasts more than 50 ms, the loop breaks (to avoid measurement errors if the sensor does not receive the echo).
- **val2 = ___HAL_TIM_GET_COUNTER(&htim1);** stores the timer value when the echo ends.

- **(val2 - val1)** gives the total travel time of the sound (in microseconds). Multiplying by 0.034 converts the time into distance (cm) using the speed of sound ( 340 $m/s = 0.034\ cm/\mu s$), and dividing by 2 accounts for the round-trip travel, giving the actual distance.

Subsequently, the distance value is converted into a frequency value. To achieve this, the **distance_to_frequency** function is used.

```
uint32_t distance_to_frequency(uint16_t d){

    if (frequency < freq_min) frequency = freq_min;
    if (frequency > freq_max) frequency = freq_max;

    if (d < distance_min) {
            d = distance_min;
        } else if (d > distance_max) {
            d = distance_max;
        }

        // Calcola il fattore di scala per aumentare la frequenza con la distanza
        const float scaling_factor = (float)(freq_max - freq_min) / (distance_max - distance_min);

        // Calcola la frequenza inversamente rispetto alla distanza
        return (uint16_t)(freq_max - scaling_factor * (distance_max - d));

}
```

The main task of the function is to apply the following formula:

$$f_{PWM} = f_{max} - \frac{f_{max} - f_{min}}{d_{max} - d_{min}} \ (d_{max} - d)$$

$$\text{Defining the scaling factor s:} \quad s = \frac{f_{max} - f_{min}}{d_{max} - d_{min}}$$

$$f_{PWM} = f_{max} - s \ (d_{max} - d)$$

This formula allows the conversion between distance and frequency, proportionally scaled to the maximum and minimum values of distance and frequency. These maximum and minimum values are stored in the variables $f_{max}, d_{max}, f_{min}$ and $d_{min}$ as follows:

```
// Definizione dei limiti della distanza
const int distance_min = 1;     // Minima distanza in cm
const int distance_max = 100;   // Massima distanza in cm

uint16_t frequency = 0;

// Definizione dei limiti delle frequenze
const float freq_min = 300;
const float freq_max = 800;
```

Finally, the **set_pwm_frequency()** function sets the frequency of the PWM that will be played through the speaker.

```
void set_pwm_frequency(TIM_HandleTypeDef *htim, uint16_t frequency) {

    uint32_t timer_clock = 72000000;  // Clock del timer a 72 MHz
    uint32_t psc_value = 259;

    // Calcola il valore del registro ARR (Auto-Reload Register)
    uint32_t arr_value = timer_clock / ( ( psc_value + 1 ) * frequency )  ;

    // Aggiorna il registro ARR del timer (Auto-Reload Register)
    __HAL_TIM_SET_AUTORELOAD(htim, arr_value);

    // Aggiorna il valore del CCR (50% duty cycle)
    TIM2->CCR1 = 50;

}
```

First, the *timer_clock* and the prescaler value *psc_value*are loaded into two variables. Then, the *arr_value* is calculated, which is modulated based on the frequency according to the formula:

$$ARR = \frac{f_{clock}}{(1 + PSC)\ f_{PWM}}$$

This value determines the sound of the speaker. Finally, the value is dynamically modified by **___HAL__TIM__SET__AUTORELOAD()** which updates the ARR.

Now, let's move on to the main:

```
while (1)
{

      distance = read_distance();

      if (distance < distance_min) {
          distance = distance_min;
      } else if (distance > distance_max) {
          distance = distance_max;
      }

      // Converte il valore di distanza in frequenza della PWM
      uint16_t frequency = distance_to_frequency(distance);

      // Imposta la frequenza della PWM del Channel 2 a quella desiderata
      set_pwm_frequency(&htim2, frequency);

      sprintf(MSG, "Distance: %lu cm, Freq: %lu Hz\r\n", distance, frequency);
      HAL_UART_Transmit(&huart2, (uint8_t *)MSG, strlen(MSG), HAL_MAX_DELAY);
      HAL_Delay(100);


  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */

}
```

First, the result of **read__distance** is stored in the **distance** variable, then an additional check is performed on distance to ensure it falls within the specified limits. Next, the result of distance__to__frequency is stored in the **frequency** variable, and this calculated frequency is used in the set__pwm__frequency function to calculate the PWM. Finally, to verify that the results and operations have been successful, the values of frequency and distance are transmitted via the serial interface, and they are displayed on RealTerm: