Mark	/11

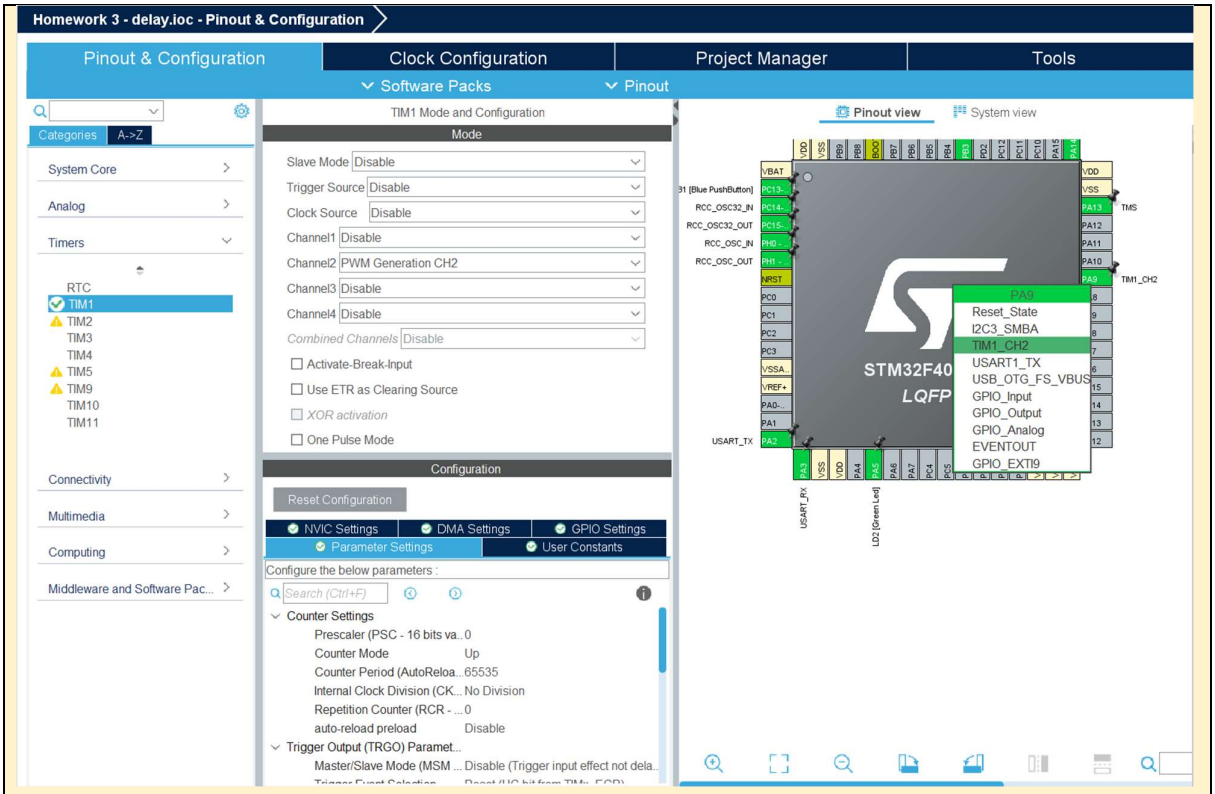
Team name:	*A14*		
Homework number:	*HOMEWORK 03*		
Due date:	5/10/25		
Contribution	NO	Partial	Full
Francesca Biondi			*x*
Lorenzo Castelli			*x*
Mattia Di Mauro			*x*
Pietro Albrigi			*x*
Notes: none			

Project name	Play a song		
Not done	Partially done (major problems)	Partially done (minor problems)	Completed
			x

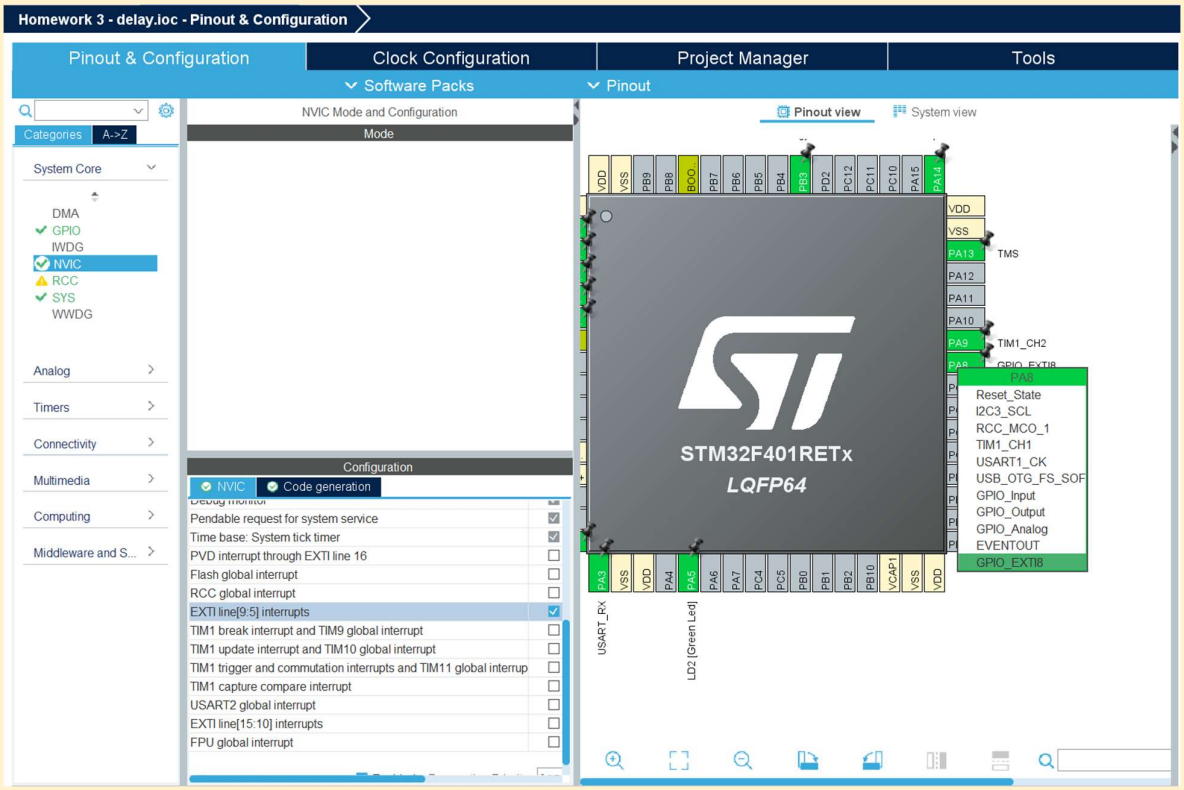
Part 1:

According to the "Hands-on Lab Schematics" document (page 3), the PWM_SPKR pin, which is the GPIO line connected to the speaker, is assigned to CN5 pin 1. Referring to the Nucleo board user manual, CN5 pin 1 maps directly to the PA9 pin of the STM32 microcontroller.

In STM32CubeIDE, we configured the speaker GPIO (PA9) to operate as TIM1 Channel 2 (PWM Generation CH2), enabling it to drive the speaker through a PWM signal:



We configured the microphone input pin (PA8) as GPIO_EXTI8, setting its mode to External Interrupt Mode with rising edge trigger detection, and enabled the EXTI interrupt in the NVIC section (EXTI lines [9:5] interrupts).



We ensured that the microphone interrupt was assigned a lower priority to prevent the interrupt from being triggered again by loud sounds from the song itself or by another snap occurring while the song is playing:

NVIC | Code generation

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
Non maskable interrupt	☑	0	0
Hard fault interrupt	☑	0	0
Memory management fault	☑	0	0
Pre-fetch fault, memory access fault	☑	0	0
Undefined instruction or illegal state	☑	0	0
System service call via SWI instruction	☑	0	0
Debug monitor	☑	0	0
Pendable request for system service	☑	0	0
Time base: System tick timer	☑	0	0
PVD interrupt through EXTI line 16	☐	0	0
Flash global interrupt	☐	0	0
RCC global interrupt	☐	0	0
EXTI line[9:5] interrupts	☑	1	0
TIM1 break interrupt and TIM9 global interrupt	☐	0	0
TIM1 update interrupt and TIM10 global interrupt	☐	0	0
TIM1 trigger and commutation interrupts and TIM11 global interrupt	☐	0	0
TIM1 capture compare interrupt	☐	0	0
USART2 global interrupt	☐	0	0

In the private define section, we assigned each note a name corresponding to its timer period value. Additionally, we defined a constant for the tempo, which specifies the minimum note duration (corresponding to a sixteenth note):

```
35 /* Private define ------------------------------------------------------------*/
36 /* USER CODE BEGIN PD */
37 #define DO 3205
38 #define DOD 3031
39 #define RE 2856
40 #define RED 2699
41 #define MI 2544
42 #define FA 2405
43 #define FAD 2269
44 #define SOL 2141
45 #define SOLD 2023
46 #define LA 1908
47 #define LAD 1801
48 #define SI 1699
49
50 #define Tempo 75
51 /* USER CODE END PD */
```

We then defined a struct to store the note period and duration and an array of structs (score) to represent the specific song melody:

```
28 /* USER CODE BEGIN PTD */
29 struct note {
30     uint16_t period;
31     uint16_t duration;
32 };
33 /* USER CODE END PTD */
```

```
/* USER CODE BEGIN PV */
struct note score[] = {
        {SOL,6},
        {LA,2},
        {SOL,4},
        {FA,4},
        {MI,4},
        {FA,4},
        {SOL,8},
        {RE,4},
        {MI,4},
        {FA,8},
        {MI,4},
        {FA,4},
        {SOL,8},
        {SOL,6},
        {LA,2},
        {SOL,4},
        {FA,4},
        {MI,4},
        {FA,4},
        {SOL,8},
        {RE,8},
        {SOL,8},
        {MI,4},
        {DO,12},
};
/* USER CODE END PV */
```

We then defined the functions playnote and playsong as follows:

```c
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
void playnote(struct note note_playing){
      /* USER CODE BEGIN TIM1_Init 0 */

      /* USER CODE END TIM1_Init 0 */

      TIM_MasterConfigTypeDef sMasterConfig = {0};
      TIM_OC_InitTypeDef sConfigOC = {0};
      TIM_BreakDeadTimeConfigTypeDef sBreakDeadTimeConfig = {0};

      /* USER CODE BEGIN TIM1_Init 1 */

      /* USER CODE END TIM1_Init 1 */
      htim1.Instance = TIM1;
      htim1.Init.Prescaler = 99;
      htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
      htim1.Init.Period = note_playing.period;
      htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
      htim1.Init.RepetitionCounter = 0;
      htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
      if (HAL_TIM_PWM_Init(&htim1) != HAL_OK)
      {
        Error_Handler();
      }
      sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
      sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
      if (HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig) != HAL_OK)
      {
        Error_Handler();
      }
      sConfigOC.OCMode = TIM_OCMODE_PWM1;
      sConfigOC.Pulse = 0.5*(note_playing.period+1) -1;
      sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
      sConfigOC.OCNPolarity = TIM_OCNPOLARITY_HIGH;
      sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
      sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;
      sConfigOC.OCNIdleState = TIM_OCNIDLESTATE_RESET;
      if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_2) != HAL_OK)
      {
        Error_Handler();
      }
      sBreakDeadTimeConfig.OffStateRunMode = TIM_OSSR_DISABLE;
      sBreakDeadTimeConfig.OffStateIDLEMode = TIM_OSSI_DISABLE;
      sBreakDeadTimeConfig.LockLevel = TIM_LOCKLEVEL_OFF;
      sBreakDeadTimeConfig.DeadTime = 0;
      sBreakDeadTimeConfig.BreakState = TIM_BREAK_DISABLE;
      sBreakDeadTimeConfig.BreakPolarity = TIM_BREAKPOLARITY_HIGH;
      sBreakDeadTimeConfig.AutomaticOutput = TIM_AUTOMATICOUTPUT_DISABLE;
      if (HAL_TIMEx_ConfigBreakDeadTime(&htim1, &sBreakDeadTimeConfig) != HAL_OK)
      {
        Error_Handler();
      }
      /* USER CODE BEGIN TIM1_Init 2 */

      /* USER CODE END TIM1_Init 2 */
      HAL_TIM_MspPostInit(&htim1);
      HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_2);
      HAL_Delay(note_playing.duration*Tempo);
      HAL_TIM_PWM_Stop(&htim1, TIM_CHANNEL_2);
}
void playsong(){
    int length = sizeof(score)/sizeof(score[0]);
           for (int i= 0; i< length; i++){
               playnote(score[i]);
           }
}
```

The playnote() function was implemented by copying the code from the MX_TIM1_Init function modifying it to dinamically reconfigure Timer 1 for every element of the score. Specifically, we modified the period (line 120) and pulse (line 135) parameters to adapt to the values of the note to be played.
For each note, the pulse (CCRx) is recalculated according to the period, while mantaining a duty cycle of 50%:

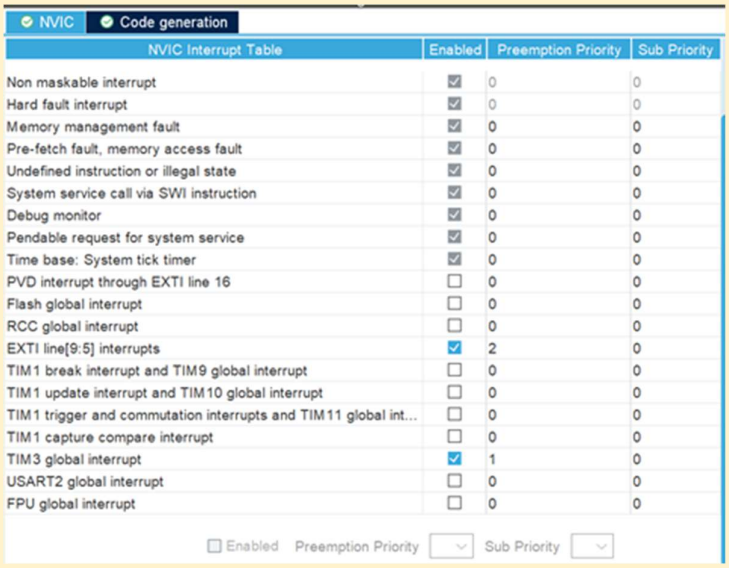
PWM Duty Cycle: $$DC = \frac{CCRx+1}{ARR+1}$$

At the end of the function, the PWM signal is started and then stopped after a delay proportional to the desired note duration.
Lastly, we implemented the EXTI callback associated with the microphone, with debouncing (tick, last_tick and delta were previously defined as global variables) and clearing of the interrupt flag:

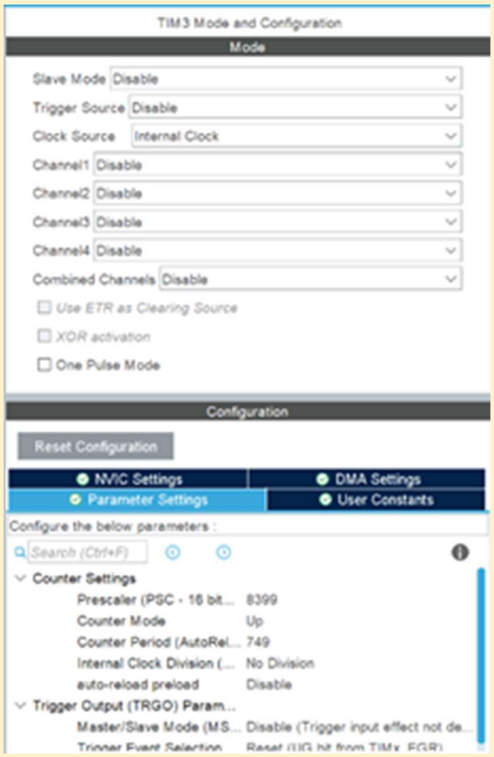
```
170 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin){
171     if(GPIO_Pin==GPIO_PIN_8){
172         __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);
173         tick = HAL_GetTick();
174         delta = tick - last_tick;
175         if(delta>=100){
176             playsong();
177         }
178         last_tick = tick;
179     }
180 }
181 /* USER CODE END 0 */
```

We executed the program on the board, and it functioned as intended.

Part 2:



In Part 2 of the assignment, we based our work on the configuration used for the previous part but removed all blocking delay mechanisms and replaced them with a non-blocking, interrupt-based approach using Timer 3.



Timer 3 was configured to generate periodic interrupts, corresponding to the base tempo unit, we defined its prescaler and ARR to do so and it uses the Internal clock as source. This timer now serves as

the core timebase for counting the duration of each note without halting the main loop or blocking the processor.

```
68  uint32_t counter = 0;
69  uint32_t current_index = 0;
70  uint8_t song_playing = 0;
```

A global *counter* variable is incremented on each Timer 3 interrupt, and once it exceeds the duration field of the currently playing note, the PWM output is stopped and the next note in the *score* array is triggered.

```
172 void playsong(){
173     if (song_playing) return;
174     song_playing = 1;
175     current_index = 0;
176     playnote(score[current_index]);
177     HAL_TIM_Base_Start_IT(&htim3);
178 }
179 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin){
180     if(GPIO_Pin==GPIO_PIN_8){
181         __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);
182         tick = HAL_GetTick();
183         delta = tick - last_tick;
184         if(delta>=100 && !(song_playing)){
185             playsong();
186         }
187         last_tick = tick;
188     }
189 }
```

Even if we modified playsong() and HAL_GPIO_EXTI_Callback() this entire logic is managed inside the HAL_TIM_PeriodElapsedCallback() function, which performs the following steps:

- Stops the PWM output if the note duration has elapsed.
- Advances to the next note in the sequence using playnote.
- Stops Timer 3 when the end of the melody is reached.

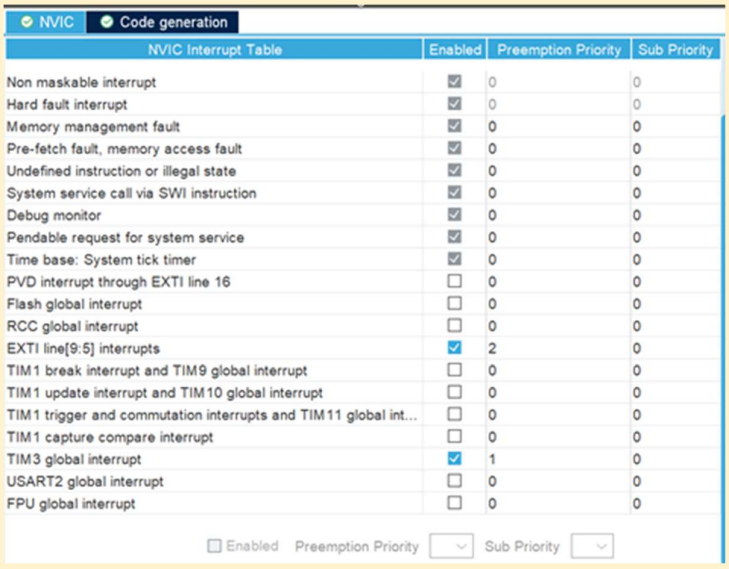
This allows continuous playback of the melody in real time without blocking the main thread.

```c
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
    if(htim == &htim3){
        counter++;
        if (counter > current_note.duration) {
            HAL_TIM_PWM_Stop(&htim1, TIM_CHANNEL_2);
            counter = 0;
            current_index ++;
            if(current_index < sizeof(score)/sizeof(score[0])){
                playnote(score[current_index]);
            }else {
                HAL_TIM_Base_Stop_IT(&htim3);
                song_playing = 0;
            }
        }
    }
}
```

To avoid re-triggering the song while it is still playing, a flag *song_playing* was introduced. The playback sequence can only be initiated from the external interrupt on pin PA8 if this flag is cleared.

Interrupt management played a crucial role in the success of the project. Timer 3 was assigned the priority 1 to guarantee a precise timing of note durations, while the GPIO external interrupt on PA8 was configured with priority 2 to prevented unintended feedback between the speaker and the microphone.

Additionally, the same debounce mechanism based on the HAL_GetTick() function prevents false triggering from rapid successive sound events as in the previous part of the homework.



The code was then tested on the board and confirmed to work as intended.

Professor comments: