Embedded Systems Take-Home Final Exam

Section 1: GPIO

To configure a GPIO pin on the STM32F107 as input or output, we follow these steps:

Step 1: Enable the GPIO Clock
First, we need to turn on the clock for the GPIO port we will use. This is done by enabling the appropriate GPIO port in the RCC_APB2ENR register. If the clock is not enabled, the GPIO pin won't work.

Step 2: Set the Pin Mode
Next, we configure the pin mode. For this, we use the GPIOx_CRL (for pins 0–7) or GPIOx_CRH (for pins 8–15) registers. Each pin takes 4 bits to set its mode and configuration.

For output, you choose the mode as General Purpose Output or Alternate Function Output. You also need to select the speed (2 MHz, 10 MHz, or 50 MHz).
For input, you set the mode to Input and choose the type (Analog, Floating, Pull-up, or Pull-down).
Step 3: Set the Output Type (if output)
If the pin is configured as output, decide whether it should be Push-Pull or Open-Drain. Push-Pull means the pin can drive high or low voltage, while Open-Drain means it can only pull the line low, and external components pull it high.

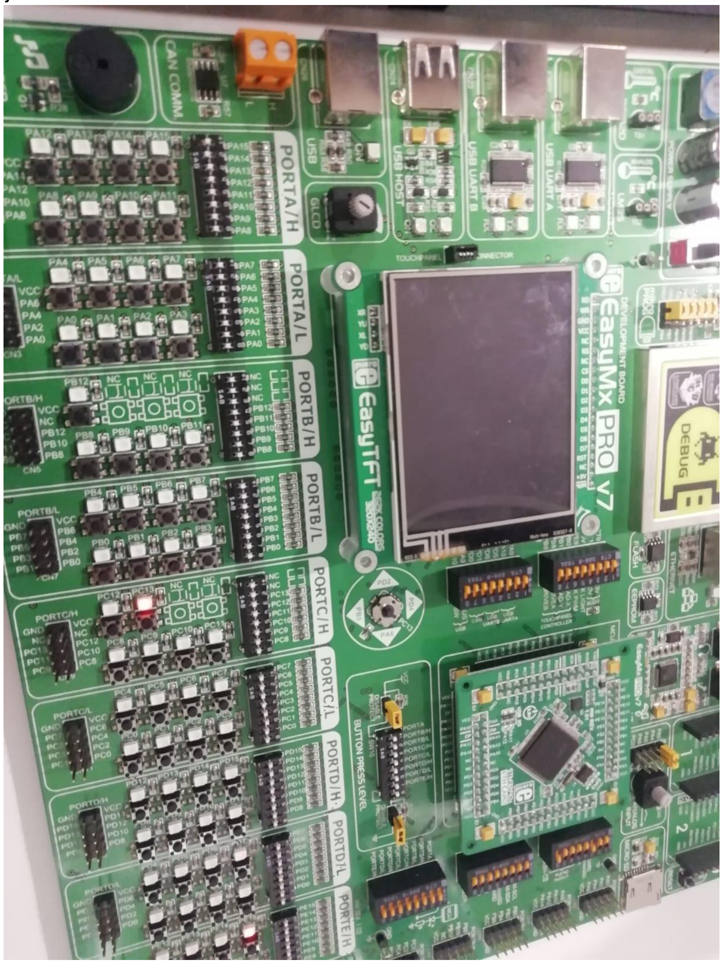
Step 4: Pull-up/Pull-down Configuration
For input pins, if the mode is set to Pull-up/Pull-down, you need to configure the pin's state using the GPIOx_ODR register.

Set the corresponding bit to enable Pull-up.
Clear it to enable Pull-down.
Why is Pull-up/Pull-down Necessary for Input Pins?
Input pins without pull-up or pull-down resistors can "float." This means they don't have a defined voltage level when no signal is connected, and their value might change randomly due to noise. Pull-up and pull-down resistors solve this by making sure the pin has a steady voltage level:

Pull-up connects the pin to a high voltage (logic 1).
Pull-down connects it to a low voltage (logic 0).
This configuration avoids unwanted behavior and ensures the microcontroller reads the correct value when no external signal is applied.

```
#include "stm32f10x.h"  // Device header

void init_GPIOC(void) {
   RCC->APB2ENR |= RCC_APB2ENR_IOPCEN;  // Enable clock for GPIOC

   GPIOC->CRH &= ~(GPIO_CRH_CNF13);     // Clear PC13 configuration bits (input mode)
   GPIOC->CRH |= GPIO_CRH_MODE13_1;     // Set PC13 as output mode, max speed 2 MHz
}
```

```
void delay_ms(uint32_t delay) {
    for (uint32_t i = 0; i < delay * 8000; i++) {
        __asm("nop");  // No operation, simple delay loop
    }
}

void toggle_LED(int state) {
    if (state) {
        GPIOC->ODR &= ~(1UL << 13);  // Set PC13 low (turn LED on, active low)
    } else {
        GPIOC->ODR  |= (1UL << 13);   // Set PC13 high (turn LED off, active low)
    }
}

int main() {
    init_GPIOC();  // Initialize GPIOC pin 13

    while (1) {
        toggle_LED(1);    // Turn LED on (active low)
        delay_ms(500);   // 1-second delay
        toggle_LED(0);    // Turn LED off (active low)
        delay_ms(500);   // 1-second delay
    }
}
```

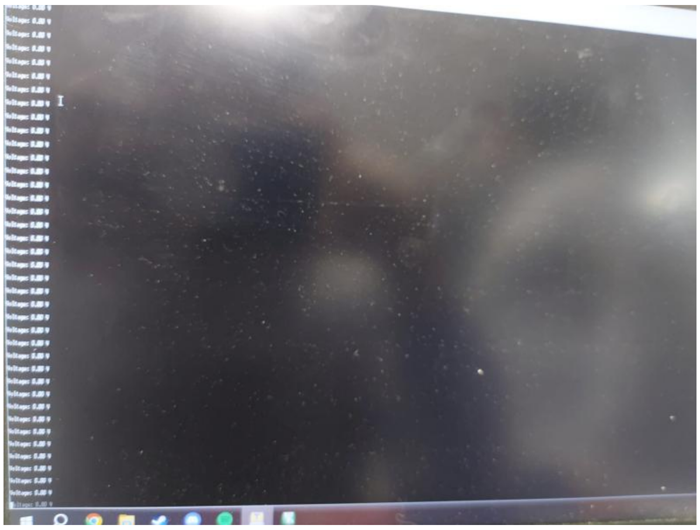


Section 2: ADC

ADC (Analog-Digital Converter) resolution determines the number of digital values representing an analog input signal, and this depends on the number of bits used. Basically, resolution determines the smallest analog signal change that the ADC can detect. Higher resolution allows better discrimination of different input signal levels and enables more precise digital representation of the analog signal.

```
#include "stm32f10x.h" // Include CMSIS library for STM32F1
#include <stdio.h>     // Include for sprintf

void SystemClock_Config(void);
void GPIO_Init(void);
void ADC1_Init(void);
void TIM3_Init(void);
void UART_Init(void);
void delay_ms(uint32_t ms);
void UART_SendString(const char *str);

int main(void) {
   SystemClock_Config(); // Configure System Clock
   GPIO_Init();          // Initialize GPIO
   ADC1_Init();          // Initialize ADC
   UART_Init();          // Initialize UART

   // Start ADC Conversion
   ADC1->CR2 |= ADC_CR2_ADON;      // Enable ADC
   ADC1->CR2 |= ADC_CR2_SWSTART;   // Start ADC conversion

   char buffer[50]; // Buffer for UART output
   while (1) {
      // Wait for ADC conversion to complete
      while (!(ADC1->SR & ADC_SR_EOC)); // Wait for End of Conversion

      uint16_t adcValue = ADC1->DR; // Read ADC value

      // Convert ADC value to voltage (0-3.3V)
      float voltage = (adcValue * 3.3f) / 4095.0f;

      // Format the voltage as a string
      sprintf(buffer, "Voltage: %.2f V\r\n", voltage);

      // Send the voltage over UART
      UART_SendString(buffer);

      delay_ms(500); // Delay for readability
   }
}

/* System Clock Configuration */
void SystemClock_Config(void) {
   RCC->CR |= RCC_CR_HSION; // Enable HSI
   while (!(RCC->CR & RCC_CR_HSIRDY)); // Wait until HSI is ready
```

```
    RCC->CFGR &= ~RCC_CFGR_SW;      // Clear SW bits
    RCC->CFGR |= RCC_CFGR_SW_HSI;   // Select HSI as system clock
    while ((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_HSI); // Wait for HSI
}

/* GPIO Initialization */
void GPIO_Init(void) {
    // Enable GPIOC clock
    RCC->APB2ENR |= RCC_APB2ENR_IOPCEN;

    // Configure PC13 as output (LED)
    GPIOC->CRH &= ~GPIO_CRH_MODE13;
    GPIOC->CRH |= GPIO_CRH_MODE13_1; // Output mode, max speed 2 MHz
    GPIOC->CRH &= ~GPIO_CRH_CNF13;   // General-purpose output push-pull
}

/* ADC Initialization */
void ADC1_Init(void) {
    // Enable ADC1 and GPIOA clock
    RCC->APB2ENR |= RCC_APB2ENR_ADC1EN | RCC_APB2ENR_IOPAEN;

    // Configure PA0 as analog input
    GPIOA->CRL &= ~GPIO_CRL_MODE0;
    GPIOA->CRL &= ~GPIO_CRL_CNF0;

    // ADC configuration
    ADC1->CR2 |= ADC_CR2_ADON;      // Enable ADC
    ADC1->SMPR2 |= ADC_SMPR2_SMP0;   // Sampling time for channel 0
}

/* UART Initialization */
void UART_Init(void) {
    // Enable USART1 and GPIOA clock
    RCC->APB2ENR |= RCC_APB2ENR_USART1EN | RCC_APB2ENR_IOPAEN;

    // Configure PA9 (TX) as alternate function push-pull
    GPIOA->CRH &= ~GPIO_CRH_MODE9;
    GPIOA->CRH |= GPIO_CRH_MODE9_1;  // Output mode, max speed 2 MHz
    GPIOA->CRH &= ~GPIO_CRH_CNF9;
    GPIOA->CRH |= GPIO_CRH_CNF9_1;   // Alternate function push-pull

    // Configure PA10 (RX) as input floating
    GPIOA->CRH &= ~GPIO_CRH_MODE10;
    GPIOA->CRH &= ~GPIO_CRH_CNF10;
    GPIOA->CRH |= GPIO_CRH_CNF10_0;  // Input floating

    // Configure USART1
    USART1->BRR = 0x1D4C; // Baud rate 9600 (assuming 72 MHz clock)
    USART1->CR1 |= USART_CR1_UE;    // Enable USART
    USART1->CR1 |= USART_CR1_TE;    // Enable transmitter
}

/* UART Send String */
void UART_SendString(const char *str) {
    while (*str) {
        while (!(USART1->SR & USART_SR_TXE)); // Wait until TXE is set
        USART1->DR = *str++;                // Send character
    }
}

/* Simple delay function */
void delay_ms(uint32_t ms) {
    for (uint32_t i = 0; i < ms * 8000; i++) {
        __NOP();
    }
}
```

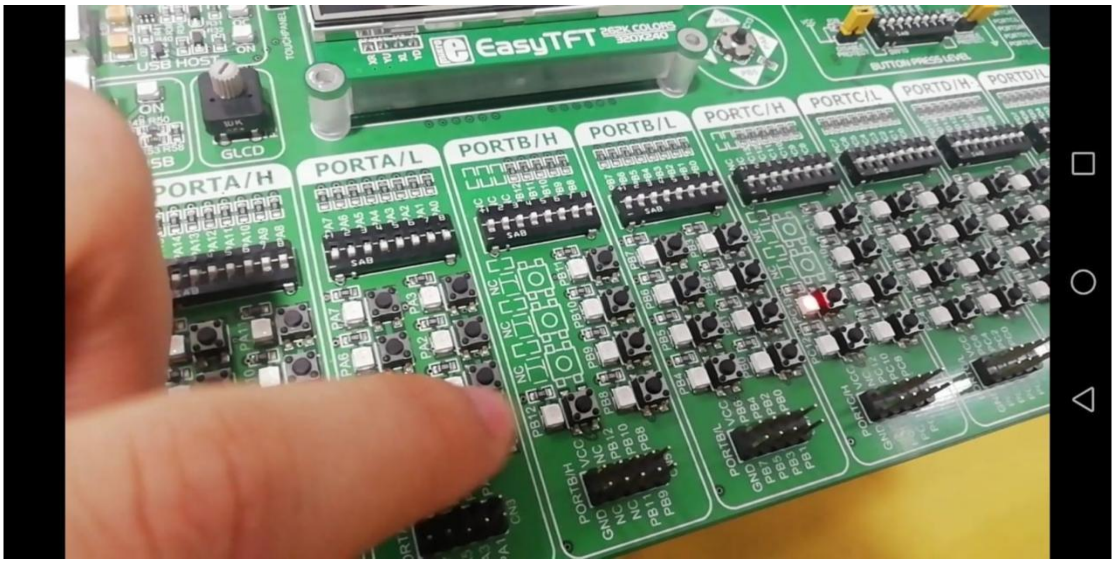


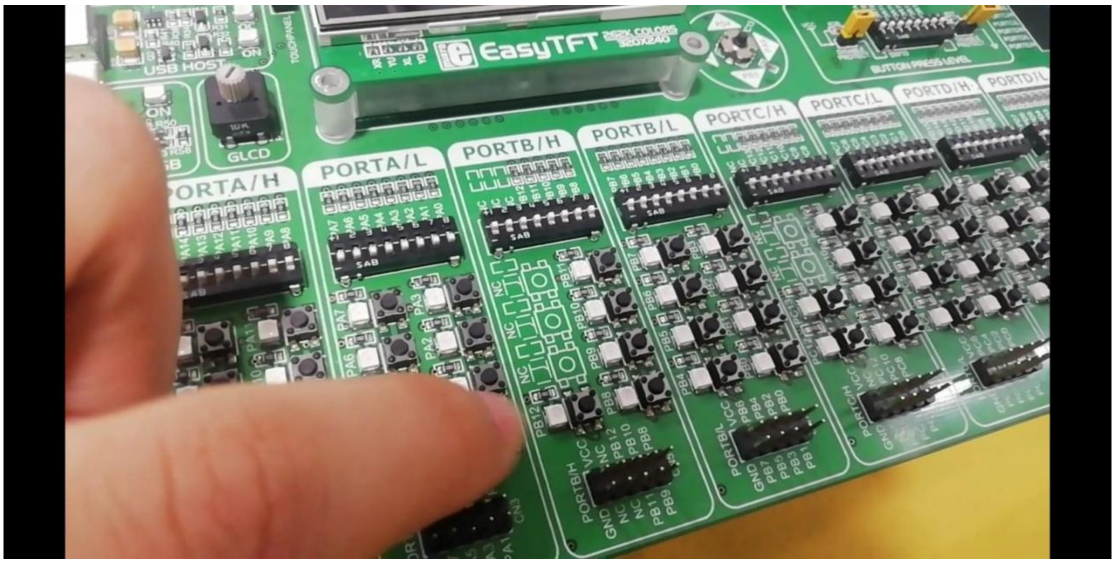
```
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
Voltage: 0.00 V
```

Section 3: Interrupts

To configuring external interrupts first the AFIO Clock must be enabled by using RCC_APB2ENR register to access the external interrupt. Then, the GPIO pin that is connected to the external interrupt is configured as an input. To configure the GPIO pin connected to the external interrupt as input, GPIOx_CRL or GPIOx_CRH register are used to set the pin as a pull-up/pull-down resistor. The EXTI controller which maps GPIO pins to interrupt lines is configured. To configure it, AFIO_EXTICR register is used to assign the GPIO pin to the correct EXTI line. Then, the interrupt for that EXTI line is activated using the EXTI_IMR register. After that, we specify the trigger type either rising edge or falling edge with EXTI_RTSR or EXTI_FTSR. Then, the interrupt for the corresponding EXTI line is enabled in the NVIC. The ISR implemented to manage the generated interrupt, whenever the interrupt is triggered the task is executed. If interrupts are used when pressing the button, the bounce can trigger multiple interrupts in a single press. This may causes ISR logic to re-run which creates undesirable behavior. To prevent issues caused by bouncing, debounce mechanisms is used. They filter out the noise and ensure only one clean signal is processed per press.

```
#include "stm32f10x.h"  // Device header

void init_GPIOC(void) {
    RCC->APB2ENR |= RCC_APB2ENR_IOPCEN;  // Enable clock for GPIOC
    GPIOC->CRH &= ~(GPIO_CRH_CNF13);     // Clear PC13 configuration bits (input mode)
    GPIOC->CRH |= GPIO_CRH_MODE13_1;     // Set PC13 as output mode, max speed 2
MHz
}

void delay_ms(uint32_t delay) {
    for (uint32_t i = 0; i < delay * 4000; i++) {
        __asm("nop");  // No operation, simple delay loop
    }
}

void toggle_LED(int state) {
    if (state) {
        GPIOC->ODR &= ~(1UL << 13);  // Set PC13 low (turn LED on, active low)
    } else {
        GPIOC->ODR |= (1UL << 13);   // Set PC13 high (turn LED off, active low)
    }
}

int main() {
    init_GPIOC();  // Initialize GPIOC pin 13

    while (1) {
        toggle_LED(1);    // Turn LED on (active low)
        delay_ms(1000);   // 1-second delay
        toggle_LED(0);    // Turn LED off (active low)
        delay_ms(1000);   // 1-second delay
    }
}
```



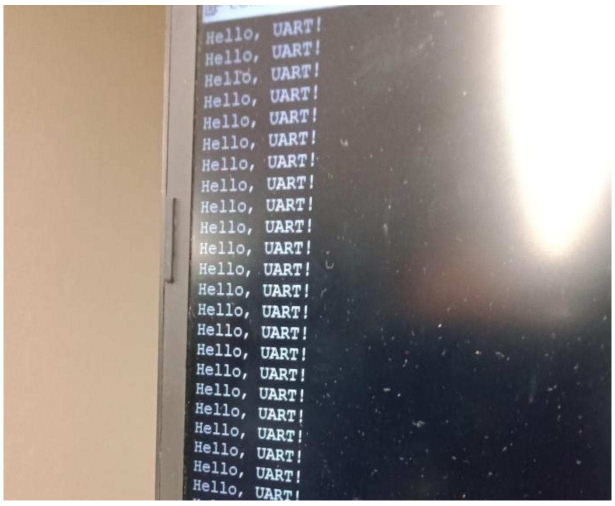


Section 4: UART

First, the clocks for both the UART peripheral and GPIO pins used for transmit (TX) and receive (RX) must be enabled using RCC (Reset and Clock Control) registers. Then the TX pin (PA9) is set to Push-Pull mode, while the RX pin (PA10) is configured as a Pull-Up. After that, the baud rate is set by the USART Baud Rate Register (USART_BRR) based on the peripheral clock (PCLK). The transmitter and receiver is activated by setting the TE and RE bits in the USART_CR1 register. The baud rate defines the data transfer rate in bits per second, correctly configuring the baud rate is critical since it ensures the communication speed matches between devices. Not correctly configuring the baud rate may result in data corruption. The UART peripheral clock (PCLK) sets the base frequency for baud rate calculation. Changes in the system clock (like setting another clock such as HSE, PLL) affect PCLK and require baud rate updates.

```
#include "stm32f10x.h"
#include <stdint.h>

#define BAUD_RATE 9600 // Define the baud rate
#define SYSCLK 8000000 // System clock frequency (8 MHz)

void initSysTick(uint32_t ticks);
void delayMicroseconds(uint32_t value);
void UART_SendString(const char *message);

int main() {
   // Initialize SysTick for delays
   initSysTick(SYSCLK / 8 / 1000000); // Configure SysTick for 1-microsecond ticks

   // Simulate UART transmission
   UART_SendString("Hello, UART!");

   while (1) {
      // Infinite loop
   }
}
```

```
void initSysTick(uint32_t ticks) {
   SysTick->CTRL = 0;              // Disable SysTick
   SysTick->LOAD = ticks - 1;       // Set reload register
   SysTick->VAL = 0;               // Reset the current value register
   SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_ENABLE_Msk; // Enable SysTick
}

void delayMicroseconds(uint32_t value) {
   while (value > 0) {
      while ((SysTick->CTRL & SysTick_CTRL_COUNTFLAG_Msk) == 0) {
         // Wait for the COUNTFLAG to be set
      }
      value--;
   }
}

void UART_SendString(const char *message) {
   // Calculate the delay per character based on the baud rate
   uint32_t charDelay = (1000000 * 10) / BAUD_RATE; // 10 bits (1 start, 8 data, 1 stop)

   while (*message) {
      // Simulate sending a character
      putchar(*message); // Use putchar to send the character (could replace with GPIO/USART
hardware)
      fflush(stdout);    // Ensure the character is printed immediately
      delayMicroseconds(charDelay); // Simulated delay for UART
      message++;          // Move to the next character
   }
}
```



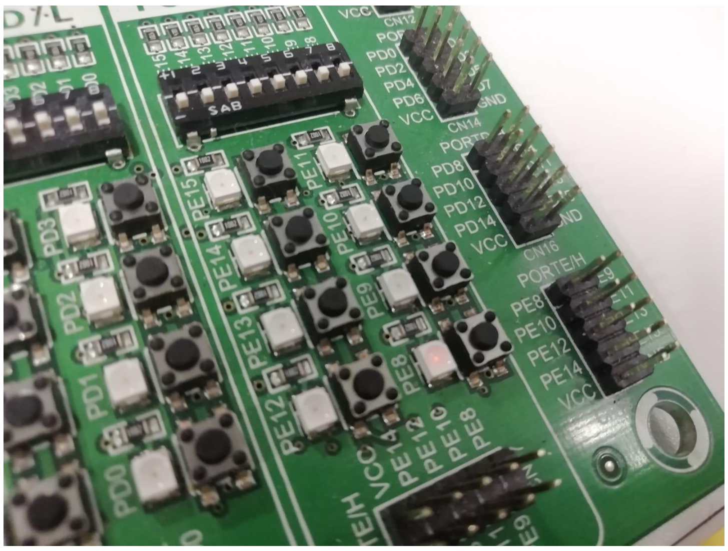
```
Hello, UART!
Hello, UART!
Hello, UART!
Hello, UART!
Hello, UART!
Hello, UART!
Hello, UART!
Hello, UART!
Hello, UART!
Hello, UART!
Hello, UART!
Hello, UART!
Hello, UART!
Hello, UART!
Hello, UART!
Hello, UART!
Hello, UART!
Hello, UART!
Hello, UART!
Hello, UART!
Hello, UART!
Hello, UART!
Hello, UART!
```

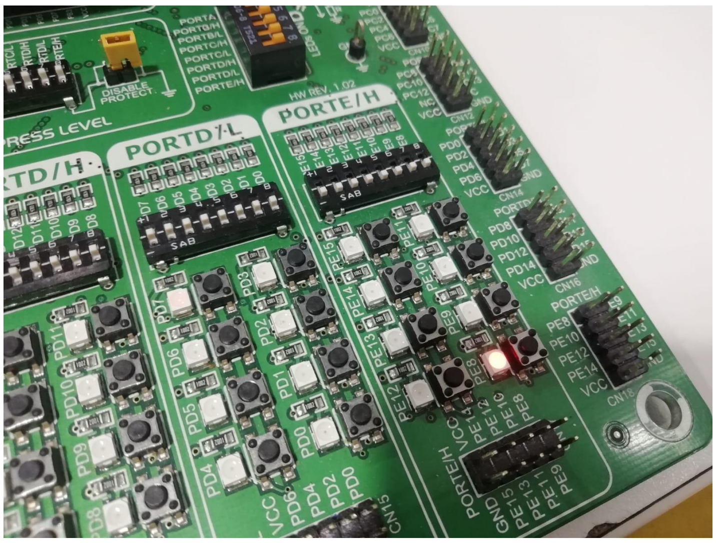
Section 5: PWM

PWM is used to generate an analog signal with a digital source. By turning the digital signal ON and OFF at sufficient speed and at a given duty cycle, the output appears as a constant voltage analog signal. It is used in embedded systems to control devices by changing the duty cycle of a periodic signal. PWM is implemented by using its built-in general-purpose timers.

To use a timer for PWM, its clock and the clock for the corresponding GPIO pin need to be enabled with RCC registers. Then configure the GPIO pin set as an Alternate Function Push-Pull output which will generate PWM signal. The timer is adjusted to generate PWM signals by adjusting the prescaler, auto-reload register (ARR), and capture/compare register (CCR). The prescaler timer's clock frequency by the system clock frequency, while ARR determines the PWM period and CCR determines the duty cycle of the PWM signal. Use the Capture/Compare Mode Registers (CCMR) and the Capture/Compare Enable Register (CCER) to configure the timer channel for PWM mode.After that, start the timer by enabling its counter. To change the duty cycle during runtime, simply update the CCR value for the desired channel

```
#include "stm32f10x.h"

void initClockPLL(void);
void initClockHSI(void);
void initClockHSE(void);
void delayMicroseconds(uint32_t value);
void initTIM1(void);
void initSysTick(uint32_t value);
void initGPIO(void);

int main() {
   // Initialize the clock to use PLL as the system clock
   initClockHSI();
   initSysTick(800);
   initTIM1();
   initGPIO();

   // Duty Cycle
   TIM1->CCR1 = 1;
}

// Function to introduce a delay
void delayMicroseconds(uint32_t value) {
   while (value > 0) {
      while ((SysTick->CTRL & SysTick_CTRL_COUNTFLAG_Msk) != SysTick_CTRL_COUNTFLAG); // Wait
until COUNTFLAG is 1
      value--;
   }
}
/**
 * @brief Initializes the clock to use 8MHz HSI as the system clock.
 */
void initClockHSI(void) {
   RCC->CR |= RCC_CR_HSION; // Enable internal HSI (RC)

   while ((RCC->CR & RCC_CR_HSIRDY) != RCC_CR_HSIRDY); // Wait for HSI to be ready

   RCC->CFGR &= ~(RCC_CFGR_SW); // Clear SW bits
   RCC->CFGR |= RCC_CFGR_SW_HSI; // Set HSI as the system clock

   while ((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_HSI); // Wait for HSI to be the system
clock

                        RCC->APB2ENR |= RCC_APB2ENR_TIM1EN; // Enable TIM1 clock
   RCC->APB2ENR |= RCC_APB2ENR_AFIOEN; // Enable alternate function clock
}

void initTIM1(void) {
   // TIM1 configuration (PWM)
   TIM1->PSC = 8 - 1;    // Set prescaler to achieve a 1MHz timer clock
   TIM1->ARR = 100;      // Set auto-reload value for a 100us period

   TIM1->CCMR1 |= TIM_CCMR1_OC1M_2 | TIM_CCMR1_OC1M_1 | TIM_CCMR1_OC1PE; // PWM
mode 1, enable preload
   TIM1->CCER |= TIM_CCER_CC1NE; // Enable the capture/compare channel 1 complementary
output

   TIM1->BDTR |= TIM_BDTR_MOE; // Enable the main output

   TIM1->CR1 |= TIM_CR1_CMS_0 | TIM_CR1_CEN; // Enable TIM1 in center-aligned mode
   TIM1->EGR |= TIM_EGR_UG; // Force update generation
}

void initSysTick(uint32_t value) {
   SysTick->CTRL &= ~SysTick_CTRL_ENABLE_Msk; // Disable SysTick

   SysTick->LOAD = value; // Set reload value
   SysTick->VAL = 0;      // Clear current value
   SysTick->CTRL &= ~SysTick_CTRL_CLKSOURCE_Msk; // Configure clock source AHB/8

   SysTick->CTRL &= ~SysTick_CTRL_TICKINT_Msk; // Disable interrupt

   SysTick->CTRL |= SysTick_CTRL_ENABLE; // Enable SysTick
}

void initGPIO(void) {
   // GPIO configuration
   AFIO->MAPR |= AFIO_MAPR_TIM1_REMAP; // Full remap
   RCC->APB2ENR |= RCC_APB2ENR_IOPEEN; // Enable Port E clock
                      RCC->APB2ENR |= RCC_APB2ENR_IOPDEN;

   // Configure PE8 for TIMCH1N
   GPIOE->CRH |= GPIO_CRH_MODE8; // Output mode
   GPIOE->CRH &= ~(GPIO_CRH_CNF8); // Alternate output push-pull
   GPIOE->CRH |= GPIO_CRH_CNF8_1;
}
```





Section 6: Timers

Timers on the STM32F107 are used to generate periodic events like blinking an LED. First, you need to enable the timer's clock in the RCC register. After that, you set the prescaler, which slows down the timer clock to a useful speed. For example, if the system clock is 25 MHz, setting the prescaler to 2500 will make the timer count every 0.1ms. Then, you set the auto-reload value, which decides when the timer resets and triggers an event. To start the timer, you set the CEN bit in the TIMx_CR1 register, and the timer starts counting. To use the timer for events, you can either enable interrupts, which will trigger an interrupt when the timer reaches the reload value, or you can check the status register to see if the event has occurred (this is called polling). Both ways let you handle events like toggling an LED or triggering other actions. If you set the wrong values, the timer may not work as expected, so it's important to check your configuration carefully.

```
#include "stm32f10x.h"

void initClockHSI(void);
void initTIM1(void);
void initGPIO(void);

int main() {
   // Initialize the system clock and peripherals
   initClockHSI();
   initTIM1();
   initGPIO();

   // Enable the timer to start generating the square wave
   TIM1->CR1 |= TIM_CR1_CEN; // Enable TIM1
}

void initClockHSI(void) {
   RCC->CR |= RCC_CR_HSION; // Enable internal HSI (RC)

   while ((RCC->CR & RCC_CR_HSIRDY) != RCC_CR_HSIRDY); // Wait for HSI to be ready

   RCC->CFGR &= ~(RCC_CFGR_SW); // Clear SW bits
   RCC->CFGR |= RCC_CFGR_SW_HSI; // Set HSI as the system clock
```

```
    while ((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_HSI); // Wait for HSI to be the system
clock

    RCC->APB2ENR |= RCC_APB2ENR_TIM1EN; // Enable TIM1 clock
    RCC->APB2ENR |= RCC_APB2ENR_AFIOEN; // Enable alternate function clock
}

void initTIM1(void) {
    // Set the timer prescaler to achieve a timer frequency of 1 MHz (1 us per tick)
    TIM1->PSC = 8 - 1;    // Prescaler value (assuming HSI clock is 8 MHz)

    // Set the auto-reload value to generate a 1 kHz signal (1 ms period)
    TIM1->ARR = 1000 - 1; // 1 kHz frequency (period = 1000 ticks at 1 MHz)

    // Configure the timer in PWM mode (toggle output on compare match)
    TIM1->CCMR1 |= TIM_CCMR1_OC1M_2 | TIM_CCMR1_OC1M_1 | TIM_CCMR1_OC1PE; // PWM
mode 1, preload enable
    TIM1->CCER |= TIM_CCER_CC1E; // Enable the capture/compare channel 1 output

    TIM1->BDTR |= TIM_BDTR_MOE; // Enable the main output
}

void initGPIO(void) {
    // GPIO configuration for PE8 (connected to TIM1 CH1)
    RCC->APB2ENR |= RCC_APB2ENR_IOPEEN; // Enable Port E clock

    // Configure PE8 as an alternate function output (PWM output)
    GPIOE->CRH &= ~GPIO_CRH_CNF8; // Reset CNF8
    GPIOE->CRH |= GPIO_CRH_CNF8_1; // Alternate function push-pull
    GPIOE->CRH |= GPIO_CRH_MODE8;  // Output mode (maximum speed 50 MHz)
}
```