Mark	

Team name:	*A1*		
Homework number:	*HOMEWORK 10-11(?)*		
Due date:	17/12/2023		
Contribution	NO	Partial	Full
1 Monti Pietro			*x*
2 Moretto Alessia			*x*
3 Pallotto Francesco			*x*
4 Perna Alessandro			*x*
5 Ventura Ludovico			*x*
Notes:			

Project name	IR communication on a single board		
Not done	Partially done (major problems)	Partially done (minor problems)	Completed
			x

We successfully completed the homework.

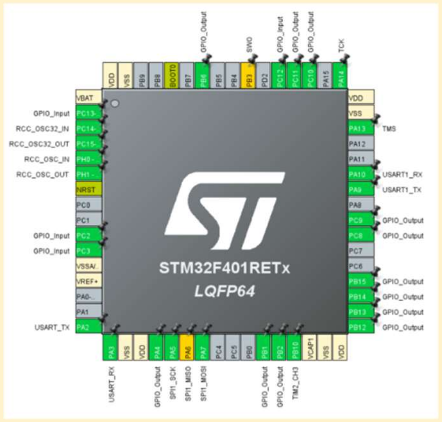
1. Introduction:

The aim of the project is:

- scan the pushbutton matrix in order to send the data corresponding to the pressed button via IR UART
- receive the IR UART data showing the corresponding character on the LED matrix

1.1 Project Settings:

The pinout configurations is set as in the image:



The appropriate column pins are set as **GPIO_Outputs**, while the rows are set as **GPIO_Inputs.**
We add the pin PB10 in order to be connected to timer 2 channel 3 which is the one connected to the infrared led. Inside timer 2 we have to enable the channel 3 as PWM generator because the infrared receiver is provided with a band-pass filter set at 38kHz and at a modulator. We need to send a signal modulated at 38kHz in order to be detected by our infrared receiver that filters the background light. We set the ARR to 10-1 and the pulse to 5.
For timer 3 we set the PSC to 3500-1 with the ARR of 10-1.
For timer 4 we set the PSC to 8400-1 with the ARR of 5**(*)**.
For timer 5 we set the PSC to 8400-1 with the ARR of 10-1.
For timer 9 we set the PSC to 8400-1 with the ARR of 10-1.
For timer 10 we set the PSC to 8400-1 with the ARR of 5**(*)**.

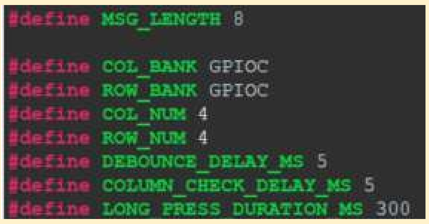
(*) Note: This value is multiplied by 10 as detailed in the **SET_START_TIMER** macro, described later in the report.

For the receiver we configured everything in order to have the correct set up for the SPI & UART communication. We enabled the pin PA5 as SPI1_SCK, the pin PA7 as SPI1_MOSI and the pin PB6 has been adjusted to be a GPIO_Output in order to control the RCLK.

Finally, we need to select PA10 and PA9 for UART1 communication with the infrared receiver. Communication has the following settings: baud rate (2400 Bits/s), word length (8 Bits), no parity and 1 start/stop bit.
We set TIM3, TIM4, TIM5, TIM9, TIM10 and USART1 as global interrupts.

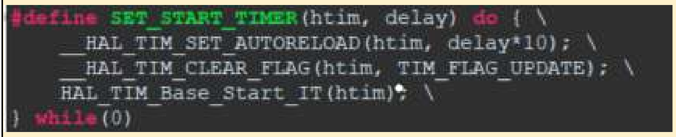
2.1 Defines & Macros

We use the defines as in the previous project to set the pins of columns and the pins of rows.

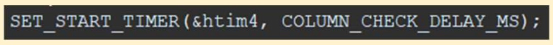


```
#define MSG_LENGTH 8

#define COL_BANK GPIOC
#define ROW_BANK GPIOC
#define COL_NUM 4
#define ROW_NUM 4
#define DEBOUNCE_DELAY_MS 5
#define COLUMN_CHECK_DELAY_MS 5
#define LONG_PRESS_DURATION_MS 300
```

- **COL_BANK and ROW_BANK**: Specifies the GPIO banks used for the columns and rows of the keypad matrix. In this code, both are set to GPIOC.
- **COL_NUM and ROW_NUM**: Defines the number of columns and rows in the keypad matrix. This code is designed for a 4x4 matrix, so COL_NUM is 4, and ROW_NUM is 4.
- **DEBOUNCE_DELAY_MS**: Specifies the duration (in milliseconds) waited by the system to allow the stabilization of the key press. This delay helps in mitigating contact bouncing issues common in mechanical switches.
- **COLUMN_CHECK_DELAY_MS**: Sets the delay (in milliseconds) between successive column scans. It determines the rate at which the code iterates between the end of the scan of a column and the next one.
- **LONG_PRESS_DURATION_MS**: Defines the duration (in milliseconds) required for a key press to be considered a long press. If a key is held down for at least this duration, it is treated as a long press and therefore printed out.

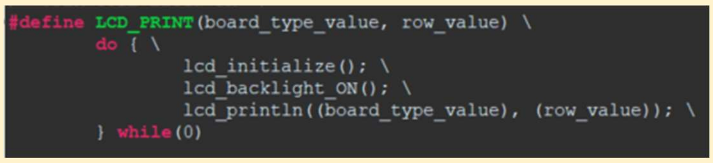


```
#define SET_START_TIMER(htim, delay) do { \
    __HAL_TIM_SET_AUTORELOAD(htim, delay*10); \
    __HAL_TIM_CLEAR_FLAG(htim, TIM_FLAG_UPDATE); \
    HAL_TIM_Base_Start_IT(htim); \
} while(0)
```



```
SET_START_TIMER(&htim4, COLUMN_CHECK_DELAY_MS);
```

SET_START_TIMER: Macro used to start a given timer in interrupt mode for a specified period of time (delay), multiplied by 10 since the prescaler of the timer is 8400-1, clearing its flag.

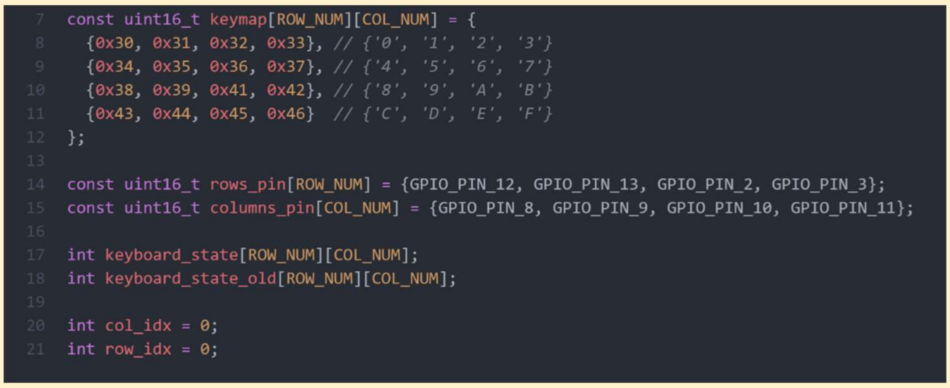


```
#define LCD_PRINT(board_type_value, row_value) \
      do { \
            lcd_initialize(); \
            lcd_backlight_ON(); \
            lcd_println((board_type_value), (row_value)); \
      } while(0)
```

The macro is designed to simplify the process of printing information to an LCD. We initialize the LCD, we turn on the backlight and we print the information.

2.2 Variables

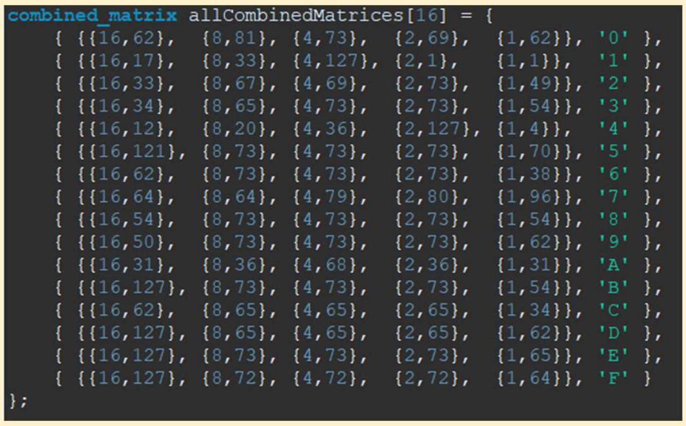
Declares some variables:



```
const uint16_t keymap[ROW_NUM][COL_NUM] = {
  {0x30, 0x31, 0x32, 0x33}, // {'0', '1', '2', '3'}
  {0x34, 0x35, 0x36, 0x37}, // {'4', '5', '6', '7'}
  {0x38, 0x39, 0x41, 0x42}, // {'8', '9', 'A', 'B'}
  {0x43, 0x44, 0x45, 0x46}  // {'C', 'D', 'E', 'F'}
};

const uint16_t rows_pin[ROW_NUM] = {GPIO_PIN_12, GPIO_PIN_13, GPIO_PIN_2, GPIO_PIN_3};
const uint16_t columns_pin[COL_NUM] = {GPIO_PIN_8, GPIO_PIN_9, GPIO_PIN_10, GPIO_PIN_11};

int keyboard_state[ROW_NUM][COL_NUM];
int keyboard_state_old[ROW_NUM][COL_NUM];

int col_idx = 0;
int row_idx = 0;
```

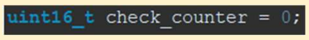
- **keymap:** A 4x4 matrix representing the characters associated to each key in the keypad. The characters are the hexadecimal representation of the ASCII code.
- **rows_pin & columns_pin:** Arrays containing the GPIO pins corresponding to the rows and columns of the keypad matrix (rows are in the physical order of the port).
- **keyboard_state & keyboard_state_old:** 2D arrays storing the current and previous state of each key in the matrix. These arrays are used for detecting key transitions.

- **col_idx & row_idx:** Variables representing the current column and row indices during keypad scanning.



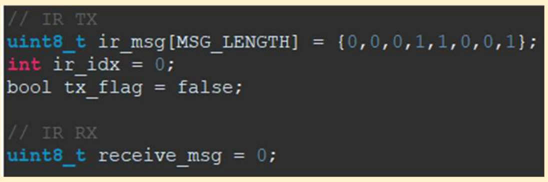
```
combined_matrix allCombinedMatrices[16] = {
    { {{16,62},  {8,81}, {4,73},  {2,69},  {1,62}}, '0' },
    { {{16,17},  {8,33}, {4,127}, {2,1},   {1,1}},  '1' },
    { {{16,33},  {8,67}, {4,69},  {2,73},  {1,49}}, '2' },
    { {{16,34},  {8,65}, {4,73},  {2,73},  {1,54}}, '3' },
    { {{16,12},  {8,20}, {4,36},  {2,127}, {1,4}},  '4' },
    { {{16,121}, {8,73}, {4,73},  {2,73},  {1,70}}, '5' },
    { {{16,62},  {8,73}, {4,73},  {2,73},  {1,38}}, '6' },
    { {{16,64},  {8,64}, {4,79},  {2,80},  {1,96}}, '7' },
    { {{16,54},  {8,73}, {4,73},  {2,73},  {1,54}}, '8' },
    { {{16,50},  {8,73}, {4,73},  {2,73},  {1,62}}, '9' },
    { {{16,31},  {8,36}, {4,68},  {2,36},  {1,31}}, 'A' },
    { {{16,127}, {8,73}, {4,73},  {2,73},  {1,54}}, 'B' },
    { {{16,62},  {8,65}, {4,65},  {2,65},  {1,34}}, 'C' },
    { {{16,127}, {8,65}, {4,65},  {2,65},  {1,62}}, 'D' },
    { {{16,127}, {8,73}, {4,73},  {2,73},  {1,65}}, 'E' },
    { {{16,127}, {8,72}, {4,72},  {2,72},  {1,64}}, 'F' }
};
```

- Each letter is a 5x2 array, we create the LED matrix.



```
uint16_t check_counter = 0;
```

- We keep track of the duration of the press



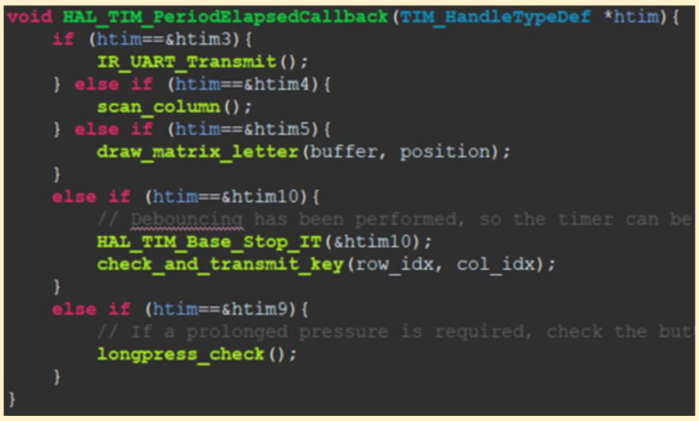
```
// IR TX
uint8_t ir_msg[MSG_LENGTH] = {0,0,0,1,1,0,0,1};
int ir_idx = 0;
bool tx_flag = false;

// IR RX
uint8_t receive_msg = 0;
```

- **ir_msg:** byte array to send & receive through IR (rng initialization).
- **ir_idx:** index for the byte array ir_msg.
- **tx_flag:** a boolean variable employed to differentiate when transmission is over.
- **receive_msg:** current received bit.

3.1 Callback Functions

3.1.1 Timer Callback: "void HAL_TIM_PeriodElapsedCallback()"

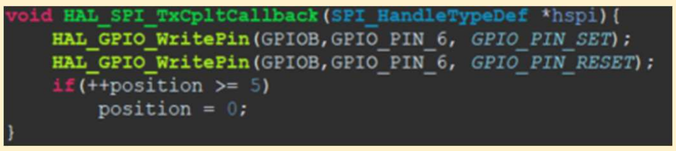
The implementation of HAL_TIM_PeriodElapsedCallback associated with:



```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
    if (htim==&htim3){
        IR_UART_Transmit();
    } else if (htim==&htim4){
        scan_column();
    } else if (htim==&htim5){
        draw_matrix_letter(buffer, position);
    }
    else if (htim==&htim10){
        // Debouncing has been performed, so the timer can be
        HAL_TIM_Base_Stop_IT(&htim10);
        check_and_transmit_key(row_idx, col_idx);
    }
    else if (htim==&htim9){
        // If a prolonged pressure is required, check the but
        longpress_check();
    }
}
```

- TIM3 → send the data corresponding to the pressed button via IR UART
- TIM4 → has the purpose of triggering the column scanning process.
- TIM5 → we call the draw_matrix_letter function to draw characters in the led matrix
- TIM10 → Debouncing has been performed, the timer can be stopped and the key value can be checked
- TIM9 → if a prolonged pressure is required, we can check if the button is pressed for the desired time.
-

3.2.6 Set and Reset the RCLK: "void HAL_SPI_TxCpltCallback()"

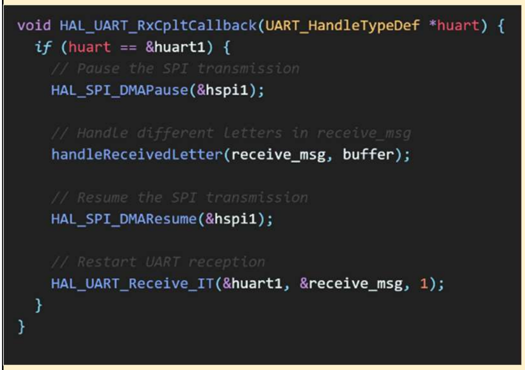


```
void HAL_SPI_TxCpltCallback(SPI_HandleTypeDef *hspi){
    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_6, GPIO_PIN_SET);
    HAL_GPIO_WritePin(GPIOB,GPIO_PIN_6, GPIO_PIN_RESET);
    if(++position >= 5)
        position = 0;
}
```

The HAL_SPI_TxCpltCallback sets and resets the RCLK pin, cycling through the columns of the letter.

The fundamental principle of this code is that the SPI sends information one column at a time. Once sent, we prepare the SPI for the next transmission and switch to the next column.

3.1.3 : “void HAL_UART_RxCpltCallback()”



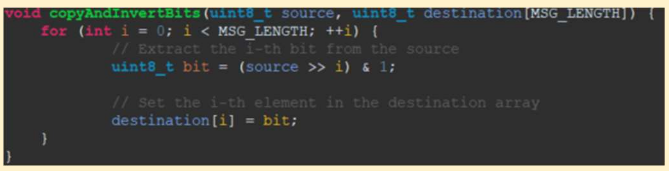
```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
  if (huart == &huart1) {
    // Pause the SPI transmission
    HAL_SPI_DMAPause(&hspi1);

    // Handle different letters in receive_msg
    handleReceivedLetter(receive_msg, buffer);

    // Resume the SPI transmission
    HAL_SPI_DMAResume(&hspi1);

    // Restart UART reception
    HAL_UART_Receive_IT(&huart1, &receive_msg, 1);
  }
}
```

The function handles the reception from the UART.

When reception through UART1 is complete, SPI_DMA communication is paused, to safely handle the data, we then change the displayed char of the LED matrix, resume the SPI_DMA communication and finally set UART1 back to reception mode.

3.2 Private Functions

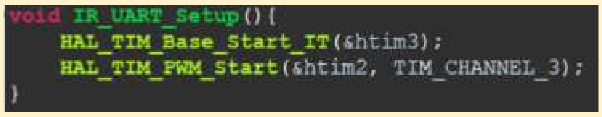
3.2.1 Copy and invert bits: “void copyAndInvertBits()”



```
void copyAndInvertBits(uint8_t source, uint8_t destination[MSG_LENGTH]) {
    for (int i = 0; i < MSG_LENGTH; ++i) {
            // Extract the i-th bit from the source
            uint8_t bit = (source >> i) & 1;

            // Set the i-th element in the destination array
            destination[i] = bit;
    }
}
```

The function takes an 8-bit source value and copies its individual bits into an array named **destination** of size **MSG_LENGTH**. The purpose of the function is to extract each bit from the source value, and then set the corresponding element in the destination array to that bit. Additionally, it inverts the bits, in order to send the byte through IR in the original bit-order.

3.2.2 IR setup: “void IR_UART_Setup()”

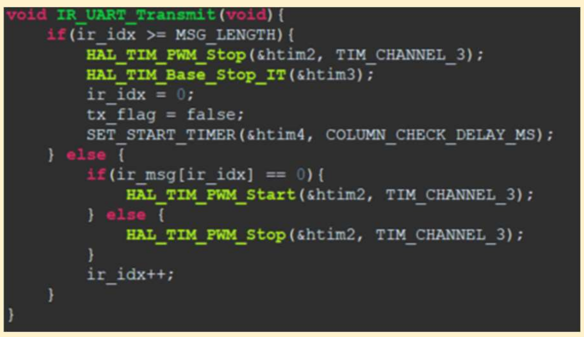


```
void IR_UART_Setup(){
    HAL_TIM_Base_Start_IT(&htim3);
    HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_3);
}
```

This function is used to initiate all the needed timers to establish a UART-like communication through IR.

Timer 3 is started to provide the baud rate (2400 Bits/s). A PWM signal is needed in order to start the IR communication (send the start bit).

3.2.3 IR UART Transmit: “void IR_UART_Transmit(void)”



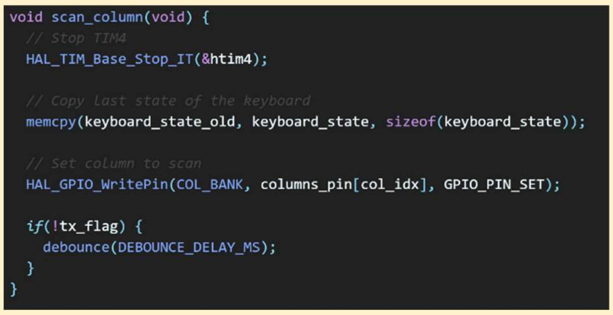
```
void IR_UART_Transmit(void){
    if(ir_idx >= MSG_LENGTH){
        HAL_TIM_PWM_Stop(&htim2, TIM_CHANNEL_3);
        HAL_TIM_Base_Stop_IT(&htim3);
        ir_idx = 0;
        tx_flag = false;
        SET_START_TIMER(&htim4, COLUMN_CHECK_DELAY_MS);
    } else {
        if(ir_msg[ir_idx] == 0){
            HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_3);
        } else {
            HAL_TIM_PWM_Stop(&htim2, TIM_CHANNEL_3);
        }
        ir_idx++;
    }
}
```

The aim of this function is implement a UART-like communication via IR.

If the index **ir_idx** exceeds or equals **MSG_LENGTH**, it stops both timers related to the IR communication, resets the index, sets the transmission flag to false, and initiates the timer for column scan. Otherwise, it starts or stops the PWM on TIM2, channel 3, based on the value of **ir_msg[ir_idx]**, and increments the index.

3.2.4 Keypad Column Scanning: “void scan_column(void)”

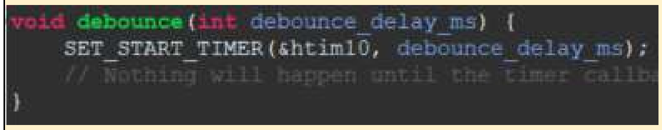
The **scan_column** function orchestrates the scanning of keypad columns, incorporating debouncing and key press detection:



```c
void scan_column(void) {
  // Stop TIM4
  HAL_TIM_Base_Stop_IT(&htim4);

  // Copy last state of the keyboard
  memcpy(keyboard_state_old, keyboard_state, sizeof(keyboard_state));

  // Set column to scan
  HAL_GPIO_WritePin(COL_BANK, columns_pin[col_idx], GPIO_PIN_SET);

  if(!tx_flag) {
    debounce(DEBOUNCE_DELAY_MS);
  }
}
```

- **Timer Stop and State Copying:** Temporarily halts TIM4 and copies the current keyboard state to allow for the comparison needed for key press detection.
- **Column Activation:** Sets the specified column for scanning by activating the corresponding GPIO pin.
- **Debounce:** If not communicating via IR, call debounce.

3.2.5 Debouncing Mechanism: “void debounce(int debounce_delay_ms)”

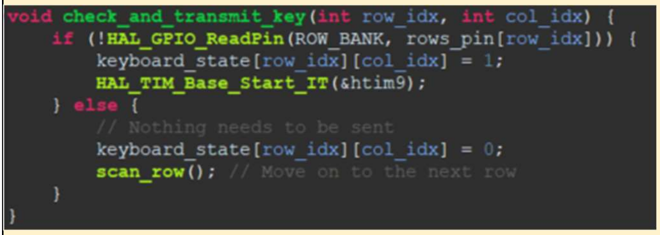


```c
void debounce(int debounce_delay_ms) {
    SET_START_TIMER(&htim10, debounce_delay_ms);
    // Nothing will happen until the timer callb
}
```

The **debounce** function is responsible for implementing a debouncing mechanism to filter out noise in the keypad buttons:

- **Initialization with Timer:** Initiates a timer (TIM10) with a specified delay (**debounce_delay_ms**). This timer is used to introduce a controlled delay, allowing for the stabilization of the keypad signal.

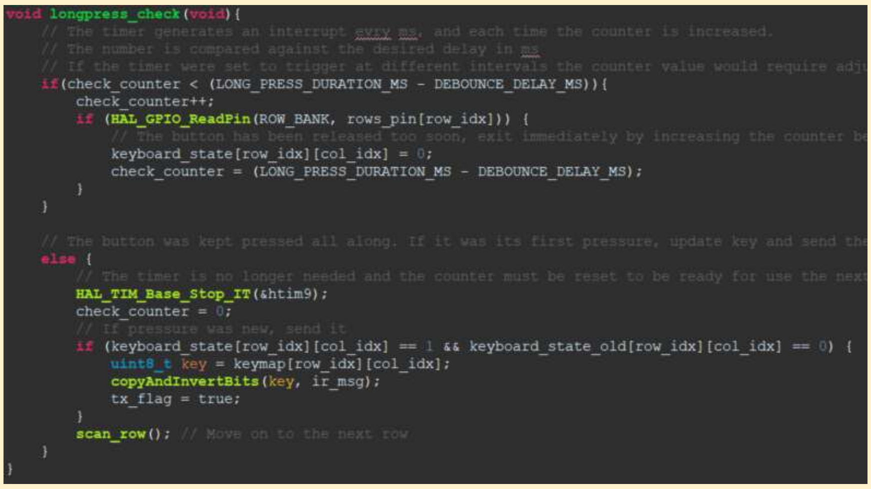
3.2.6 Key Press Detection & Transmission: “void check_and_transmit_key(int row_idx, int col_idx)”



```c
void check_and_transmit_key(int row_idx, int col_idx) {
    if (!HAL_GPIO_ReadPin(ROW_BANK, rows_pin[row_idx])) {
        keyboard_state[row_idx][col_idx] = 1;
        HAL_TIM_Base_Start_IT(&htim9);
    } else {
        // Nothing needs to be sent
        keyboard_state[row_idx][col_idx] = 0;
        scan_row(); // Move on to the next row
    }
}
```

The **check_and_transmit_key** function is responsible for updating the keyboard state, detecting key presses, if key is pressed timer 9 is started to check whether the button is held down for long_press mode. The function calls next scan_row when there is no press detection.

3.2.7 Long press check: “void longpress_check(void)”

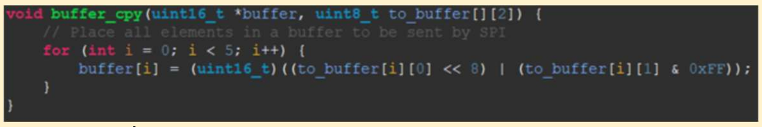


```c
void longpress_check(void){
    // The timer generates an interrupt evry ms, and each time the counter is increased.
    // The number is compared against the desired delay in ms
    // If the timer were set to trigger at different intervals the counter value would require adj
    if(check_counter < (LONG_PRESS_DURATION_MS - DEBOUNCE_DELAY_MS)){
        check_counter++;
        if (HAL_GPIO_ReadPin(ROW_BANK, rows_pin[row_idx])) {
            // The button has been released too soon, exit immediately by increasing the counter be
            keyboard_state[row_idx][col_idx] = 0;
            check_counter = (LONG_PRESS_DURATION_MS - DEBOUNCE_DELAY_MS);
        }
    }

    // The button was kept pressed all along. If it was its first pressure, update key and send the
    else {
        // The timer is no longer needed and the counter must be reset to be ready for use the next
        HAL_TIM_Base_Stop_IT(&htim9);
        check_counter = 0;
        // If pressure was new, send it
        if (keyboard_state[row_idx][col_idx] == 1 && keyboard_state_old[row_idx][col_idx] == 0) {
            uint8_t key = keymap[row_idx][col_idx];
            copyAndInvertBits(key, ir_msg);
            tx_flag = true;
        }
        scan_row(); // Move on to the next row
    }
}
```

The timer generates an interrupt every ms and each time we increase the counter.

The check_counter is compared to the desired delay in ms. If the button has been released too soon we exit setting the check_counter above the upper limit, otherwise if the button has been correctly pressed for enough time we need to check if the pressure is new (we send it) or if we already pressed it (we stop the timer and we reset the counter in order to be ready for the next key pressure). In this case we transmit the scanned button before scanning next row.

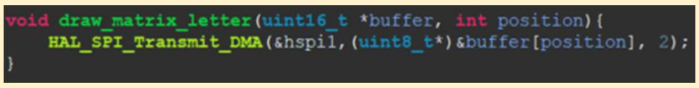
3.2.8 Placing the elements: "void buffer_cpy()"



```c
void buffer_cpy(uint16_t *buffer, uint8_t to_buffer[][2]) {
    // Place all elements in a buffer to be sent by SPI
    for (int i = 0; i < 5; i++) {
        buffer[i] = (uint16_t)((to_buffer[i][0] << 8) | (to_buffer[i][1] & 0xFF));
    }
}
```

The function places all the elements in a buffer. This is needed to send the char representation data through SPI in the correct order.

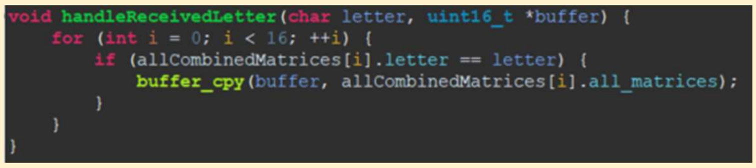
3.2.9 Draw the characters in LED matrix: "void draw_matrix_letter()"



```c
void draw_matrix_letter(uint16_t *buffer, int position){
    HAL_SPI_Transmit_DMA(&hspi1,(uint8_t*)&buffer[position], 2);
}
```

The function sends data via SPI_DMA to the LED matrix controllers.

3.2.10 Handle the receive of a character: "void handleReceivedLetter()"



```c
void handleReceivedLetter(char letter, uint16_t *buffer) {
    for (int i = 0; i < 16; ++i) {
        if (allCombinedMatrices[i].letter == letter) {
            buffer_cpy(buffer, allCombinedMatrices[i].all_matrices);
        }
    }
}
```

The IR-received symbol is compared with the "letter" field in **allCombinedMatrices**. The values at the matching index (corresponding to each column and row) are then arranged in the buffer variable in the correct order for the upcoming SPI transfer.

Professor comments: