Obstacle Avoidance Car Design

Obstacle Avoidance Car Project

1. Project Introduction

This project involves developing a robot using four motors, two buttons, and four LEDs. The robot moves in a continuous path so that the car avoid any object in front.

1.1. Car Components

- Four motors (M1, M2, M3, M4)
- One button to change the default direction (PB0)
- Four LEDs (LED1, LED2, LED3, LED4)
- LCD
- Keypad
- Ultrasonic

1.2. System Requirements

1. The car starts initially from 0 speed.
2. The default rotation direction is to the right.
3. When PB2 is pressed to start or stop the robot.
4. After pressing Start:
    1. The LCD will display a centered message in line1 “Set Def.Rot”.
    2. The LCD will display a centered message in line2 “Right”.
    3. The robot will wait for 5 second to choose between Right and left.
    4. When PB1 is pressed once, the default rotation will be left and the LCD line2 will be updated.
    5. When PB1 is pressed again, the default rotation will be Right and the LCD line2 will be updated.
    6. For each press the default rotation will changed and the LCD line2 is updated.
    7. After the 5 seconds the default value of rotation is set.
5. The robot will move after 2 seconds from setting the default direction of rotation.

1.3. Assumptions

- 4WD Robot Specifications - [4WD Complete Mini Plastic Robot Chassis Kit]
- For the Robot to rotate in place around its pivot point we calculated that:
    - Left motors going forward, Right motors going backward
    - Rotation frequency: 100 Hz
    - Rotation duration: 620 ms
    - Rotation duty cycle: 50%

2. High Level Design

2.1. System Architecture

2.1.1. Definition

Layered Architecture (*Figure 1*) describes an architectural pattern composed of several separate horizontal layers that function together as a single unit of software.

Microcontroller Abstraction Layer (*MCAL*) is a software module that directly accesses on-chip MCU peripheral modules and external devices that are mapped to memory, and makes the upper software layer independent of the MCU.

Hardware Abstraction Layer (*HAL*) is a layer of programming that allows a computer OS to interact with a hardware device at a general or abstract level rather than at a detailed hardware level.

Service Layer (*SL*) is the topmost layer of Basic Software Architecture. The service layer constitutes an operating system, delay_functions, exti_handler, which runs from the application layer to the microcontroller at the bottom.

2.1.2. Layered Architecture

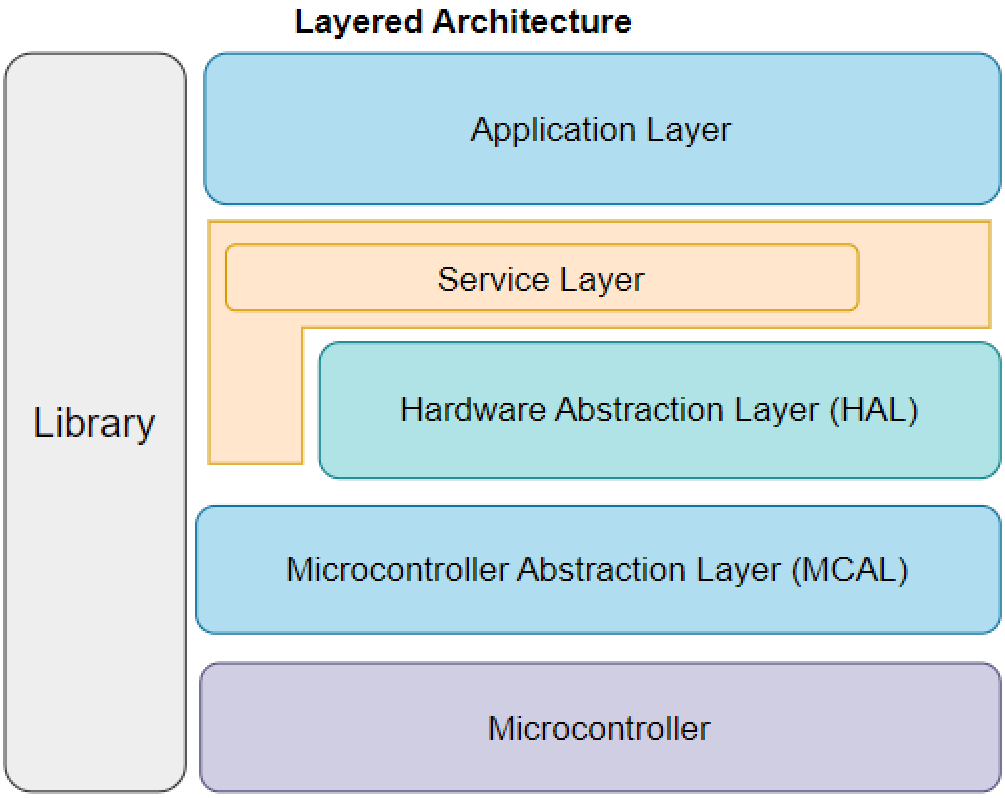


Figure 1. Layered Architecture Design

2.2. Modules Description

2.2.1. DIO (Digital Input/Output) Module

The *DIO* module is responsible for reading input signals from the system's sensors (such as buttons) and driving output signals to the system's actuators (such as *LEDs*). It provides a set of APIs to configure the direction and mode of each pin (input/output, pull-up/down resistor), read the state of an input pin, and set the state of an output pin.

2.2.2 EXI Module

The *EXI* (External Interrupt) module is responsible for detecting external events that require immediate attention from the microcontroller, such as a button press. It provides a set of APIs to enable/disable external interrupts for specific pins, set the interrupt trigger edge (rising/falling/both), and define an interrupt service routine (*ISR*) that will be executed when the interrupt is triggered.

2.2.3 TIMER Module

The *TIMER* module is responsible for generating timing events that are used by other modules in the system. It provides a set of APIs to configure the timer clock source and prescaler, set the timer mode (count up/down), set the timer period, enable/disable timer interrupts, and define an ISR that will be executed when the timer event occurs.

2.2.4 LED Module

The *LED* (Light Emitting Diode) module is responsible for controlling the state of the system's *LEDs*. It provides a set of APIs to turn on/off each *LED* and toggle its state.

2.2.5 BTN Module

The *BTN* (Button) module is responsible for reading the state of the system's buttons. It provides a set of APIs to enable/disable button interrupts, set the button trigger edge (rising/falling/both), and define an ISR that will be executed when a button press is detected.

2.2.6 DCM Module

The *DCM* (DC Motor) module is responsible for controlling the speed and direction of the system's DC motors. It provides a set of APIs to set the speed and direction of each motor, and to stop all motors. It also uses the *TIMER* module to generate *PWM* (Pulse Width Modulation) signals that control the motor speed.

2.2.7 ICU Module

The *ICU* (input capture) function is used in many applications such as: Pulse width measurement, Period measurement, Capturing the time of an event

2.2.8 Ultrasonic Module

The *Ultrasonic HC-SR04* Sensor Module is a very popular sensor, it is used in many applications where measuring distance and detecting objects are required. Its works on the same principle as a radar system. Ultrasonic sensors work by emitting high-frequency sound waves that is not heard by humans. It sends out a high-frequency sound pulse from the transmitter and then the receiver receives this sound when it reflects back from any object surface. This way the sensors detect objects. It can measure distance or detect objects in the range of 2cm-400cm.

2.2.9 LCD Module

The *LiquidCrystal* allows you to control LCD displays that are compatible with the Hitachi HD44780 driver. There are many of them out there, and you can usually tell them by the 16-pin interface.

2.2.10 Keypad Module

The *keypad* module comprises a **key panel, an electrode pad, a circuit board and a spacer**. The key panel is marked with a plurality of key characters. The electrode pad is arranged on an inner side of the key panel.

2.2.11 PWM Module

The PWM stands for pulse width modulation which consists of a square wave with the help of which we can control the up or high time. It is simple but digital way to control the digital signals that we use to vary the energy that is send to a load or to encode information within the signal. The wave oscillation can be limit by using PWM signal maximum and minimum voltage values. The space between the maximum and minimum value is called amplitude. A cycle is the interval of the wave where you can find one full repetition the time a cycle takes to finish is called period. The frequency is 1 over a time period which gives you how many cycles are in a time unit.

2.2.7. Design

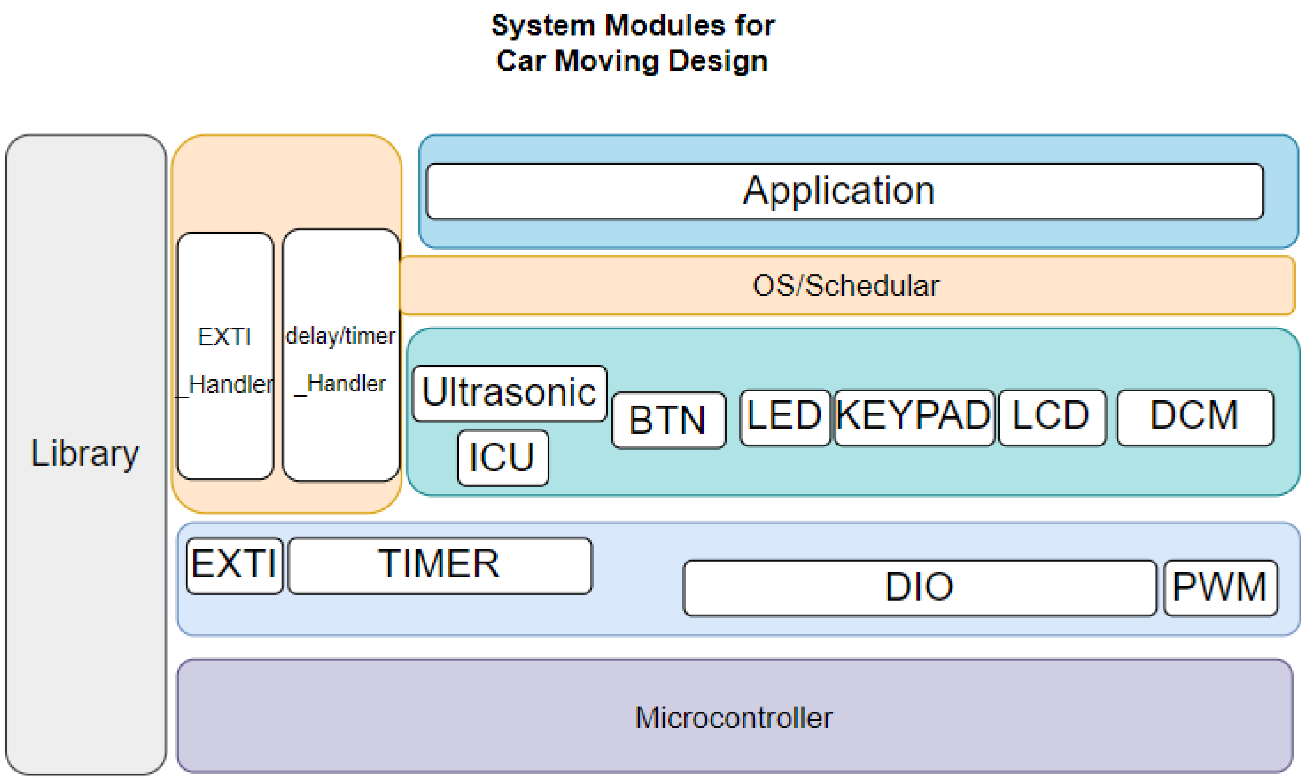


Figure 3. System Modules Design

2.3. Drivers’ Documentation (APIs)

2.3.1 Definition

An *API* is an *Application Programming Interface* that defines a set of *routines*, *protocols* and *tools* for creating an application. An *API* defines the high level interface of the behavior and capabilities of the component and its inputs and outputs.

An *API* should be created so that it is generic and implementation independent. This allows for the API to be used in multiple applications with changes only to the implementation of the API and not the general interface or behavior.

2.3.2. MCAL APIs

2.3.2.1. DIO Driver

```
| Enumeration of possible DIO ports
typedef enum EN_DIO_PORT_T
{
      PORT_A, /*!< Port A */
      PORT_B, /*!< Port B */
      PORT_C, /*!< Port C */
      PORT_D  /*!< Port D */
}EN_DIO_PORT_T;

| Enumeration for DIO direction.
|
| This enumeration defines the available directions for a
| Digital Input/Output (DIO) pin.
|
| Note
|     This enumeration is used as input to the DIO driver functions
|     for setting the pin direction.
|
typedef enum EN_DIO_DIRECTION_T
{
      DIO_IN = 0,       /**< Input direction */
      DIO_OUT = 1       /**< Output direction */
} EN_DIO_DIRECTION_T;

| Enumeration of DIO error codes
typedef enum EN_DIO_ERROR_T
{
      DIO_OK,     /**< Operation completed successfully */
      DIO_ERROR   /**< An error occurred during the operation */
} EN_DIO_ERROR_T;
```

```
| Initializes a pin of the DIO interface with a given direction
|
| Parameters
|        [in] u8_a_pinNumber  The pin number of the DIO interface to initialize
|        [in] en_a_portNumber The port number of the DIO interface to initialize
|                                (PORT_A, PORT_B, PORT_C or |PORT_D)
|        [in] en_a_direction  The direction to set for the pin
|                                (DIO_IN or DIO_OUT)
| Returns
|        An EN_DIO_ERROR_T value indicating the success or failure of the
|        operation (DIO_OK if the operation succeeded, DIO_ERROR otherwise)
|
EN_DIO_ERROR_T DIO_init(u8 u8_a_pinNumber, EN_DIO_PORT_T en_a_portNumber,
EN_DIO_DIRECTION_T en_a_direction);


| Reads the value of a pin on a port of the DIO interface
|
| Parameters
|        [in] u8_a_pinNumber  The pin number to read from the port
|        [in] en_a_portNumber The port number to read from
|                                (PORT_A, PORT_B, PORT_C or |PORT_D)
|        [out] u8_a_value     Pointer to an unsigned 8-bit integer where
|                                the value of the pin will be stored
| Returns
|        An EN_DIO_ERROR_T value indicating the success or failure of the
|        operation (DIO_OK if the operation succeeded, DIO_ERROR otherwise)
|
EN_DIO_ERROR_T DIO_read(u8 u8_a_pinNumber, EN_DIO_PORT_T en_a_portNumber, u8 *
u8_a_value);


| Writes a digital value to a specific pin in a specific port.
|
| Parameters
|        [in] u8_a_pinNumber  The pin number to write to
|        [in] en_a_portNumber The port number to write to
|                                (PORT_A, PORT_B, PORT_C or |PORT_D)
|        [in] u8_a_value     The digital value to write
|                                (either DIO_U8_PIN_HIGH or DIO_U8_PIN_LOW)
| Returns
|        EN_DIO_ERROR_T Returns DIO_OK if the write is successful,
|        DIO_ERROR otherwise.
|
EN_DIO_ERROR_T DIO_write(u8 u8_a_pinNumber, EN_DIO_PORT_T en_a_portNumber, u8
u8_a_value);
```

```
| Initializes a port of the DIO interface with a given direction and mask
|
| Parameters
|     [in] en_a_portNumber The port number of the DIO interface to initialize
|                              (PORT_A, PORT_B, PORT_C or PORT_D)
|     [in] en_a_dir       The direction to set for the port (INPUT or OUTPUT)
|     [in] u8_a_mask      The mask to use when setting the DDR of the port
|                              (DIO_NO_MASK, DIO_MASK_BITS_n..)
| Returns
|     An EN_DIO_ERROR_T value indicating the success or failure of the
|     operation (DIO_OK if the operation succeeded, DIO_ERROR otherwise)
|
EN_DIO_ERROR_T DIO_portInit(EN_DIO_PORT_T en_a_portNumber, EN_DIO_DIRECTION_T
en_a_dir, u8 u8_a_mask);

| Writes a byte to a port of the DIO interface
|
| Parameters
|     [in] en_a_portNumber  The port number of the DIO interface to write to
|                              (PORT_A, PORT_B, PORT_C or PORT_D)
|     [in] u8_a_portValue   The byte value to write to the port
|                              (DIO_U8_PORT_LOW, DIO_U8_PORT_HIGH)
|     [in] u8_a_mask      The mask to use when setting the PORT of the port
|                              (DIO_NO_MASK, DIO_MASK_BITS_n..)
| Returns
|     An EN_DIO_ERROR_T value indicating the success or failure of the operation
|     (DIO_OK if the operation succeeded, DIO_ERROR otherwise)
|
EN_DIO_ERROR_T DIO_portWrite(EN_DIO_PORT_T en_a_portNumber, u8 u8_a_portValue,
u8 u8_a_mask);

| Toggles the state of the pins of a port of the DIO interface
|
| Parameters
|     [in] en_a_portNumber  The port number of the DIO interface to toggle
|                              (PORT_A, PORT_B, PORT_C or PORT_D)
|     [in] u8_a_mask        The mask to use when toggling the PORT of the port
|                              (DIO_NO_MASK, DIO_MASK_BITS_n..)
| Returns
|     An EN_DIO_ERROR_T value indicating the success or failure of the operation
|     (DIO_OK if the operation succeeded, DIO_ERROR otherwise)
|
EN_DIO_ERROR_T DIO_portToggle(EN_DIO_PORT_T en_a_portNumber, u8 u8_a_mask);
```

2.3.2.2. EXI Driver

```
| The function enables a specific external interrupt with a specified sense control.
|
| Parameters
|      [in] u8_a_interruptId specifies the ex. interrupt ID
|                                  (EXI_U8_INT0, EXI_U8_INT1, or EXI_U8_INT2)
|      [in] u8_a_senseControl specifies sense control for the EXI.
|                                  (EXI_U8_SENSE_LOW_LEVEL,...)
| Return
|      If the function executes successfully, it will return STD_OK (0)
|      If there is an error, it will return STD_NOK (1).
|
u8 EXI_enablePIE(u8 u8_a_interruptId, u8 u8_a_senseControl);

| The function disables a specified external interrupt.
|
| Parameters
|      [in] u8_a_interruptId interrupt ID to disable. It should be a
|            value between 0 and 2, where 0 represents INT0, 1 represents INT1, and 2
|            represents INT2.
| Return
|     STD_OK if the function executed successfully, and STD_NOK if there was an error
|
u8 EXI_disablePIE(u8 u8_a_interruptId);

| function sets a callback function for a specific interrupt and returns an error
state.
|
| Parameters
|      [in] u8_a_interruptId An unsigned 8-bit integer representing
|            the ID of the interrupt. It should be in the range of 0 to 2, inclusive.
|      [in] pf_a_interruptAction A pointer to a function that will be
|            executed when the specified interrupt occurs.
| Return
|     a u8 value which represents the error state. It can be either
|     STD_OK (0) or STD_NOK (1).
|
u8 EXI_intSetCallBack(u8 u8_a_interruptId, void (*pf_a_interruptAction)(void))
```

2.3.2.3. TIMER Driver

```
| Initializes timer0 at normal mode
|
| This function initializes/selects the timer_0 normal mode for the
|  timer, and enables the ISR for this timer.
| Parameters
|            [in] en_a_interrputEnable value to set the interrupt
|                                      bit for timer_0 in the TIMSK reg.
|             [in] **u8_a_shutdownFlag double pointer, acts as a main switch for
|                                      timer0 operations.
| Return
|      An EN_TIMER_ERROR_T value indicating the success or failure of
|             the operation (TIMER_OK if the operation succeeded, TIMER_ERROR
|             otherwise)
|
EN_TIMER_ERROR_T TIMER_timer0NormalModeInit(EN_TIMER_INTERRPUT_T
en_a_interrputEnable, u8 ** u8_a_shutdownFlag);



| Creates a delay using timer_0 in overflow mode
|
| This function Creates the desired delay on timer_0 normal mode.
| Parameters
|            [in] u16_a_interval value to set the desired delay.
| Return
|      An EN_TIMER_ERROR_T value indicating the success or failure of
|             the operation (TIMER_OK if the operation succeeded, TIMER_ERROR
|             otherwise)
|
EN_TIMER_ERROR_T TIMER_timer0Delay(u16 u16_a_interval);



| Start the timer by setting the desired prescaler.
|
| This function sets the prescaler for timer_0.
| Parameters
|            [in] u16_a_prescaler value to set the desired prescaler.
| Return
|      An EN_TIMER_ERROR_T value indicating the success or failure of
|             the operation
|              (TIMER_OK if the operation succeeded, TIMER_ERROR otherwise)
|
EN_TIMER_ERROR_T TIMER_timer0Start(u16 u16_a_prescaler);
```

```
| Stop the timer by setting the prescaler to be 000--> timer is stopped.
|
| This function clears the prescaler for timer_0.
|
| Return
|     void
|
void TIMER_timer0Stop(void);

| Initializes timer2 at normal mode
|
| This function initializes/selects the timer_2 normal mode for the
|  timer, and enables the ISR for this timer.
| Parameters
|            [in] en_a_interrputEnable value to set
|            the interrupt bit for timer_2 in the TIMSK reg.
|             [in] **u8_a_shutdownFlag double pointer, acts as a main switch for
|                                         timer0 operations.
| Return
|     An EN_TIMER_ERROR_T value indicating the success or failure of
|            the operation (TIMER_OK if the operation succeeded, TIMER_ERROR
|            otherwise)
|
EN_TIMER_ERROR_T TIMER_timer2NormalModeInit(EN_TIMER_INTERRPUT_T
en_a_interrputEnable, u8 **u8_a_shutdownFlag);

| Stop the timer by setting the prescaler to be 000--> timer is stopped.
|
| This function clears the prescaler for timer_2.
| Parameters
|            [in] void.
| Return
|     void
|
void TIMER_timer2Stop(void);

| Start the timer by setting the desired prescaler.
|
| This function sets the prescaler for timer_2.
| Parameters
|            [in] u16_a_prescaler value to set the desired prescaler.
| Return
|     An EN_TIMER_ERROR_T value indicating the success or failure of
|            the operation (TIMER_OK if the operation succeeded, TIMER_ERROR
|            otherwise)
|
EN_TIMER_ERROR_T TIMER_timer2Start(u16 u16_a_prescaler);
```

```
| Creates a delay using timer_2 in overflow mode
|
| This function Creates the desired delay on timer_2 normal mode.
| Parameters
|           [in] u16_a_interval value to set the desired delay.
| Return
|     An EN_TIMER_ERROR_T value indicating the success or failure of
|           the operation
|            (TIMER_OK if the operation succeeded, TIMER_ERROR otherwise)
|
EN_TIMER_ERROR_T TIMER_timer2Delay(u16 u16_a_interval);



| Set callback function for timer overflow interrupt
|
| Parameters
|           void_a_pfOvfInterruptAction Pointer to the function to be
|                                       called on timer overflow interrupt
| Return
|     EN_TIMER_ERROR_T Returns TIMER_OK if callback function is set
|                      successfully, else returns TIMER_ERROR
|
EN_TIMER_ERROR_T TIMER_ovfSetCallback(void
(*void_a_pfOvfInterruptAction)(void));



| Interrupt Service Routine for Timer Overflow.
|      This function is executed when Timer2 Overflows.
|      It increments u16_g_overflow2Ticks counter and checks whether
|      u16_g_overflow2Numbers is greater than u16_g_overflow2Ticks.
|      If true, it resets u16_g_overflow2Ticks and stops Timer2.
|      It then checks whether void_g_pfOvfInterruptAction is not null.
|      If true, it calls the function pointed to by
|     void_g_pfOvfInterruptAction.
|
| Return
|     void
|
ISR(TIMER_ovfVect);
```

2.3.3. HAL APIs

2.3.3.1. LED APIs

```
| Initializes a single LED pin as output
|
| Parameters
|           [in] en_a_ledPort The port where the LED is located
|                             (PORT_A, PORT_B, PORT_C or PORT_D)
|           [in] u8_a_ledPin The pin number of the LED
|                            (DIO_U8_PIN_0 to DIO_U8_PIN_7)
| Return
|      EN_LED_ERROR_t Returns LED_OK if the LED was initialized
|                  successfully, LED_ERROR otherwise.
|
EN_LED_ERROR_t LED_init(EN_DIO_PORT_T en_a_ledPort, u8 u8_a_ledPin);



| Turn on an LED connected to a specific pin on a specific port.
|
| Parameters
|           [in] en_a_ledPort The port where the LED is connected.
|                              (PORT_A, PORT_B, PORT_C, or PORT_D)
|           [in] u8_a_ledPin The pin number where the LED is connected.
|                             (DIO_U8_PIN_0 to DIO_U8_PIN_7)
| Return
|      The status of the LED operation, either LED_OK or LED_ERROR.
|
EN_LED_ERROR_t LED_on(EN_DIO_PORT_T en_a_ledPort, u8 u8_a_ledPin);



| Turns off an LED on a specific port and pin.
| Parameters
|      [in]    en_a_ledPort    The port of the LED to turn off
|                                (PORT_A, PORT_B, PORT_C, or PORT_D)
|      [in]    u8_a_ledPin       The pin number of the LED to turn off
|                                (DIO_U8_PIN_0 to DIO_U8_PIN_7)
| Returns
|      EN_LED_ERROR_t LED_OK if successful,
|                    or LED_ERROR if there was an error.
|
EN_LED_ERROR_t LED_off(EN_DIO_PORT_T en_a_ledPort, u8 u8_a_ledPin);
```

```
| Initializes a group of LEDs connected to a specific port and pins
|  with a specified mask as output.
|
| Parameters
|      [in]    en_a_ledPort    The port to which the LEDs are connected.
|      [in]    u8_a_mask       The mask to set the direction of the
|                                specified pins.
| Returns
|      EN_LED_ERROR_t Returns LED_OK if the operation is successful,
|                     and LED_ERROR if the operation fails.
|
EN_LED_ERROR_t LED_arrayInit(EN_DIO_PORT_T en_a_ledPort, u8 u8_a_mask);



| Turns on the specified LED pins by setting the corresponding bits in
|  the specified LED port to HIGH.
|
| Parameters
|             [in] en_a_ledPort The LED port to turn on the LED pins from
|                          (PORT_A, PORT_B, PORT_C or PORT_D).
|             [in] u8_a_mask The bit mask specifying which LED pins to turn
|                          on. (DIO_NO_MASK, DIO_MASK_BITS_n..)
| Return
|      EN_LED_ERROR_t Returns LED_OK if the LED pins were successfully
|                   turned on, or LED_ERROR otherwise.
|
EN_LED_ERROR_t LED_arrayOn(EN_DIO_PORT_T en_a_ledPort, u8 u8_a_mask);



| Turns off the specified LED pins by setting the corresponding bits in
|  the specified LED port to LOW.
|
| Parameters
|             [in] en_a_ledPort The LED port to turn off the LED pins from
|                              (PORT_A, PORT_B, PORT_C or PORT_D).
|             [in] u8_a_mask    The bit mask specifying which LED pins to turn off
|                               (DIO_NO_MASK, DIO_MASK_BITS_n..)
| Return
|      EN_LED_ERROR_t Returns LED_OK if the LED pins were successfully
|                   turned off, or LED_ERROR otherwise
|
EN_LED_ERROR_t LED_arrayOff(EN_DIO_PORT_T en_a_ledPort, u8 u8_a_mask);
```

2.3.3.2. BTN APIs

```
| Initialize a GPIO pin as an input pin
|
| This function initializes a specified GPIO pin as an input pin using
|  the DIO_init() function.
|
| Parameters
|            [in]u8_a_pinNumber The pin number to be initialized (0-7).
|            [in]en_a_portNumber The port number to which the pin belongs
|                               (PORT_A, ..).
| Return
|     STD_OK if the pin initialization was successful, STD_NOK
|            otherwise.
|
u8 BTN_init(u8 u8_a_pinNumber, EN_DIO_PORT_T en_a_portNumber);



| This function reads the current state of a specified button by calling
|  the DIO_read() function.
|
| Parameters
|             [in]u8_a_btnId The ID of the button to read (BTN_U8_1 to BTN_U8_8).
|            [out]u8ptr_a_returnedBtnState A pointer to an 8-bit unsigned
|                                          integer where the button state will be stored.
| Return
|     STD_OK if the button state was read successfully, STD_NOK otherwise.
|
u8 BTN_getBtnState(u8 u8_a_btnId, u8 *u8ptr_a_returnedBtnState);
```

2.3.3.3. DCM APIs

```
| Initialize the DC Motors by initializing their pins.
|
| Parameters
|           u8_a_shutdownFlag Pointer to the Shutdown flag variable that
|                             acts as a main kill switch.
| Return
|      EN_DCM_ERROR_T Returns DCM_OK if initialization is successful, or
|                    DCM_ERROR if initialization failed.
|
EN_DCM_ERROR_T DCM_motorInit(u8 ** u8_a_shutdownFlag);



| Rotates the DC motor.
|
| This function rotates the DC motor by changing its direction to right,
| setting the duty cycle of the PWM signal to a predefined value,
| and then changing the direction of the motor again to the right.
|
| Return
|      EN_DCM_ERROR_T DCM_OK if the operation is successful, DCM_ERROR
|                    otherwise.
|
EN_DCM_ERROR_T DCM_rotateDCM(void);



| Changes the direction of the motor rotation for the specified motor.
|
| Parameters
|           en_a_motorNum The motor number whose direction needs to be changed.
| Return
|      EN_DCM_ERROR_T DCM_OK if the operation is successful, DCM_ERROR otherwise.
|
EN_DCM_ERROR_T DCM_changeDCMDirection(EN_DCM_MOTORSIDE en_a_motorNum);
```

```
| Sets the duty cycle of the PWM for the motor.
|
| This function sets the duty cycle of the PWM for the motor. The duty
|  cycle value
| provided should be between 0 and 100, where 0 indicates a duty cycle of
|  0% and 100
| indicates a duty cycle of 100%.
|
| Parameters
|            u8_a_dutyCycleValue The duty cycle value for the motor.
| Return
|      EN_DCM_ERROR_T The error status of the function.
|      - DCM_OK: The function executed successfully.
|      - DCM_ERROR: The duty cycle value provided was out of range.
|
EN_DCM_ERROR_T DCM_setDutyCycleOfPWM(u8 u8_a_dutyCycleValue);

| Stops the DC motors by setting the PWM output pins to low and resetting
|  the stop flag.
|
void DCM_vdStopDCM(void);

| Updates the stop flag.
|
| This function is called by the timer overflow callback function to
|  update the stop flag.
| It sets the `en_g_stopFlag` variable to TRUE, which is used by other
|  functions to stop the
| motor movement.
|
void DCM_updateStopFlag(void);
```

2.3.4. APP APIs

```
| Initializes the application by initializing MCAL and HAL.
| This function initializes the General Interrupt Enable (GIE), sets up
|  callback functions
|
| for interrupt service routines, initializes the timers and buttons,
|  initializes an LED array,
| initializes the DC motor, and sets the application mode to "Car Stop".
|
| Return
|     None
|
void APP_initialization(void);



| This function starts the car program and keeps it running indefinitely.
| The function uses a while loop to continuously check for the required
|  app mode.
| The app mode is checked using a switch statement, which contains
|  various cases
| that correspond to the different modes of operation for the car
|  program. Each
| case contains a series of steps to be executed to perform the desired
|  action
| for that mode.
|
| Return
|     void
|
void APP_startProgram(void);



| ISR Callback function for starting the car
void APP_startCar(void);



| ISR Callback function for stopping the car immediately
void APP_stopCar(void);
```

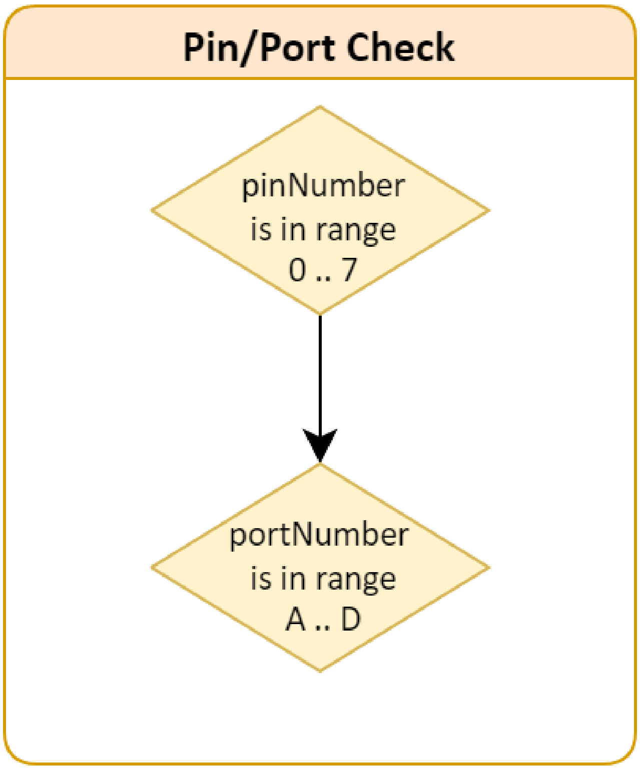
3. Low Level Design

3.1. MCAL Layer

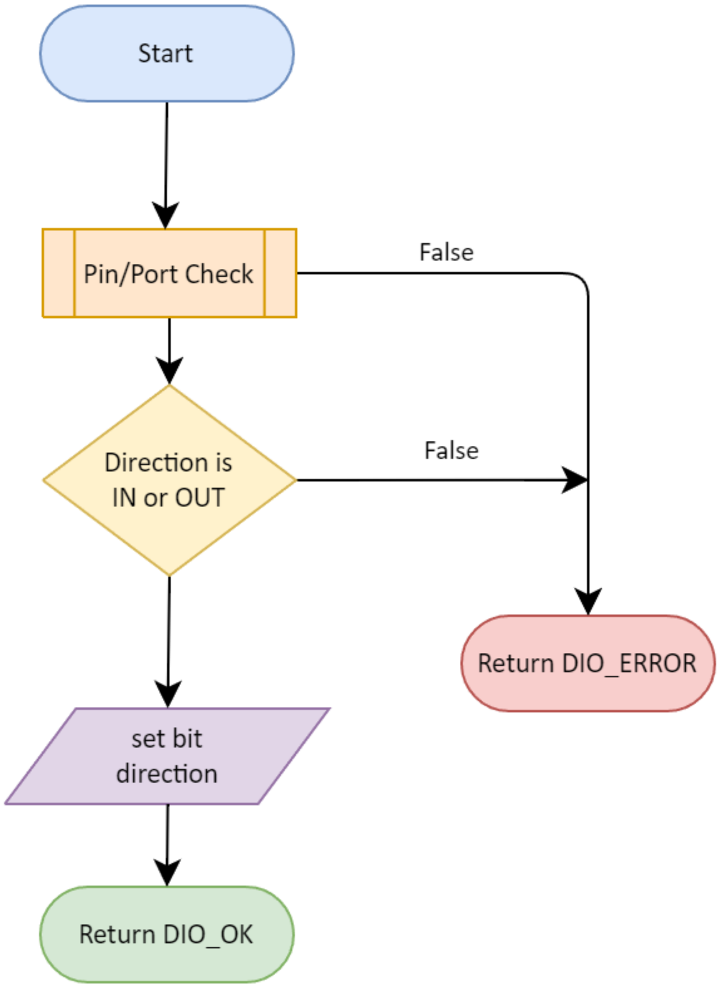
3.1.1. DIO Module

3.1.1.a. sub process

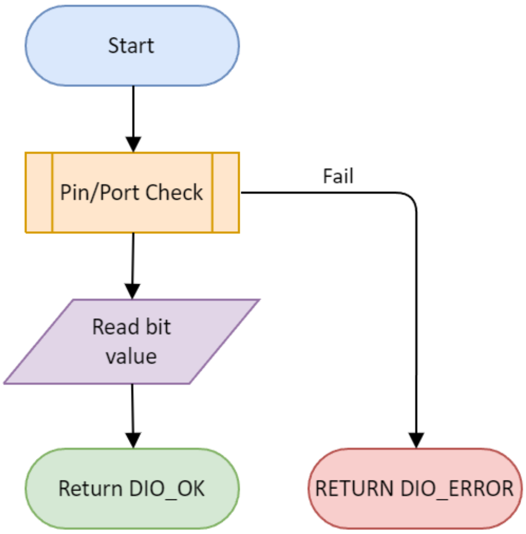
The following Pin/Port check subprocess is used in some of the DIO APIs flowcharts



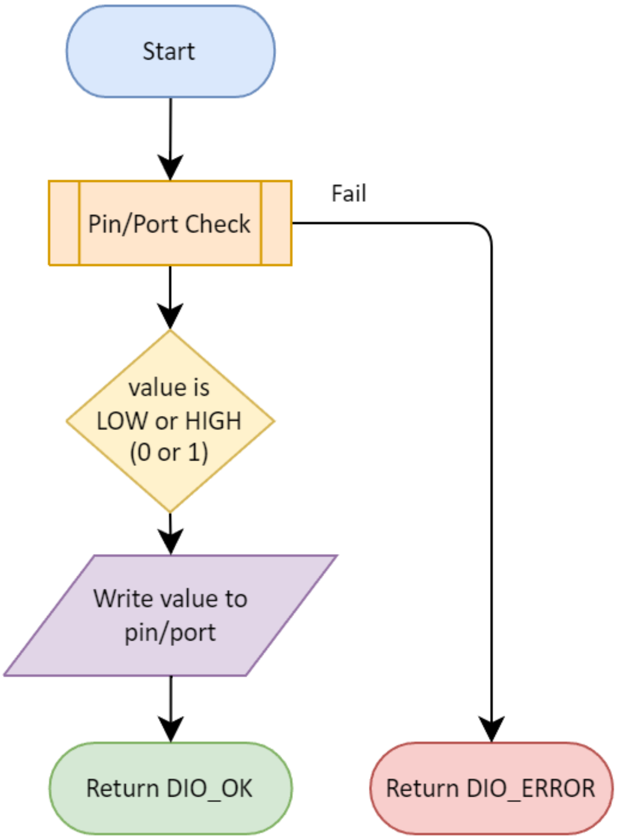
3.1.1.1. DIO_init



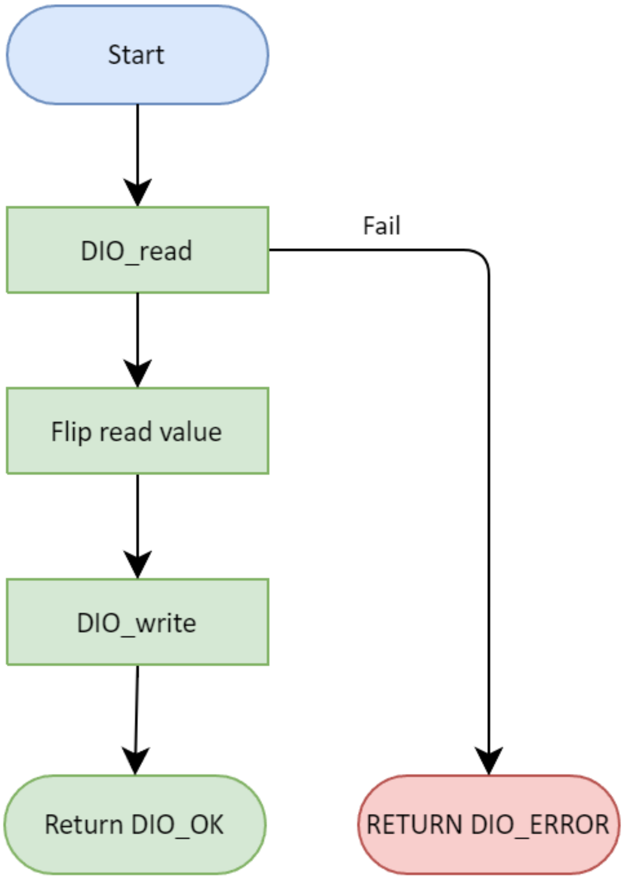
3.1.1.2. DIO_read



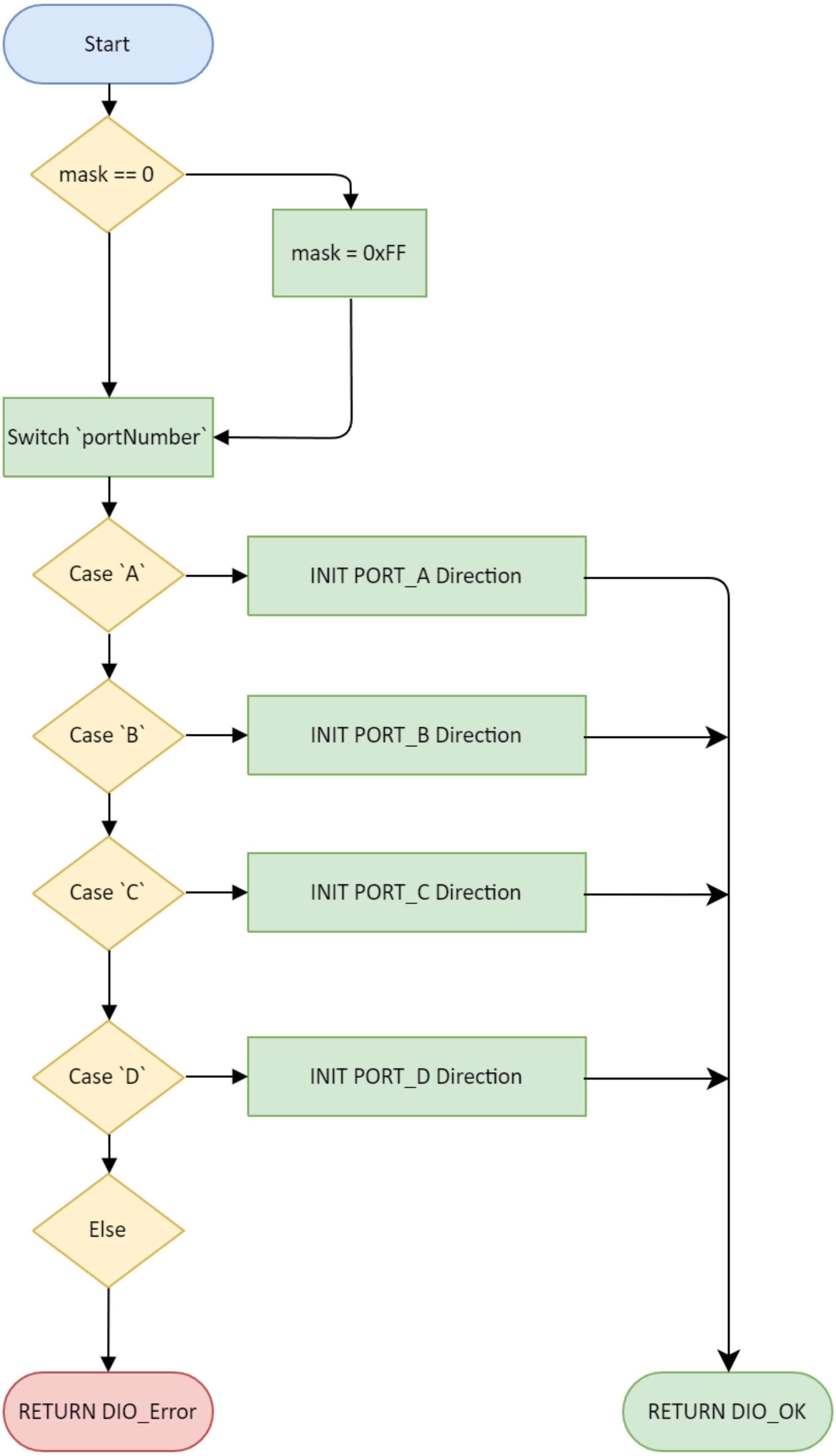
3.1.1.3. DIO_write



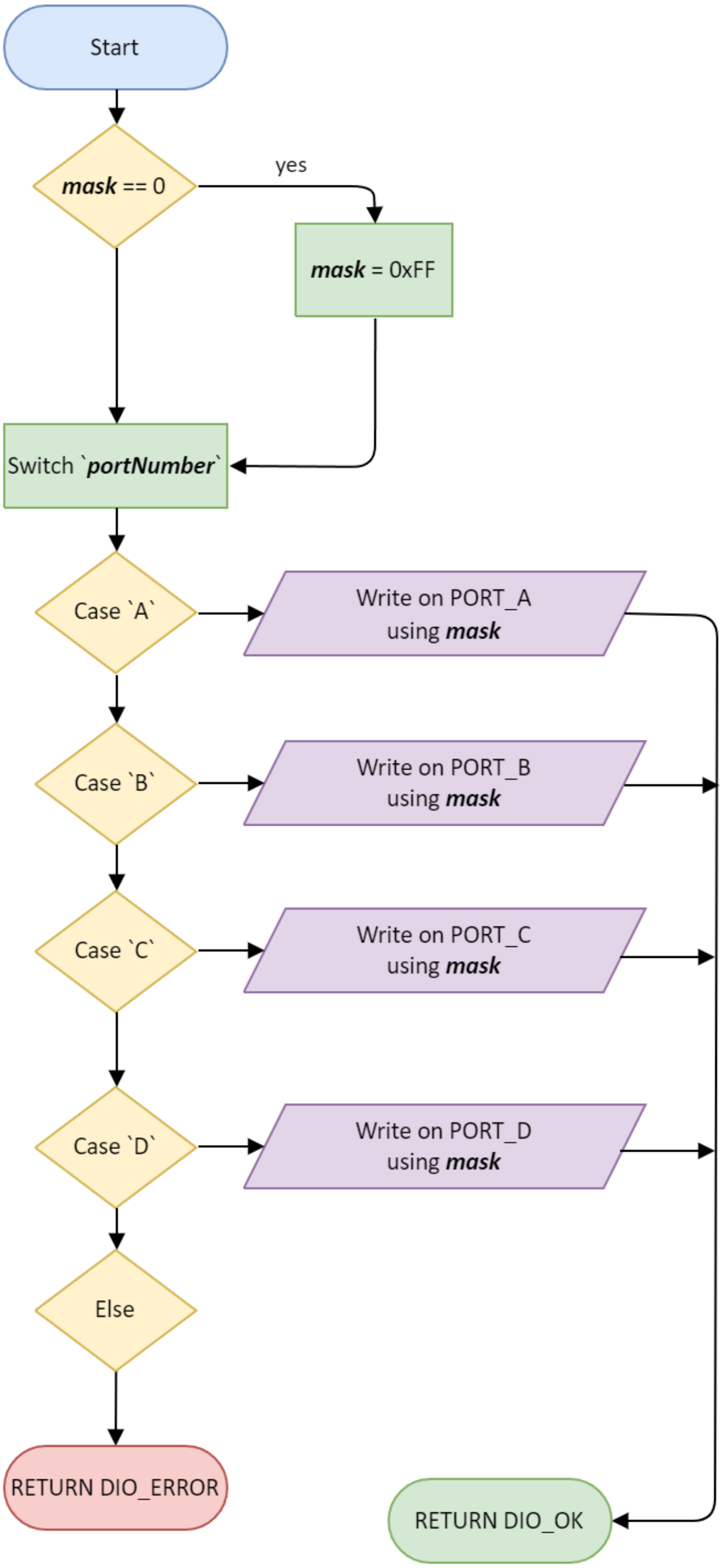
3.1.1.4. DIO_toggle



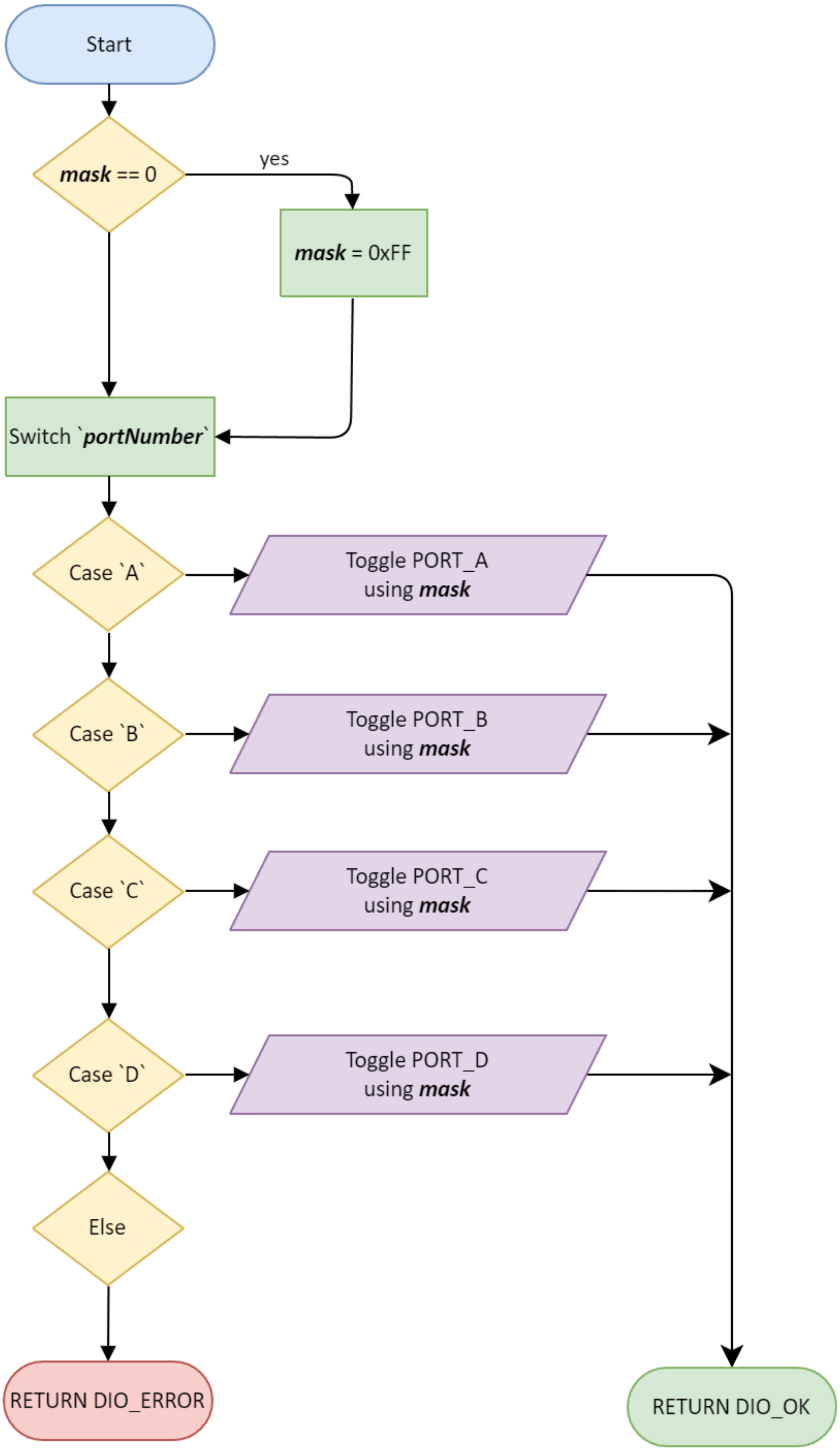
3.1.1.5. DIO_portInit



3.1.1.6. DIO_portWrite

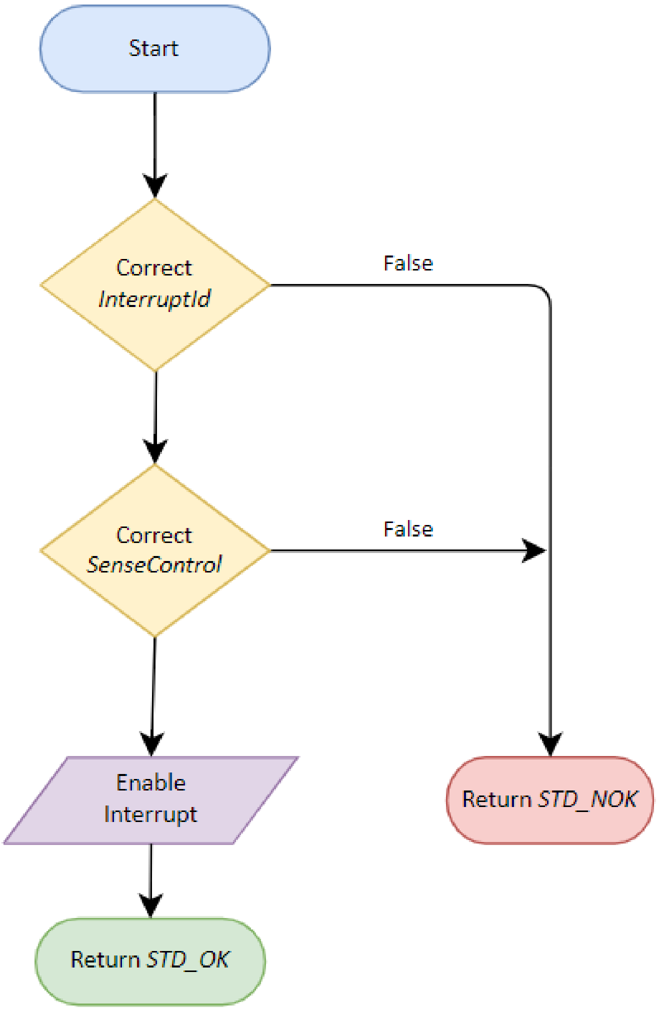


3.1.1.7. DIO_portToggle

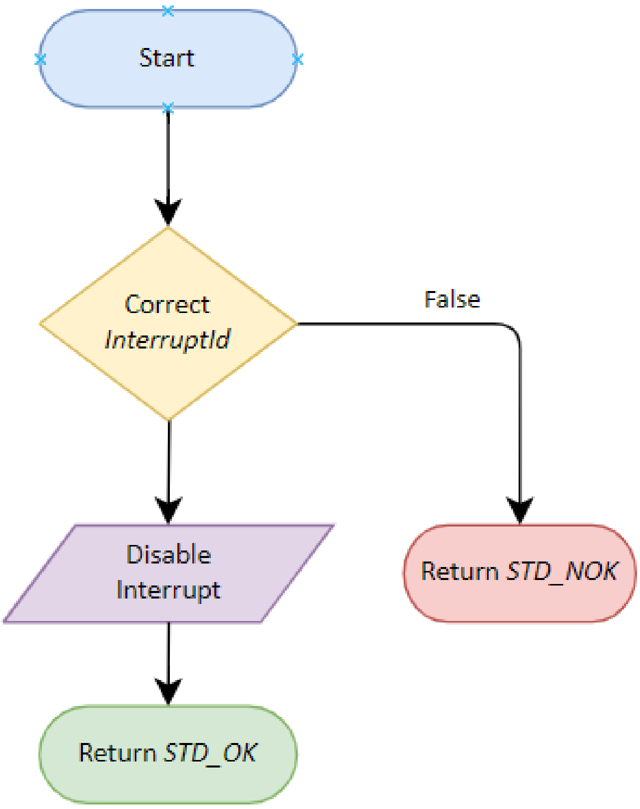


3.1.2. EXI Module

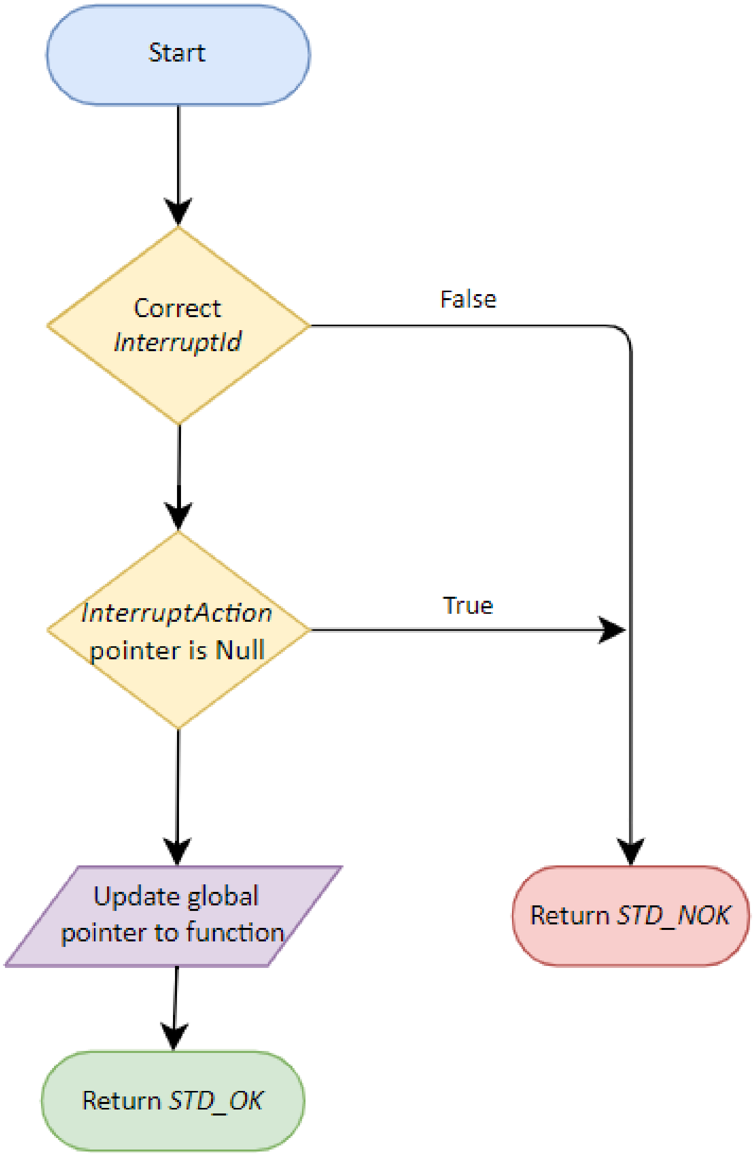
3.1.2.1. EXI_enablePIE



3.1.2.2. EXI_disablePIE

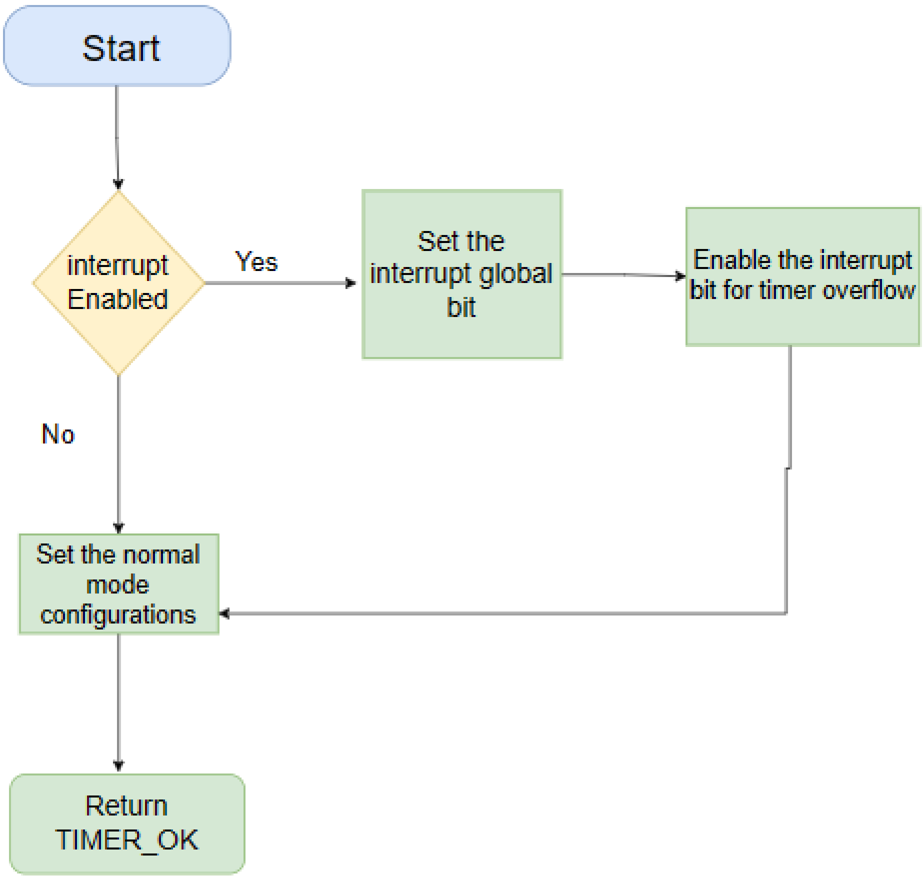


3.1.2.3. EXI_intSetCallback

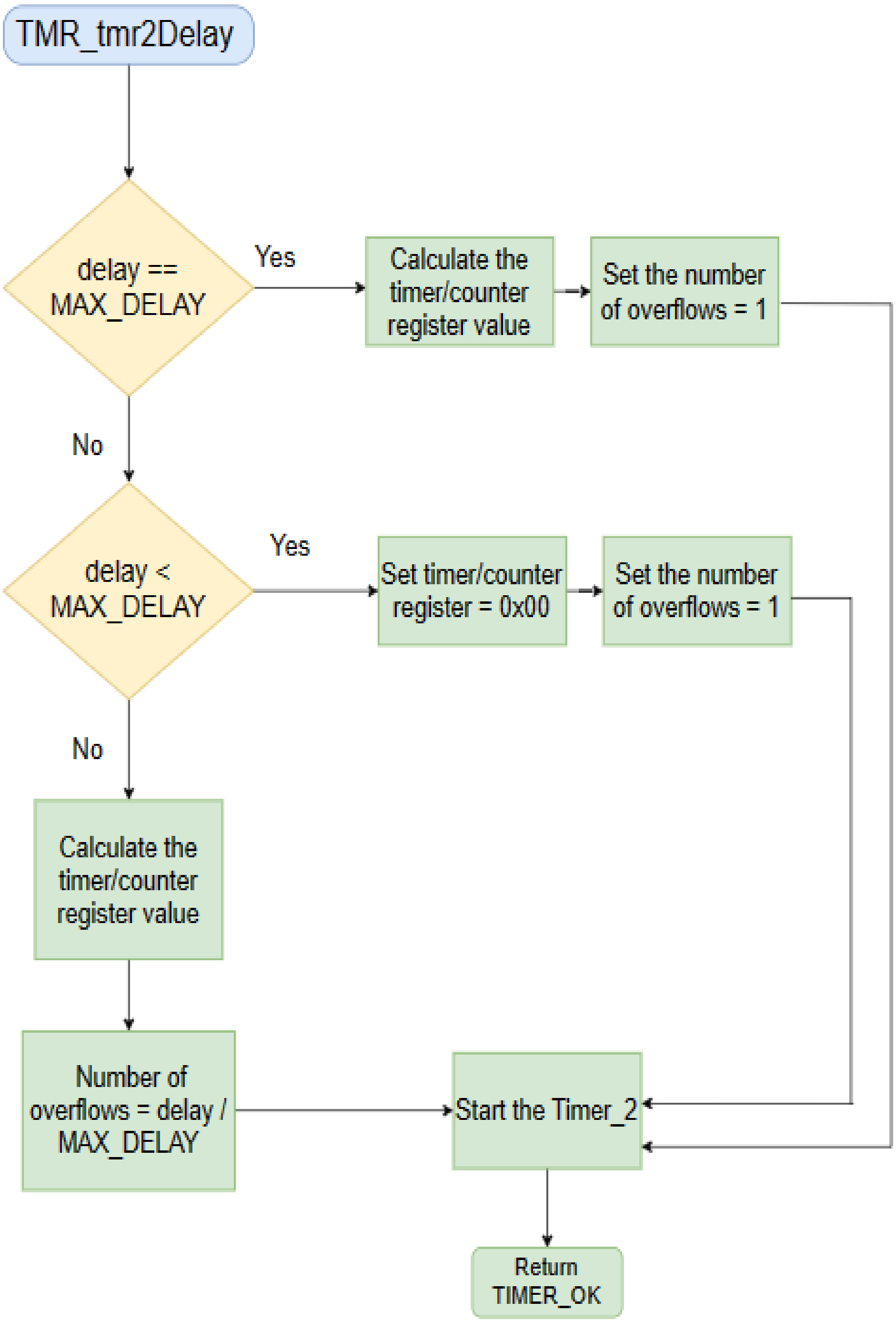


3.1.3. Timer Module

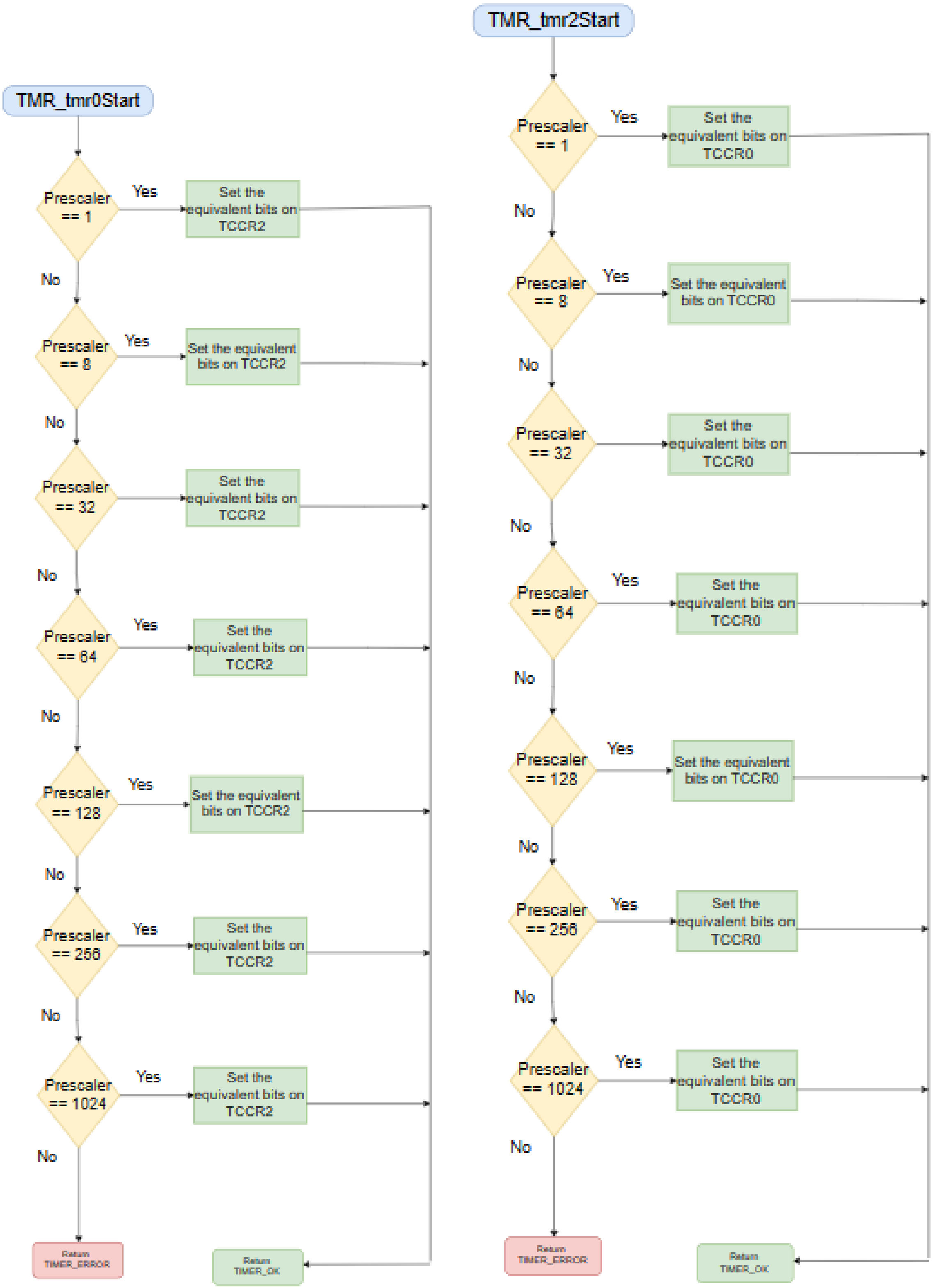
3.1.3.1. TMR_tmr0NormalModeInit / TMR_tmr2NormalModeInit



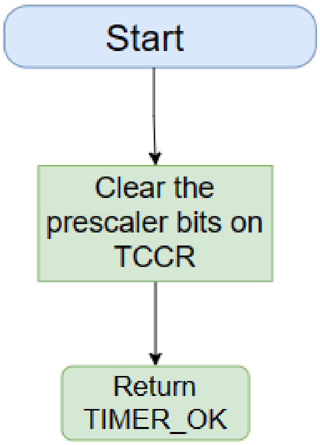
3.1.3.2. TMR_tmr0Delay / TMR_tmr2Delay



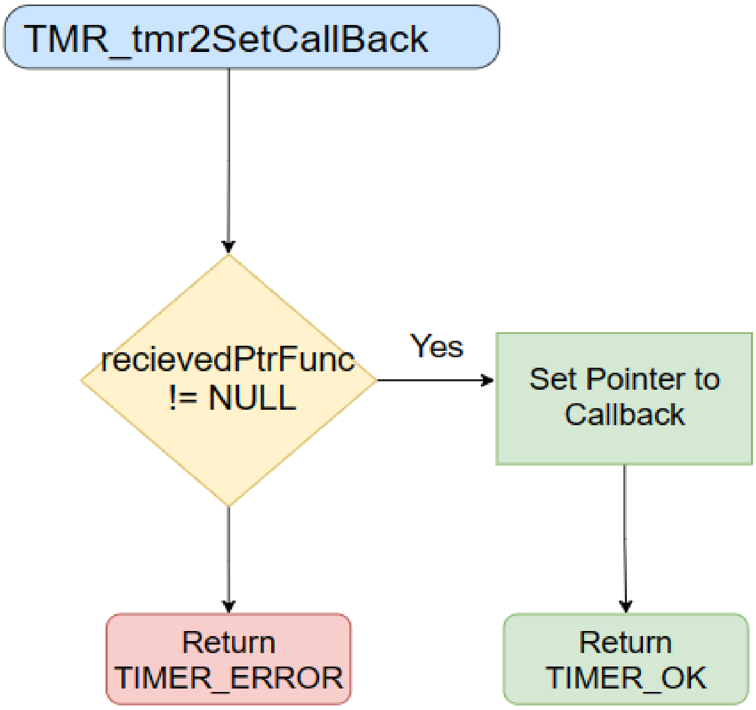
3.1.3.3. TMR_tmr0Start / TMR_tmr2Start



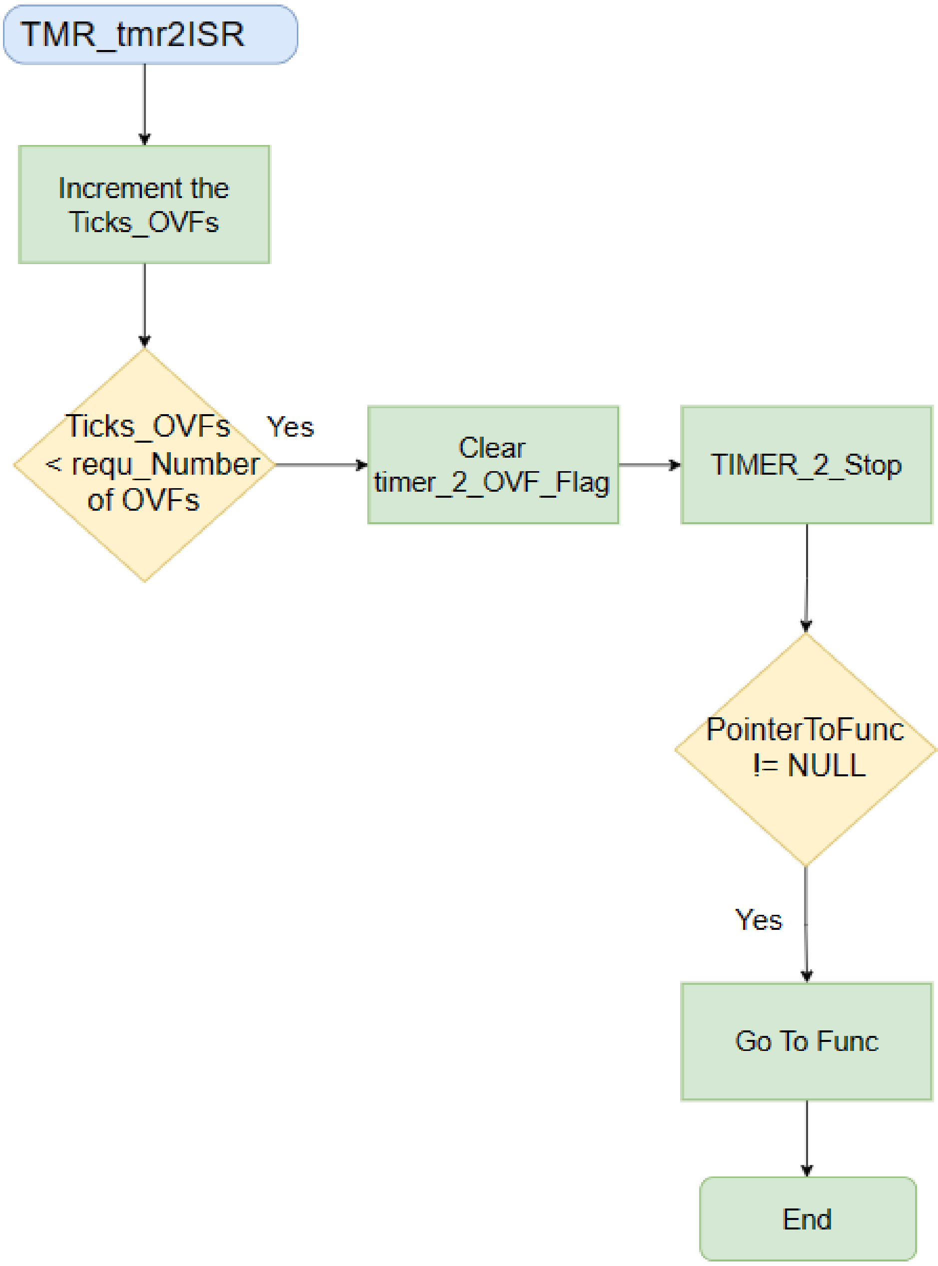
3.1.3.4. TMR_tmr0Stop / TMR_tmr2Stop



3.1.3.5. TMR_ovfSetCallback



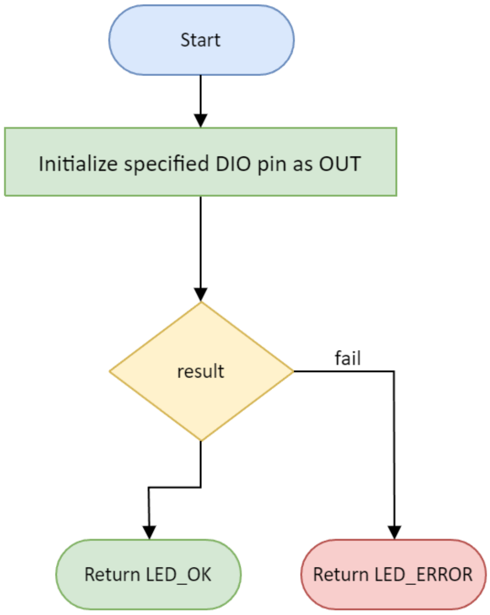
3.1.3.6. TMR_ovfVect



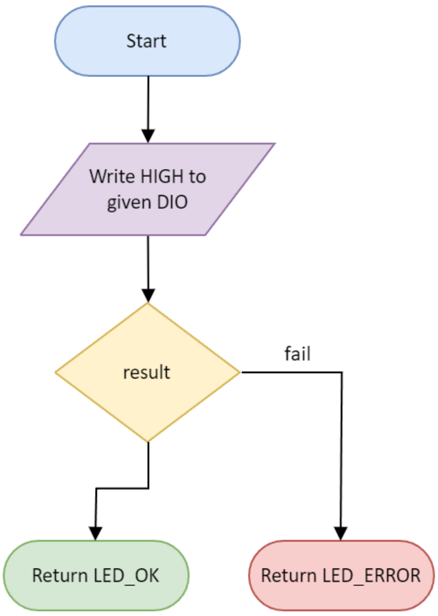
3.2. HAL Layer

3.2.1. LED Module

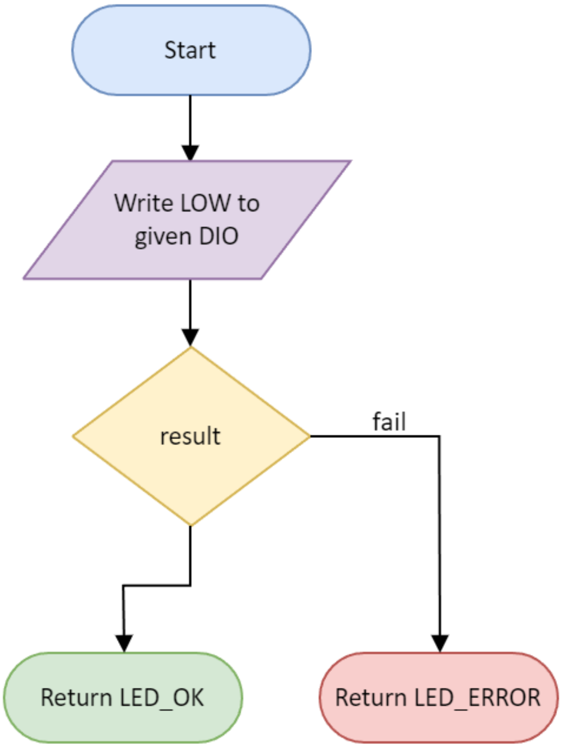
3.2.1.1. LED_init



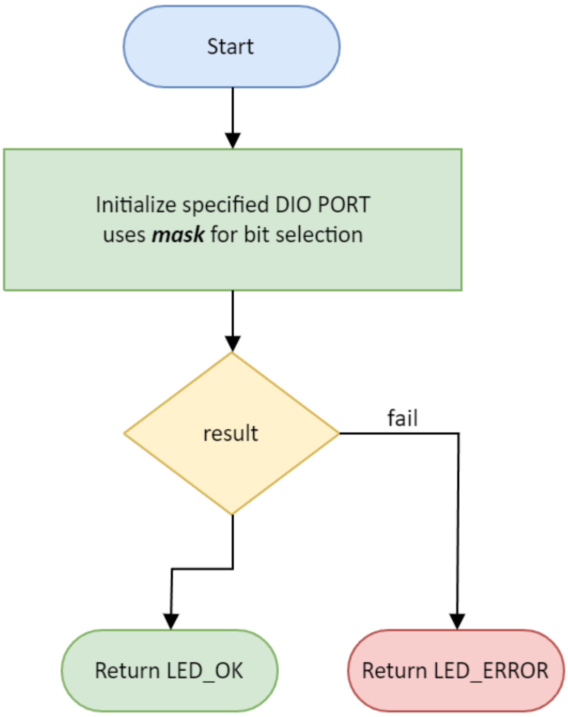
3.2.1.2. LED_on



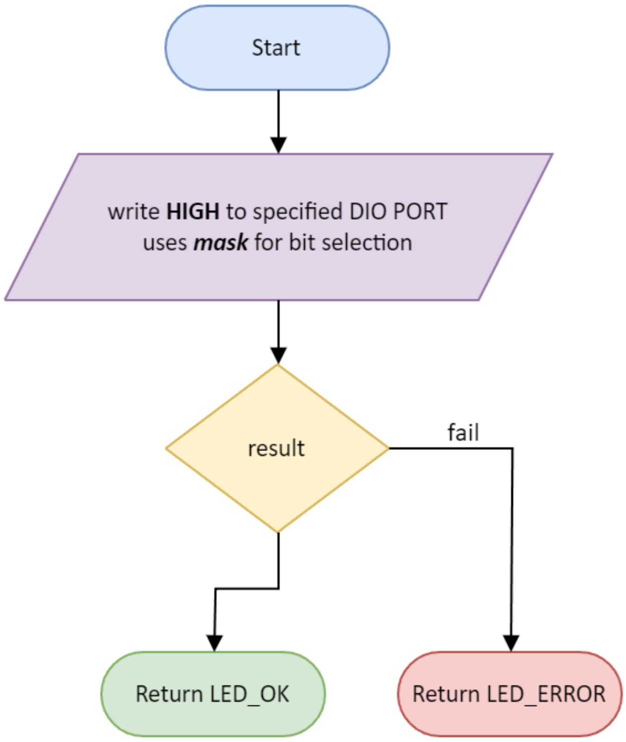
3.2.1.3. LED_off



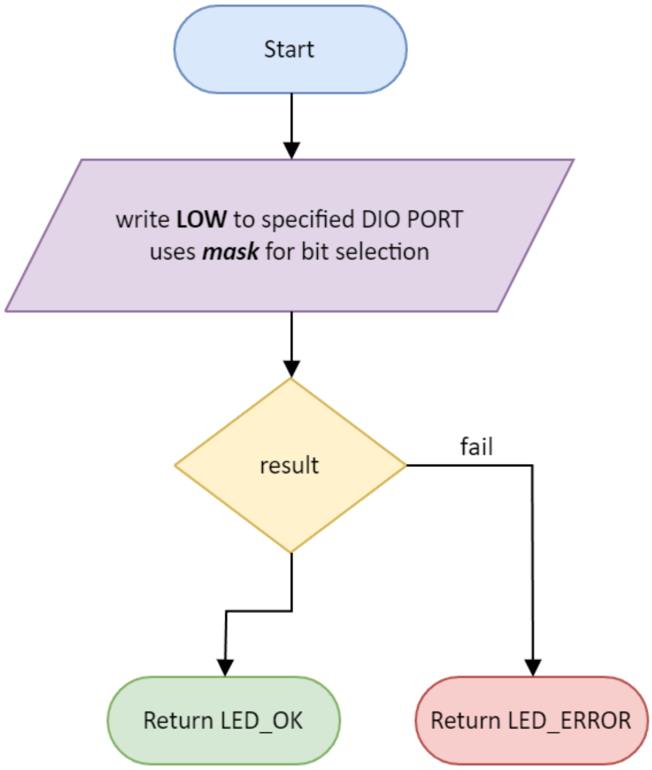
3.2.1.4. LED_arrayInit



3.2.1.5. LED_arrayOn

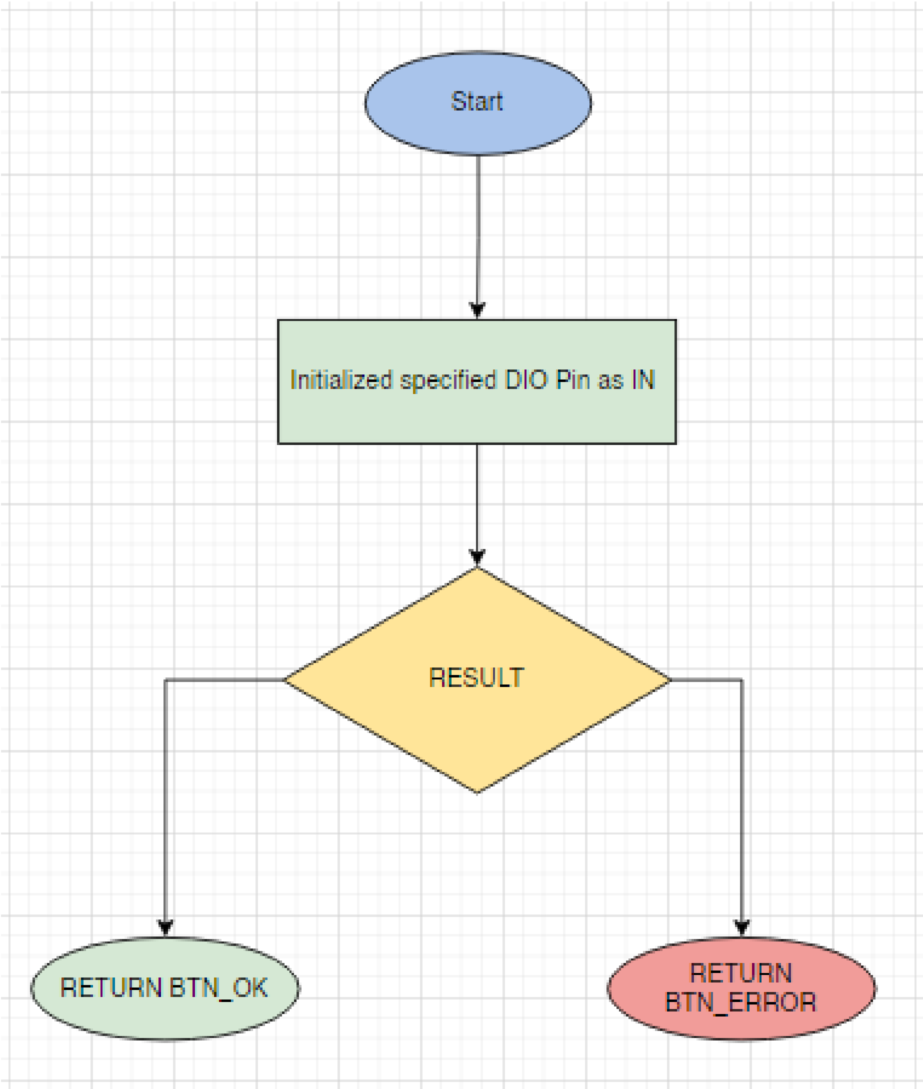


3.2.1.6. LED_arrayOff

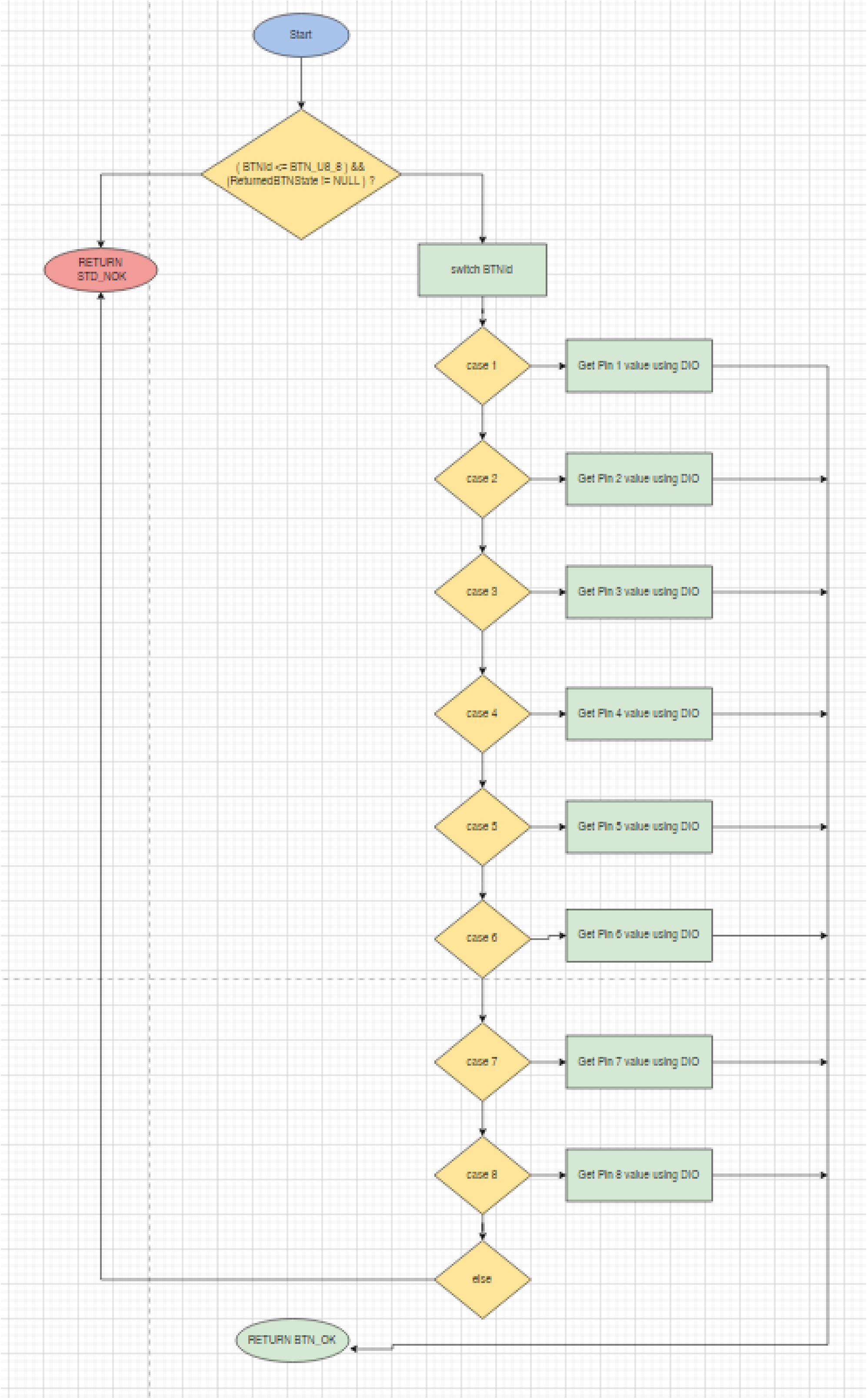


3.2.2. BTN Module

3.2.2.1. BTN_init

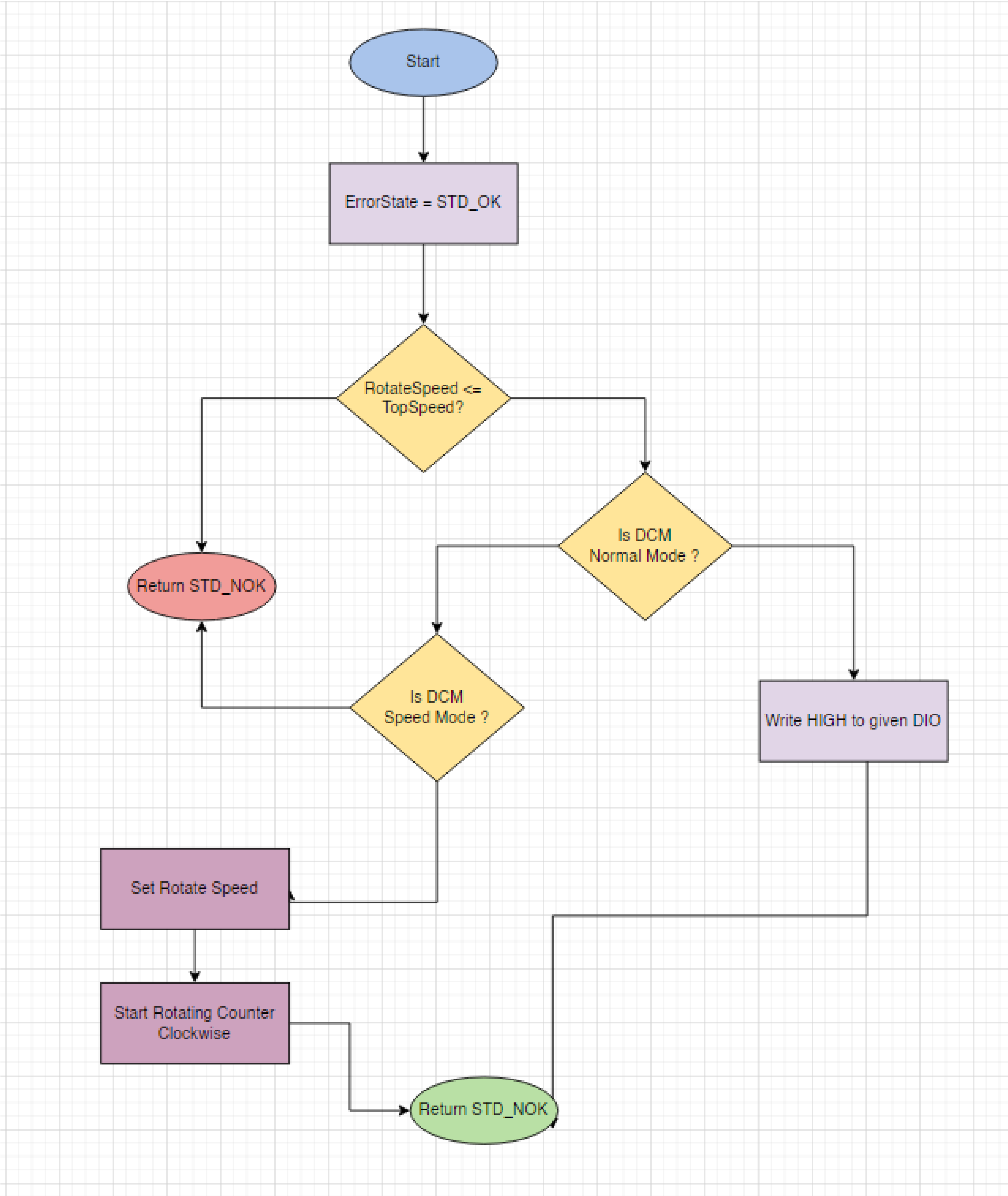


3.2.3.2. BTN_getBTNState

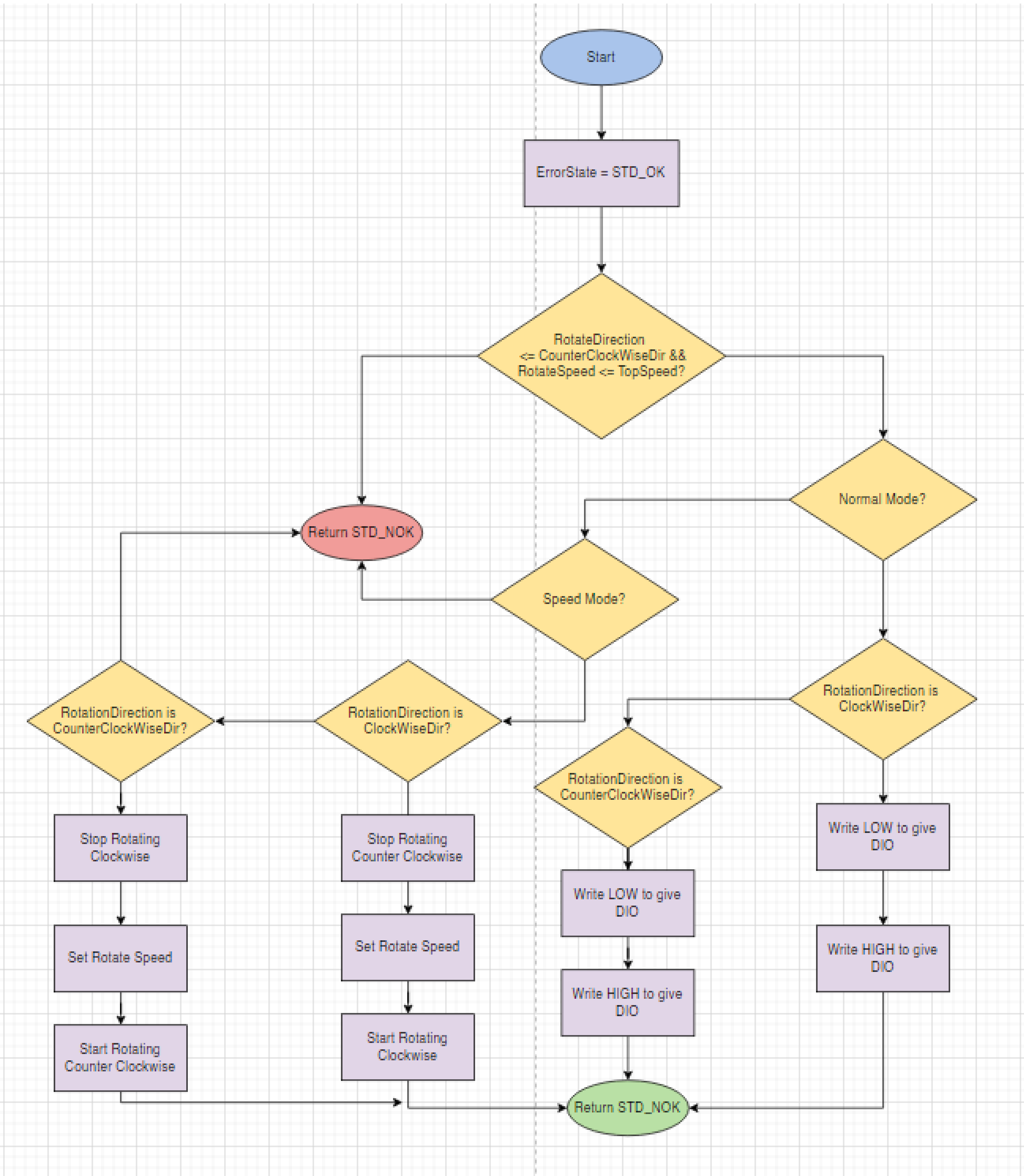


3.2.3. DCM Module

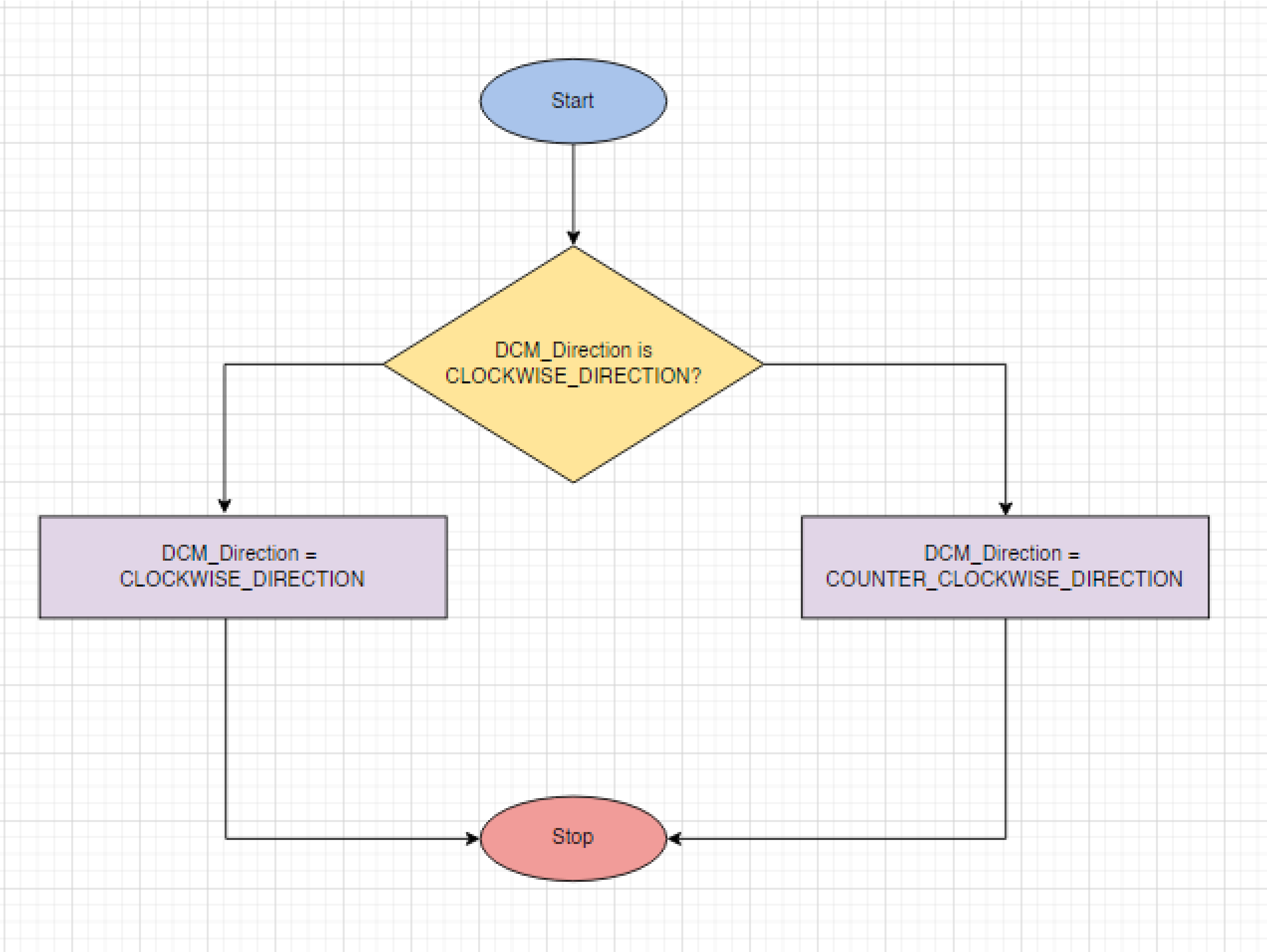
3.2.3.1. DCM_rotateDCMInOneDirection



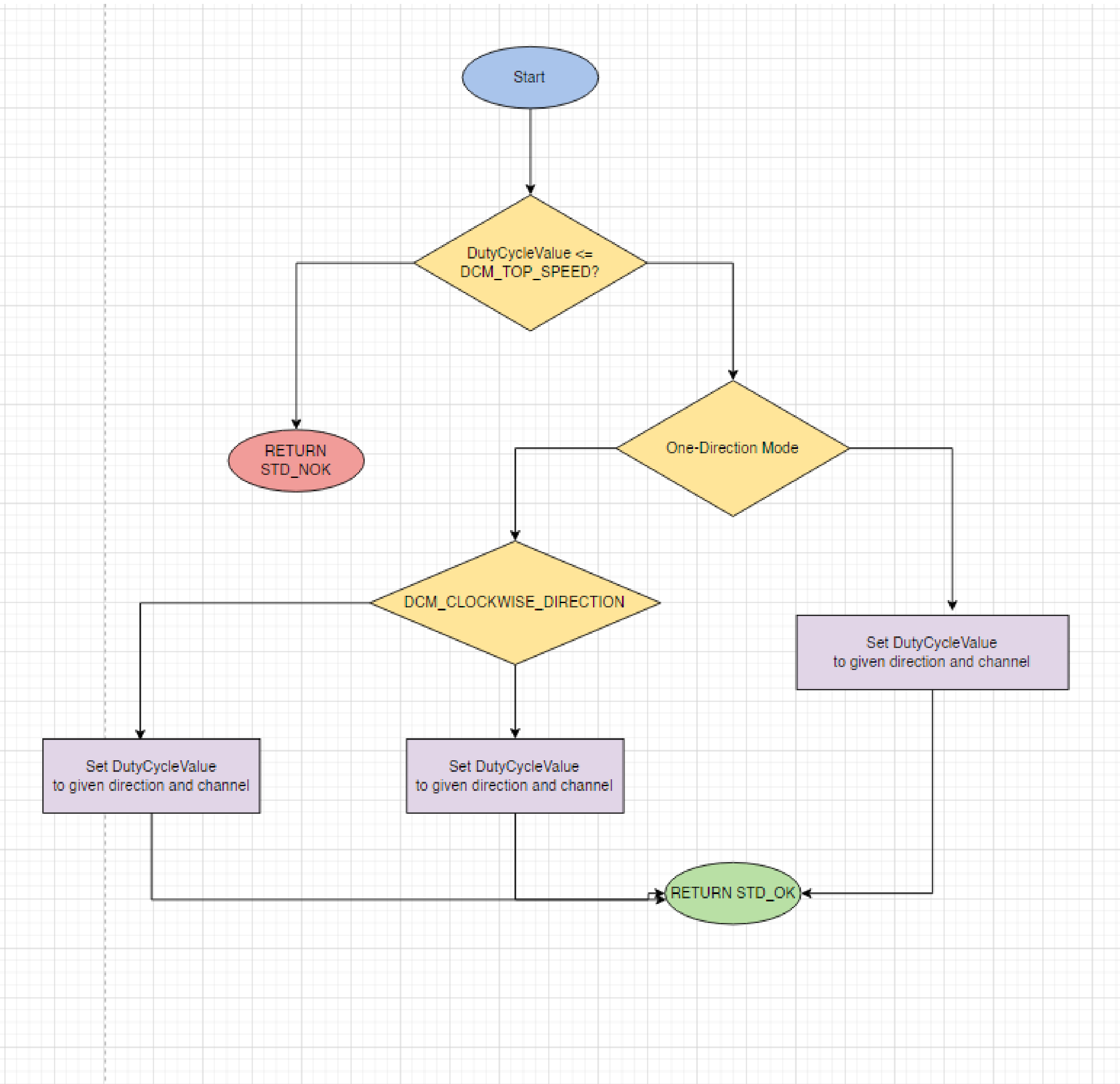
3.2.3.2. DCM_rotateDCMInTwoDirections



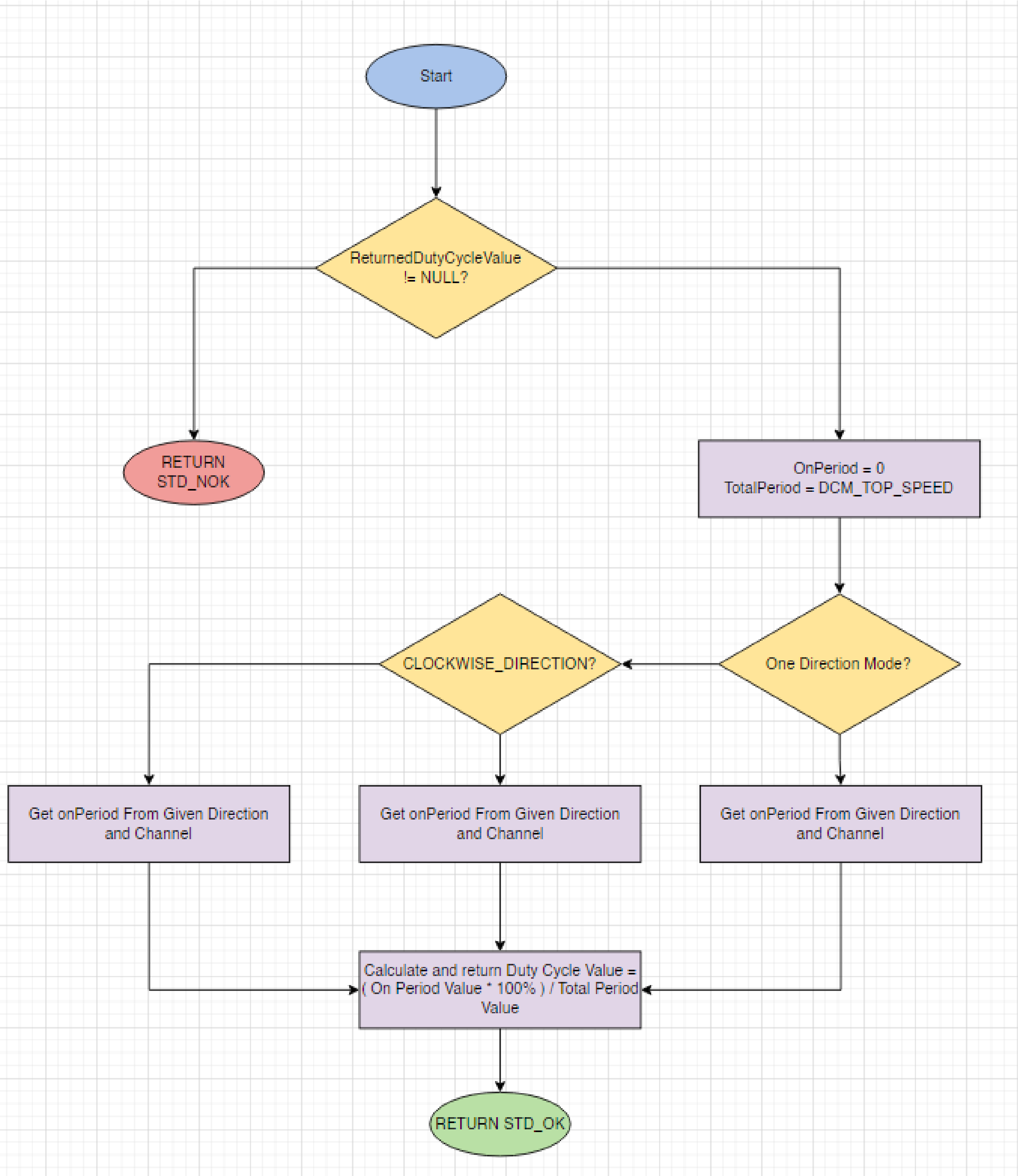
3.2.3.3. DCM_changeDCMDirection



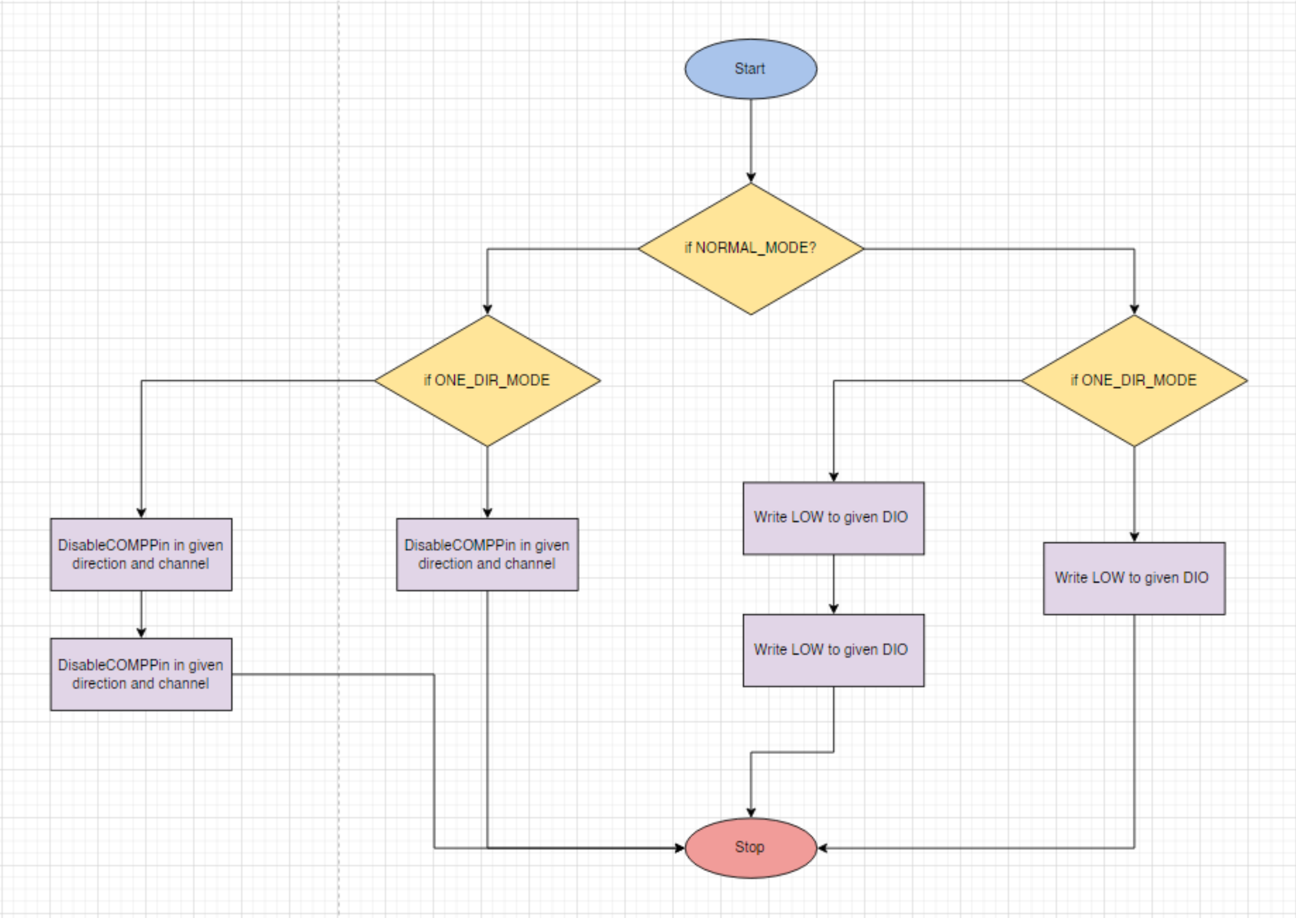
3.2.3.4. DCM_setDutyCycleOfPWM



3.2.3.5. DCM_getDutyCycleOfPWM

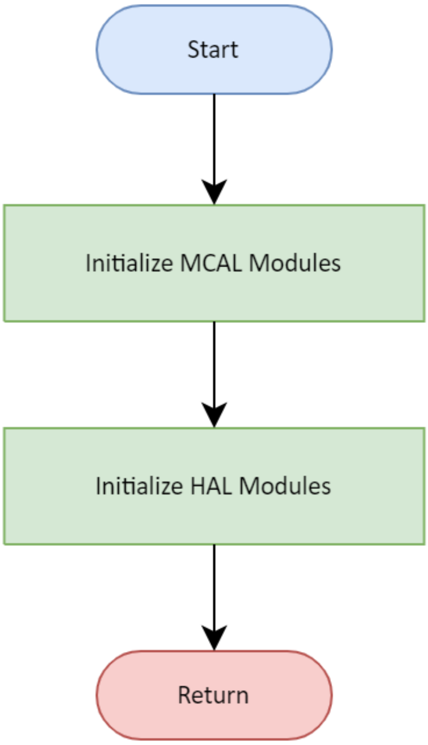


3.2.3.6. DCM_stopDCM

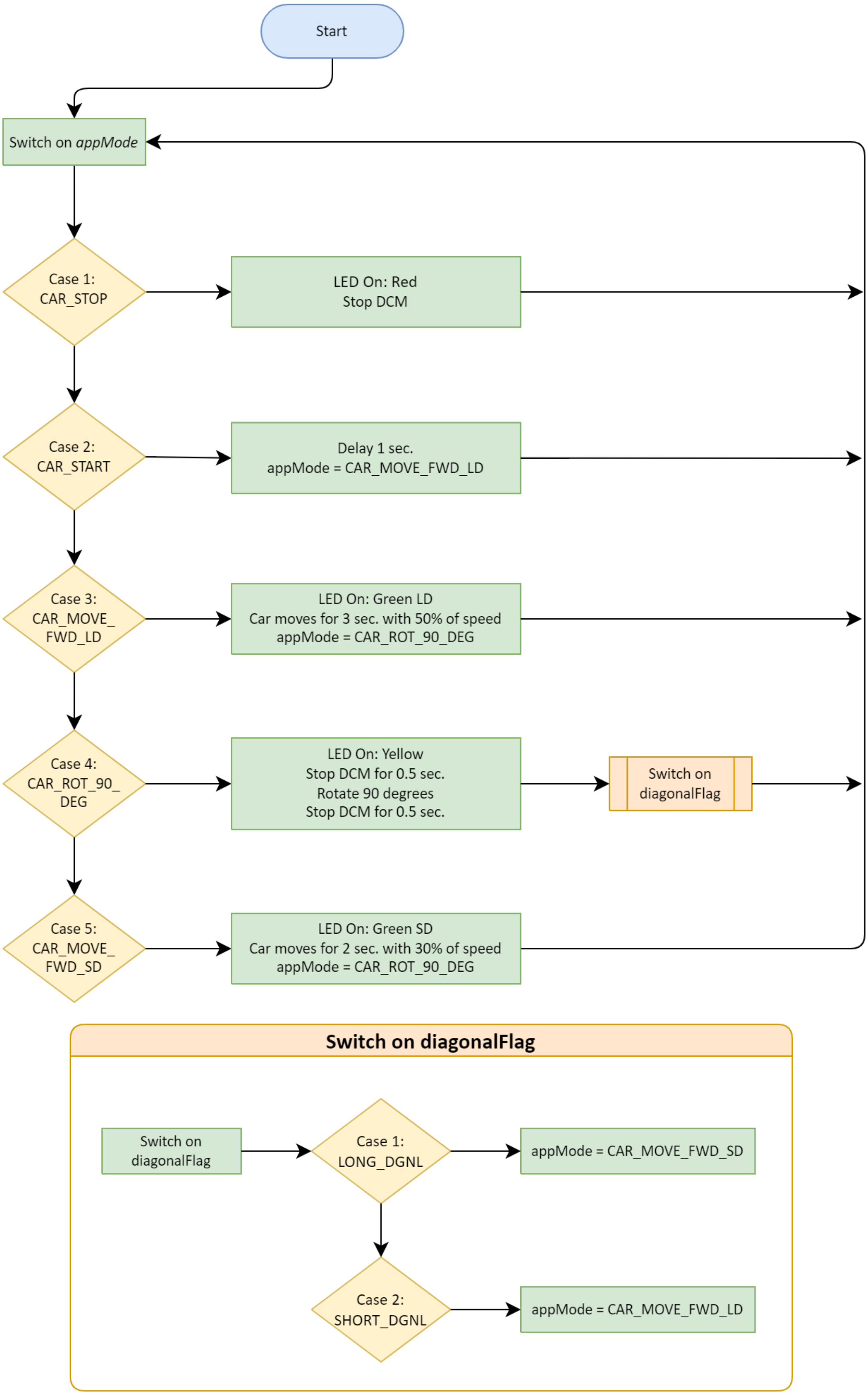


3.3. APP Layer

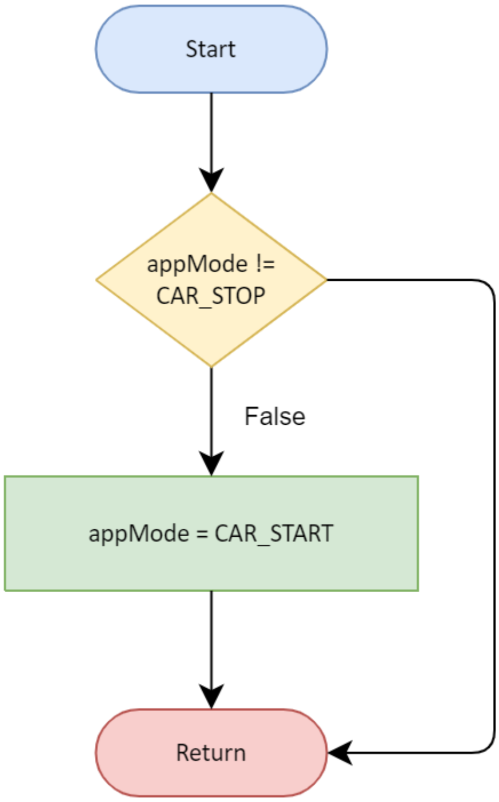
3.3.1. APP_initialization



3.3.2. APP_startProgram



3.3.3. APP_startCar



3.3.4. APP_stopCar