Sprints

# DESIGN AN OBSTACLE AVOIDANCE CAR

Youssef Ahmed Abbas Mohamed

Sprints

# Contents

Sprints

# Project Description:

The goal of this project is to design a system for a four-driving wheel car that enables it to autonomously avoid any obstacles encountered in its path. The objective is to develop a robust and efficient obstacle avoidance mechanism that ensures the safe navigation of the car.

## *Hardware Requirements:*

1. ATMEGA32A: This microcontroller will serve as the main control unit for the robot, processing sensor data, executing algorithms, and controlling motor movements.
2. Four Motors (M1, M2, M3, M4): These motors will be responsible for driving the four wheels of the car. The control signals from the microcontroller will regulate their speed and direction to enable movement and maneuverability.
3. One Button (PBUTTON0): This button will be used to change the default direction of rotation of the wheels. Pressing the button will toggle the direction, allowing the robot to move forward or backward as needed.
4. Keypad Button 1 (Start)
5. Keypad Button 2 (Stop)
6. Ultrasonic Sensor: The ultrasonic sensor will be used to detect obstacles in front of the robot. It emits ultrasonic waves and measures the time taken for the waves to bounce back after hitting an object. This information will be used to determine the distance between the robot and the obstacle.
7. LCD: The LCD (Liquid Crystal Display) will provide visual feedback and display important information to the user. It will present real-time data such as distance measurements, operational status, and any relevant messages or notifications.

*Software Requirements:*

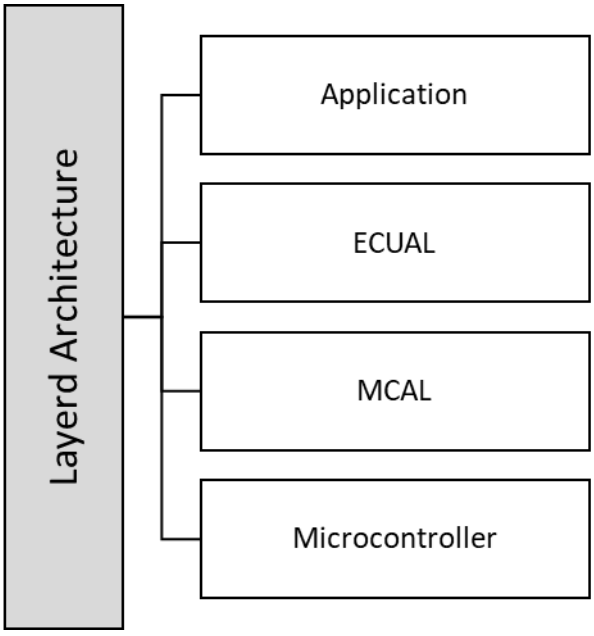
1. Initial Speed and Rotation: The car starts from 0 speed, and the default rotation direction is set to the right.
2. Start/Stop Functionality: Press PB2 to start or stop the robot.
3. Setting Default Rotation: After pressing Start:
    - The LCD will display a centered message in line 1: "Set Def. Rot."
    - The LCD will display the selected option in line 2: "Right"
    - The robot will wait for 5 seconds to choose between Right and Left.
4. Changing Default Rotation:
    - When PB1 is pressed once, the default rotation will be set to Left, and the LCD line 2 will be updated accordingly.
    - When PB1 is pressed again, the default rotation will be set to Right, and the LCD line 2 will be updated.
    - For each press, the default rotation will change, and the LCD line 2 will be updated.
    - After the 5 seconds, the default value of rotation is set.
5. Delay before Movement: The robot will start moving 2 seconds after setting the default direction of rotation.
6. No Obstacles or Objects > 70 cm:
    - The robot will move forward with 30% speed for 5 seconds.
    - After 5 seconds, it will continue moving with 50% speed as long as no obstacles are detected within a distance greater than 70 centimeters.
    - The LCD will display the speed and moving direction in line 1: "Speed: 00% Dir: F/B/R/S" (F: forward, B: backwards, R: rotating, S: stopped).
    - The LCD will display the object distance in line 2: "Dist.: 000 Cm".

7. Obstacles between 30 and 70 cm:
    - The robot will decrease its speed to 30%.
    - The LCD data will be updated accordingly.
8. Obstacles between 20 and 30 cm:
    - The robot will stop and rotate 90 degrees to the right or left based on the chosen configuration.
    - The LCD data will be updated accordingly.
9. Obstacles less than 20 cm:
    - The robot will stop and move backward with 30% speed until the distance is greater than 20 cm and less than 30 cm.
    - The LCD data will be updated accordingly.
    - Then, the system will repeat point 8.

## High Level Design:

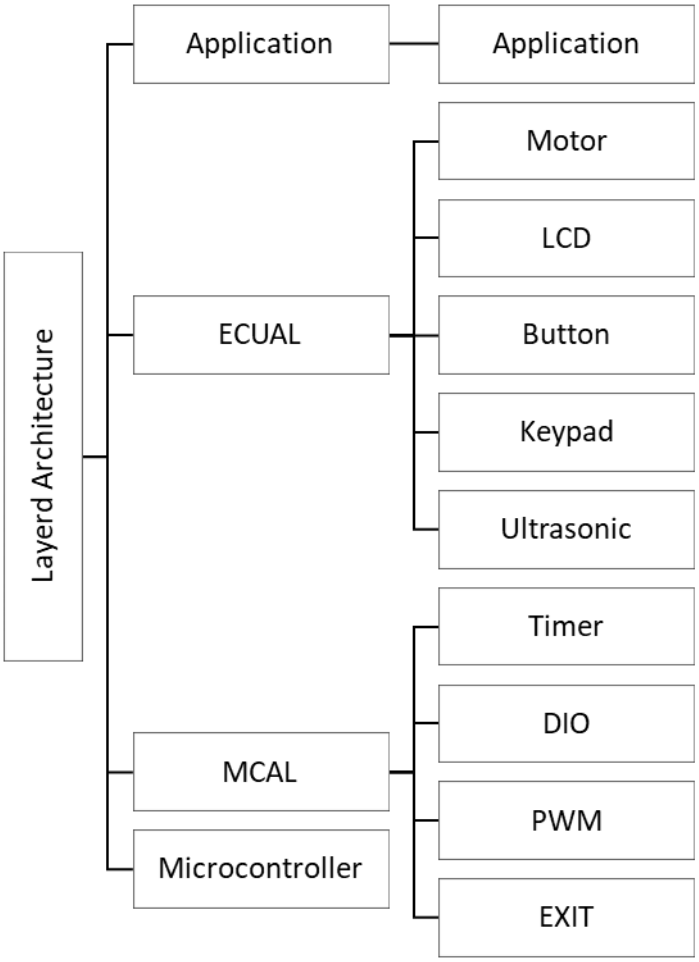
*Layered Architecture:*

1. Application
2. ECUAL
3. MCAL
4. Microcontroller



*Modules Description:*

1. Application:
2. ECUAL
    a. Motor
    b. LCD
    c. Keypad
    d. Button
    e. Ultrasonic
3. MCAL
    a. DIO
    b. Timer
    c. PWM
    d. EINT
4. Microcontroller



*Driver Documentation API's*

Motor:

1. **MOTOR_INIT(const ST_motor_t* motor)**: This function initializes the motor as an output. It takes a pointer to a structure that contains the port and pin numbers for the motor. It returns **MOTOR_OK** if the initialization is successful or **MOTOR_NOK** if there is an error.
2. **MOTOR_FORWARD(const ST_motor_t* motor)**: This function moves the motor forward. It takes a pointer to a structure that contains the port and pin numbers for the motor, as well as the speed of the motor. It returns **MOTOR_OK** if the motor moves forward successfully or **MOTOR_NOK** if there is an error.
3. **MOTOR_BACKWARD(const ST_motor_t* motor)**: This function moves the motor backward. It takes a pointer to a structure that contains the port and pin numbers for the motor, as well as the speed of the motor. It returns **MOTOR_OK** if the motor moves backward successfully or **MOTOR_NOK** if there is an error.
4. **MOTOR_STOP(const ST_motor_t* motor)**: This function stops the motor. It takes a pointer to a structure that contains the port and pin numbers for the motor. It returns **MOTOR_OK** if the motor stops successfully or **MOTOR_NOK** if there is an error.

LCD:

1. **LCD_init(st_config)**: This function initializes the LCD display. It takes a pointer to an LCD configuration structure as input. It returns an error code (**u8_en_lcdErrorsType**) indicating the result of the initialization.
2. **LCD_cmd(st_config, cmd)**: This function sends a command to the LCD display. It takes an LCD configuration structure and a command (**cmd**) as inputs. It returns an error code indicating the result of the command execution.
3. **LCD_char(st_config, data)**: This function writes a character to the LCD display. It takes an LCD configuration structure and a character (**data**) as inputs. It does not return a value.
4. **LCD_clear()**: This function clears the LCD display. It does not take any inputs and returns an error code indicating the result of the clear operation.
5. **LCD_setCursor(u8_row, u8_col)**: This function sets the cursor position on the LCD display. It takes the row number (**u8_row**) and column number (**u8_col**) as inputs. It returns an error code indicating the result of setting the cursor.
6. **LCD_writeString(u8_data)**: This function writes a string of characters to the LCD display. It takes a pointer to the string data (**u8_data**) as an input. It returns an error code indicating the result of writing the string.
7. **LCD_writeSpChar(u8_SpChar)**: This function writes a special character to the LCD display. It takes a special character (**u8_SpChar**) as an input. It returns an error code indicating the result of writing the special character.

Button:

1. **BUTTON_init(u8_a_port, u8_a_pin, en_btnId)**: This function initializes a button for input. It takes the GPIO port number (**u8_a_port**), pin number (**u8_a_pin**), and button ID (**en_btnId**) as inputs. It returns the state of the button (**u8_en_btnStateType**) indicating whether the initialization was successful.

2. **BUTTON_getState(en_btnId)**: This function retrieves the current state of a button. It takes the button ID (**en_btnId**) as an input and returns the state of the button (**u8_en_btnStateType**), which can be either "pressed" or "released".

Keypad:

1. **KEYPAD_init(st_config)**: This function initializes the keypad. It takes a pointer to a keypad configuration structure (**st_config**) as input. It returns an error code (**u8_en_keypadErrorsType**) indicating the result of the initialization.

2. **KEYPAD_read(u8_data)**: This function reads the input from the keypad. It takes a pointer to a variable (**u8_data**) where the read data will be stored. It returns an error code (**u8_en_keypadErrorsType**) indicating the result of the read operation.

Ultrasonic:

- **ultrasonic_init()**: This function initializes the ultrasonic sensor module.

- **ultrasonic_get_distance(uint32 *distance)**: This function measures and returns the distance detected by the ultrasonic sensor.

- **ultrasonic_trigger_measurement()**: This function triggers a measurement using the ultrasonic sensor.

DIO:

1. **DIO_init(configurations)**: This function initializes the DIO pin(s) based on the provided configurations. It takes a pointer to a DIO configuration structure (**configurations**) as input. It returns an error code (**EN_DIO_ERROR_t**) indicating the result of the initialization.

2. **DIO_write(configurations, data)**: This function writes a digital data value (**data**) to the DIO pin(s) based on the provided configurations. It takes a pointer to a DIO configuration structure (**configurations**) and the data value (**data**) as inputs. It returns an error code indicating the result of the write operation.

3. **DIO_read(configurations, data)**: This function reads the digital data value from the DIO pin(s) based on the provided configurations. It takes a pointer to a DIO configuration structure (**configurations**) and a pointer to a variable (**data**) where the read value will be stored. It returns an error code indicating the result of the read operation.

4. **DIO_toggle(configurations)**: This function toggles the state of the DIO pin(s) based on the provided configurations. It takes a pointer to a DIO configuration structure (**configurations**) as input. It returns an error code indicating the result of the toggle operation.

Timer:

1. **TIMER_init(st_config)**: This function initializes the timer driver based on the provided timer configuration. It sets the timer to work in interrupt mode, specifies the initial timer value, selects the timer ID (Timer0, Timer1, Timer2), and sets the timer mode to normal mode. It returns a status code (**u8_en_timerErrorsType**) indicating the success or failure of the function.
2. **TIMER_start(st_config)**: This function starts the timer by configuring the timer clock. It takes a reference to the timer configuration (**st_config**) as input and returns a status code indicating the success or failure of the function.
3. **TIMER_stop(u8_a_timerNum)**: This function halts (stops) the specified timer. It takes the timer type (**u8_a_timerNum**) as input and returns a status code indicating the success or failure of the function.
4. **TIMER_reset(st_config)**: This function resets the timer, making it start again from the initial value. It takes a reference to the timer configuration (**st_config**) as input and returns a status code indicating the success or failure of the function.
5. **TIMER_setCallBack(a_timerCallBack, u8_a_timerNum)**: This function sets a callback function to be called in the application after the timer completes its job. It takes a pointer to the callback function (**a_timerCallBack**) and the timer type (**u8_a_timerNum**) as inputs. It returns a status code indicating the success or failure of the function.

EINT:

1. **EXI_enablePIE(u8_a_interruptId, u8_a_senseControl)**: This function enables the specified external interrupt (**u8_a_interruptId**) with the provided sense control (**u8_a_senseControl**). It allows the external interrupt to trigger an interrupt request. It returns a status code (**u8**) indicating the success or failure of enabling the interrupt.
2. **EXI_disablePIE(u8_a_interruptId)**: This function disables the specified external interrupt (**u8_a_interruptId**). It prevents the external interrupt from triggering an interrupt request. It returns a status code (**u8**) indicating the success or failure of disabling the interrupt.
3. **EXI_intSetCallBack(u8_a_interruptId, pf_a_interruptAction)**: This function sets a callback function (**pf_a_interruptAction**) to be executed when the specified external interrupt (**u8_a_interruptId**) occurs. The callback function is typically implemented in the application code and is executed in response to the external interrupt. It returns a status code (**u8**) indicating the success or failure of setting the callback function.

## Low Level Design:

*Flowchart:*

Motor:

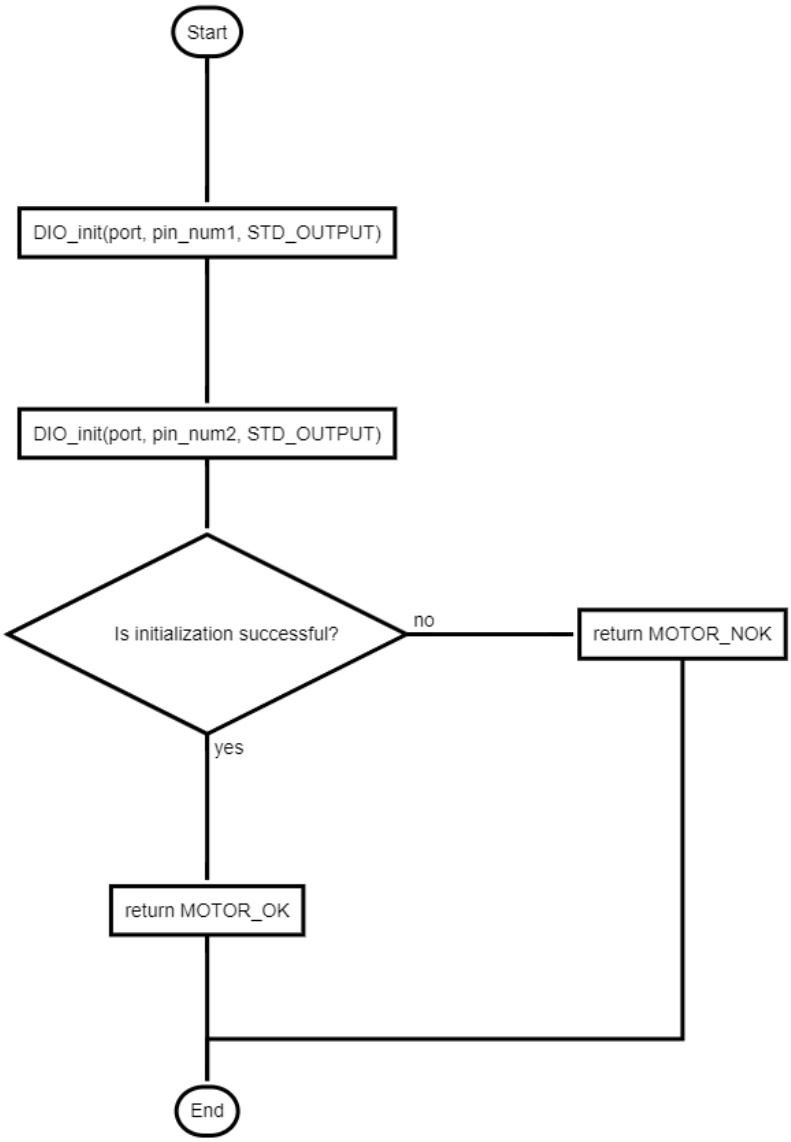



*Figure 1 Motor INIT*

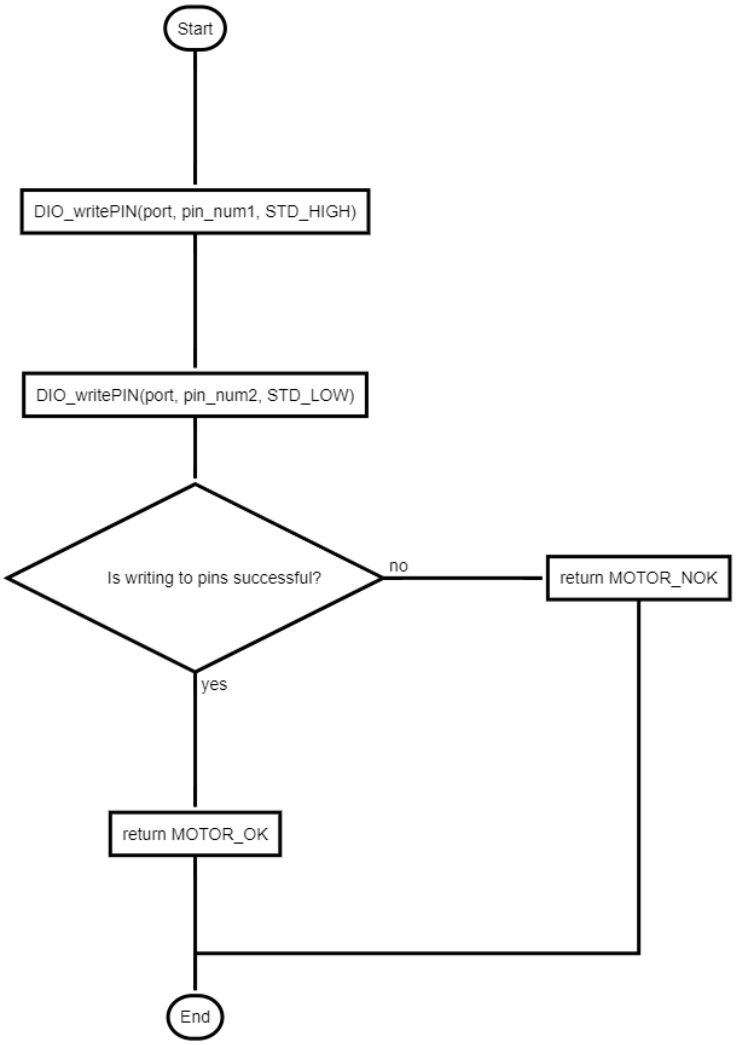



*Figure 2 Motor Forward*

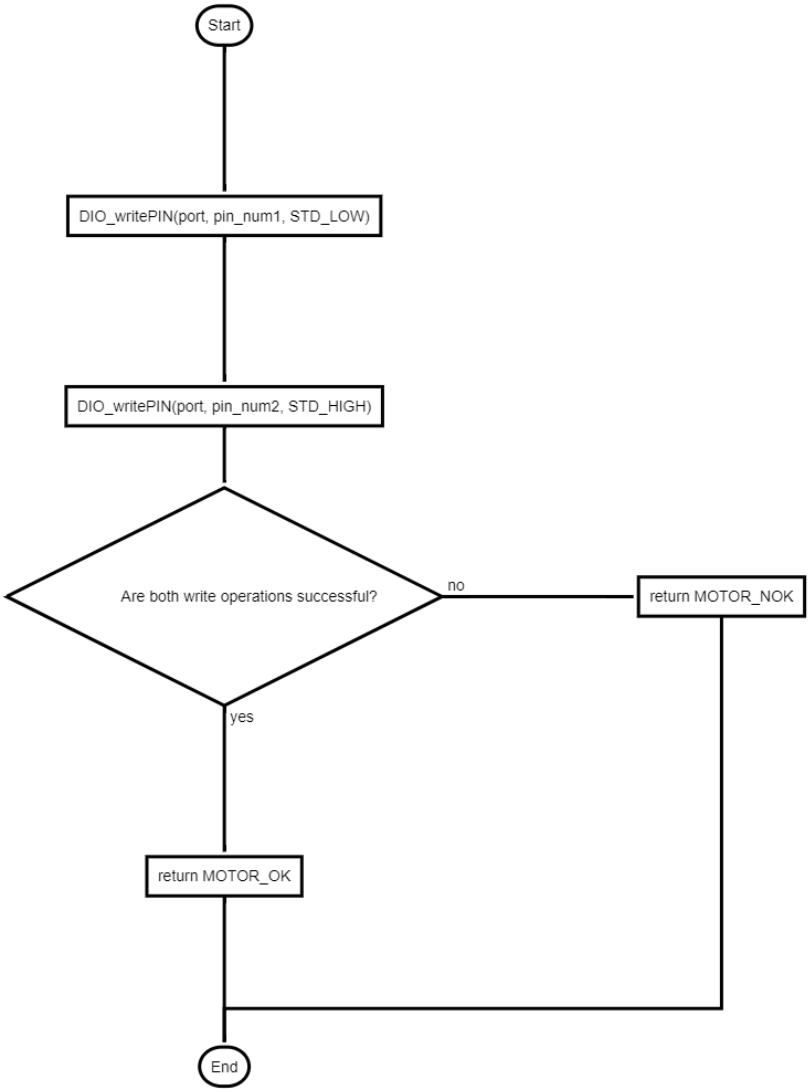



*Figure 3 Motor Backward*

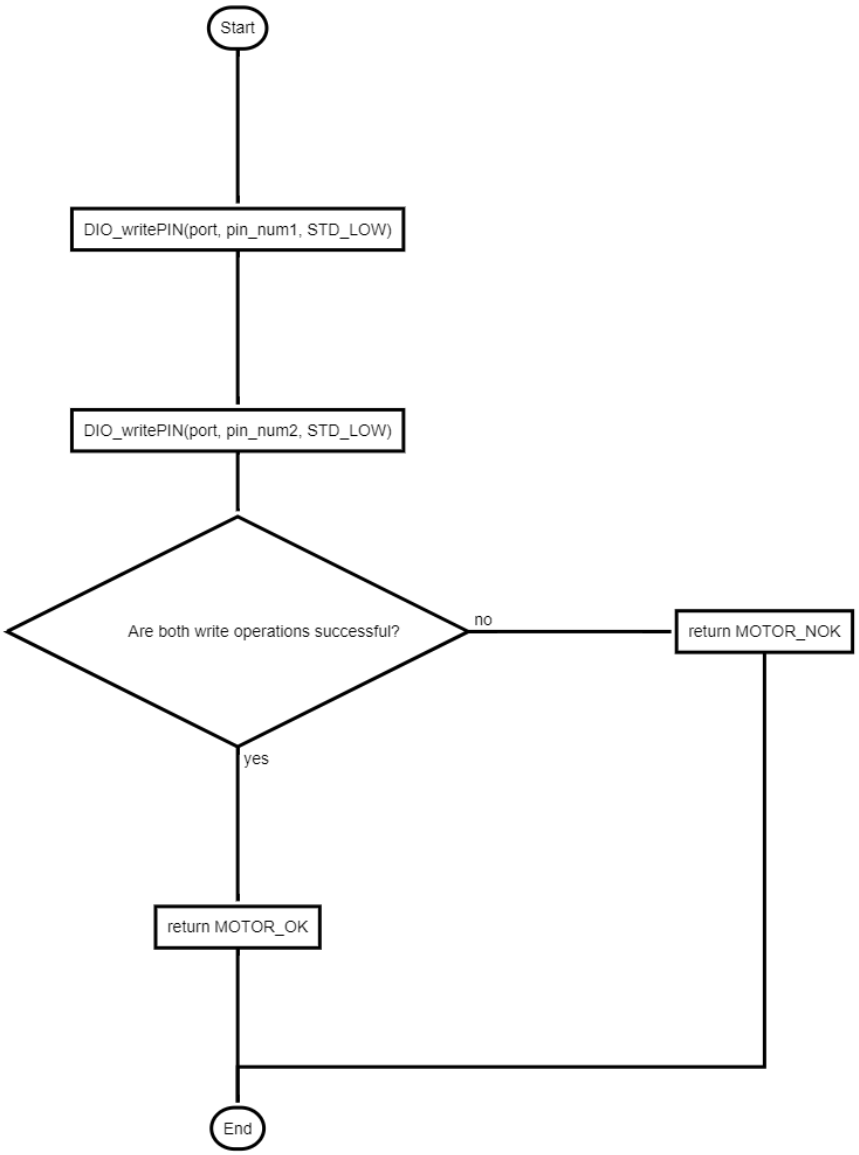



*Figure 4 Motor Stop*

LCD:

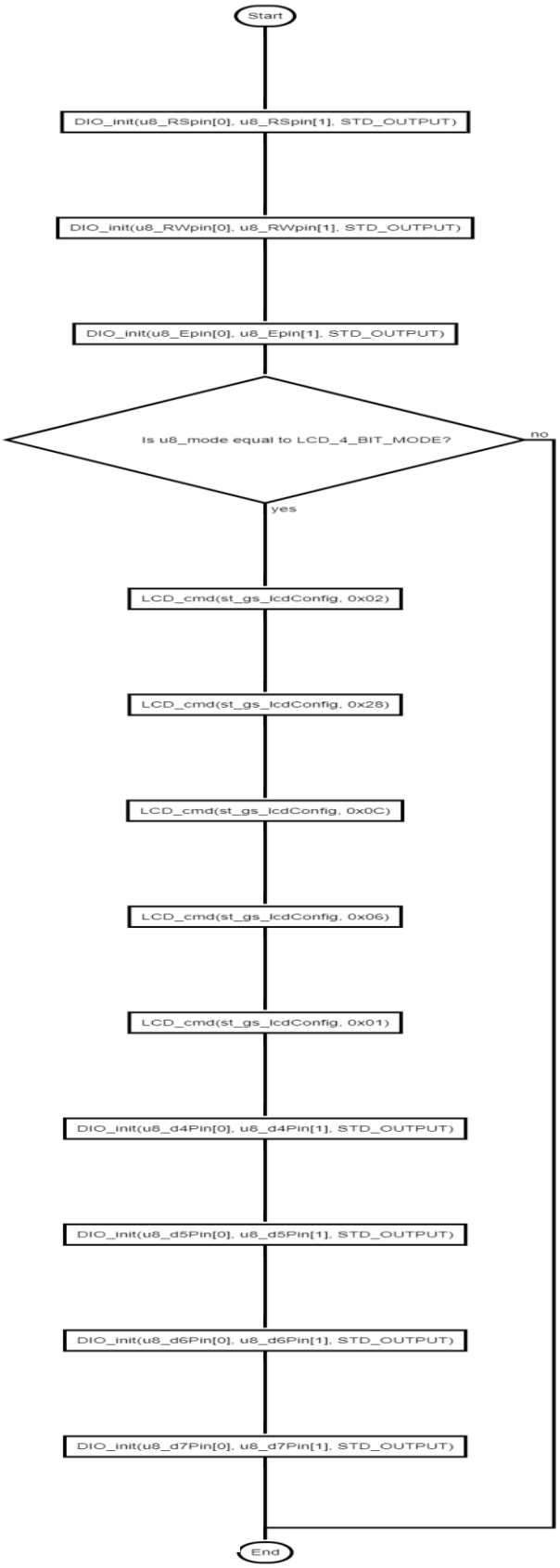



*Figure 5 LCD_init*

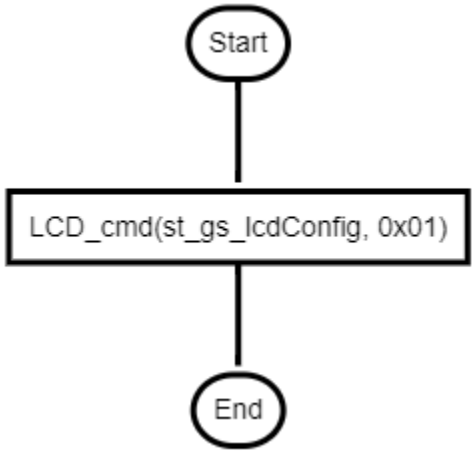



*Figure 6 LCD_clear*

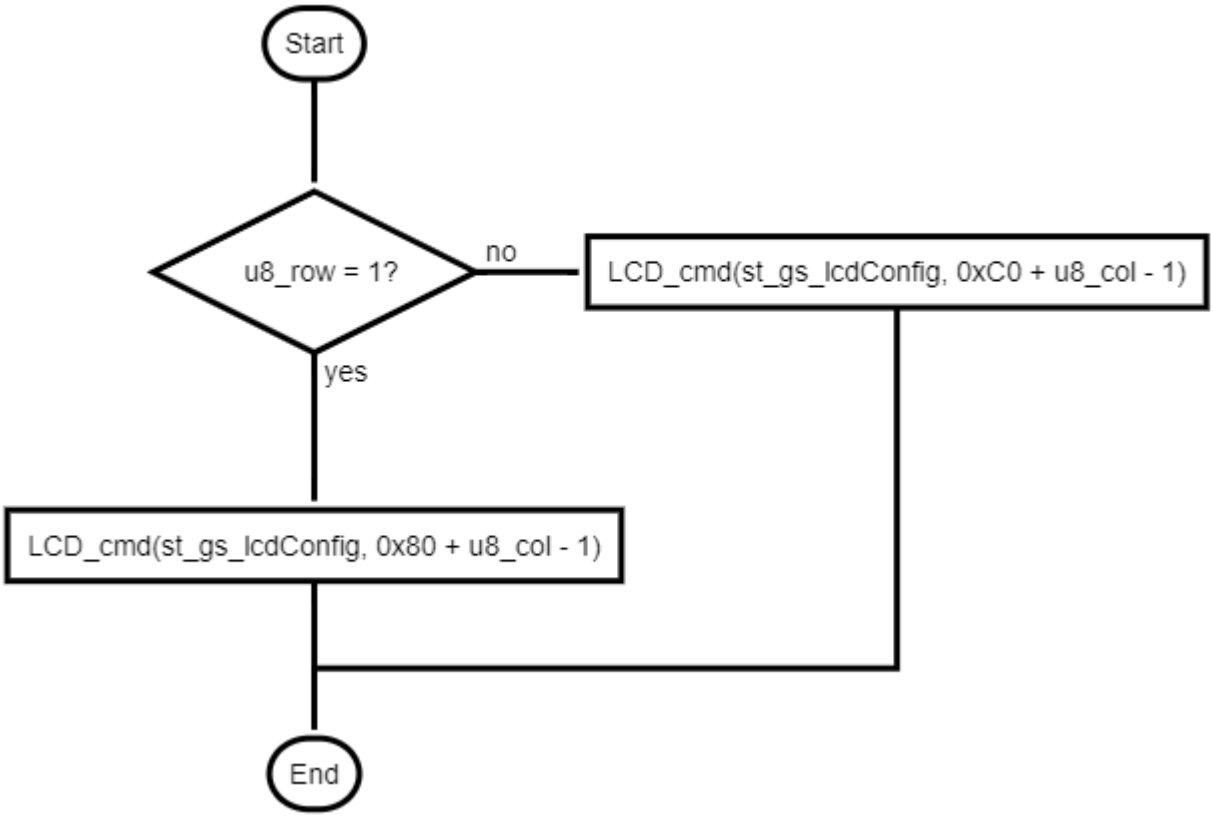



*Figure 7 LCD_setCursor*

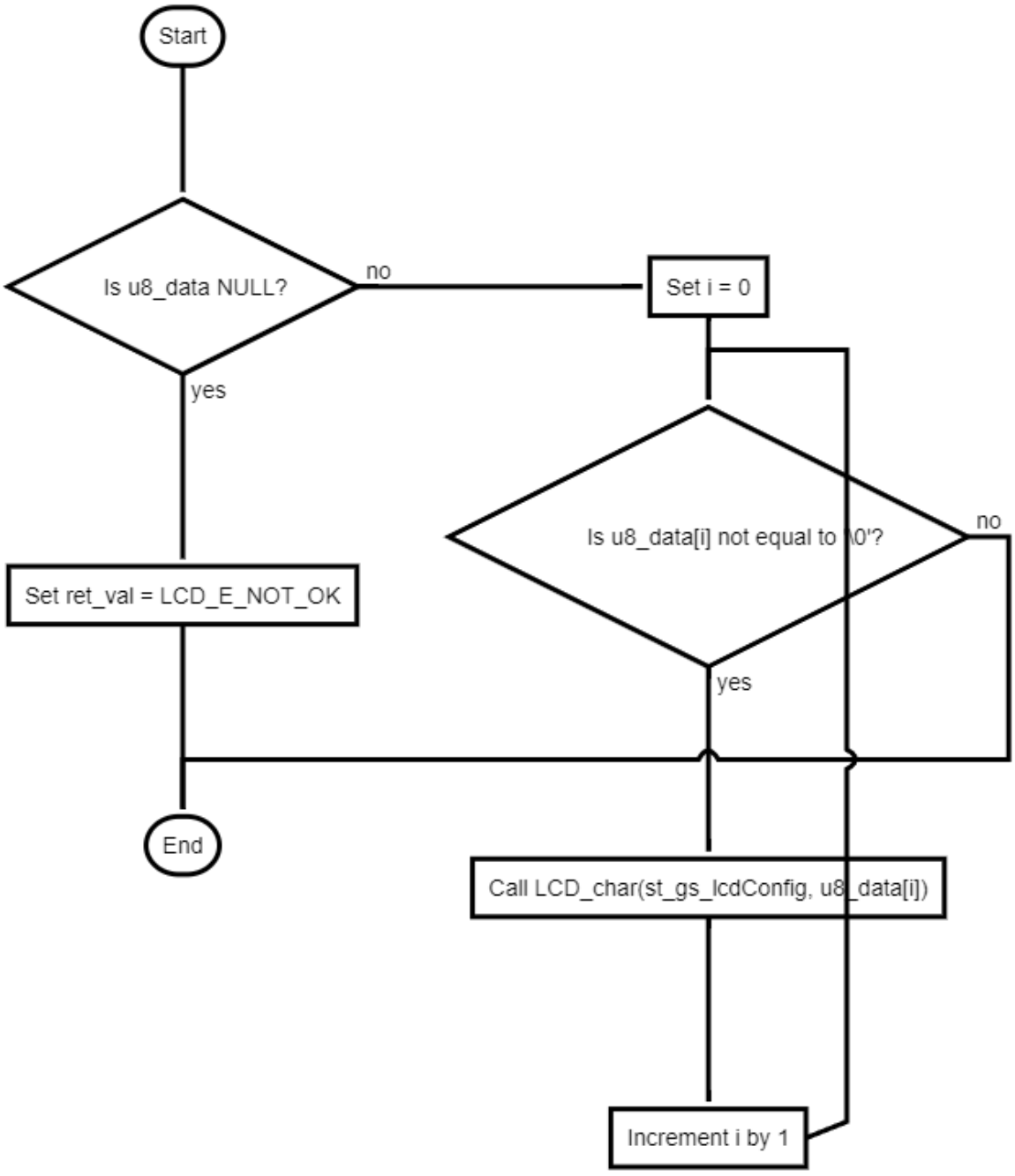



*Figure 8 LCD_writeString*

Button:

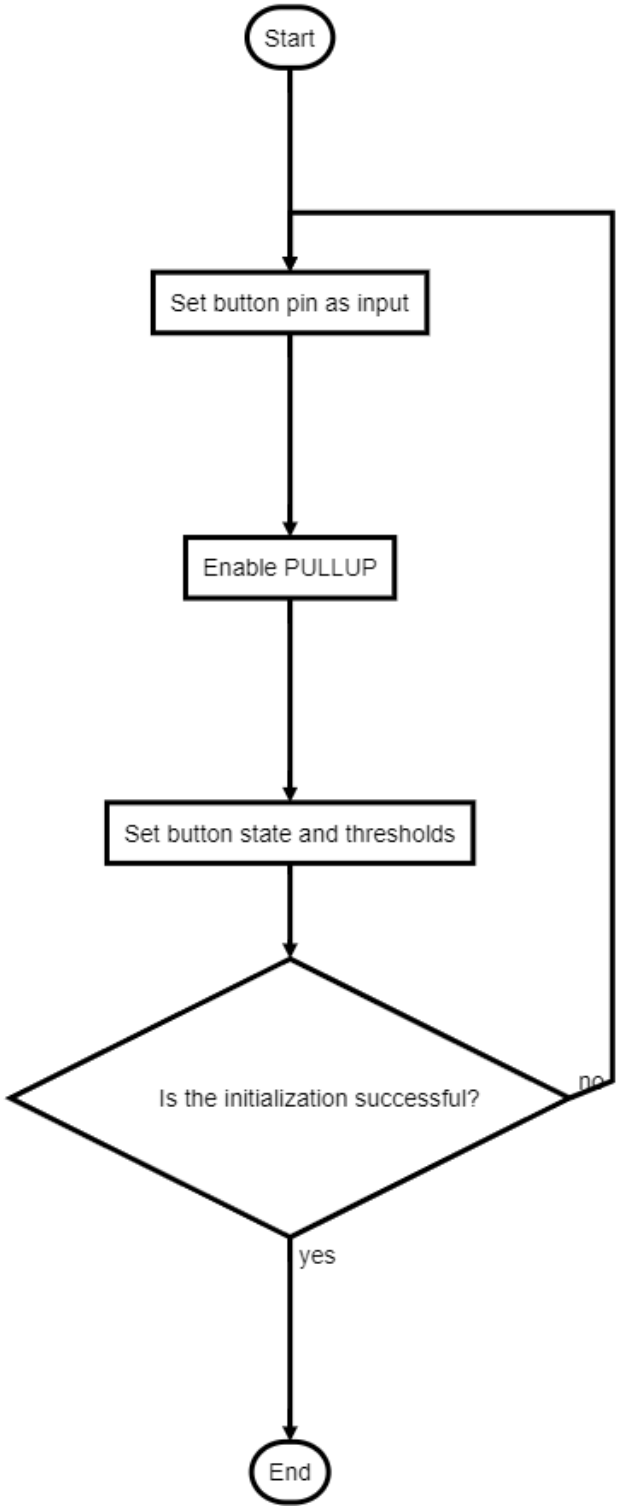



*Figure 9 BTN_init*

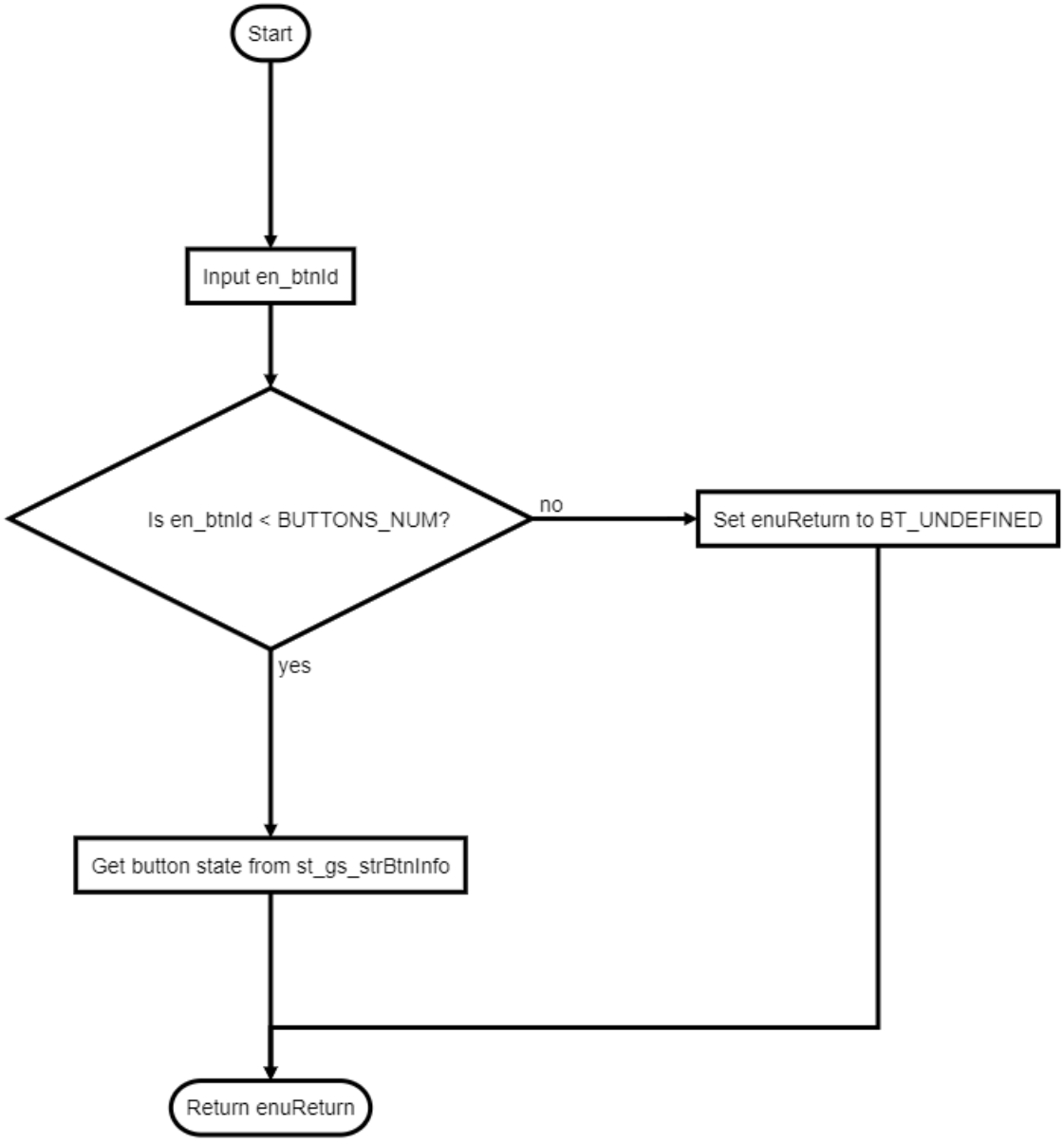



*Figure 10 BUTTON_getState*

Keypad:

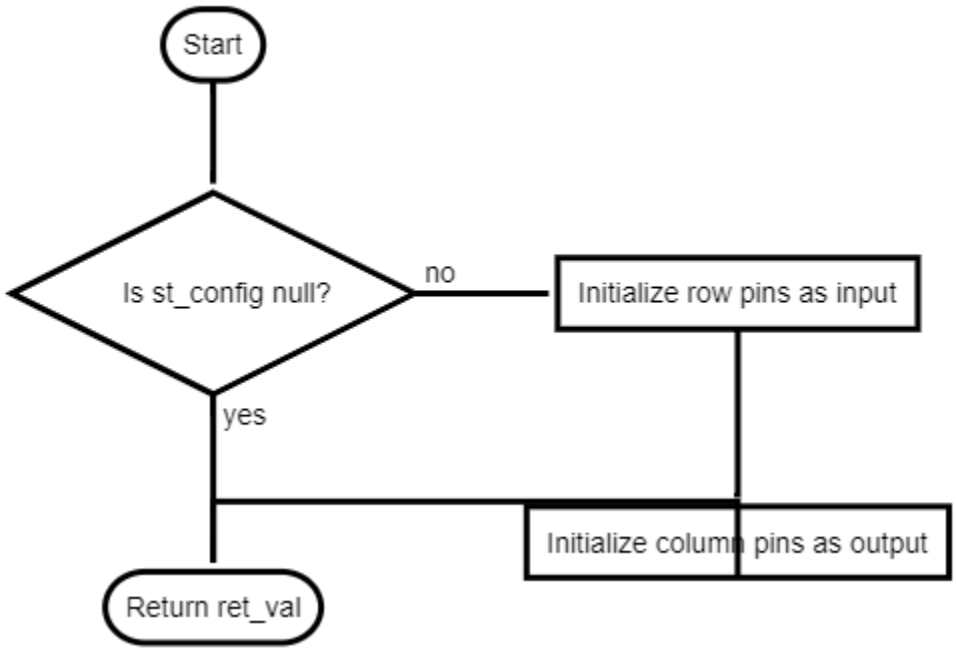



*Figure 11 KEYPAD_init*

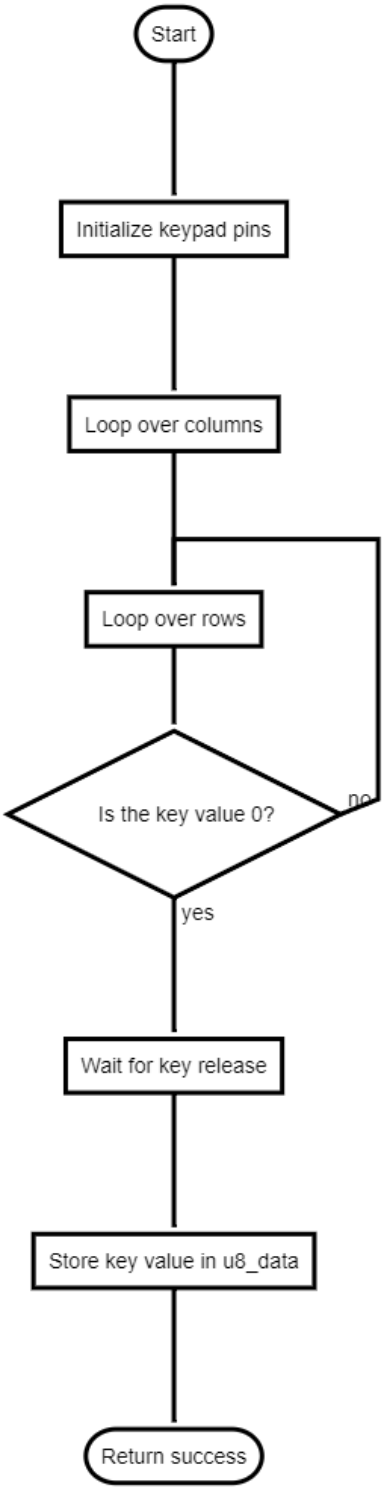



*Figure 12 KEYPAD_read*

*Pre-compiling & Linking Configuration:*

LCD

```
typedef struct
{
	uint8_t u8_mode;
	uint8_t u8_d0Pin[2];
	uint8_t u8_d1Pin[2];
	uint8_t u8_d2Pin[2];
	uint8_t u8_d3Pin[2];
	uint8_t u8_d4Pin[2];
	uint8_t u8_d5Pin[2];
	uint8_t u8_d6Pin[2];
	uint8_t u8_d7Pin[2];

	uint8_t u8_RSpin[2];
	uint8_t u8_RWpin[2];
	uint8_t u8_Epin[2];
}st_lcdConfigType;


typedef uint8_t u8_en_lcdModeType;

#define LCD_4_BIT_MODE                          ((u8_en_lcdModeType)0x00)
#define LCD_8_BIT_MODE                          ((u8_en_lcdModeType)0x01)
#define LCD_INVALID_MODE                ((u8_en_lcdModeType)0x02)


typedef uint8_t u8_en_lcdErrorsType;

#define LCD_E_OK                                ((u8_en_lcdErrorsType)0x00)
#define LCD_E_NOT_OK                    ((u8_en_lcdErrorsType)0x05)


typedef uint8_t u8_en_lcdSpCharType;


#define LCD_BELL                                ((u8_en_lcdSpCharType)0X03)
```

Keypad

```
typedef struct
{
	uint8_t u8_row1Pin[2];
	uint8_t u8_row2Pin[2];
	uint8_t u8_row3Pin[2];
	uint8_t u8_col1Pin[2];
	uint8_t u8_col2Pin[2];
	uint8_t u8_col3Pin[2];
	uint8_t u8_col4Pin[2];
}st_keypadConfigType;


typedef uint8_t u8_en_keypadErrorsType;
#define KEYPAD_E_OK                                    ((u8_en_keypadErrorsType)0x00)
#define KEYPAD_E_NOT_OK                                ((u8_en_keypadErrorsType)0x07)
```