KING MONGKUT'S INSTITUTE OF TECHNOLOGY LATKRABANG
SCHOOL OF ENGINEERING
DEPARTMENT OF ROBOTICS AND AI ENGINEERING



01416318 - MICROPROCESSOR AND INTERFACE
FINAL PROJECT REPORT

INSTRUCTED BY: DR POOM KONGKHUATROB
DR SARUCHA YANYONG

NAME: NAPON, CHINDANANI, KANACHOK, POONYAPHAT, LUCKY, TANADOL
STUDENT ID: 67011179, 67011098, 67011604, 67011275, 67011698, 67110134
GROUP NO: 8
DATE OF SUB.: 4/4/2025

1. Abstract

This project involves controlling a BLDC motor using the MODBUS protocol over an RS485 interface, with an STM32 microcontroller as the core controller. The MODBUS communication allows for speed control of the motor, while real-time data such as target speed, current speed, and motor angle is displayed on an OLED screen. An emergency stop button is integrated for immediate motor shutdown to enhance safety. A custom PCB was designed to interconnect all components, and a 3D-printed base was created to house the system.

2. Hardware Design

Our hardware design of our system is focused on delivering a miniaturized, organized, and operational physical structure to accommodate and protect all the essential electrical elements that make up the BLDC motor control system. The design is organized into two separate halves: the Base Holder and the Top Holder, both of which are custom-designed and are employed for various reasons in accommodating, protecting, and interfacing various system components.

2.1. Base Holder Design

Base Holder is a solid, box-shaped base on which the main enclosure for core electronic parts sits. It is designed to support and deal with essential items like the STM32 microcontroller board, MAX485 module, USB to RS-485, PCB, FOC driver shield, and internal wiring. The base provides protection to all devices and securely mounts them in position. Additionally, specific cutouts and wire channels are incorporated in the base to facilitate clean cable management, easy wiring access, and a well-organized internal structure. The structural design is such that it provides electrical stability and maintenance simplicity.

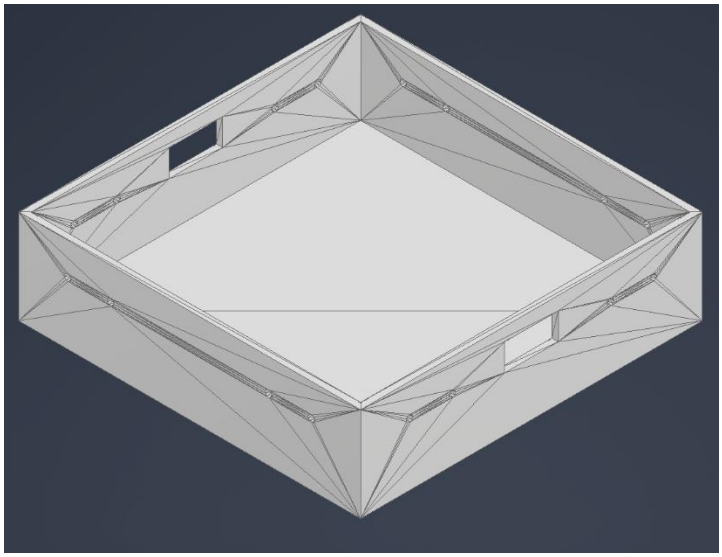


Figure 1 Base Holder Design

2.2. Top Holder Design

The Top Holder is both a structural and interactive part. Likewise box-shaped, it is mounted atop the base and serves as the ceiling for the Base Holder. The most prominent feature of this section is its mounting point for the BLDC motor, whose placement has been taken with care to allow steady motor alignment and operation. The Top Holder also contains holes placed strategically for wire tracing, such that motor, OLED display, and control circuitry connections are secure and well-arranged.

Moreover, it has a dedicated hole aligned with the emergency stop button such that users can easily press the button with their finger in case of an emergency without opening the case. The OLED display is also mounted on the front surface of the Top Holder to provide the user with current data on motor state, speed, and angle. This arrangement enables top-class user interaction and monitoring without compromising on the inner design.

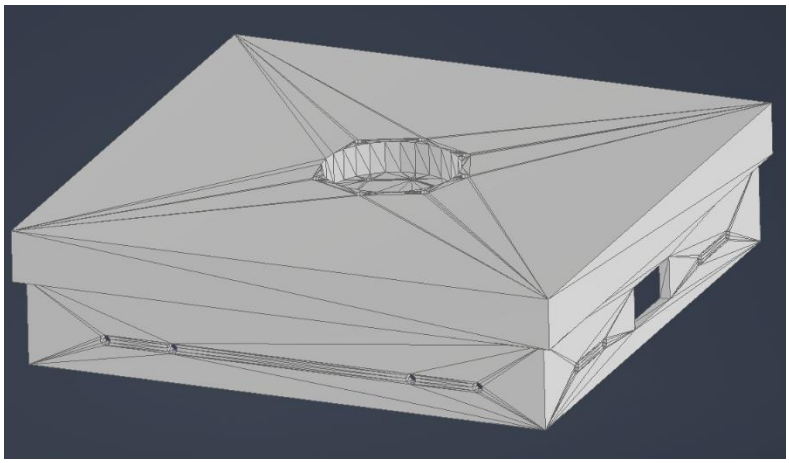


Figure 2 – Top Holder Design

2.3. PCB Design

We fabricated our own custom PCB by a manual PCB etching method. We first traced the PCB layout on a copper board using ironing a printed design. We then manually corrected any missing connections by chipping and using a permanent marker to repair broken traces. After fixing it, we put the board into a chemical bath to etch away the excess copper, so that only the circuit traces would be left behind. This allowed us to house all the hardware components into a miniature and organized system.

2.3.1. Schematic Overview

Our PCB schematic is designed around the STM32F401RET6 microcontroller, which serves as the core processing unit. The circuit integrates multiple essential components:

- **MAX485 Module:** This enables RS485 communication, enabling reliable data transmission between the STM32 and external devices such as the PC for MODBUS communication. The TX, DE, and RO pins are properly connected to the STM32 for data handling.

- **OLED Display:** Connected via the I2C interface (SCL and SDA), this display shows real-time information such as motor speed, angular position, and emergency stop status.
- **BLDC Motor Driver Interface:** The STM32 is interfaced with a BLDC driver module using PWM and direction control signals to provides precise Field-Oriented Control (FOC) of the motor.
- **Push Buttons**: A switch circuit which directly connected with the microcontroller to serve various different functions.
- **LED Indicators**: Two LEDs are included to provide visual status indicators, which can signal various states like power-on, communication activity, or errors.
- **Encoder:** Connected via the SPI interface to provide the real time information on the motor's angular position and velocity.
- **Variable Resistor:** Can be used for various applications such as setting the speed of the motor, tuning the PID values etc.

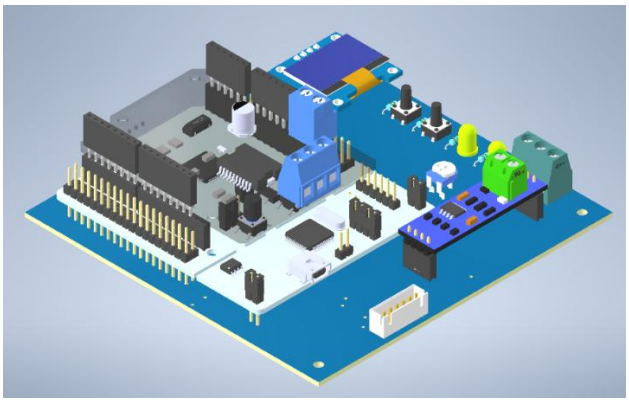


Figure 3 – PCB Design

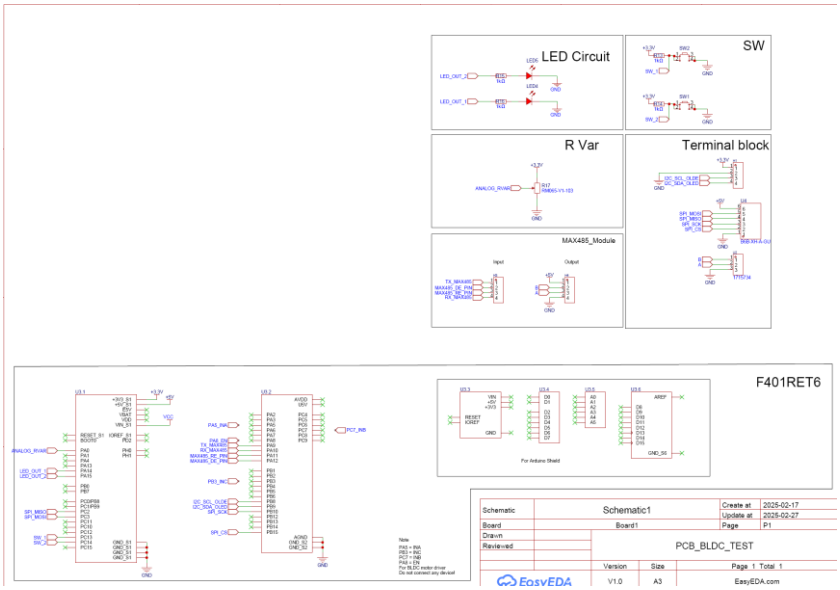


Figure 4 – PCB Schematic

3. Software Design

```
#include "main.h"
#include "bldc_driver.h"
#include "bldc_motor.h"
#include "as5048a.h"
#include "ssd1306.h"
#include "ssd1306_fonts.h"
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
unsigned long timer_counter = 0;
float speed;
I2C_HandleTypeDef hi2c1;

SPI_HandleTypeDef hspi2;

TIM_HandleTypeDef htim1;
TIM_HandleTypeDef htim2;
TIM_HandleTypeDef htim3;
TIM_HandleTypeDef htim4;

UART_HandleTypeDef huart1;
UART_HandleTypeDef huart2;


char uart_buffer[128];
float p;
uint8_t RxData[8];
uint8_t TxData[8];
uint16_t data;
char dta[100];
char s[100];
char v[100];
float step = 1.0;
float f_spd;

bool is_stopping = false;
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_TIM2_Init(void);
static void MX_TIM3_Init(void);
static void MX_SPI2_Init(void);
static void MX_TIM4_Init(void);
static void MX_TIM1_Init(void);
static void MX_I2C1_Init(void);
static void MX_USART1_UART_Init(void);

/* USER CODE BEGIN 0 */
int ButtonPressed(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin) {
   if (HAL_GPIO_ReadPin(GPIOx, GPIO_Pin) == GPIO_PIN_RESET) {
```

```
        HAL_Delay(50);
        if (HAL_GPIO_ReadPin(GPIOx, GPIO_Pin) == GPIO_PIN_RESET) {
            return 1;
        }
    }
    return 0;
}

void MAX485_EnableTransmit(){
    HAL_GPIO_WritePin(MAX485_DE_PIN_GPIO_Port,MAX485_DE_PIN_Pin,
GPIO_PIN_SET);
    HAL_GPIO_WritePin(MAX485_RE_PIN_GPIO_Port,MAX485_RE_PIN_Pin,
GPIO_PIN_SET);
}

void MAX485_EnableRecieve(){
    HAL_GPIO_WritePin(MAX485_DE_PIN_GPIO_Port,MAX485_DE_PIN_Pin,
GPIO_PIN_RESET);
    HAL_GPIO_WritePin(MAX485_RE_PIN_GPIO_Port,MAX485_RE_PIN_Pin,
GPIO_PIN_RESET);
}

void HAL_UARTEx_RxEventCallback(UART_HandleTypeDef *huart, uint16_t
Size) {
            memcpy(TxData, RxData, Size);
            MAX485_EnableTransmit();
            HAL_UART_Transmit_IT(&huart1, TxData,8);

            data = RxData[4] << 8 | RxData[5];
    }

void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart) {
    if (huart->Instance == USART1) {
        MAX485_EnableRecieve();
        HAL_UARTEx_ReceiveToIdle_IT(&huart1, RxData, sizeof(RxData));
    }
}

int main(void)
{
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_TIM2_Init();
  MX_TIM3_Init();
  MX_SPI2_Init();
  MX_TIM4_Init();
  MX_TIM1_Init();
  MX_I2C1_Init();
  MX_USART1_UART_Init();
  /* USER CODE BEGIN 2 */
     MAX485_EnableRecieve();
```

```
      HAL_UARTEx_ReceiveToIdle_IT(&huart1, RxData, 8);
      ssd1306_Init();
      ssd1306_Fill(Black);


  as5048a_init(&hspi2, SPI_CS_GPIO_Port, SPI_CS_Pin);

  driver_init(&htim2,
              TIM_CHANNEL_1,
              &htim3,
              TIM_CHANNEL_2,
              &htim2,
              TIM_CHANNEL_2,
              FOC_EN_GPIO_Port,
              FOC_EN_Pin,
              20.0f,
              20.0f);
  bldc_vel_ctrl_init(7,
                     5.0f,
                     5.0f,
                     (struct lpf_s){.tf = 0.00001f},
                     (struct pid_s){
                       .p = 5.0f,
                       .i = 30.0f,
                       .d = 0.0f,
                       .ramp = 3000.0f,
                       .limit = 20.0f
                     },
                     20);


   foc_init();

    HAL_TIM_Base_Start(&htim1);
  HAL_TIM_Base_Start_IT(&htim4);

  /* USER CODE END 2 */

  while (1)
{
      if (is_stopping) {
            ssd1306_Fill(Black);
            ssd1306_SetCursor(10,10);
            ssd1306_WriteString("EMERGENCY" , Font_11x18, White);
            ssd1306_SetCursor(10,40);
            ssd1306_WriteString("STOP" , Font_11x18, White);
            ssd1306_UpdateScreen();
      }

      else {
              ssd1306_Fill(Black);
              sprintf(v , "VEL: %.2f rad/s" , get_vel());
              sprintf(s , "ANG: %.2f rad", get_angle());
```

```
            sprintf(dta, "VEL_TARGET: %d" , data);
            ssd1306_SetCursor(10,10);
            ssd1306_WriteString(s , Font_7x10, White);
            ssd1306_SetCursor(10,30);
            ssd1306_WriteString(v, Font_7x10, White);
            ssd1306_SetCursor(10,50);
            ssd1306_WriteString(dta, Font_7x10, White);
            ssd1306_UpdateScreen();
        }

        f_spd = (float) data;

        if (ButtonPressed(GPIOC, GPIO_PIN_13)) {
             speed = 0.0f;
             is_stopping = !is_stopping;
        }


        while (speed != f_spd && !is_stopping) {

            if (speed < f_spd) {

                speed += step;

                if (speed > f_spd)
                  speed = f_spd;

            }

            else if (speed > f_spd) {

                speed -= step;
                if (speed < data)
                  speed = f_spd;

            }

            HAL_Delay(100);
        }
    }
}

void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
  __HAL_RCC_PWR_CLK_ENABLE();
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE2);
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
```

```
  RCC_OscInitStruct.PLL.PLLM = 16;
  RCC_OscInitStruct.PLL.PLLN = 336;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV4;
  RCC_OscInitStruct.PLL.PLLQ = 7;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

   RCC_ClkInitStruct.ClockType =
RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) !=
HAL_OK)
  {
    Error_Handler();
  }
}
static void MX_I2C1_Init(void)
{

  hi2c1.Instance = I2C1;
  hi2c1.Init.ClockSpeed = 100000;
  hi2c1.Init.DutyCycle = I2C_DUTYCYCLE_2;
  hi2c1.Init.OwnAddress1 = 0;
  hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
  hi2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
  hi2c1.Init.OwnAddress2 = 0;
  hi2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
  hi2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
  if (HAL_I2C_Init(&hi2c1) != HAL_OK)
  {
    Error_Handler();
  }

}

static void MX_SPI2_Init(void)
{
  hspi2.Instance = SPI2;
  hspi2.Init.Mode = SPI_MODE_MASTER;
  hspi2.Init.Direction = SPI_DIRECTION_2LINES;
  hspi2.Init.DataSize = SPI_DATASIZE_16BIT;
  hspi2.Init.CLKPolarity = SPI_POLARITY_LOW;
  hspi2.Init.CLKPhase = SPI_PHASE_2EDGE;
  hspi2.Init.NSS = SPI_NSS_SOFT;
  hspi2.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_2;
  hspi2.Init.FirstBit = SPI_FIRSTBIT_MSB;
```

```
  hspi2.Init.TIMode = SPI_TIMODE_DISABLE;
  hspi2.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
  hspi2.Init.CRCPolynomial = 10;
  if (HAL_SPI_Init(&hspi2) != HAL_OK)
  {
    Error_Handler();
  }
}

static void MX_TIM1_Init(void)
{

   TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};
  htim1.Instance = TIM1;
  htim1.Init.Prescaler = 84-1;
  htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim1.Init.Period = 0xffff-1;
  htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim1.Init.RepetitionCounter = 0;
  htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim1) != HAL_OK)
  {
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim1, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig) !=
HAL_OK)
  {
    Error_Handler();
  }
}

static void MX_TIM2_Init(void)
{

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};
  TIM_OC_InitTypeDef sConfigOC = {0};


  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 84-1;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 40-1;
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
```

```
  if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
  {
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_TIM_PWM_Init(&htim2) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) !=
HAL_OK)
  {
    Error_Handler();
  }
  sConfigOC.OCMode = TIM_OCMODE_PWM1;
  sConfigOC.Pulse = 0;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1) !=
HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_2) !=
HAL_OK)
  {
    Error_Handler();
  }
  HAL_TIM_MspPostInit(&htim2);

}

static void MX_TIM3_Init(void)
{

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};
  TIM_OC_InitTypeDef sConfigOC = {0};
  htim3.Instance = TIM3;
  htim3.Init.Prescaler = 84-1;
  htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim3.Init.Period = 40-1;
  htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
  {
    Error_Handler();
```

```
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_TIM_PWM_Init(&htim3) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) !=
HAL_OK)
  {
    Error_Handler();
  }
  sConfigOC.OCMode = TIM_OCMODE_PWM1;
  sConfigOC.Pulse = 0;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_PWM_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_2) !=
HAL_OK)
  {
    Error_Handler();
  }
  HAL_TIM_MspPostInit(&htim3);

}

static void MX_TIM4_Init(void)
{

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  htim4.Instance = TIM4;
  htim4.Init.Prescaler = 84-1;
  htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim4.Init.Period = 400-1;
  htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim4.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim4) != HAL_OK)
  {
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim4, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
```

```
  if (HAL_TIMEx_MasterConfigSynchronization(&htim4, &sMasterConfig) !=
HAL_OK)
  {
    Error_Handler();
  }
 }

static void MX_USART1_UART_Init(void)
{
  huart1.Instance = USART1;
  huart1.Init.BaudRate = 9600;
  huart1.Init.WordLength = UART_WORDLENGTH_8B;
  huart1.Init.StopBits = UART_STOPBITS_1;
  huart1.Init.Parity = UART_PARITY_NONE;
  huart1.Init.Mode = UART_MODE_TX_RX;
  huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart1.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart1) != HAL_OK)
  {
    Error_Handler();
  }
}

static void MX_USART2_UART_Init(void)
{
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 115200;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }
}

static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
/   __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

   HAL_GPIO_WritePin(SPI_CS_GPIO_Port, SPI_CS_Pin, GPIO_PIN_SET);

   HAL_GPIO_WritePin(GPIOA,
FOC_EN_Pin|MAX485_RE_PIN_Pin|MAX485_DE_PIN_Pin|GPIO_PIN_14
                          |GPIO_PIN_15, GPIO_PIN_RESET);
```

```
  GPIO_InitStruct.Pin = GPIO_PIN_13;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  GPIO_InitStruct.Pull = GPIO_PULLUP;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

  GPIO_InitStruct.Pin = GPIO_PIN_14;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

  GPIO_InitStruct.Pin = SPI_CS_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(SPI_CS_GPIO_Port, &GPIO_InitStruct);


  GPIO_InitStruct.Pin =
FOC_EN_Pin|MAX485_RE_PIN_Pin|MAX485_DE_PIN_Pin|GPIO_PIN_14
                          |GPIO_PIN_15;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

}
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  if (htim->Instance == TIM4)
  {
    timer_counter = __HAL_TIM_GET_COUNTER(&htim1);
    __HAL_TIM_SET_COUNTER(&htim1, 0);

    const float dt_while = (float)timer_counter * 1e-6f;

    loop_foc();
    bldc_move(speed, dt_while);
  }

}
void Error_Handler(void)
{
  __disable_irq();
  while (1) { }
}

#ifdef  USE_FULL_ASSERT
void assert_failed(uint8_t *file, uint32_t line)
{
  }
#endif
```

The code works by first receiving the data of the MODBUS protocol over UART1 from the MAX485 Module via the RS485 interface. When the data is received, the values are stored in a buffer called `RxData`. An Interrupt Service Routine is called in which, the MAX485 is set to transmit mode and the data from the `RxData` varible is copied to the `TxData` variable and it is sent over UART1 back to the MODBUS application. Once the data is sent, another interrupt service routine is called which enables the receiving mode back and starts the receiving again. The speed value is extracted from the RxData by `data = RxData[4] << 8 | RxData[5];`. This line converts the 2 high and low 8-bit integer values of the speed data into a 16-bit integer value and stores it in the variable `Data`.

In the main function, first the MAX485 is set to receiving mode and the UART interrupt receive is called to get the data over MODBUS. Then the OLED display is initialized and set to fill the screen black. The motor's encoder is initialized with the SPI channel and the enable pin as parameters and the motor driver is also initialized with the timer channel for each phase of the motor and voltage supply and limit. After that, the closed loop control with the tuned PID values along with the field-oriented control is started and the timers 1 and 4 are started too.

In the while loop, the system can have 2 states. In the first state, the motor speed is set to 0 which cannot be modified by the MODBUS control and the OLED display will show "EMERGENCY STOP". This state is toggled on and off by the pushbutton at pin A13. In the other state, the target velocity and the current velocity in radians per second is displayed on the OLED alongside the current anglular position. The current angular position and the current speed values are taken from the motor encoder using the `get_vel()` and `get_angle()` functions respectively. The velocity of the motor is gradually increased or decreased to the target velocity by first comparing the speed value to the target value and if it is higher, then it is decremented by 1 in each loop until it reaches the target velocity. If it goes below the target velocity, then it is set to the target velocity and vice versa if the velocity is lower than the target velocity. A delay of 0.1s is added to ensure the motor speed change is smooth.

Finally a timer interrupt is used to set the speed to the motor. In this interrupt service routine, first timer 4 triggers interrupts at regular intervals, which are essential for controlling the motor speed. Upon each interrupt, Timer 1 tracks the elapsed time (dt) since the last interrupt, providing a precise time interval for motor speed adjustments.

The loop_foc() function implements Field-Oriented Control, updating the motor's control signals based on the rotor position and speed. The bldc_move() function uses the target speed (f_spd) and the elapsed time (dt) to increase or decrease the motor's speed, with smooth transitions. By updating the motor speed incrementally in small steps, the system avoids instant speed changes. This process repeats with each interrupt, making the motor's speed update continuously. The MODBUS communication and the Speed control logic are both shown in the figures below.

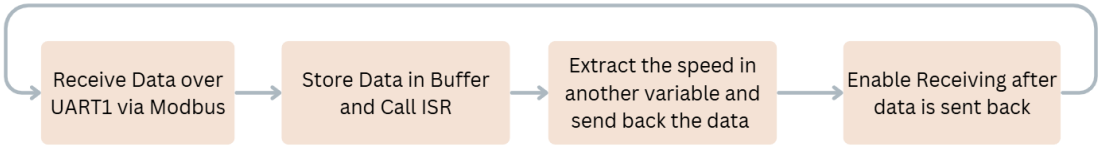


Figure 5 – MODBUS Communication

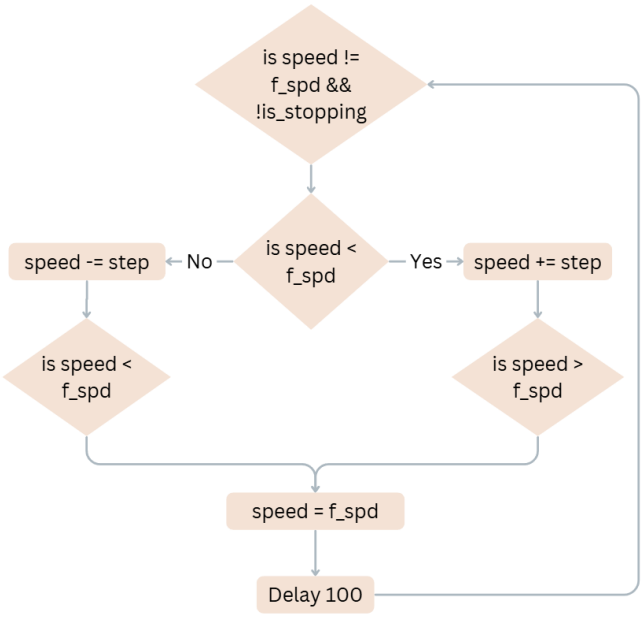


Figure 6 – Speed Control

4. Result

The results of our project confirm that the system is functioning as required with all components running smoothly and in an error-free manner. The BLDC motor was efficiently driven using the MODBUS protocol, enabling seamless and precise speed control through PC command instructions. The OLED screen accurately projected live information like the current motor angle, speed, and target speed for good user feedback. Furthermore, the emergency stop button worked properly, enabling instantaneous motor shutdown for enhanced operational safety. These outcomes validate the effective integration of both hardware and software components in our motor control system.



Figure 7 – Result Video

5. Application: Haptic Feedback-based Gear Selectors for Electric Vehicles

The application of Brushless DC (BLDC) motors in modern automotive systems has gained popularity due to their high efficiency, precise control, and fast response. One of the emerging applications is the application of BLDC motors in haptic feedback-based gear selectors for electric vehicles (EVs). Our design presents a BLDC motor-based gear selection system, wherein a rotary knob in the middle provides haptic feedback to the driver.

Our system uses a BLDC motor as a prime mover for a rotary gear selector. Instead of conventional mechanical linkages, the motor precisely commands the rotation of the knob such that the driver can seamlessly switch between driving modes (e.g., Park, Reverse, Neutral, Drive). The motor is programmed to give haptic feedback, which provides a physical indication to the driver of successful gear selection. It may be in terms of vibration signals, detents (click-like stops), or torque resistance.

The rotation and velocity of the BLDC motor are controlled by software such that the knob rotates within particular angular ranges based on different gear positions. The control system also communicates with the vehicle ECU (Electronic Control Unit), ensuring the gear position is correct, safe, and in coordination with other vehicle functions.

Using a BLDC Motor as a Gear Selector offers several unique advantages. Some of them include:

1. **Precise Control** – BLDC motors allow for precise placement of the gear shift knob, ensuring error-free choice.

2. **Rapid Response Time** – The motor quickly responds to driver inputs, providing instant feedback and improving the driving experience.

3. **Personalization** – The haptic feedback, angle of rotation, and torque resistance are software-controlled, allowing personal gear shift sensations for different cars or drivers.

4. **Better Durability** – Unlike mechanical gear levers, a BLDC motor-driven selector has fewer moving parts, with less wear and tear and increasing the system life.

5. **Integration with Vehicle Electronics** – It is possible to integrate the system with Advanced Driver-Assistance System (ADAS), allowing for gear selection on automation in autonomous driving or assist-driving modes.

Shown below is the 3D CAD of our model.

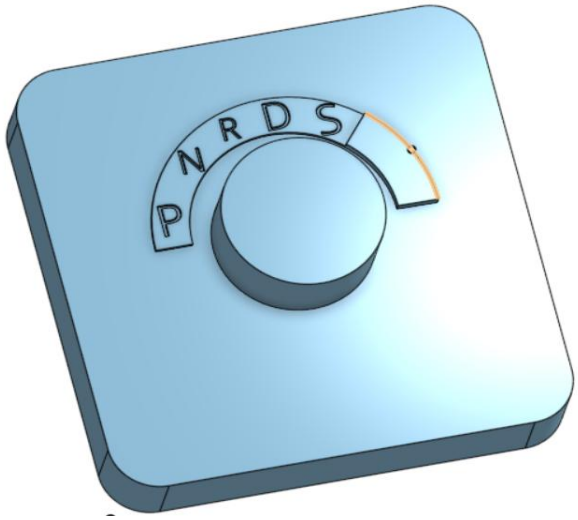


Figure 8 – Application Idea

6. Conclusion

In conclusion, our project was successfully able to achieve its primary objective of controlling a BLDC motor using a MODBUS protocol over an STM32 microcontroller and RS485 communication. Using the system, we were able to obtain the target motor speed from a PC, process it via UART interrupt-based communication, and smoothly control the motor speed with software-controlled acceleration and deceleration. Timer interrupt was also utilized to enable real-time motor control with FOC (Field Oriented Control) principles and precise timing through hardware timers.

We also enhanced system safety and usability by implementing an emergency stop button that stops the motor immediately when pressed and by displaying real-time data like the current angle, current speed, and target speed on an OLED display. Hardware design in the shape of a systematic top holder and base holder allowed for straightforward component mounting of components such as the STM32 board, MAX485 module, and motor, with space for user access via the emergency button and screen visibility.

7. Score Distribution

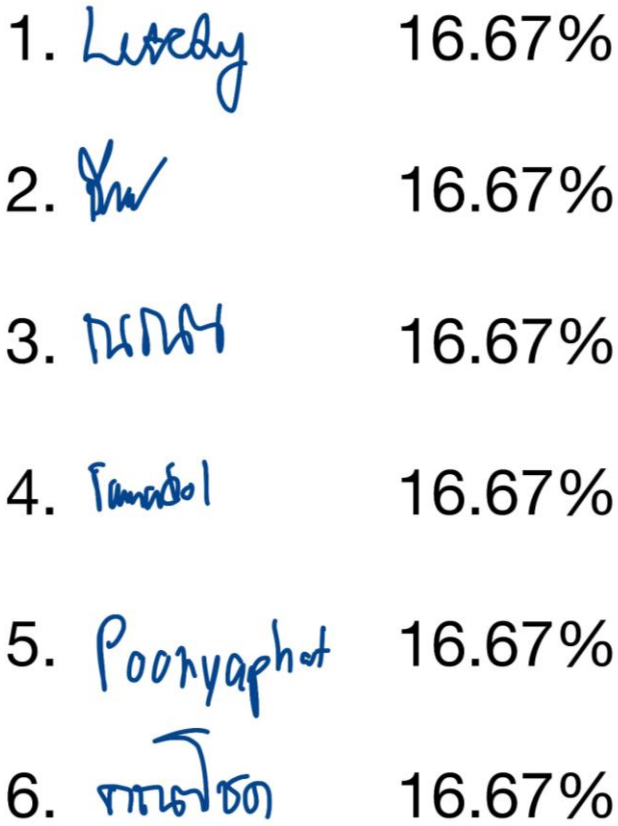


Figure 9 – Score Distribution