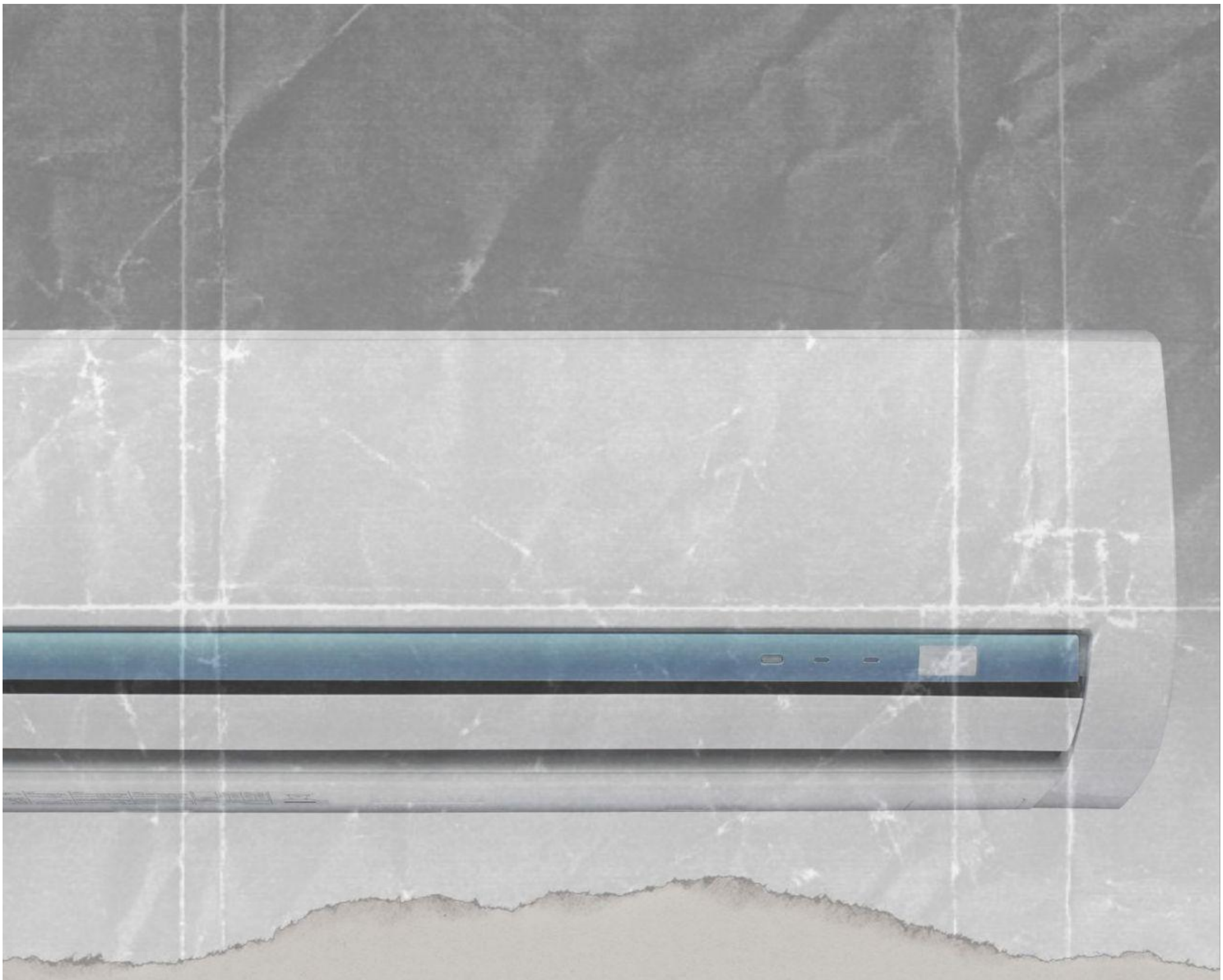


AC CONTROLLER

AUTHOR

TEAM 1 - HACKER KERMIT

MEMBERS

HOSSAM ELWAHSH

ABDELRHMAN WALAA

MAHMOUD MOWAFEY

TAREK GOHRY

AC Control Project

1. Project Introduction

This project involves developing software to control an AC unit.

1.1. Project Components

- LCD 2x16
- Temperature sensor LM35
- Buzzer
- Keypad 3x3 (5 buttons required)

1.2. Assumptions

- We assume that room temperature won't go below **10 °C**, or go higher than **99 °C**

1.3. System Requirements

1. **Constants:**
    a. Default temperature: 20 °C
    b. Minimum temperature: 18 °C
    c. Maximum temperature: 35 °C
2. **Flow:**
    a. Show "welcome" (1 second)
    b. Show "default temperature: 20 °C" (1 second)
    c. Goto Adjust Screen
3. **Adjust Screen**
    a. Show "Please choose required temperature" (0.5 second)
    b. Show **Adjust Screen model:**
    ```
    ====================
    Min: 18     20      Max: 35
    ||                          < progress bar to visualize temperature
    ====================
    ```
    c. Wait for user input to increase/decrease temperature
        i. Use keypad button 1 for increasing (updates temp & progress bar)

ii. Use keypad button 2 for decreasing (updates temp & progress bar)

iii. Use keypad button 3 for Set and Start AC

d. Timeout after 10 seconds if no input, set desired temperature to 20 °C

e. Otherwise if **Set** button was pressed, save the desired temperature in memory then go to **Running Screen**

4. **Running Screen**

a. Button 1,2,3 are disabled (increment/decrement/set) show an error if pressed for 0.5 second.

b. Buttons 4,5 are enabled (4: Adjust, 5: Reset)

c. **Running Screen:**

```
==================
                  🔔  < buzzer icon visible if current temp from sensor
Current Temp:   20        is greater than desired temperature
==================
```

d. Current temp is constantly updated from temperature sensor

e. If current temp. is greater than desired temp. Show buzzer icon and turn the buzzer on until temperature goes back down

f. If Button-4 (Adjust) was pressed, halt and go back to adjust screen to allow re-adjusting temperature

g. If Button-5 (Reset) was pressed, halt, reset desired temperature to default (20 °C, show “Reset to default temp: 20 °C” then resume back again to **Running Screen**

1.4. Extras

- We added that the buzzer will buzz for 1 second if the user tries to set the temperature below or above the min / max amounts allowed.
- We added an extra screen before adjust to show the controls used for adjusting e.g. 1 (+) 2(-) 3(set)
- Instead of showing an error if (set/inc/dec) were pressed during running mode, we completely disabled these buttons. Similarly buttons (adjust/reset) are also disabled in Adjust mode.

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

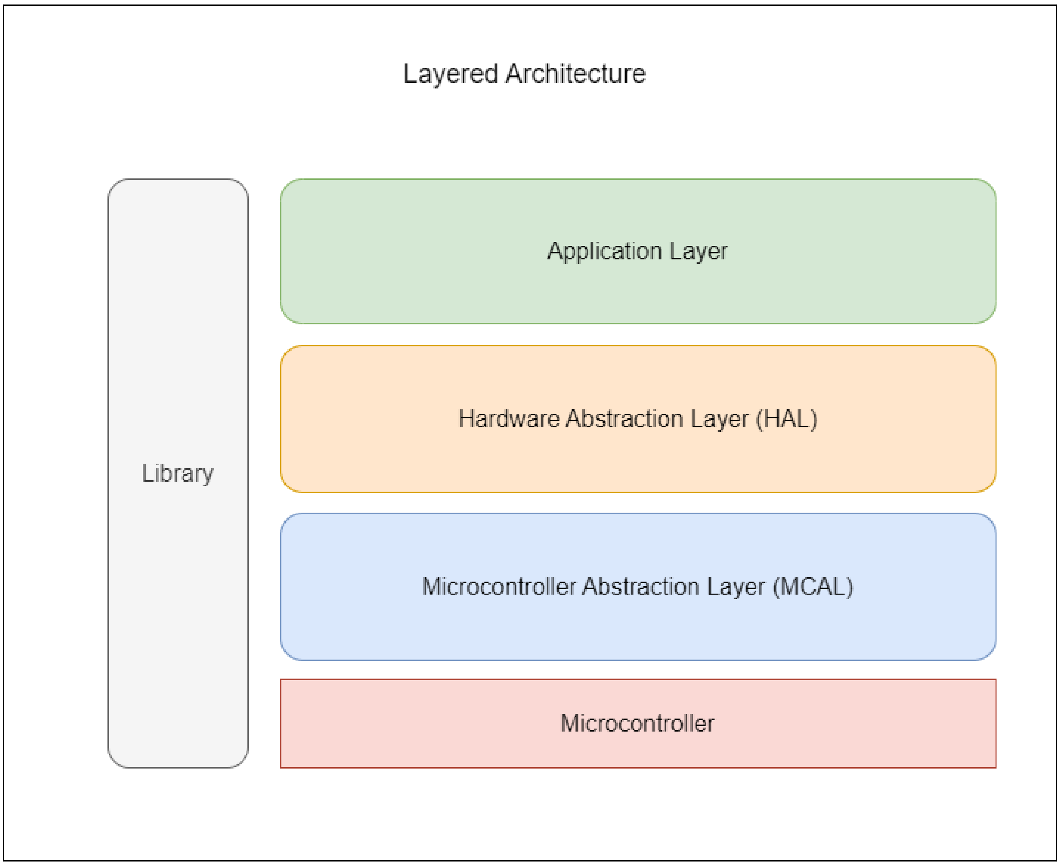


Figure 1. Layered Architecture Design

2.1.3. Project Circuit Schematic

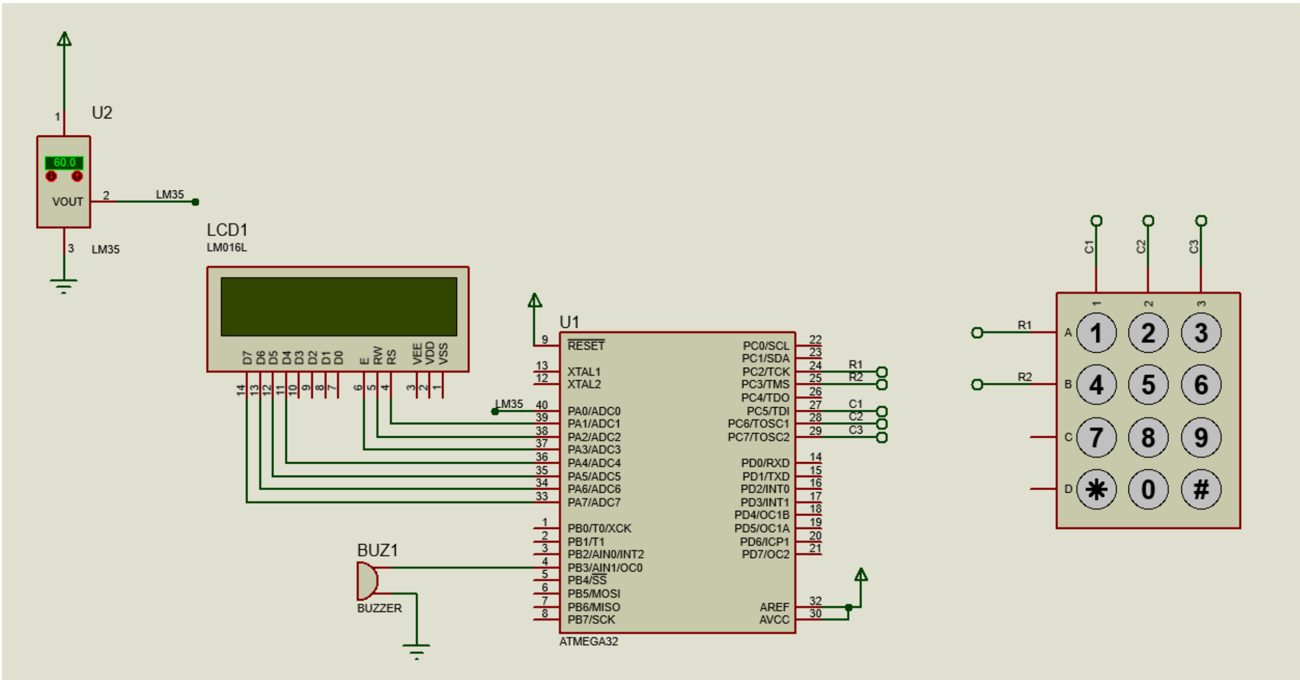


Figure 2. Project Circuit Schematic

2.2. Modules Description

2.2.1. DIO (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. 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.

2.2.3. ADC Module

An *Analog-to-Digital Converter ADC* is used to convert an analog signal such as voltage to a digital form so that it can be read and processed by a microcontroller. Most microcontrollers nowadays have built-in *ADC* converters. It is also possible to connect an external ADC converter to any type of microcontroller. *ADC* converters are usually 10 or 12 bits, having 1024–4096 quantization levels.

2.2.4. 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. The module assumes that this pin is connected to a digital output pin on the microcontroller. The pin number is typically configurable so that it can be easily changed if necessary. The module also assumes that the *Buzzer* operates at a fixed frequency and duty cycle.

2.2.5. 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.

2.2.6. TEMP SENSOR Module

A temperature sensor is an electronic device that measures the temperature of its environment and converts the input data into electronic data to record, monitor, or signal temperature changes. The LM35 device is rated to operate over a −55°C to 150°C temperature range, while the LM35C device is rated for a −40°C to 110°C range (−10° with improved accuracy).

2.2.7. 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.

2.2.8. Design

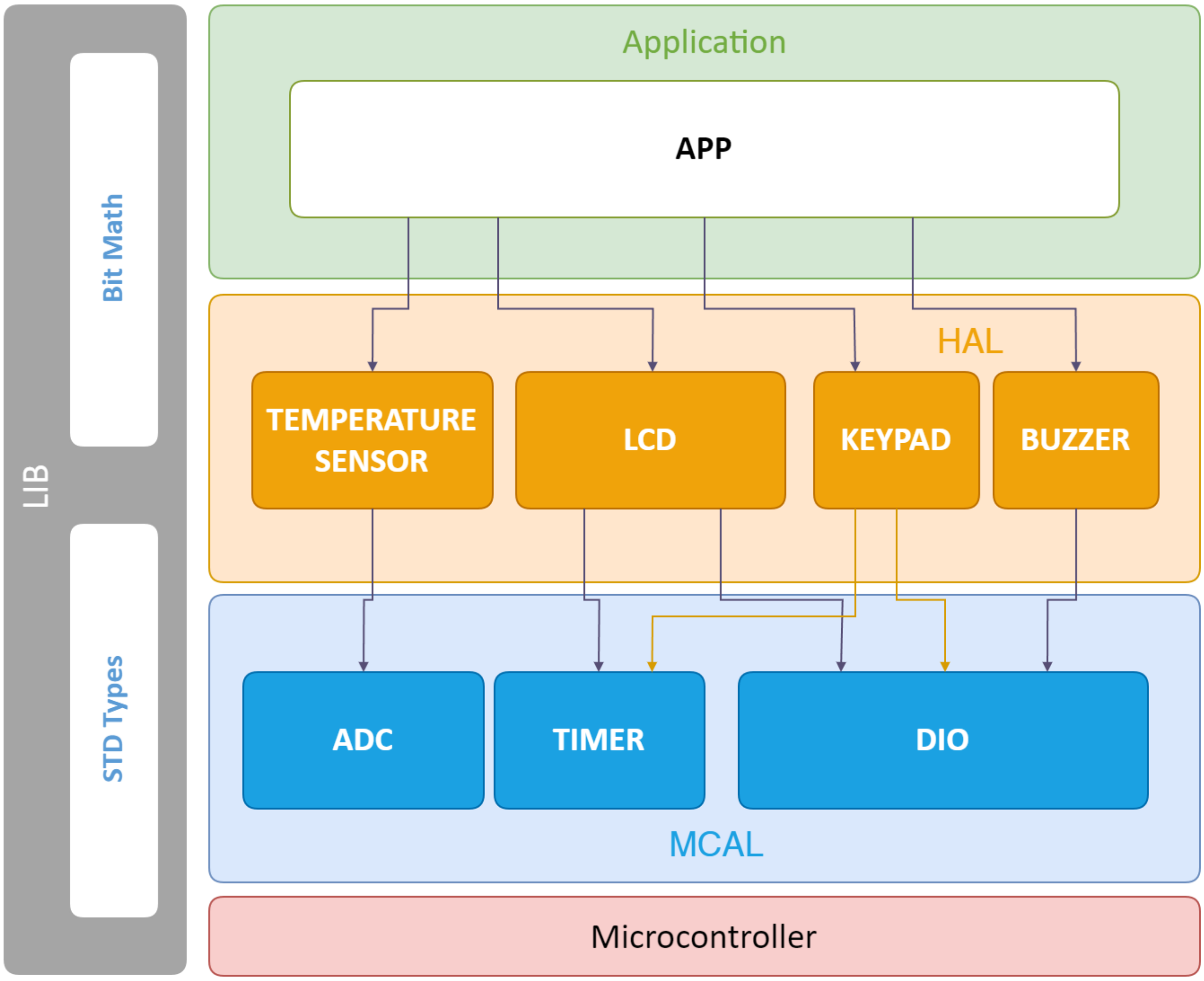


Figure 3. System Modules Design

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_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. 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
|             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);
```

2.3.2.3. ADC Driver

```
| This function initializes the ADC module by setting various configuration
| parameters.
|
| Return
|     void
|
void ADC_initialization(void);

| This function starts an ADC conversion for a specified channel and returns an error
| state if the channel ID is not valid.
|
| Parameters
|           [in] u8_a_channelId An 8-bit unsigned integer representing the ID of the
|                ADC channel to be used.
| Return
|     STD_OK if the channel ID is valid and the conversion is started successfully, or
|      STD_NOK if the channel ID is not valid.
|
u8 ADC_startConversion(u8 u8_a_channelId);

| This function gets the digital value from the ADC register based on the specified
| interruption mode and returns an error state.
|
| Parameters
|           [in] u8_a_interruptionMode This parameter specifies the mode of operation
|                for the ADC. It can be either POLLING_MODE or INT_MODE.
|           [in] pu16_a_returnedDigitalValue A pointer to a variable where the
|                digital value obtained from the ADC will be stored.
| Return
|     Error state, which is either STD_OK or STD_NOK.
|
u8 ADC_getDigitalValue(u8 u8_a_interruptionMode, u16
*pu16_a_returnedDigitalValue);

| This function sets a callback function for the ADC interrupt and returns an error
| state.
|
| Parameters
|           [in] pf_a_interruptAction A pointer to a function that will be called
|                when an ADC interrupt occurs.
| Return
|     It will return STD_OK if the pointer to function is not NULL. It will return
|      STD_NOK if the pointer to function is NULL.
|
u8 ADC_setCallBack(void (*pf_a_interruptAction)(void));
```

2.3.3. HAL APIs

2.3.3.1. 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();
```

2.3.3.2. 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);



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

2.3.3.3. TEMP SENSOR APIs

```
| Initializing the ADC module.
|
| Parameters
|            void
| Return
|     void
|
void TEMPSENSOR_init(void);
```

```
| taking the ADC value and mapping this value to the corresponding temperature value.
| the o/p scale factor of theLM35 sensor is 10mV/°C and it
| provides an output voltage of 250 mV at 25°C .
| Parameters
|            void
| Return
|     void
|
void TEMPSENSOR_getValue(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);
```

2.3.4. APP APIs

```
| Initializes the application and its components.
|
| This function initializes the timers, temperature sensor, buzzer, keypad,
| LCD and other components required for the application to function.
| It also displays a welcome message and sets the initial temperature to
| 20 degrees Celsius. Finally, it switches the application state to adjust
| the AC control procedure.
|
void APP_initialization( void );

|
| Start the AC Control program.
|
| This function contains the main program loop that controls the AC system.
| It switches between two states, adjust and running, and performs different tasks
| in each state based on user input and temperature readings.
|
| Return
|     void.
|
void APP_startProgram  ( void );

| Switch between AC state's "running" and "adjust"
|
| Parameters
|            [in]u8_a_state state to set (STATE_RUNNING/STATE_ADJUST)
| Return
|     void
|
void APP_switchState(u8 u8_a_state);

| Updates the desired temperature value based on user input.
|
| Parameters
|            [in]u8_a_action The action to perform on the temperature
|                          value (ACTION_INCREMENT/ACTION_DECREMENT).
|
| This function updates the UI temperature value and progress bar based
|  on the provided action.
| If the new desired temperature value is out of range (MAXIMUM_TEMP or
|  MINIMUM_TEMP), the function will buzz the user and revert the value
| back to the original.
|
void APP_changeTemp(u8 u8_a_action);

| Resets the desired temperature value to the default temperature.
void APP_resetToDefault();
```

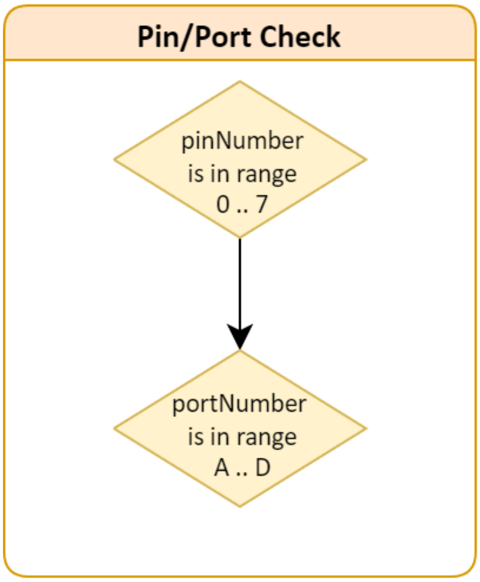
3. Low Level Design

3.1. MCAL Layer

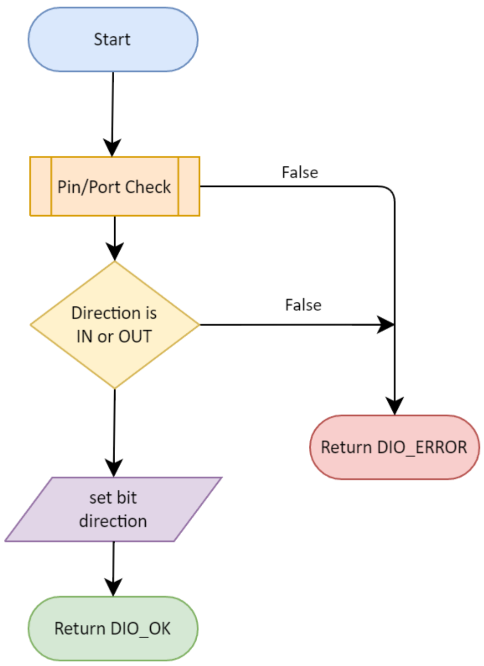
3.1.1. DIO Module

3.1.1.a. sub process

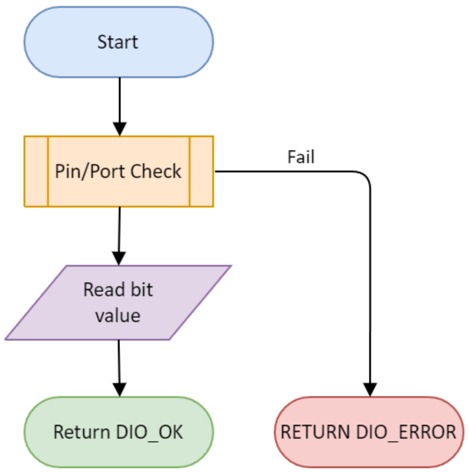
The following Pin/Port check subprocess is used in some of the DIO APIs flowcharts



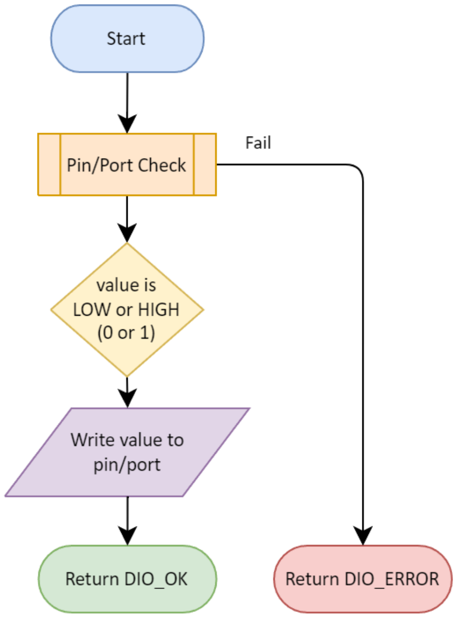
3.1.1.1. DIO_init



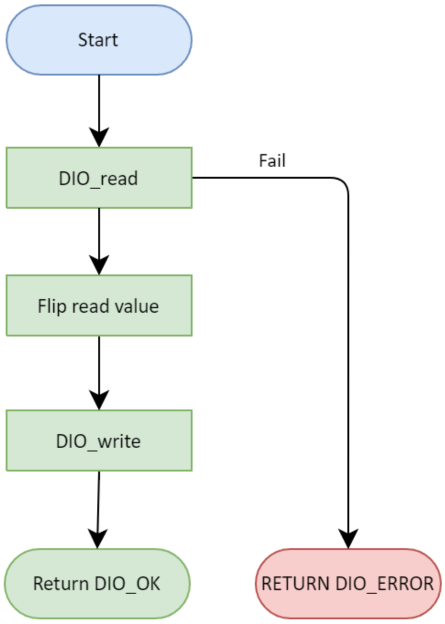
3.1.1.2. DIO_read



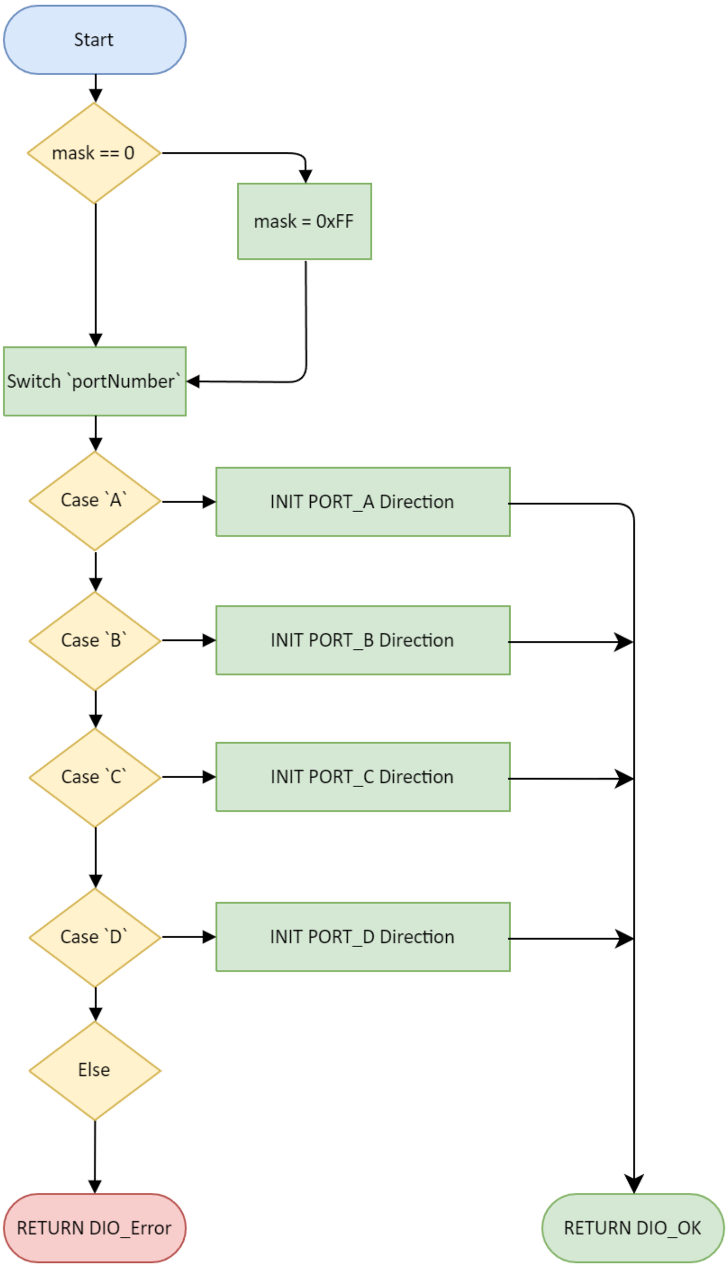
3.1.1.3. DIO_write



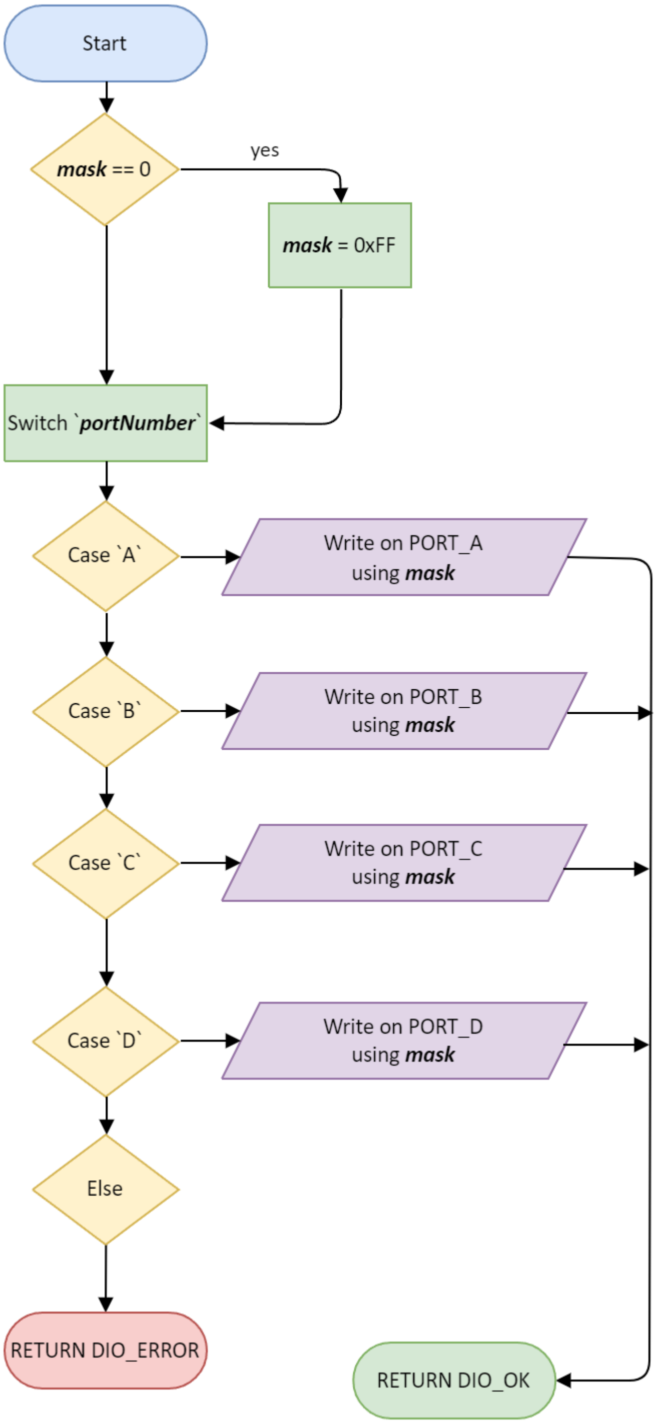
3.1.1.4. DIO_toggle



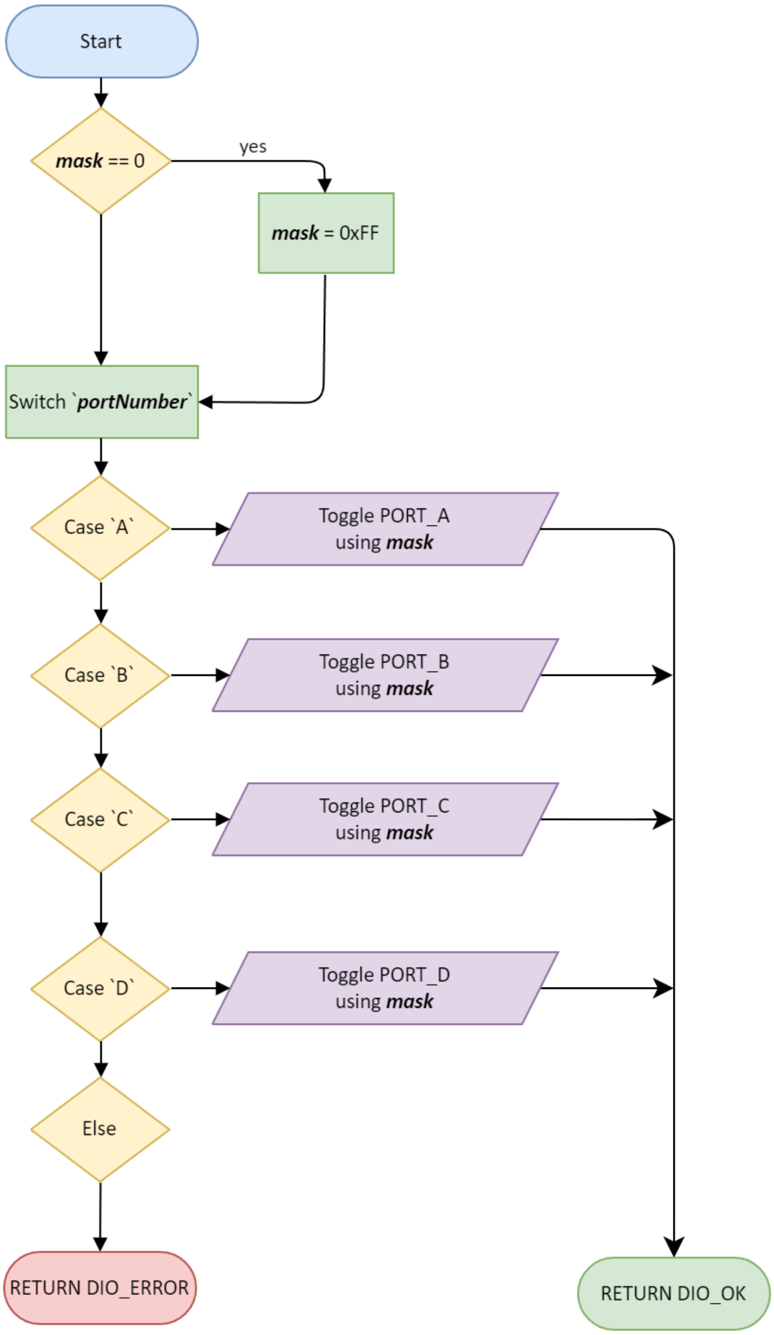
3.1.1.5. DIO_portInit



3.1.1.6. DIO_portWrite

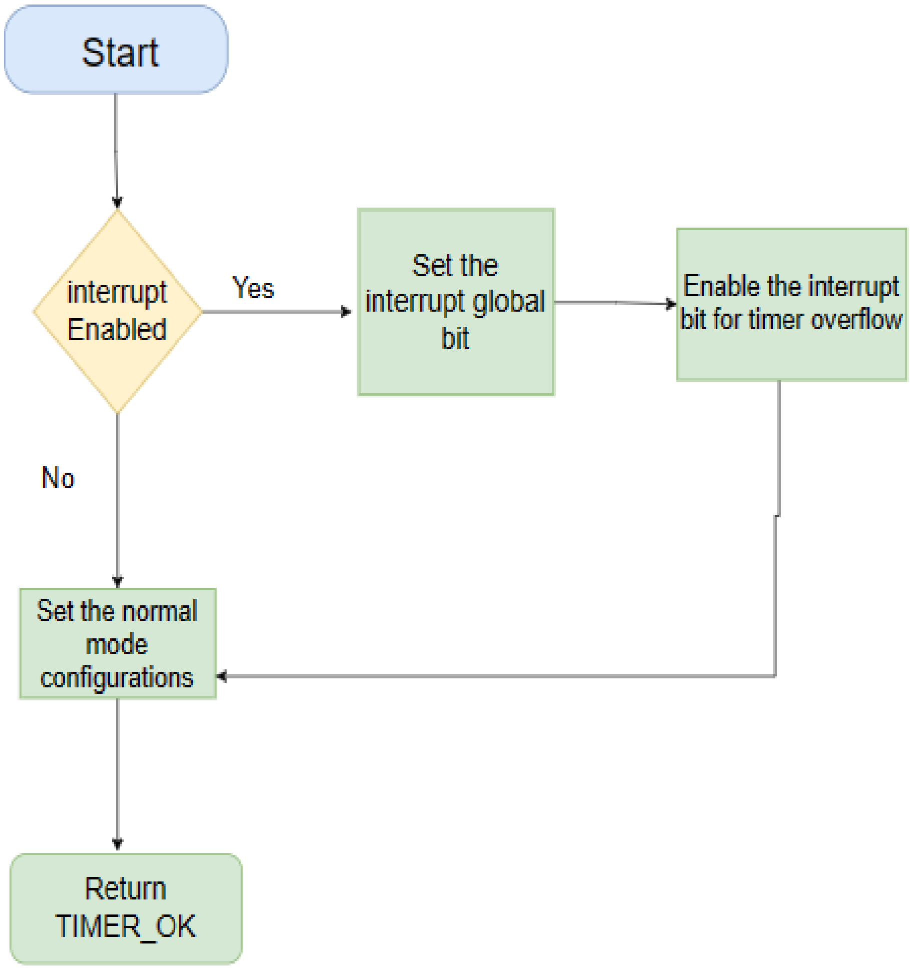


3.1.1.7. DIO_portToggle

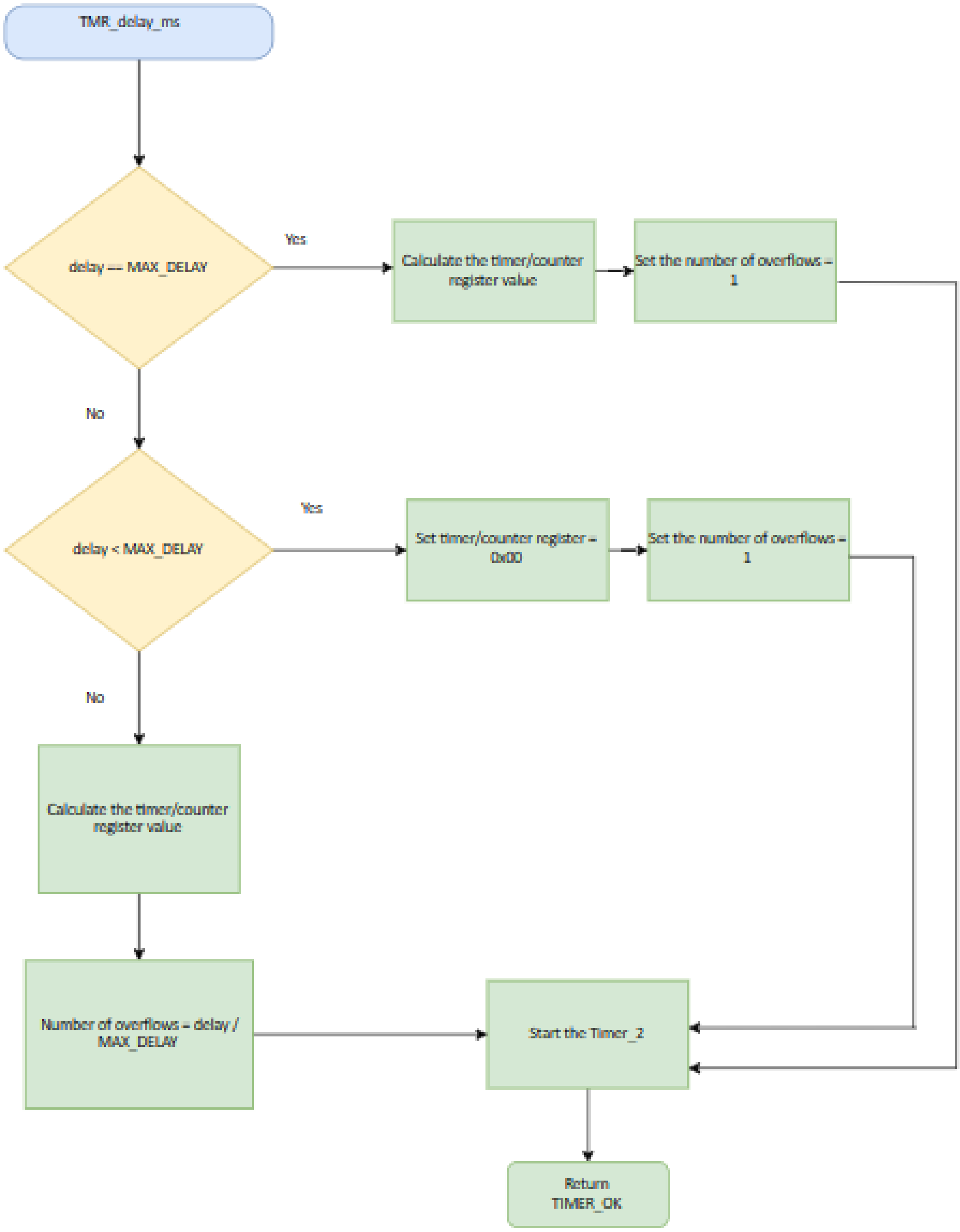


3.1.2. Timer Module

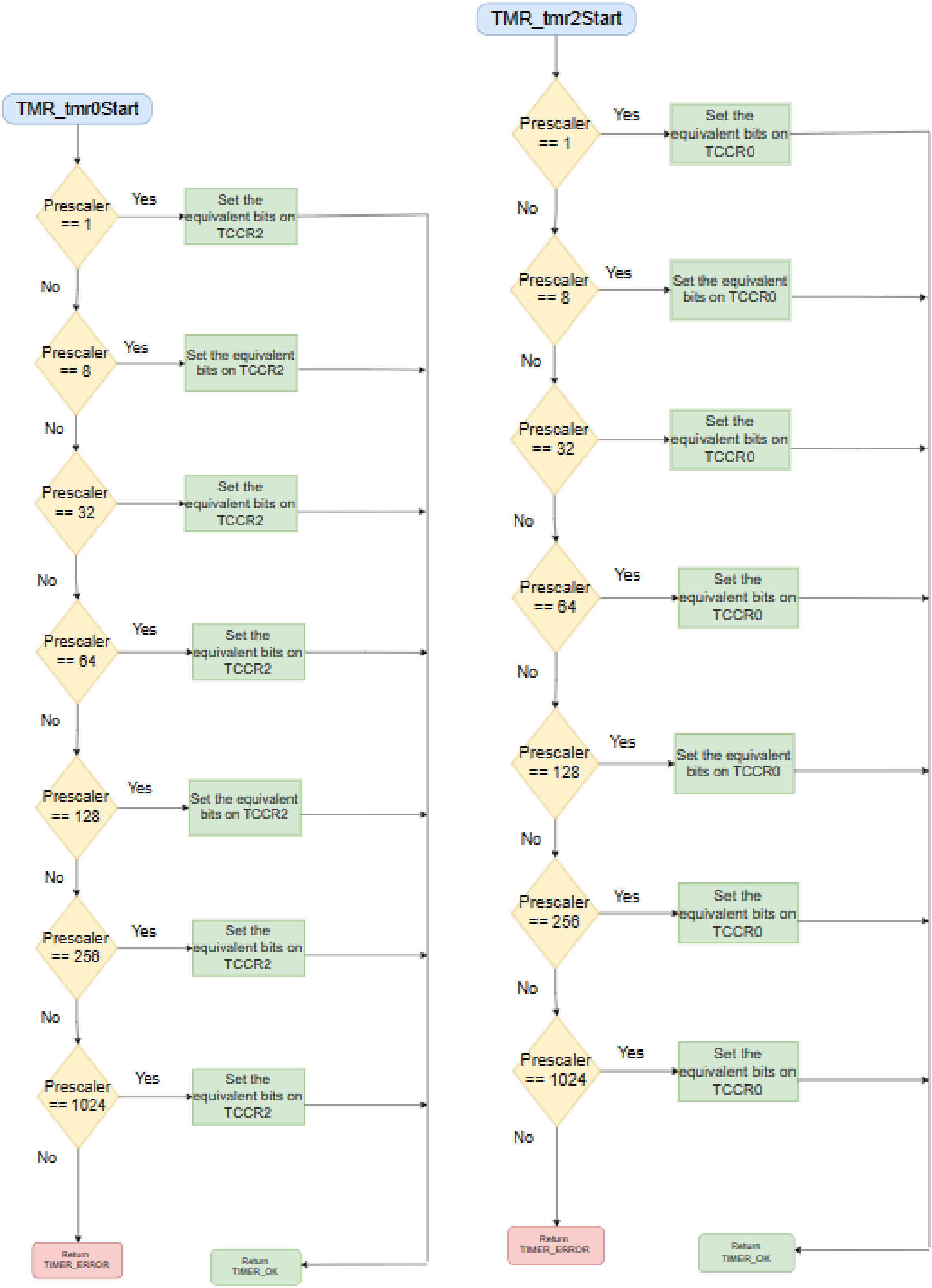
3.1.2.1. TMR_tmr0NormalModeInit / TMR_tmr2NormalModeInit



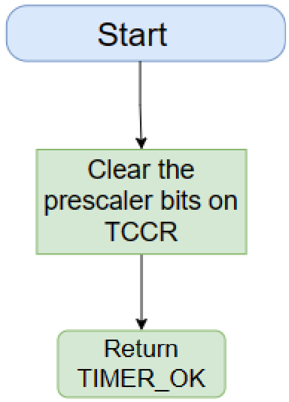
3.1.2.2. TMR_tmr0Delay / TMR_tmr2Delay



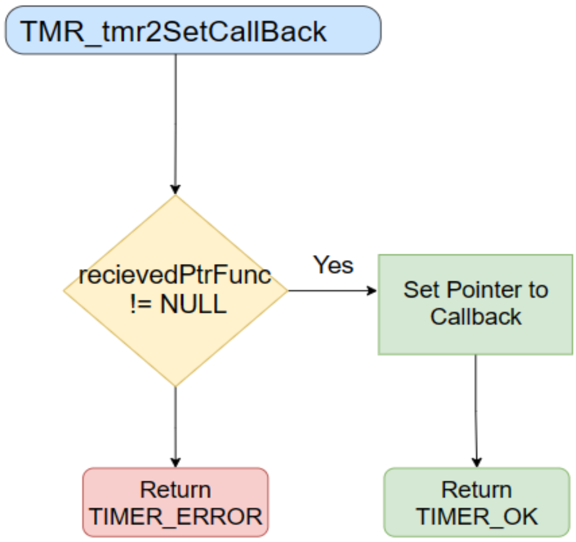
3.1.2.3. TMR_tmr0Start / TMR_tmr2Start



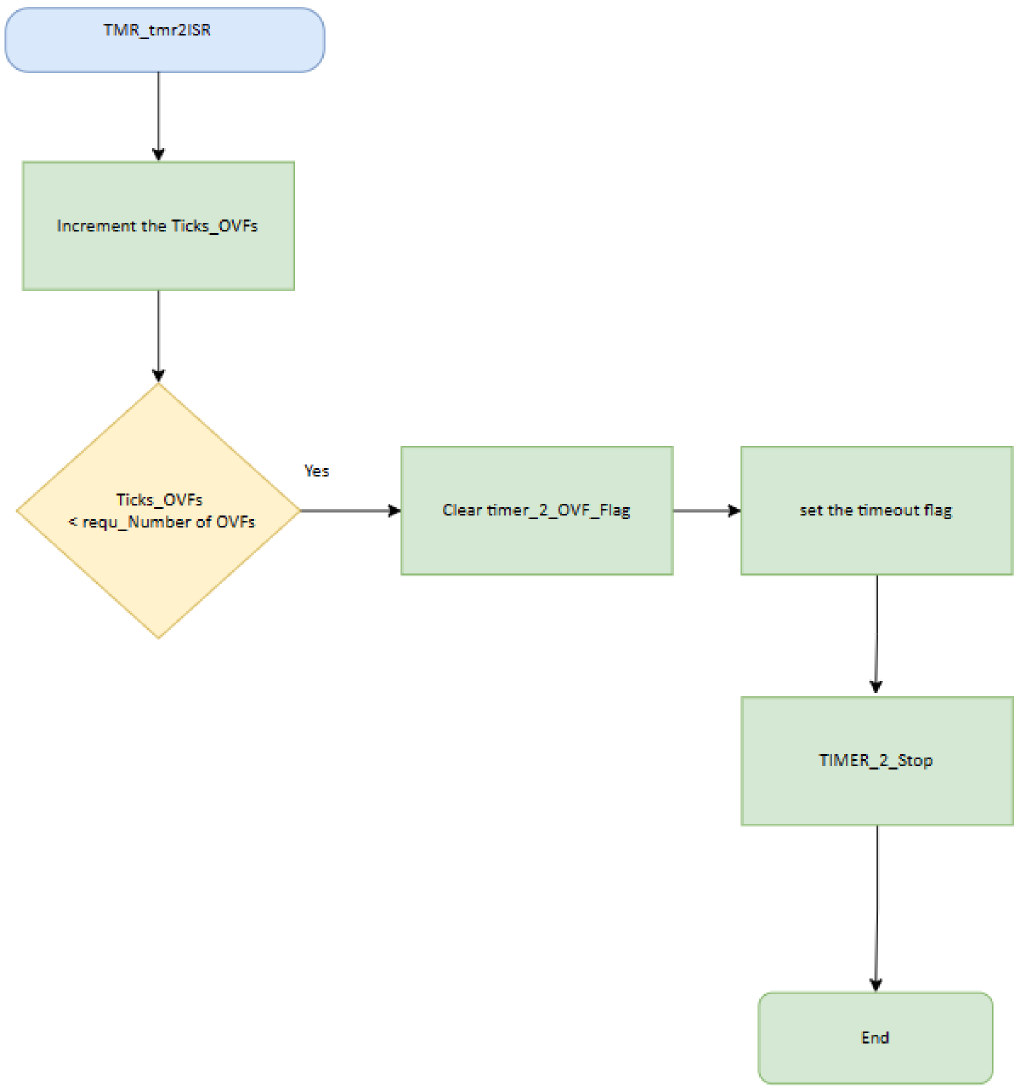
3.1.2.4. TMR_tmr0Stop / TMR_tmr2Stop



3.1.2.5. TMR_ovfSetCallback

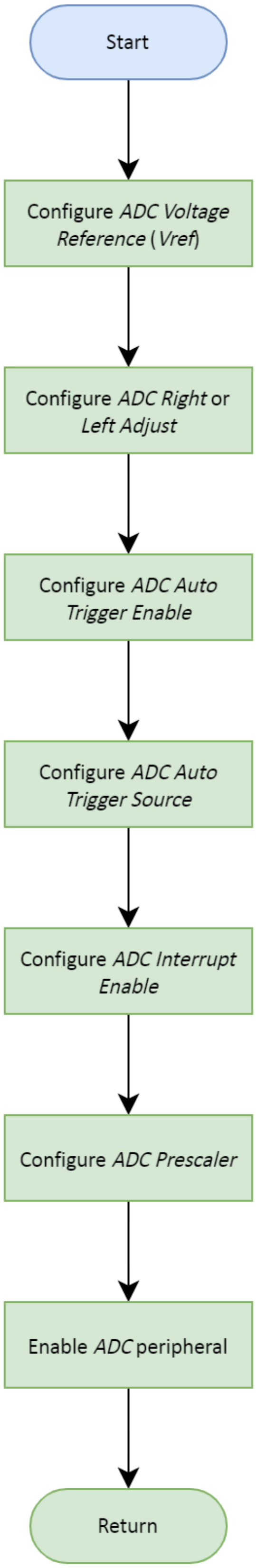


3.1.2.6. TMR2_ovfVect

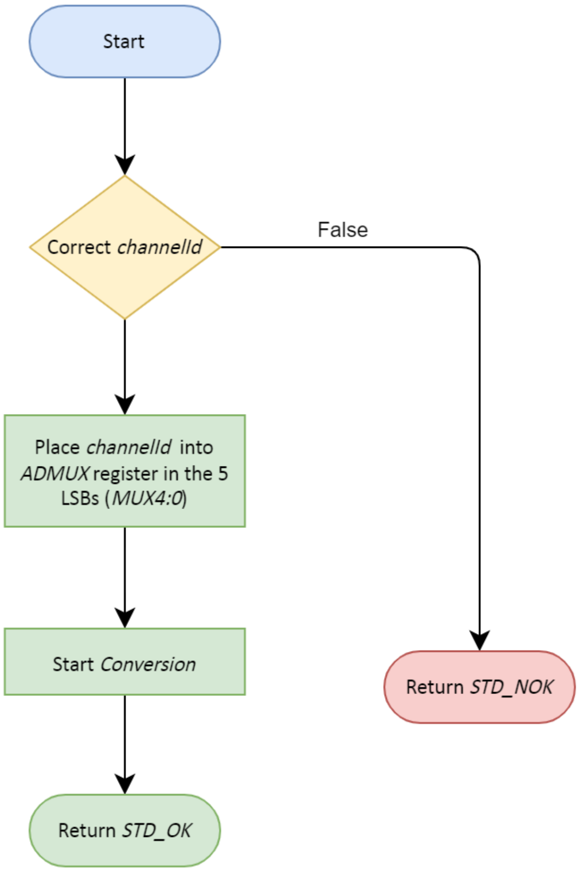


3.1.3. ADC Module

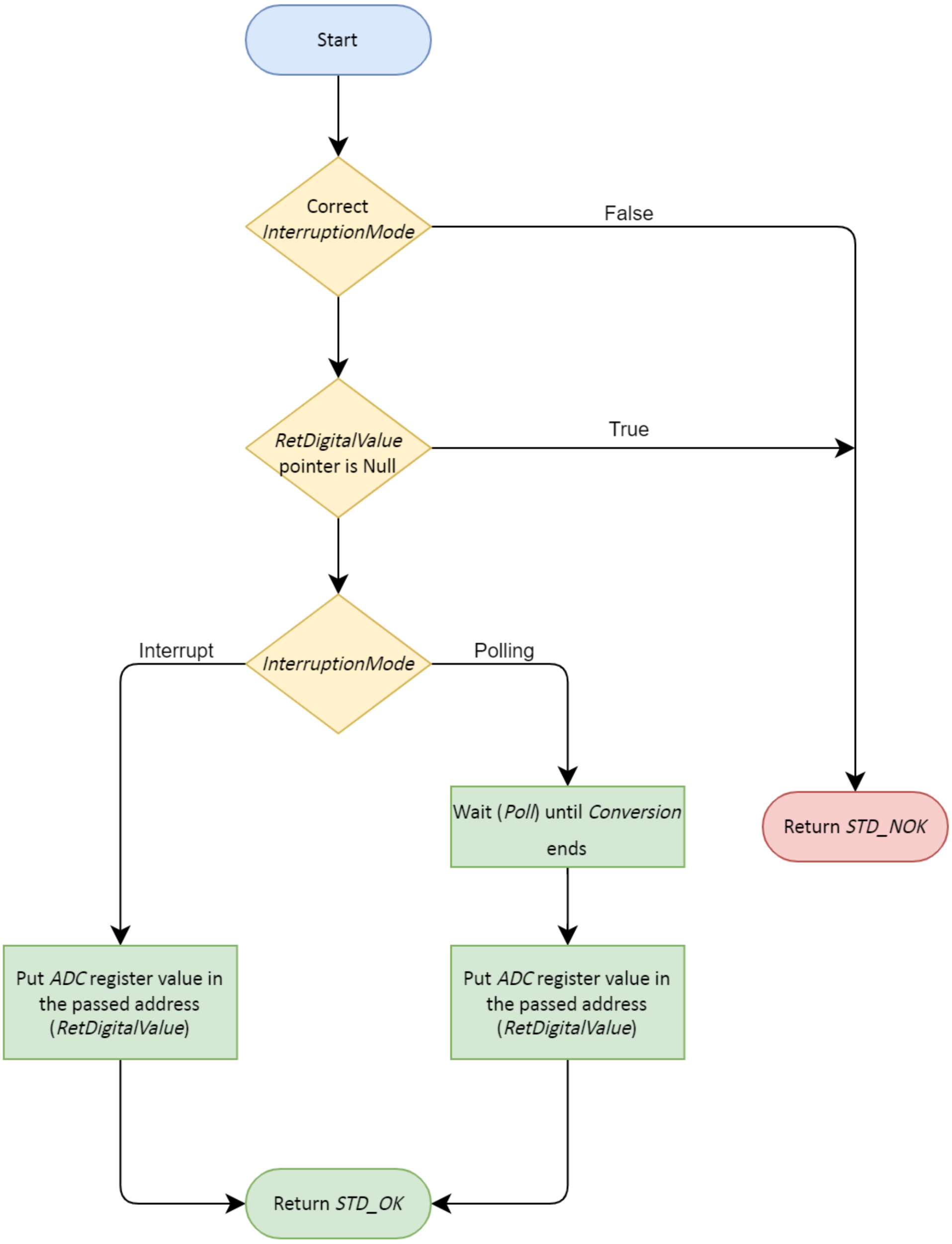
3.1.3.1. ADC_initialization



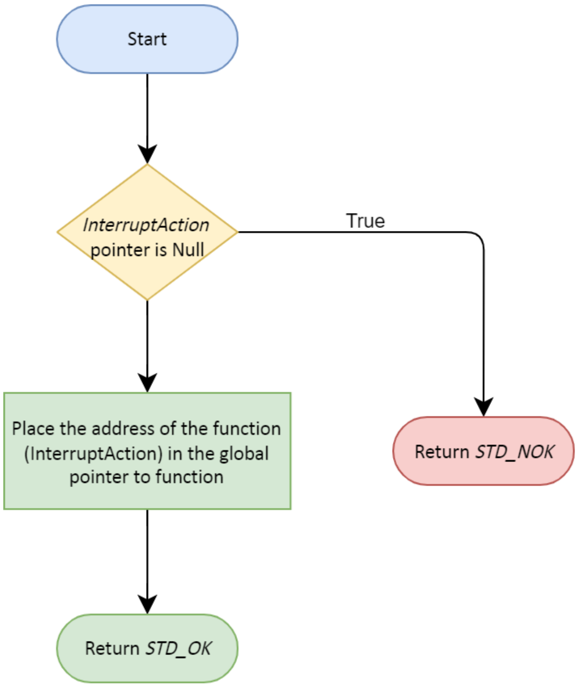
3.1.3.2. ADC_startConversion



3.1.3.3. ADC_getDigitalValue



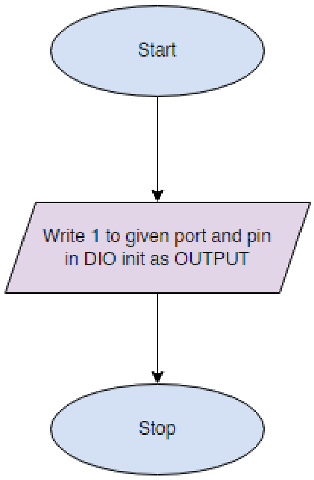
3.1.3.4. ADC_setCallBack



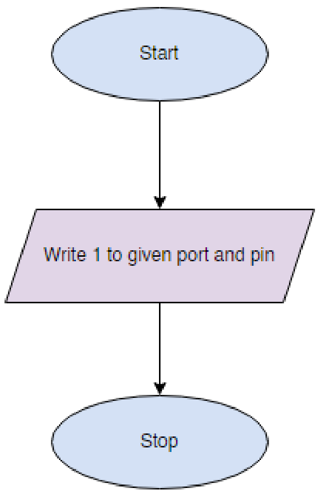
3.2. HAL Layer

3.2.1. Buzzer Module

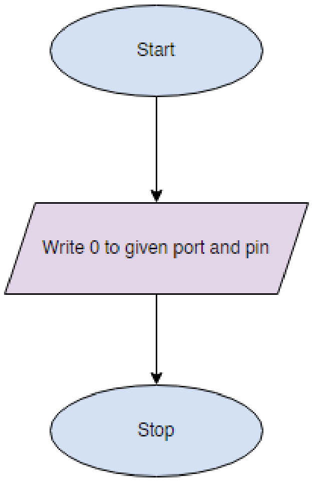
3.2.1.1. BUZZER_init



3.2.1.2. BUZZER_on

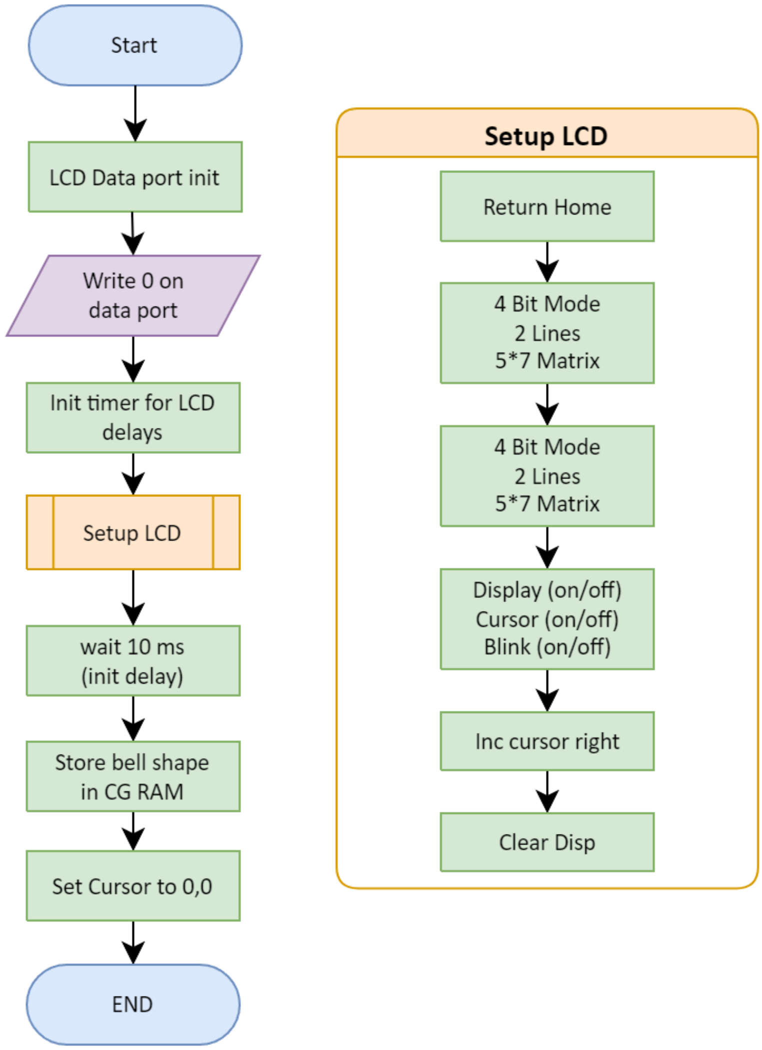


3.2.1.3. BUZZER_off

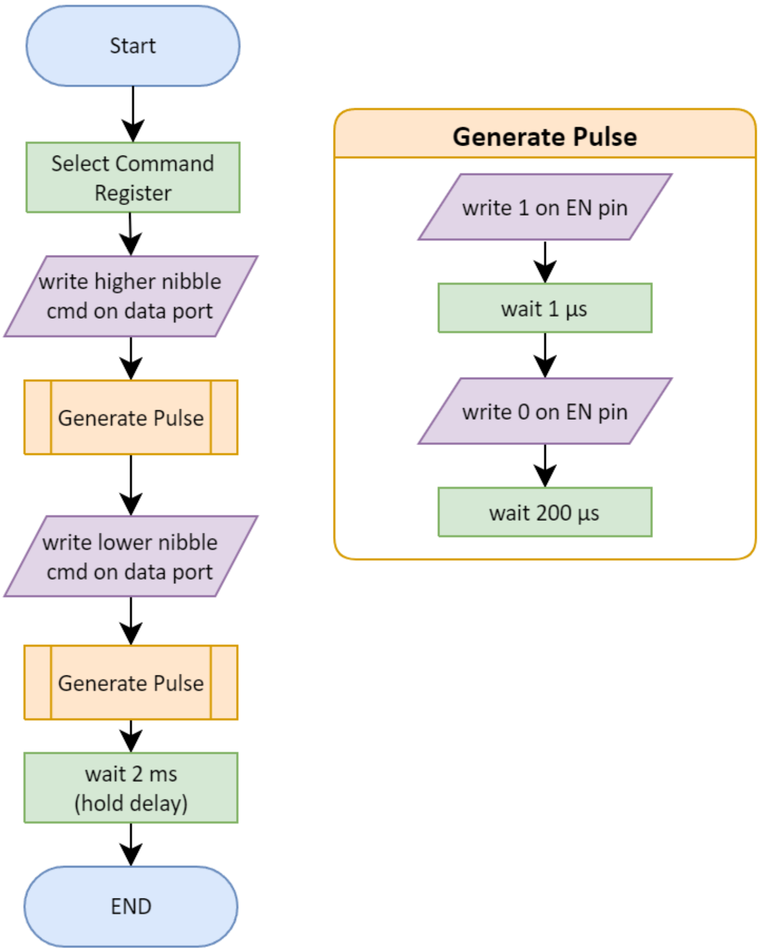


3.2.2. LCD Module

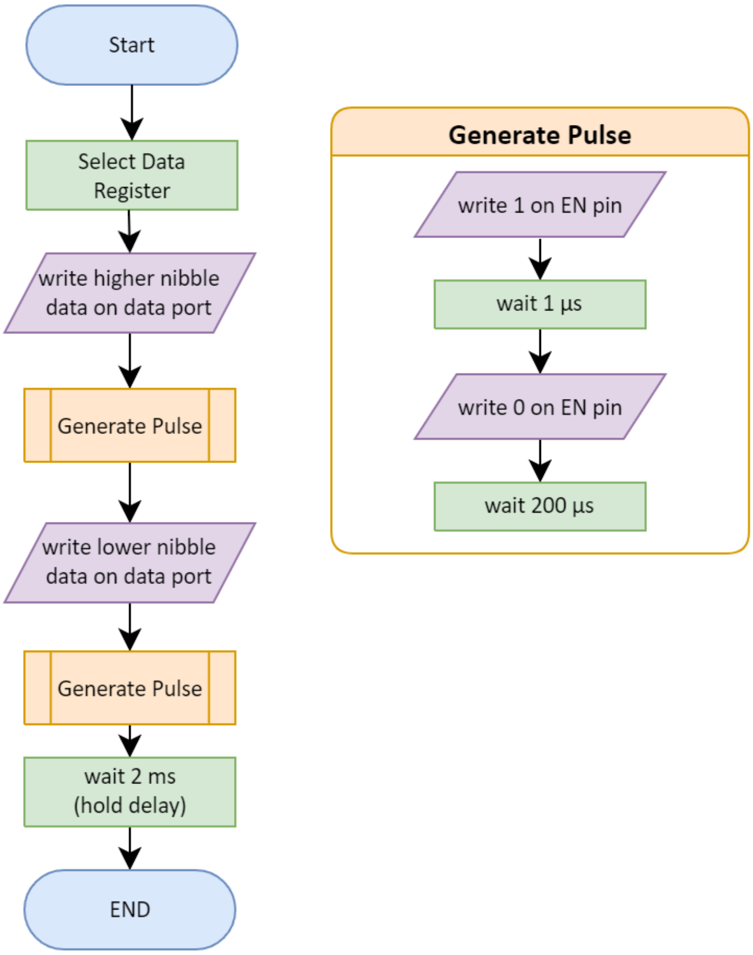
3.2.2.1. LCD_init



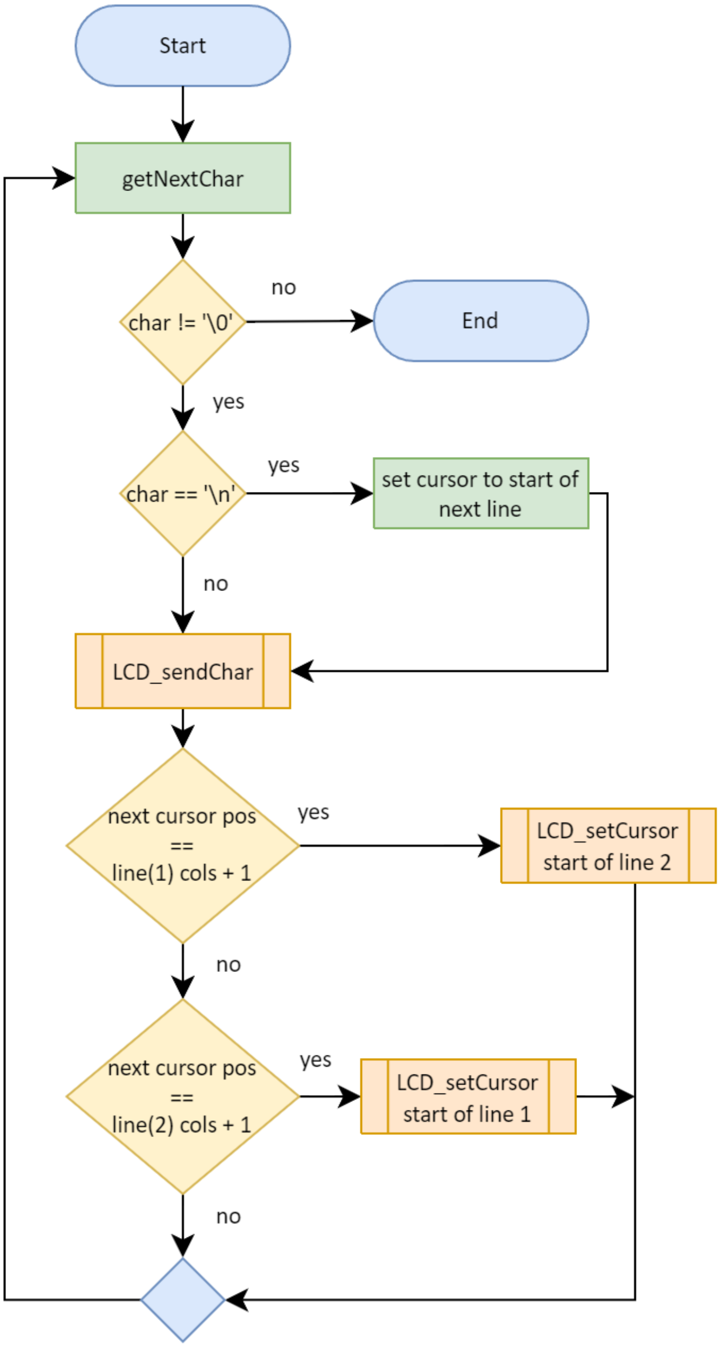
3.2.2.2. LCD_sendCommand



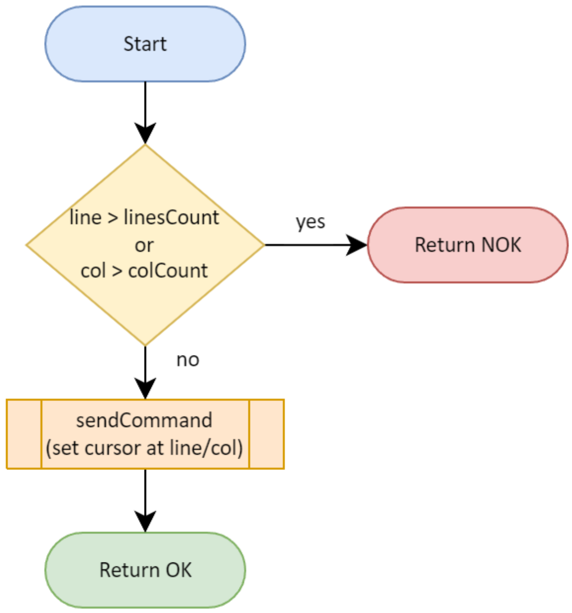
3.2.2.3. LCD_LCD_sendChar



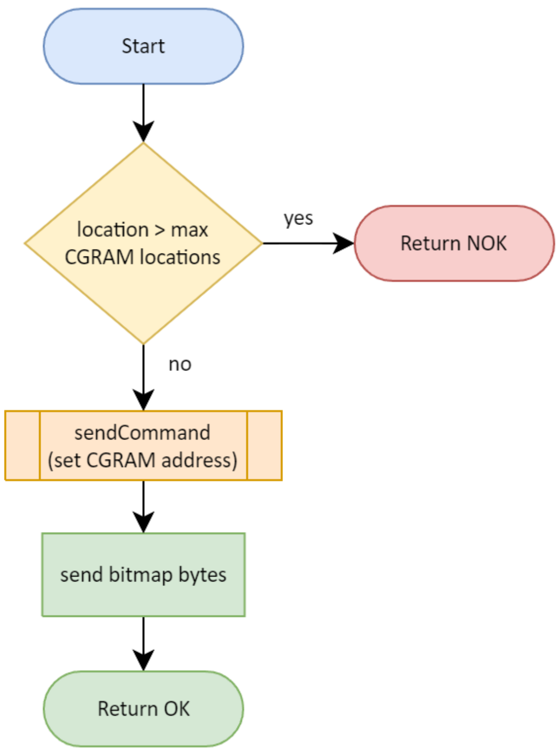
3.2.2.4. LCD_sendString



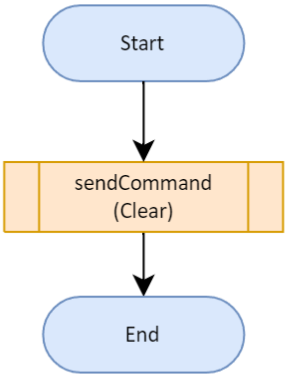
3.2.2.5. LCD_setCursor



3.2.2.6. LCD_storeCustomCharacter

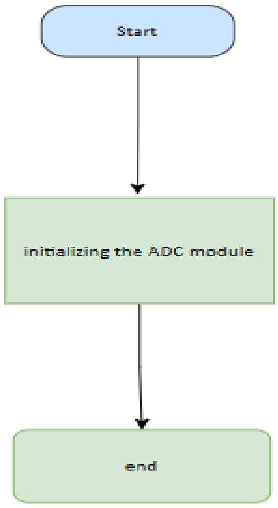


3.2.2.7. LCD_clear

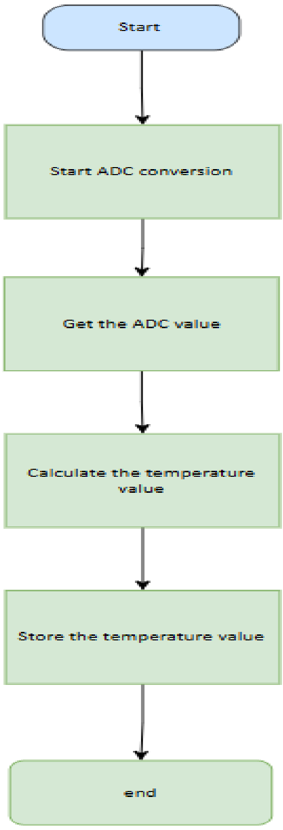


3.2.3. TEMP SENSOR Module

3.2.3.1. TEMPSENSOR_init

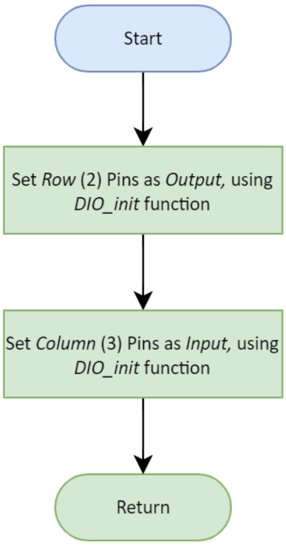


3.2.3.1. TEMPSENSOR_getValue

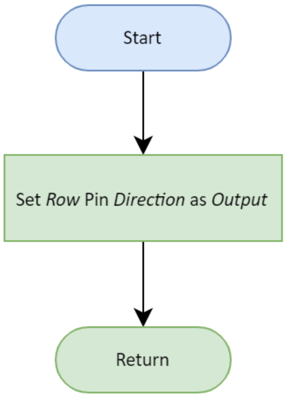


3.2.4. KPD Module

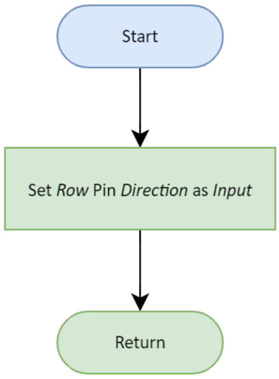
3.2.4.1. KPD_initKPD



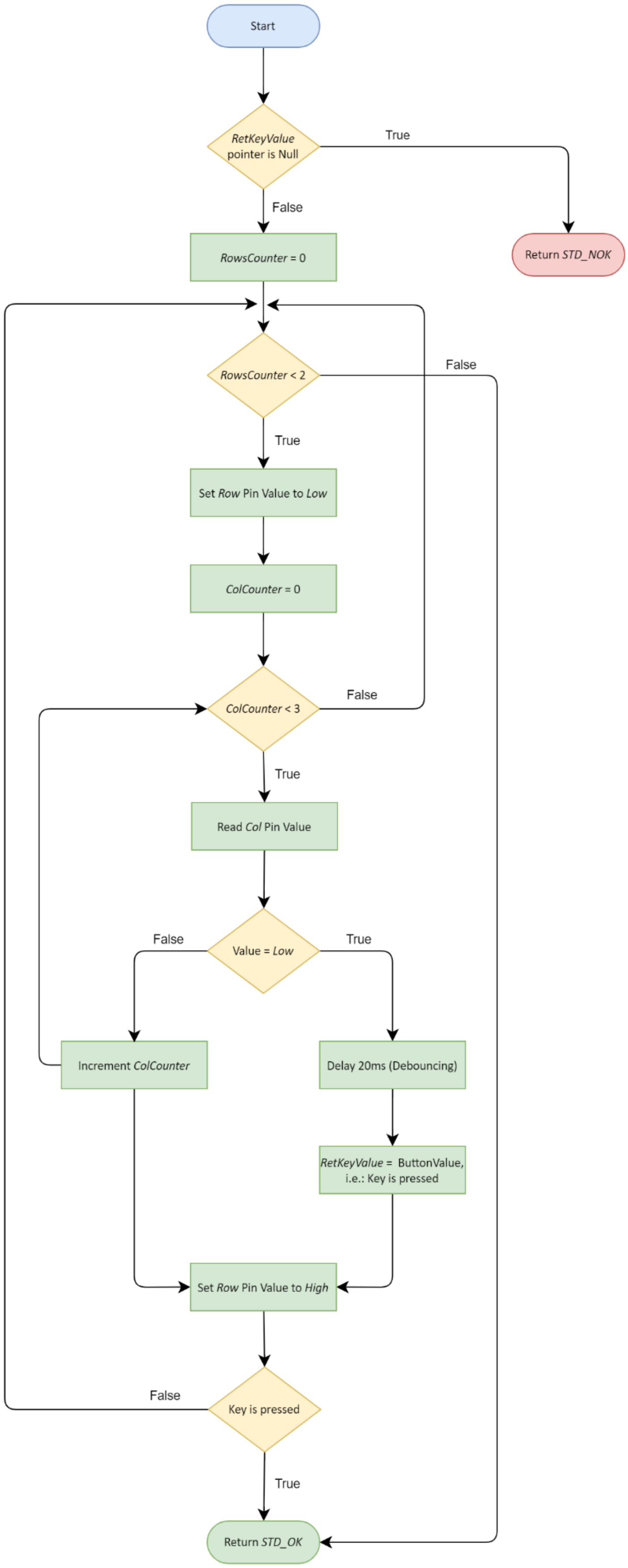
3.2.4.2. KPD_enableKPD



3.2.4.3. KPD_disableKPD

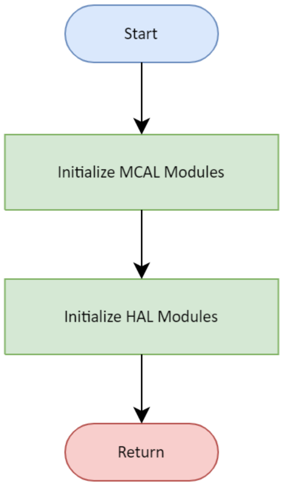


3.2.4.4. KPD_getPressedKey

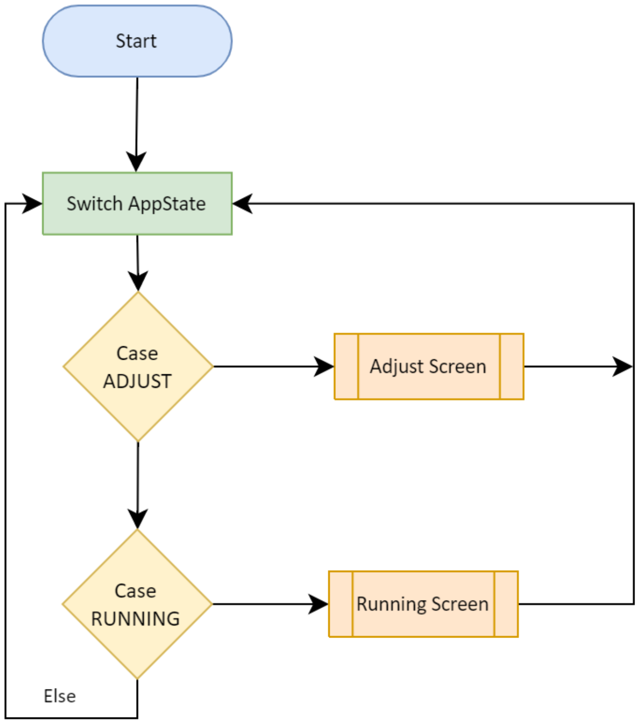


3.3. APP Layer

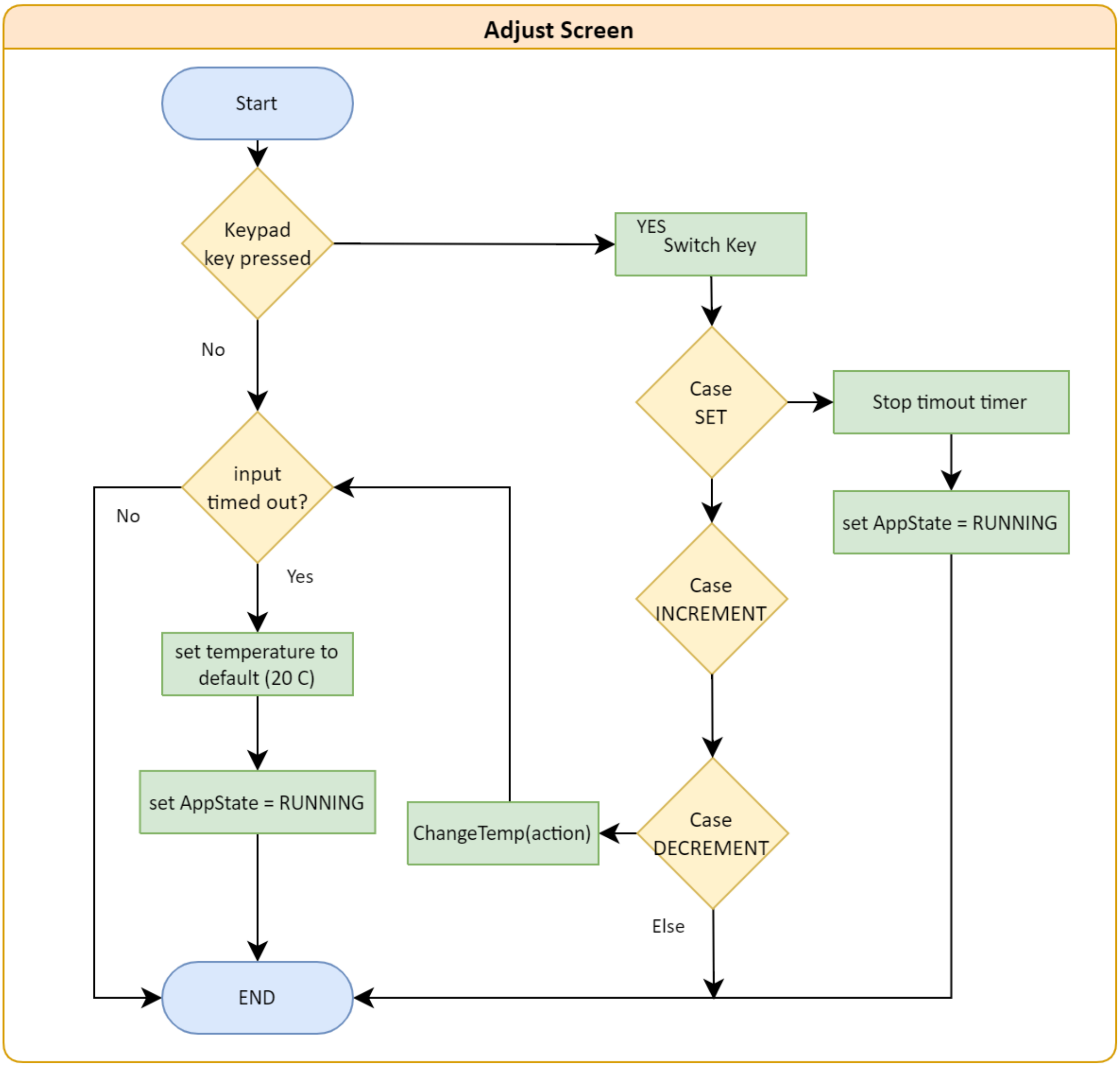
3.3.1. APP_initialization



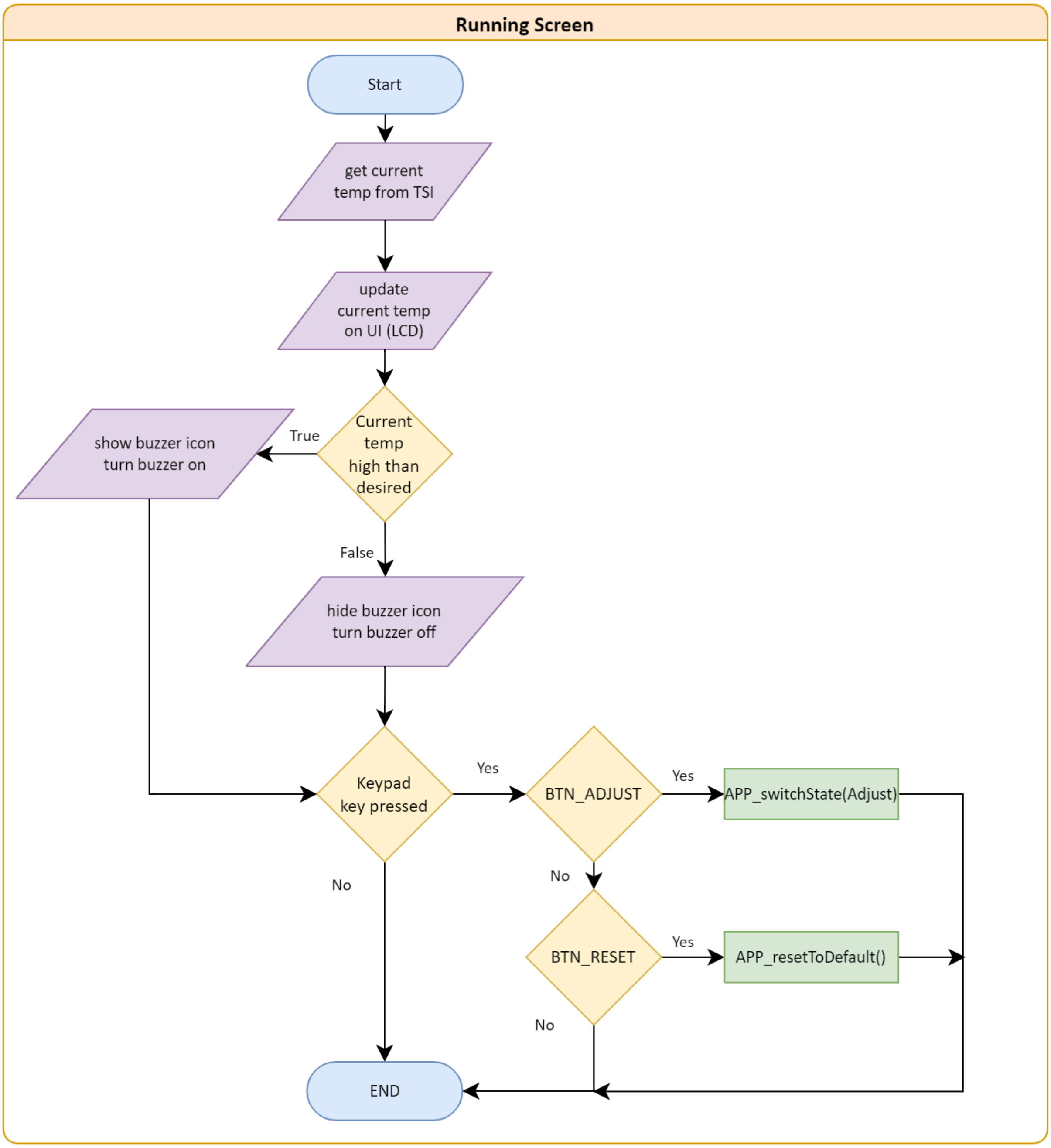
3.3.2. APP_startProgram



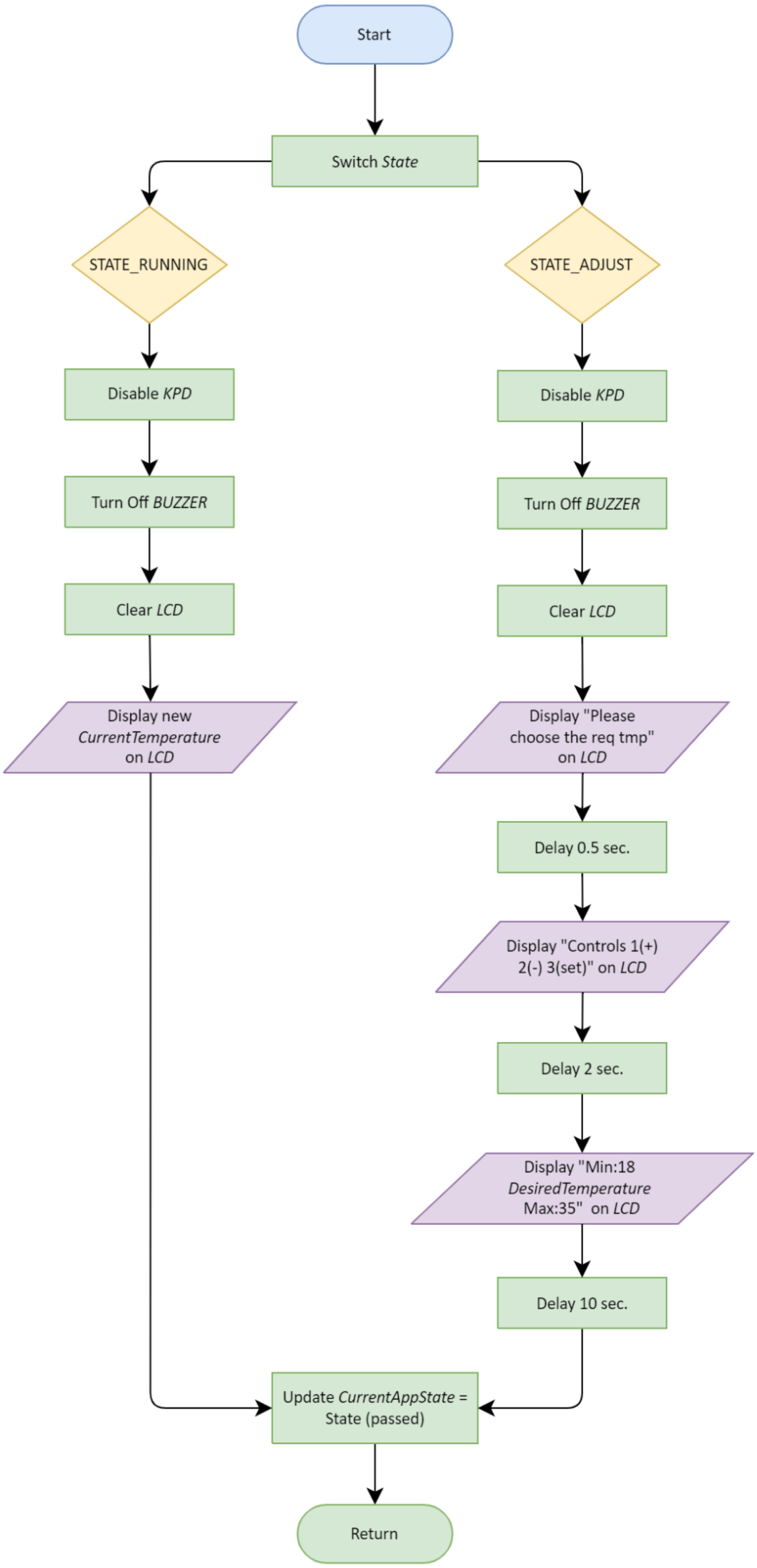
3.3.2.a. Adjust Screen (sub process)



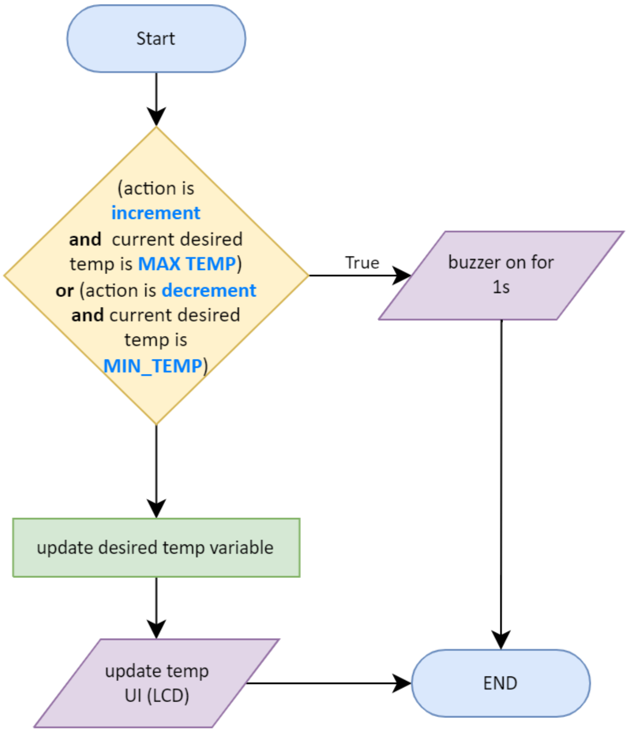
3.3.2.b. Running Screen (sub process)



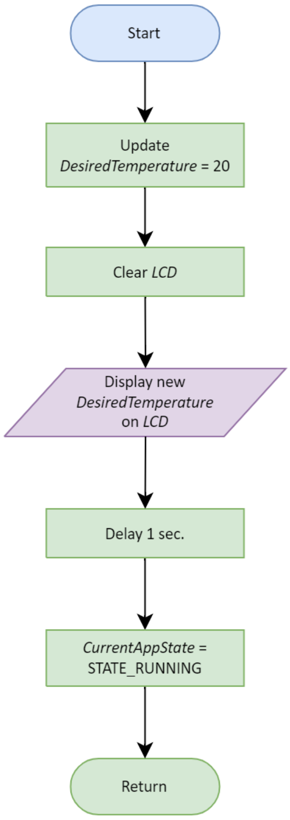
3.3.3. APP_switchState



3.3.4. APP_changeTemp



3.3.5. APP_resetToDefault



4. Issues that caused delivery delays

1. Pinpointing the Root Cause of App Delays and Character Scrambling on LCD

The application delays were not functioning correctly, resulting in the LCD displaying rapidly changing characters before settling on the last UI page with scrambled characters. After exhaustive and time-consuming debugging, including multiple runs with permutations, the root cause was identified as issues with reading from problematic arrays, as well as errors in the timer calculations and flags.

2. Solving Application Unresponsiveness due to Nested Switch in Main Loop

In the main loop, a nested switch was implemented, causing the application to become unresponsive after the initial execution. Despite thorough debugging of the secondary loop, no issues were identified with the arguments provided, which mirrored those used during the first run. However, the software continued to exit the nested switch prematurely, rather than entering the correct case. Despite additional debugging efforts, the underlying cause could not be identified, and a decision was made to replace the nested switch with an if/else block, which successfully resolved the issue.

3. Setting ADC driver configurations to output the correct temperature

Incorrect and inaccurate temperature readings were occurring due to the use of inaccurate ADC configurations. For instance, the temperature sensor was configured to read 60°C, but the ADC was outputting a temperature of 30°C, leading to erroneous temperature readings. To resolve this issue, we reconfigured the ADC driver by implementing a 128 prescaler and utilizing an internal 2.56v voltage reference.

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*