Fachhochschule Wedel

Embedded Systems Workshop

Implementation of Pong Game with STM32 Nucleo Board

Submitted by: Chou, Li-Chieh

Field of Study: IT Engineering

Matriculation Number: ITE104934

Semester of Study: 5

Contents

1. Introduction **1**
1.1. Project description 1
1.2. Project motivation 2
2. User manual **2**
2.1. Components 2
2.2. Getting started 4
2.3. Playing the game 4
2.4. Controlling the paddles 5
2.5. AI mode 5
2.6. Resetting the game 5
2.7. Powering off the game 5
3. Project analysis **5**
3.1. Modularization 5
3.2. Abstract / Algorithmic description 6
3.3. Analyzation 7
4. Implementation **9**
4.1. Specific description 9
4.2. Software structure 19
4.3. Schematic 19
5. Test **21**
5.1. Specific test cases and expected results 21
6. Conclusion **23**
References **24**

1. Introduction

1.1. Project description

This project is a classic Pong game that utilizes an LCD display and rotary encoders for paddle movement. Pong is a two-player sports game that simulates table tennis. In the game, the players control a paddle by dragging it vertically across the screen's left or right side. The objective is to hit the ball back and forth between the two paddles. The game has a scoring system that keeps track of each player's score. The game's speed increases as the score goes up, making it more challenging for the players to keep up. In addition to the two-player mode, an AI mode has also been implemented. The AI will move the paddle automatically and try to hit the ball. The game has been implemented using the C programming language and the STM32CubeIDE development environment.

1.2. Project motivation

The motivation behind this project is to apply the skills and knowledge acquired during the Embedded Systems Workshop. This final assignment provides an opportunity to showcase the practical skills in developing an embedded system that can perform a useful function. The project provides an opportunity to explore the integration of different components such as the LCD display and rotary encoders, while also improving skills in programming, debugging, and project management.

The Pong game is a popular and well-known classic, making it an interesting and engaging project to work on. By implementing a game, students can gain a deeper understanding of the challenges and limitations of embedded systems. Furthermore, by incorporating features such as AI and increasing difficulty, the project can help students develop more advanced programming and design skills.

2. User manual

2.1. Components

The Pong game implementation with the STM32 Nucleo Board consists of the following components:

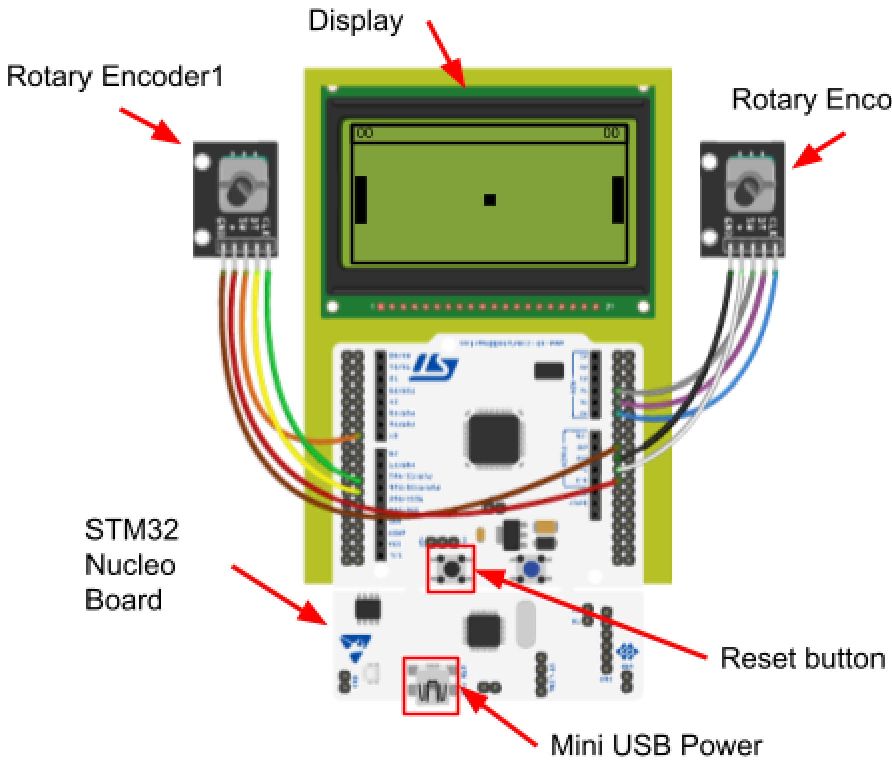

Figure 1: Components

- **STM32 Nucleo Board**
  This is the main control board for the game. It provides the necessary inputs and outputs to run the game.

  - **Mini USB power**
    The Mini USB power on the STM32 Nucleo Board allows the user to connect a USB cable to provide power to the board.

  - **Reset button**
    The reset button on the Nucleo Board is used to reset the system and restart the game.

- **Display**
  This is the screen that displays the game information and playfield. There are two areas on the screen, the information area and the playfield area

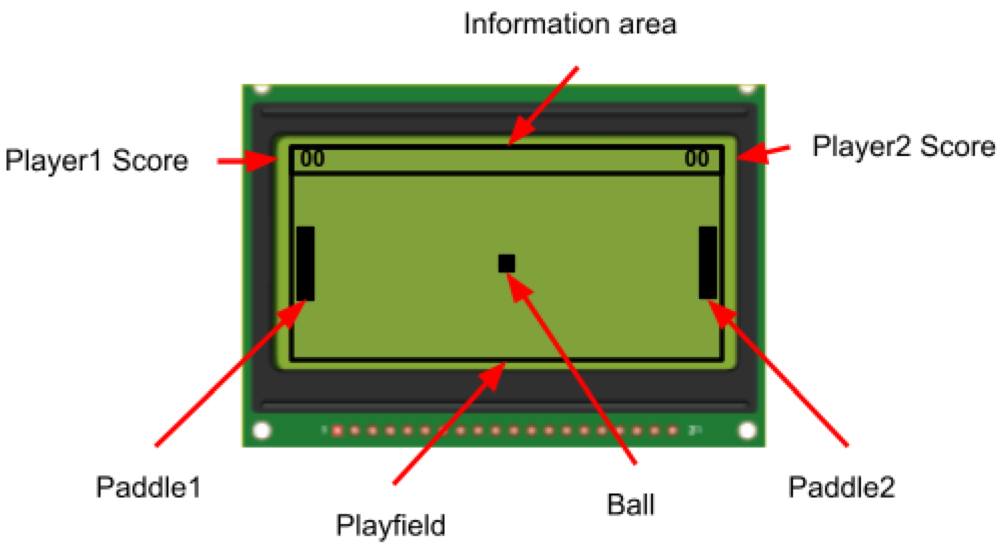

Figure 2: Display

  - **Information area**
    The information area on the LCD display is used to show game information, such as the current score.

  - **Playfield**
    The playfield is the main area of the LCD display and is used to show the game graphics, including the ball, and paddles. The playfield is updated in real-time as the game progresses, and the ball and paddles move in response to user input and game logic.

- **Rotary Encoders**
  These are the controllers for the game paddles. They allow the players to move the paddles up and down and switch between player and AI control.

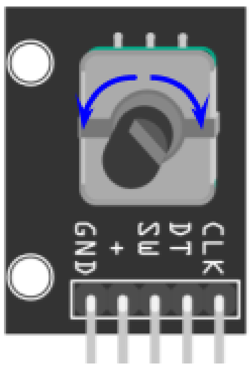

Figure 3: Rotary Encoder

2.2. Getting started

To start the game, make sure the STM32 Nucleo board is connected to power and the LCD screen is displaying the game interface.

2.3. Playing the game

The objective of the game is to score more points than the opponent by hitting the ball with a paddle and making it past the opponent's paddle.

- **Rules**
  - The game starts with the ball in the center of the field.
  - The ball is served by one player at the start of the game, and alternates between players after each point is scored.
  - The player on the left controls the left paddle, and the player on the right controls the right paddle. The players use their paddles to hit the ball back and forth.
  - If a player misses the ball, the other player scores a point.
  - The ball bounces off the top and bottom walls of the field, as well as off the paddles.
  - The speed of the ball gradually increases as the gameplay progresses, adding to the challenge of the game. The ball's speed increases up to ten times and then remains constant for the rest of the game.

2.4. Controlling the paddles

Use the two rotary encoders to control the paddles. Rotating the encoder clockwise will move the paddle down, while rotating the encoder counterclockwise will move the paddle up.

2.5. AI mode

To switch one of the paddles to AI control, press the button on the rotary encoder. When in AI mode, the paddle will move automatically to try and hit the ball. To switch back to manual control, press the button again.

2.6. Resetting the game

To reset the game, press the reset button on the STM32 Nucleo board. This will reset the score and start a new game.

2.7. Powering off the game

When finishing playing the game, turn off the power to the STM32 Nucleo board.

3. Project analysis

3.1. Modularization

The modularization of the project is an important consideration during the design phase. The project is divided into several modules, each responsible for a specific task. This approach allows for better code organization and increased reusability. It also makes code maintenance and debugging easier, as each module can be tested and updated independently without affecting the rest of the system. The following modules are implemented:

- **Main Module**
  The module handles the initialization of the system and sets up the necessary peripherals.

- **App Module**
  The App module contains the core of the game's logic, such as collision detection, ball and paddles movement, and score keeping. It communicates with the other modules to receive input from the user and update the game's display on the LCD screen. This module also manages the game state and transitions between game states, such as starting a new game or switching to the AI mode.

- **Timer Module**
  The Timer module is responsible for handling the game's timing by registering functions that will be called after a certain delay time. The app module can use this module to register functions responsible for updating the positions of the ball, paddles, and AI-controlled paddles.

- **Rotary Encoder Module**
  The rotary encoder module acts as a driver for the rotary encoders that control the game paddles. It reads the encoder signals and converts them into counter values, which the app module can use to update the paddles' positions. Furthermore, the module reads the button values on the encoder, enabling the app module to switch a paddle between manual mode and AI mode.

- **LCD Module**
  The LCD module provides a driver for the DEM 128o64B SYH-PY LCD display. It is responsible for drawing the game screen on the LCD and updating the screen with the current game state. This module receives commands from the App module to draw or update specific elements of the game screen, such as the ball, the paddles, and the score.

3.2. Abstract / Algorithmic description

The Pong game consists of several algorithms that are responsible for handling the game logic, input/output, and timing. The following is a brief overview of the algorithms used in the game:

- **Ball bounce behavior**
  In the Pong game, the ball needs to bounce off the paddles and the top and bottom wall of the playfield at a specific angle. One common solution is to use a 45-degree bounce, where the ball bounces off surfaces at a 45-degree angle. This solution is relatively simple to implement and provides a good balance of challenge and playability. However, it is important to ensure that the ball's trajectory is calculated accurately to prevent unintended behavior, such as the ball getting stuck in a loop or passing through objects.

- **Paddle Movement**
  The movement of the game paddles is controlled by the rotary encoders connected to each paddle. The rotary encoders generate signals that are read by the rotary encoder module, which converts them into counter values that represent the current position of each paddle. The app module regularly checks the current counter values and uses them to update the position of each paddle on the screen.

- **AI Mode**
  When a player switches a paddle to AI mode, the app module uses an algorithm to calculate the paddle's target position based on the ball's position. The algorithm takes into account the paddle's current position, and the ball's predicted trajectory. The app module then updates the paddle's counter value to move it towards the target position.

- **Increasing speed**
  As the game progresses, the difficulty level increases by raising the speed of the ball each time a player misses it with their paddle. This is achieved by decreasing the delay time of calling the function responsible for moving the ball, as well as the AI-controlled paddles.

- **Points on paddle miss**
  This Pong game includes a point system for missed paddle hits. Whenever a player misses the ball, their opponent earns a point. To determine whether a paddle has missed or hit the ball, the game checks whether the paddle's position overlaps with the ball's position. If the paddle and the ball overlap, it is considered a successful hit and the game continues. However, if the paddle and the ball do not overlap, it is considered a miss and the opposing player earns a point.

3.3. Analyzation

- **Reading values from rotary encoder**
  A rotary encoder produces two signals that are 90 degrees out of phase with each other. The order of the transitions in these signals can be analyzed to count the steps and detect the direction.

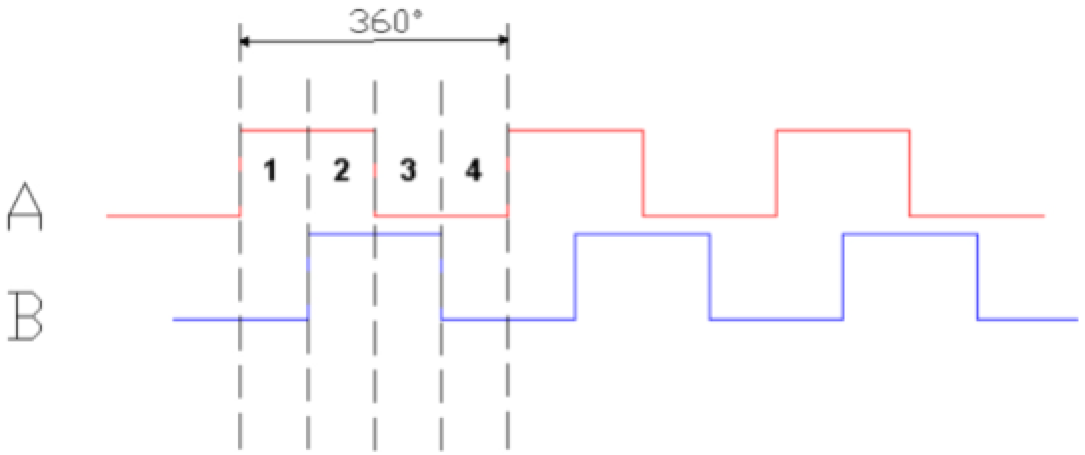

Figure 4: Rotary Encoder signals
source: STM32 Rotary Encoder - Stm32World Wiki

To decode the rotary encoder for controlling the paddle movement, there are two ways on the STM32 Nucleo Board: using a timer to decode rotary encoders or using an interrupt-driven state machine.

  - **Using a Timer to decode rotary encoders**
    The STM32 microcontroller has a built-in hardware encoder mode that simplifies the task of reading rotary encoder values. The Timer peripheral counts the number of encoder pulses and generates an interrupt when the encoder position changes. By connecting the encoder signals to the Timer peripheral's input pins, the Timer peripheral captures the pulse signals and determines the direction and magnitude of the rotation. To configure the Timer peripheral in encoder mode, the relevant registers need to be set up with appropriate values. The interrupt handler updates the position of the paddle according to the encoder's direction.

  - **Interrupt-driven state machine**
    Another way to read the values from the rotary encoder is to use an interrupt-driven state machine. This approach involves setting up the STM32 GPIO pins connected to the encoder signals as interrupt sources and handling the interrupt events in a state machine. The state machine reads the four possible states of the encoder (00, 01, 10, 11) and updates the paddle position based on the current and previous states. While this method is more flexible, it requires more code.

By implementing both approaches, it was found that the second method performed better. The first method sometimes caused 2-3 ticks to result in only one count, whereas the second method did not have this issue. As a result, the interrupt-driven state machine approach was used in this project.

- **Comparison of LCD Driver Implementation Approaches**
  To implement the LCD driver, there are various approaches available. Here's a comparison of three options:

  - **Direct Control Approach**
    In this approach, the display is controlled directly by the code. However, it has limitations since it requires setting the data of the entire page, including unwanted data. Thus, data needs to be read from the display first and then combined with the desired data using the OR or AND gate before writing it back to the display.

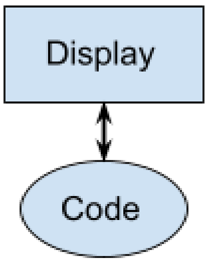

Figure 5: Direct Control

  - **Buffer Approach**
    This approach involves using a buffer to store the data for each pixel on the display. The buffer is an array where each element represents a pixel, eliminating the need to read data from the display. Instead, the data can be read from the buffer.

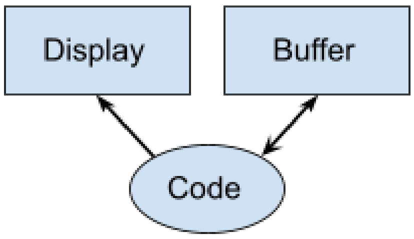

Figure 6: Buffer Control

  - **App Code and Driver code Approach**
    This method allows the application code to write directly to the buffer, while the driver code continuously updates the display from the buffer. This approach simplifies the logic and enhances the code's overall readability.

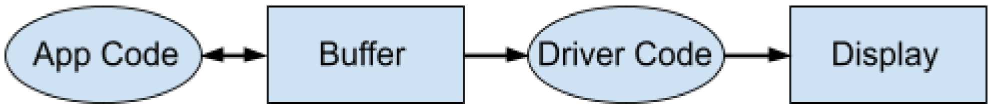

Figure 7: App Code Control

  The third approach of implementing the LCD driver is the easiest to implement. Since the microchip has enough memory for the buffer, this approach is a perfect fit for this project. The use of this approach also simplifies the maintenance and expansion of the code in the future.

4. Implementation

4.1. Specific description

The Pong game is implemented using the C programming language and the STM32CubeIDE development environment. The code is divided into several modules, each responsible for a specific aspect of the game.

- **rotary encoder driver**
  - **button**
    To read the encoder button value, the interrupt mode is used. To set up the pin for interrupt mode, navigate to the .ioc file and ensure that the button pins are set to interrupt mode and pulled up to high. Once this is done, the button state can be read in the interrupt handler using the following code:

```
case BTN1_PIN:
  if (HAL_GPIO_ReadPin(GPIOA, BTN1_PIN)) btn1 = RELEASED;
  else btn1 = PRESSED;
  break;
```

  - **encoder**
    An interrupt-driven state machine approach is used to read the rotary encoder values. The state machine reads the four possible phases of the rotary encoder (00, 01, 10, 11) and updates the paddle position based on the current and previous phases.

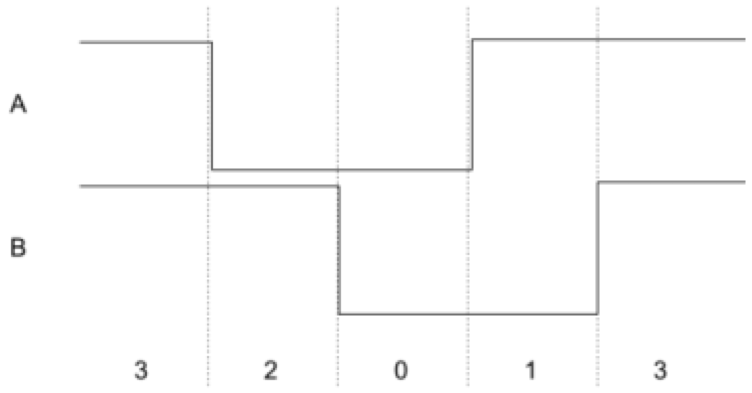

Figure 8: rotary encoder outputs phases
source: STM32 Rotary Encoder - Stm32World Wiki

The rotary encoder has four distinct phases: 0, 1, 2, and 3. Rotating the encoder in one direction causes the phases to transition as follows:

3 -> 2 -> 0 -> 1 -> 3

Conversely, rotating the encoder in the opposite direction causes the sequence to be reversed:

3 -> 1 -> 0 -> 2 -> 3

The following code detects rotary encoder1 rotation on pins `ENC1_A_PIN` and `ENC1_B_PIN`, using variables `enc1_old_state` and `enc1_new_state` to track changes. The if-else statements update the `enc1` variable based on the direction of rotation, incrementing it for clockwise rotation and decrementing it for counterclockwise. The `enc1_count` variable stores the current count value, calculated by dividing enc1 by 4 to account for the detection of 4 distinct phases.

```
case ENC1_A_PIN:
case ENC1_B_PIN:
  enc1_new_state=(uint8_t)((HAL_GPIO_ReadPin(GPIOA,ENC1_B_PIN)<<1)
                              | (HAL_GPIO_ReadPin(GPIOA, ENC1_A_PIN)));

   if ((enc1_old_state == 3 && enc1_new_state == 2)
    || (enc1_old_state == 2 && enc1_new_state == 0)
    || (enc1_old_state == 0 && enc1_new_state == 1)
    || (enc1_old_state == 1 && enc1_new_state == 3))
   {
      if (enc1 < enc_max*4) enc1++;
   }
   else if ((enc1_old_state == 3 && enc1_new_state == 1)
         || (enc1_old_state == 1 && enc1_new_state == 0)
         || (enc1_old_state == 0 && enc1_new_state == 2)
         || (enc1_old_state == 2 && enc1_new_state == 3))
   {
      if (enc1 > enc_min*4) enc1--;
   }
   enc1_count = enc1/4;
   enc1_old_state = enc1_new_state;
   break;
```

The following function is used to set the encoder value. The function first checks if the input value is within the range of `enc_min` and `enc_max`. If the value is within the range, it multiplies the input value by 4 and assigns it to enc1.

```
uint8_t set_enc1_val(int32_t value)
{
   if ((value <= (enc_max)) && (value >= (enc_min)))
   {
      enc1 = value*4;
      enc1_count = value;
      return 0;
   }
   else
   {
      return 1;
   }
}
```

- **LCD driver**
  The LCD driver is implemented using the approach that the App Code writes data to the buffer, and the Driver code updates the display from the buffer.
  In this approach, a `lcd_dsp_buffer` is needed. Regardless of whether the buffer data has changed or not. `lcd_loop` continuously updates the screen.

  - **lcd_dsp_buffer**
    the `lcd_dsp_buffer` is a 2D array in the LCD module that holds the pixel data for each character on the display. Each element in the array represents a pixel and its value determines whether it should be turned on or off.

    ```
    static bool lcd_dsp_buffer[LCD_COLUMNS][LCD_ROWS];
    ```

  - **lcd_loop**
    The `lcd_loop` function is responsible for continuously updating the display on the LCD screen with the contents of the display buffer. This function uses a nested loop to iterate over each row and column of the display buffer and send the corresponding data to the LCD screen using the `lcd_write_addr` function.

    ```
    void lcd_loop(void)
    {
        uint8_t data;

        for(uint8_t col = 0;col < LCD_COLUMNS;col++)
        {
            for(uint8_t page = 0;page < LCD_PAGES;page++)
            {
                data = 0;
                for(uint8_t i = 0;i < 8;i++)
                {
                    uint8_t row = page * 8 + i;
                    data += lcd_dsp_buffer[col][row] << i;
                }

                lcd_write_addr(page,col,data);
            }
        }
    }
    ```

- **lcd_write_ addr**
  This function writes a byte of data to a specific address on the LCD screen. The address is specified by the `page` and `column` arguments, and the data to be written is specified by the `data` argument. The function first determines whether the column falls within the left or right half of the screen(as the screen is divided into two separate address spaces), and sets the appropriate chip select line accordingly. Then, it sends the appropriate commands and data to the LCD screen via the `lcd_write` function.

```
void lcd_write_addr(uint8_t page, uint8_t column, uint8_t data)
{
    bool cs;

    // Determine which chip select line to use based on the column number
    if (column < 64) cs = false;
    else cs = true;

    // Send commands and data to the LCD screen
    lcd_write(!cs, cs, 0, 0b10111000 | page);
    lcd_write(!cs, cs, 0, 0b01000000 | column);
    lcd_write(!cs, cs, 1, data);
}
```

- **lcd_write**
  In order to generate the signals for writing data to the display, it is necessary to first check the timing diagram.

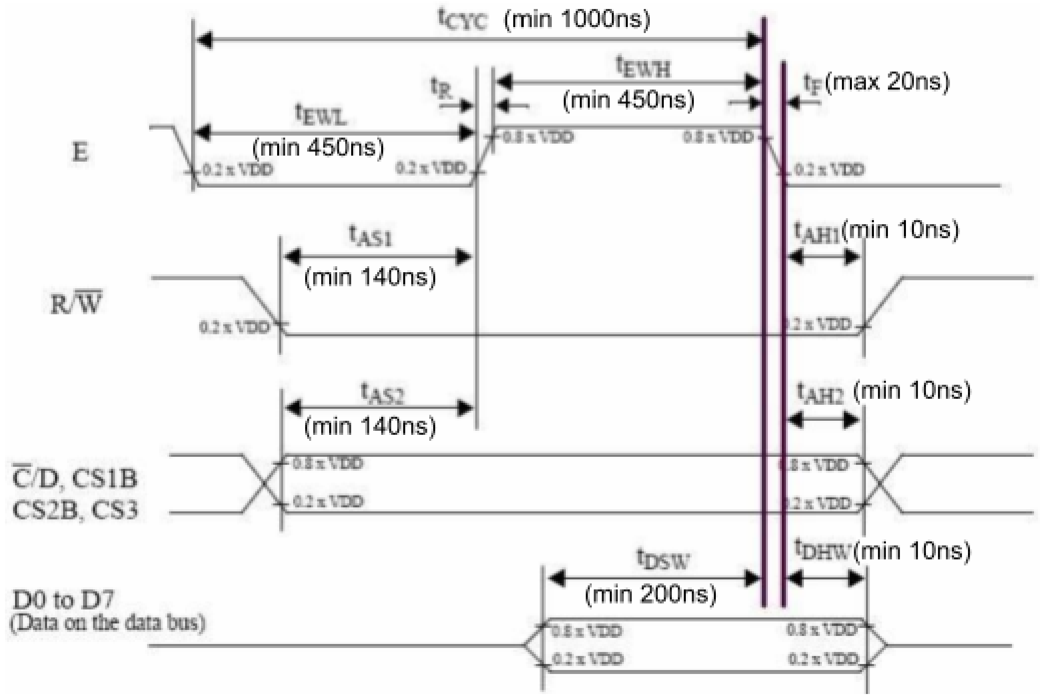

Figure 9: AC timing for writing to the SBN0064G
source: DEM 128064B SYH-PY_Ver. 4.1.4

To generate the signals with code, a nanosecond delay is required. However, the HAL driver only provides a minimum 1ms delay, so an alternative approach is needed. One way to achieve this is by using Timer 1 in counter mode, which runs at 84 MHz. This timer can be used to implement a `delay_12nano_sec` function, as it counts every 1/84 MHz ≈ 12 nanoseconds.

```
static void delay_12_nanosec(int dlytime)
{
    __HAL_TIM_SetCounter(&htim1, 0);
    while(__HAL_TIM_GetCounter(&htim1) < dlytime);
}
```

The `lcd_write` function follows the timing diagram to write data to the LCD screen. It takes four arguments: `cs1`, `cs2`, `di`, and `data`. `cs1` and `cs2` determine which half of the LCD screen the data is being written to, `di` specifies whether the data is an instruction or data, and `data` is the 8-bit data to be written to the screen.

```
static void lcd_write(bool cs1, bool cs2, bool di, uint8_t data)
{
    HAL_GPIO_WritePin(LCD_GPIO_Port, EN_Pin, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(LCD_GPIO_Port, RW_Pin, GPIO_PIN_SET);
    HAL_GPIO_WritePin(LCD_GPIO_Port, CS1_Pin, cs1);
    HAL_GPIO_WritePin(LCD_GPIO_Port, CS2_Pin, cs2);
    HAL_GPIO_WritePin(LCD_GPIO_Port, DI_Pin, di);
    HAL_GPIO_WritePin(LCD_GPIO_Port, DB7_Pin, (data&0b10000000)>>7);
    HAL_GPIO_WritePin(LCD_GPIO_Port, DB6_Pin, (data&0b01000000)>>6);
    HAL_GPIO_WritePin(LCD_GPIO_Port, DB5_Pin, (data&0b00100000)>>5);
    HAL_GPIO_WritePin(LCD_GPIO_Port, DB4_Pin, (data&0b00010000)>>4);
    HAL_GPIO_WritePin(LCD_GPIO_Port, DB3_Pin, (data&0b00001000)>>3);
    HAL_GPIO_WritePin(LCD_GPIO_Port, DB2_Pin, (data&0b00000100)>>2);
    HAL_GPIO_WritePin(LCD_GPIO_Port, DB1_Pin, (data&0b00000010)>>1);
    HAL_GPIO_WritePin(LCD_GPIO_Port, DB0_Pin, (data&0b00000001));

    delay_12_nanosec(26);//t_EWL - T_AS1 min 450 - 140 = 310ns

    HAL_GPIO_WritePin(LCD_GPIO_Port, RW_Pin, GPIO_PIN_RESET);

    delay_12_nanosec(12);//t_AS1 min 140ns

    HAL_GPIO_WritePin(LCD_GPIO_Port, EN_Pin, GPIO_PIN_SET);

    delay_12_nanosec(38);//t_EWH min 450ns

    HAL_GPIO_WritePin(LCD_GPIO_Port, EN_Pin, GPIO_PIN_RESET);

    delay_12_nanosec(1);//t_AH1 min 10ns

    HAL_GPIO_WritePin(LCD_GPIO_Port, RW_Pin, GPIO_PIN_SET);

    delay_12_nanosec(1);
}
```

The Lcd module also provides some functions that the app module can use to write or clear data on the screen. These functions include `lcd_clear`, `lcd_draw_clear_rect`, and `lcd_write_number`.

- **lcd_clear**
  The lcd_clear function sets all the display buffer values to zero, effectively clearing the display.

```
void lcd_clear()
{
    for(uint8_t col = 0;col < LCD_COLUMNS;col++)
    {
        for(uint8_t row = 0;row < LCD_ROWS;row++)
        {
            lcd_dsp_buffer[col][row] = 0;
        }
    }
}
```

- **lcd_draw_clear_rect**
  The lcd_draw_clear_rect function either draws or clears a rectangle on the LCD screen by setting specific bits in the buffer to ones or zeros.

```
void lcd_draw_clear_rect(uint8_t posX, uint8_t posY, uint8_t length, uint8_t
width, bool data)
{
    for(uint8_t col = posX;col < (width+posX);col++)
    {
        for(uint8_t row = posY;row < (length+posY);row++)
        {
            lcd_dsp_buffer[col][row] = data;
        }
    }
}
```

- **lcd_write_number**
  This function writes a decimal number to the specified location on the LCD screen using a font library for the number graphics.

```
void lcd_write_number(uint8_t posX, uint8_t posY, uint8_t number)
{
    for(uint8_t x = 0;x < FONT_COLUMNS;x++)
    {
        for(uint8_t y = 0;y < FONT_ROWS;y++)
        {
            lcd_dsp_buffer[posX+x][posY+y] = (font[number+'0'][y]&(1 << x)) >> x;
        }
    }
}
```

- **Timer module**
  The Timer module provides a simple timer function that allows functions to be registered and to be called every certain delay time.

  - **'timerFunctionReg' Struct**
    This struct is used to register timer functions with the Timer_RegisterFunction function. Each struct contains a pointer to the timer function, a divider value, and a counter value.

    - **funcPointer:** a pointer to the timer function to be called when the counter reaches zero.
    - **divider:** the value used to divide the timer clock frequency to produce the desired timer period.
    - **counter:** the current value of the timer counter. When it reaches the value of the divider, the timer function pointed to by funcPointer will be called.

```
typedef struct timerFunctionReg timerFunctionReg;
struct timerFunctionReg
{
   timer_fp_t funcPointer;
   uint16_t divider;
   uint16_t counter;
};
```

  - **timer register**
    The `timer_register` function can be used to register timer functions by providing a function pointer and a divider value. The function pointer should point to the function to be executed by the timer, while the divider value determines the frequency of the function execution.
    When the `timer_register` function is used, the registered function will be called regularly by the timer interrupt handler. If the registration is successful, the function returns 0, otherwise, if the maximum number of timers has already been registered, it returns 1. The maximum number of timers allowed can be found in the `TIMER_MAX_TIMERS` macro defined in timer.h.

```
int timer_register(timer_fp_t timerfp, uint32_t div)
{
   if (funcNum < TIMER_MAX_TIMERS)
   {
      timerFuncReg[funcNum].funcPointer = timerfp;
      timerFuncReg[funcNum].divider = div;
      funcNum++;
      return 0;
   }
   else
   {
      return 1;
   }
}
```

- **timer tick**
  The `timer_tick` function is used to execute registered timer functions on each timer tick. This function iterates through the list of registered timer functions, updates their counters, and executes any functions that have reached their specified divider value.

```c
void timer_tick(void)
{
   // Iterate through the list of registered timer functions
   for(int i = 0;i < funcNum;i++)
   {
      // Increment the counter for this timer function
      timerFuncReg[i].counter++;

      // If the counter has reached the specified divider value,
      // execute the timer function and reset the counter
      if((timerFuncReg[i].counter >= (timerFuncReg[i].divider)))
      {
         timerFuncReg[i].funcPointer();
         timerFuncReg[i].counter = 0;
      }
   }
}
```

- **timer function divider update**
  If the frequency at which a registered function should be executed needs to be changed, the divider value can be updated using the `timer_func_divider_update` function. This function takes a function pointer and a new divider value as arguments. It loops through the registered timer function list and checks if the timer function pointer matches the input timer function pointer. If a match is found, the function updates the timer function's divider.

```c
int timer_func_divider_update(timer_fp_t timerfp, uint32_t div)
{
   for(int i = 0;i < funcNum;i++)
   {
      if(timerFuncReg[i].funcPointer == timerfp)
      {
         timerFuncReg[i].divider = div;
         return 0;
      }
   }
   return 1;
}
```

- **App module (Game Logic)**
  - **Updating the ball's position**

    The ball's position is updated in the `update_ball_position` function in app.c. This function is called repeatedly by the timer.

    The following code checks if the ball collides with paddle 1 or misses it. If the ball hits the paddle, the code sets the ball's x-direction to 1, and if the ball's y-direction is 0, sets it to 1 to make the ball bounce towards the opposite direction in 45-degree. If the ball misses the paddle, the `player2_score` is incremented by 1, and the game is initialized using the `game_init` function.

```
if(ball_x <= PADDLE_WIDTH + 1)
{
   if((ball_y + BALL_SIZE > enc1_count) && (ball_y < enc1_count +
                                              PADDLE_LENGTH))
   {
      // The ball hits the paddle 1
      ball_dx = 1;
      if(ball_dy == 0) ball_dy = 1;
   }
   else
   {
      //paddle 1 miss
      if (player2_score < 99) player2_score++;
      game_init();
   }
}
```

    The following code checks if the ball has hit the top or bottom wall. If the ball has hit the wall, then the code inside the if statement is executed. The ball's y-direction is reversed by multiplying the `ball_dy` variable by -1.

```
else if((ball_y <= INFO_AREA_SIZE + 2) || (ball_y+BALL_SIZE >=
                                               LCD_ROWS - 1))
{
   // the ball hit the top or bottom wall
   ball_dy *= -1;
}
```

    The following lines of code update and draw the ball's position on the display. The first line updates the ball's x-position by adding the `ball_dx` value and the second line updates the ball's y-position by adding the `ball_dy` value. The third line draws the ball on the display.

```
// Update the ball's position
ball_x += ball_dx;
ball_y += ball_dy;
lcd_draw_clear_rectangle(ball_x, ball_y, BALL_SIZE, BALL_SIZE, 1);
```

- **Updating the paddles' position**

  The following code is to update the paddle1's position. If the encoder count for paddle 1 (`enc1_count`) has changed from its previous value (`enc1_old`), the function clears the old position of paddle 1 on the display and draws it into the new position. It then updates the value of `enc1_old` to match the current encoder count.

```
if((enc1_old != enc1_count))
{
   //update the paddle1's position
   lcd_draw_clear_rect(1,(uint8_t)enc1_old,
                            PADDLE_LENGTH,PADDLE_WIDTH,0);

   lcd_draw_clear_rect(1,(uint8_t)enc1_count,
                            PADDLE_LENGTH,PADDLE_WIDTH,1);

   enc1_old = enc1_count;
}
```

- **AI mode**

  The `ai_mode` function is responsible for controlling the movement of the paddle when the AI mode is enabled. This function checks if the AI mode is enabled for the paddle, and if so, calculates the middle position of the paddle and the ball. If the middle position of the paddle is larger than the ball, the paddle moves up, otherwise it moves down.

```
if((player1_AI_mode) && (btn1 != PRESSED))
{
   if((enc1_count + (PADDLE_LENGTH/2)) > (ball_y+(BALL_SIZE/2)))
   {
      set_enc1_val(enc1_count - 1); // Move the paddle 1 up
   }
   else if((enc1_count + (PADDLE_LENGTH/2)) < (ball_y+(BALL_SIZE/2)))
   {
      set_enc1_val(enc1_count + 1); // Move the paddle 1 down
   }
}
```

- **Increasing speed**

  This code includes a dynamic ball speed adjustment feature based on the cumulative score of both players. As the total score increases, the delay time of the `update_ball_position` and `ai_mode` timer functions are reduced. This allows for faster ball movement and more challenging gameplay as the game progresses.

```
if((player1_score+player2_score) <= 10)
{
   timer_func_divider_update(update_ball_position,
                        UPDATE_BALL_DLY-(player1_score+player2_score)*3);
   timer_func_divider_update(ai_mode,
                                 AI_DLY-(player1_score+player2_score)*4);
}
```

4.2. Software structure

The Pong game project is organized into several modules that handle different aspects of the game. These modules communicate with each other to create the game loop and update the display.

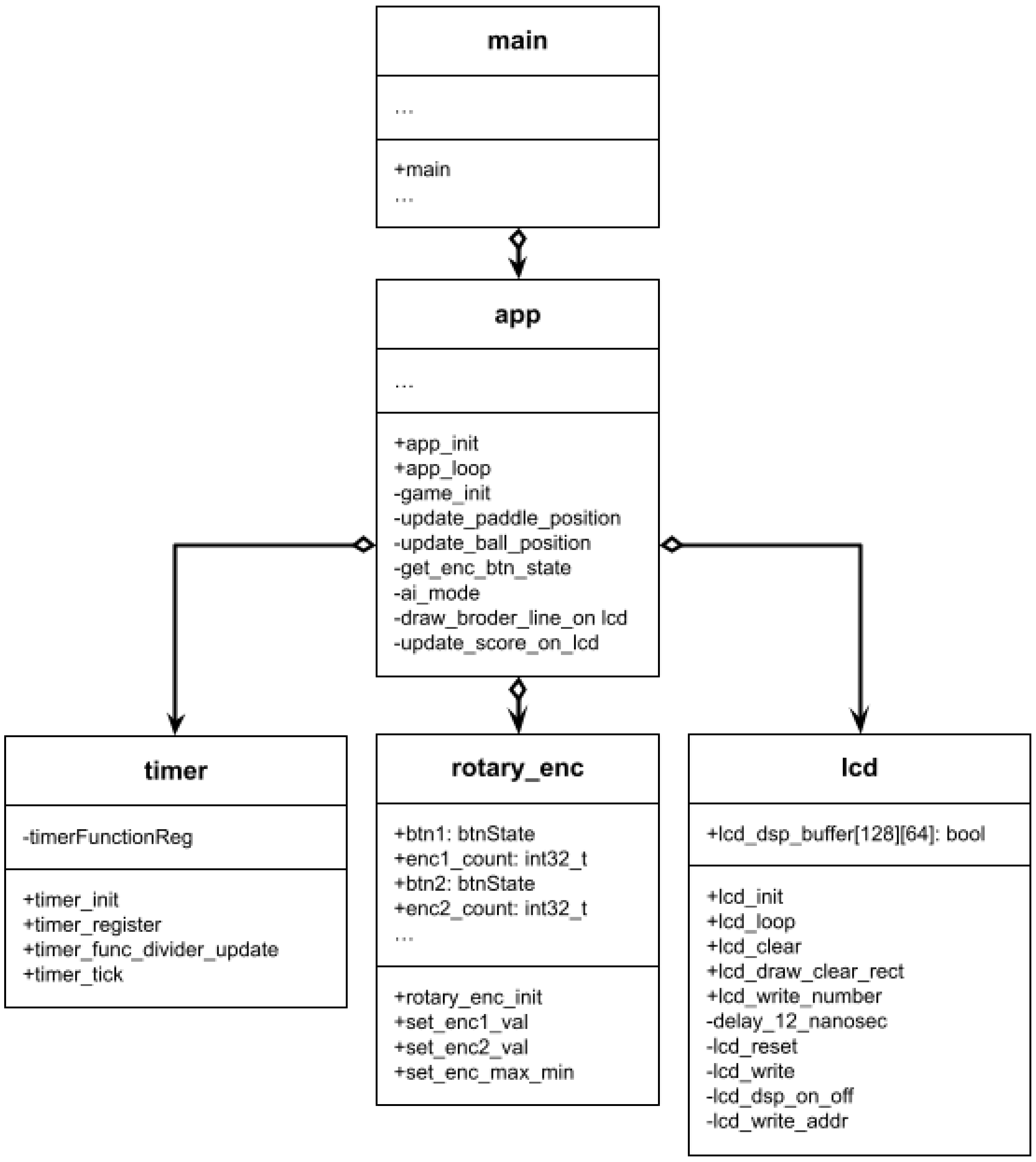

Figure 10: Software structure

4.3. Schematic

The electrical schematic for the Pong game project is shown in Figure 11. This schematic shows the connections between the STM32 Nucleo Board, the LCD display, and the two rotary encoders.

- **Pin Allocations**
  - **Rotary Encoder 1**

ENC1	SW	DT	CLK
STM	PA8	PA6	PA7

  - **Rotary Encoder 2**

ENC2	SW	DT	CLK
STM	PA4	PA1	PA0

  - **LCD**

LCD	DB0	DB1	DB2	DB3	DB4	DB5	DB6	DB7
STM	PC0	PC1	PC2	PC3	PC4	PC5	PC6	PC7
LCD	CS1	CS2	RST	RW	DI	EN		
STM	PC8	PC9	PC10	PC11	PC12	PC13		

- **Schematic**

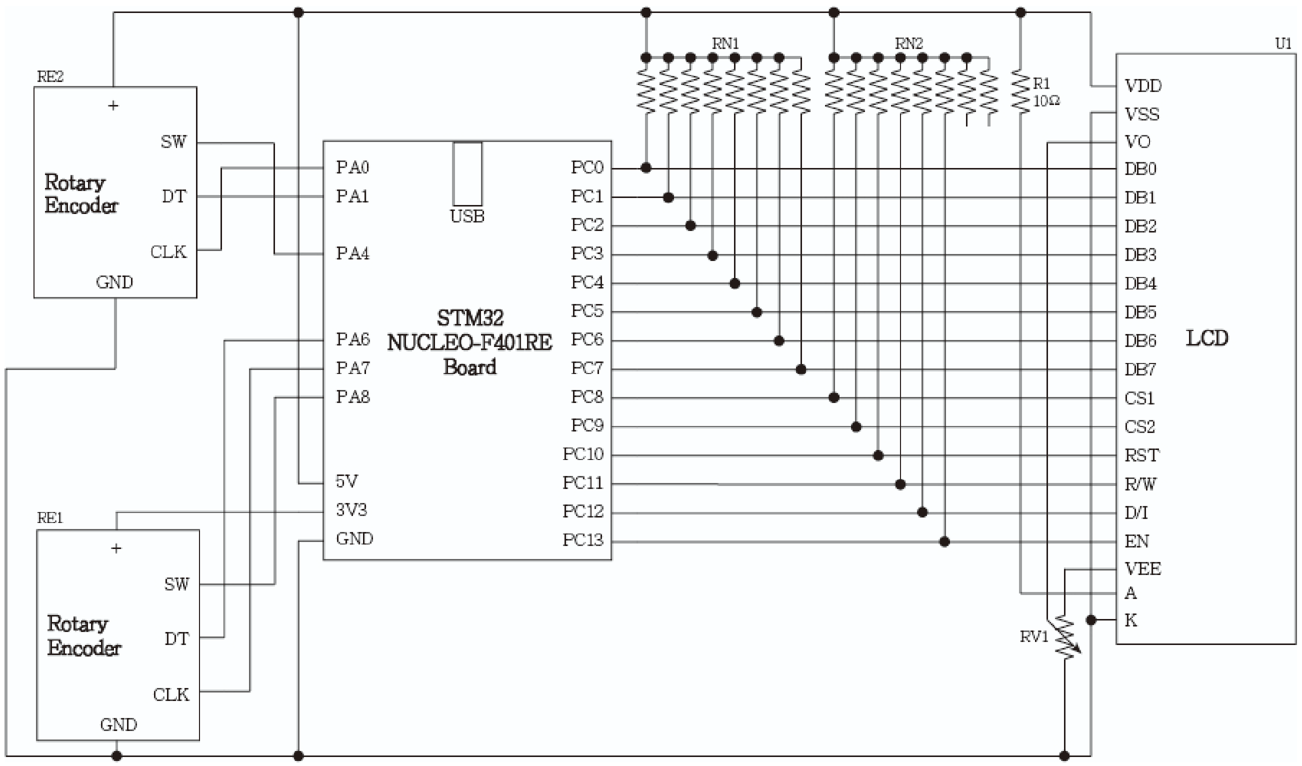

Figure 11: Schematic

5. Test

5.1. Specific test cases and expected results

- **Paddles are moved with encoders**

	Test case	Expect result	pass / fail
1	Rotate the encoder1 by one step clockwise	Paddle1 moves down	pass
2	Rotate the encoder1 by one step counterclockwise	Paddle1 moves up	pass
3	Rotate the encoder2 by one step clockwise	Paddle2 moves down	pass
4	Rotate the encoder2 by one step counterclockwise	Paddle2 moves up	pass

- **The ball moves and bounces in 45-degree angle**

	Test case	Expect result	pass / fail
1	Start the game	The ball move when the game start	pass
2	Move the paddle1 to catch the ball	The ball bounces in 45-degree angle	pass
3	Move the paddle2 to catch the ball	The ball bounces in 45-degree angle	pass
4	When the ball touch the top border line	The ball bounces in 45-degree angle	pass
5	When the ball touch the bottom border line	The ball bounces in 45-degree angle	pass

- **Points on paddle miss**

	Test case	Expect result	pass / fail
1	Let the ball pass the paddle 1	Player 2 should score a point, and the ball should reset to the middle of the screen.	pass
2	Let the ball pass the paddle 2	Player 1 should score a point, and the ball should reset to the middle of the screen.	pass

- **Speed increase**

	Test case	Expect result	pass / fail
1	In the situation that player1’s score + player2’s score is smaller or equal to 10, move the paddle1 and miss the ball with the paddle	The game restart and the speed increase	pass
2	In the situation that player1’s score + player2’s score is greater than 10, move the paddle2 and miss the ball with the paddle	The game restart but the speed doesn’t increase anymore	pass

- **AI mode**

	Test case	Expect result	pass / fail
1	Press the button on rotary encoder 1	The corresponding paddle should start moving automatically to try and hit the ball	pass
2	Press the button on rotary encoder 2	The corresponding paddle should start moving automatically to try and hit the ball	pass

- **Reset button**

	Test case	Expect result	pass / fail
1	Press the reset button on the Nucleo Board	The game should reset, and the ball should appear in the middle of the screen again. Scores should be reset to 0.	pass

6. Conclusion

In conclusion, we have successfully developed a Pong game on the STM32 Nucleo Board that utilizes an LCD display and rotary encoders for paddle movement. The game simulates table tennis and features basic gameplay mechanics such as ball bouncing, paddle movement, scoring, and increasing game speed. Additionally, we have implemented an AI mode that automatically controls the paddles.

Through our implementation of two different ways to read the rotary encoder, we found that the Interrupt-driven state machine method performed better than the other approach. We also compared three different methods for implementing the LCD module and determined that using a buffer for the display provided the most efficient approach.

This project allowed us to gain valuable experience in software design and development for embedded systems, as well as to enhance our skills in C programming, hardware interfacing, and problem-solving. We hope that our Pong game provides an enjoyable and engaging experience for users and serves as a helpful example for others interested in developing similar projects.

References

[1] STM32 Rotary Encoder - Stm32World Wiki
Link: https://stm32world.com/wiki/STM32_Rotary_Encoder

[2] Microsecond/Nanoseconds delay in STM32 » ControllersTech
Link: https://controllerstech.com/create-1-microsecond-delay-stm32/

[3] DEM 128064B SYH-PY_Ver. 4.1.4
Link:https://www.tme.eu/Document/58c47b640ee636cf5a2f433dbafcb0c2/DEM%20128064B%20SYH-PY.pdf