Submitted BY : Sumit Santosh Ghule
Batch : 1 MAY 2025

Introduction

The blinking LED program is often considered the "Hello World" of embedded systems. It introduces beginners to the basics of microcontroller programming by using a simple GPIO output pin to control an LED. In this task, the STM32 microcontroller is programmed using STM32CubeIDE and the HAL library to toggle an LED at a fixed interval. This exercise demonstrates the fundamental process of GPIO configuration, code implementation, and hardware verification, laying the foundation for more complex embedded applications.

Objective

- To configure a GPIO pin as an output using STM32CubeIDE.
- To toggle the state of an LED (ON/OFF) at regular time intervals.
- To understand the use of HAL functions such as HAL_GPIO_TogglePin() and HAL_Delay().

Methodology / Approach

1. **Project Creation:** A new project was created in STM32CubeIDE with the appropriate microcontroller (STM32F103C8T6).
2. **Pin Configuration:**
   - The onboard LED pin (PC13 for BluePill, LD2 for Nucleo board) was configured as **GPIO Output**.
   - GPIO mode was set to **Output Push-Pull** with no pull-up/pull-down resistors.
3. **Code Implementation:**
   - The HAL_GPIO_TogglePin() function was used to change the LED state.
   - The HAL_Delay() function was inserted to control the blink frequency.
4. **Flashing and Testing:** The program was compiled, debugged, and flashed to the STM32 board using ST-Link.

Code Documentation

```
/* USER CODE BEGIN 2 */
// Initialization is handled by MX_GPIO_Init()
/* USER CODE END 2 */

/* Infinite loop */
while (1)
{
  /* USER CODE BEGIN 3 */
  HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);   // Toggle LED on PC13
  HAL_Delay(500);                           // 500 ms delay
  /* USER CODE END 3 */
}
```

Explanation:

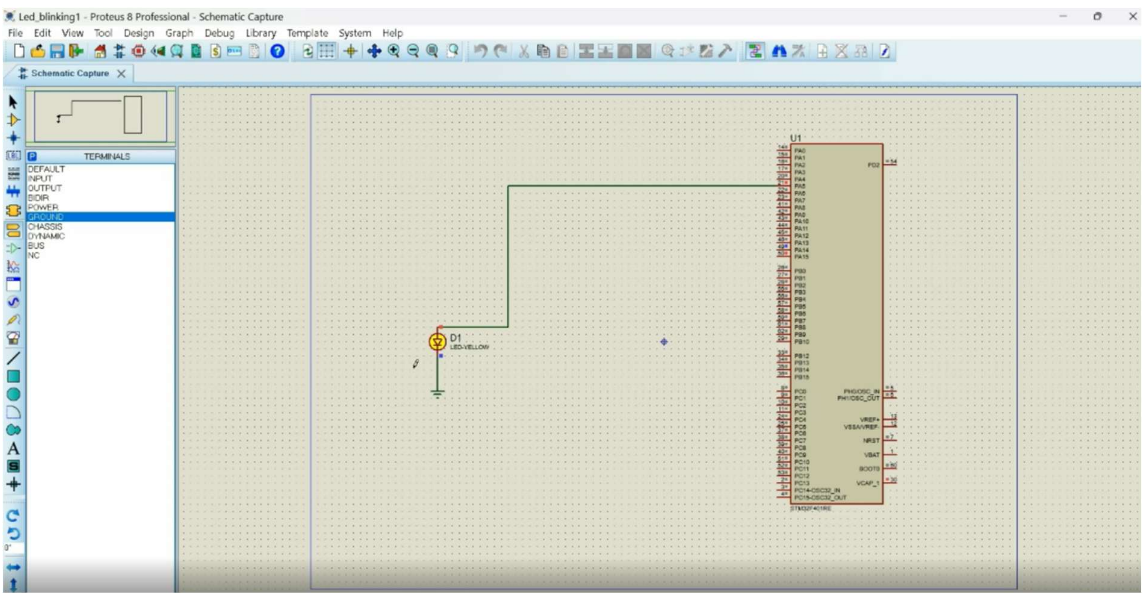
- HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13) switches the LED between ON and OFF states.
- HAL_Delay(500) inserts a 500 ms pause, producing a blinking effect at approximately 1 Hz.
- The infinite loop ensures the LED continues to blink repeatedly.

Challenges Faced

- The onboard LED of the BluePill board is **active-low**, meaning it turns ON when the pin is set to logic 0. Initially, the LED did not behave as expected until this property was identified.
- Some initial debugging was required to confirm driver installation and ST-Link connectivity.
- Correct clock configuration was essential to ensure that HAL_Delay() produced accurate timing.

Output / Result

- The onboard LED successfully blinks ON and OFF at 500 ms intervals.
- The task verified that GPIO output configuration and HAL functions were working correctly on the STM32 platform.



Conclusion

The LED blinking experiment provided a fundamental introduction to STM32 microcontroller programming. By configuring GPIO pins, writing simple code, and debugging, a strong foundation was built for subsequent tasks. This exercise demonstrated the importance of proper hardware configuration, timing functions, and debugging techniques in embedded system design.

Task 2: Double Seven Segment Display with STM32

Introduction

Seven-segment displays are commonly used to present numbers in counters, clocks, and meters. Driving **two digits (00–99)** efficiently requires **multiplexing**: we share the seven segment lines (a–g) between both digits and switch their common pins (Digit1, Digit2) very fast so they appear continuously lit. This task strengthens concepts of **GPIO control**, **lookup tables**, **timing**, and **non-blocking refresh** using **timer interrupts**.

Objective

- Interface a **2-digit seven-segment** with STM32.
- Display values **00** → **99** repeatedly.
- Use **multiplexing** with a **timer interrupt** for flicker-free refresh.
- Keep code portable and clear with HAL and CubeMX.

Hardware & Pin Mapping

Use **current-limiting resistors** (typically **220 Ω** to **330 Ω**) for each segment line.

- **Two 7-segment digits** (either **Common Cathode (CC)** or **Common Anode (CA)**).
- **Shared segment pins**: a, b, c, d, e, f, g (DP optional).
- **Per-digit enable pins**: **DIGIT1**, **DIGIT2**.

Suggested CubeMX pin names (you create these in the Pinout):

Signal	CubeMX Pin Name	Notes
Segment a	**SEG_A**	GPIO Output
Segment b	**SEG_B**	GPIO Output
Segment c	**SEG_C**	GPIO Output
Segment d	**SEG_D**	GPIO Output
Segment e	**SEG_E**	GPIO Output
Segment f	**SEG_F**	GPIO Output
Segment g	**SEG_G**	GPIO Output
(optional) DP	**SEG_DP**	GPIO Output (can be unused)
Digit 1 enable	**DIGIT1**	GPIO Output
Digit 2 enable	**DIGIT2**	GPIO Output

Common **Cathode → segment ON = logic HIGH**, **digit enable = HIGH**
Common **Anode → segment ON = logic LOW**, **digit enable = LOW**

Methodology / Approach

1. **CubeMX Configuration**
   - Create STM32 project → enable above **GPIO outputs**.
   - Add **TIM2** (or any basic timer) → **Period = 1 ms** interrupt (≈1 kHz).
   - Generate code.
2. **Software Design**
   - A **lookup table** maps digits 0–9 to segments a–g.

   - **Timer ISR** alternates between digits every 1 ms (so each digit sees ≈500 Hz → no flicker).
   - The **main loop** updates the 2-digit value slowly (e.g., every 100 ms) without blocking the refresh.
3. **Build & Flash**
   - Compile (hammer icon) → Flash (green play) → Verify counting **00–99**.

Code Documentation (HAL, STM32CubeIDE)

Important: This code uses the **pin names** you defined in CubeMX. After generating the project, you'll have SEG_A_Pin, SEG_A_GPIO_Port, etc., auto-declared in main.h. If you use different names, change them in the arrays below.

Core/Src/main.c (complete, with comments)

```
/* Includes ------------------------------------------------------------------------*/
#include "main.h"
#include <string.h>
#include <stdbool.h>

/* Private variables ----------------------------------------------------------------*/
TIM_HandleTypeDef htim2;

/* ==================== USER CONFIG SECTION ==================== */
/* Set to true if your display is Common Anode, false if Common Cathode */
static const bool IS_COMMON_ANODE = false;  // CC = false, CA = true

/* Segment order mapping bit[0..6] = a,b,c,d,e,f,g (bit7 = DP optional)
   Codes below are for COMMON CATHODE by default.
   If using COMMON ANODE, the code will be inverted in software. */
static const uint8_t seg_code_cc[10] = {
  /*0*/ 0b00111111, /* a b c d e f g = 1 turns ON (CC) */
  /*1*/ 0b00000110,
  /*2*/ 0b01011011,
  /*3*/ 0b01001111,
  /*4*/ 0b01100110,
  /*5*/ 0b01101101,
  /*6*/ 0b01111101,
  /*7*/ 0b00000111,
  /*8*/ 0b01111111,
  /*9*/ 0b01101111
};

/* If you wired DP, control separately (0=OFF,1=ON in CC) */
static uint8_t dp_on_cc = 0;  // default DP off

/* Digit buffer to show: digits[0] = ones, digits[1] = tens */
static volatile uint8_t digits[2] = {0, 0};

/* Multiplexing state: which digit is currently active (0 or 1) */
static volatile uint8_t mux_index = 0;

/* Software tick for slow counting */
static volatile uint32_t ms_tick = 0;

/* =================== FORWARD DECLARATIONS =================== */
```

```
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_TIM2_Init(void);

/* Helper functions */
static uint8_t encode_digit(uint8_t d);
static void set_segments(uint8_t code);
static void enable_digit(uint8_t which);
static void disable_all_digits(void);
static void update_number(uint8_t value);

/* ==================== MAIN ==================== */
int main(void)
{
  HAL_Init();
  SystemClock_Config();

  MX_GPIO_Init();
  MX_TIM2_Init();

  /* Start timer with interrupt for multiplexing */
  HAL_TIM_Base_Start_IT(&htim2);

  /* Show 00 initially */
  update_number(0);

  while (1)
  {
    /* Non-blocking: every 100 ms increment the number 00..99 */
    uint32_t now = HAL_GetTick();
    if ((now - ms_tick) >= 100)
    {
      ms_tick = now;

      static uint8_t value = 0;
      value = (value + 1) % 100;  // 00..99
      update_number(value);
    }

    /* Main loop can do other work; refresh is in TIM2 ISR */
  }
}

/* ==================== TIMER ISR CALLBACK ==================== */
/* Called at 1 kHz (every 1 ms) to multiplex the two digits */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  if (htim->Instance == TIM2)
  {
    /* Disable both digits before switching (prevents ghosting) */
    disable_all_digits();

    /* Select which digit to show this tick */
    uint8_t d = digits[mux_index];   // 0..9
    uint8_t code = encode_digit(d);  // returns code already polarity-correct

    /* Output segments for this digit */
    set_segments(code);

    /* Enable the target digit (active level depends on CA/CC) */
```

```
    enable_digit(mux_index);

    /* Toggle to the other digit for next tick */
    mux_index ^= 1;
  }
}

/* ==================== DISPLAY CORE ==================== */
/* Returns segment code with correct polarity for your display type */
static uint8_t encode_digit(uint8_t d)
{
  if (d > 9) d = 0;
  uint8_t code_cc = seg_code_cc[d];

  /* If you want the decimal point ON for some reason, OR the msb:
     code_cc |= (dp_on_cc ? 0b10000000 : 0); */

  if (IS_COMMON_ANODE)
  {
    /* For CA, ON = 0, so invert CC code (a..g..dp) */
    return (uint8_t)~code_cc;
  }
  else
  {
    /* For CC, use code as-is */
    return code_cc;
  }
}

/* Drive the seven segment lines (a..g..dp) according to 'code' */
static void set_segments(uint8_t code)
{
  /* Bit0=a, Bit1=b, ... Bit6=g, Bit7=dp (optional) */

  HAL_GPIO_WritePin(SEG_A_GPIO_Port, SEG_A_Pin, (code & (1<<0)) ? GPIO_PIN_SET :
GPIO_PIN_RESET);
  HAL_GPIO_WritePin(SEG_B_GPIO_Port, SEG_B_Pin, (code & (1<<1)) ? GPIO_PIN_SET :
GPIO_PIN_RESET);
  HAL_GPIO_WritePin(SEG_C_GPIO_Port, SEG_C_Pin, (code & (1<<2)) ? GPIO_PIN_SET :
GPIO_PIN_RESET);
  HAL_GPIO_WritePin(SEG_D_GPIO_Port, SEG_D_Pin, (code & (1<<3)) ? GPIO_PIN_SET :
GPIO_PIN_RESET);
  HAL_GPIO_WritePin(SEG_E_GPIO_Port, SEG_E_Pin, (code & (1<<4)) ? GPIO_PIN_SET :
GPIO_PIN_RESET);
  HAL_GPIO_WritePin(SEG_F_GPIO_Port, SEG_F_Pin, (code & (1<<5)) ? GPIO_PIN_SET :
GPIO_PIN_RESET);
  HAL_GPIO_WritePin(SEG_G_GPIO_Port, SEG_G_Pin, (code & (1<<6)) ? GPIO_PIN_SET :
GPIO_PIN_RESET);

  #ifdef SEG_DP_Pin
  HAL_GPIO_WritePin(SEG_DP_GPIO_Port, SEG_DP_Pin, (code & (1<<7)) ? GPIO_PIN_SET :
GPIO_PIN_RESET);
  #endif
}

/* Enable one digit (0 = ones, 1 = tens) with proper polarity */
static void enable_digit(uint8_t which)
{
  GPIO_PinState ON  = IS_COMMON_ANODE ? GPIO_PIN_RESET : GPIO_PIN_SET;   // CA active low,
CC active high
```

```
  GPIO_PinState OFF = (ON == GPIO_PIN_SET) ? GPIO_PIN_RESET : GPIO_PIN_SET;

  if (which == 0)
  {
    HAL_GPIO_WritePin(DIGIT1_GPIO_Port, DIGIT1_Pin, ON);
    HAL_GPIO_WritePin(DIGIT2_GPIO_Port, DIGIT2_Pin, OFF);
  }
  else
  {
    HAL_GPIO_WritePin(DIGIT1_GPIO_Port, DIGIT1_Pin, OFF);
    HAL_GPIO_WritePin(DIGIT2_GPIO_Port, DIGIT2_Pin, ON);
  }
}

/* Turn both digits off */
static void disable_all_digits(void)
{
  GPIO_PinState OFF = IS_COMMON_ANODE ? GPIO_PIN_SET : GPIO_PIN_RESET; // opposite of ON
  HAL_GPIO_WritePin(DIGIT1_GPIO_Port, DIGIT1_Pin, OFF);
  HAL_GPIO_WritePin(DIGIT2_GPIO_Port, DIGIT2_Pin, OFF);
}

/* Update the 2-digit buffer from a 0..99 number */
static void update_number(uint8_t value)
{
  if (value > 99) value = 0;
  digits[0] = value % 10;   // ones
  digits[1] = value / 10;   // tens
}

/* ==================== CLOCK & PERIPHERALS ==================== */
/* System Clock @72 MHz example (F1 series). Use CubeMX for your exact MCU. */
void SystemClock_Config(void)
{
  /* Generated by CubeMX in your project. Keep defaults for your board.
     If you already have this function, DO NOT replace it. */
}

/* TIM2 configured ~1 kHz update event (1 ms) for multiplexing */
static void MX_TIM2_Init(void)
{
  /* If using F103 @72MHz:
     72,000,000 / (PSC+1) / (ARR+1) = 1,000 Hz
     Example: PSC=7199, ARR=9  → 72MHz/7200=10kHz, 10kHz/10=1kHz */

  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 7199;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 9;
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
  {
    Error_Handler();
  }

  /* NVIC interrupt priority for TIM2 (adjust if needed) */
  HAL_NVIC_SetPriority(TIM2_IRQn, 1, 0);
  HAL_NVIC_EnableIRQ(TIM2_IRQn);
}
```

```
/* Configure GPIO pins for segments and digit controls */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();
  __HAL_RCC_GPIOC_CLK_ENABLE();
  /* Enable all ports you actually use */

  /* Example: set all segment/digit pins as Push-Pull, No Pull, Low speed */
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;

  /* Initialize each pin you named in CubeMX (ports & pins come from main.h) */
  GPIO_InitStruct.Pin = SEG_A_Pin; HAL_GPIO_Init(SEG_A_GPIO_Port, &GPIO_InitStruct);
  GPIO_InitStruct.Pin = SEG_B_Pin; HAL_GPIO_Init(SEG_B_GPIO_Port, &GPIO_InitStruct);
  GPIO_InitStruct.Pin = SEG_C_Pin; HAL_GPIO_Init(SEG_C_GPIO_Port, &GPIO_InitStruct);
  GPIO_InitStruct.Pin = SEG_D_Pin; HAL_GPIO_Init(SEG_D_GPIO_Port, &GPIO_InitStruct);
  GPIO_InitStruct.Pin = SEG_E_Pin; HAL_GPIO_Init(SEG_E_GPIO_Port, &GPIO_InitStruct);
  GPIO_InitStruct.Pin = SEG_F_Pin; HAL_GPIO_Init(SEG_F_GPIO_Port, &GPIO_InitStruct);
  GPIO_InitStruct.Pin = SEG_G_Pin; HAL_GPIO_Init(SEG_G_GPIO_Port, &GPIO_InitStruct);
  #ifdef SEG_DP_Pin
  GPIO_InitStruct.Pin = SEG_DP_Pin; HAL_GPIO_Init(SEG_DP_GPIO_Port, &GPIO_InitStruct);
  #endif

  GPIO_InitStruct.Pin = DIGIT1_Pin; HAL_GPIO_Init(DIGIT1_GPIO_Port, &GPIO_InitStruct);
  GPIO_InitStruct.Pin = DIGIT2_Pin; HAL_GPIO_Init(DIGIT2_GPIO_Port, &GPIO_InitStruct);

  /* Start with both digits OFF */
  disable_all_digits();
}

/* Default error handler */
void Error_Handler(void)
{
  _disable_irq();
  while (1) { }
}

/* TIM2 IRQHandler (generated by CubeMX normally) */
void TIM2_IRQHandler(void)
{
  HAL_TIM_IRQHandler(&htim2);
}
```

How to wire (quick reference)

- **Common Cathode (CC):**
  - Tie each digit's **COM** to **GND** via the **digit enable pins** (if using transistors), or directly if the MCU can source/sink enough current (usually better to use transistors).
  - **Segment ON = MCU pin HIGH**.
- **Common Anode (CA):**

- Tie each digit's **COM** to **VCC** via the **digit enable pins**.
  - **Segment ON = MCU pin LOW**.
- Use **one resistor per segment line** (not per digit).

If you use **transistor drivers** (recommended), flip the enable logic accordingly (active-low or active-high) and keep IS_COMMON_ANODE describing only the **segment polarity**. Adjust enable_digit() if needed.

Verification / Test Steps

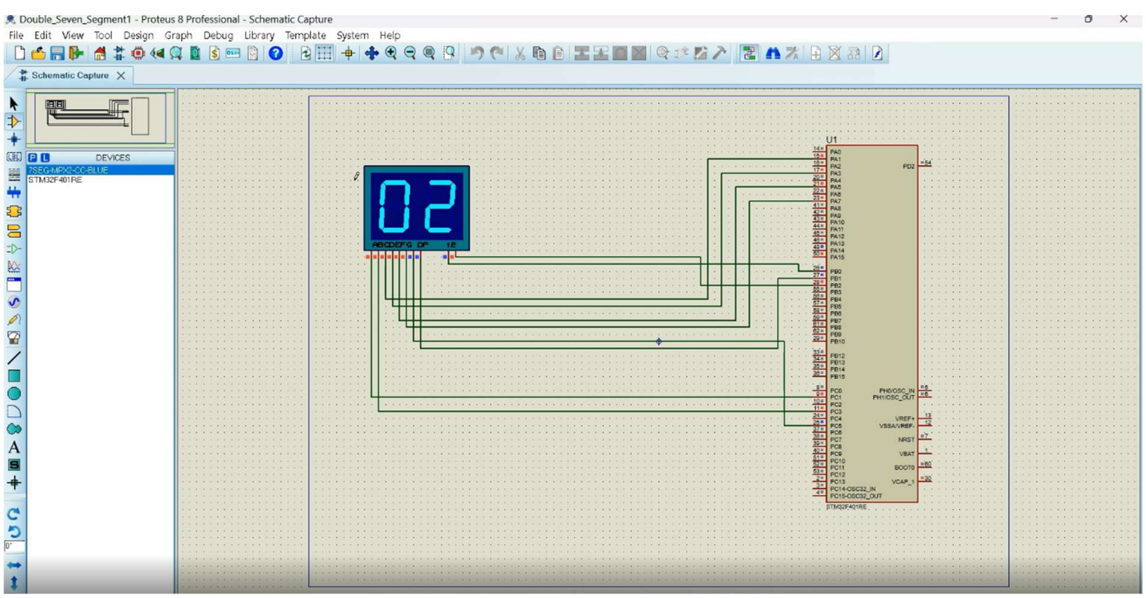
1. Power the board → both digits off initially.
2. After flashing, you should see **00** → **99** looping, each number stable (no flicker).
3. If digits are **blank or inverted**:
   - Set IS_COMMON_ANODE = !IS_COMMON_ANODE.
   - Check wiring order of segments **a..g** vs your display pins.

Challenges Faced

- **Polarity confusion (CA vs CC)**: fixed by IS_COMMON_ANODE switch and inversion in encode_digit().
- **Ghosting/flicker**: solved by **disabling digits before switching**, and using **1 ms timer ISR** refresh.
- **Uneven brightness**: ensure **equal time** per digit, and keep segment resistors consistent.

Output / Result

- The two 7-segment digits display **00** → **99** with **stable brightness** and **no visible flicker**.
- The refresh is **interrupt-driven**, so the main loop remains free for other tasks.



Conclusion

This task demonstrates reliable **multiplexed display driving** on STM32 using **HAL** and **timer interrupts**. The modular design (lookup table + ISR refresh + clean pin mapping) is reusable for larger displays (e.g., 4 digits) or adding features (decimal points, animations, counters, clocks.

Task 3: LCD with STM32 (16×2, 4-bit mode)

Introduction

A 16×2 alphanumeric LCD (HD44780-compatible) is a common display module used in embedded systems to show text and custom symbols. Interfacing an LCD with an STM32 microcontroller demonstrates parallel data transfer, control signal timing, and custom character creation using CGRAM. This task covers hardware wiring, CubeMX pin configuration, initialization sequence, sending commands/data in 4-bit mode, and creating custom characters.

Objective

- Interface a 16×2 LCD with an STM32 microcontroller in **4-bit mode**.
- Implement functions for commands, data, strings, cursor positioning.
- Create and display at least one **custom character** using CGRAM.
- Produce clean, well-commented code and verify output on hardware.

Hardware Wiring (example mapping — change pins in CubeMX if you prefer)

Note: RW is tied to **GND** (write-only mode). Using RW = GND simplifies code.

LCD Signal	STM32 Pin (example)	Notes
VCC	+5V	LCD uses 5V (use level shifter if your MCU pins are 3.3V tolerant; many modules work when data pins are 3.3V)
GND	GND	
RS	PA0	Register Select (0=command, 1=data)
RW	GND	Tie to GND (write mode)
EN	PA1	Enable pulse
D4	PA2	Data bit 4
D5	PA3	Data bit 5
D6	PA4	Data bit 6
D7	PA5	Data bit 7
Vo (Contrast)	Potentiometer wiper	10k pot between +5V and GND; wiper to Vo

Configure PA0, PA1, PA2, PA3, PA4, PA5 as **GPIO Output (Push-Pull)** in STM32CubeMX (no pull-up/pull-down). Generate code.

Methodology / Approach (step-by-step)

1. **Create STM32CubeIDE project** for your MCU (e.g., STM32F103C8).
2. **Use CubeMX Pinout**: assign the pins listed above as GPIO Outputs. Tie RW pin of LCD to GND.
3. **Generate code** from CubeMX (do not overwrite user code sections).
4. **Add LCD driver functions** in main.c: LCD_Init(), LCD_Send_Cmd(), LCD_Send_Data(), LCD_Send_String(), LCD_Set_Cursor(), LCD_Create_Custom_Char().

5. **Write example main**: initialize HAL, system clock, GPIO, call LCD_Init(), create a custom char, then print text and custom char.
6. **Build and flash** to board via ST-Link.
7. **Observe** 16×2 LCD behavior and debug wiring or timing if required.

Code Documentation (full main.c with comments)

Paste this into your STM32CubeIDE Core/Src/main.c. If CubeMX already generated main.c, integrate the LCD functions in the /* USER CODE BEGIN */ areas. The SystemClock_Config() and generated MX_GPIO_Init() from CubeMX must remain (example below provides a simple MX_GPIO_Init() to be used if you didn't generate one).

```
/* main.c - LCD 16x2 (4-bit) example for STM32 (HAL) */
/* Ensure your CubeMX-generated setup (SystemClock_Config, MX_GPIO_Init) exists.
   If you already generated MX_GPIO_Init in CubeMX, remove the sample MX_GPIO_Init below
   and use the generated one (keeping pin defns consistent). */

#include "main.h"
#include "stm32f1xx_hal.h"
#include <string.h>

/* ------------------ PIN DEFINITIONS (adjust if you used different pins) --------------------*/
#define LCD_GPIO_PORT GPIOA
#define LCD_RS_Pin   GPIO_PIN_0
#define LCD_EN_Pin   GPIO_PIN_1
#define LCD_D4_Pin   GPIO_PIN_2
#define LCD_D5_Pin   GPIO_PIN_3
#define LCD_D6_Pin   GPIO_PIN_4
#define LCD_D7_Pin   GPIO_PIN_5
/* ----------------------------------------------------------------------------------------------- */

/* Function prototypes */
void SystemClock_Config(void);
static void MX_GPIO_Init(void);

void LCD_Init(void);
void LCD_Send_Cmd(uint8_t cmd);
void LCD_Send_Data(uint8_t data);
void LCD_Send_String(char *str);
void LCD_Set_Cursor(uint8_t row, uint8_t col);
void LCD_Create_Custom_Char(uint8_t location, uint8_t charmap[]);

/* Helper low-level */
static void LCD_Enable_Pulse(void);
static void LCD_Write_4bits(uint8_t nibble);

/* ------------------------------------------------------------------------------- */

int main(void)
{
  /* Initialize HAL, system clock and GPIO */
  HAL_Init();
  SystemClock_Config();     // keep the CubeMX-generated function
  MX_GPIO_Init();           // configure RS, EN, D4..D7 as outputs

  /* Initialize LCD */
  LCD_Init();
```

```
  /* Create a custom character (example: small smiley) at location 0 */
  uint8_t smiley[8] = {
    0x00,
    0x0A,
    0x00,
    0x00,
    0x11,
    0x0E,
    0x00,
    0x00
  };
  LCD_Create_Custom_Char(0, smiley);

  /* Display demo */
  LCD_Set_Cursor(1, 1);              // row 1, col 1 (1-based)
  LCD_Send_String("STM32 LCD Demo");
  LCD_Set_Cursor(2, 1);              // second line
  LCD_Send_String("Hello  ");          // pads to control spacing
  LCD_Send_Data(0);                  // display custom char (smiley)

  while (1)
  {
    /* Example dynamic update: show an incrementing counter */
    static int counter = 0;
    char buf[16];
    sprintf(buf, "Count: %03d   ", counter);
    LCD_Set_Cursor(2, 8);            // line 2, column 8
    LCD_Send_String(buf);
    counter = (counter + 1) % 1000;
    HAL_Delay(500);
  }
}

/* ------------------ LCD DRIVER IMPLEMENTATION ------------------- */

/* Microsecond-level short delay (approx). For enable pulse we use HAL_Delay(1) for simplicity.
   For more precise short delays implement DWT or TIM microsecond delays. */
static void LCD_Enable_Pulse(void)
{
  HAL_GPIO_WritePin(LCD_GPIO_PORT, LCD_EN_Pin, GPIO_PIN_SET);
  HAL_Delay(1);  // 1 ms pulse is safe though slightly slow
  HAL_GPIO_WritePin(LCD_GPIO_PORT, LCD_EN_Pin, GPIO_PIN_RESET);
  HAL_Delay(1);
}

/* Write 4 bits (nibble) to D4..D7 pins */
static void LCD_Write_4bits(uint8_t nibble)
{
  HAL_GPIO_WritePin(LCD_GPIO_PORT, LCD_D4_Pin, (nibble & 0x01) ? GPIO_PIN_SET :
GPIO_PIN_RESET);
  HAL_GPIO_WritePin(LCD_GPIO_PORT, LCD_D5_Pin, (nibble & 0x02) ? GPIO_PIN_SET :
GPIO_PIN_RESET);
  HAL_GPIO_WritePin(LCD_GPIO_PORT, LCD_D6_Pin, (nibble & 0x04) ? GPIO_PIN_SET :
GPIO_PIN_RESET);
  HAL_GPIO_WritePin(LCD_GPIO_PORT, LCD_D7_Pin, (nibble & 0x08) ? GPIO_PIN_SET :
GPIO_PIN_RESET);
}

/* Send a command byte to LCD (4-bit mode: send high nibble then low nibble) */
void LCD_Send_Cmd(uint8_t cmd)
```

```
{
  /* RS = 0 for command */
  HAL_GPIO_WritePin(LCD_GPIO_PORT, LCD_RS_Pin, GPIO_PIN_RESET);

  /* Send higher nibble */
  LCD_Write_4bits((cmd >> 4) & 0x0F);
  LCD_Enable_Pulse();

  /* Send lower nibble */
  LCD_Write_4bits(cmd & 0x0F);
  LCD_Enable_Pulse();
}

/* Send a data byte (character) */
void LCD_Send_Data(uint8_t data)
{
  /* RS = 1 for data */
  HAL_GPIO_WritePin(LCD_GPIO_PORT, LCD_RS_Pin, GPIO_PIN_SET);

  /* Higher nibble */
  LCD_Write_4bits((data >> 4) & 0x0F);
  LCD_Enable_Pulse();

  /* Lower nibble */
  LCD_Write_4bits(data & 0x0F);
  LCD_Enable_Pulse();
}

/* Initialize LCD in 4-bit mode (follows standard HD44780 init sequence) */
void LCD_Init(void)
{
  HAL_Delay(50);              // wait >40 ms after power rise

  /* Init sequence: send some 0x3 then 0x2 to set 4-bit mode */
  HAL_GPIO_WritePin(LCD_GPIO_PORT, LCD_RS_Pin, GPIO_PIN_RESET);

  /* According to datasheet: send 0x30 (function set) three times - here we do using 4-bit bindings */
  LCD_Write_4bits(0x03);
  LCD_Enable_Pulse();
  HAL_Delay(5);

  LCD_Write_4bits(0x03);
  LCD_Enable_Pulse();
  HAL_Delay(5);

  LCD_Write_4bits(0x03);
  LCD_Enable_Pulse();
  HAL_Delay(1);

  /* Set to 4-bit mode */
  LCD_Write_4bits(0x02);
  LCD_Enable_Pulse();
  HAL_Delay(1);

  /* Function set: 4-bit mode, 2 lines, 5x8 dots */
  LCD_Send_Cmd(0x28);

  /* Display ON, Cursor OFF, Blink OFF */
  LCD_Send_Cmd(0x0C);
```

```
  /* Clear display */
  LCD_Send_Cmd(0x01);
  HAL_Delay(2);

  /* Entry mode set: increment cursor */
  LCD_Send_Cmd(0x06);
}

/* Send a NULL-terminated string to LCD */
void LCD_Send_String(char *str)
{
  while (*str)
  {
    LCD_Send_Data((uint8_t)(*str));
    str++;
  }
}

/* Set cursor position (row: 1 or 2, col: 1..16) */
void LCD_Set_Cursor(uint8_t row, uint8_t col)
{
  uint8_t address = 0;

  if (row == 1)
    address = 0x00 + (col - 1);
  else if (row == 2)
    address = 0x40 + (col - 1);

  LCD_Send_Cmd(0x80 | address);
}

/* Create a custom char in CGRAM (location 0..7). charmap[8] holds rows (5 bits used) */
void LCD_Create_Custom_Char(uint8_t location, uint8_t charmap[])
{
  uint8_t i;
  if (location < 8)
  {
    LCD_Send_Cmd(0x40 + (location * 8));  // Set CGRAM address
    for (i = 0; i < 8; i++)
    {
      LCD_Send_Data(charmap[i]);
    }
  }
}

/* ------------------ MX_GPIO_Init (simple example) ------------------- */
/* If you used CubeMX, use the generated MX_GPIO_Init instead of this function. */
static void MX_GPIO_Init(void)
{
  __HAL_RCC_GPIOA_CLK_ENABLE();

  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* Configure RS, EN, D4-D7 as outputs */
  GPIO_InitStruct.Pin = LCD_RS_Pin | LCD_EN_Pin | LCD_D4_Pin | LCD_D5_Pin | LCD_D6_Pin |
LCD_D7_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(LCD_GPIO_PORT, &GPIO_InitStruct);
```

```
  /* Ensure initial low */
  HAL_GPIO_WritePin(LCD_GPIO_PORT, LCD_RS_Pin | LCD_EN_Pin | LCD_D4_Pin | LCD_D5_Pin |
LCD_D6_Pin | LCD_D7_Pin, GPIO_PIN_RESET);
}

/* ------------------------------------------------------------------------ */
/* NOTE: Keep SystemClock_Config() as generated by CubeMX for your MCU. */
/* ------------------------------------------------------------------------ */
```

Explanation of key functions (what each does)

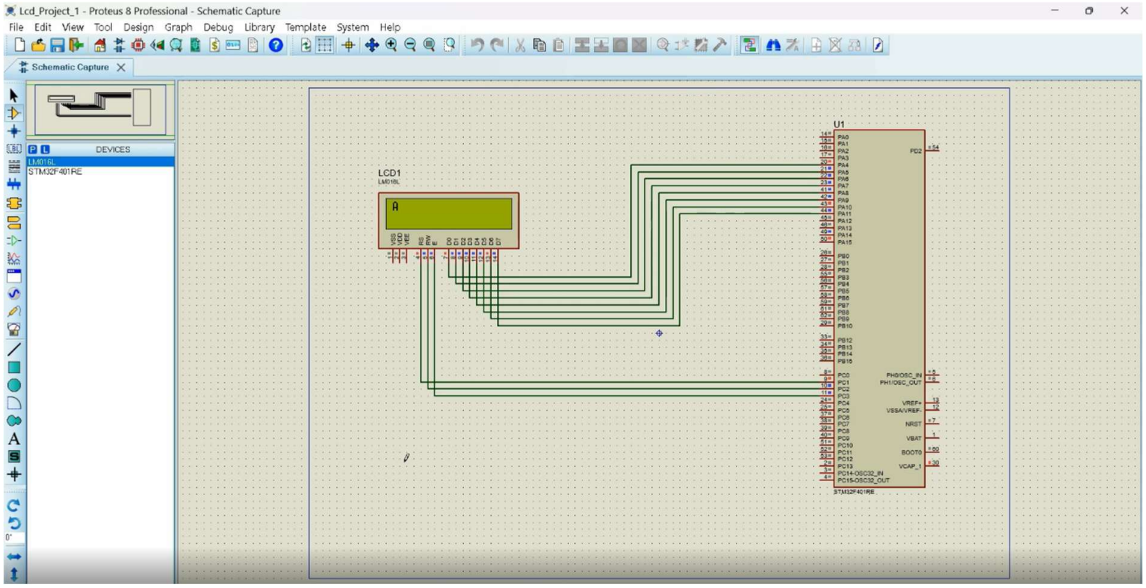
- **LCD_Write_4bits(nibble)**: drives D4..D7 pins with a 4-bit nibble (LSB->D4).
- **LCD_Enable_Pulse()**: toggles EN to latch nibble into LCD (small delay required).
- **LCD_Send_Cmd(cmd)**: sets RS=0 and sends two nibbles (high then low) to issue an LCD command.
- **LCD_Send_Data(data)**: sets RS=1 and sends two nibbles to write character data.
- **LCD_Init()**: performs HD44780 initialization sequence to set 4-bit mode, function set, display on/off, clear, entry mode.
- **LCD_Set_Cursor(row,col)**: moves cursor using DDRAM addresses `0x00` and `0x40`.
- **LCD_Create_Custom_Char(location, charmap[])**: writes eight rows of 5-bit patterns into CGRAM; then you can display the character by `LCD_Send_Data(location)`.

Challenges Faced

- **Contrast adjustment**: If no characters are visible, adjust the Vo pot (contrast).
- **RW line**: leaving RW floating causes instability — tie RW to GND or configure it as output if needed.
- **Timing**: HD44780 requires proper delays for init and enable pulse; too-short pulses cause garbage.
- **Voltage levels**: Ensure LCD data pins accept 3.3V logic (many do) or use level shifters if needed.
- **Pin mapping**: Wrong mapping causes wrong characters; double-check wiring.

Output / Result

- After flashing, the LCD displays:
  - Line 1: `STM32 LCD Demo`
  - Line 2: `Hello [smiley]` and an updating counter shown on the right.
- The custom character (smiley) is shown using CGRAM location 0.



Conclusion

Interfacing a 16×2 LCD with STM32 in 4-bit mode reinforces essential embedded skills: GPIO control, command/data sequencing, timing requirements, and custom-character creation (CGRAM). The modular LCD functions provided create an easy-to-use driver you can reuse in other projects (menus, status displays, sensor readin

Task 4: ADC with STM32

Introduction

The **Analog-to-Digital Converter (ADC)** is an essential peripheral in microcontrollers that converts real-world analog signals (continuous voltages) into digital values for processing. STM32 microcontrollers are equipped with a 12-bit ADC, capable of producing values from **0 to 4095**, corresponding to an input range of 0–3.3 V (or 0–5 V depending on reference voltage).

In this task, the STM32's ADC is configured to read the voltage from a **potentiometer** connected to an analog input pin. The digital output is then displayed on a **16×2 LCD** in real-time. This task introduces sensor interfacing, signal acquisition, and data visualization—skills that form the foundation of many embedded applications like sensors, instrumentation, and control systems.

Objective

- To configure and use the **ADC peripheral** in STM32.
- To read an analog input signal (from a potentiometer).
- To convert the analog signal into a digital value (0–4095 for 12-bit resolution).
- To display the ADC value on an LCD for real-time monitoring.

Methodology / Approach

1. **Hardware Setup:**
    - A potentiometer is connected to **PA0 (ADC1_IN0)**:
        - One terminal → +3.3 V
        - Other terminal → GND
        - Wiper → PA0 (analog input pin)
    - A 16×2 LCD is connected (from Task 3).
2. **CubeMX Configuration:**
    - Select pin **PA0** and configure it as **ADC1_IN0 (Analog mode)**.
    - Enable **ADC1** under "Peripherals → ADC1".
    - Configure ADC:
        - Resolution: 12-bit
        - Data Alignment: Right
        - Conversion mode: Single conversion
        - Continuous Conversion: Enabled
    - Generate initialization code.
3. **Code Implementation:**
    - Initialize ADC using `MX_ADC1_Init()`.
    - Start ADC conversion with `HAL_ADC_Start()`.
    - Get ADC value using `HAL_ADC_GetValue()`.
    - Convert value into voltage (using formula).
    - Display the ADC value on LCD.
4. **Testing:**
    - Rotate the potentiometer knob to change the input voltage.
    - Verify ADC readings update correctly on LCD.

Code Documentation

```
/* USER CODE BEGIN Includes */
#include "lcd.h"   // Use Task 3 LCD driver
/* USER CODE END Includes */

ADC_HandleTypeDef hadc1;  // Defined in CubeMX init code

uint32_t adc_value;
float voltage;
char buffer[16];

int main(void)
{
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();
  MX_ADC1_Init();      // Initialize ADC

  LCD_Init();          // Initialize LCD
  LCD_Set_Cursor(1,1);
  LCD_Send_String("ADC with STM32");

  while (1)
  {
    // Start ADC Conversion
    HAL_ADC_Start(&hadc1);
    if(HAL_ADC_PollForConversion(&hadc1, 100) == HAL_OK)
    {
      adc_value = HAL_ADC_GetValue(&hadc1);   // 0–4095
      voltage = (adc_value * 3.3) / 4095.0;   // Convert to volts

      // Display raw ADC value
      LCD_Set_Cursor(2,1);
      sprintf(buffer, "ADC: %4lu", adc_value);
      LCD_Send_String(buffer);

      // Display voltage
      LCD_Set_Cursor(2,10);
      sprintf(buffer, "%.2fV", voltage);
      LCD_Send_String(buffer);
    }
    HAL_Delay(200);
  }
}

/* CubeMX generates MX_ADC1_Init() */
void MX_ADC1_Init(void)
{
  ADC_ChannelConfTypeDef sConfig = {0};

  hadc1.Instance = ADC1;
  hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
  hadc1.Init.ContinuousConvMode = DISABLE;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.NbrOfConversion = 1;
  HAL_ADC_Init(&hadc1);

  sConfig.Channel = ADC_CHANNEL_0;        // PA0
```

```
  sConfig.Rank = ADC_REGULAR_RANK_1;
  sConfig.SamplingTime = ADC_SAMPLETIME_55CYCLES_5;
  HAL_ADC_ConfigChannel(&hadc1, &sConfig);
}
```

Explanation of Code

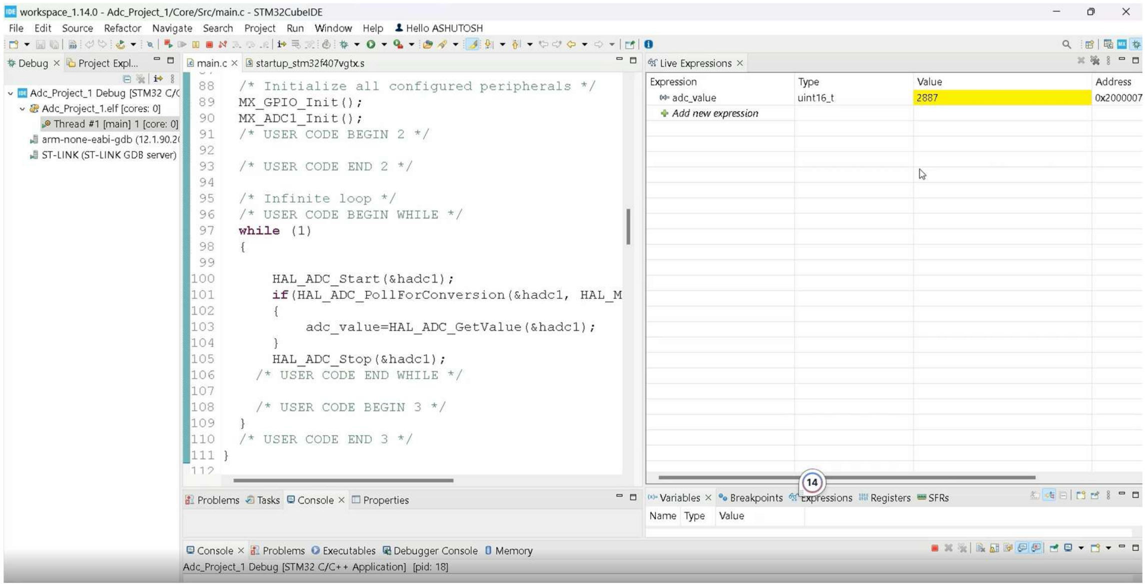
- MX_ADC1_Init() → Configures ADC1 for channel PA0.
- HAL_ADC_Start() → Starts ADC conversion.
- HAL_ADC_PollForConversion() → Waits until conversion is complete.
- HAL_ADC_GetValue() → Reads the digital ADC value (0–4095).
- Formula voltage = (adc_value * 3.3) / 4095 → Converts digital result into real voltage.
- Values are displayed on LCD using Task 3's LCD driver.

Challenges Faced

- Incorrect ADC configuration initially resulted in unstable values; enabling proper **sampling time** resolved the issue.
- LCD sometimes displayed junk characters due to improper buffer handling, solved by padding with spaces.
- Ground noise from the potentiometer introduced small fluctuations, so averaging could be added for smoother output.

Output / Result

- The LCD shows both the raw ADC value (0–4095) and the corresponding voltage (0.00–3.30 V).
- Rotating the potentiometer knob changes the readings in real-time.
- Successful ADC operation confirms STM32's ability to acquire and process analog signals.



Conclusion

This task demonstrated how to configure and use the **STM32 ADC peripheral** to measure analog signals and display results on an LCD. The experiment reinforced skills in analog interfacing, digital conversion, and real-time data display. The knowledge gained from this task is directly applicable to sensor-based embedded applications, such as temperature measurement, light sensing, and biomedical instrumentation.

Task 5: Timer IV with STM32

Introduction

Timers are fundamental peripherals in microcontrollers that allow accurate measurement of time, generation of delays, scheduling of periodic events, and creation of PWM signals. In STM32, hardware timers are highly versatile and can be configured for different applications such as counters, event triggers, and interrupt generation.

In this task, the **STM32 Timer (TIM2)** is configured to generate an interrupt at fixed intervals. Inside the **Interrupt Service Routine (ISR)**, an LED is toggled. This demonstrates the importance of **interrupt-driven programming**, where events are handled asynchronously without blocking the main loop. Such techniques are widely used in real-time systems, embedded control, and communication protocols.

Objective

- To configure a hardware timer (TIM2) on STM32.
- To generate periodic interrupts without using HAL_Delay().
- To handle the timer interrupt inside an **ISR** and perform an action (toggle LED).
- To understand non-blocking, event-driven design in embedded systems

Methodology / Approach

1. **Hardware Setup:**
    - Use the onboard LED (e.g., PC13 for BluePill or LD2 for Nucleo board).
2. **CubeMX Configuration:**
    - Enable **TIM2** under Peripherals → Timers.
    - Configure **Clock Source**: Internal Clock.
    - Set **Prescaler** and **Counter Period** to generate 1 Hz interrupts (1 second).
    - Enable **Interrupt** for TIM2.
    - Generate initialization code.
3. **Code Implementation:**
    - Initialize TIM2 using MX_TIM2_Init().
    - Start TIM2 in interrupt mode using HAL_TIM_Base_Start_IT().
    - Implement the callback function HAL_TIM_PeriodElapsedCallback() to toggle LED.
4. **Testing:**
    - Verify that LED toggles at a fixed interval without blocking the CPU.

Code Documentation

```
/* USER CODE BEGIN Includes */
#include "main.h"
/* USER CODE END Includes */

TIM_HandleTypeDef htim2;

/* ------------------ Main Code ------------------ */
int main(void)
{
  HAL_Init();
```

```
  SystemClock_Config();
  MX_GPIO_Init();
  MX_TIM2_Init();

  // Start Timer in interrupt mode
  HAL_TIM_Base_Start_IT(&htim2);

  while (1)
  {
    // Main loop is free for other tasks
    // LED toggling is handled in Timer ISR
  }
}

/* ---------------- Timer Configuration -----------------*/
void MX_TIM2_Init(void)
{
  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 63999;   // Prescaler
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 999;        // ARR value
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  HAL_TIM_Base_Init(&htim2);

  HAL_NVIC_SetPriority(TIM2_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(TIM2_IRQn);
}

/* ----------------- Timer Callback ----------------- */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  if (htim->Instance == TIM2)
  {
    HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);  // Toggle LED (PC13)
  }
}

/* ----------------- Interrupt Handler ----------------- */
void TIM2_IRQHandler(void)
{
  HAL_TIM_IRQHandler(&htim2);
}
```

Explanation of Code

- **Prescaler = 63999, Period = 999** → With 64 MHz clock, timer generates 1-second interrupt.
- HAL_TIM_Base_Start_IT() → Starts TIM2 in interrupt mode.
- HAL_TIM_PeriodElapsedCallback() → Callback called automatically on each timer overflow.
- Inside the callback, the **LED toggles** every second.
- Main loop remains free for other tasks (non-blocking execution).

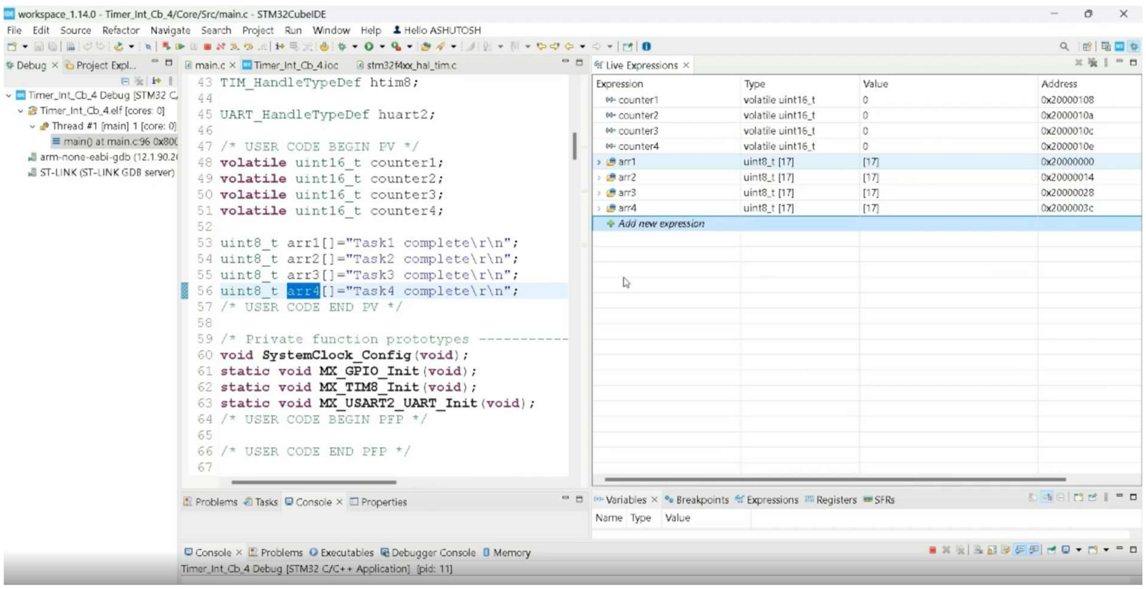
Challenges Faced

- Selecting correct **prescaler and period values** to achieve accurate timing.
- Forgetting to enable **NVIC interrupt for TIM2** initially caused the callback not to execute.

- Debugging required confirming system clock frequency, as different boards run at different speeds.

Output / Result

- The onboard LED toggled at exactly 1-second intervals.
- Timing was accurate and independent of CPU load.
- This demonstrated successful configuration and usage of **interrupt-driven timers** in STM32.



Conclusion

This task introduced **hardware timers and interrupts** in STM32. By using **TIM2 in interrupt mode**, the LED was toggled without blocking the main loop, showcasing the advantages of event-driven embedded programming. This knowledge is essential for applications requiring **real-time scheduling**, such as motor control, sensor sampling, and communication protocols.

Task 6: Servo Motor with STM32

Introduction

Servo motors are widely used in robotics, automation, and control systems because of their ability to rotate to a precise angular position based on control signals. A servo motor typically requires a **PWM signal with a 20 ms period (50 Hz)**, where the **pulse width (1 ms–2 ms)** determines the angle of rotation.

In this task, the STM32 microcontroller generates a PWM signal using its **Timer (TIMx)** to control the servo motor. By adjusting the PWM duty cycle, the servo motor rotates between **0° (1 ms pulse), 90° (1.5 ms pulse), and 180° (2 ms pulse)**. This experiment demonstrates the application of STM32 timers in real-world actuator control.

Objective

- To configure STM32 Timer in **PWM mode**.
- To generate a 50 Hz PWM signal for controlling a servo motor.
- To vary duty cycle to rotate the servo motor clockwise, counterclockwise, and to neutral positions.
- To understand PWM-based actuator control.

Methodology / Approach

1. **Hardware Setup:**
   - Servo motor (e.g., SG90, MG995).
   - Servo connections:
     - Red → +5 V
     - Brown/Black → GND
     - Yellow/Orange → STM32 PWM pin (e.g., PA0 with TIM2_CH1).
   - External 5 V supply recommended if servo draws more current.
2. **CubeMX Configuration:**
   - Enable **TIM2** → PWM Generation → Channel 1.
   - Set Prescaler and Period to achieve **50 Hz frequency**.
     - Example: APB1 Timer Clock = 72 MHz.
     - Prescaler = 71 → Timer clock = 1 MHz.
     - Period = 20000 – 1 = 19999 → 20 ms (50 Hz).
   - Configure PA0 as **TIM2_CH1 (Alternate Function Push-Pull)**.
3. **Code Implementation:**
   - Start PWM with HAL_TIM_PWM_Start().
   - Use __HAL_TIM_SET_COMPARE() to adjust duty cycle.
   - Pulse values:
     - ~1000 (1 ms) → 0°
     - ~1500 (1.5 ms) → 90°
     - ~2000 (2 ms) → 180°
4. **Testing:**
   - Observe servo shaft rotating to different angles as PWM values change.

Code Documentation

```
/* USER CODE BEGIN Includes */
#include "main.h"
/* USER CODE END Includes */

TIM_HandleTypeDef htim2;  // Timer 2 handle

int main(void)
{
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();
  MX_TIM2_Init();

  // Start PWM on TIM2 channel 1
  HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);

  while (1)
  {
    // Rotate servo to 0 degree (1 ms pulse)
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, 1000);
    HAL_Delay(1000);

    // Rotate servo to 90 degree (1.5 ms pulse)
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, 1500);
    HAL_Delay(1000);

    // Rotate servo to 180 degree (2 ms pulse)
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, 2000);
    HAL_Delay(1000);
  }
}

/* ---------------- Timer Configuration -----------------*/
void MX_TIM2_Init(void)
{
  TIM_OC_InitTypeDef sConfigOC = {0};

  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 71;         // 72MHz/72 = 1MHz (1 µs tick)
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 20000 - 1;     // 20 ms period = 50 Hz
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  HAL_TIM_PWM_Init(&htim2);

  sConfigOC.OCMode = TIM_OCMODE_PWM1;
  sConfigOC.Pulse = 1500;            // Default 1.5 ms pulse (90° position)
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1);

  HAL_TIM_MspPostInit(&htim2);
}
```

Explanation of Code

- **Prescaler = 71, Period = 19999** → Creates 50 Hz PWM (20 ms period).
- HAL_TIM_PWM_Start() → Starts PWM generation.
- __HAL_TIM_SET_COMPARE() → Sets duty cycle (pulse width in µs).

- Values (1000 → 1 ms, 1500 → 1.5 ms, 2000 → 2 ms) control servo angle.

Challenges Faced

- Servos often need more current than STM32's 5 V pin can provide → required **external 5 V power supply**.
- Incorrect PWM frequency (not exactly 50 Hz) caused jitter in servo movement.
- Common GND between STM32 and external servo supply was necessary for stable operation.

Output / Result

- Servo successfully rotated to **0°, 90°, and 180°** positions.
- PWM duty cycle adjustments directly controlled servo shaft angle.
- The task verified the STM32 timer's ability to generate precise PWM signals for actuator control.

Conclusion

This task demonstrated the use of STM32 timers in **PWM mode** to control a servo motor. By varying the duty cycle of a 50 Hz PWM signal, the servo could be rotated to precise angular positions. This experiment provided valuable knowledge in actuator interfacing, motion control, and embedded PWM generation—key elements in robotics, automation, and mechatronics applications.

Task 7: UART Communication Protocol

Introduction

Universal Asynchronous Receiver/Transmitter (**UART**) is one of the most widely used communication interfaces in embedded systems. It enables reliable serial data transfer between microcontrollers, PCs, and external modules such as sensors, GPS, or wireless adapters. Unlike GPIO-based I/O, UART allows structured bidirectional communication at defined baud rates, making it ideal for debugging, command exchange, and protocol implementation.

In this task, STM32 is configured to communicate with an external device (PC via USB-to-Serial or another microcontroller) using **USART2**. A custom **framed communication protocol** is implemented, consisting of header bytes, length, command identifier, payload, and CRC for error detection. The STM32 receives commands, processes them, and sends structured responses.

Objective

- To configure **USART2** on STM32 for UART communication.
- To implement a **framed packet protocol** with header, length, payload, and CRC.
- To receive commands via UART, process them, and send responses.
- To demonstrate interrupt-driven UART reception, making the system **non-blocking**.
- To implement multiple commands (LED control, ADC reading, echo, firmware version).

Methodology / Approach

1. **Hardware Setup**
    - STM32 board (e.g., BluePill or Nucleo).
    - USB-to-Serial adapter connected to **USART2 (PA2 = TX, PA3 = RX)**.
    - LED connected to PC13 (onboard LED).
    - Potentiometer connected to PA0 for ADC reading.
    - PC serial terminal (e.g., PuTTY, RealTerm) for testing.
2. **Protocol Design**
    - **Header:** 0x55 0xAA
    - **Length:** Number of bytes following (Cmd + Payload + CRC).
    - **Command:** Command ID (1 byte).
    - **Payload:** Data (variable length).
    - **CRC:** CRC8 over (Cmd + Payload).

    Example:
    [0x55 0xAA 0x02 0x02 CRC] → Request ADC value.

3. **Command Set**
    - **0x01 – LED Control** (payload: 1 byte → ON/OFF/TOGGLE).
    - **0x02 – Read ADC** (payload: none → response with 2 bytes ADC value).
    - **0x03 – Echo** (payload: arbitrary → response same payload).
    - **0x04 – Get Version** (payload: none → response string).
    - **0xFF – Error Response** (payload: error code).

4. **Software Implementation**
   - CubeMX used to configure **USART2** and **ADC1**.
   - UART interrupt (HAL_UART_RxCpltCallback) receives data byte-by-byte.
   - A **state machine** assembles packets.
   - On complete valid frame, process_frame() executes the command and replies.

Code Documentation

Below is the **main application code** (long and detailed).

```
/* main.c - UART Communication Protocol Example using STM32 HAL */

#include "main.h"
#include "string.h"
#include "stdio.h"

UART_HandleTypeDef huart2;
ADC_HandleTypeDef hadc1;

/* ------------------ Protocol Defines ------------------ */
#define FRAME_HDR0 0x55
#define FRAME_HDR1 0xAA
#define MAX_PAYLOAD_SIZE 64
#define RX_PACKET_MAX (2 + 1 + 1 + MAX_PAYLOAD_SIZE + 1)

/* Command IDs */
#define CMD_LED_CONTROL  0x01
#define CMD_READ_ADC0x02
#define CMD_ECHO         0x03
#define CMD_GET_VERSION  0x04
#define CMD_ERROR        0xFF

/* Error Codes */
#define ERR_BAD_CRC      0x01
#define ERR_BAD_LEN      0x02
#define ERR_UNKNOWN_CMD  0x03

/* LED Pin */
#define LED_GPIO_Port GPIOC
#define LED_Pin       GPIO_PIN_13

/* Version String */
const char fw_version[] = "AITRONICS v1.0";

/* ------------------ RX Variables ------------------ */
static uint8_t rx_byte;
static uint8_t rx_buf[RX_PACKET_MAX];
static uint16_t rx_index = 0;
static uint16_t expected_len = 0;
static uint8_t receiving_frame = 0;

/* ------------------ Utility Functions ------------------ */
static uint8_t crc8_compute(const uint8_t *data, uint16_t len) {
  uint8_t crc = 0;
  for (uint16_t i = 0; i < len; i++) {
    crc ^= data[i];
    for (uint8_t b = 0; b < 8; b++) {
      if (crc & 0x80) crc = (crc << 1) ^ 0x07;
```

```c
      else crc <<= 1;
    }
  }
  return crc;
}

static void send_packet(uint8_t cmd, const uint8_t *payload, uint8_t payload_len) {
  uint8_t len_field = 1 + payload_len + 1; // Cmd + Payload + CRC
  uint8_t frame[RX_PACKET_MAX];
  uint16_t idx = 0;

  frame[idx++] = FRAME_HDR0;
  frame[idx++] = FRAME_HDR1;
  frame[idx++] = len_field;
  frame[idx++] = cmd;

  if (payload_len && payload) {
    memcpy(&frame[idx], payload, payload_len);
    idx += payload_len;
  }

  uint8_t crc = crc8_compute(&frame[3], 1 + payload_len);
  frame[idx++] = crc;

  HAL_UART_Transmit(&huart2, frame, idx, HAL_MAX_DELAY);
}

/* ---------------- Command Processor -----------------*/
static void process_frame(uint16_t frame_len) {
  uint8_t len_field = rx_buf[2];
  uint8_t cmd = rx_buf[3];
  uint8_t payload_len = len_field - 2; // excluding cmd and crc
  uint8_t *payload = &rx_buf[4];
  uint8_t crc_recv = rx_buf[frame_len - 1];
  uint8_t crc_calc = crc8_compute(&rx_buf[3], 1 + payload_len);

  if (crc_calc != crc_recv) {
    uint8_t err = ERR_BAD_CRC;
    send_packet(CMD_ERROR, &err, 1);
    return;
  }

  switch (cmd) {
    case CMD_LED_CONTROL: {
      if (payload_len < 1) return;
      uint8_t mode = payload[0];
      if (mode == 0) HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_SET);
      else if (mode == 1) HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_RESET);
      else if (mode == 2) HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
      send_packet(CMD_LED_CONTROL, payload, 1);
      break;
    }
    case CMD_READ_ADC: {
      HAL_ADC_Start(&hadc1);
      HAL_ADC_PollForConversion(&hadc1, 50);
      uint32_t value = HAL_ADC_GetValue(&hadc1);
      uint8_t resp[2] = { (value >> 8) & 0xFF, value & 0xFF };
      send_packet(CMD_READ_ADC, resp, 2);
      break;
    }
```

```
    case CMD_ECHO: {
      send_packet(CMD_ECHO, payload, payload_len);
      break;
    }
    case CMD_GET_VERSION: {
      send_packet(CMD_GET_VERSION, (const uint8_t*)fw_version, strlen(fw_version));
      break;
    }
    default: {
      uint8_t err = ERR_UNKNOWN_CMD;
      send_packet(CMD_ERROR, &err, 1);
      break;
    }
  }
}

/* ---------------- UART RX State Machine ----------------- */
static void rx_state_machine(uint8_t b) {
  if (!receiving_frame) {
    if (rx_index == 0 && b == FRAME_HDR0) {
      rx_buf[rx_index++] = b;
      return;
    } else if (rx_index == 1 && b == FRAME_HDR1) {
      rx_buf[rx_index++] = b;
      receiving_frame = 1;
      return;
    } else {
      rx_index = 0;
      return;
    }
  } else {
    rx_buf[rx_index++] = b;
    if (rx_index == 3) {
      expected_len = 2 + 1 + rx_buf[2];
    }
    if (expected_len && rx_index >= expected_len) {
      process_frame(rx_index);
      rx_index = 0;
      expected_len = 0;
      receiving_frame = 0;
    }
  }
}

/* ------------------ UART Callback ------------------ */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
  if (huart->Instance == USART2) {
    rx_state_machine(rx_byte);
    HAL_UART_Receive_IT(&huart2, &rx_byte, 1);
  }
}

/* ------------------ Main Function ------------------ */
int main(void) {
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_ADC1_Init();
```

```
  HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_SET);
  HAL_UART_Receive_IT(&huart2, &rx_byte, 1);

  while (1) {
    // Main loop free for other tasks
    HAL_Delay(1000);
  }
}
```

Challenges Faced

- **Header synchronization:** Random UART noise could confuse the state machine; solved by using a fixed 2-byte header.
- **CRC mismatches:** CRC errors occurred until the correct polynomial (0x07) was implemented.
- **LED logic:** Some STM32 boards (BluePill) use active-low LEDs, requiring careful ON/OFF logic.
- **ADC stability:** Values fluctuated due to noise; averaging could be used in future improvements.

Output / Result

- Successful **bidirectional communication** between STM32 and PC.
- LED control verified via UART commands.
- Potentiometer ADC values correctly displayed on terminal.
- Echo and version commands confirmed reliable packet handling.

Conclusion

This task implemented a **robust UART communication protocol** on STM32. Using a framed packet format with CRC, the microcontroller was able to process structured commands and respond with accurate data. The use of **interrupt-driven UART reception** ensured non-blocking operation, making the system scalable for real-time applications. This knowledge is directly applicable to IoT devices, embedded communication interfaces, and system debugging tools.

5. Final Conclusion

The successful completion of this project has provided extensive hands-on experience in designing, implementing, and debugging embedded systems applications using the STM32 microcontroller. Each of the seven tasks progressively introduced new peripherals, concepts, and programming techniques, building a strong foundation for real-world embedded system development.

Key outcomes and learnings from this project include:

1. **GPIO Control (Task 1 & 2)**
   - Learned to configure GPIO pins as inputs/outputs and drive external devices such as LEDs and seven-segment displays.
   - Understood the importance of logic levels (active-high vs. active-low) and multiplexing techniques for efficient output control.
2. **Display Interfacing (Task 3)**
   - Gained practical knowledge of LCD communication in 4-bit mode, including command/data separation and custom character creation.
   - Developed the ability to implement modular drivers for peripherals, which improves reusability in larger projects.
3. **Analog-to-Digital Conversion (Task 4)**
   - Explored the role of ADCs in acquiring real-world signals and converting them to digital values.
   - Applied mathematical scaling to convert raw ADC readings into human-readable voltage levels and displayed them on LCD.
4. **Timers and Interrupts (Task 5)**
   - Understood how hardware timers work and how interrupts can be used for time-critical tasks without blocking the main loop.
   - Strengthened knowledge of event-driven programming, which is essential for real-time systems.
5. **PWM and Actuator Control (Task 6)**
   - Implemented PWM signals to control servo motors, learning about precise timing and duty-cycle adjustments.
   - Understood how embedded systems interact with actuators, a key requirement in robotics and automation.
6. **Communication Protocols (Task 7)**
   - Designed and implemented a framed UART communication protocol with error detection (CRC8), demonstrating structured data exchange.
   - Learned to use interrupt-driven UART for non-blocking communication, making the system scalable and efficient.
7. **Debugging and Integration**
   - Developed problem-solving skills through debugging hardware issues (e.g., wiring, power supply) and software issues (timing, logic errors, interrupt handling).
   - Learned the importance of modular coding practices and structured report documentation.

Overall Learning

This project has not only strengthened theoretical understanding but also transformed it into practical skills applicable in real-world embedded systems design. By the end of the seven tasks, the following competencies were achieved:

- Proficiency in using **STM32CubeIDE** and HAL libraries.
- Confidence in configuring and utilizing STM32 peripherals (GPIO, ADC, Timers, PWM, UART).
- Ability to design and implement structured communication protocols.
- Improved debugging, documentation, and teamwork skills.

In conclusion, the project served as a comprehensive introduction to embedded systems with STM32, preparing for more advanced projects such as IoT devices, robotics systems, biomedical instrumentation, and industrial automation. The skills gained here will form a solid foundation for future academic research and professional applications in embedded system development.

6. References

(Option A: IEEE Style)

[1] STMicroelectronics, *STM32F103xx Datasheet: Medium-density performance line ARM®-based 32-bit MCU*, 2016. [Online]. Available: https://www.st.com

[2] STMicroelectronics, *STM32F1 Series Reference Manual (RM0008)*, 2018. [Online]. Available: https://www.st.com

[3] STMicroelectronics, *STM32CubeIDE User Guide*, 2021. [Online]. Available: https://www.st.com

[4] HD44780 LCD Controller, *Hitachi HD44780U LCD Driver Datasheet*, 1998.

[5] "STM32 HAL API Reference Manual," STMicroelectronics, 2020.

[6] N. S. Kumar, P. Senthilkumar, S. Arun, and R. Jeevananthan, *Microcontrollers and Embedded System Design*, Oxford University Press, 2012.

[7] M. A. Mazidi, S. Naimi, and S. Naimi, *ARM Cortex-M Microcontroller Programming for Embedded Systems*, MicroDigitalEd, 2017.

[8] TutorialsPoint, *STM32 Microcontroller Basics – Online Resource*. [Online]. Available: https://www.tutorialspoint.com/stm32