ATM MACHINE

AUTHOR

TEAM 1 - HACKER KERMIT

MEMBERS

HOSSAM ELWAHSH

ABDELRHMAN WALAA

MAHMOUD MOWAFEY

Simple ATM Machine Project

1. Project Introduction

This project involves developing software to simulate a simple ATM machine transaction.

1.1. System Requirements

1.1.1. Hardware Requirements

1.1.2.1. ATM ECU

- ATM MCU
- 16x2 LCD
- 3x3 Keypad
- Buzzer
- Enter/Set Button

1.1.2.2. CARD ECU

- CARD MCU
- EEPROM
- Serial Terminal (Putty on PC)

1.1.2. Software Requirements

1.1.2.1. ATM MCU

This application will handle transaction main flows:

1. **After Reset**
   - 1.1. Welcome message is displayed for **1s** ***"Welcome to ATM"***
   - 1.2. ***"Insert a Card"*** message is displayed in the first line
   - 1.3. No action can be taken further and ll other input devices are blocked until a trigger signal came from the CARD ECU
2. **After a trigger signal is received from the CARD ECU**
   - 2.1. ***"Enter Your PIN"*** message is displayed in the first line
   - 2.2. Waiting for the input from the keypad and type it in ***'* * * *'*** format in the second line
   - 2.3. PIN is only four numeric characters
   - 2.4. Pressing Enter/Zero button for **2s** will initiate a communication between the ATM ECU and CARD ECU to validate if the PIN is correct or not
   - 2.5. If PIN is not correct, repeat for further 2 trial (total 3 trials), and then if it is still wrong, sound the alarm and lock every input in the ATM, this blocking can be revealed by a hard reset

2.6. If PIN is correct, then ***"Enter Amount"*** message is displayed in the first line and wait for the amount to be entered from the keypad and appeared in the second line

2.7. Amount is float string with max **4** integer digits and **2** decimal digits ***"0000.00"***

2.8. You can enter ***'0'*** by pressing Enter/Zero for less than ***2 seconds***

2.9. After entering the amount to withdraw, several checks on the database are done to finalize the transaction

2.9.1. Check if there is an account attached to this card

2.9.2. Check if the card is blocked or not

2.9.3. Check if the amount is required exceeds daily limit or not

2.9.4. Check for available amount

2.10. If one of the checks failed, a declined message will appear accordingly

2.10.1. ***"This fraud card"*** - if the card PAN is not found - Alarm will be initiated

2.10.2. ***"This card is stolen"*** - if the card is blocked - Alarm will be initiated

2.10.3. ***"Maximum limit is exceeded"*** - if the required amount exceeds the maximum allowed limit

2.10.4. ***"Insufficient fund"*** - if the balance is lower than the required amount

2.11. If all checks are passed then ***"Approved Transaction"*** message is displayed for 1s and the remaining balance is displayed for ***1s "New Balance: 0000.00"***

2.12. After the checks are done and messages are displayed, display ***"Ejecting Card"*** message for ***1s***

2.13. Repeat from the after reset again

3. Database

3.1. The database will be hard coded array of structures that contains (PAN, Account State (blocked/running), and balance)

3.2. The maximum allowed limit will be hard coded **"5000.00"**

1.1.2.2.CARD MCU

This application has two modes of operation:

1. **Programming Mode**
    - 1.1. The CARD MCU will enter this mode after reset
    - 1.2. For the first time only the MCU will send the following message on the terminal
        - 1.2.1. ***"Please Enter Card PAN:"*** and wait for the PAN
        - 1.2.2. ***"Please Enter New PIN:"*** and wait for the PIN
        - 1.2.3. ***"Please Confirm New PIN:"*** and wait for the PIN
        - 1.2.4. If PIN is matched, then change to user mode
        - 1.2.5. If PIN is not matched, not numeric and exceeds 4 characters, then ***"Wrong PIN"*** message is displayed and repeat from step 2
    - 1.3. For any further resets
        - 1.3.1. ***"Please press 1 for entering user mode and 2 for programming mode:"*** message is sent to the terminal and wait for a valid response, only accepts 1 or 2
    - 1.4. PAN is 16 to 19 length numeric string
    - 1.5. PIN is 4 numeric digits
    - 1.6. All data taken will be stored in the EEPROM
2. **User Mode**
    - 2.1. The CARD MCU will enter this mode
        - 2.1.1. After completing the programming mode
        - 2.1.2. Or after choosing 2 in any further after resets
    - 2.2. In this mode the CARD ECU will send a trigger signal to the ATM ECU that will make the ATM initiate its flow

1.2. Extras

1.2.1. Communication Map

An *SPI* communication map has been developed, in order to validate each byte transmitted or received, and not to have any corruption issues with any series of bytes in case there was any noise. In the following sections *SPI* commands (ATM MCU) and responses (CARD MCU) are discussed briefly.

1.2.1.1. ATM MCU Commands

1. ATM sends PIN value is ready
    - Macro Value: CMD_ATM_PIN_READY
    - Hex Number: ***0xC1***
2. ATM waits for CARD response
    - Macro Value: CMD_ATM_WAIT_FOR_CARD_RESP
    - Hex Number: ***0xC2***
3. ATM requests PAN
    - Macro Value: CMD_ATM_PAN_REQ
    - Hex Number: ***0xC3***
4. ATM requests PAN length
    - Macro Value: CMD_ATM_PAN_LEN_REQ
    - Hex Number: ***0xC4***
5. ATM requests PAN digit with index 0
    - Macro Value: CMD_ATM_PAN_INDEX_0_REQ
    - Hex Number: ***0xA0***
6. ATM receives full PAN digits
    - Macro Value: CMD_ATM_PAN_OK
    - Hex Number: ***0xF0***

1.2.1.2. CARD MCU Responses

1. CARD responses with acknowledge
    - Macro Value: RESP_CARD_ACK
    - Hex Number: ***0x1A***

2. CARD requests PIN digit with index 0
    - Macro Value: RESP_CARD_PIN_INDEX_0_REQ
    - Hex Number: ***0xA0***
3. CARD responses with PIN verification OK, PIN matches
    - Macro Value: RESP_CARD_PIN_OK
    - Hex Number: ***0xF0***
4. CARD responses with PIN verification WRONG, PIN doesn't match
    - Macro Value: RESP_CARD_PIN_WRONG
    - Hex Number: ***0xF5***

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.

2.1.2. Layered Architecture

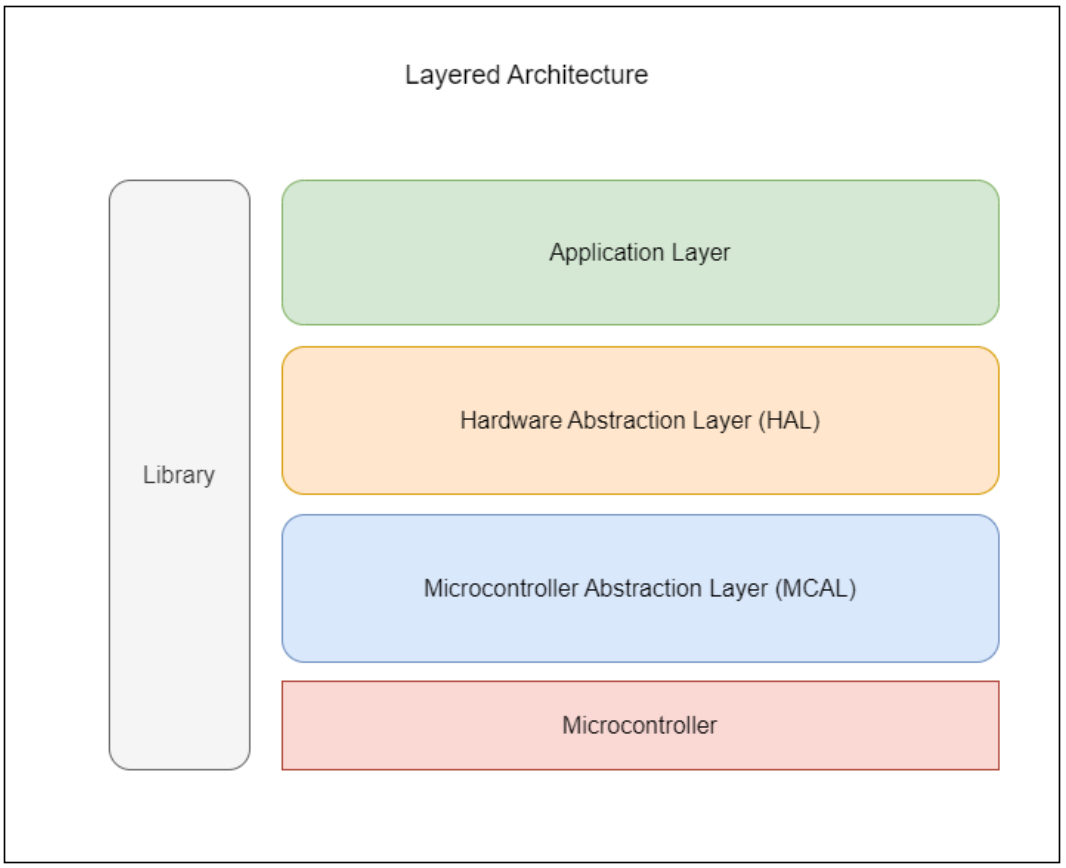


Figure 1. Layered Architecture Design

2.1.3. Project Circuit Schematic

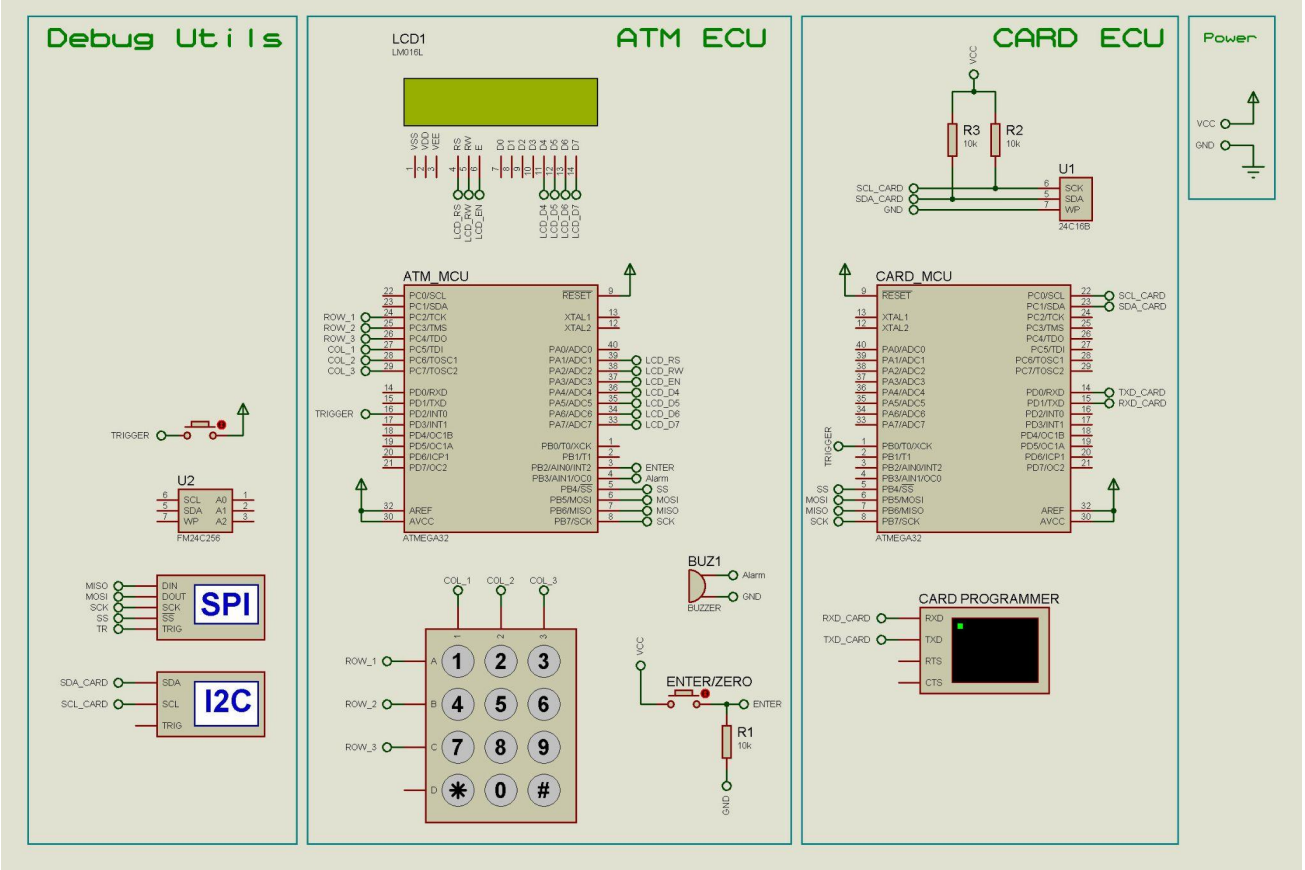


Figure 2. Project Circuit Schematic

2.2. Modules Description

2.2.1. DIO Module

The *DIO* (Digital Input/Output) 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. TWI Module

The Inter-Integrated Circuit (*I2C*) is a two-wire communication protocol that enables the CARD MCU in our Simple ATM Machine project to communicate with the 24C256 *EEPROM*. The *I2C* driver controls the data transfer between the CARD MCU and the *EEPROM* by managing the communication protocol between them. The *I2C* driver allows for bidirectional data transfer with a minimum of wiring, making it ideal for communication between a microcontroller and peripheral devices such as *EEPROM*s. By using the *I2C* driver in our project, we can efficiently read and write data to the *EEPROM*, storing important information such as user account details and transaction history. This ensures that our ATM machine operates smoothly and securely.

2.2.5. SPI Module

The Serial Peripheral Interface (*SPI*) is a synchronous communication protocol that enables the ATM MCU (master) to communicate with the CARD MCU (slave) in our Simple ATM Machine project. The *SPI* driver controls the data transfer between the two MCUs by sending and receiving data in a specific order, with the ATM MCU initiating the communication and the CARD MCU responding accordingly. This allows for high-speed, full-duplex communication between the two MCUs. By utilizing the *SPI* driver in our project, we can ensure reliable and efficient communication between the ATM ECU and the CARD ECU.

2.2.6. UART Module

The Universal Asynchronous Receiver-Transmitter (*UART*) is a popular asynchronous communication protocol used in our Simple ATM Machine project for communication between the CARD MCU and the card programmer. The *UART* driver is responsible for managing the data transfer between the two devices. The *UART* protocol allows for two-way data transfer, making it ideal for communication with external devices. By utilizing the *UART* driver in our project, we can configure the card to interact with our ATM machine, enabling users to access their accounts securely.

2.2.7. MBTN Module

The Multifunction Button (*MBTN*) driver in our Simple ATM Machine project is responsible for controlling the functionality of a single button on the ATM ECU. The *MBTN* driver enables the button to be used as either a 0 digit input when clicked briefly or as an Enter button when held down for more than two seconds. The *MBTN* driver is programmed to detect the duration of the button press and send the appropriate signal to the ATM MCU. By using the *MBTN* driver in our project, we can provide users with an intuitive and efficient way to input digits and submit transactions. The driver ensures that the button operates reliably, registering input accurately and responding quickly to user commands. This makes our ATM machine more user-friendly and helps to provide a better overall user experience.

2.2.8. BUZZER Module

The *Buzzer* module provides a set of functions that enable an embedded system to control a *Buzzer*. The module is typically used when the system needs to provide audio feedback to the user or signal an event. The module is designed to be simple and easy to use. It provides functions to initialize the *Buzzer* pin and turn it on and off. The initialization function typically sets the *Buzzer* pin as an output and performs any necessary configuration.

2.2.9. KEYPAD Module

Keypad is an analog switching device which is generally available in matrix structure. It is used in many embedded system applications for allowing the user to perform a necessary task. A matrix *Keypad* consists of an arrangement of switches connected in matrix format in rows and columns. The rows and columns are connected with a microcontroller such that the rows of switches are connected to one pin and the columns of switches are connected to another pin of a microcontroller.

2.2.10. LCD Module

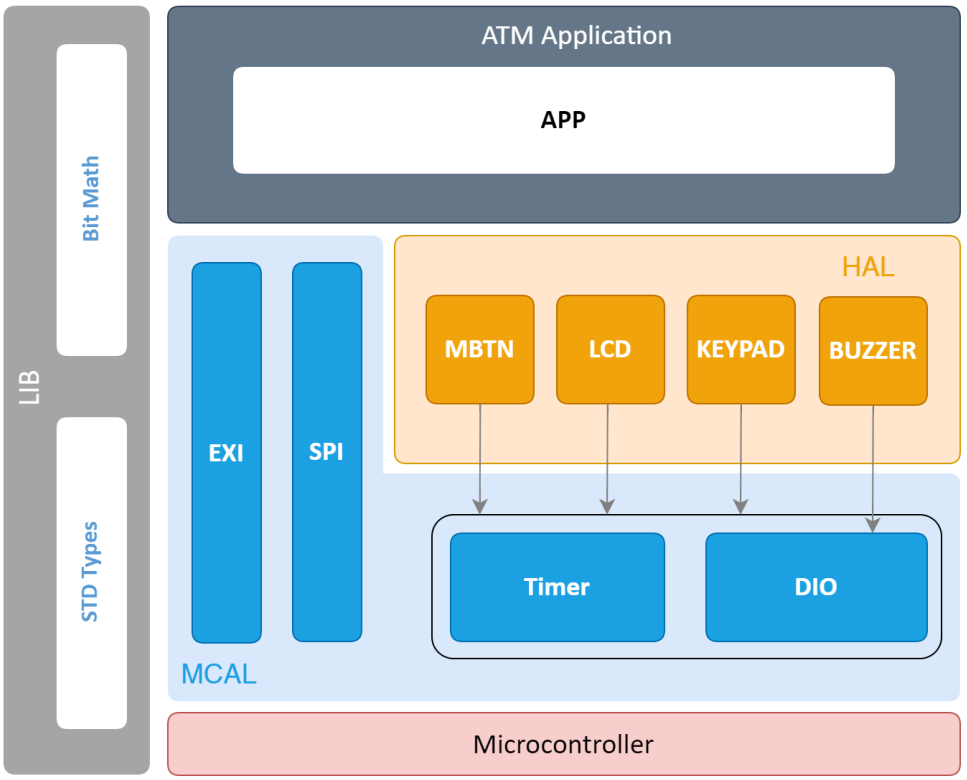
LCD stands for "*Liquid Crystal Display*," which is a type of flat-panel display used in electronic devices to display text and graphics. The module is being used in our project to display information about the current and desired temperature of an air conditioning system. We're using the *4-bit* mode to reduce the number of I/O pins needed to interface with the *LCD*. The module includes a controller, a display, and a backlight. By interfacing with the *LCD* module and writing the necessary code, we're able to provide the user with real-time information about the temperature of the air conditioning system and the ability to set a desired temperature.

2.2.11. EEPROM Module

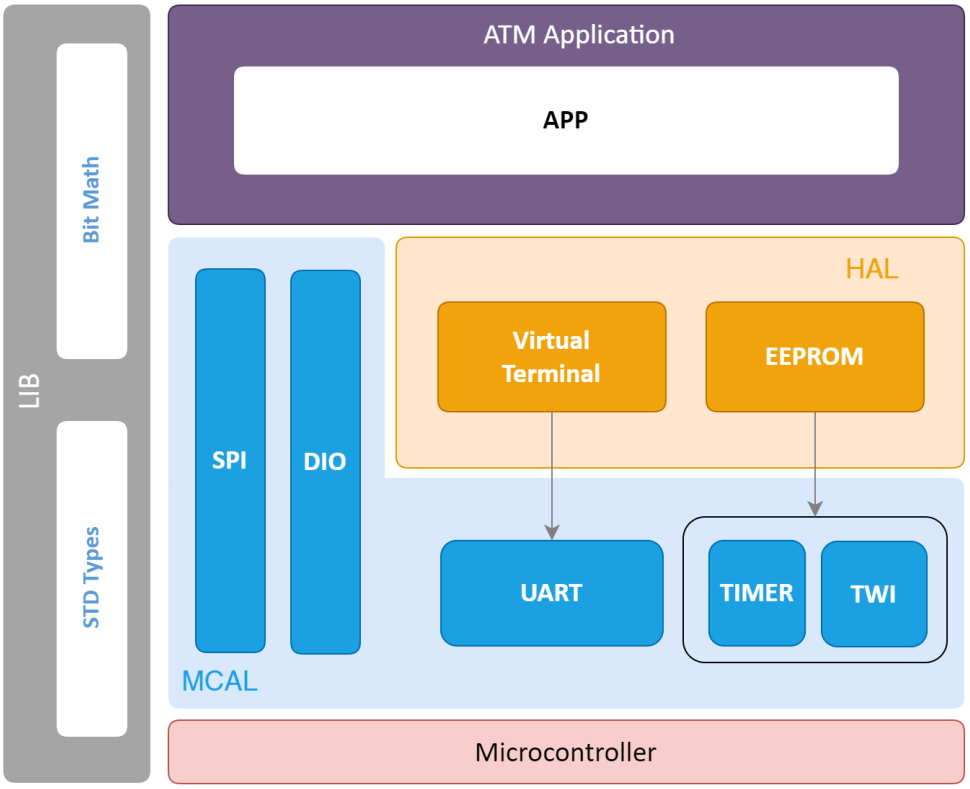
The *EEPROM* driver in our Simple ATM Machine project is responsible for controlling the read and write operations to the 24C256 *EEPROM*. The driver is used by the CARD MCU to store the verified PAN and PIN of a single card in the *EEPROM* after successful authentication. The *EEPROM* driver ensures that the data is written accurately and reliably, and can be retrieved as needed by the CARD MCU for future transactions. By utilizing the *EEPROM* driver in our project, we can ensure that user data is securely stored and easily accessible. The driver ensures that the *EEPROM* operates efficiently and accurately, providing a high level of reliability and data integrity. This helps to make our ATM machine more secure and user-friendly, providing a better overall user experience.

2.2.12. Design

2.2.12.1. ATM ECU



2.2.12.2. CARD ECU



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_portDir    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_PORT_DIRECTION_T en_a_portDir, 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_delay_ms(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.
|
| 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);


| 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 timeout delay in msy 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_intDelay_ms(u16 u16_a_interval);



| Set callback function for timer overflow interrupt
|
| Parameters
|           [in] 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 Timer2 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(TIMER2_ovfVect);
```

2.3.2.4. TWI Driver

```
| Initializes  protocol
void TWI_init();


| Write  one byte data/address via I2C
void TWI_write(void);


| Stop the frame
void TWI_stop();


| start the frame
void TWI_start();


| read one byte with send Ack via I2C
|
| Return
|      Received byte
|
u8 TWI_readWithAck(void);


| read one byte with send Ack via I2C
|
| Return
|      Received byte
|
u8 TWI_readWithNAck(void);


| checks the last operation via get the data of status register
|
| Return
|      current status
|
u8 TWI_getStatus(void);
```

2.3.2.4. SPI Driver

```
| Initializes SPI protocol
void SPI_init();



| Receive and Transmit one byte via SPI
|
| Parameters
|            [in] u8_a_byte byte to send
| Return
|     Received byte
|
u8 SPI_transceiver(u8 u8Ptr_a_byte);



| Syncs and restarts SPI communications between Master and Slave by
|  setting SS pin High then Low again
|
void SPI_restart();



| stops SPI communications by setting SS pin to HIGH
void SPI_stop();
```

2.3.2.4. UART Driver

```
| The function initializes the UART communication protocol by configuring various
| settings such as mode, speed, parity, data size, and enabling/disabling receiver
| and transmitter.
void UART_initialization(void);


| This function receives a byte from the UART communication interface in either
| polling or interrupt mode.
|
| Parameters
|           [in] u8_a_interruptionMode This parameter specifies the mode of
|                operation for receiving data from the UART. It can be either
|                polling mode or interrupt mode.
|           [in] pu8_a_returnedReceiveByte A pointer to a variable where the
|                                          received byte will be stored.
| Return
|     An error state, which is either STD_OK or STD_NOK.
|
u8 UART_receiveByte(u8 u8_a_interruptionMode, u8 *pu8_a_returnedReceiveByte);


| This function receives a byte through UART communication and returns an
|  error state.
|
| Parameters
|           [in] pu8_a_returnedReceiveByte A pointer to a variable where the
|                                          received byte will be stored.
| Return
|     An error state, which is either STD_OK or STD_NOK.
|
u8 UART_receiveByteBlock(u8 *pu8_a_returnedReceiveByte);


| This function transmits a byte through UART communication in either
|  polling or interrupt mode.
|
| Parameters
|           [in] u8_a_interruptionMode This parameter specifies the mode of
|                operation for receiving data from the UART. It can be either
|                polling mode or interrupt mode.
|           [in] u8_a_transmitByte The byte to be transmitted through UART.
| Return
|     An error state, which can be either STD_OK or STD_NOK.
|
u8 UART_transmitByte(u8 u8_a_interruptionMode, u8 u8_a_transmitByte);
```

```
| The function transmits a string through UART communication and returns an error
| state.
|
| Parameters
|           [in] pu8_a_string A pointer to a string that needs to be
|                transmitted via UART.
| Return
|     An error state, which is either STD_OK or STD_NOK.
|
u8 UART_transmitString(u8 *pu8_a_string);


| This function sets a callback function for the UART receive interrupt and
| returns an error state.
|
| Parameters
|           [in] vpf_a_RXCInterruptAction A pointer to a function that will be
|           called when a UART receive interrupt occurs.
| Return
|     An error state, which can be either STD_OK or STD_NOK.
|
u8 UART_RXCSetCallBack(void(*vpf_a_RXCInterruptAction)(void));


| This function sets a callback function for the UART UDRE interrupt and
| returns an error state.
|
| Parameters
|           [in] vpf_a_UDREInterruptAction A pointer to a function that will be
|                called when the UART Data Register Empty (UDRE) interrupt occurs.
| Return
|     An error state, which can be either STD_OK or STD_NOK.
|
u8 UART_UDRESetCallBack(void(*vpf_a_UDREInterruptAction)(void));


| This function sets a callback function for the UART transmit interrupt and
| returns an error state.
|
| Parameters
|           [in] vpf_a_TXCInterruptAction A pointer to a function that will be
|                called when a UART transmitter interrupt occurs.|
| Return
|     An error state, which can be either STD_OK or STD_NOK.
|
u8 UART_TXCSetCallBack(void(*vpf_a_TXCInterruptAction)(void));
```

2.3.3. HAL APIs

2.3.3.1. MBTN APIs

```
| Initializes a push button connected to a specific pin in a specific port.
|
| This function initializes a push button connected to a specific pin in
|  a specific port.
| It configures the pin as input, enables the pull-up resistor, and
|  initializes the timer.
|
| Parameters
|            [in]u8_a_pinNumber The number of the pin to which the button is
|                        connected.
|            [in]en_a_portNumber The port to which the button is connected.
|
|
| Return
|      Returns the state of the initialization operation. STD_OK if
|             successful and STD_NOK if failed.
|
u8 MBTN_init(u8 u8_a_pinNumber, EN_DIO_PORT_T en_a_portNumber);


| This function gets the state of a mechanical button (pressed, not
|  pressed, long press)
|
| Parameters
|            [in]u8_a_pinNumber The pin number of the button
|            [in]en_a_portNumber The port number of the button
|            [out]u8Ptr_a_returnedBtnState A pointer to the variable to store
|                                     the button state (pressed, not pressed, long
press)
|
|
| Return
|      Returns the status of the function (STD_OK, STD_NOK)
|
u8 MBTN_getBtnState(u8 u8_a_btnId, EN_DIO_PORT_T en_a_portNumber, u8
*u8ptr_a_returnedBtnState);
```

2.3.3.2. BUZ APIs

```
| Initialize the buzzer pin
|
| Return
|      void
|
void BUZZER_init();

| Turn the buzzer on
|
| Return
|      void
|
void BUZZER_on();

| Turn the buzzer off
|
| Return
|      void
|
void BUZZER_off();
```

2.3.3.3. LCD APIs

```
| Initializes the LCD module.
|
| This function initializes the LCD module by configuring the data port,
| configuring the LCD to 4-bit mode,setting the display to on with cursor
| and blink, setting the cursor to increment to the right, and clearing
|  the display.
| It also pre-stores a bell shape at CGRAM location 0.
|
| Return
|     void
|
void LCD_init(void);



| Sends a command to the LCD controller
|
| Sends the upper nibble of the command to the LCD's data pins, selects
|  the command register by setting RS to low,
| generates an enable pulse, delays for a short period, then sends the
|  lower nibble of the command and generates
| another enable pulse. Finally, it delays for a longer period to ensure
|  the command has been executed by the LCD
| controller.
|
| Parameters
|            [in] u8_a_cmd The command to be sent
|
void LCD_sendCommand(u8 u8_a_cmd);



| Sends a single character to the LCD display
|
| This function sends a single character to the LCD display by selecting
|  the data register and sending the
| higher nibble and lower nibble of the character through the data port.
| The function uses a pulse on the enable pin to signal the LCD to read
|  the data on the data port.
| The function also includes delays to ensure proper timing for the LCD
|  to read the data.
|
| Parameters
|            [in] u8_a_data single char ASCII data to show
|
void LCD_sendChar(u8 u8_a_data);
```

```
| Displays a null-terminated string on the LCD screen.
|
| This function iterates through a null-terminated string and displays it
| on the LCD screen. If the character '\n' is encountered, the cursor is
| moved to the beginning of the next line.
|
| Parameters
|            [in]u8Ptr_a_str A pointer to the null-terminated string to be
|                        displayed.
| Return
|     void
|
void LCD_sendString(u8 * u8Ptr_a_str);




| Set the cursor position on the LCD.
|
| Parameters
|            [in]u8_a_line the line number to set the cursor to, either
|                        LCD_LINE0 or LCD_LINE1
|            [in]u8_a_col the column number to set the cursor to, from
|                        LCD_COL0 to LCD_COL15
| Return
|     STD_OK if the operation was successful, STD_NOK otherwise.
|
u8 LCD_setCursor(u8 u8_a_line, u8 u8_a_col);




| Stores a custom character bitmap pattern in the CGRAM of the LCD module
|
| Parameters
|            [in] u8_a_pattern Pointer to an array of 8 bytes representing
|                        the bitmap pattern of the custom character
|            [in] u8_a_location The CGRAM location where the custom
|                        character should be stored (from LCD_CUSTOMCHAR_LOC0 to 7)
| Return
|     STD_OK if successful, otherwise STD_NOK
|
u8 LCD_storeCustomCharacter(u8 * u8_a_pattern, u8 u8_a_location);
```

```
| Show/Hide cursor
|
| Parameters
|            [in]u8_a_show 0: hide, otherwise: show
|
void LCD_changeCursor(u8 u8_a_show);


| Shift clears the LCD display
void LCD_shiftClear(void);

| Clears the LCD display
void LCD_clear(void);
```

2.3.3.4. KPD APIs

```
| Initializes the KPD module.
|
| This function initializes the pins of a keypad by setting some as output and others
| as input.
|
| Return
|      void
|
void KPD_initKPD(void);


| Enables or Re-enables the KPD module.
|
| This function enables or re-enables a keypad by setting one pin as output and the
| other three as input.
|
| Return
|      void
|
void KPD_enableKPD(void);


| Disables the KPD module.
|
| This function disables the keypad by setting its output pins to input.
|
| Return
|      void
|
void KPD_disableKPD(void);


| This function reads input from a keypad and returns the pressed key value after
| debouncing.
|
| Parameters
|  -          [in] pu8_a_returnedKeyValue Pointer to a u8 variable that will hold the
|                  value of the pressed key.
| Return
|      STD_OK if successful, otherwise STD_NOK
|
u8 KPD_getPressedKey(u8 *pu8_a_returnedKeyValue);
```

2.3.3.3. EEPROM APIs

```
| Initializes TWI protocol
void EEPROM_init();


| Write one byte data to specific address in the eeprom
|
| Parameter
|       [in] u16_l_byteAddress in memory
|       [in] u8_l_byteData (byte) to be written on the EEPROM
| Return
|       u8 error_state
|
u8 EEPROM_writeByte(u16 u16_l_byteAddress, u8 u8_l_byteData);


| Read one byte data to specific address in the eeprom
|
| Parameter
|         [in] u16_l_byteAddress in memory
|        [out] u8_l_byteData pointer to read data (byte) into
|
| Return
|       u8 error_state
|
u8 EEPROM_readByte(u16 u16_l_byteAddress, u8 *u8_l_byteData);


| Write array of data to specific address in the eeprom
|
| Parameter
|         [in] u16_l_byteAddress to store the string
|        [out] u8_l_ptrToStr to store the string
| Return
|      u8 error_state
|
u8 EEPROM_writeArray(u16 u16_l_byteAddress, u8 *u8_l_ptrToStr);


| Read one byte with send Ack via TWI
|
| Parameter
|         [in] u16_l_byteAddres of the stored string
| Return
|          u8 pointer to static string which contains the data inside this address
|
u8 *EEPROM_readArray(u16 u16_l_byteAddres);
```

2.3.4. ATM APP APIs

```
| Initializes the application by initializing the MCAL and HAL components.
|
| Details
|       This function initializes the external interrupt, SPI, timer0,
|       buzzer, button, keypad, and LCD.
|       It also clears the LCD's display and switches to the entry point state.
|
| Return
|     void
|
void APP_initialization(void);



| Start the application program flow for ATM
|
| This function starts the application program and enters an infinite
| loop that continuously polls the application state and executes the
| corresponding actions according to the current state.
|
| Return
|     void
|
void APP_startProgram(void);



| Used to switch between app states to initialize standard UI elements
| before main app flow (loop)
|
| Parameters
|           [in] u8_a_state state to set (APP_STATE_LAUNCH, APP_STATE_...)
|
| Return
|     void
|
void APP_switchState(u8 u8_a_state);



| Ran when an INT0 interrupt is fired / Card inserted, switches app state to
| INSERT_PIN
void APP_trigger(void);
```

2.3.5. CARD APP APIs

```
| This function initializes various MCAL and HAL components.
void APP_initialization(void);

| The function starts the program and checks for data in memory before switching
| between programmer mode, user mode, and check mode based on the current app mode.
void APP_startProgram(void);

| This function checks for the presence and validity of PAN and PIN data in memory.
|
| Return
|     [out] u8_l_dataFlag value which represents the data flag. It can be either
|           APP_U8_DATA_FOUND or APP_U8_DATA_NOT_FOUND.
|
u8 APP_checkDataInMemory(void);

| The function checks for user input and sets the application mode accordingly.
void APP_checkUserInput(void);

| This function receives PAN and PIN from a terminal and transmits a message
| through UART.
void APP_programmerMode(void);

| This function receives a card PAN from the terminal, checks its validity,
| saves it in memory (EEPROM), and displays a success message.
void APP_receivePANFromTerminal(void);

| This function receives a card PIN from the terminal, checks its validity,
| saves it in memory (EEPROM), and displays a success message.
void APP_receivePINFromTerminal(void);

| This function receives a PIN from an ATM ECU and sends a PAN to the ECU, with a
| maximum of three trials for invalid PINs.
void APP_userMode(void);

| This function receives a PIN from an ATM ECU, compares it with a PIN from card ECU,
| and sends a response to the ATM ECU indicating whether the PIN is correct or not.
|
| Return
|     [out] u8_l_errorState a variable represents the error state. It will return
|           STD_OK if the PIN received from the ATM matches the PIN on the card,
|           and STD_NOK if it doesn't.
u8 APP_receivePINFromATM(void);

| This function sends the card PAN to the ATM ECU in response to requests and receives
| acknowledgements using SPI communication.
void APP_sendPANToATM(void);
```

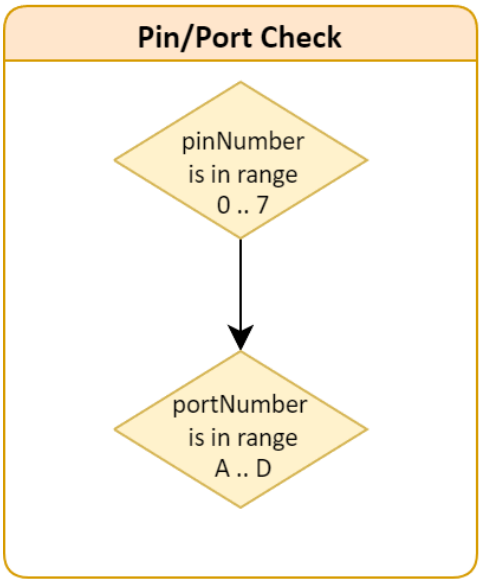
3. Low Level Design

3.1. MCAL Layer

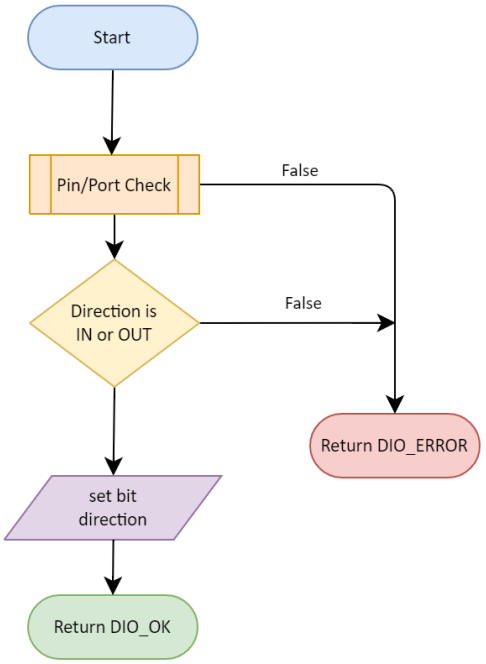
3.1.1. DIO Module

3.1.1.a. sub process

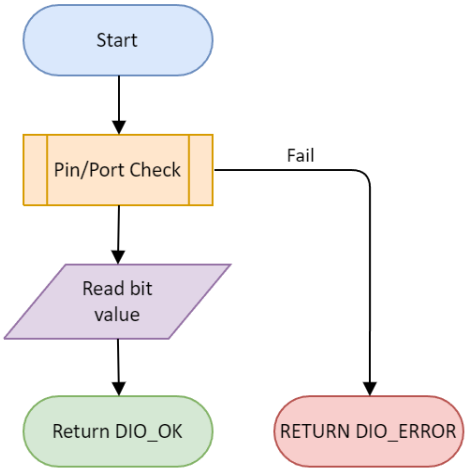
The following Pin/Port check subprocess is used in some of the DIO APIs flowcharts



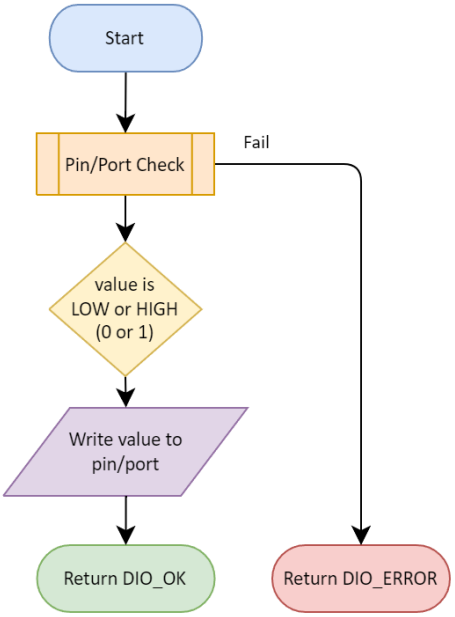
3.1.1.1. DIO_init



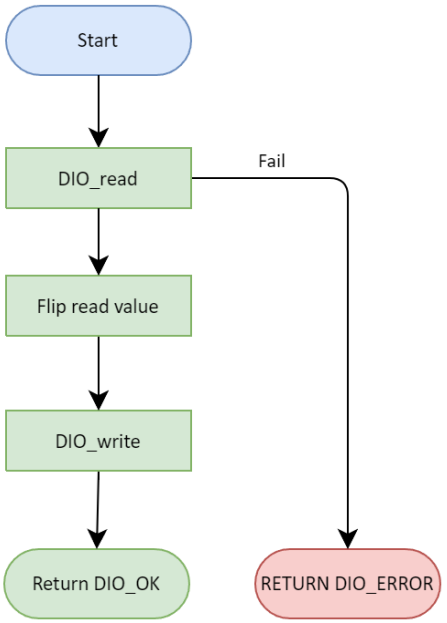
3.1.1.2. DIO_read



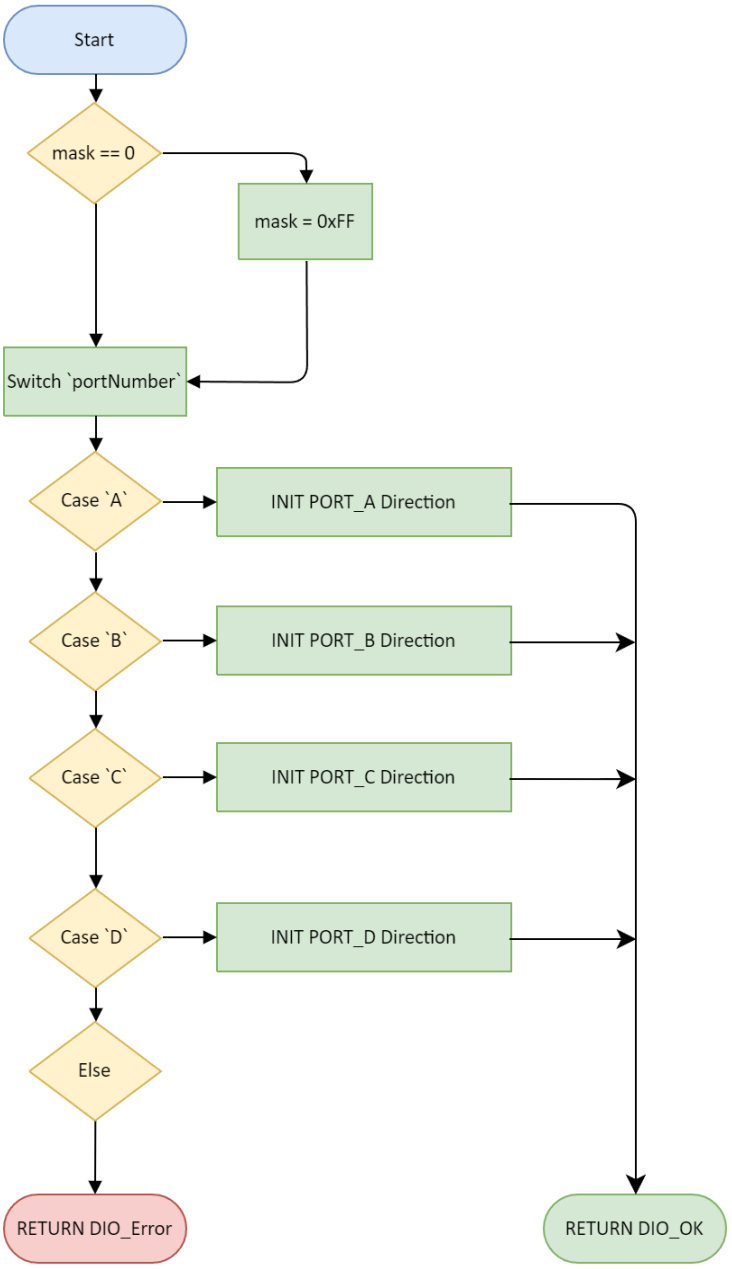
3.1.1.3. DIO_write



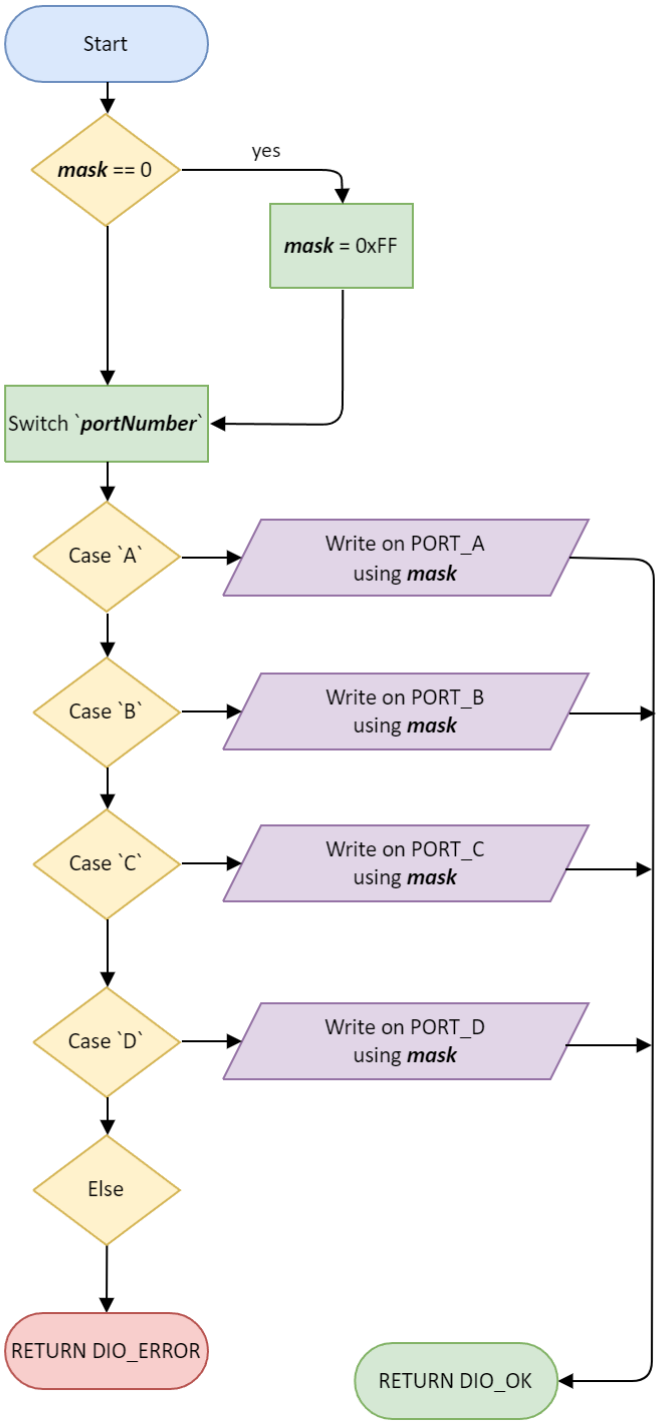
3.1.1.4. DIO_toggle



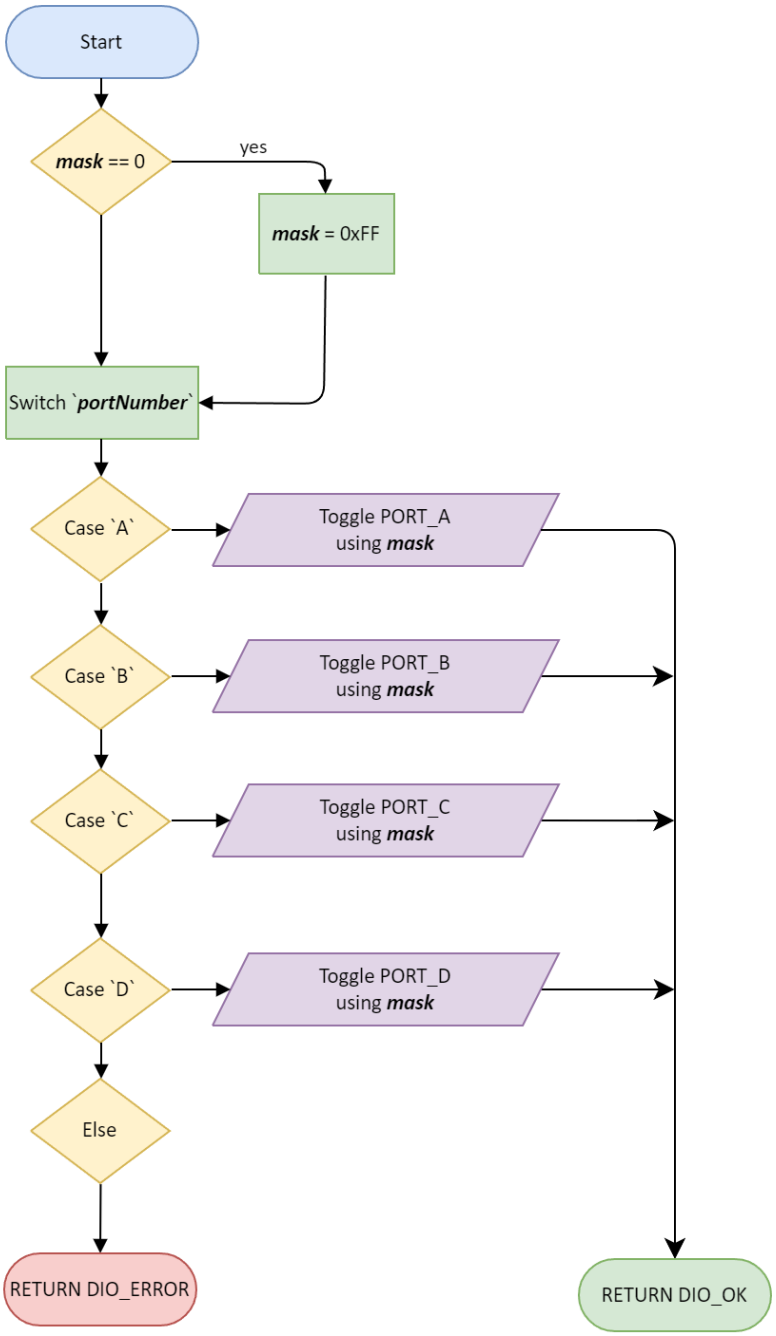
3.1.1.5. DIO_portInit



3.1.1.6. DIO_portWrite

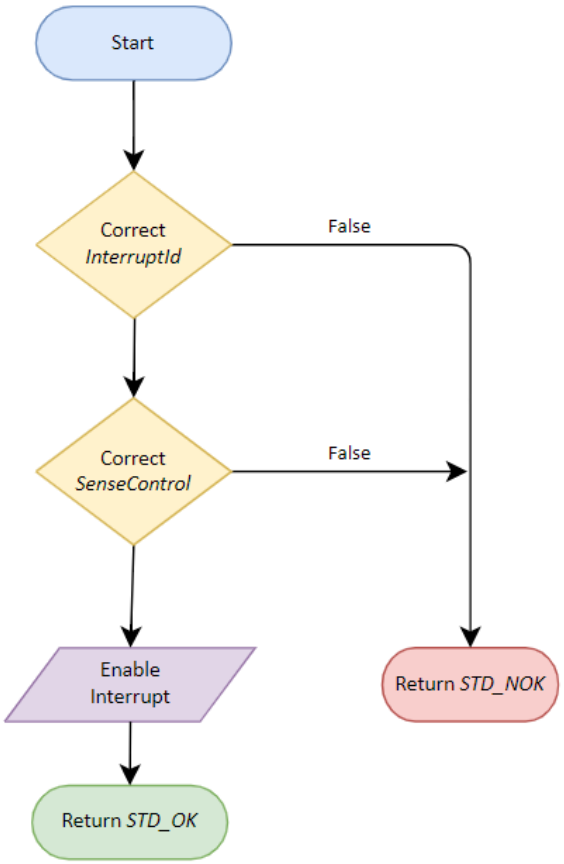


3.1.1.7. DIO_portToggle

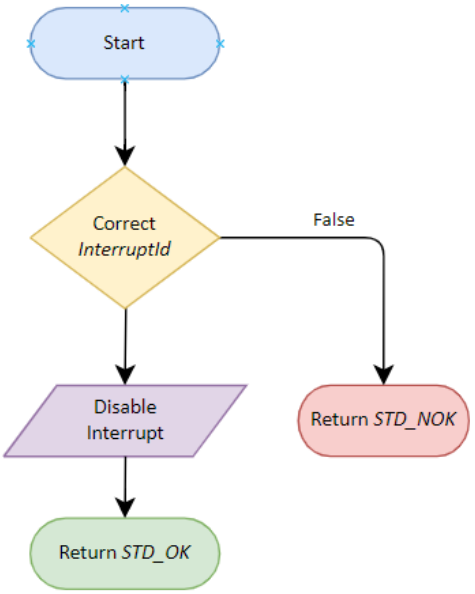


3.1.2. EXI Module

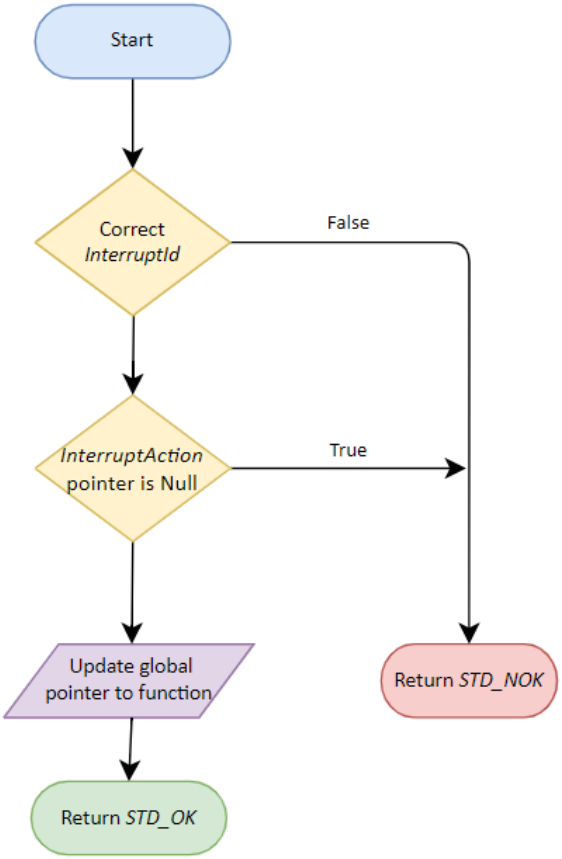
3.1.2.1. EXI_enablePIE



3.1.2.2. EXI_disablePIE

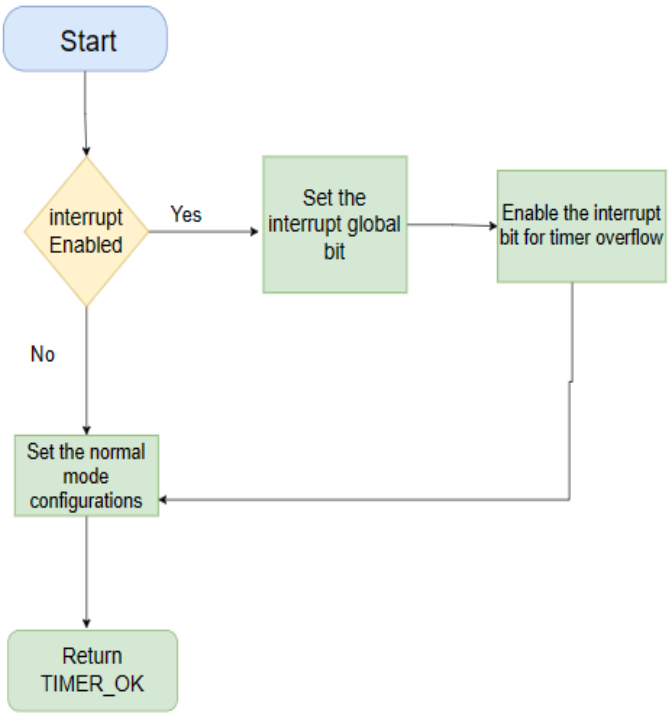


3.1.2.3. EXI_intSetCallback

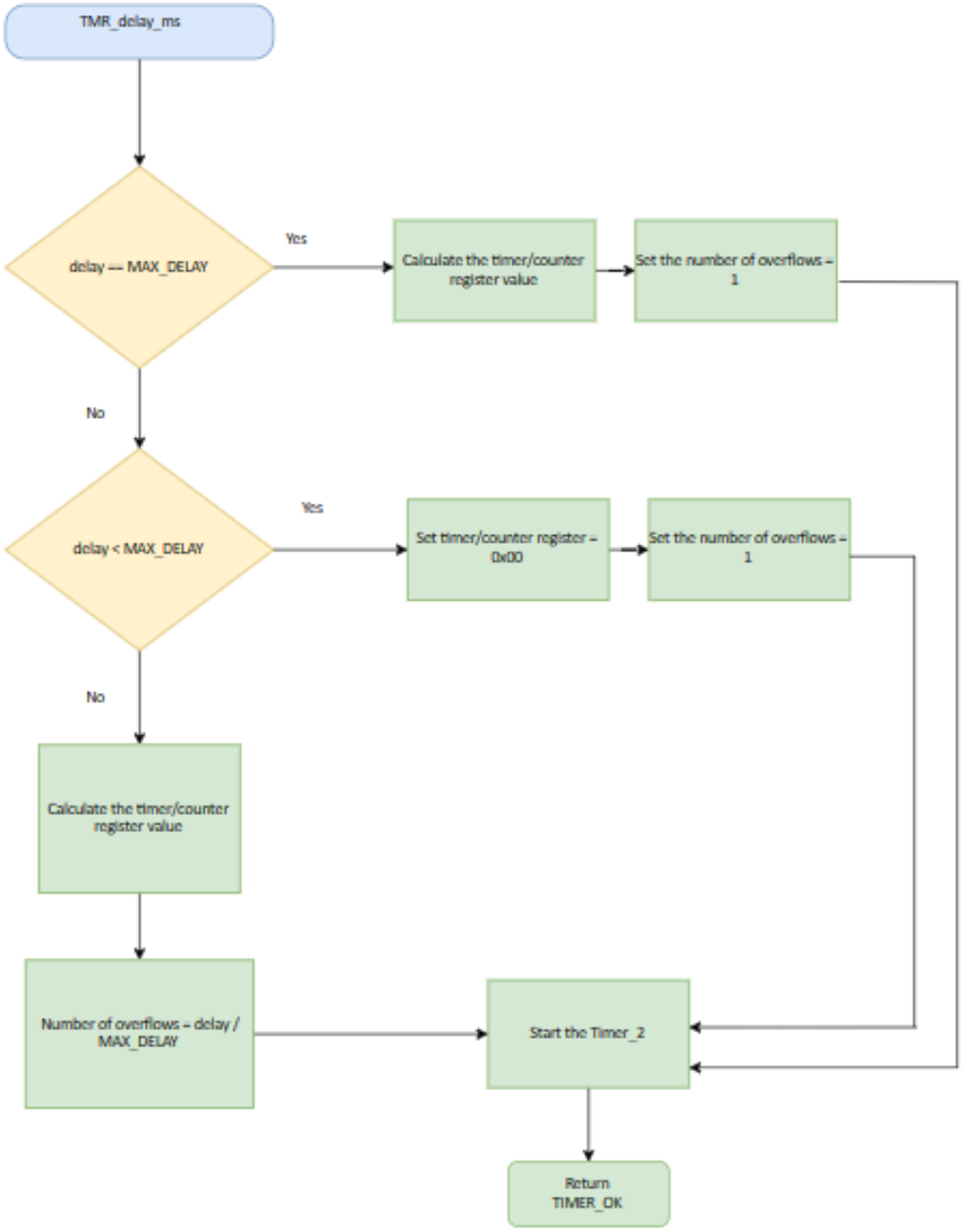


3.1.3. Timer Module

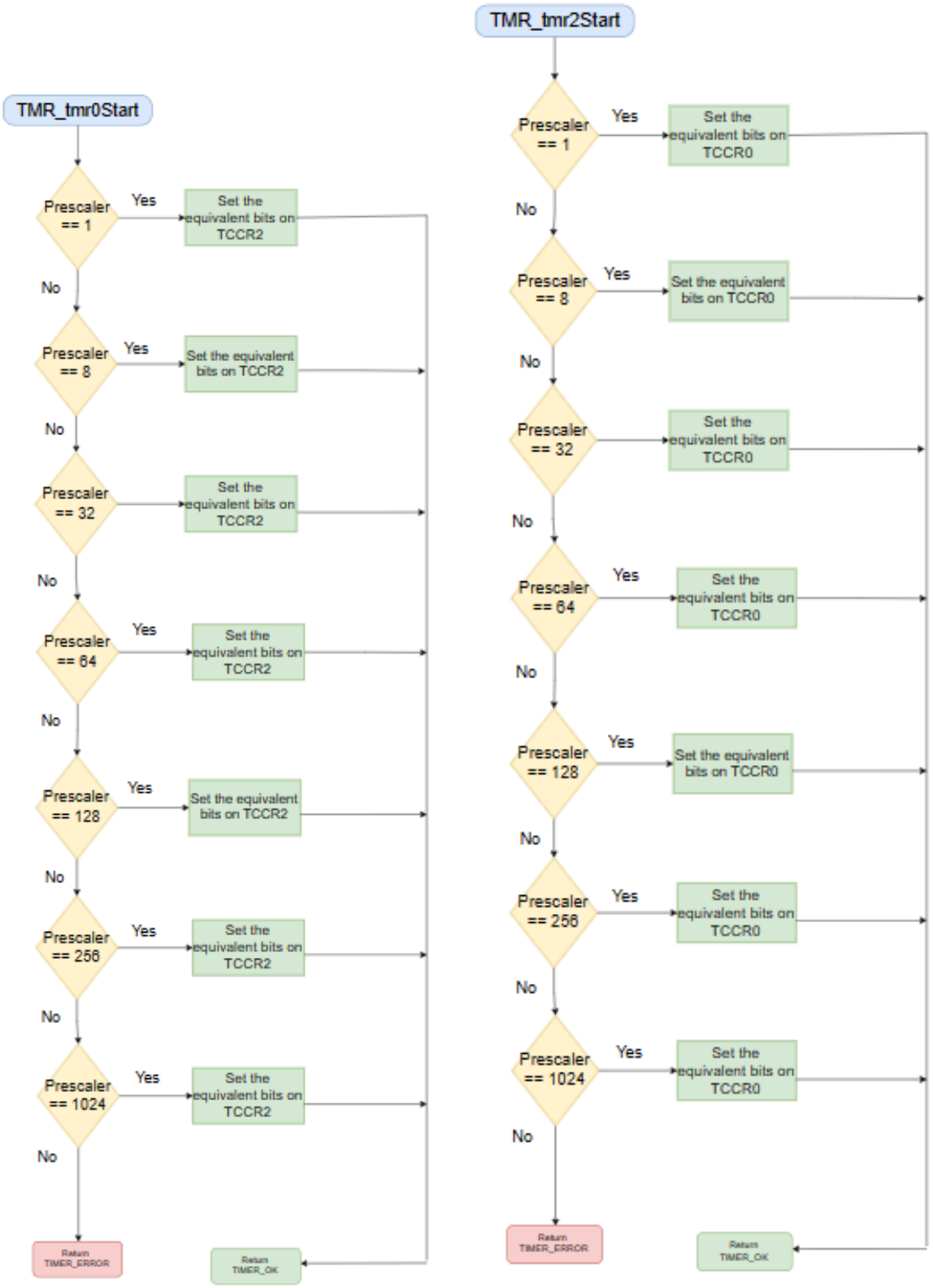
3.1.3.1. TMR_tmr0NormalModeInit / TMR_tmr2NormalModeInit



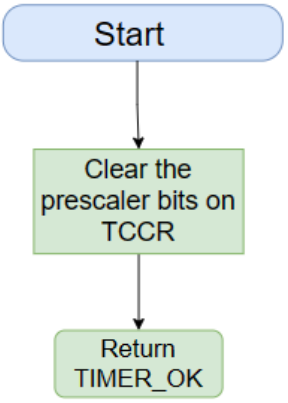
3.1.3.2. TMR_tmr0Delay / TMR_tmr2Delay



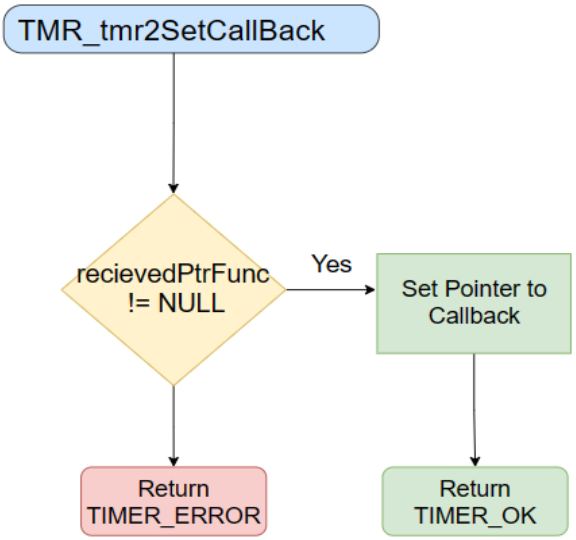
3.1.3.3. TMR_tmr0Start / TMR_tmr2Start



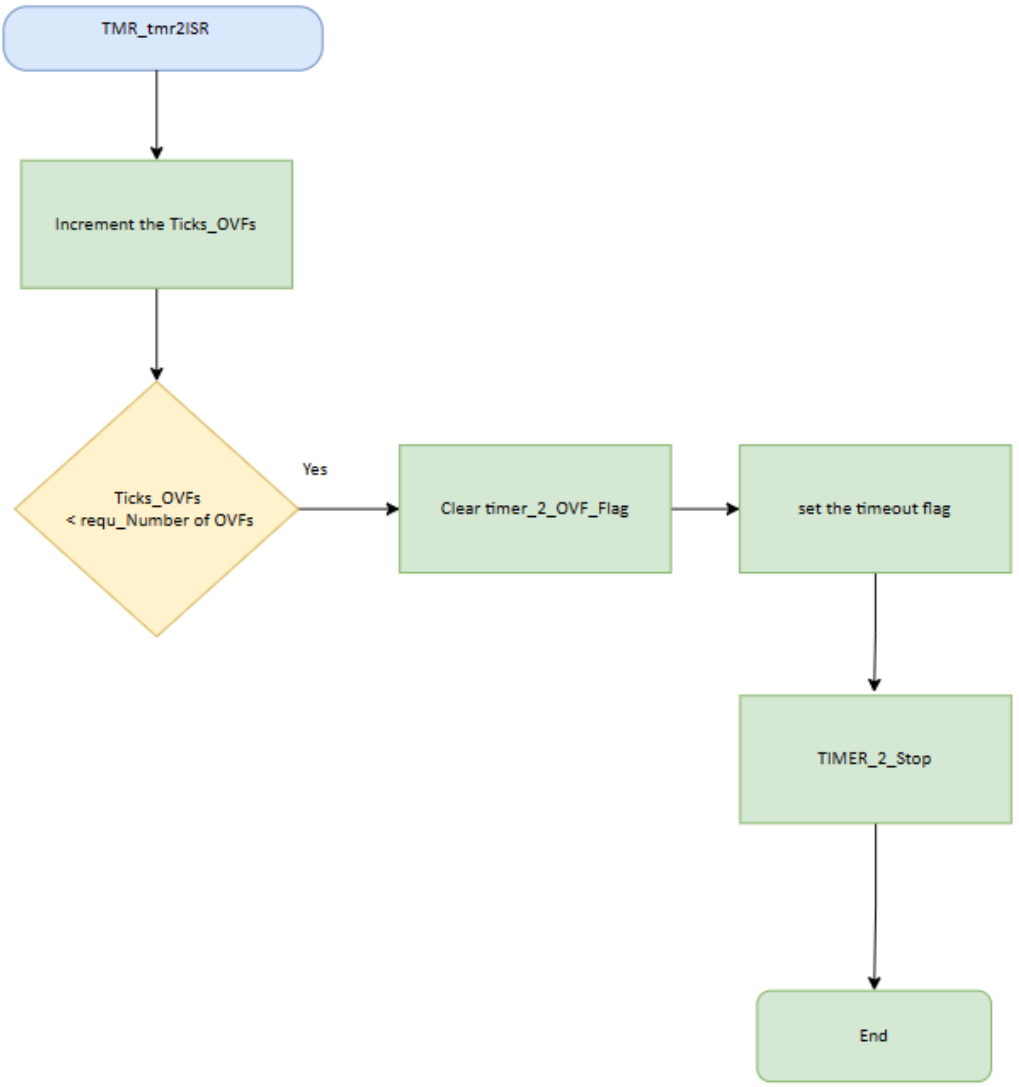
3.1.3.4. TMR_tmr0Stop / TMR_tmr2Stop



3.1.3.5. TMR_ovfSetCallback

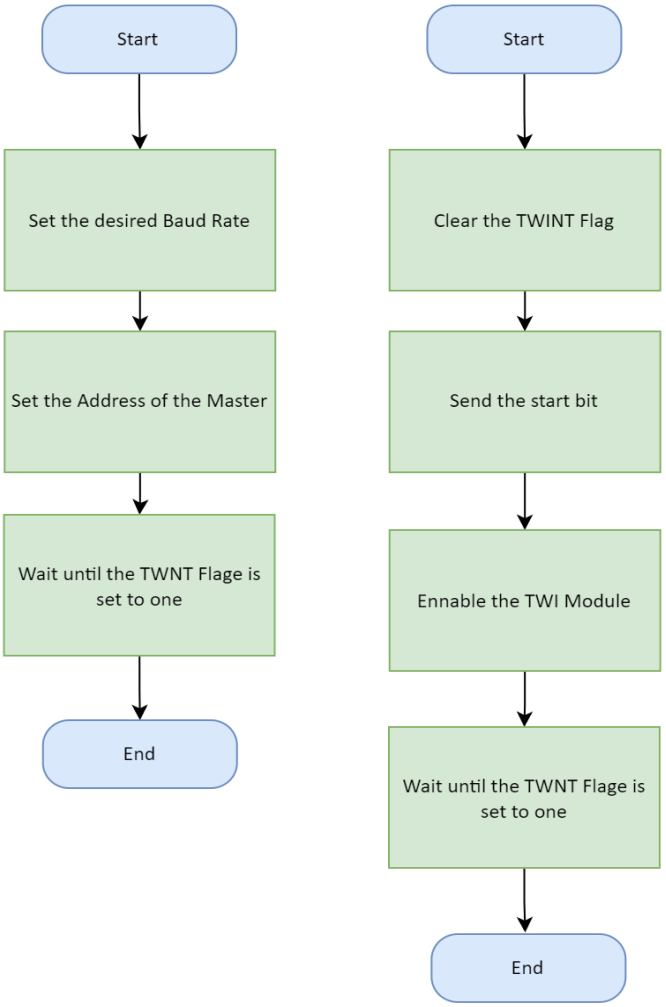


3.1.3.6. TMR2_ovfVect

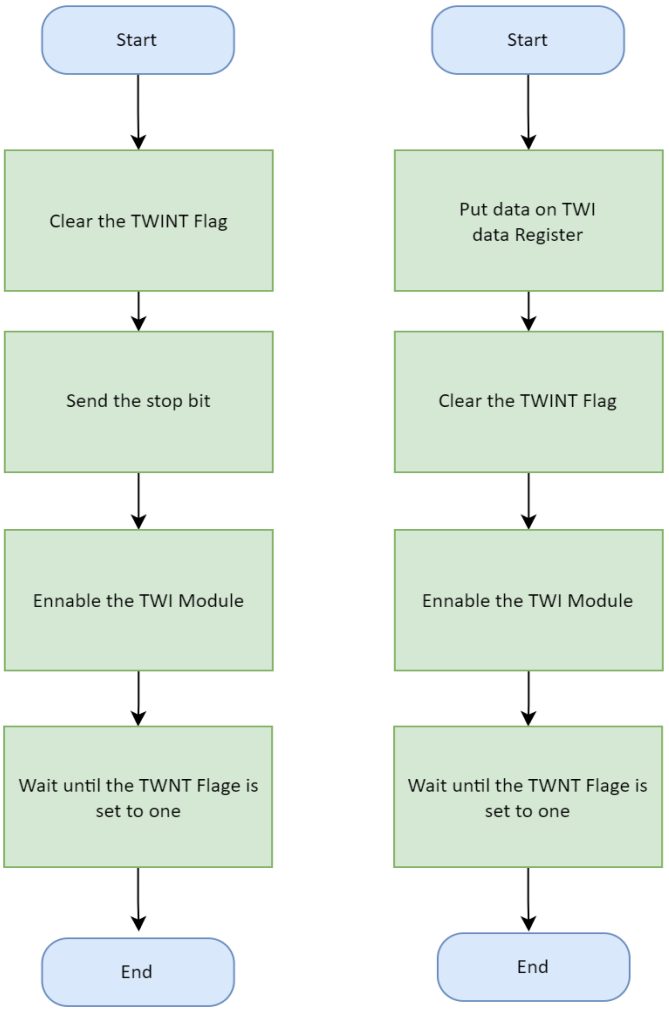


3.1.4. TWI Module

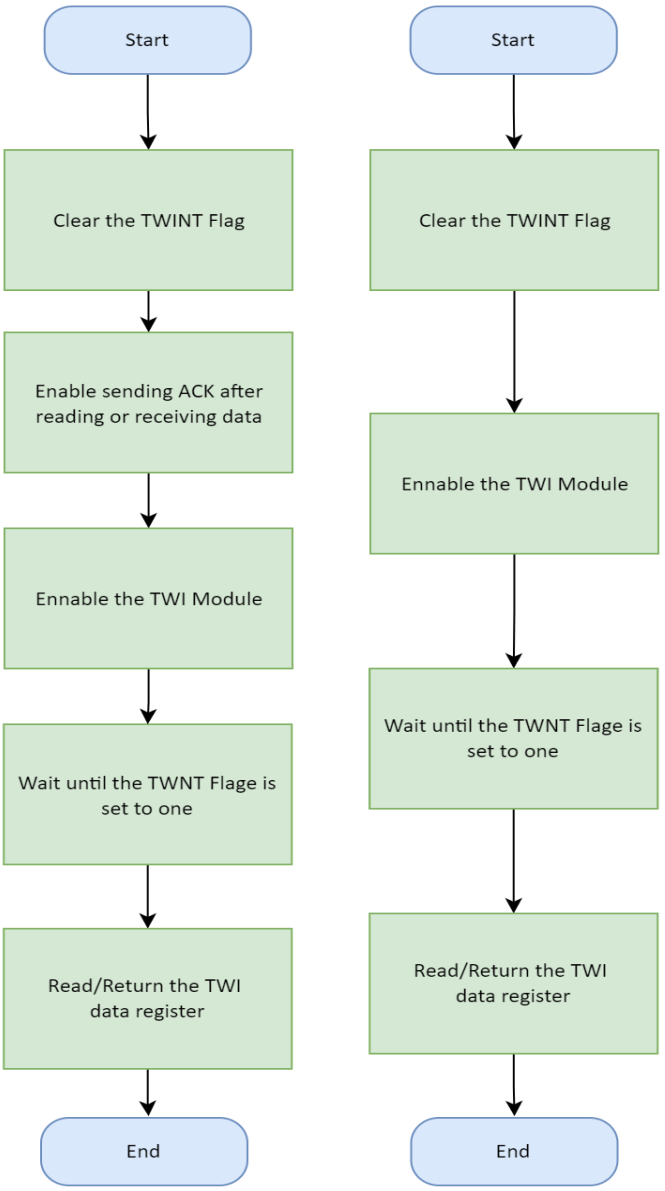
3.1.4.1 TWI_init & TWI_start



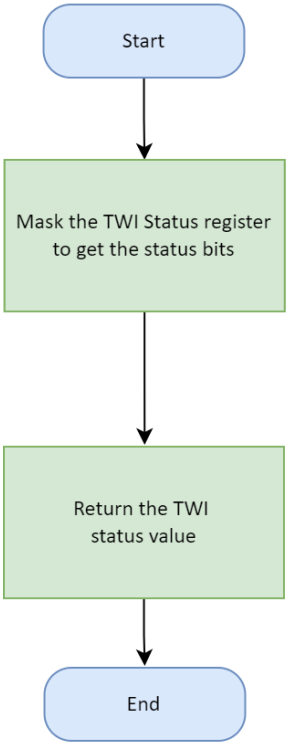
3.1.4.2 TWI_stop & TWI_write



3.1.4.3 TWI_readWithAck & TWI_readWithNAck

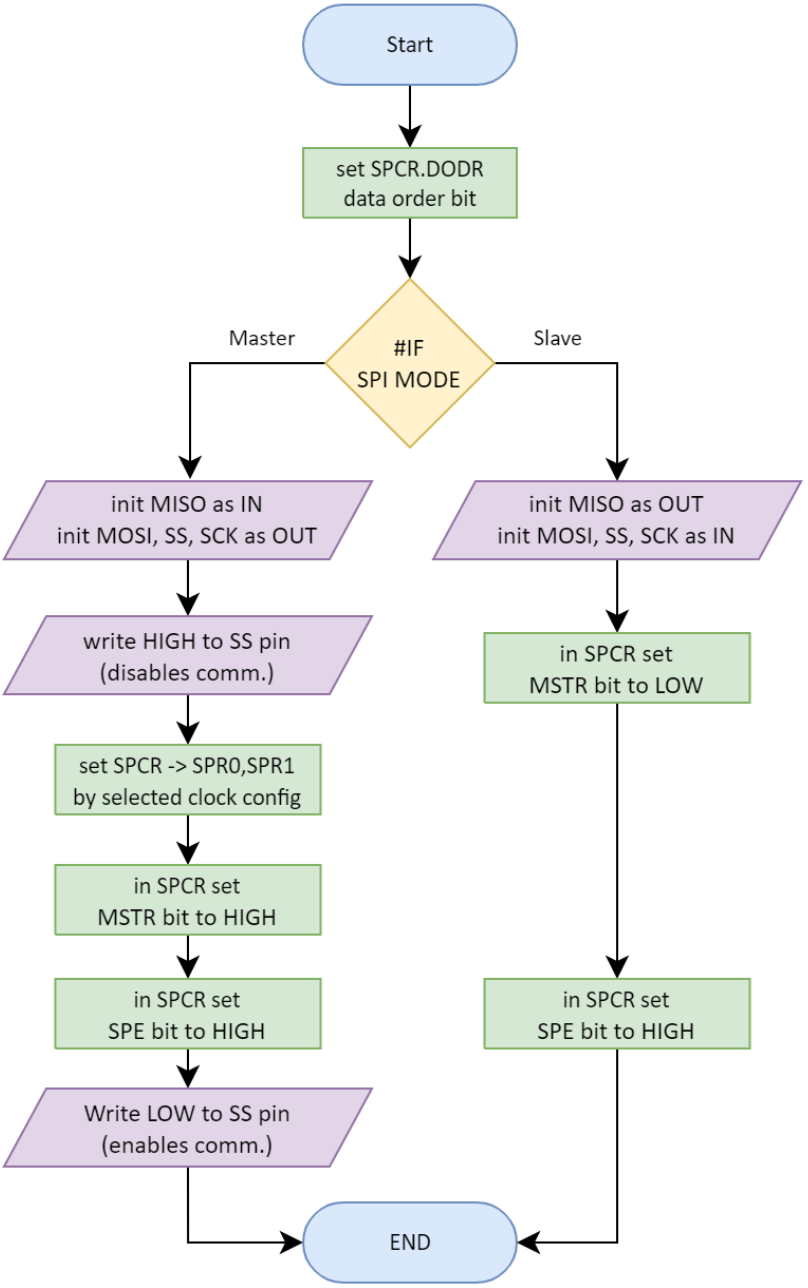


3.1.4.4 TWI_getStatus

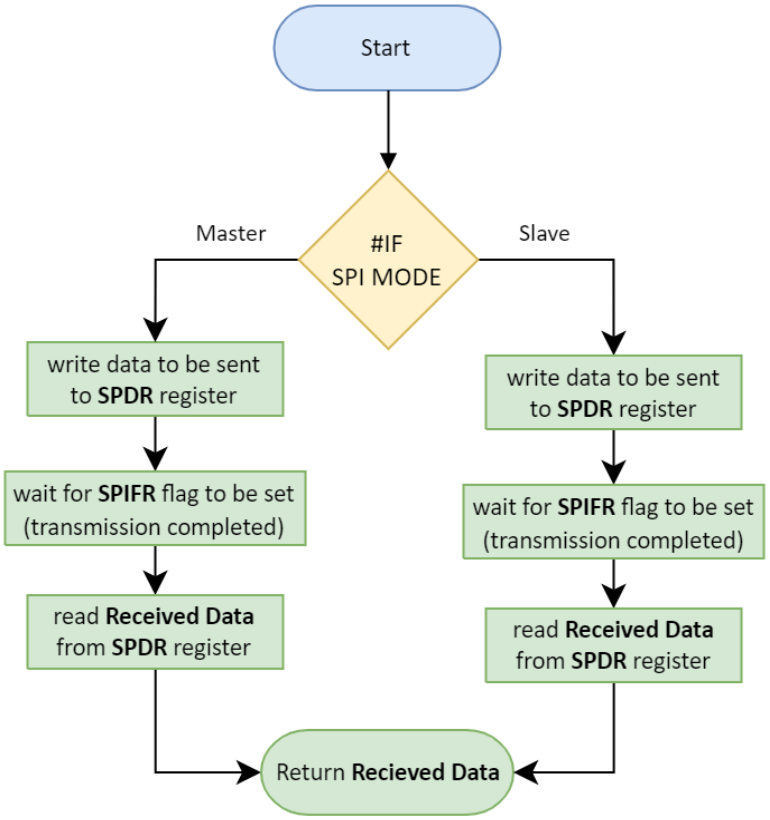


3.1.5. SPI Module

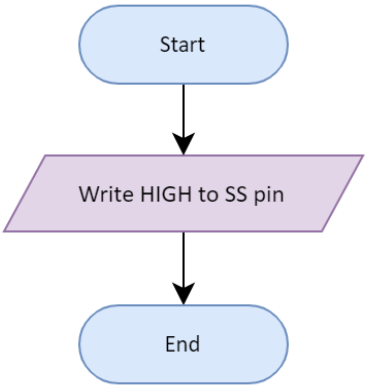
3.1.5.1. SPI_init



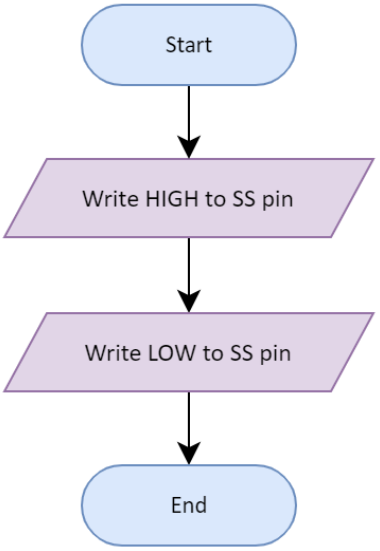
3.1.5.2. SPI_tranceiver



3.1.5.3. SPI_stop

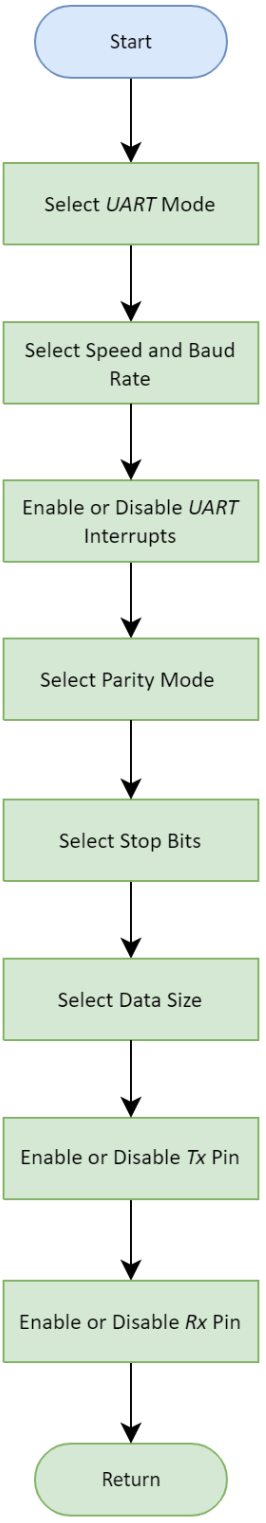


3.1.5.4. SPI_restart

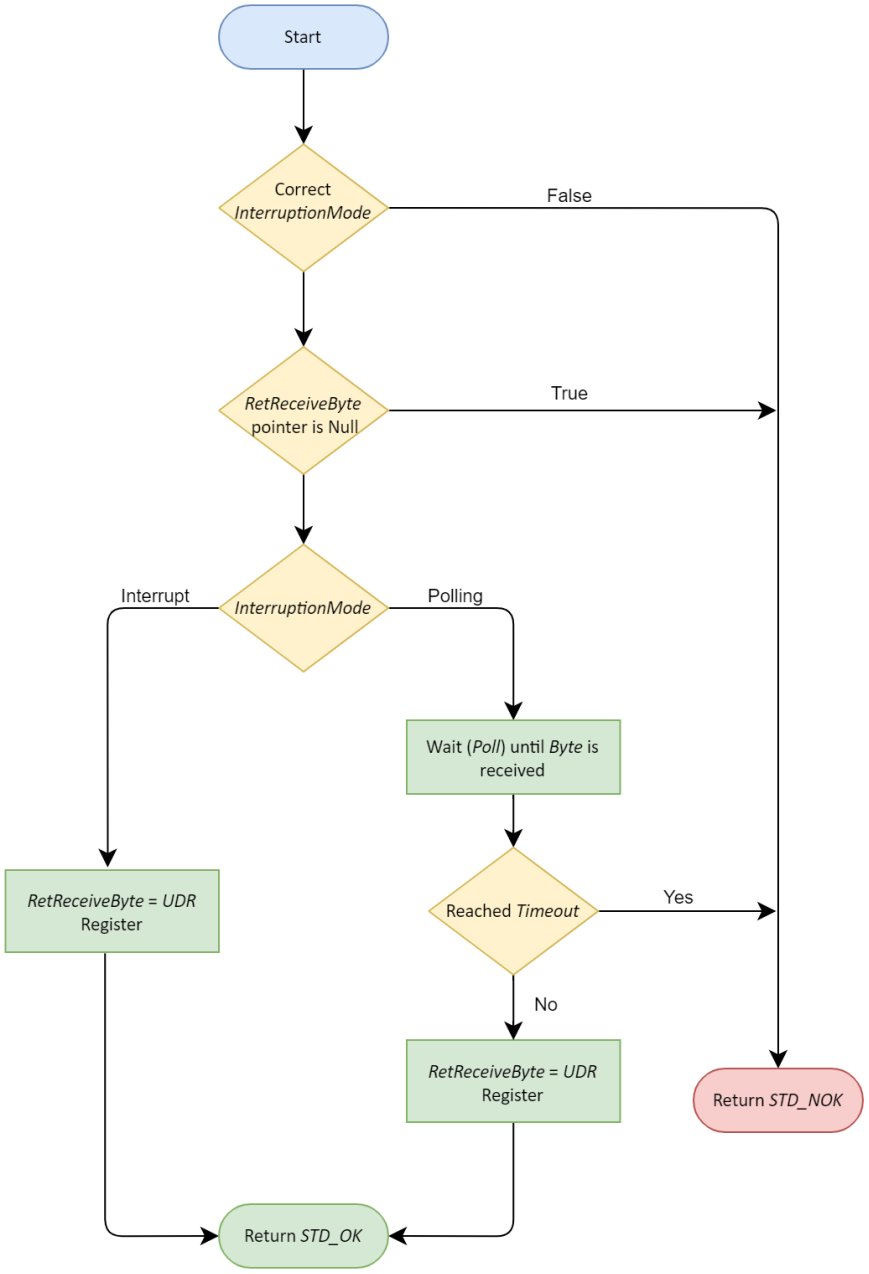


3.1.6. UART Module

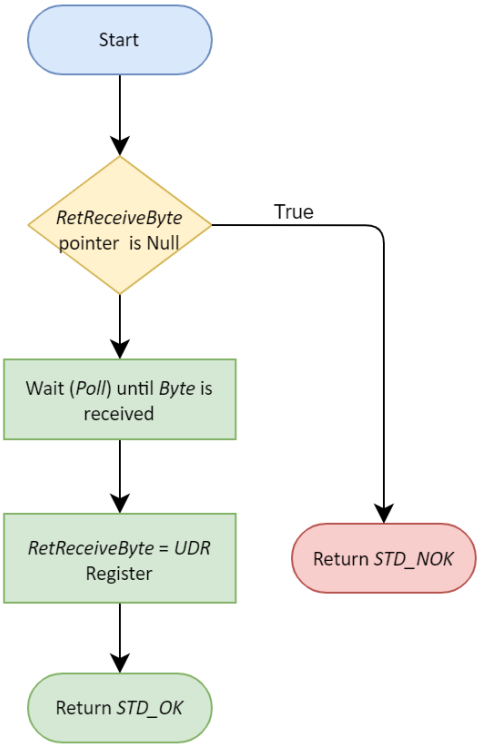
3.1.6.1. UART_initialization



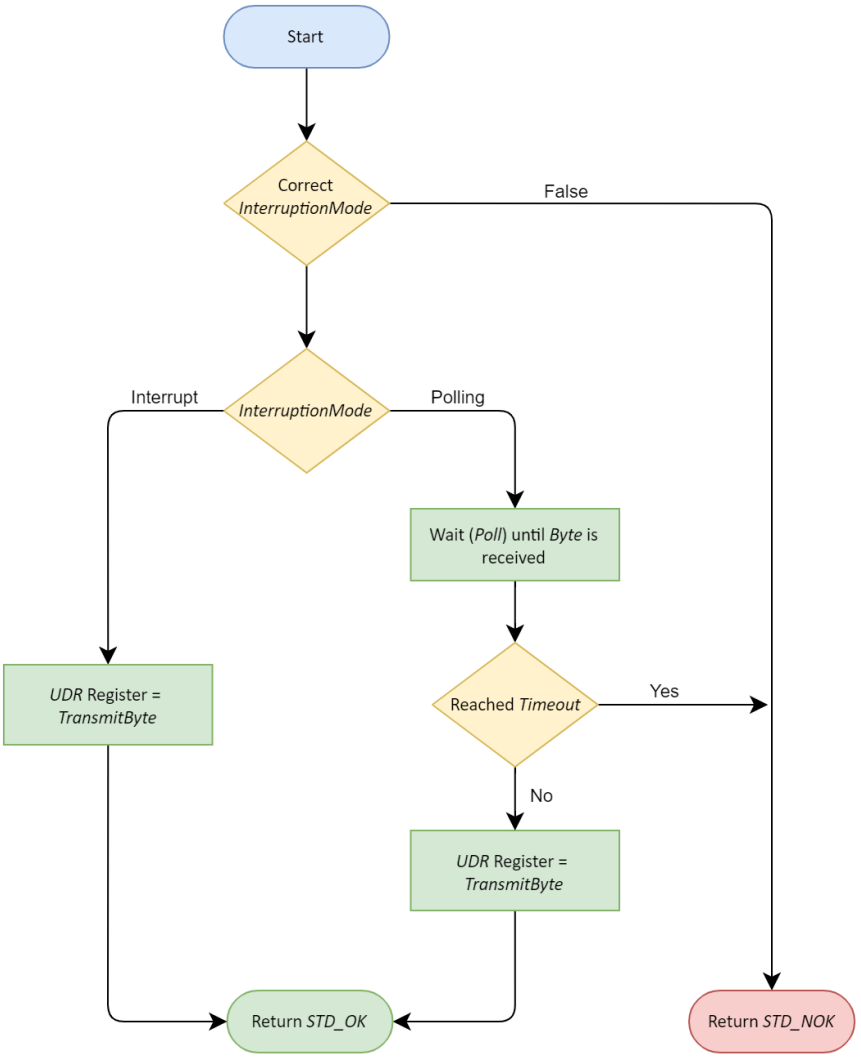
3.1.6.2. UART_receiveByte



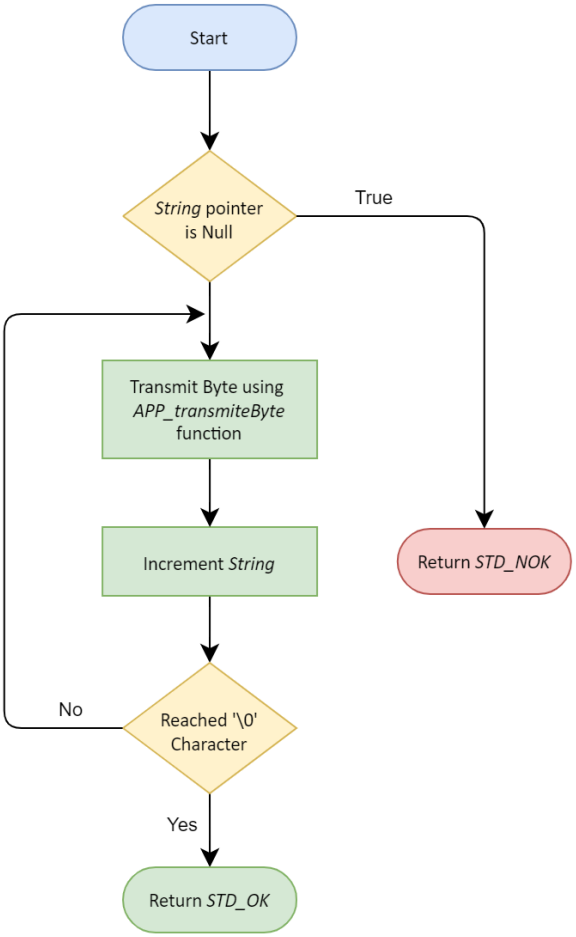
3.1.6.3. UART_receiveByteBlock



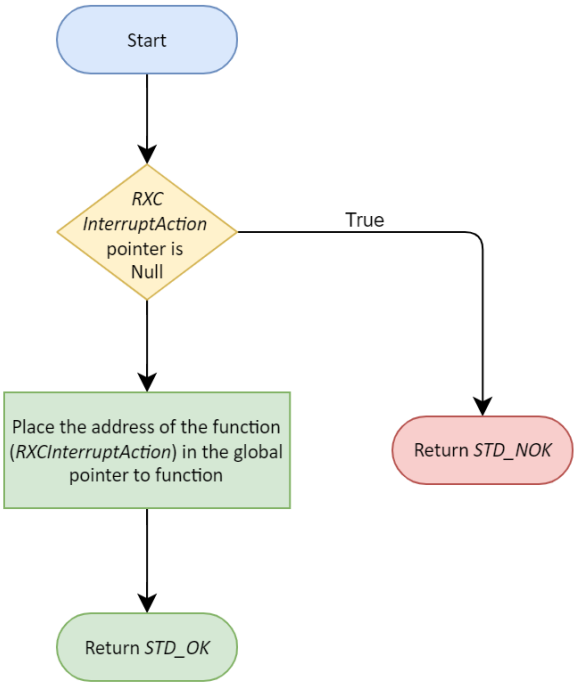
3.1.6.4. UART_transmitByte



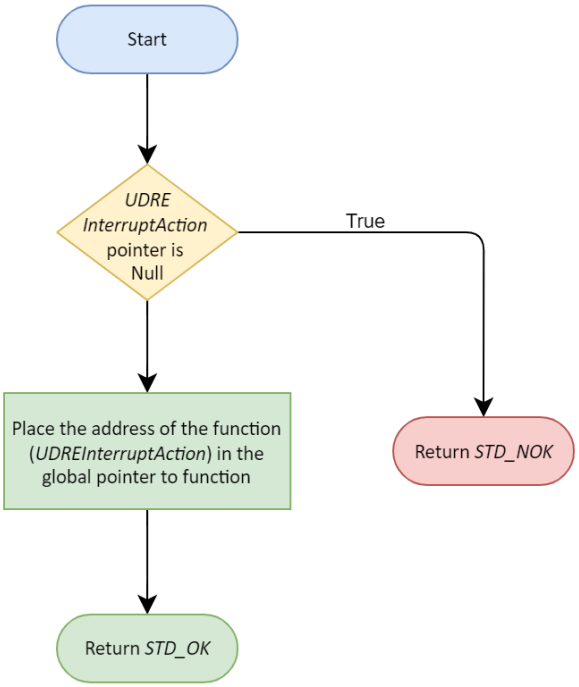
3.1.6.5. UART_transmitString



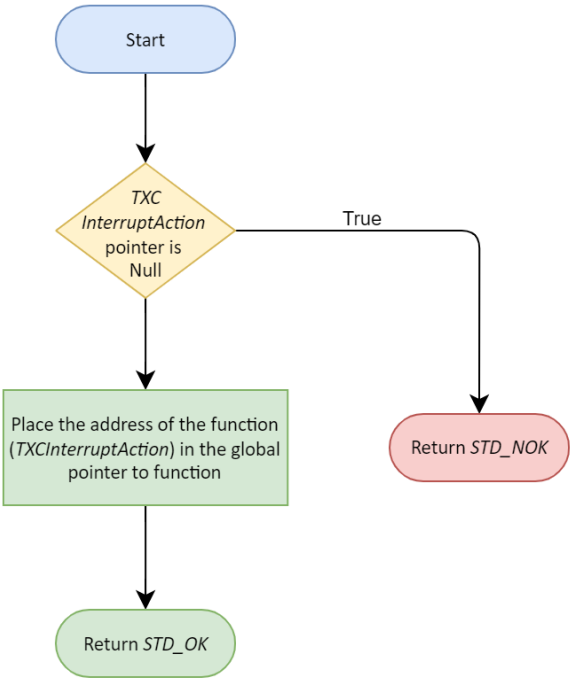
3.1.6.6. UART_RXCSetCallBack



3.1.6.7. UART_UDRESetCallBack



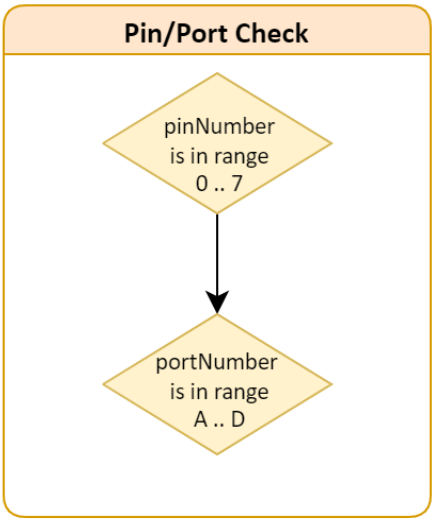
3.1.6.8. UART_TXCSetCallBack



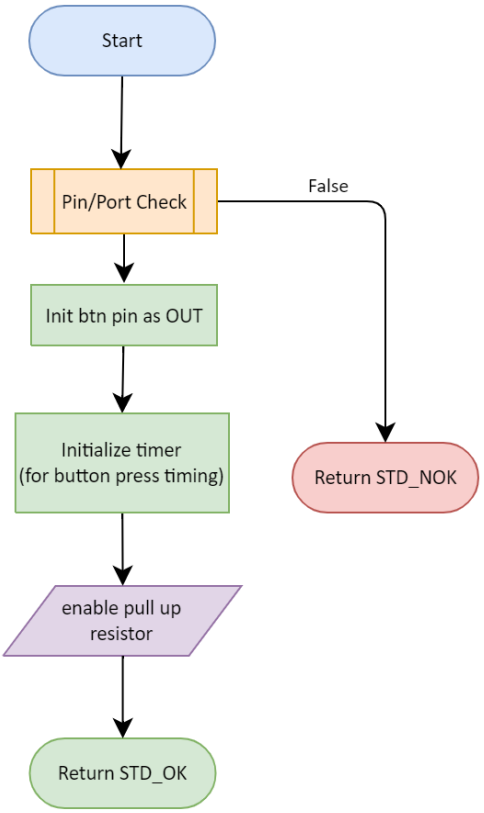
3.2. HAL Layer

3.2.1. MBTN Module

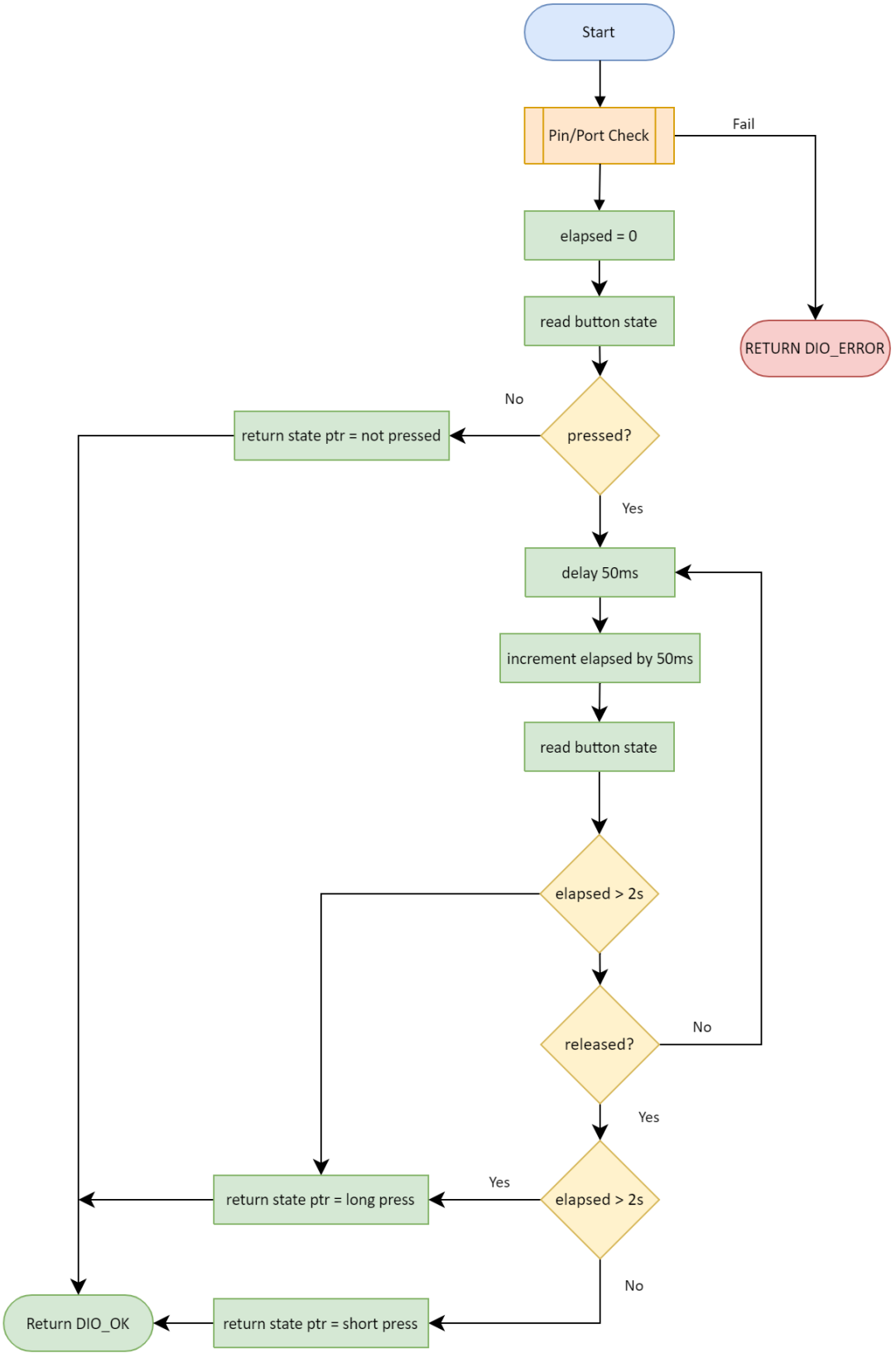
3.2.1.a. Pin/Port Check Sub-process



3.2.1.1. MBTN_init

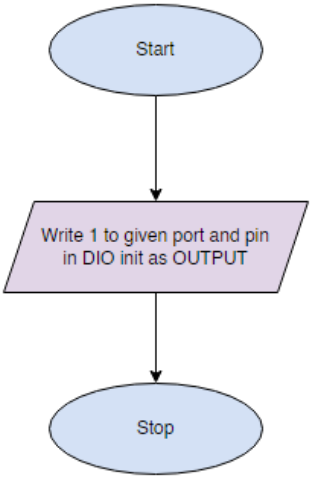


3.2.1.2. MBTN_getBTNState

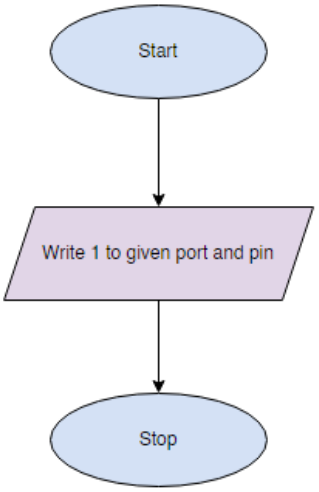


3.2.2. Buzzer Module

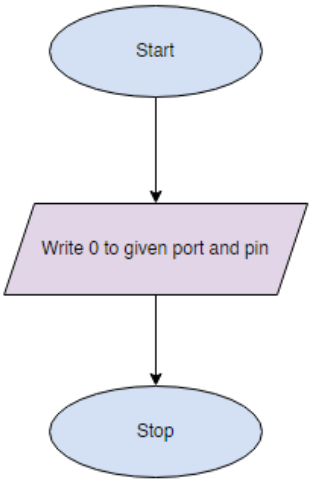
3.2.2.1. BUZZER_init



3.2.2.2. BUZZER_on

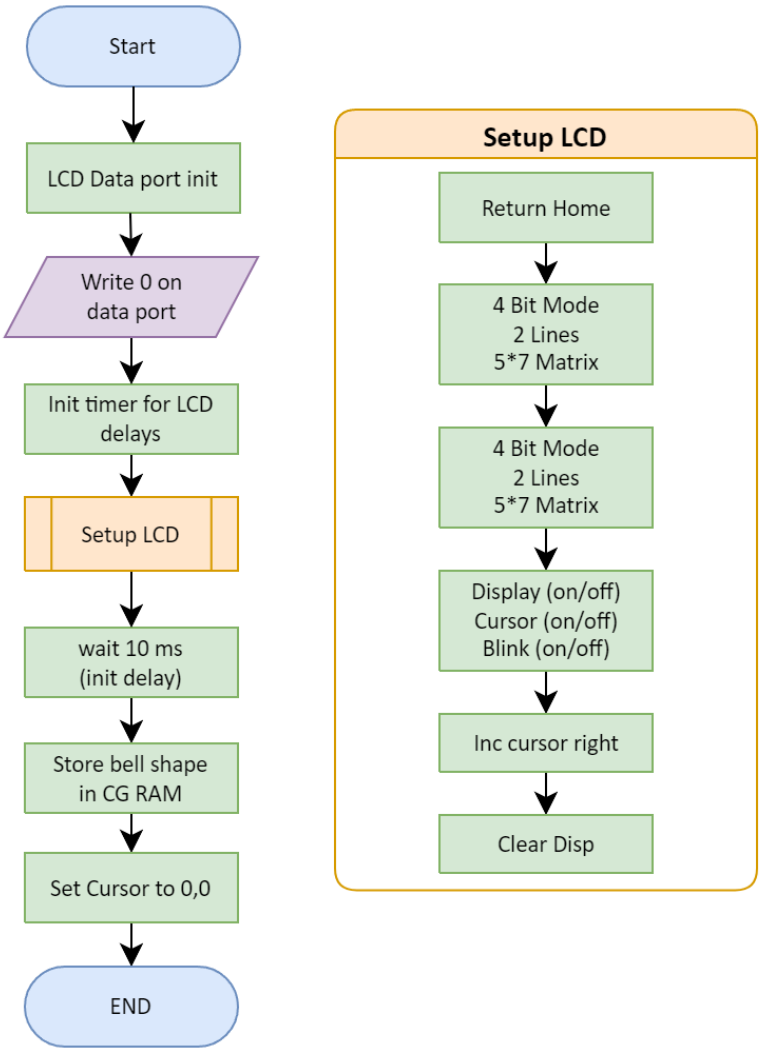


3.2.2.3. BUZZER_off

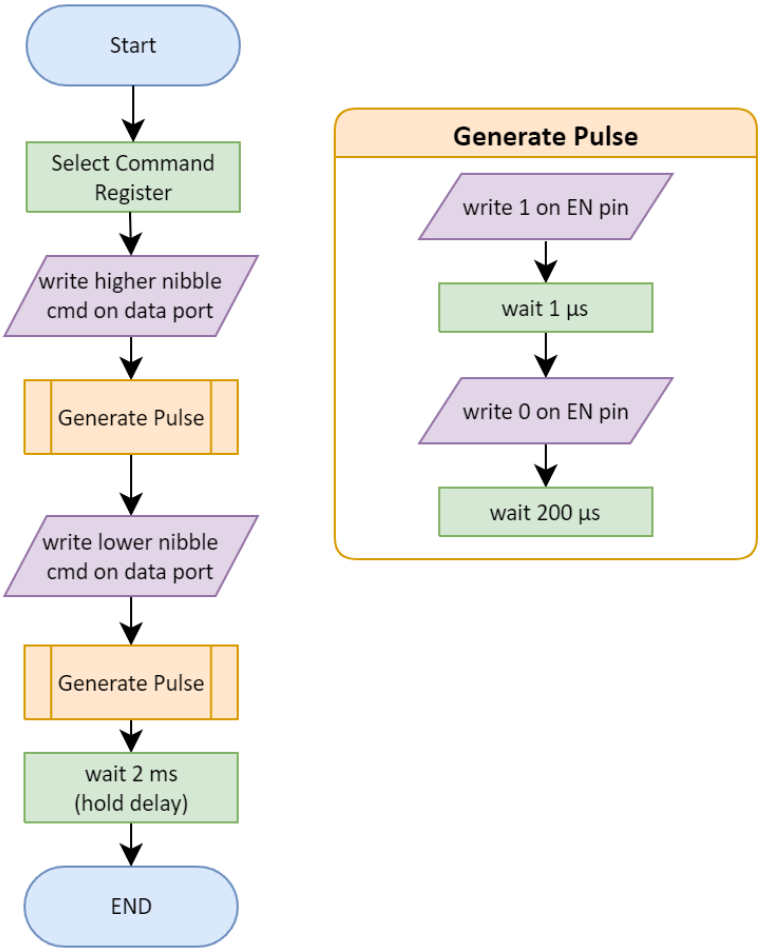


3.2.3. LCD Module

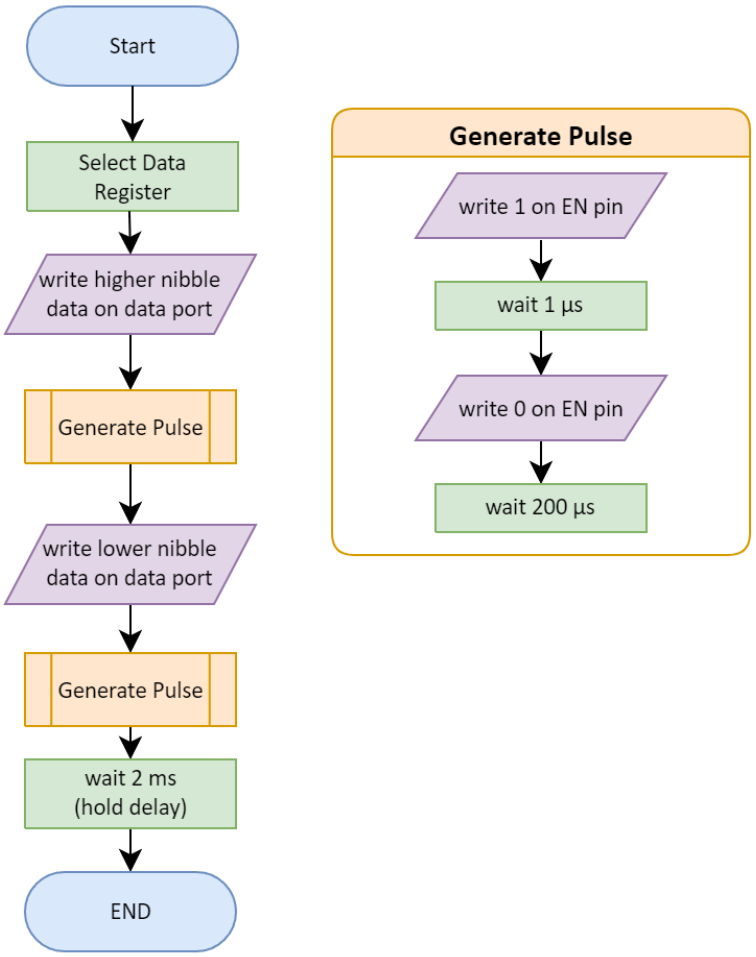
3.2.3.1. LCD_init



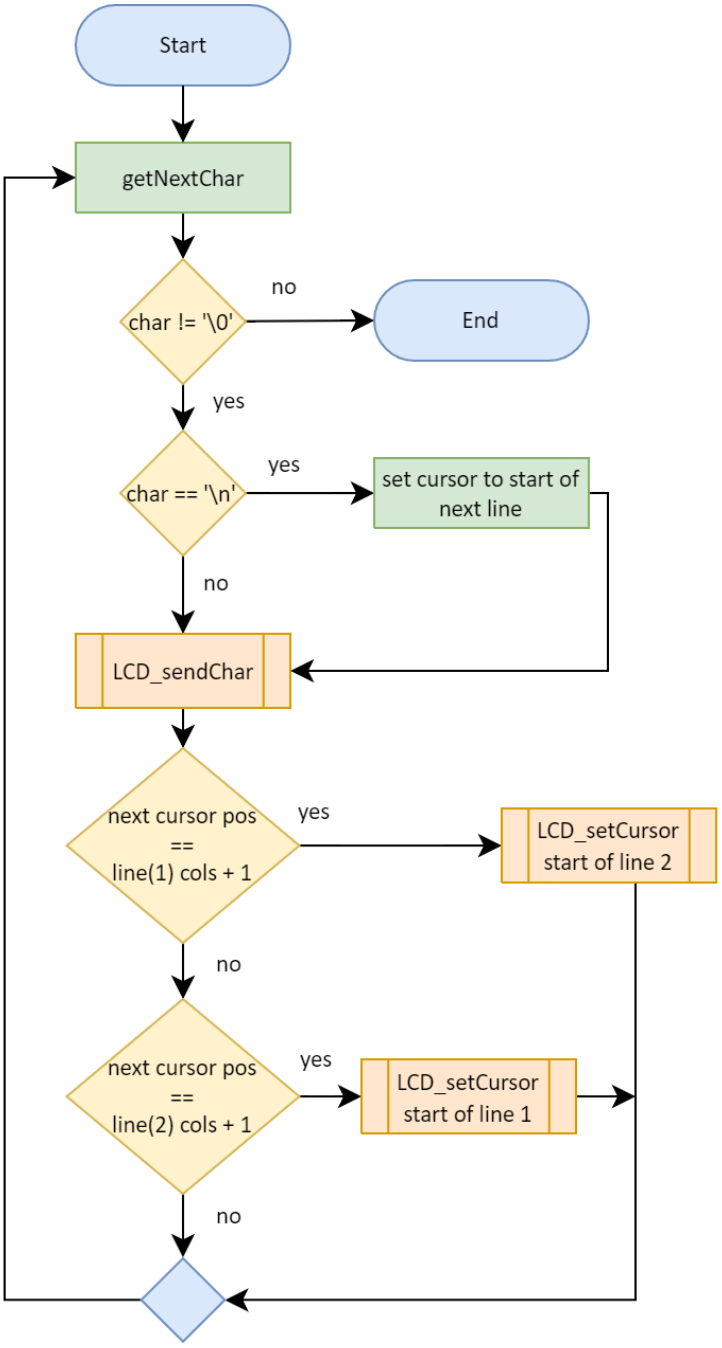
3.2.3.2. LCD_sendCommand



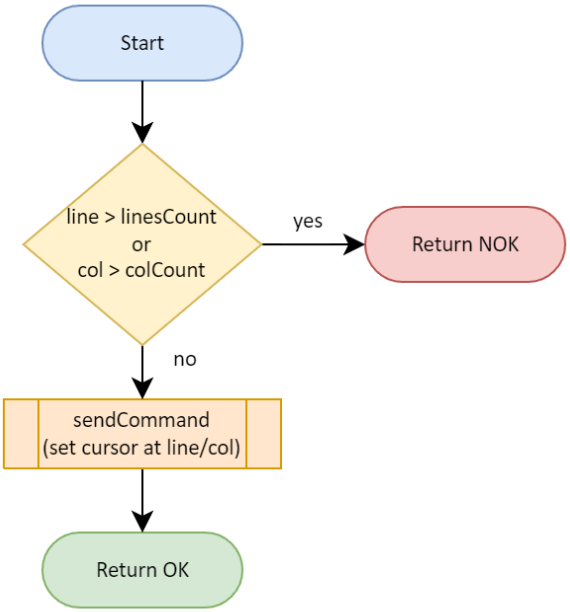
3.2.3.3. LCD_LCD_sendChar



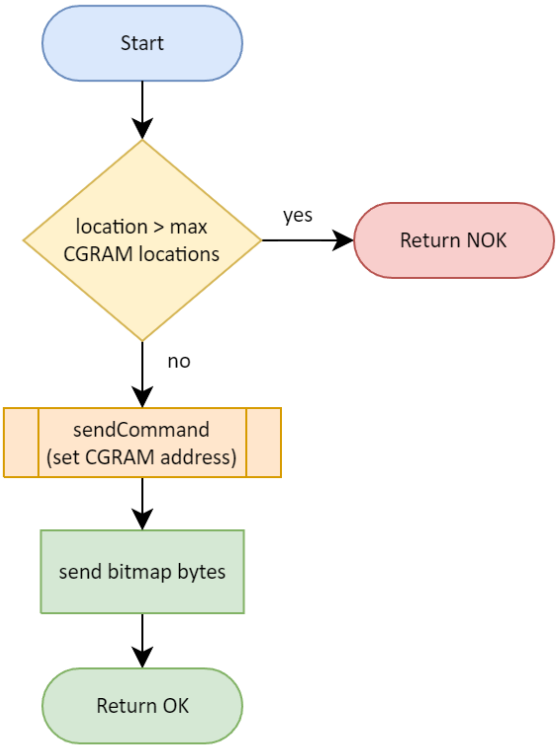
3.2.3.4. LCD_sendString



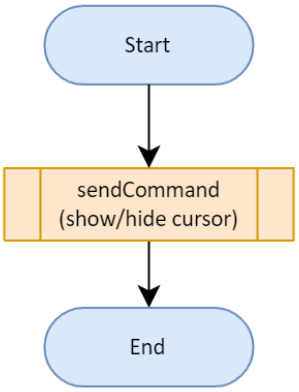
3.2.3.5. LCD_setCursor



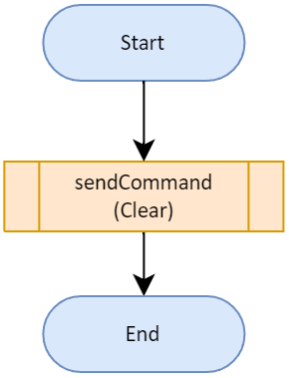
3.2.3.6. LCD_storeCustomCharacter



3.2.3.7. LCD_changeCursor

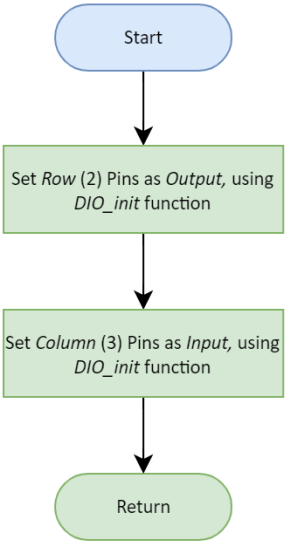


3.2.3.8. LCD_clear

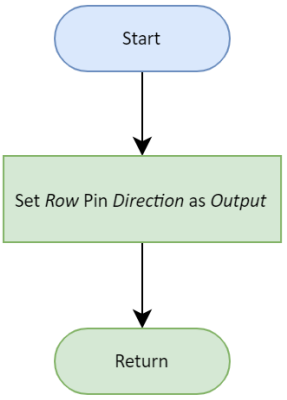


3.2.4. KPD Module

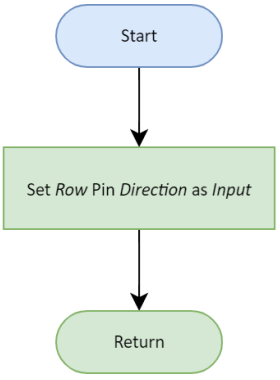
3.2.4.1. KPD_initKPD



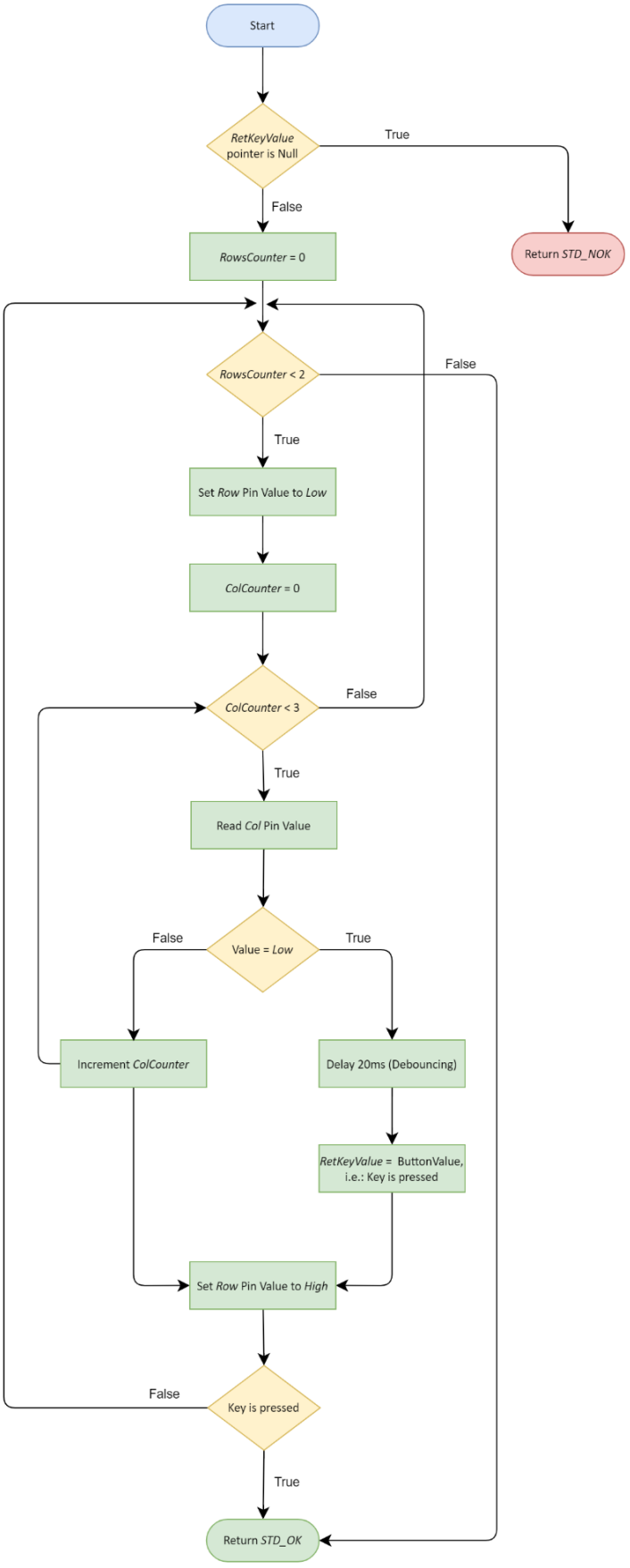
3.2.4.2. KPD_enableKPD



3.2.4.3. KPD_disableKPD

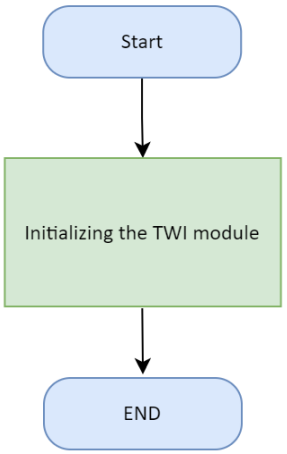


3.2.4.4. KPD_getPressedKey

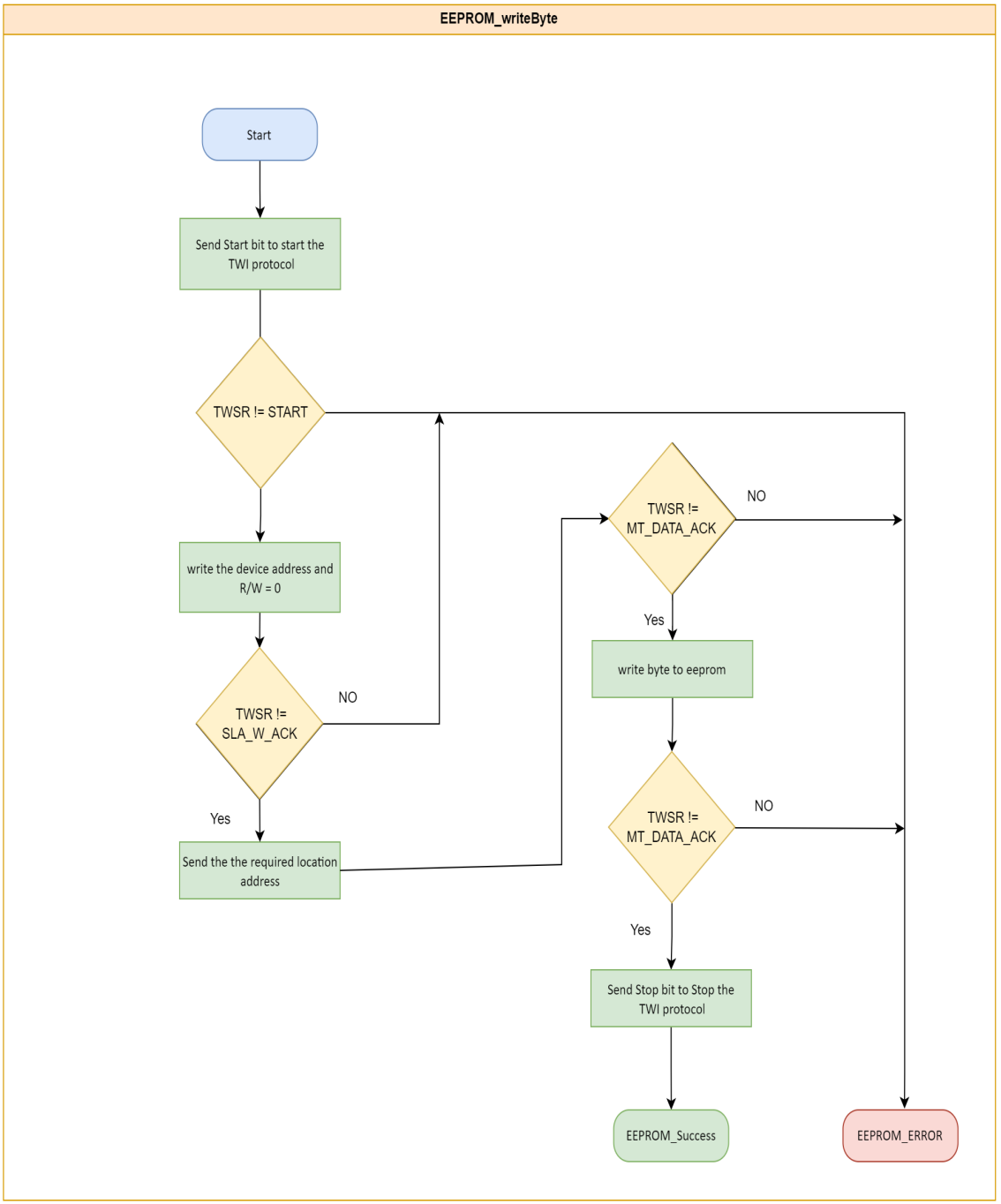


3.2.5. EEPROM Module

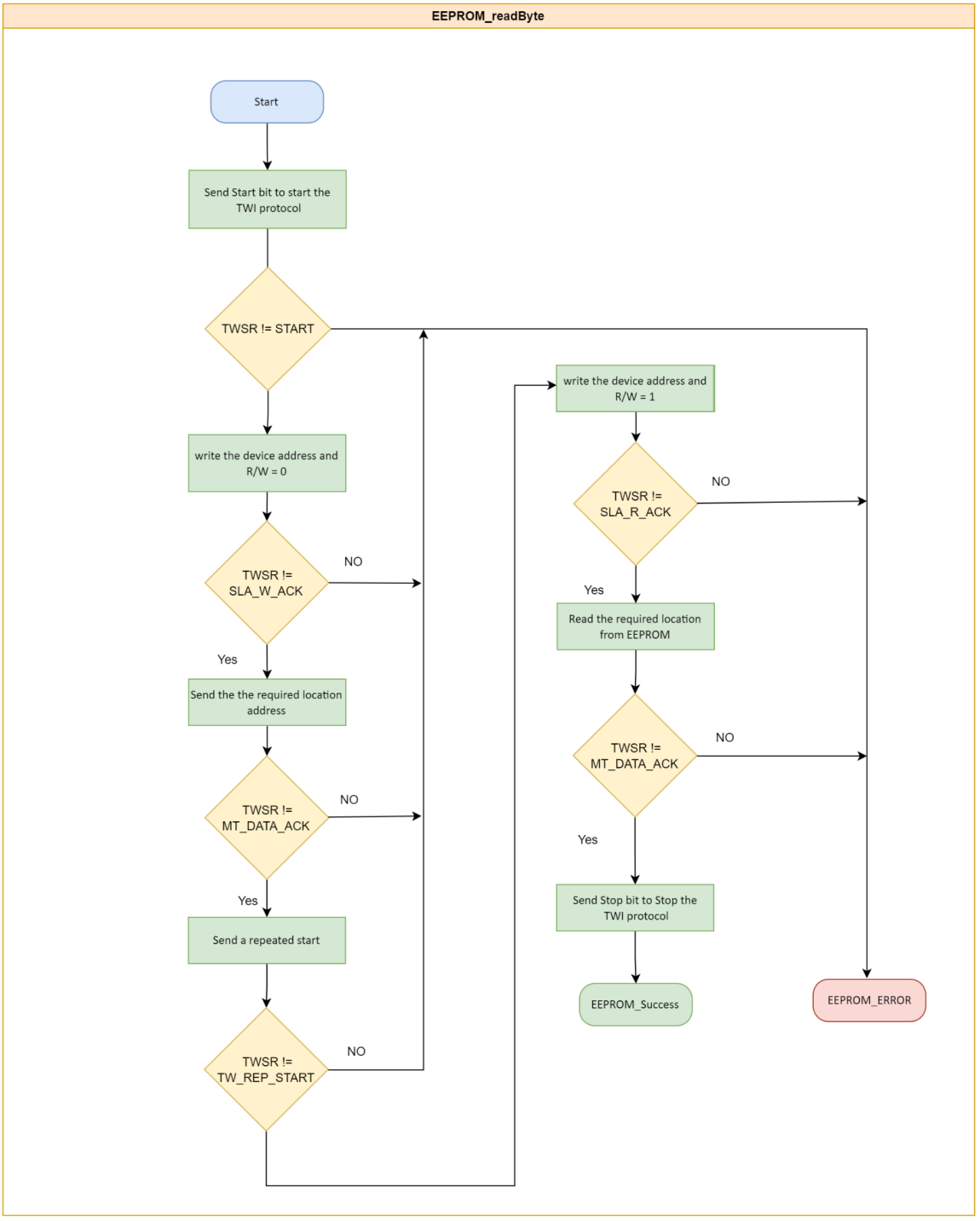
3.2.5.1 EEPROM_initialization



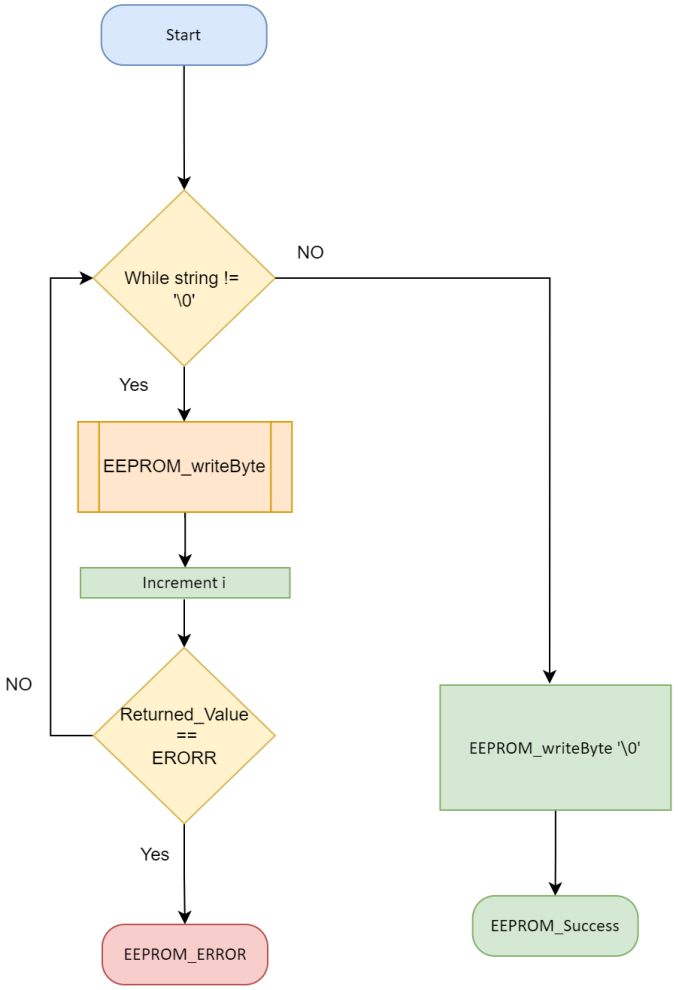
3.2.5.2 EEPROM_writeByte



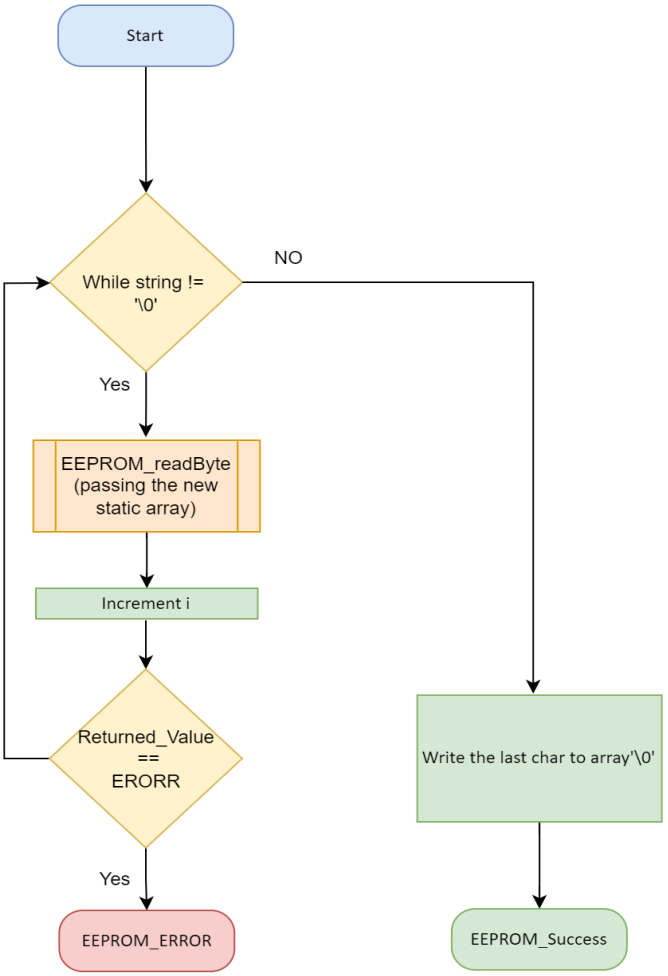
3.2.5.3 EEPROM_readByte



3.2.5.4 EEPROM_writeArray

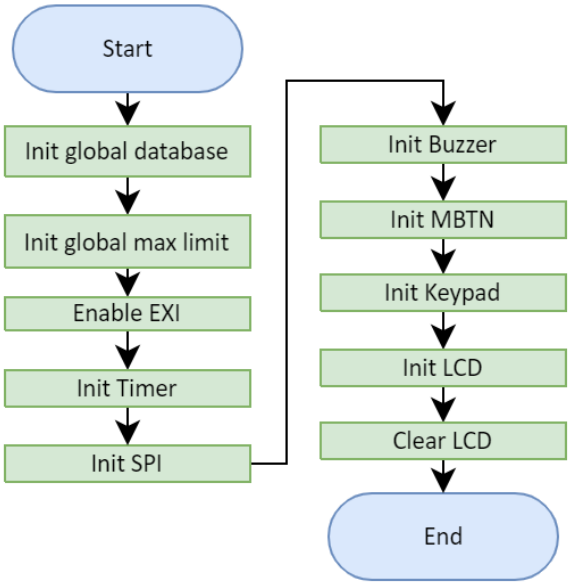


3.2.5.5 EEPROM_readArray

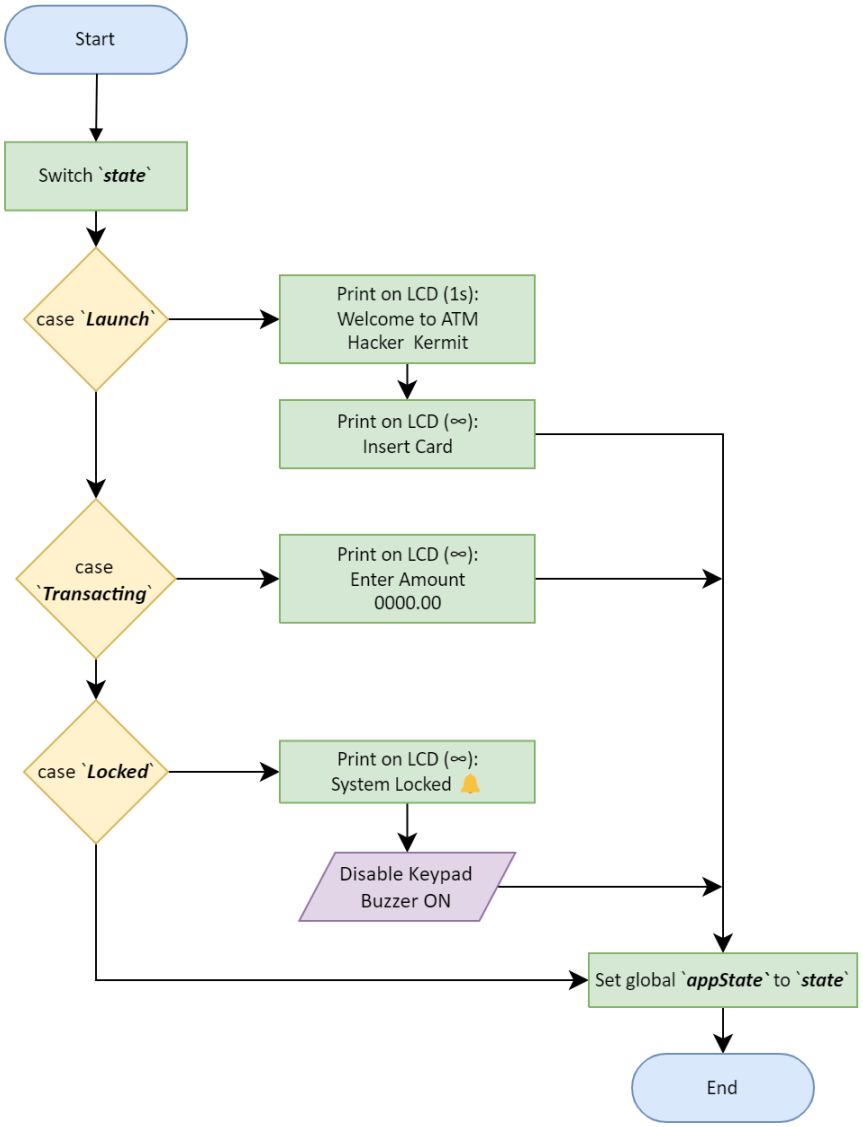


3.3. ATM APP Layer

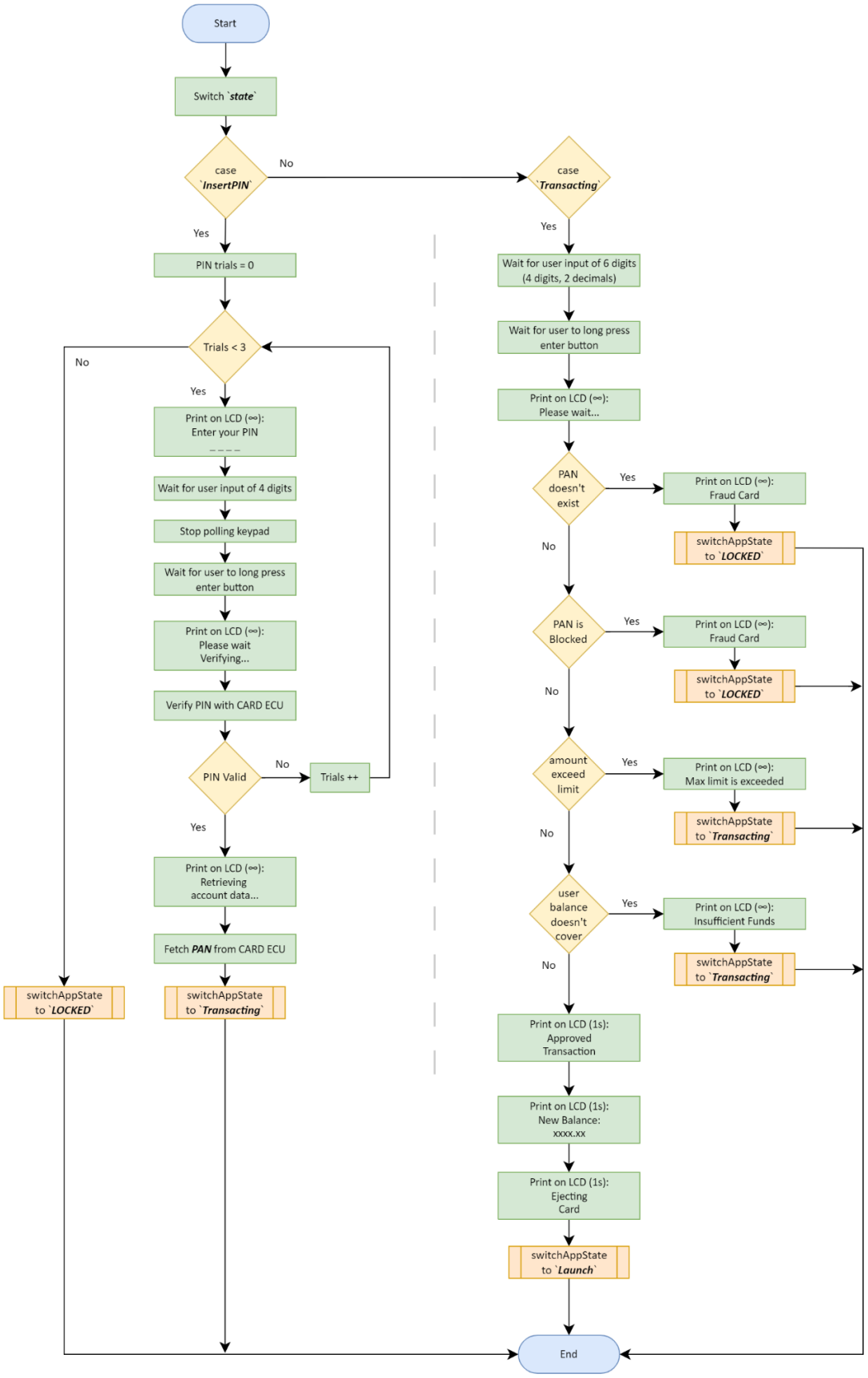
3.3.1. APP_initialization



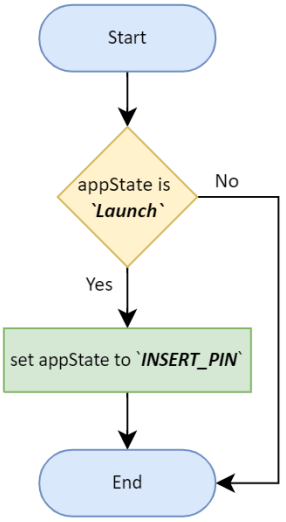
3.3.2. APP_switchState



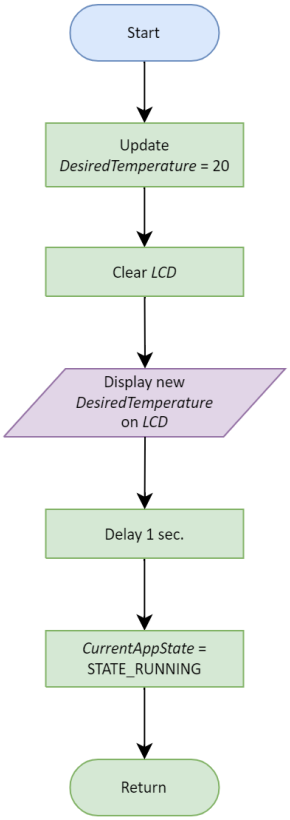
3.3.3. APP_startProgram (for HQ click me)



3.3.4. APP_trigger

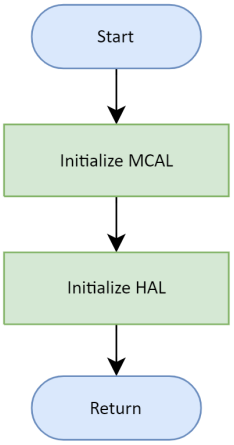


3.3.5. APP_resetToDefault

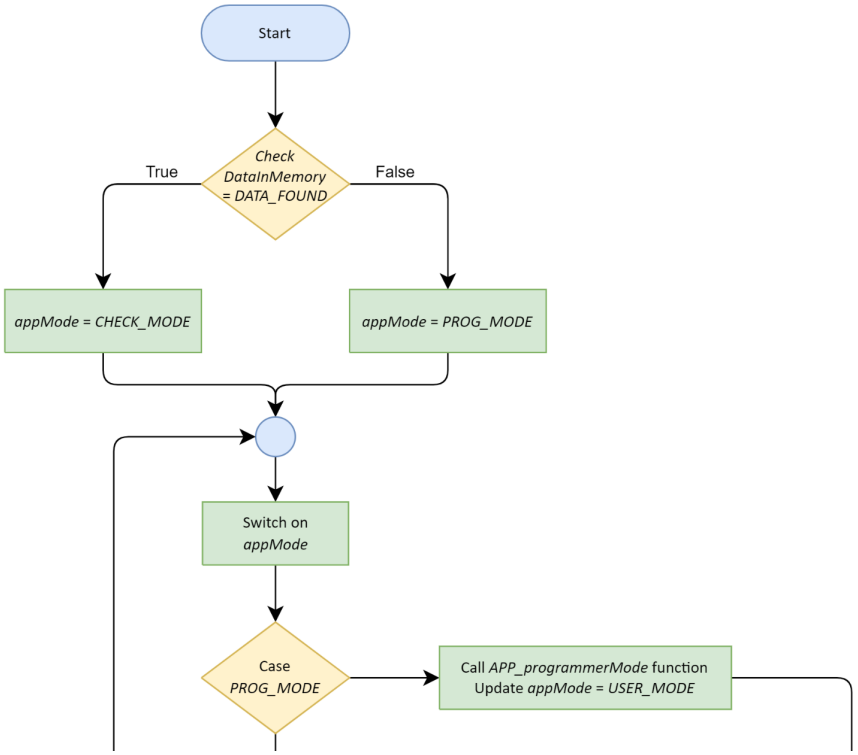


3.4. CARD APP Layer

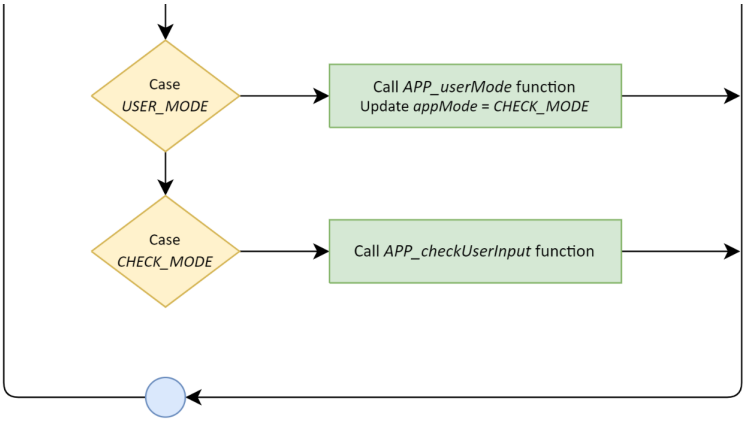
3.4.1. APP_initialization



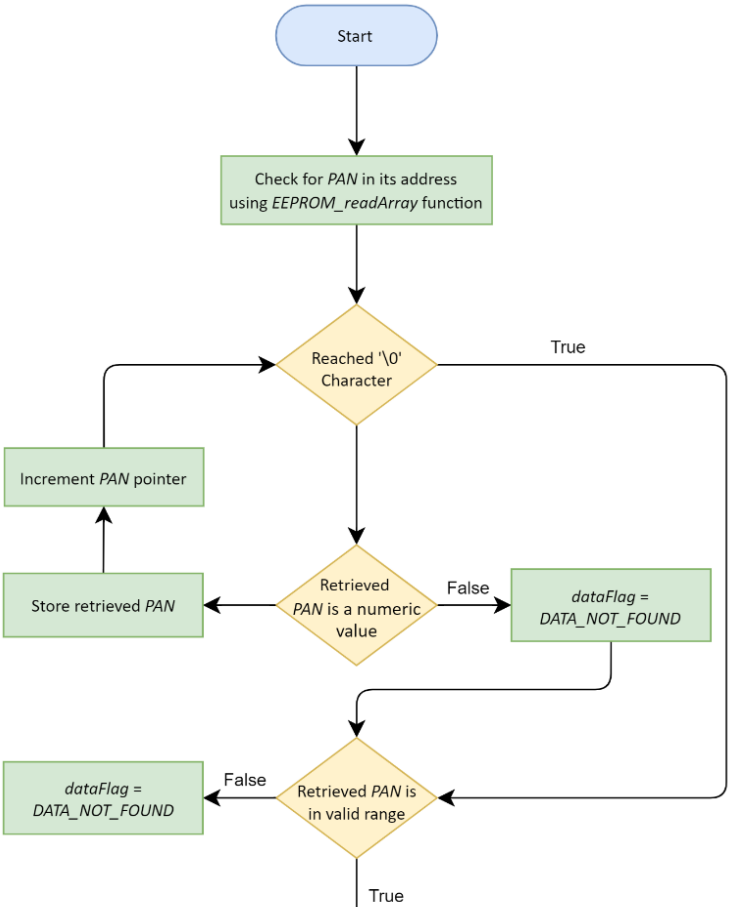
3.4.2. APP_startProgram



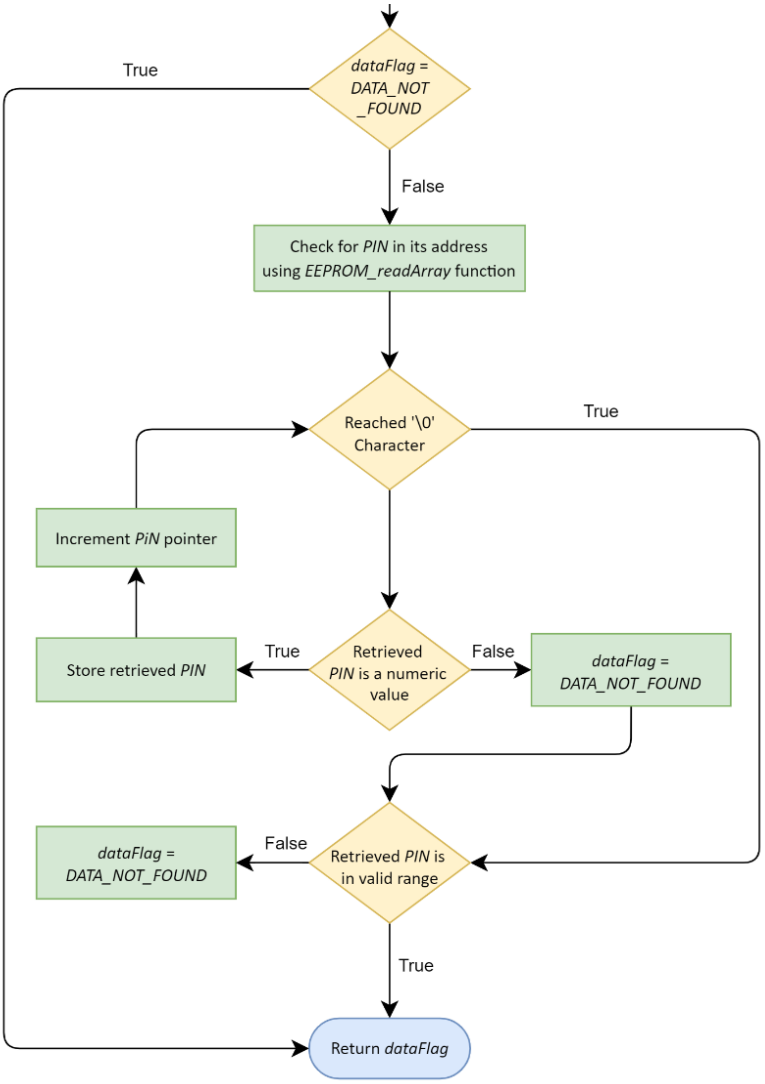
Continue the flowchart on the next page



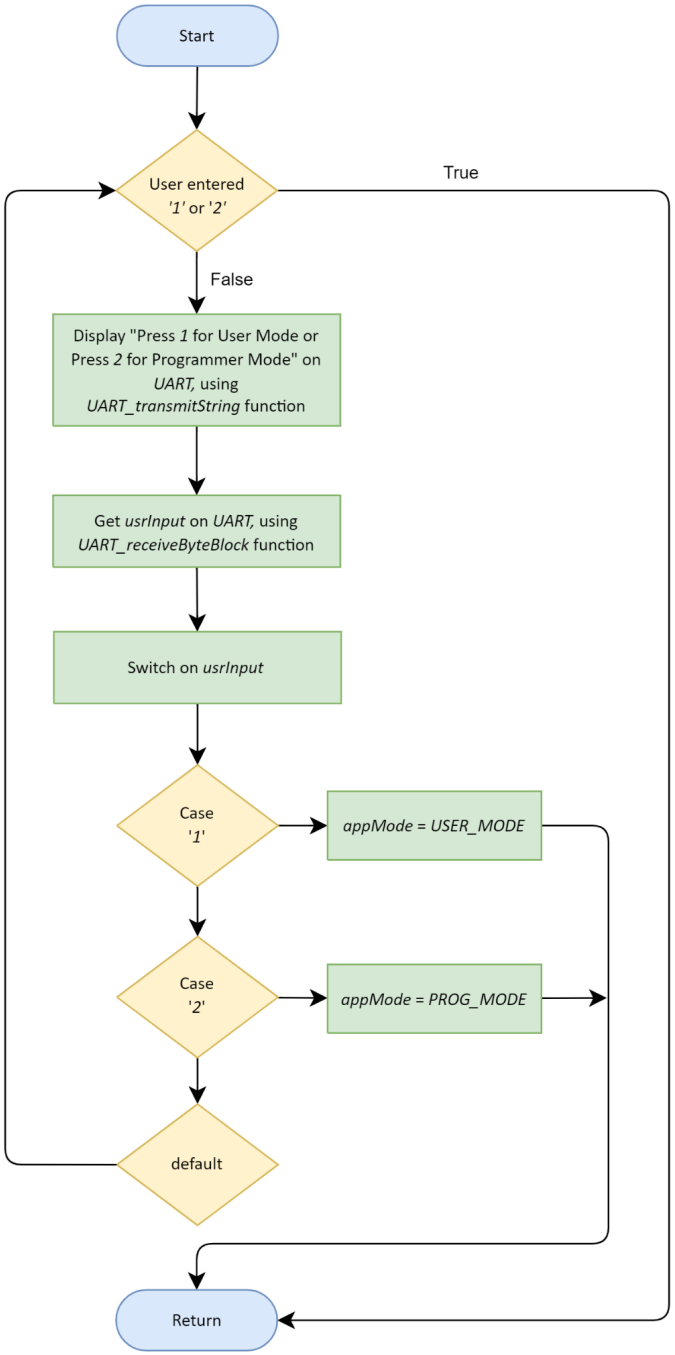
3.4.3. APP_checkDataInMemory



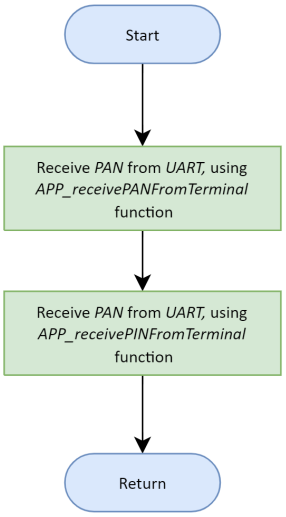
Continue the flowchart on the next page


dataFlag = DATA_NOT _FOUND
True
False
Check for PIN in its address using EEPROM_readArray function
Reached '\0' Character
True
Increment PiN pointer
Store retrieved PIN
True
Retrieved PIN is a numeric value
False
dataFlag = DATA_NOT_FOUND
dataFlag = DATA_NOT_FOUND
False
Retrieved PIN is in valid range
True
Return dataFlag

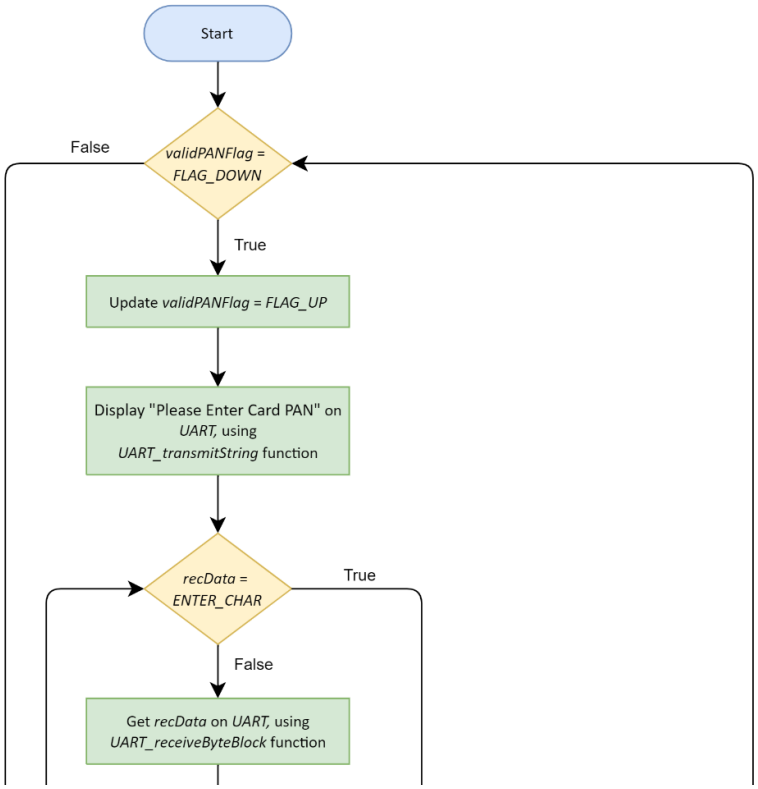
3.4.4. APP_checkUserInput



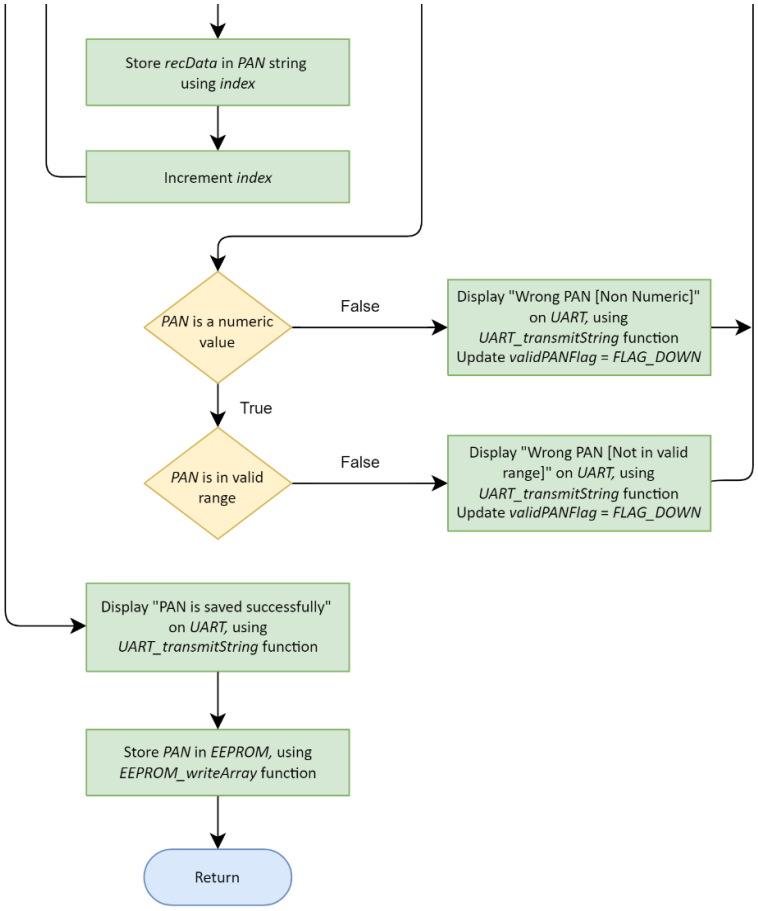
3.4.5. APP_programmerMode



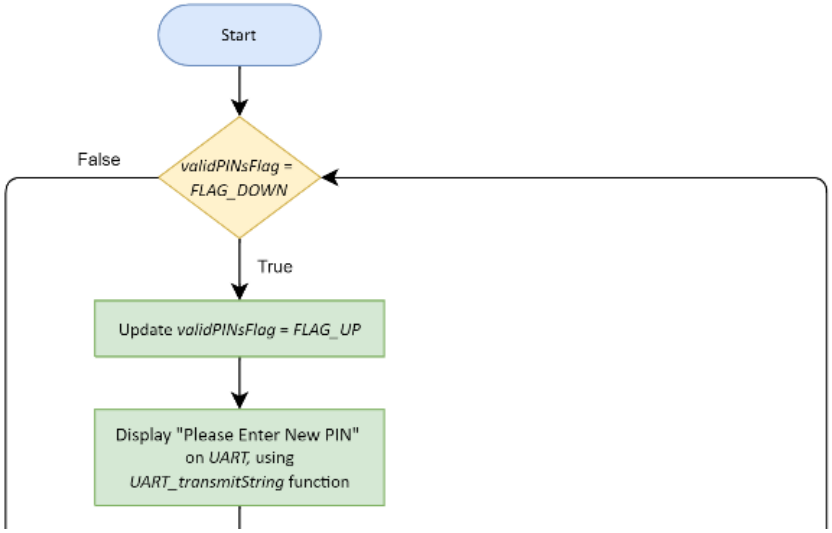
3.4.6. APP_receivePANFromTerminal



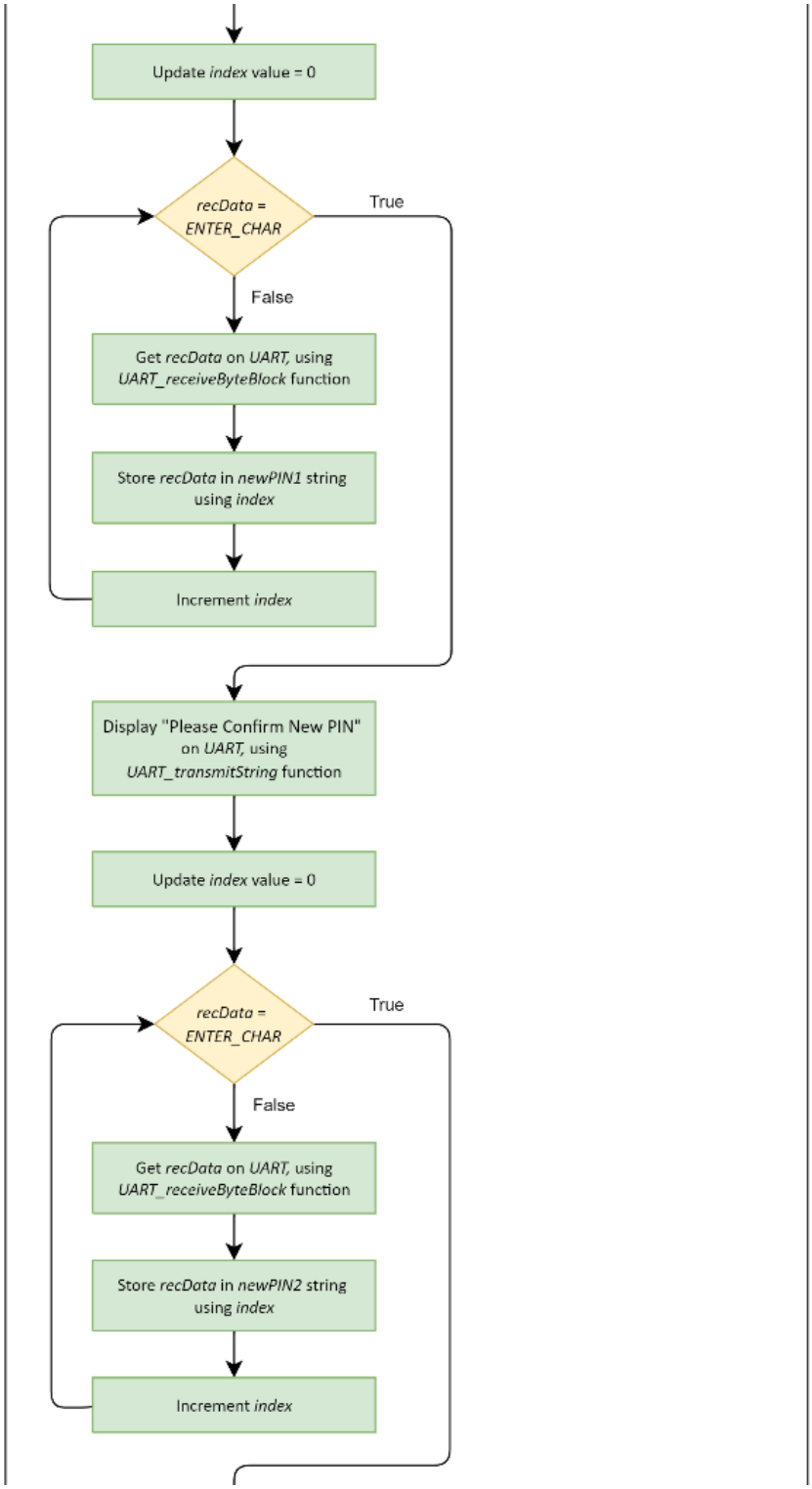
Continue the flowchart on the next page



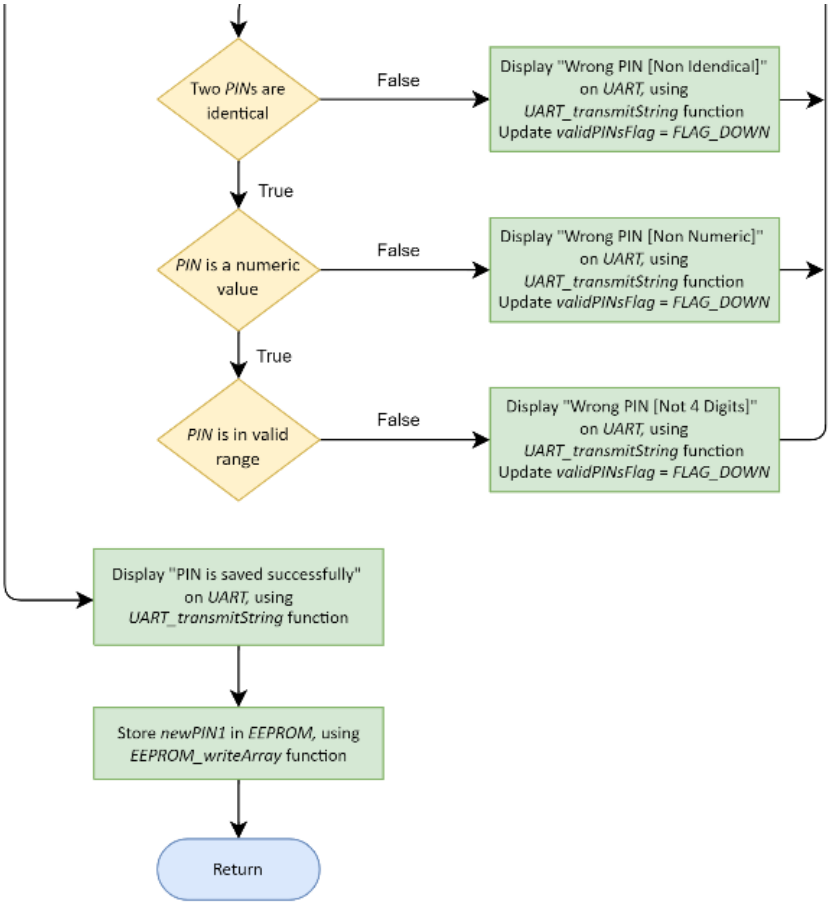
3.4.7. APP_receivePINFromTerminal



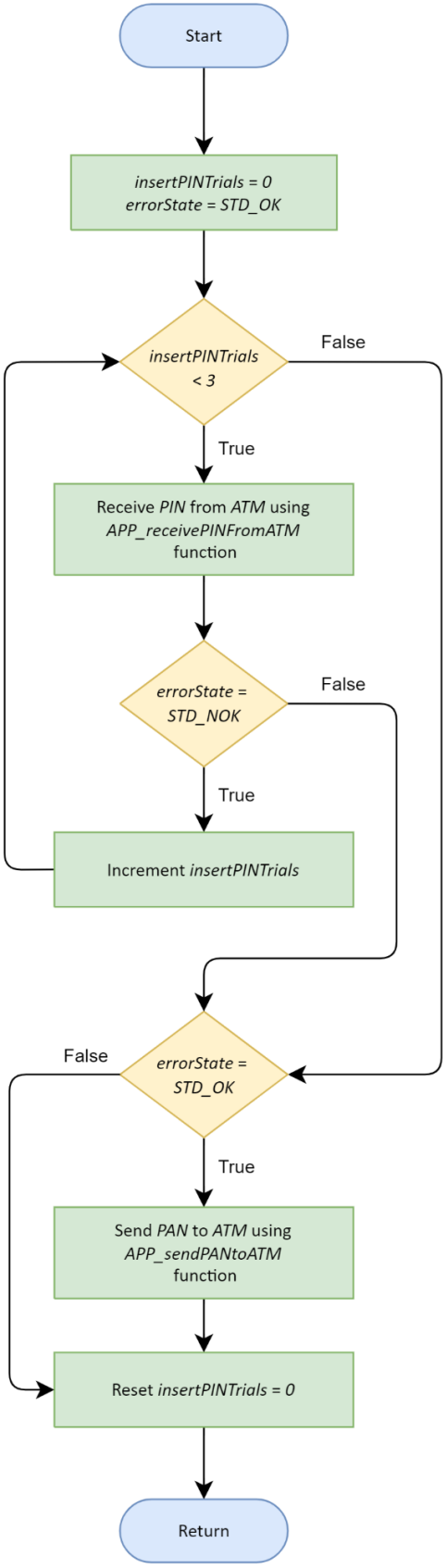
Continue the flowchart on the next page



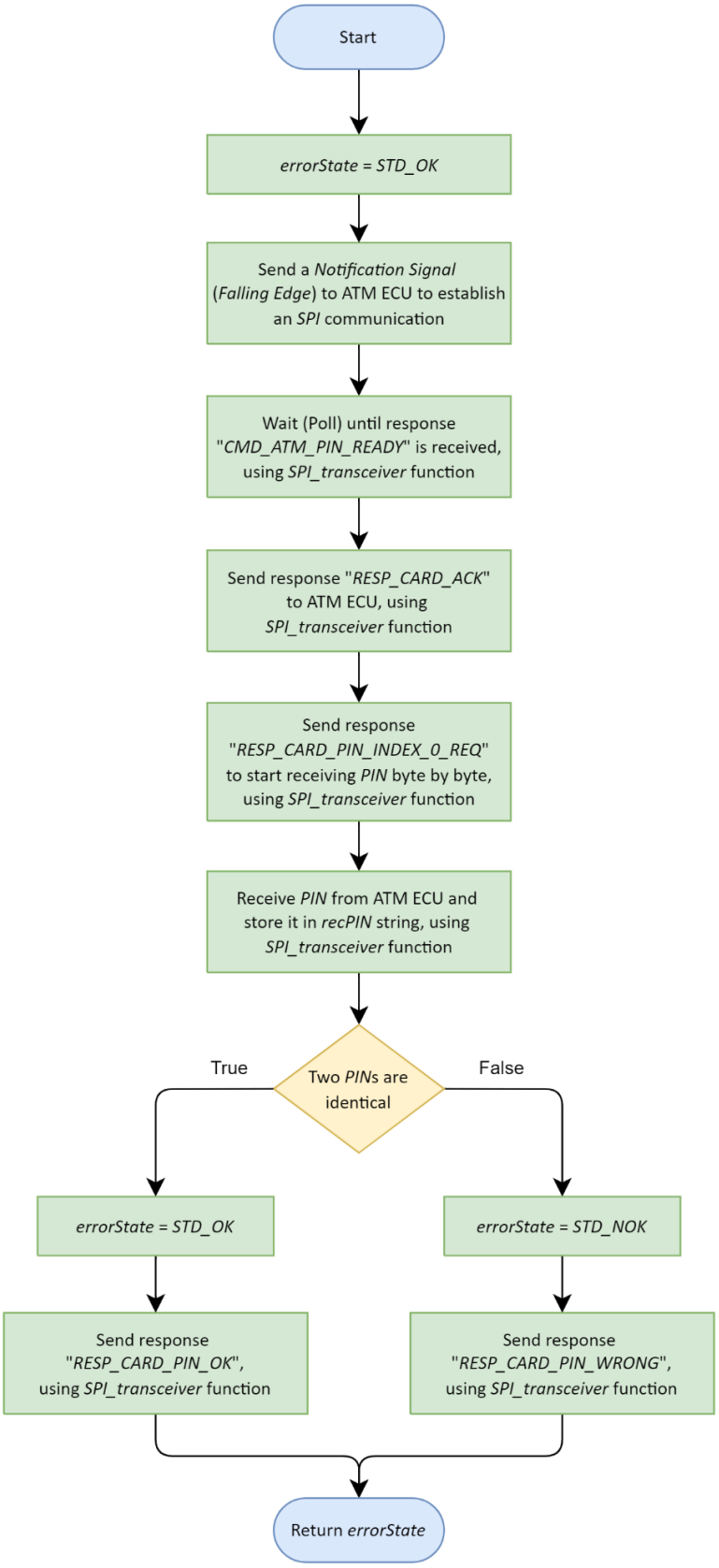
Continue the flowchart on the next page


Two PINs are identical
False
Display "Wrong PIN [Non Idendical]" on UART, using UART_transmitString function
Update validPINsFlag = FLAG_DOWN
True
PIN is a numeric value
False
Display "Wrong PIN [Non Numeric]" on UART, using UART_transmitString function
Update validPINsFlag = FLAG_DOWN
True
PIN is in valid range
False
Display "Wrong PIN [Not 4 Digits]" on UART, using UART_transmitString function
Update validPINsFlag = FLAG_DOWN
Display "PIN is saved successfully" on UART, using UART_transmitString function
Store newPIN1 in EEPROM, using EEPROM_writeArray function
Return

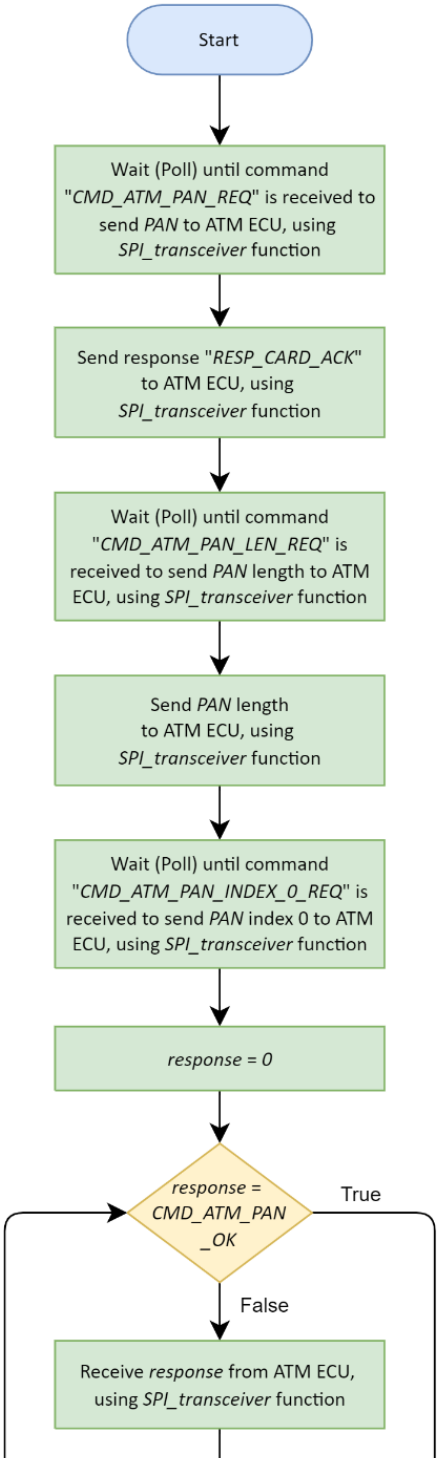
3.4.8. APP_userMode



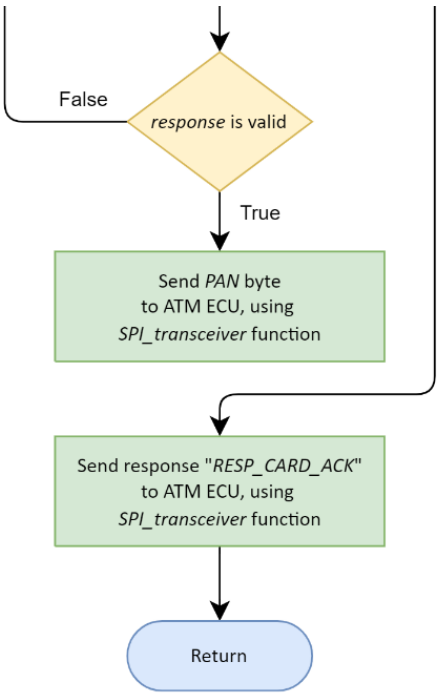
3.4.9. APP_receivePINFromATM



3.4.10. APP_sendPANToATM



Continue the flowchart on the next page


False
response is valid
True
Send PAN byte
to ATM ECU, using
SPI_transceiver function
Send response "RESP_CARD_ACK"
to ATM ECU, using
SPI_transceiver function
Return

4. Issues that we faced during development

1. Pinpointing the Root Cause of SPI slave not sending the correct data

We had this issue that the SPI **slave** is set to echo what it receives from **master** however the echoed data wasn't correct, missing a bit / shifted right, tended out to be that the **master** shouldn't set the SS pin to high after every send as this resets the Slave circuitry and dismissed any data in there.

2. Hardware Issues

We faced multiple issues during testing our project on the hardware kits, the following is one of the issues that we ran through. One of the two kits didn't work at all. As well as no test program was burned/available on its MCU. Khazama AVR Programmer didn't recognize the MCU internal configurations nor it's signature to be a valid ATmega32 signature, we think there is a problem with the kit on-board programmer (i.e. ISP).

5. References

1. *Draw IO*
2. *Layered Architecture | Baeldung on Computer Science*
3. *Microcontroller Abstraction Layer (MCAL) | Renesas*
4. *Hardware Abstraction Layer - an overview | ScienceDirect Topics*
5. *What is a module in software, hardware and programming?*
6. *Embedded Basics – API's vs HAL's*
7. *Analog-to-Digital Converter - an overview | ScienceDirect Topics*
8. *Temperature Sensor Types | TE Connectivity*
9. *LM35 data sheet, product information and support | TI.com*
10. *Embedded System Keypad Programming - javatpoint*