Obstacle Avoiding Car Design

2023

Author :

Sherif Ashraf Khadr

Table of Contents

Project Introduction

Car Components

1 - Atmega32 Microcontroller

2 - Four Motors (M1,M2,M3,M4)

3 - One Push Button To Change Direction Of Rotation

4 - Keypad Button 1 To Start

5 - Keypad Button 2 To Stop

6 - One Ultrasonic Sensor

7 - Lcd

Description

1 - In First Start The Robot Will Wait For 2 Seconds To Choose The Default Direction Of Rotation Using The Push Button & Display On The Lcd User Can Choose Between Right & Left

2 - If No Obstacles Or Object Is Far Than 70 Centimetres Then The Robot Will Move Forward With 30% Speed For 5 Seconds & Then It Will Move With Speed Of 50% As Long As Was No Object Or Object Are Located At More Than 70 Centimetres distance

3 - If Object Located Between 30 & 70 Centimetres Then The Robot Will Decrease Its Speed To 30%

4 - If Object Located Between 20 & 30 Centimetres Then The Robot Will Stop & Rotates 90 Degrees To Right Or Left According To The Chosen Configuration

5 - If Object Located Less Than 20 Centimetres Then The Robot Will Stop Turn Direction To Backwards With Speed 30% Until The Distance Is Greater Than 20 And Less Than 30 Then Perform Point 4

6 - If Obstacles Surrounding The Robot In All Direction Without Finding Any Distance Greater Than 20 It Will Stop Robot Will Check Frequently Each 3 Seconds If Any Obstacles Was Removed Or Not And Move In The Furthest Object

Layered Architecture

Layered architecture in embedded systems organizes software components into distinct layers, each with a specific responsibility and level of abstraction. The layers are arranged hierarchically, starting from the hardware layer, followed by MCAL, HAL and ending with the application layer. Layering provides benefits like modularity, scalability, and maintainability by separating software components into distinct layers with clear separation of concerns.

Mcal Layer

Mcal is a layer in embedded systems that provides a standardized interface to the microcontroller hardware. It includes functions/drivers for controlling hardware peripherals and is critical for enabling higher-level software layers to be developed independently of the hardware platform.

Ecu Layer

In a layered architecture for embedded systems, the ECU (Electronic Control Unit) layer sits above the MCAL layer and provides a higher-level, more application-specific interface for controlling the system's functions. It implements the system's primary functionality and interacts with the MCAL layer through well-defined APIs and interfaces. The ECU layer may also include functionality for managing system diagnostics, error handling, and

communication with other ECUs or external systems.

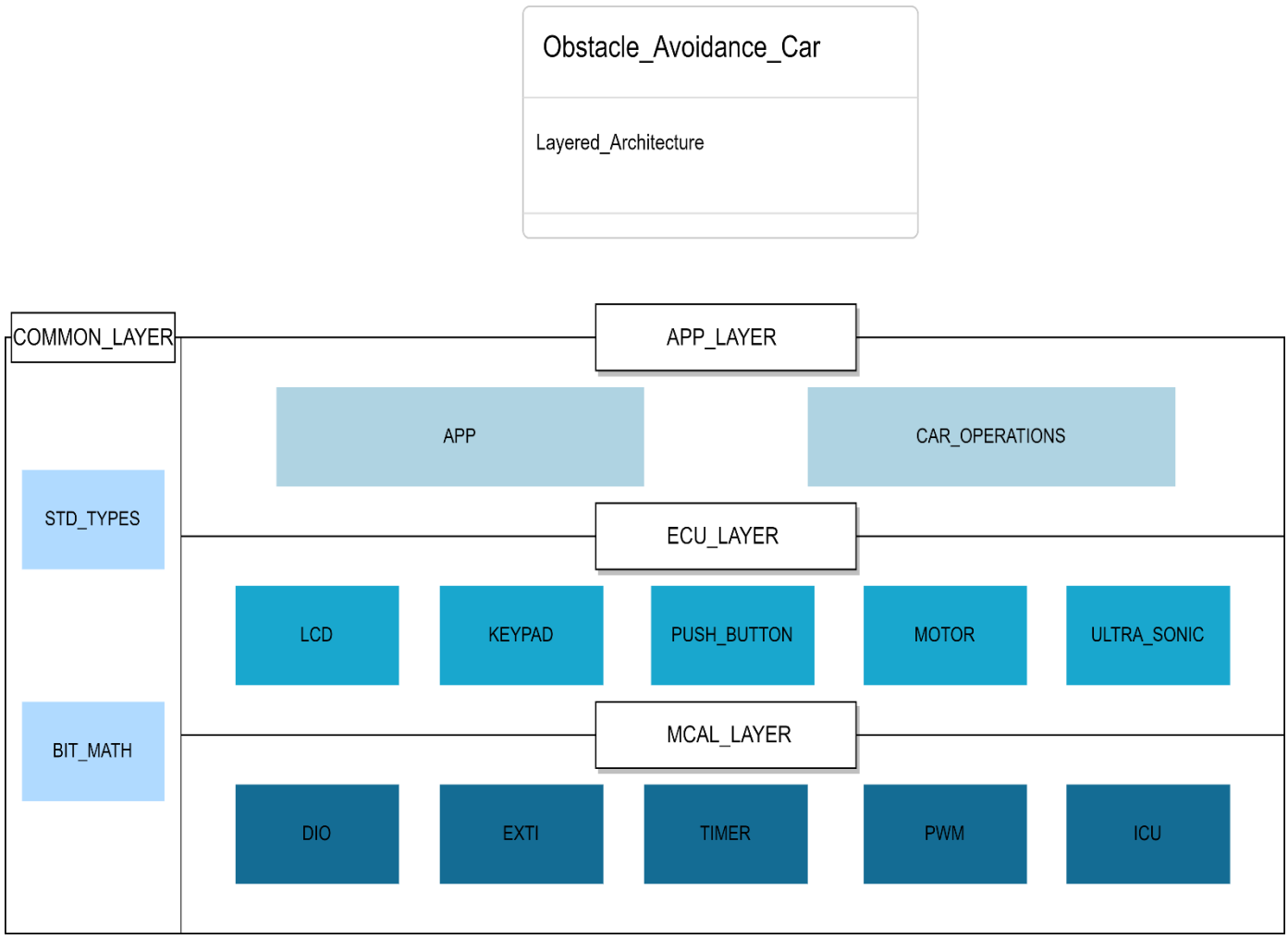
App Layer

In a layered architecture for embedded systems, the APP (Application) layer is responsible for providing the system's primary functionality and interacts with the lower layers through well-defined APIs and interfaces. It may include software components for implementing specific functions, managing system diagnostics, error handling, and communication with other systems.

Common Layer

In a layered architecture for embedded systems, the common layers provide shared services and functionality used by multiple software components. They typically include components for managing system resources, configuration settings, and common functionality such as logging or debugging. Separating common functionality into distinct layers enables the system to function cohesively and be more modular and maintainable.

Layered_Architecture_Figure



High Level Design
Drivers Documentation
Mcal Layer

1 - Dio Driver

/*
Enum : EN_logic_t

Description: Enumeration for the two possible logic states of a GPIO (General Purpose Input/Output) pin on a micro-controller.

Members:
- GPIO_LOGIC_LOW : This member represents the logic level of a GPIO pin when it is set to low (0V).
- GPIO_LOGIC_HIGH : This member represents the logic level of a GPIO pin when it is set to high (3.3V or 5V depending on the specific micro-controller).

Overall, the EN_logic_t enumeration provides a standardized way to represent and configure the logic levels of GPIO pins on a micro-controller, making it easier to write and maintain software that interacts with these pins.
*/

```
typedef enum{
   GPIO_LOGIC_LOW = 0,
   GPIO_LOGIC_HIGH
}EN_logic_t;
```

/*
Enum : EN_direction_t

Description: Enumeration for the two possible directions of a GPIO (General Purpose Input/Output) pin on a micro-controller.

Members:
- GPIO_DIRECTION_INPUT : This member represents the direction of a GPIO pin when it is set to input mode, allowing it to receive and read external signals.
- GPIO_DIRECTION_OUTPUT : This member represents the direction of a GPIO pin when it is set to output mode, allowing it to send signals to external devices.
Overall, the EN_direction_t enumeration provides a standardized way to represent and configure the direction

of GPIO pins on a micro-controller, making it easier to write and maintain software that interacts with these pins.
*/

```
typedef enum
{
   GPIO_DIRECTION_INPUT = 0,
   GPIO_DIRECTION_OUTPUT
}EN_direction_t;
```

/*
Enum : EN_pin_index_t

Description : Enumeration for the index of each GPIO (General Purpose Input/Output) pin on a port.

Members:
- GPIO_PIN0 : This member represents the index of the first GPIO pin on a port.
- GPIO_PIN1 : This member represents the index of the second GPIO pin on a port.
- GPIO_PIN2 : This member represents the index of the third GPIO pin on a port.
- GPIO_PIN3 : This member represents the index of the fourth GPIO pin on a port.
- GPIO_PIN4 : This member represents the index of the fifth GPIO pin on a port.
- GPIO_PIN5 : This member represents the index of the sixth GPIO pin on a port.
- GPIO_PIN6 : This member represents the index of the seventh GPIO pin on a port.
- GPIO_PIN7 : This member represents the index of the eighth GPIO pin on a port.

Overall, the EN_pin_index_t enumeration provides a standardized way to represent and identify each GPIO pin on a port, making it easier to write and maintain software that interacts with these pins.
*/
```
typedef enum
{
  GPIO_PIN0 = 0,
  GPIO_PIN1,
  GPIO_PIN2,
  GPIO_PIN3,
  GPIO_PIN4,
  GPIO_PIN5,
  GPIO_PIN6,
  GPIO_PIN7
}EN_pin_index_t;
```

/*
Enum : EN_port_index_t

Description : Enumeration for the index of each GPIO (General Purpose Input/Output) port on a micro-controller.

Members :
- GPIO_PORTA_INDEX : This member represents the index of the first GPIO port on a micro-controller.
- GPIO_PORTB_INDEX : This member represents the index of the second GPIO port on a micro-controller.
- GPIO_PORTC_INDEX : This member represents the index of the third GPIO port on a micro-controller.
- GPIO_PORTD_INDEX : This member represents the index of the fourth GPIO port on a micro-

controller.

Overall, the EN_port_index_t enumeration provides a standardized way to represent and identify each GPIO port on a micro-controller, making it easier to write and maintain software that interacts with these ports and their associated pins.
*/
typedef enum
{
   GPIO_PORTA_INDEX = 0,
   GPIO_PORTB_INDEX,
   GPIO_PORTC_INDEX,
   GPIO_PORTD_INDEX,
}EN_port_index_t;

/*
Struct : ST_pin_config_t

Description : A structure that contains the configuration settings for a GPIO (General Purpose Input/Output)pin on a micro-controller.

Members:
- port : A 3-bit field that specifies the index of the GPIO port to which the pin belongs (0-3).
- pin : A 3-bit field that specifies the index of the GPIO pin on the port (0-7).
- direction : A 1-bit field that specifies the direction of the GPIO pin (input or output).
- logic : A 1-bit field that specifies the logic level of the GPIO pin (high or low).

Overall, the ST_pin_config_t structure provides a standardized way to represent and configure the settings for a single GPIO pin on a micro-controller, making it easier to write and maintain software that interacts with these pins. The use of bit fields for the port, pin , direction , and logic members allows for efficient use of memory and reduces the size of the structure. The Uchar8_t type is likely a custom data type defined in the software to represent an 8-bit unsigned character.
*/

typedef struct
{
   Uchar8_t port : 3;
   Uchar8_t pin : 3;
   Uchar8_t direction : 1;
   Uchar8_t logic : 1;
}ST_pin_config_t;

/*

Function: GPIO_pin_direction_intialize

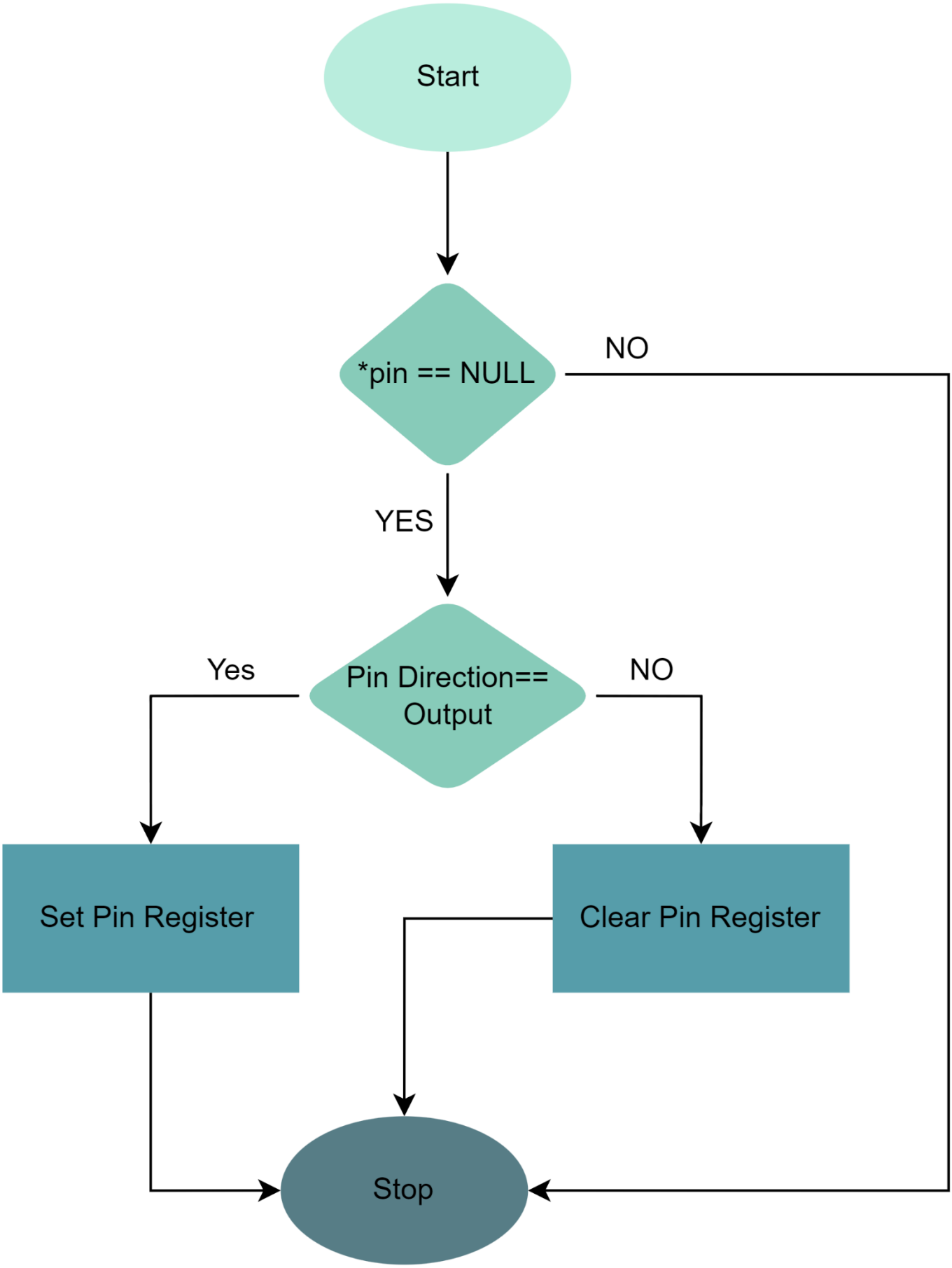
Description: Initializes the direction of a GPIO pin on a micro-controller based on the pin configuration specified in the input parameter.

Parameters: _pin_config: A pointer to an instance of the ST_pin_config_t struct that contains the pin number, port number, pin logic on the pin and desired direction of the GPIO pin.

Return Type: Std_ReturnType. This is a standard type used in AUTOSAR (Automotive Open System Architecture) software development to indicate the success or failure of a function call.
The possible return values for this function are:
E_OK ：The function has completed successfully.
E_NOT_OK: The function has encountered an error and could not complete successfully.

Overall, the GPIO_pin_direction_intialize function provides a way to initialize the direction of a GPIO pin based on its configuration, allowing the software to set the direction of the pin as needed for its specific functionality.
*/

Std_ReturnType GPIO_pin_direction_intialize(const ST_pin_config_t *_pin_config);


Start
*pin == NULL
NO
YES
Pin Direction==
Output
Yes
NO
Set Pin Register
Clear Pin Register
Stop

/*
Function: GPIO_pin_get_direction_status

Description: Gets the current direction status of a GPIO (General Purpose Input/Output) pin on a micro-controller based on the pin configuration specified in the input parameter.

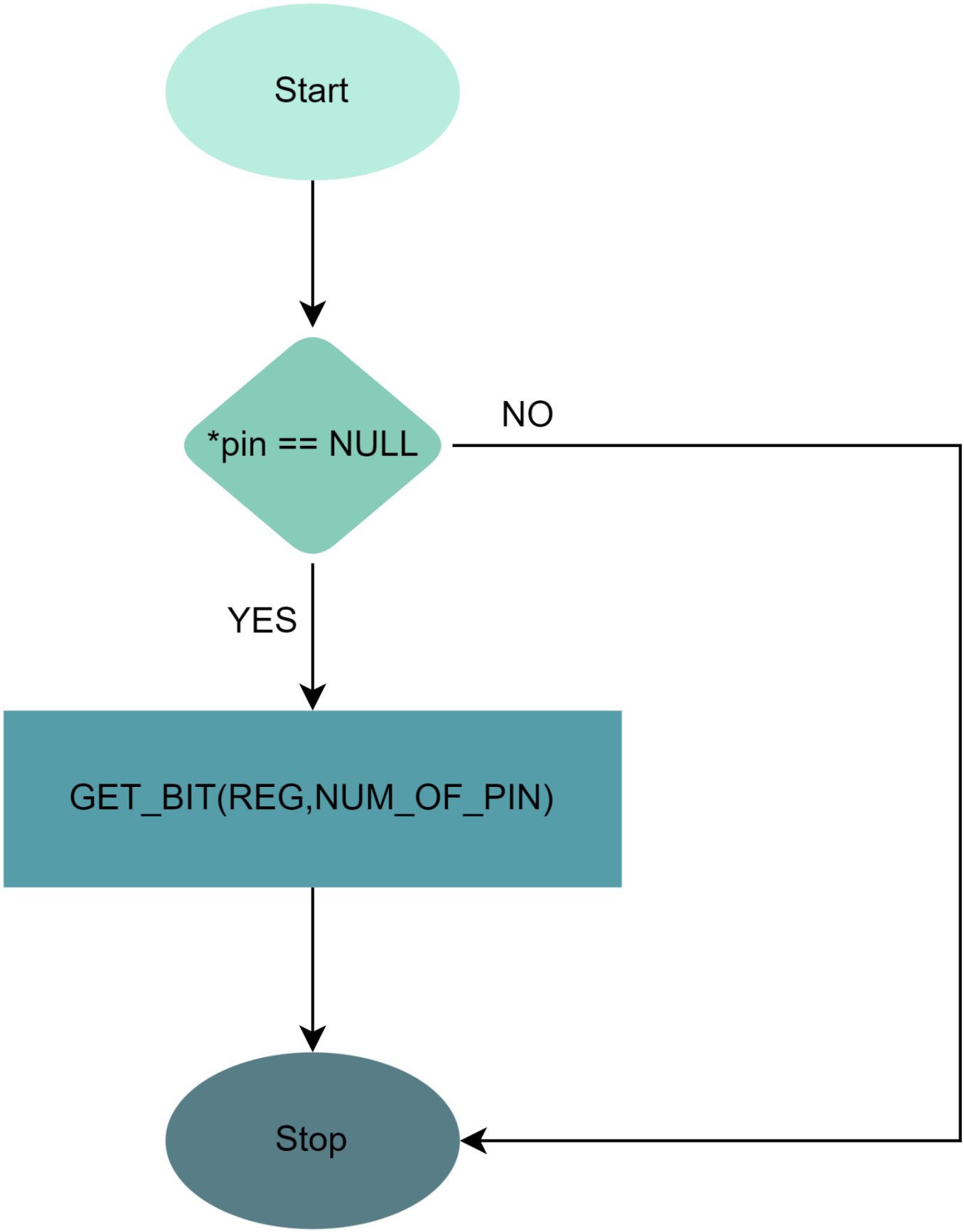
Parameters:
- _pin_config : A pointer to an instance of the ST_pin_config_t struct that contains the pin number, port number,pin logic and desired direction of the GPIO pin.
- direction_status : A pointer to a variable of type EN_direction_t where the current direction status of the GPIO pin will be stored.

Return Type : Std_ReturnType. This is a standard type used in AUTOSAR (Automotive Open System Architecture) software development to indicate the success or failure of a function call.

The possible return values for this function are:
- E_OK : The function has completed successfully.
- E_NOT_OK : The function has encountered an error and could not complete successfully.

Overall, the GPIO_pin_get_direction_status function provides a way to retrieve the current direction status of a GPIO pin based on its configuration, allowing the software to check the pin's direction as needed.
*/

Std_ReturnType GPIO_pin_get_direction_status(const ST_pin_config_t *_pin_config , EN_direction_t *direction_status);


Start
*pin == NULL
NO
YES
GET_BIT(REG,NUM_OF_PIN)
Stop

/*
Function: GPIO_pin_write_logic

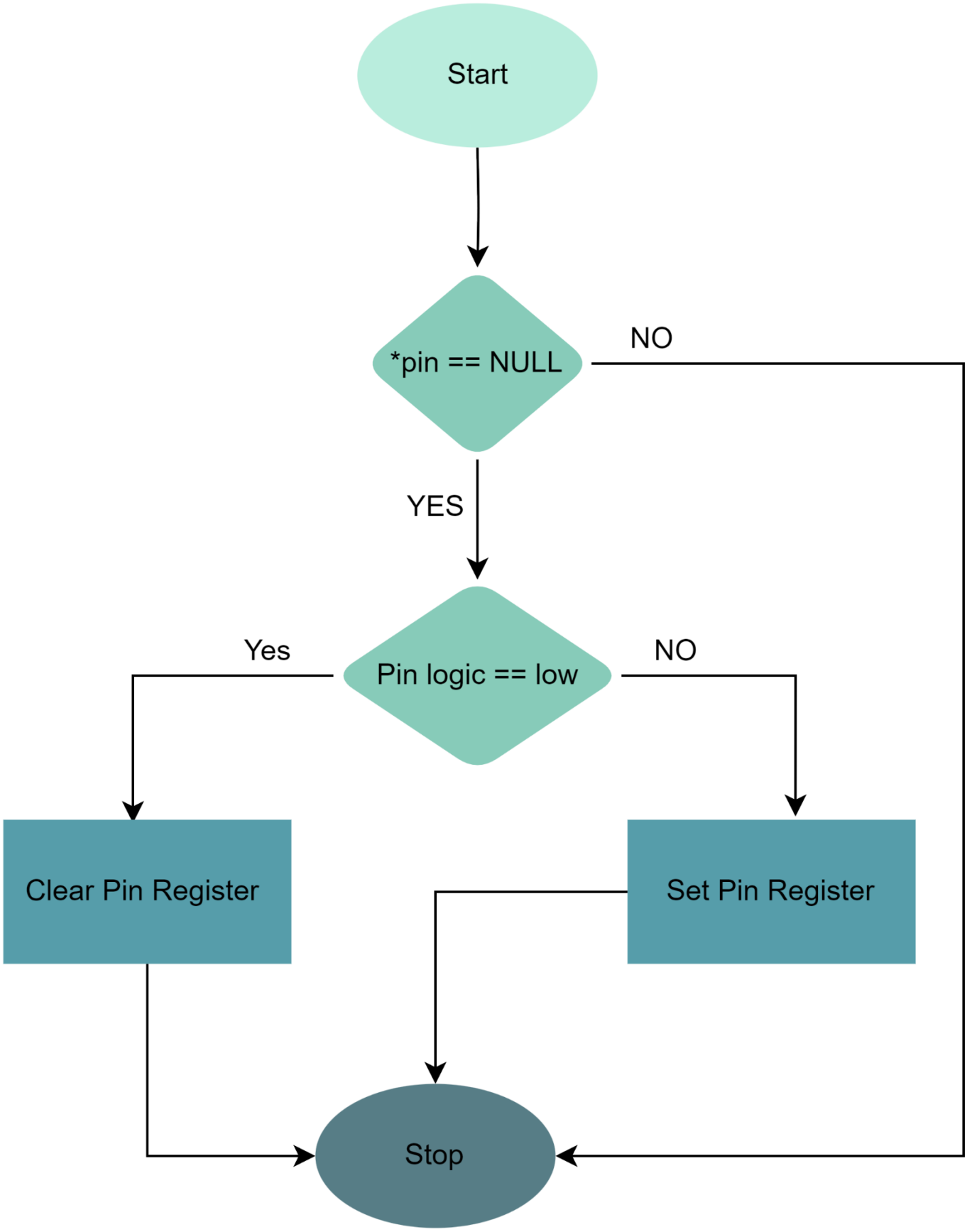
Description : Writes the logic level of a GPIO (General Purpose Input/Output) pin on a micro-controller based on the pin configuration and desired logic level specified in the input parameters.

Parameters:
- _pin_config : A pointer to an instance of the ST_pin_config_t struct that contains the pin number, port number, pin logic and current direction of the GPIO pin.
- logic : The desired logic level to be written to the GPIO pin, either GPIO_LOGIC_LOW or GPIO_LOGIC_HIGH.

Return Type : Std_ReturnType. This is a standard type used in AUTOSAR (Automotive Open System Architecture) software development to indicate the success or failure of a function call.
The possible return values for this function are:
- E_OK : The function has completed successfully.
- E_NOT_OK : The function has encountered an error and could not complete successfully.

Overall, the GPIO_pin_write_logic function provides a way to write a logic level to a GPIO pin based on its configuration, allowing the software to set the output of the pin as needed for its specific functionality.
*/

Std_ReturnType GPIO_pin_write_logic(const ST_pin_config_t *_pin_config , EN_logic_t logic);


Start
*pin == NULL
NO
YES
Yes
Pin logic == low
NO
Clear Pin Register
Set Pin Register
Stop

/*
Function: GPIO_pin_read_logic

Description: Reads the current logic level of a GPIO (General Purpose Input/Output) pin on a micro-controller based on the pin configuration specified in the input parameter.

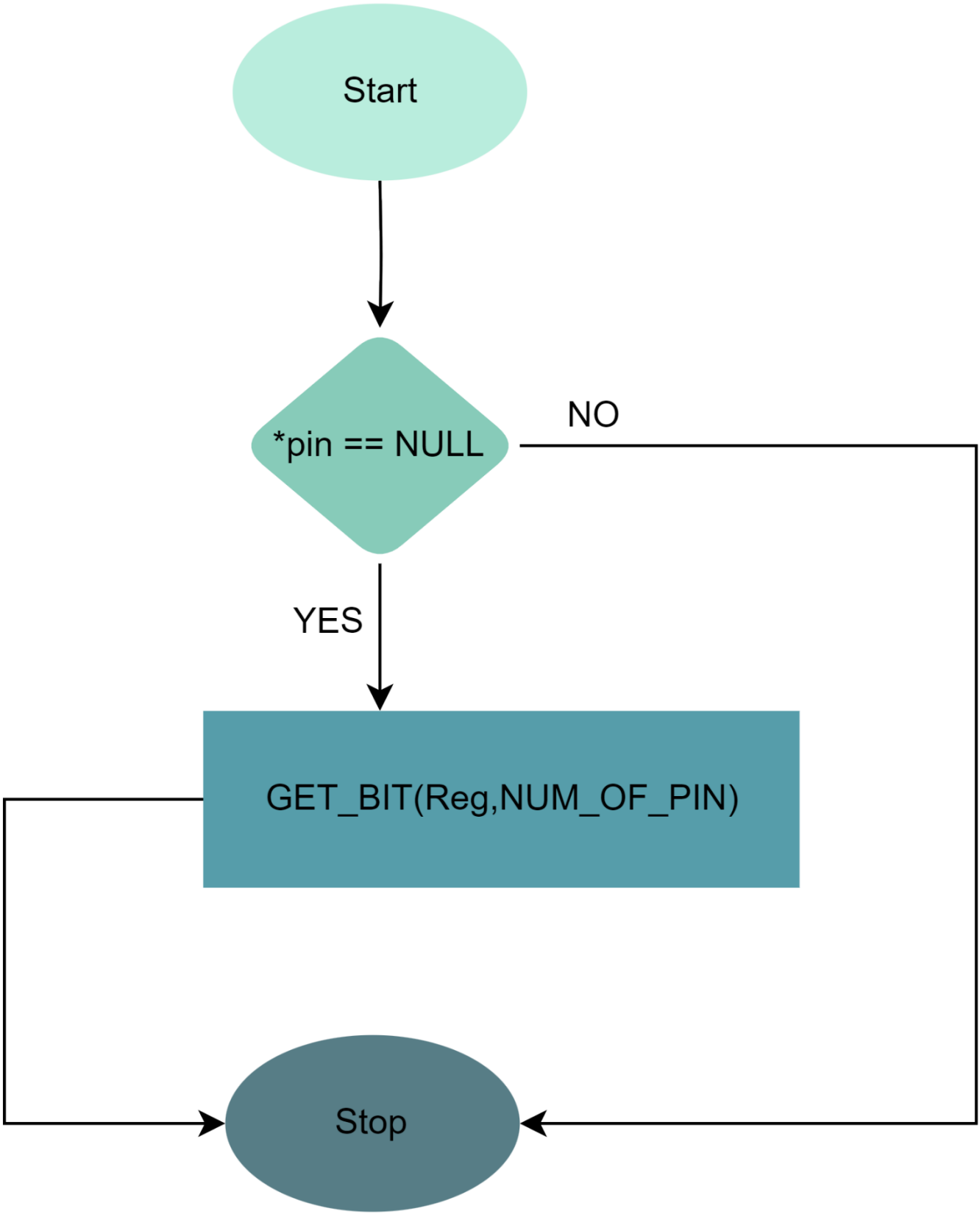
Parameters:
- _pin_config : A pointer to an instance of the ST_pin_config_t struct that contains the pin number, port number, pin logic and current direction of the GPIO pin.

- logic_status : A pointer to a variable of type EN_logic_t where the current logic level of the GPIO pin will be stored.

Return Type: Std_ReturnType. This is a standard type used in AUTOSAR (Automotive Open System Architecture) software development to indicate the success or failure of a function call.
The possible return values for this function are:
- E_OK : The function has completed successfully.
- E_NOT_OK : The function has encountered an error and could not complete successfully.

Overall, the GPIO_pin_read_logic function provides a way to read the current logic level of a GPIO pin based on its configuration, allowing the software to check the input of the pin as needed for its specific functionality The current logic level is stored in the logic_status parameter.
*/

Std_ReturnType GPIO_pin_read_logic(const ST_pin_config_t *_pin_config , EN_logic_t *logic_status);


Start
*pin == NULL
NO
YES
GET_BIT(Reg,NUM_OF_PIN)
Stop

/*
Function: GPIO_pin_toggle_logic

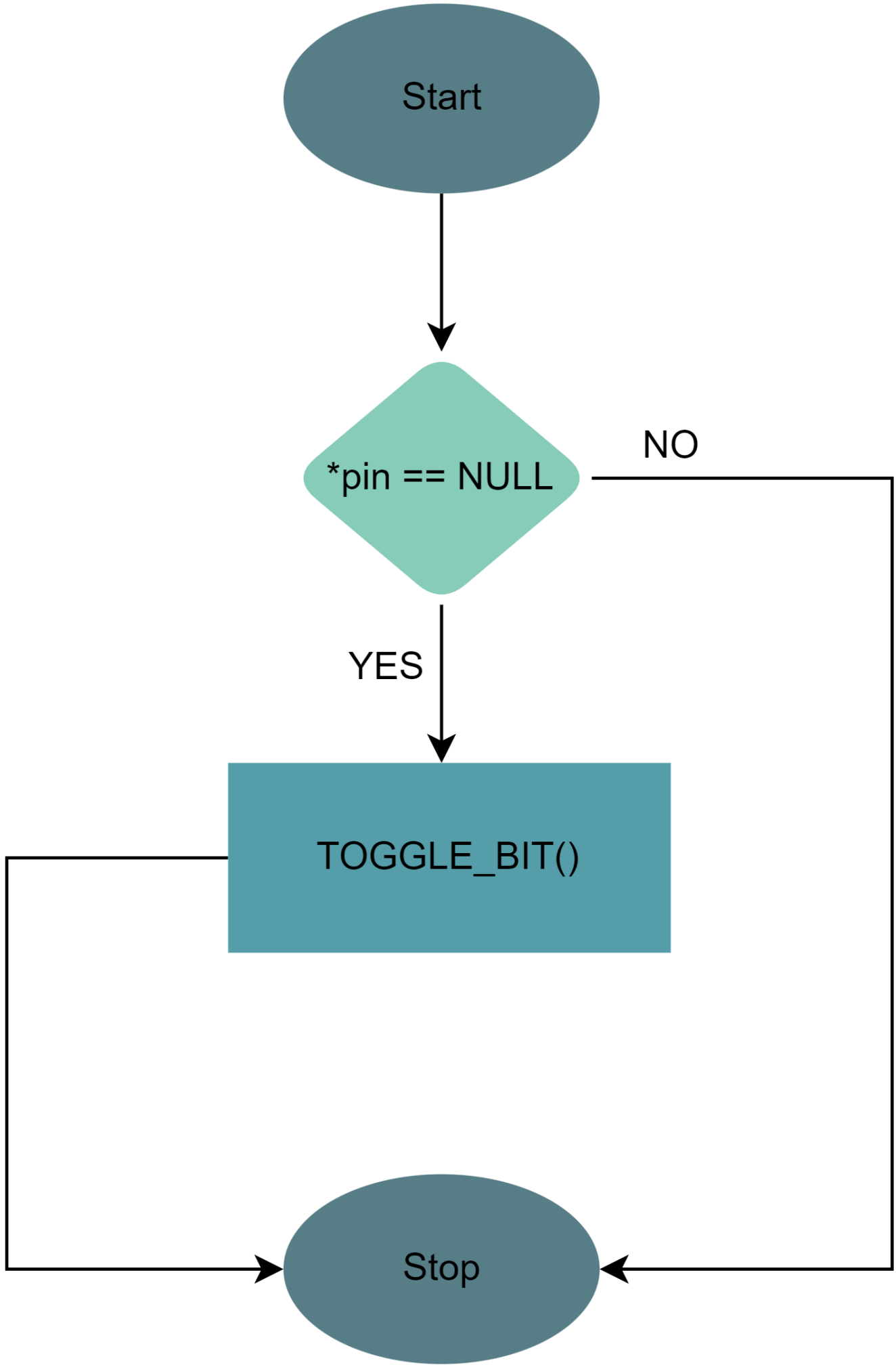
Description: Toggles the logic level of a GPIO (General Purpose Input/Output) pin on a micro-controller based on the pin configuration specified in the input parameter.

Parameters:
- _pin_config : A pointer to an instance of the ST_pin_config_t struct that contains the pin number, port number, pin logic and current direction of the GPIO pin.

Return Type : Std_ReturnType. This is a standard type used in AUTOSAR (Automotive Open System Architecture) software development to indicate the success or failure of a function call.
The possible return values for this function are:
- E_OK : The function has completed successfully.
- E_NOT_OK : The function has encountered an error and could not complete successfully.

Overall, the GPIO_pin_toggle_logic function provides a way to toggle the logic level of a GPIO pin based on its configuration, allowing the software to change the output of the pin between high and low states as needed for its specific functionality.
*/

Std_ReturnType GPIO_pin_toggle_logic(const ST_pin_config_t *_pin_config);


Start
*pin == NULL
NO
YES
TOGGLE_BIT()
Stop

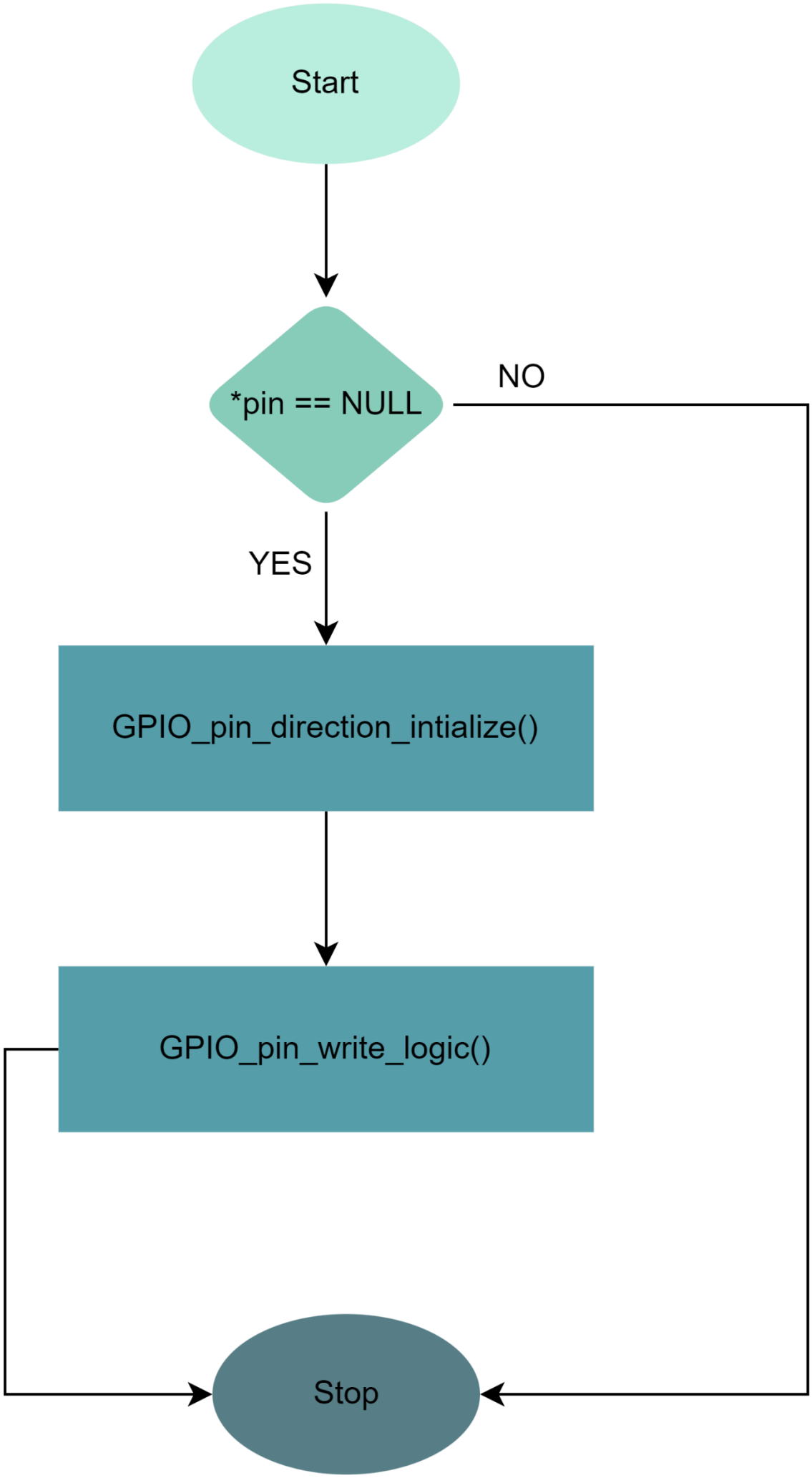
/*
Function: GPIO_pin_intialize

Description : Initializes a GPIO (General Purpose Input/Output) pin on a micro-controller based on the pin configuration specified in the input parameter.

Parameters:
- _pin_config : A pointer to an instance of the ST_pin_config_t struct that contains the pin number, port number, direction, and initial logic level of the GPIO pin.

Return Type : Std_ReturnType. This is a standard type used in AUTOSAR (Automotive Open System Architecture) software development to indicate the success or failure of a function call.
The possible return values for this function are:
- E_OK : The function has completed successfully.
- E_NOT_OK : The function has encountered an error and could not complete successfully.

Overall, the GPIO_pin_intialize function provides a way to initialize a GPIO pin based on its configuration, allowing the software to set the initial direction and logic level of the pin as needed for its specific functionality.
*/
Std_ReturnType GPIO_pin_intialize(const ST_pin_config_t *_pin_config);


Start
*pin == NULL
NO
YES
GPIO_pin_direction_intialize()
GPIO_pin_write_logic()
Stop

2 - External_Interrupt

/*
Enum: EN_MCUCR_REG_BITS

Description: An enumeration that defines the bit fields for the `MCUCR` register on a micro-controller.

Members:
- MCUCR_REG_ISC00_BITS : Represents the bit field for the `ISC00` bit of the `MCUCR` register.
- MCUCR_REG_ISC01_BITS : Represents the bit field for the `ISC01` bit of the `MCUCR` register.
- MCUCR_REG_ISC10_BITS : Represents the bit field for the `ISC10` bit of the `MCUCR` register.
- MCUCR_REG_ISC11_BITS : Represents the bit field for the `ISC11` bit of the `MCUCR` register.

Overall, the EN_MCUCR_REG_BITS enumeration provides a way to represent and manage the individual bit fields within the `MCUCR` register on a micro-controller in a standardized and easy-to-understand manner.By using this enumeration, the software can read and modify the individual bits within this register as needed for interrupt configuration and other purposes.
*/

```
typedef enum
{
	MCUCR_REG_ISC00_BITS = 0,
	MCUCR_REG_ISC01_BITS,
	MCUCR_REG_ISC10_BITS,
	MCUCR_REG_ISC11_BITS

}EN_MCUCR_REG_BITS;
```

/*
Enum: EN_MCUCSR_REG_BITS

Description: An enumeration that defines the bit fields for the `MCUCSR` register on a micro-controller.

Members:
- MCUCSR_REG_ISC2_BITS : Represents the bit field for the `ISC2` bit of the `MCUCSR` register.

Overall, the EN_MCUCSR_REG_BITS enumeration provides a way to represent and manage the individual bit fields within the `MCUCSR` register on a micro-controller in a standardized and easy-to-understand manner. By using this enumeration, the software can read and modify the individual bits within this register as needed for interrupt configuration and other purposes.
*/

```
typedef enum
{
	MCUCSR_REG_ISC2_BITS = 6,

}EN_MCUCSR_REG_BITS;
```

/*
Enum: EN_GICR_REG_BITS

Description: An enumeration that defines the bit fields for the `GICR` register on a micro-controller.

Members:
- GICR_REG_INT2_BITS : Represents the bit field for the `INT2` bit of the `GICR` register.
- GICR_REG_INT0_BITS : Represents the bit field for the `INT0` bit of the `GICR` register.
- GICR_REG_INT1_BITS : Represents the bit field for the `INT1` bit of the `GICR` register.

Overall, the EN_GICR_REG_BITS enumeration provides a way to represent and manage the individual bit fields within the `GICR` register on a micro-controller in a standardized and easy-to-understand manner. By using this enumeration, the software can read and modify the individual bits within this register as needed for interrupt configuration and other purposes.
*/

```
typedef enum
{
	GICR_REG_INT2_BITS = 5,
	GICR_REG_INT0_BITS,
	GICR_REG_INT1_BITS

}EN_GICR_REG_BITS;
```

/*
Enum: EN_GIFR_REG_BITS

Description: An enumeration that defines the bit fields for the `GIFR` register on a micro-controller.

Members:
- GIFR_REG_INTF2_BITS : Represents the bit field for the `INTF2` bit of the `GIFR` register.
- GIFR_REG_INTF0_BITS : Represents the bit field for the `INTF0` bit of the `GIFR` register.
- GIFR_REG_INTF1_BITS : Represents the bit field for the `INTF1` bit of the `GIFR` register.

Overall, the EN_GIFR_REG_BITS enumeration provides a way to represent and manage the individual bit fields within the `GIFR` register on a micro-controller in a standardized and easy-to-understand manner. By using this enumeration, the software can read and modify the individual bits within this register as needed for interrupt handling and other purposes.
*/

```
typedef enum
{
	GIFR_REG_INTF2_BITS = 5,
	GIFR_REG_INTF0_BITS,
	GIFR_REG_INTF1_BITS

}EN_GIFR_REG_BITS;
```

/*
Enum: EN_EXT_INTERRUPT_Sense_Control

Description: An enumeration that defines the possible sense control modes for external interrupts on a micro-controller.

Members:
- LOW_LEVEL_SENSE_CONTROL : Represents the sense control mode where the interrupt is triggered when the input signal is at a low level.
- ANY_LOGICAL_SENSE_CONTROL : Represents the sense control mode where the interrupt is triggered when there is any change in the logical state of the input signal.
- FALLING_EDGE_SENSE_CONTROL : Represents the sense control mode where the interrupt is triggered when the input signal changes from a high level to a low level.

- RISING_EDGE_SENSE_CONTROL : Represents the sense control mode where the interrupt is triggered when the input signal changes from a low level to a high level.

Overall, the EN_EXT_INTERRUPT_Sense_Control enumeration provides a way to represent and manage the different sense control modes for external interrupts on a micro-controller in a standardized and easy-to-understand manner. By using this enumeration, the software can configure and handle external interrupts based on the desired sense control mode for the specific input signal being used.
*/

```
typedef enum
{
	LOW_LEVEL_SENSE_CONTROL = 0,
	ANY_LOGICAL_SENSE_CONTROL,
	FALLING_EDGE_SENSE_CONTROL,
	RISING_EDGE_SENSE_CONTROL

}EN_EXT_INTERRUPT_Sense_Control;
```

/*
Enum: EN_EXT_INTERRUPTS

Description: An enumeration that defines the external interrupts available on a micro-controller.

Members:
- EXT0_INTERRUPTS : Represents external interrupt 0.
- EXT1_INTERRUPTS : Represents external interrupt 1.
- EXT2_INTERRUPTS : Represents external interrupt 2.

Overall, the EN_EXT_INTERRUPTS enumeration provides a way to represent and manage the available external interrupts on a micro-controller in a standardized and easy-to-understand manner. By using this enumeration, the software can configure and handle external interrupts based on the specific interrupt being used.

*/

```
typedef enum
{
	EXT0_INTERRUPTS = 0,
	EXT1_INTERRUPTS,
	EXT2_INTERRUPTS
}EN_EXT_INTERRUPTS;
```

/*
Struct: ST_EXT_INTERRUPTS_CFG

Description: A structure that contains the configuration settings for an external interrupt on a micro-controller.
Members:
- INTERRUPT_EXTERNAL_HANDLER : A function pointer to the interrupt service routine (ISR) for the external interrupt(call-back function).
- EXTERNAL_INTERRUPRT_Number : An instance of the EN_EXT_INTERRUPTS enum that specifies the external interrupt number to be configured.
- EXTERNAL_INTERRUPRT_Sense_Control : An instance of the EN_EXT_INTERRUPT_Sense_Control enum that specifies the sense control mode for the external interrupt.

Overall, the ST_EXT_INTERRUPTS_CFG structure provides a way to represent and manage the configuration settings for an external interrupt on a micro-controller in a standardized and easy-to-understand manner. By using this structure, the software can configure and handle external interrupts based on the desired interrupt number and sense control mode, and execute the appropriate ISR when the interrupt is triggered.

*/
typedef struct
{
	void(*INTERRUPT_EXTERNAL_HANDLER)(void);
	EN_EXT_INTERRUPTS EXTERNAL_INTERRUPRT_Number;
	EN_EXT_INTERRUPT_Sense_Control EXTERNAL_INTERRUPRT_Sense_Control;

}ST_EXT_INTERRUPTS_CFG;

/*
Function: EXT_vINTERRUPT_Init

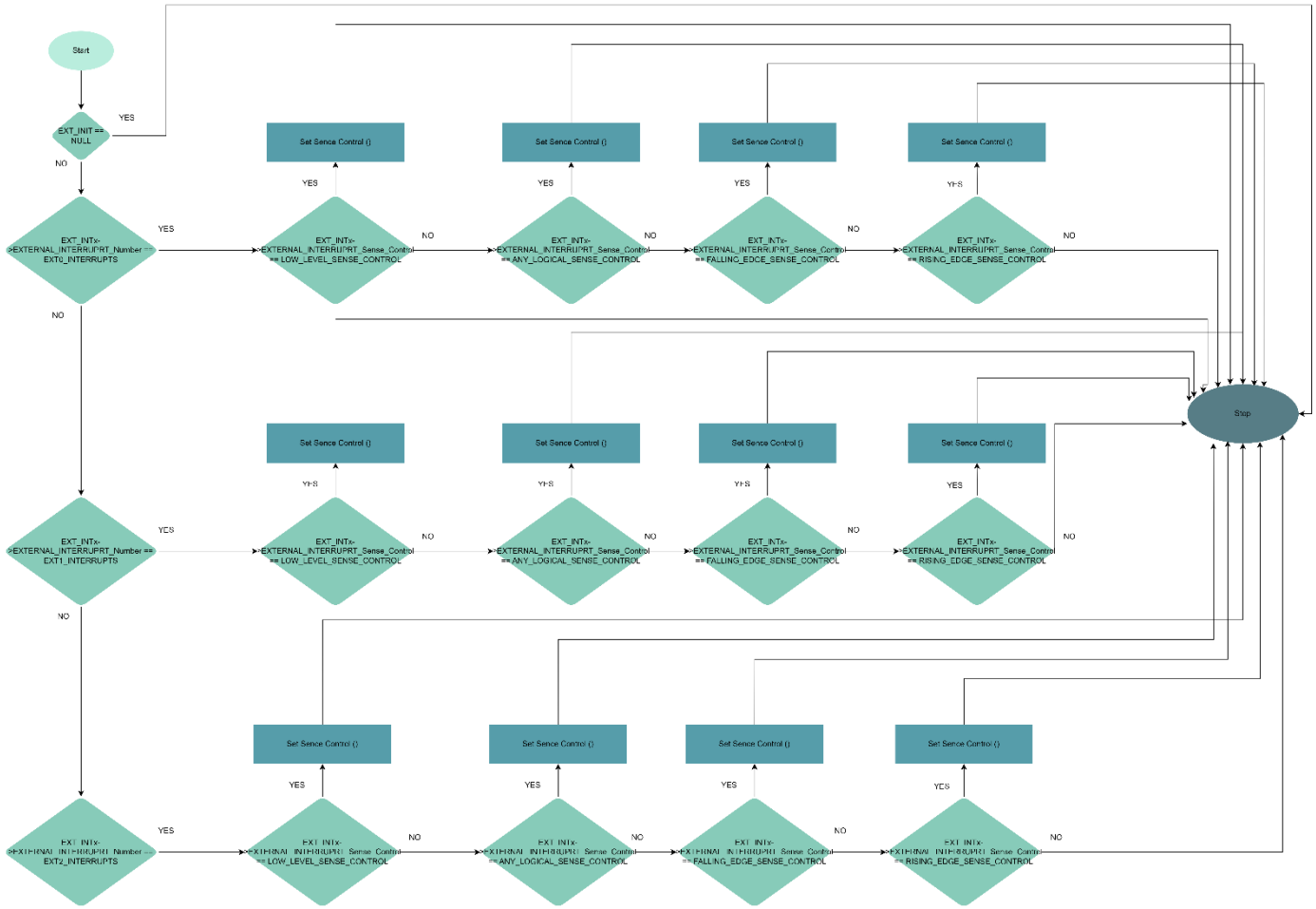
Description: Initializes an external interrupt on a micro-controller with the specified configuration settings.

Parameters:
- EXT_INTx : A pointer to an ST_EXT_INTERRUPTS_CFG struct that contains the configuration settings for the external interrupt.

Return Type: Std_ReturnType. This is a standard type used in AUTOSAR (Automotive Open System Architecture) software development to indicate the success or failure of a function call.
The possible return values for this function are:
- E_OK : The function has completed successfully.
- E_NOT_OK : The function has encountered an error and could not complete successfully.

Overall, the EXT_vINTERRUPT_Init function provides a way to initialize an external interrupt on a micro-controller with the desired configuration settings. By using this function, the software can set up and handle external interrupts based on the specific interrupt number and sense control mode, and execute the appropriate ISR when the interrupt is triggered.
*/

Std_ReturnType EXT_vINTERRUPT_Init(const ST_EXT_INTERRUPTS_CFG *EXT_INTx);


Start
EXT_INIT == NULL
YES
NO
EXT_INTx->EXTERNAL_INTERRUPT_Number == EXT0_INTERRUPTS
EXT_INTx->EXTERNAL_INTERRUPRT_Sense_Control == LOW_LEVEL_SENSE_CONTROL
EXT_INTx->EXTERNAL_INTERRUPRT_Sense_Control == ANY_LOGICAL_SENSE_CONTROL
EXT_INTx->EXTERNAL_INTERRUPRT_Sense_Control == FALLING_EDGE_SENSE_CONTROL
EXT_INTx->EXTERNAL_INTERRUPRT_Sense_Control == RISING_EDGE_SENSE_CONTROL
Set Sence Control ()
EXT_INTx->EXTERNAL_INTERRUPRT_Number == EXT1_INTERRUPTS
EXT_INTx->EXTERNAL_INTERRUPRT_Number == EXT2_INTERRUPTS
Stop

/*
Function: EXT_vINTERRUPT_Denit

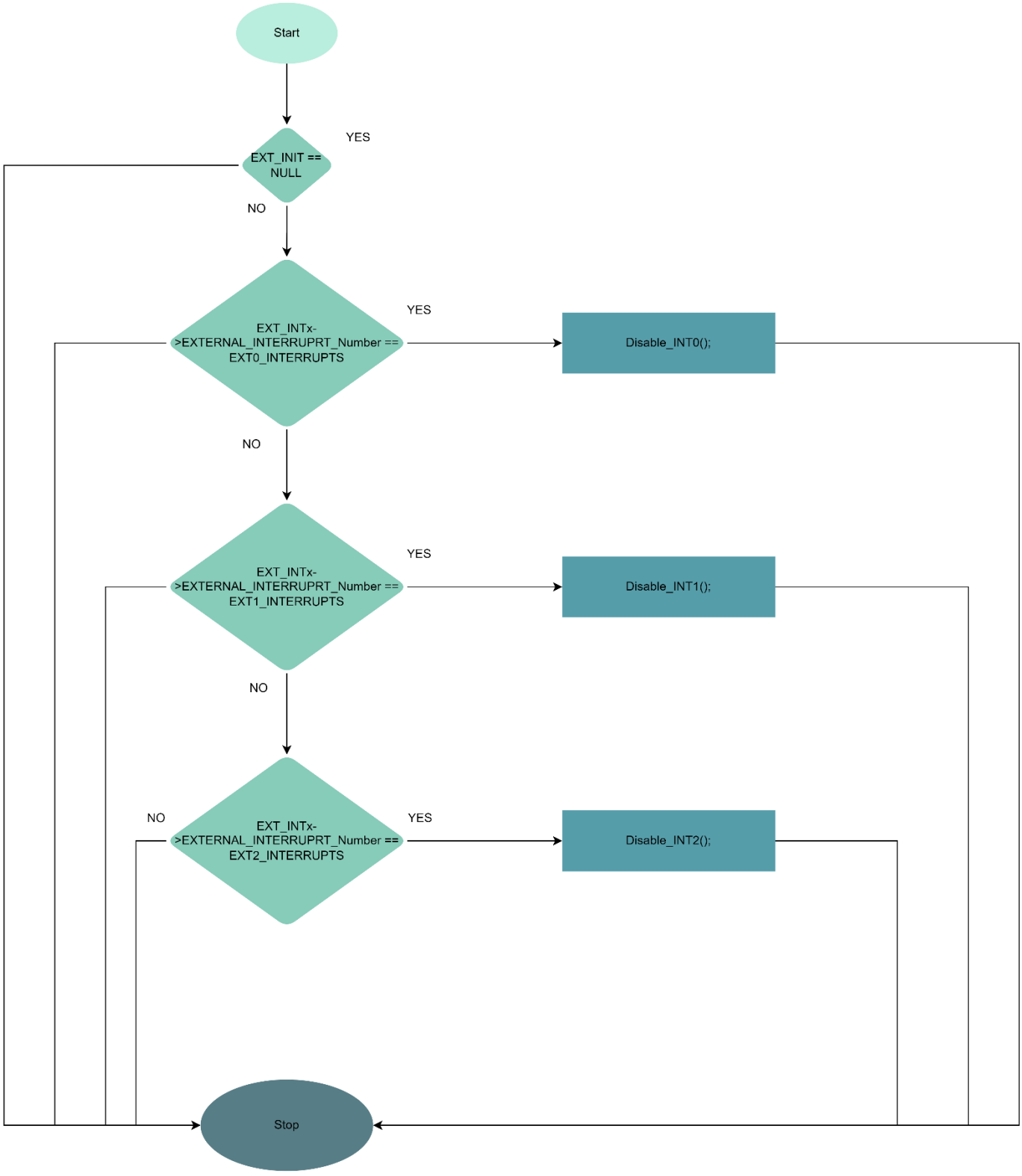
Description: Deinitializes an external interrupt on a micro-controller with the specified configuration settings.

Parameters:
- EXT_INTx : A pointer to an ST_EXT_INTERRUPTS_CFG struct that contains the configuration settings for the external interrupt.

Return Type: Std_ReturnType. This is a standard type used in AUTOSAR (Automotive Open System Architecture) software development to indicate the success or failure of a function call.
The possible return values for this function are:
- E_OK : The function has completed successfully.
- E_NOT_OK : The function has encountered an error and could not complete successfully.

Overall, the EXT_vINTERRUPT_Denit function provides a way to deinitialize an external interrupt on a micro-controller with the desired configuration settings. By using this function, the software can remove the interrupt and associated ISR, freeing up resources and ensuring proper operation of the micro-controller.
*/

Std_ReturnType EXT_vINTERRUPT_Denit(const ST_EXT_INTERRUPTS_CFG *EXT_INTx);


Start
EXT_INIT == NULL
YES
NO
EXT_INTx->EXTERNAL_INTERRUPRT_Number == EXT0_INTERRUPTS
YES
Disable_INT0();
NO
EXT_INTx->EXTERNAL_INTERRUPRT_Number == EXT1_INTERRUPTS
YES
Disable_INT1();
NO
EXT_INTx->EXTERNAL_INTERRUPRT_Number == EXT2_INTERRUPTS
NO
YES
Disable_INT2();
Stop

3 - TIMERS

Timers With All Modes (Normal – Ctc – Pwm - Icu)

/*
Documentation for the Timer Channel Enum:

The enu_TimerChannel typedef enum provides a list of options for different timer channels. The options are as follows:

TIMER0_NORMAL: This option selects normal timer mode for Timer0.
TIMER0_CTC: This option selects Clear Timer on Compare (CTC) mode for Timer0.
TIMER1_NORMAL: This option selects normal timer mode for Timer1.
TIMER1_CTC_OCR: This option selects CTC mode for Timer1, using OCR1A as the comparison value.
TIMER1_CTC_ICR: This option selects CTC mode for Timer1, using ICR1 as the comparison value.
TIMER2_NORMAL: This option selects normal timer mode for Timer2.
TIMER2_CTC: This option selects CTC mode for Timer2.
*/

```
typedef enum{
        TIMER0_NORMAL,
        TIMER0_CTC,
        TIMER1_NORMAL,
        TIMER1_CTC_OCR,
        TIMER1_CTC_ICR,
        TIMER2_NORMAL,
        TIMER2_CTC
}enu_TimerChannel;
```

/*
Documentation for the Prescale Modes Enum:

The enu_prescale_modes typedef enum provides a list of options for different prescale modes. The options are as follows:

TMR_PRE_NO_CLK: This option selects no clock source for the timer, effectively stopping it.
TMR_PRE_0: This option selects the system clock with no prescaling.
TMR_PRE_8: This option selects the system clock divided by 8 as the clock source.
TMR_PRE_64: This option selects the system clock divided by 64 as the clock source.
TMR_PRE_256: This option selects the system clock divided by 256 as the clock source.
TMR_PRE_1024: This option selects the system clock divided by 1024 as the clock source.
TMR_PRE_EXT_FALLING: This option selects an external clock source with falling edge triggering as the clock source.
TMR_PRE_EXT_RISING: This option selects an external clock source with rising edge triggering as the clock source.
TMR_PRE_32: This option selects the system clock divided by 32 as the clock source.
TMR_PRE_128: This option selects the system clock divided by 128 as the clock source.
*/

```
typedef enum{
        TMR_PRE_NO_CLK,
        TMR_PRE_0,
        TMR_PRE_8,
        TMR_PRE_64,
        TMR_PRE_256,
```

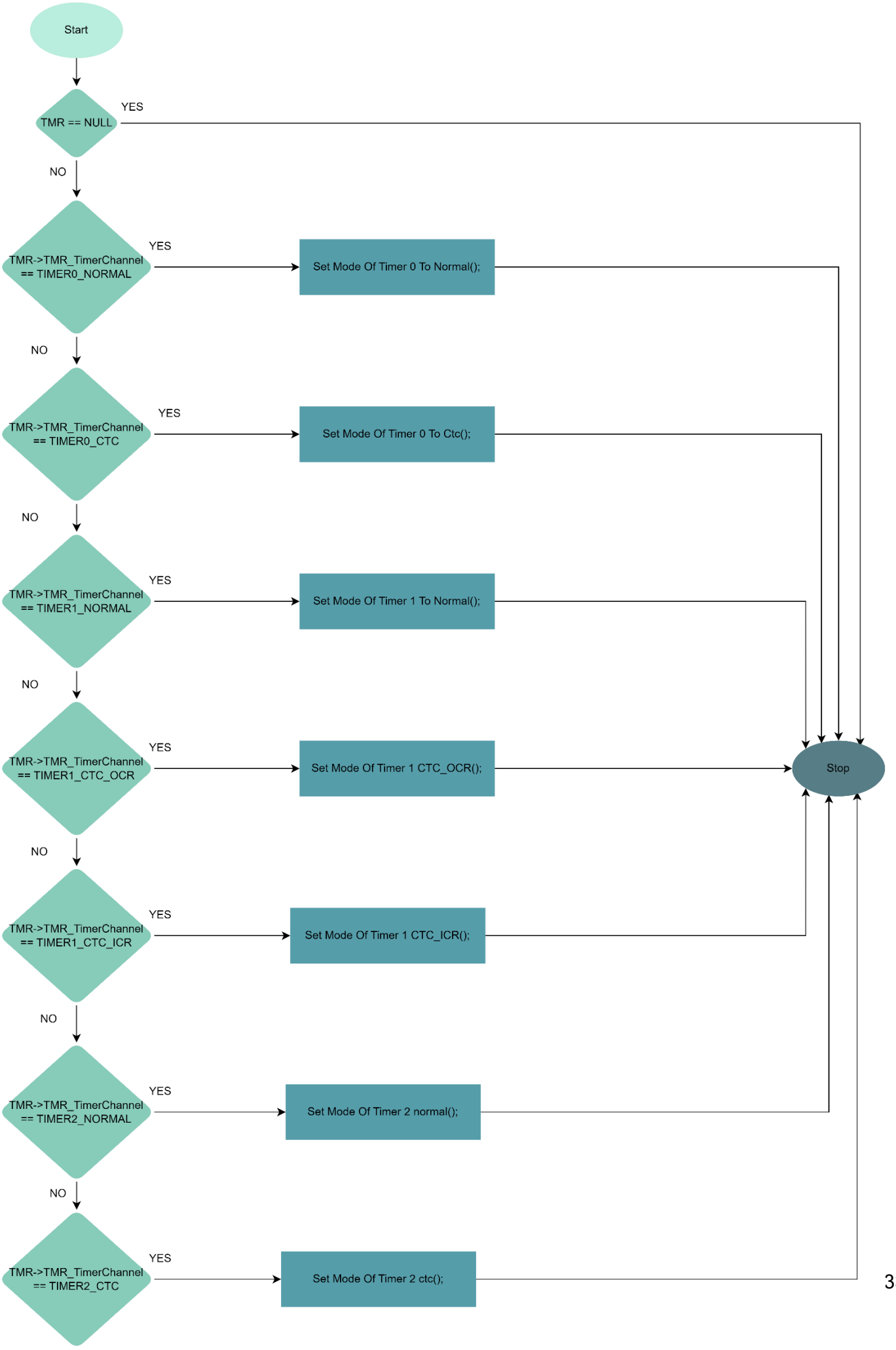
```
        TMR_PRE_1024,
        TMR_PRE_EXT_FALLING,
        TMR_PRE_EXT_RISING,
        TMR_PRE_32,
        TMR_PRE_128
}enu_prescale_modes;

/*
Documentation for the Timer Toggle Mode Enum:

The enu_TimerToggleMode typedef enum provides a list of options for different timer toggle modes. The options are as follows:

TMR_OCMode: This option selects Output Compare (OC) mode for the timer.
TMR_InterruptMode: This option selects Interrupt mode for the timer.
*/
typedef enum{
        TMR_OCMode,
        TMR_InterruptMode
}enu_TimerToggleMode;

/*
Documentation for the Timer Output Compare Mode Enum:

The enu_TimerOCMode typedef enum provides a list of options for different timer output compare modes. The options are as follows:

OC_Disconnected: This option disconnects the output compare pin from the timer.
OC_Toggle: This option toggles the output compare pin on a compare match.
OC_Clear: This option clears the output compare pin on a compare match.
OC_Set: This option sets the output compare pin on a compare match.
*/
typedef enum{
        OC_Disconnected,
        OC_Toggle,
        OC_Clear,
        OC_Set
}enu_TimerOCMode;
```

/*
Function Description: The TMR_vInit function initializes and sets up a timer with the settings specified in the input TMR_cfg_t configuration structure. Once the timer is initialized and set up, it can be used for various timing applications such as measuring elapsed time, generating interrupts at specified intervals, or controlling the timing of other operations in a system.

Function Parameters:
- TMR: a pointer to a TMR_cfg_t structure containing the desired timer channel, prescale mode, toggle mode, and output compare mode settings.

Function Return Type: void
*/

void TMR_vInit(const TMR_cfg_t *TMR);

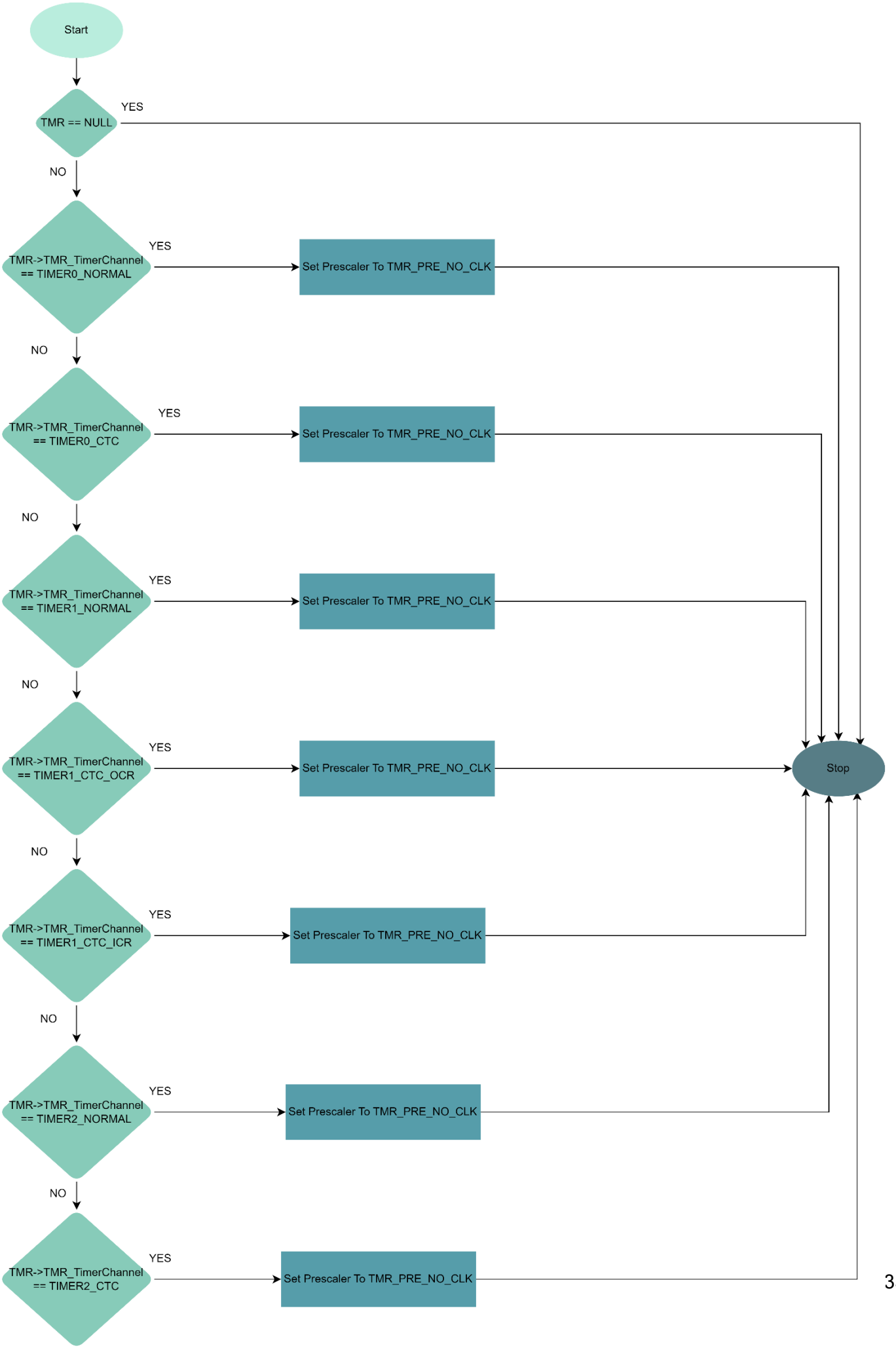

Start
TMR == NULL
YES
NO
TMR->TMR_TimerChannel == TIMER0_NORMAL
YES
Set Mode Of Timer 0 To Normal();
NO
TMR->TMR_TimerChannel == TIMER0_CTC
YES
Set Mode Of Timer 0 To Ctc();
NO
TMR->TMR_TimerChannel == TIMER1_NORMAL
YES
Set Mode Of Timer 1 To Normal();
NO
TMR->TMR_TimerChannel == TIMER1_CTC_OCR
YES
Set Mode Of Timer 1 CTC_OCR();
NO
TMR->TMR_TimerChannel == TIMER1_CTC_ICR
YES
Set Mode Of Timer 1 CTC_ICR();
NO
TMR->TMR_TimerChannel == TIMER2_NORMAL
YES
Set Mode Of Timer 2 normal();
NO
TMR->TMR_TimerChannel == TIMER2_CTC
YES
Set Mode Of Timer 2 ctc();
Stop

/*
Function Name: TMR_vStop

Function Description: The TMR_vStop function is used to stop a timer that is configured with the settings specified in the input TMR_cfg_t configuration structure.

Function Parameters:
- TMR: a pointer to a TMR_cfg_t structure containing the settings for the timer to be stopped.

Function Return Type: void
*/
void TMR_vStop(const TMR_cfg_t *TMR);

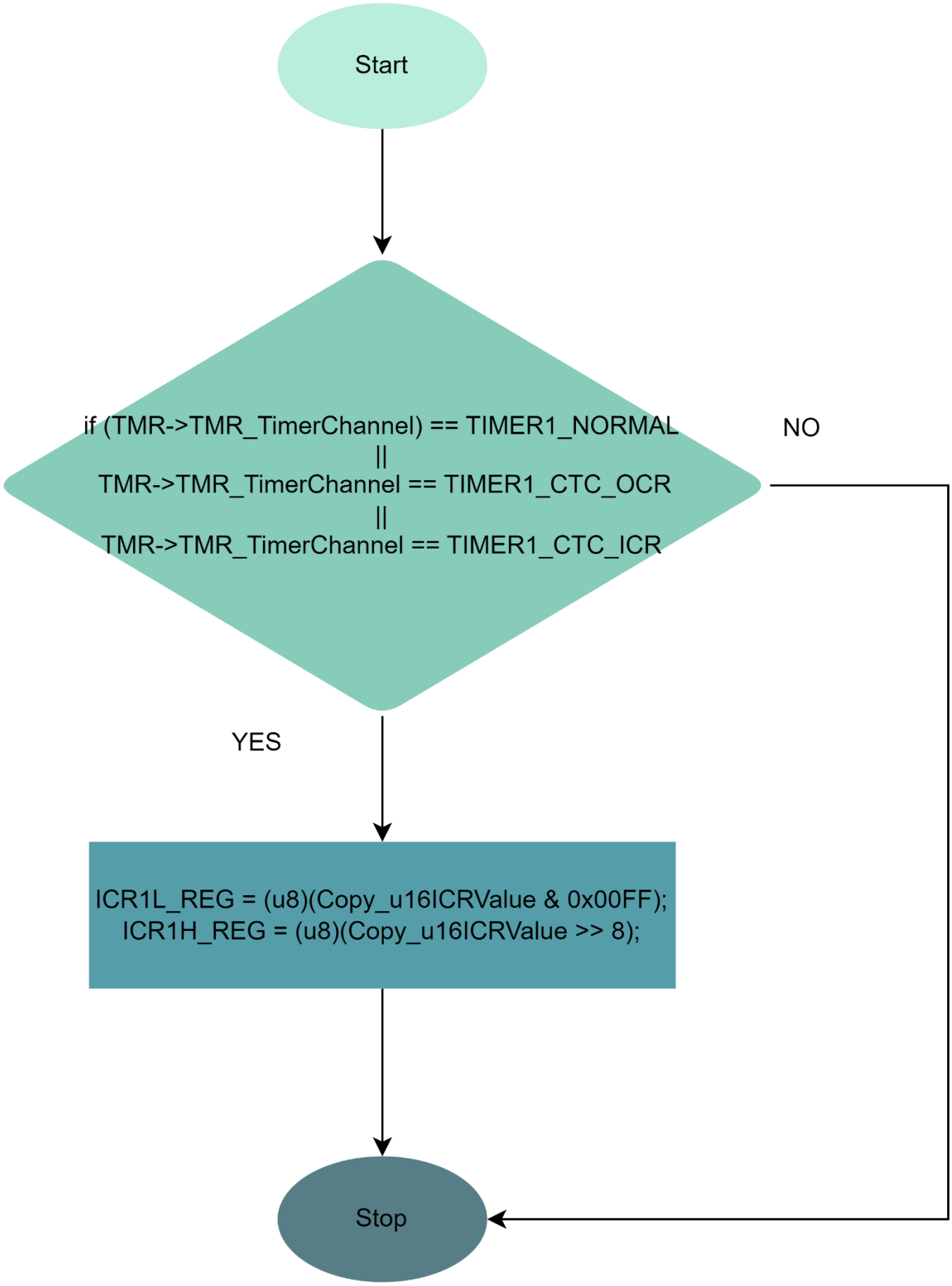

Start
TMR == NULL
YES
NO
TMR->TMR_TimerChannel == TIMER0_NORMAL
YES
Set Prescaler To TMR_PRE_NO_CLK
NO
TMR->TMR_TimerChannel == TIMER0_CTC
YES
Set Prescaler To TMR_PRE_NO_CLK
NO
TMR->TMR_TimerChannel == TIMER1_NORMAL
YES
Set Prescaler To TMR_PRE_NO_CLK
NO
TMR->TMR_TimerChannel == TIMER1_CTC_OCR
YES
Set Prescaler To TMR_PRE_NO_CLK
NO
TMR->TMR_TimerChannel == TIMER1_CTC_ICR
YES
Set Prescaler To TMR_PRE_NO_CLK
NO
TMR->TMR_TimerChannel == TIMER2_NORMAL
YES
Set Prescaler To TMR_PRE_NO_CLK
NO
TMR->TMR_TimerChannel == TIMER2_CTC
YES
Set Prescaler To TMR_PRE_NO_CLK
Stop

/*
Function Name: TMR_vSetICRValue

Function Description: The TMR_vSetICRValue function sets the Input Capture Register (ICR) value for the timer that is configured with the settings specified in the input TMR_cfg_t configuration structure.

Function Parameters:

TMR: a pointer to a TMR_cfg_t structure containing the settings for the timer to be configured.
Copy_u16ICRValue: a 16-bit unsigned integer value that specifies the value to which the ICR register will be set.
Function Return Type: void
*/
void TMR_vSetICRValue(const TMR_cfg_t *TMR, u16 Copy_u16ICRValue);

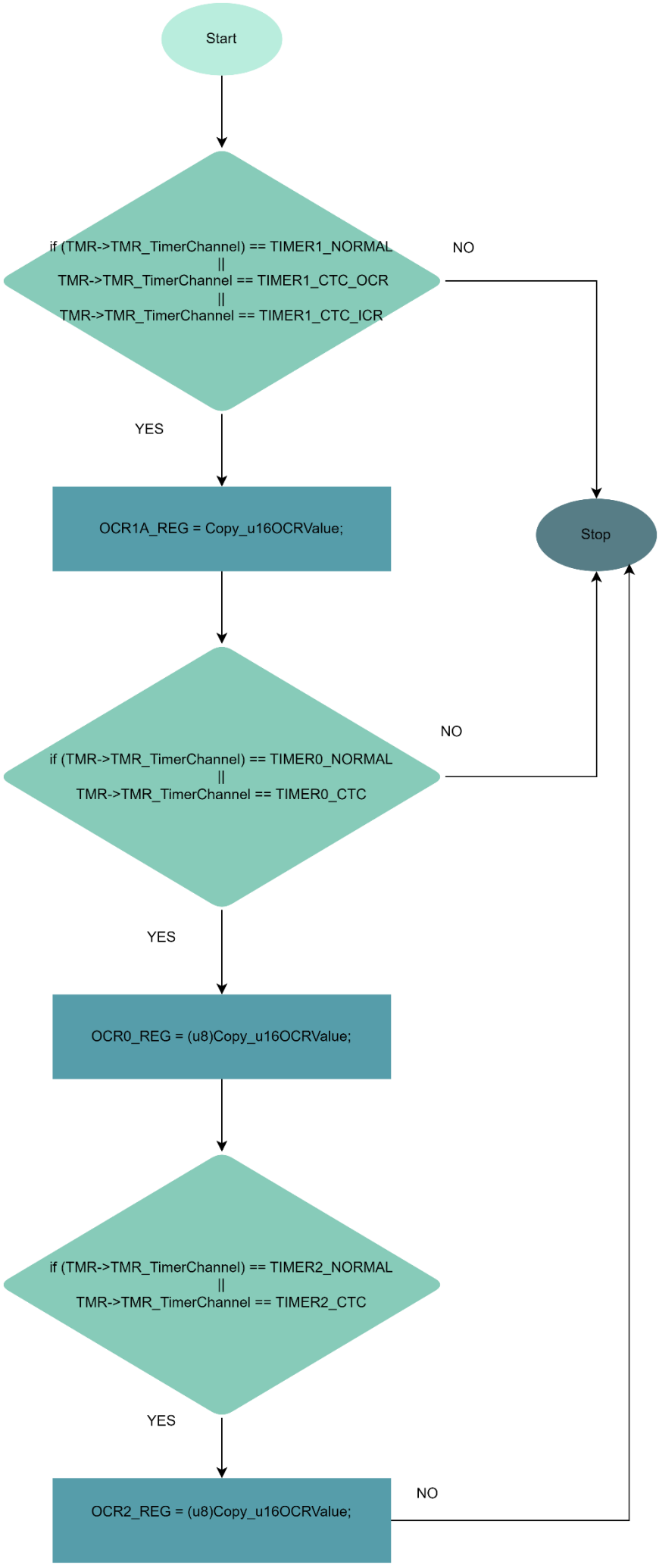

Start
if (TMR->TMR_TimerChannel) == TIMER1_NORMAL
||
TMR->TMR_TimerChannel == TIMER1_CTC_OCR
||
TMR->TMR_TimerChannel == TIMER1_CTC_ICR
NO
YES
ICR1L_REG = (u8)(Copy_u16ICRValue & 0x00FF);
ICR1H_REG = (u8)(Copy_u16ICRValue >> 8);
Stop

/*
Function Name: TMR_vSetOCRValue

Function Description: The TMR_vSetOCRValue function sets the Output Compare Register (OCR) value for the timer that is configured with the settings specified in the input TMR_cfg_t configuration structure.

Function Parameters:

TMR: a pointer to a TMR_cfg_t structure containing the settings for the timer to be configured.
Copy_u16OCRValue: a 16-bit unsigned integer value that specifies the value to which the OCR register will be set.
Function Return Type: void
*/
void TMR_vSetOCRValue(const TMR_cfg_t *TMR, u16 Copy_u16OCRValue);

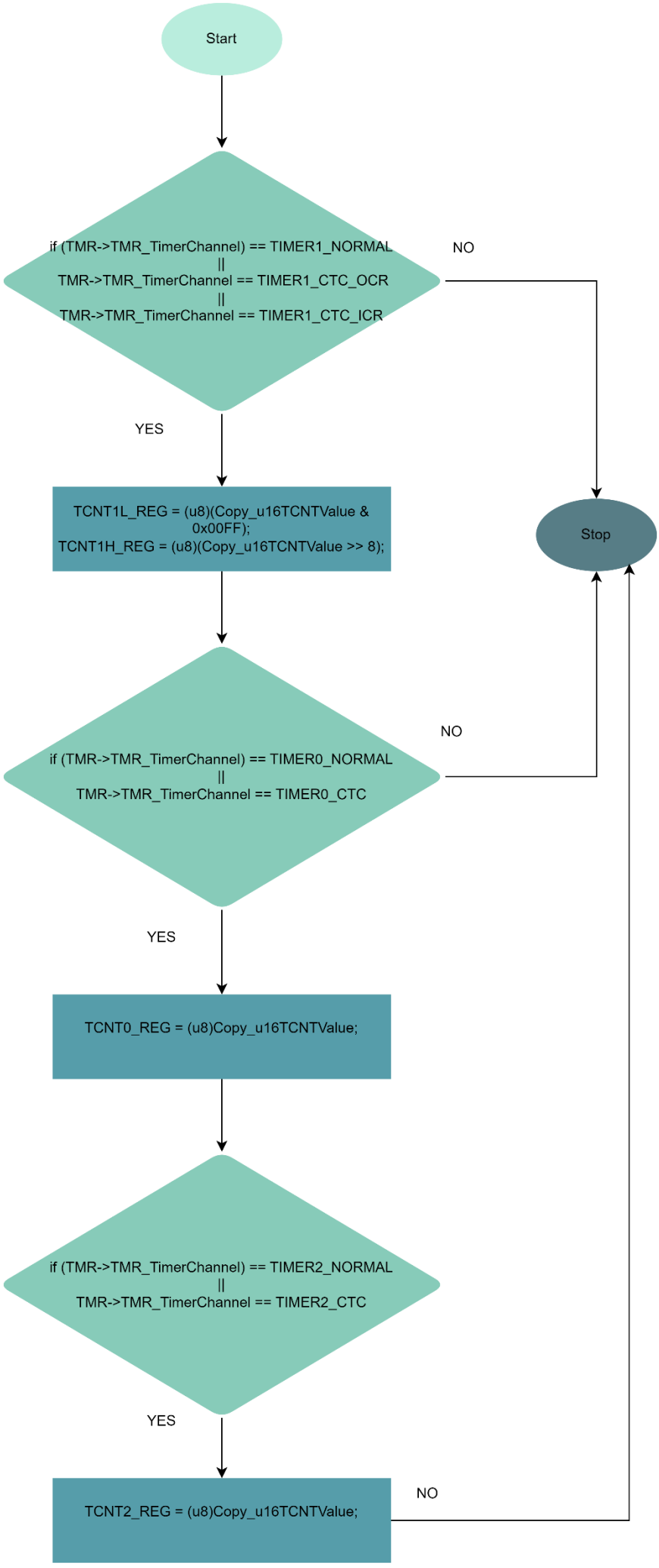

Start
if (TMR->TMR_TimerChannel) == TIMER1_NORMAL
||
TMR->TMR_TimerChannel == TIMER1_CTC_OCR
||
TMR->TMR_TimerChannel == TIMER1_CTC_ICR
NO
YES
OCR1A_REG = Copy_u16OCRValue;
Stop
if (TMR->TMR_TimerChannel) == TIMER0_NORMAL
||
TMR->TMR_TimerChannel == TIMER0_CTC
NO
YES
OCR0_REG = (u8)Copy_u16OCRValue;
if (TMR->TMR_TimerChannel) == TIMER2_NORMAL
||
TMR->TMR_TimerChannel == TIMER2_CTC
YES
OCR2_REG = (u8)Copy_u16OCRValue;
NO

/*
Function Name: TMR_vSetTCNTValue

Function Description: The TMR_vSetTCNTValue function sets the Timer/Counter Register (TCNT) value for the timer that is configured with the settings specified in the input TMR_cfg_t configuration structure.

Function Parameters:

TMR: a pointer to a TMR_cfg_t structure containing the settings for the timer to be configured.
Copy_u16TCNTValue: a 16-bit unsigned integer value that specifies the value to which the TCNT register will be set.
Function Return Type: void
*/
void TMR_vSetTCNTValue(const TMR_cfg_t *TMR, u16 Copy_u16TCNTValue);

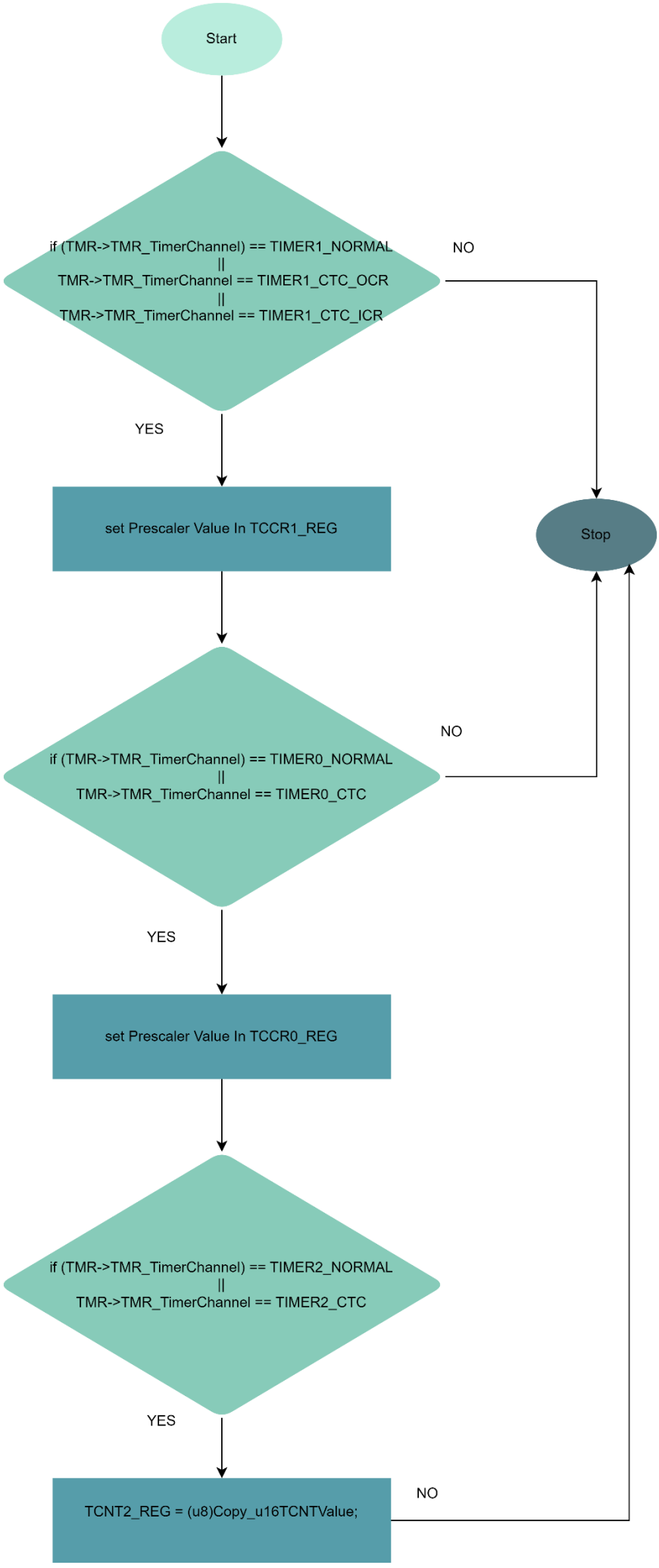

Start
if (TMR->TMR_TimerChannel) == TIMER1_NORMAL
||
TMR->TMR_TimerChannel == TIMER1_CTC_OCR
||
TMR->TMR_TimerChannel == TIMER1_CTC_ICR
NO
YES
TCNT1L_REG = (u8)(Copy_u16TCNTValue &
0x00FF);
TCNT1H_REG = (u8)(Copy_u16TCNTValue >> 8);
Stop
if (TMR->TMR_TimerChannel) == TIMER0_NORMAL
||
TMR->TMR_TimerChannel == TIMER0_CTC
NO
YES
TCNT0_REG = (u8)Copy_u16TCNTValue;
if (TMR->TMR_TimerChannel) == TIMER2_NORMAL
||
TMR->TMR_TimerChannel == TIMER2_CTC
YES
TCNT2_REG = (u8)Copy_u16TCNTValue;
NO

/*
Function Name: TMR_vStartTimer

Function Description: The TMR_vStartTimer function starts the timer that is configured with the settings specified in the input TMR_cfg_t configuration structure.

Function Parameters:

TMR: a pointer to a TMR_cfg_t structure containing the settings for the timer to be started.
Function Return Type: void
*/
void TMR_vStartTimer(const TMR_cfg_t *TMR);


Start
if (TMR->TMR_TimerChannel) == TIMER1_NORMAL
||
TMR->TMR_TimerChannel == TIMER1_CTC_OCR
||
TMR->TMR_TimerChannel == TIMER1_CTC_ICR
NO
YES
set Prescaler Value In TCCR1_REG
Stop
if (TMR->TMR_TimerChannel) == TIMER0_NORMAL
||
TMR->TMR_TimerChannel == TIMER0_CTC
NO
YES
set Prescaler Value In TCCR0_REG
if (TMR->TMR_TimerChannel) == TIMER2_NORMAL
||
TMR->TMR_TimerChannel == TIMER2_CTC
YES
TCNT2_REG = (u8)Copy_u16TCNTValue;
NO

Ecu Layer

1 - Push Button

/*
Enum: EN_PUSH_BTN_state_t

Description : An enumeration that defines two possible states for a push button: pressed or released.

Members:
- PUSH_BTN_STATE_PRESSED : Represents the state of a push button when it is pressed down or activated.
- PUSH_BTN_STATE_RELEASED : Represents the state of a push button when it is not pressed or deactivated.

Overall, the EN_PUSH_BTN_state_t enumeration provides a way to represent the two possible states of a push button in a standardized and easy-to-understand manner. By using this enumeration, the software can check the state of a push button and take appropriate action based on whether it is pressed or released.
*/

```
typedef enum
{
   PUSH_BTN_STATE_PRESSED = 0,
   PUSH_BTN_STATE_RELEASED
}EN_PUSH_BTN_state_t;
```

/*
Enum: EN_PUSH_BTN_active_t

Description: An enumeration that defines two possible active states for a push button: pull-up or pull-down.

Members:
- PUSH_BTN_PULL_UP : Represents the active state of a push button when it is connected to a pull-up resistor. In this state, the button is normally open and the pull-up resistor pulls the voltage of the pin to a high state.
- PUSH_BTN_PULL_DOWN : Represents the active state of a push button when it is connected to a pull-down resistor. In this state, the button is normally closed and the pull-down resistor pulls the voltage of the pin to a low state.

Overall, the EN_PUSH_BTN_active_t enumeration provides a way to represent the two possible active states of a push button in a standardized and easy-to-understand manner. By using this enumeration, the software can determine the active state of a push button and configure the pin accordingly.
*/

```
typedef enum
{
   PUSH_BTN_PULL_UP = 0,
   PUSH_BTN_PULL_DOWN
}EN_PUSH_BTN_active_t;
```

/*
Struct : ST_PUSH_BTN_t

Description : A structure that contains the configuration and current state information for a push button.

Members:
- PUSH_BTN_pin : An instance of the ST_pin_config_t struct that contains the configuration settings for the pin used by the push button.
- PUSH_BTN_state : An instance of the EN_PUSH_BTN_state_t enum that represents the current state of the push button (pressed or released).
- PUSH_BTN_connection : An instance of the EN_PUSH_BTN_active_t enum that represents the active state of the push button (pull-up or pull-down).

Overall, the ST_PUSH_BTN_t structure provides a standardized way to represent and manage the configuration and state information for a push button on a micro-controller. By using this structure, the software can easily read the current state of the push button and take appropriate action based on its configuration and connection type. The use of enums for the state and connection fields allows for consistent and easy-to-understand representation of these values.
*/
```
typedef struct
{
   ST_pin_config_t PUSH_BTN_pin;
   EN_PUSH_BTN_state_t PUSH_BTN_state;
   EN_PUSH_BTN_active_t PUSH_BTN_connection;
}ST_PUSH_BTN_t;
```

/*
Function: PUSH_BTN_intialize

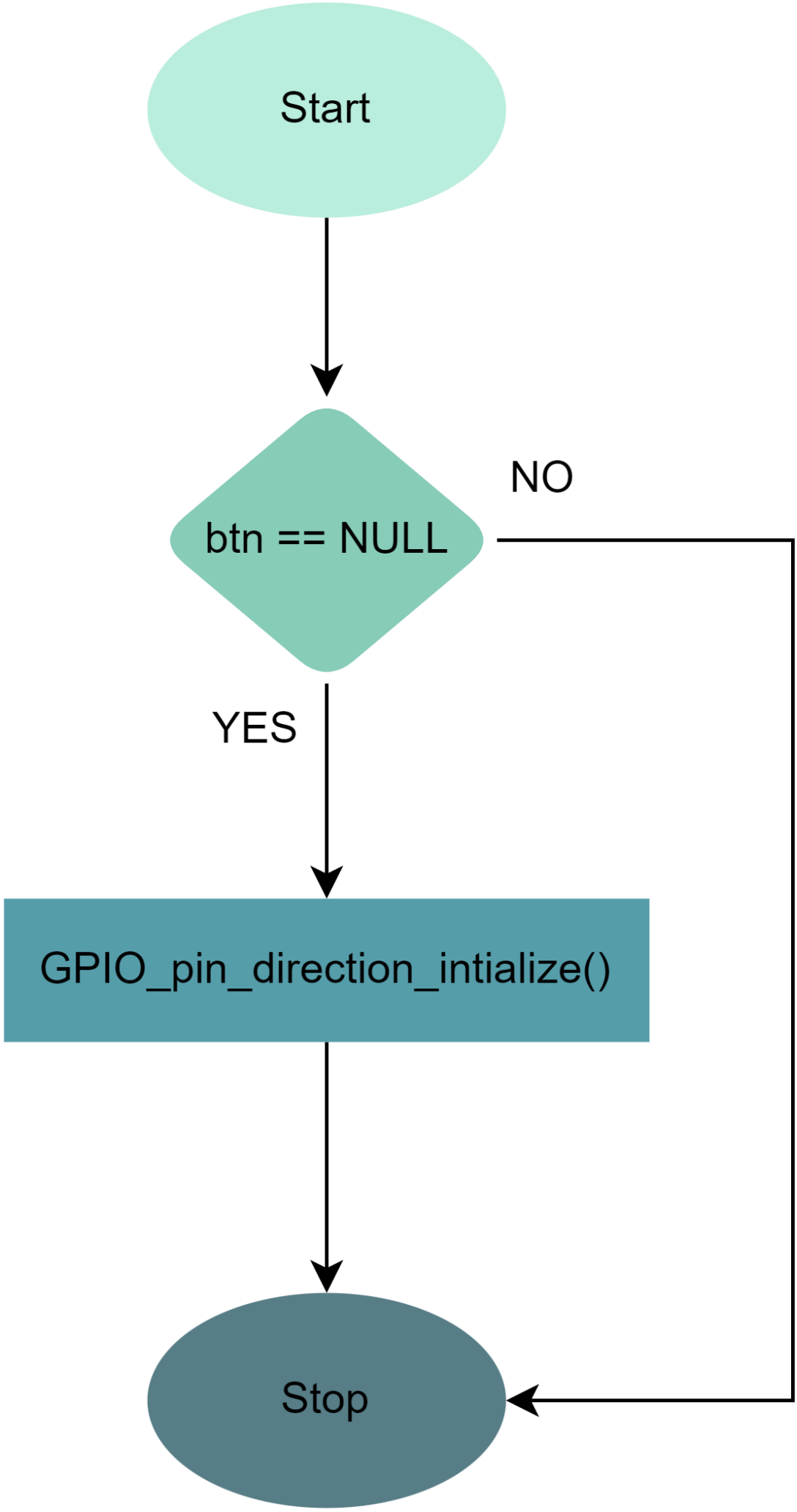
Description: Initializes a push button based on the configuration settings specified in the input parameter.

Parameters:
- btn : A pointer to an ST_PUSH_BTN_t struct that contains the configuration settings for the push button.

Return Type: Std_ReturnType. This is a standard type used in AUTOSAR (Automotive Open System Architecture) software development to indicate the success or failure of a function call.
The possible return values for this function are:
- E_OK : The function has completed successfully.
- E_NOT_OK : The function has encountered an error and could not complete successfully.

Overall, the PUSH_BTN_intialize function provides a way to initialize a push button based on the specified configuration settings. By using this function, the software can ensure that the pin used by the push button is configured correctly and the push button is ready for use.
*/

```
Std_ReturnType PUSH_BTN_intialize(const ST_PUSH_BTN_t *btn);
```


Start
btn == NULL
NO
YES
GPIO_pin_direction_intialize()
Stop

/*
Function : PUSH_BTN_read_state

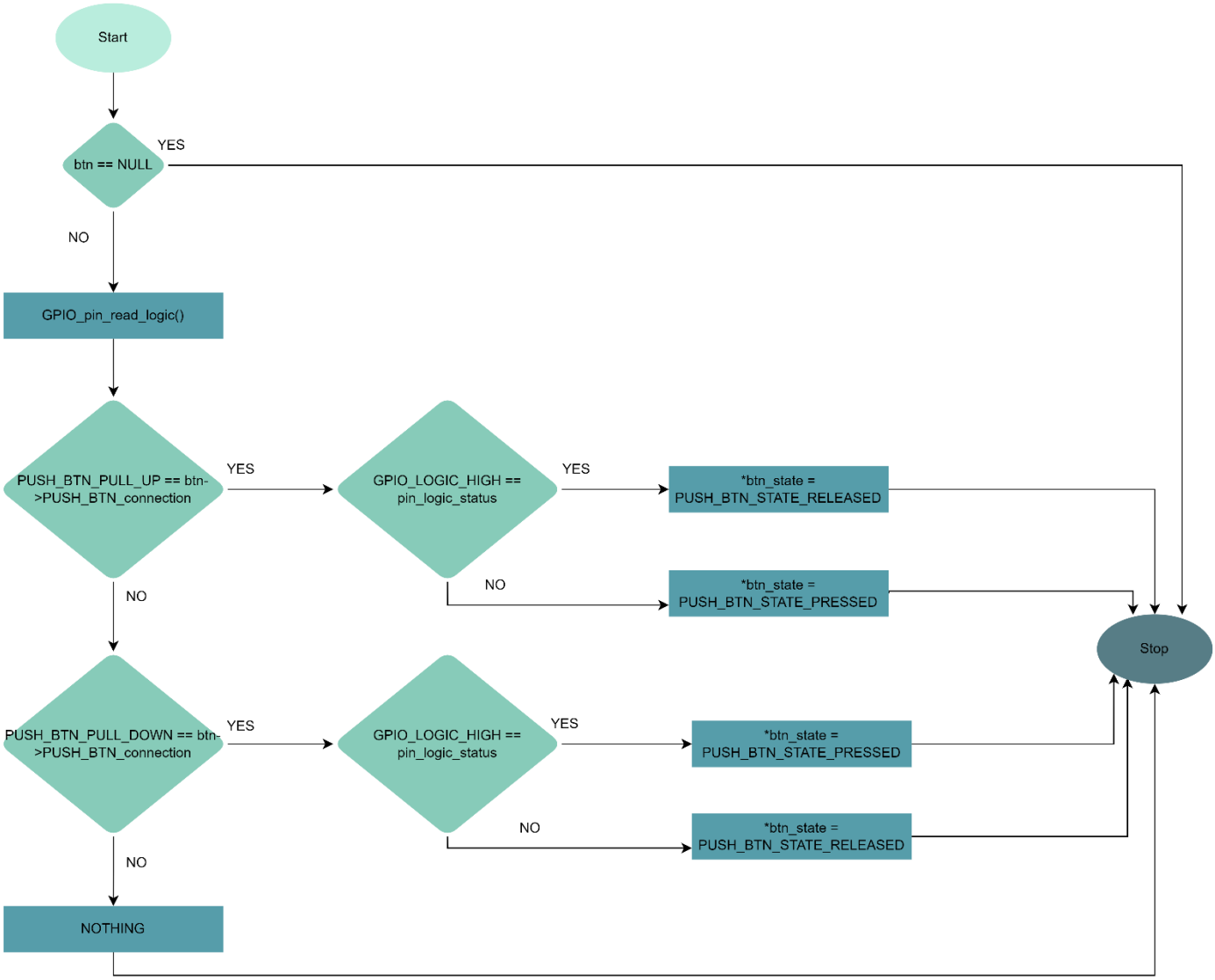
Description : Reads the current state of a push button and returns its value.

Parameters:
- btn : A pointer to an ST_PUSH_BTN_t struct that contains the configuration settings and current state information for the push button.
- btn_state : A pointer to an EN_PUSH_BTN_state_t enum where the current state of the push button will be stored.

Return Type : Std_ReturnType. This is a standard type used in AUTOSAR (Automotive Open System Architecture) software development to indicate the success or failure of a function call.
The possible return values for this function are:
- E_OK : The function has completed successfully.
- E_NOT_OK : The function has encountered an error and could not complete successfully.

Overall, the PUSH_BTN_read_state function provides a way to read the current state of a push button and return
its value. By using this function, the software can determine whether the push button is currently pressed or
released and take appropriate action based on its state.
*/

Std_ReturnType PUSH_BTN_read_state(const ST_PUSH_BTN_t *btn , EN_PUSH_BTN_state_t *btn_state);


Start
btn == NULL
YES
NO
GPIO_pin_read_logic()
PUSH_BTN_PULL_UP == btn->PUSH_BTN_connection
YES
GPIO_LOGIC_HIGH == pin_logic_status
YES
*btn_state = PUSH_BTN_STATE_RELEASED
NO
*btn_state = PUSH_BTN_STATE_PRESSED
NO
PUSH_BTN_PULL_DOWN == btn->PUSH_BTN_connection
YES
GPIO_LOGIC_HIGH == pin_logic_status
YES
*btn_state = PUSH_BTN_STATE_PRESSED
NO
*btn_state = PUSH_BTN_STATE_RELEASED
NO
NOTHING
Stop

2 - Dc Motor

/*
The `dc_motor_t` structure is used to specify the pin assignments for a DC motor in a microcontroller-based system. Its members are defined as follows:

- `motor_port`: a 2-bit bit-field that specifies the port to which the DC motor is connected.
- `motor_pin1`: a 3-bit bit-field that specifies the first pin of the DC motor.
- `motor_pin2`: a 3-bit bit-field that specifies the second pin of the DC motor.

This configuration structure allows for easy and efficient assignment of pin assignments for a DC motor in embedded systems, using the specified port and pin numbers.
*/

```
typedef struct{
	u8 motor_port :2;
	u8 motor_pin1 :3;
	u8 motor_pin2 :3;
}dc_motor_t;
```

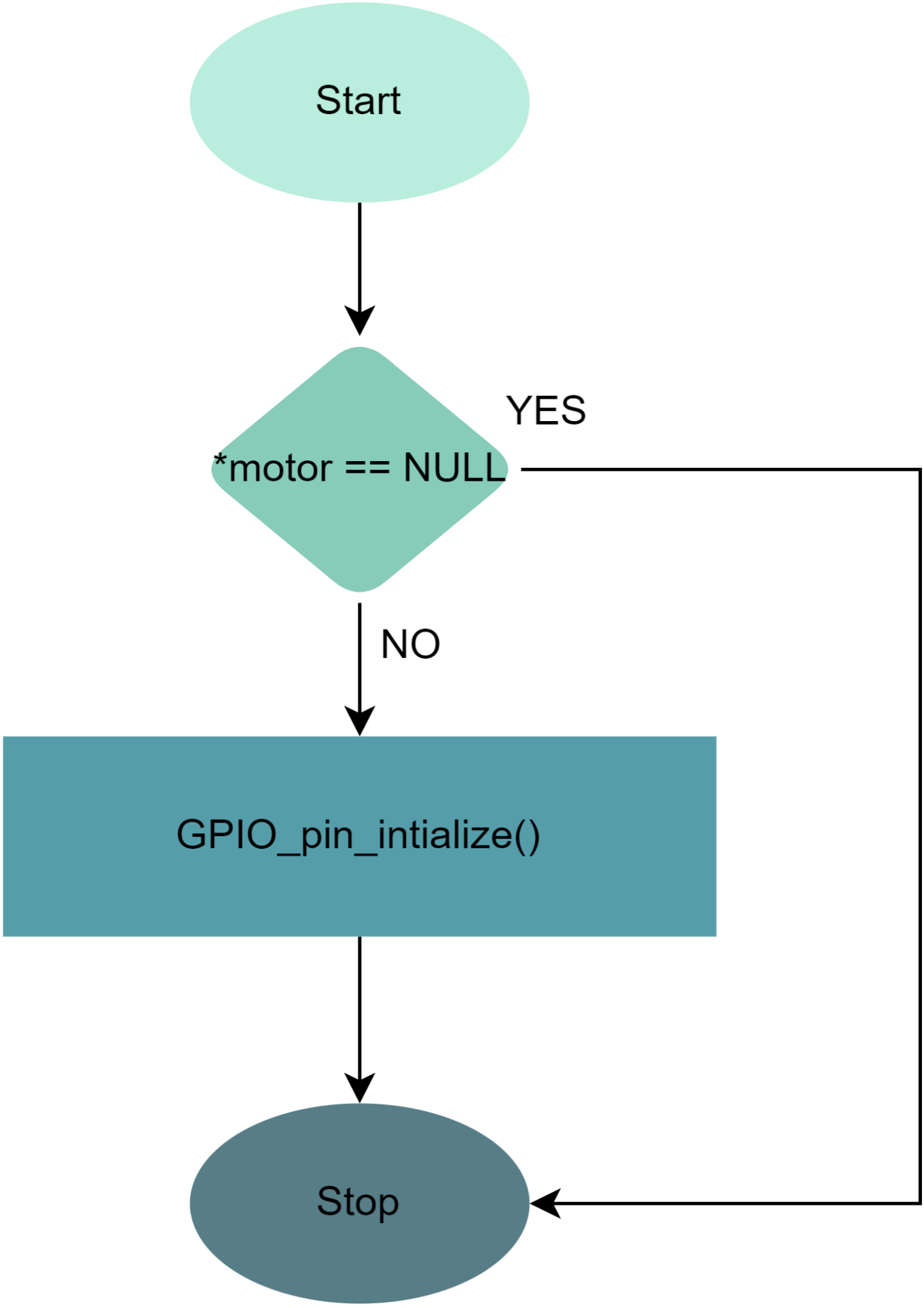
/*
Function Name: DC_MOTOR_vInit

Function Description: The DC_MOTOR_vInit function is used to initialize the pins and settings for a DC motor connected to a microcontroller-based system.

Function Parameters:

_MOTOR: a pointer to a dc_motor_t structure containing the pin assignments and configuration settings for the DC motor.
Function Return Type: void
*/

```
void DC_MOTOR_vInit(const dc_motor_t *_MOTOR);
```

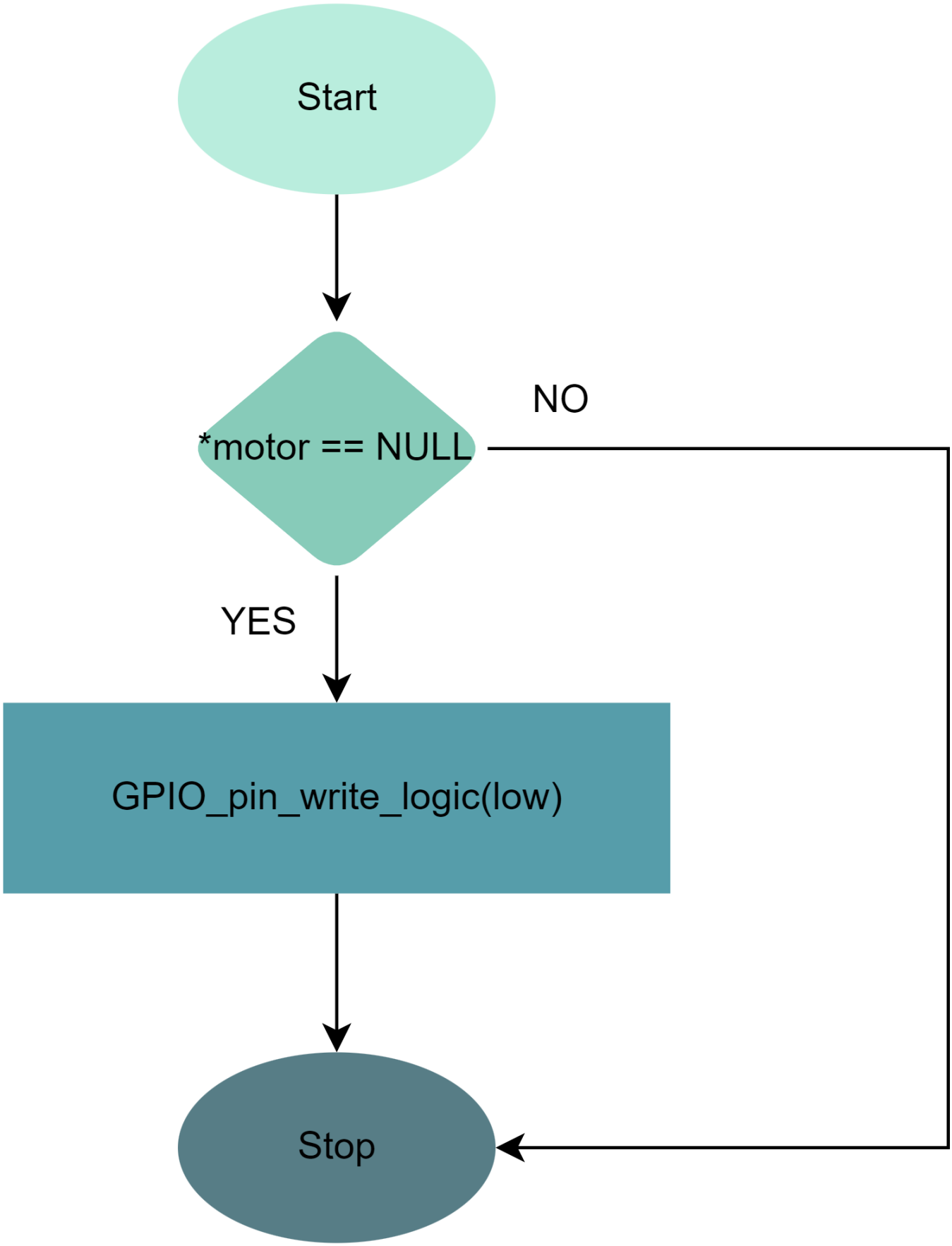

Start
*motor == NULL
YES
NO
GPIO_pin_intialize()
Stop

/*
Function Name: DC_MOTOR_Turn_Off

Function Description: The DC_MOTOR_Turn_Off function is used to turn off a DC motor that is connected to a microcontroller-based system.

Function Parameters:

_MOTOR: a pointer to a dc_motor_t structure containing the pin assignments and configuration settings for the DC motor.
Function Return Type: void
*/
void DC_MOTOR_Turn_Off(const dc_motor_t *_MOTOR);

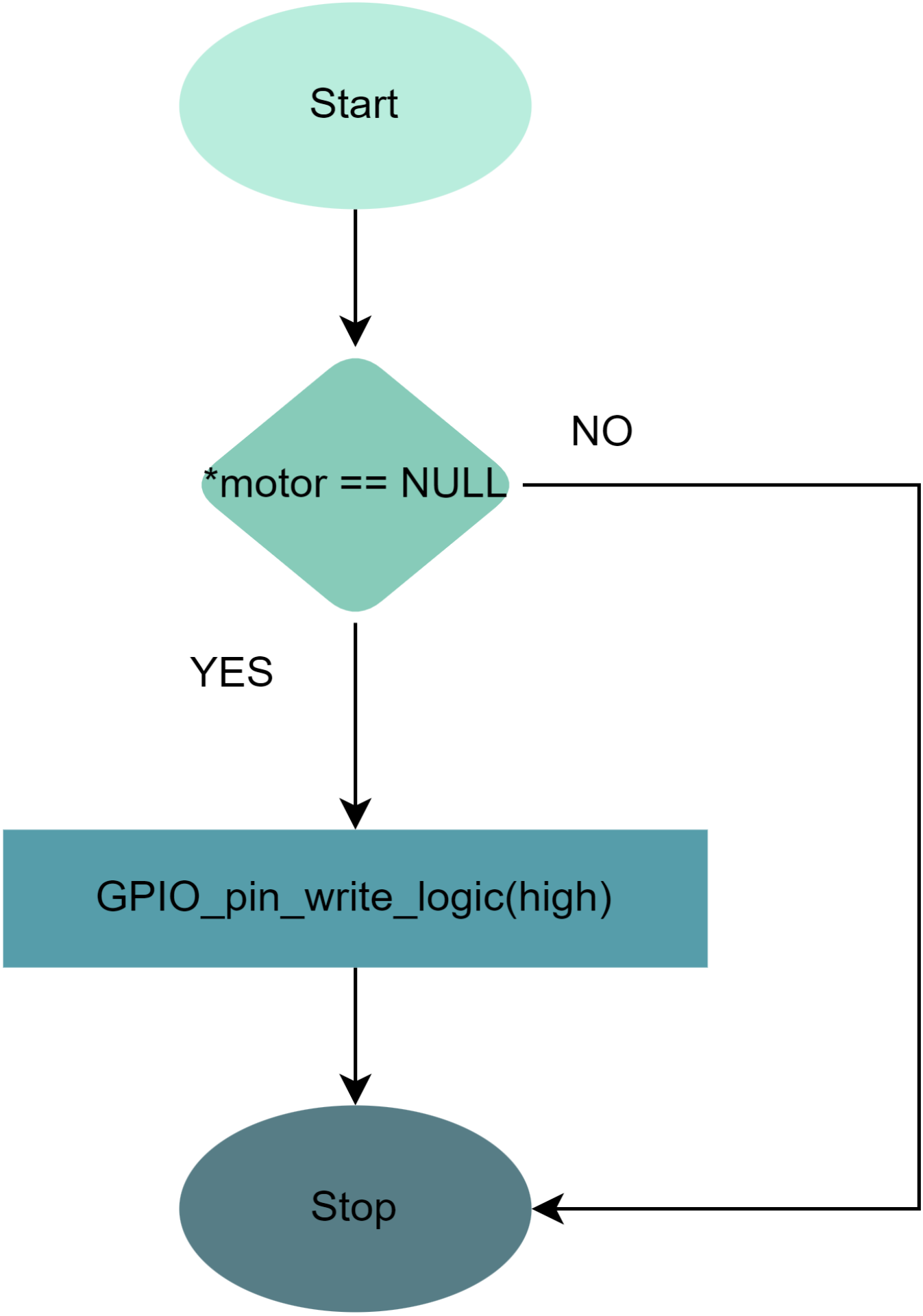

Start
*motor == NULL
NO
YES
GPIO_pin_write_logic(low)
Stop

/*
Function Name: DC_MOTOR_Turn_On

Function Description: The DC_MOTOR_Turn_On function is used to turn on a DC motor that is connected to a microcontroller-based system.

Function Parameters:

_MOTOR: a pointer to a dc_motor_t structure containing the pin assignments and configuration settings for the DC motor.
Function Return Type: void
*/
```
void DC_MOTOR_Turn_On(const dc_motor_t *_MOTOR);
```


Start
*motor == NULL
NO
YES
GPIO_pin_write_logic(high)
Stop

3 - Lcd

/*
The LCD_4bit_cfg structure is a configuration structure used to specify the pin assignments for a 4-bit mode LCD in a microcontroller-based system. Its members are defined as follows:

LCD_PORT: an unsigned 8-bit integer that specifies the port to which the LCD is connected. This member is a bit-field that is 2 bits wide.
LCD_RS_PIN: an unsigned 8-bit integer that specifies the register select (RS) pin of the LCD. This member is a bit-field that is 3 bits wide.
LCD_RW_PIN: an unsigned 8-bit integer that specifies the read/write (RW) pin of the LCD. This member is a bit-field that is 3 bits wide.
LCD_EN_PIN: an unsigned 8-bit integer that specifies the enable (EN) pin of the LCD. This member is a bit-field that is 3 bits wide.
LCD_D4_PIN: an unsigned 8-bit integer that specifies the data 4 (D4) pin of the LCD. This member is a bit-field that is 3 bits wide.
LCD_D5_PIN: an unsigned 8-bit integer that specifies the data 5 (D5) pin of the LCD. This member is a bit-field that is 3 bits wide.
LCD_D6_PIN: an unsigned 8-bit integer that specifies the data 6 (D6) pin of the LCD. This member is a bit-field that is 3 bits wide.
LCD_D7_PIN: an unsigned 8-bit integer that specifies the data 7 (D7) pin of the LCD. This member is a bit-field that is 3 bits wide.
By using this configuration structure, you can easily specify the pin assignments for a 4-bit mode LCD in your system. The exact implementation of the LCD_4bit_cfg structure will depend on the specific hardware and software platform being used, as well as the requirements of the application.
*/

```
typedef struct{
	u8 LCD_PORT		:2;
	u8 LCD_RS_PIN		:3;
	u8 LCD_RW_PIN		:3;
	u8 LCD_EN_PIN		:3;
	u8 LCD_D4_PIN		:3;
	u8 LCD_D5_PIN		:3;
	u8 LCD_D6_PIN		:3;
	u8 LCD_D7_PIN		:3;
}LCD_4bit_cfg;
```

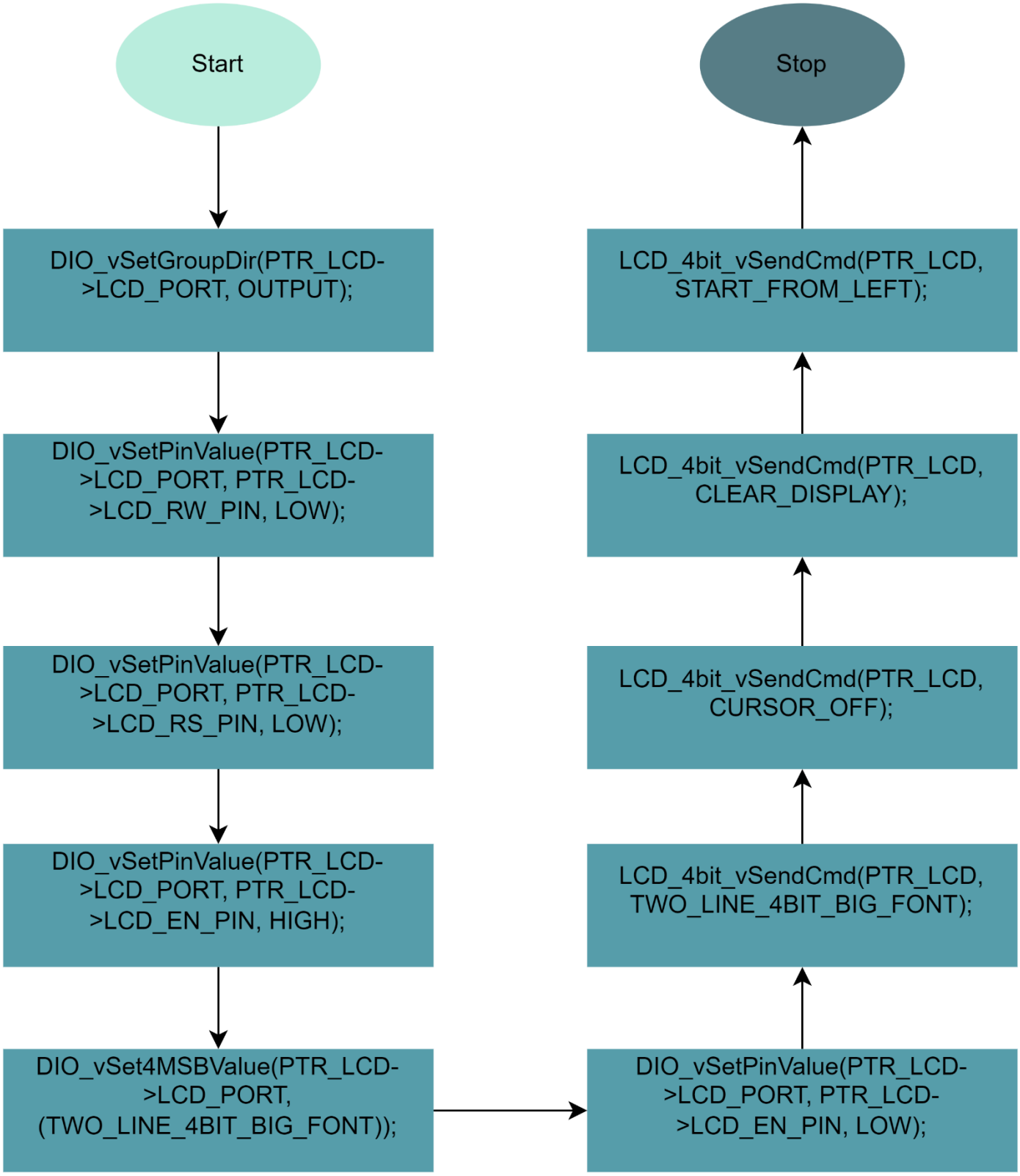
/*
Function Name: LCD_4bit_vInit

Function Description: The LCD_4bit_vInit function is used to initialize the pins and settings for a 4-bit mode LCD connected to a microcontroller-based system.

Function Parameters:

PTR_LCD: a pointer to an LCD_4bit_cfg structure containing the pin assignments and configuration settings for the 4-bit mode LCD.
Function Return Type: void
*/

```
void LCD_4bit_vInit(const LCD_4bit_cfg *PTR_LCD);
```

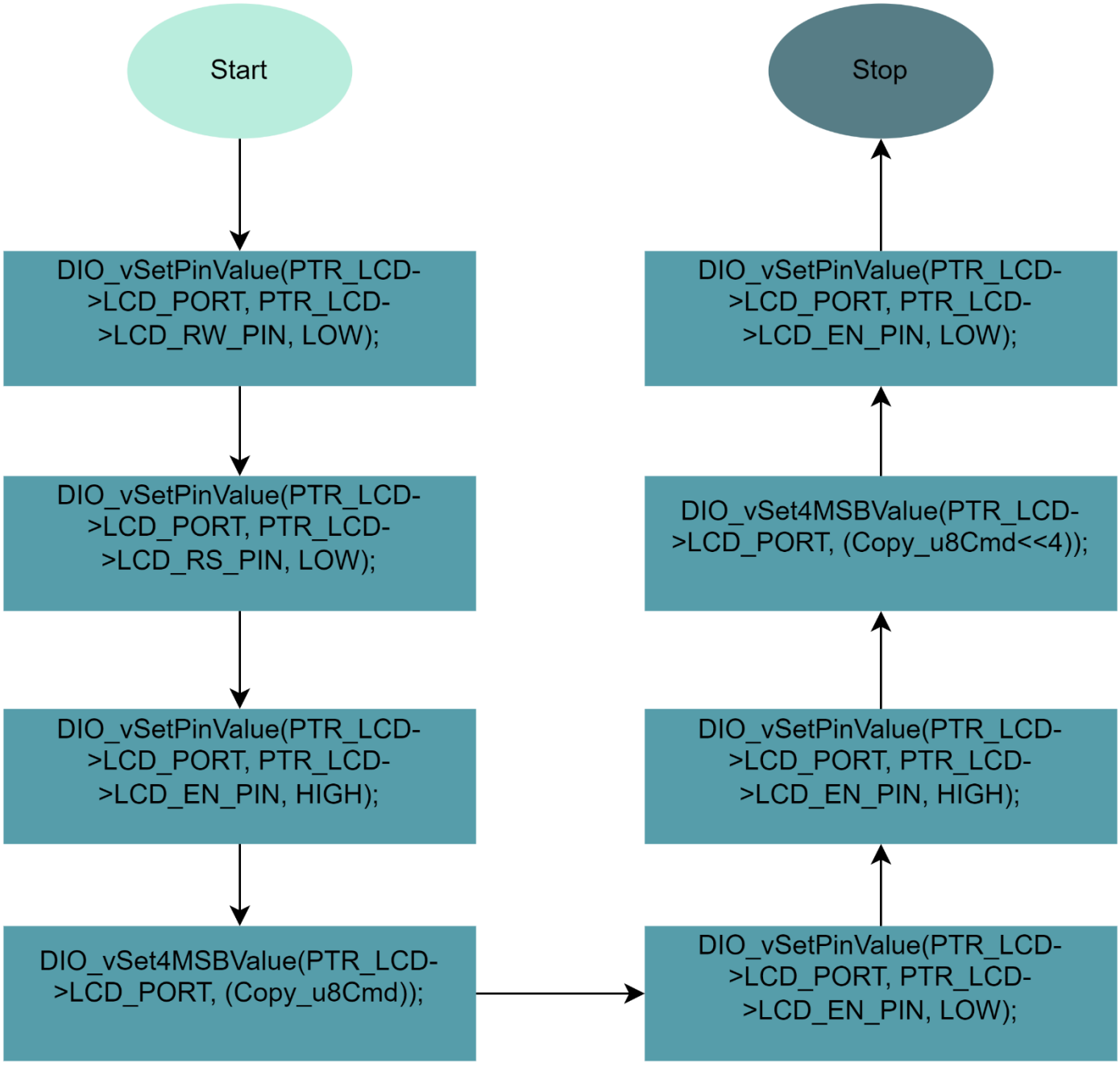

Start
DIO_vSetGroupDir(PTR_LCD->LCD_PORT, OUTPUT);
DIO_vSetPinValue(PTR_LCD->LCD_PORT, PTR_LCD->LCD_RW_PIN, LOW);
DIO_vSetPinValue(PTR_LCD->LCD_PORT, PTR_LCD->LCD_RS_PIN, LOW);
DIO_vSetPinValue(PTR_LCD->LCD_PORT, PTR_LCD->LCD_EN_PIN, HIGH);
DIO_vSet4MSBValue(PTR_LCD->LCD_PORT, (TWO_LINE_4BIT_BIG_FONT));
DIO_vSetPinValue(PTR_LCD->LCD_PORT, PTR_LCD->LCD_EN_PIN, LOW);
LCD_4bit_vSendCmd(PTR_LCD, TWO_LINE_4BIT_BIG_FONT);
LCD_4bit_vSendCmd(PTR_LCD, CURSOR_OFF);
LCD_4bit_vSendCmd(PTR_LCD, CLEAR_DISPLAY);
LCD_4bit_vSendCmd(PTR_LCD, START_FROM_LEFT);
Stop

/*
Function Name: LCD_4bit_vSendCmd

Function Description: The LCD_4bit_vSendCmd function is used to send a command to an LCD in 4-bit mode.

Function Parameters:

PTR_LCD: a pointer to an LCD_4bit_cfg structure containing the pin assignments and configuration settings for the 4-bit mode LCD.
Copy_u8Cmd: an 8-bit unsigned integer that specifies the command to be sent to the LCD.
Function Return Type: void
*/
void LCD_4bit_vSendCmd(const LCD_4bit_cfg *PTR_LCD, u8 Copy_u8Cmd);

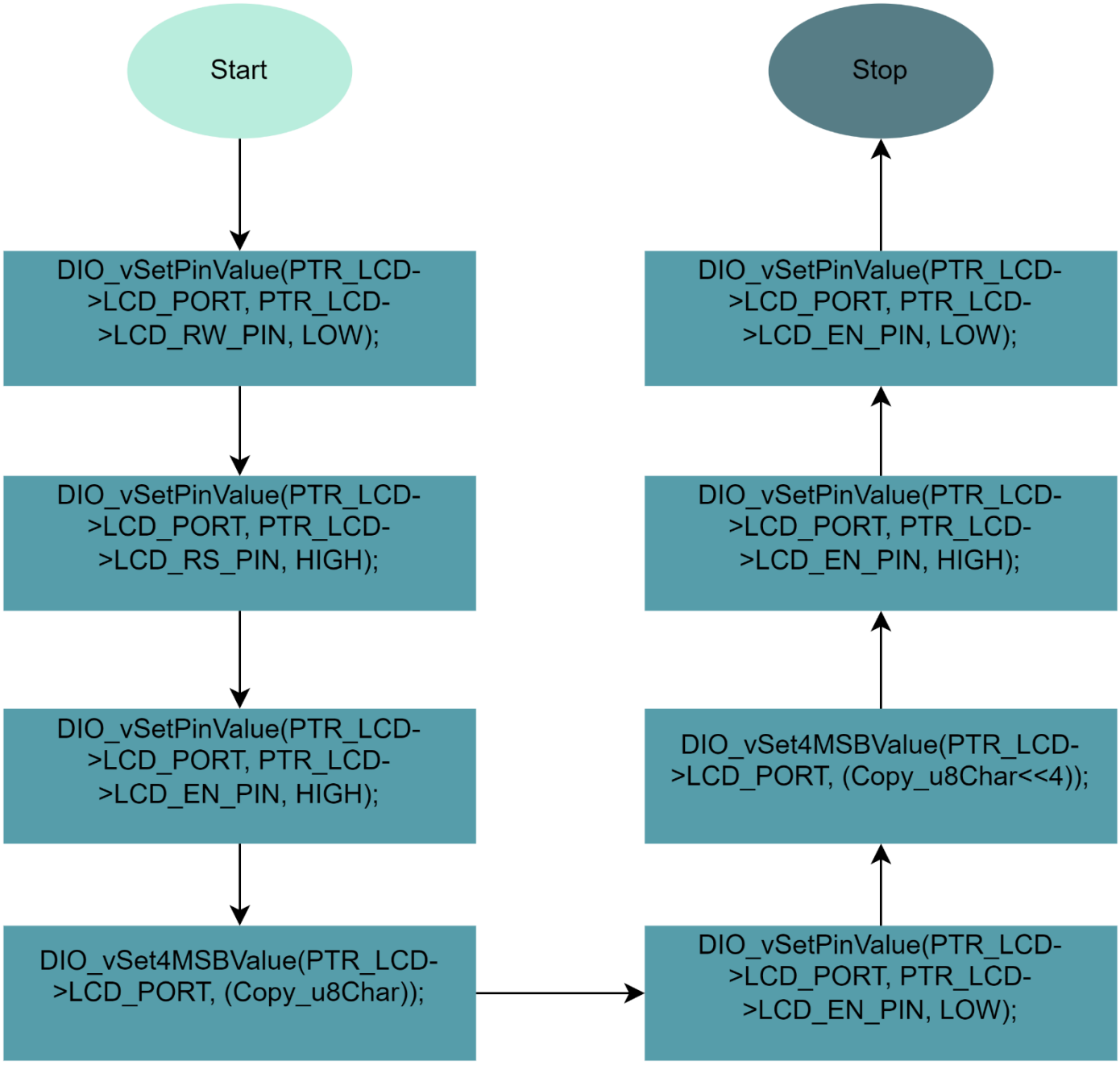


/*
Function Name: LCD_4bit_vSendChar

Function Description: The LCD_4bit_vSendChar function is used to send a character to an LCD in 4-bit mode.

Function Parameters:

PTR_LCD: a pointer to an LCD_4bit_cfg structure containing the pin assignments and configuration settings for the 4-bit mode LCD.
Copy_u8Char: an 8-bit unsigned integer that specifies the character to be sent to the LCD.
Function Return Type: void
*/
void LCD_4bit_vSendChar(const LCD_4bit_cfg *PTR_LCD, u8 Copy_u8Char);

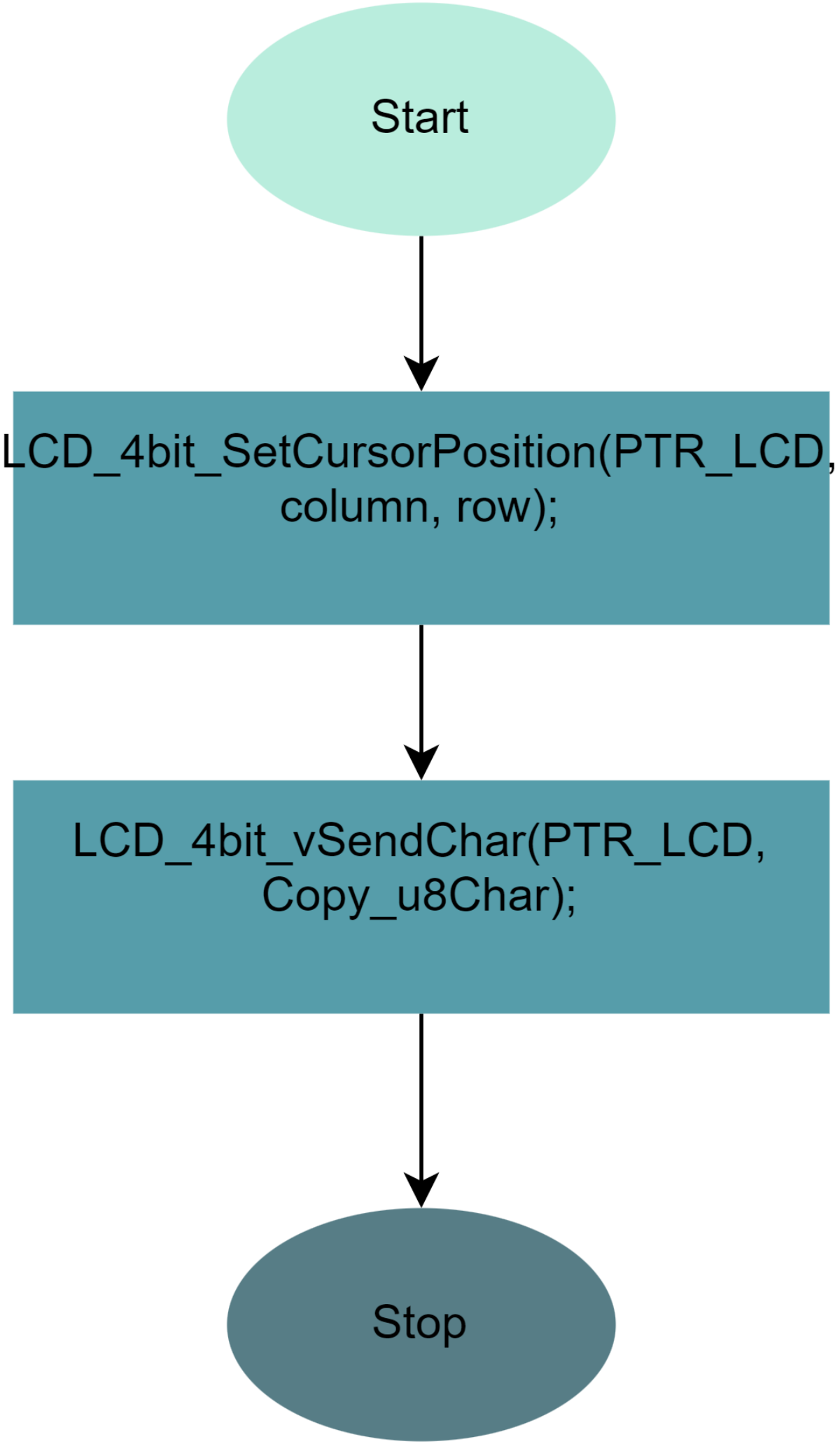


/*
Function Name: LCD_4bit_vSendChar_pos

Function Description: The LCD_4bit_vSendChar_pos function is used to send a character to a specific position on an LCD in 4-bit mode.

Function Parameters:

PTR_LCD: a pointer to an LCD_4bit_cfg structure containing the pin assignments and configuration settings for the 4-bit mode LCD.
Copy_u8Char: an 8-bit unsigned integer that specifies the character to be sent to the LCD.
row: an 8-bit unsigned integer that specifies the row (0 or 1) of the LCD where the character should be displayed.
column: an 8-bit unsigned integer that specifies the column (0 to 15) of the LCD where the character should be displayed.
Function Return Type: void
*/
void LCD_4bit_vSendChar_pos(const LCD_4bit_cfg *PTR_LCD, u8 Copy_u8Char, u8 row, u8 column);

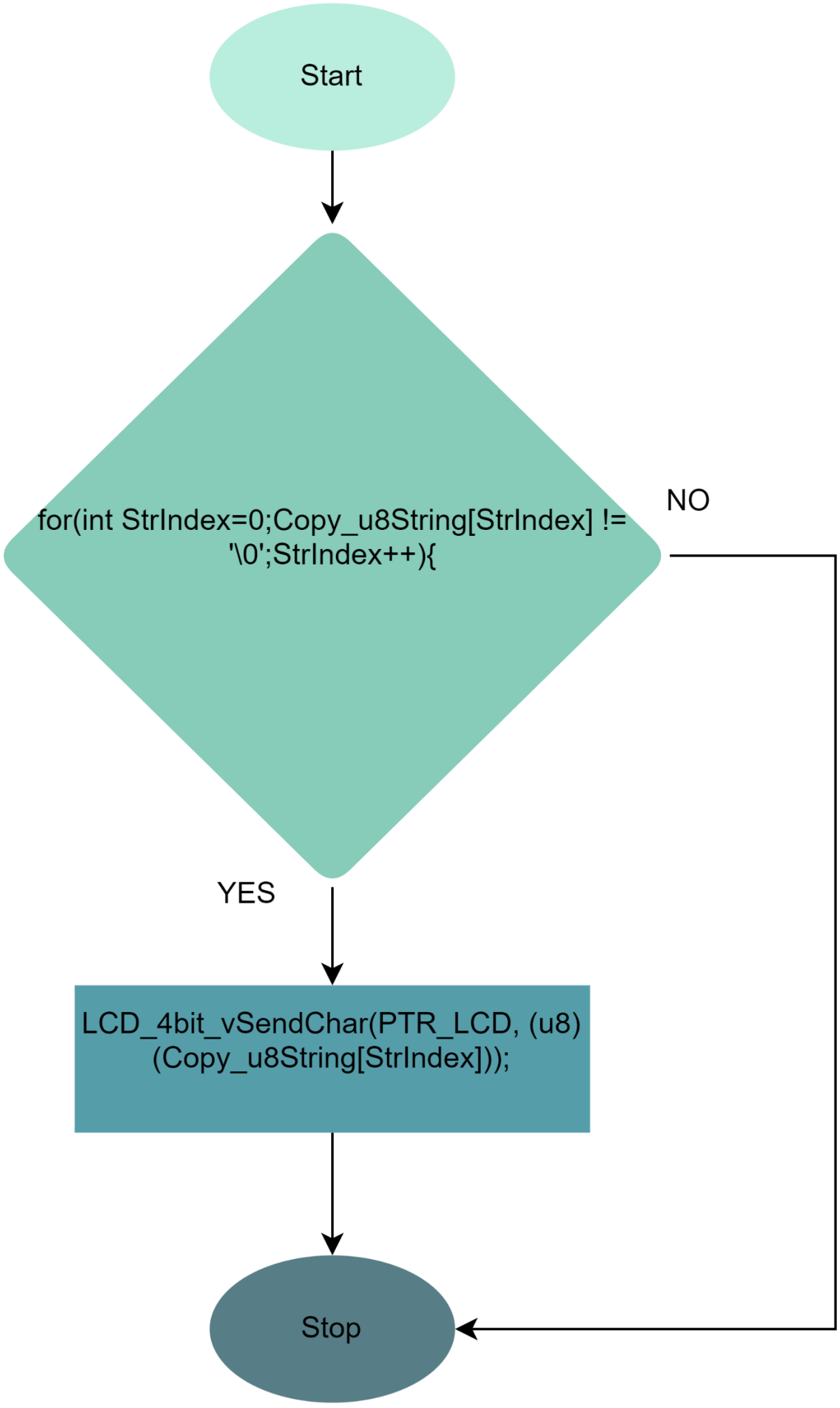

Start
LCD_4bit_SetCursorPosition(PTR_LCD, column, row);
LCD_4bit_vSendChar(PTR_LCD, Copy_u8Char);
Stop

/*
Function Name: LCD_4bit_vSendString

Function Description: The LCD_4bit_vSendString function is used to send a string of characters to an LCD in 4-bit mode.

Function Parameters:

PTR_LCD: a pointer to an LCD_4bit_cfg structure containing the pin assignments and configuration settings for the 4-bit mode LCD.
Copy_u8String: a pointer to an array of 8-bit unsigned integers that contains the string to be sent to the LCD.
Function Return Type: void
*/
void LCD_4bit_vSendString(const LCD_4bit_cfg *PTR_LCD, u8 *Copy_u8String);

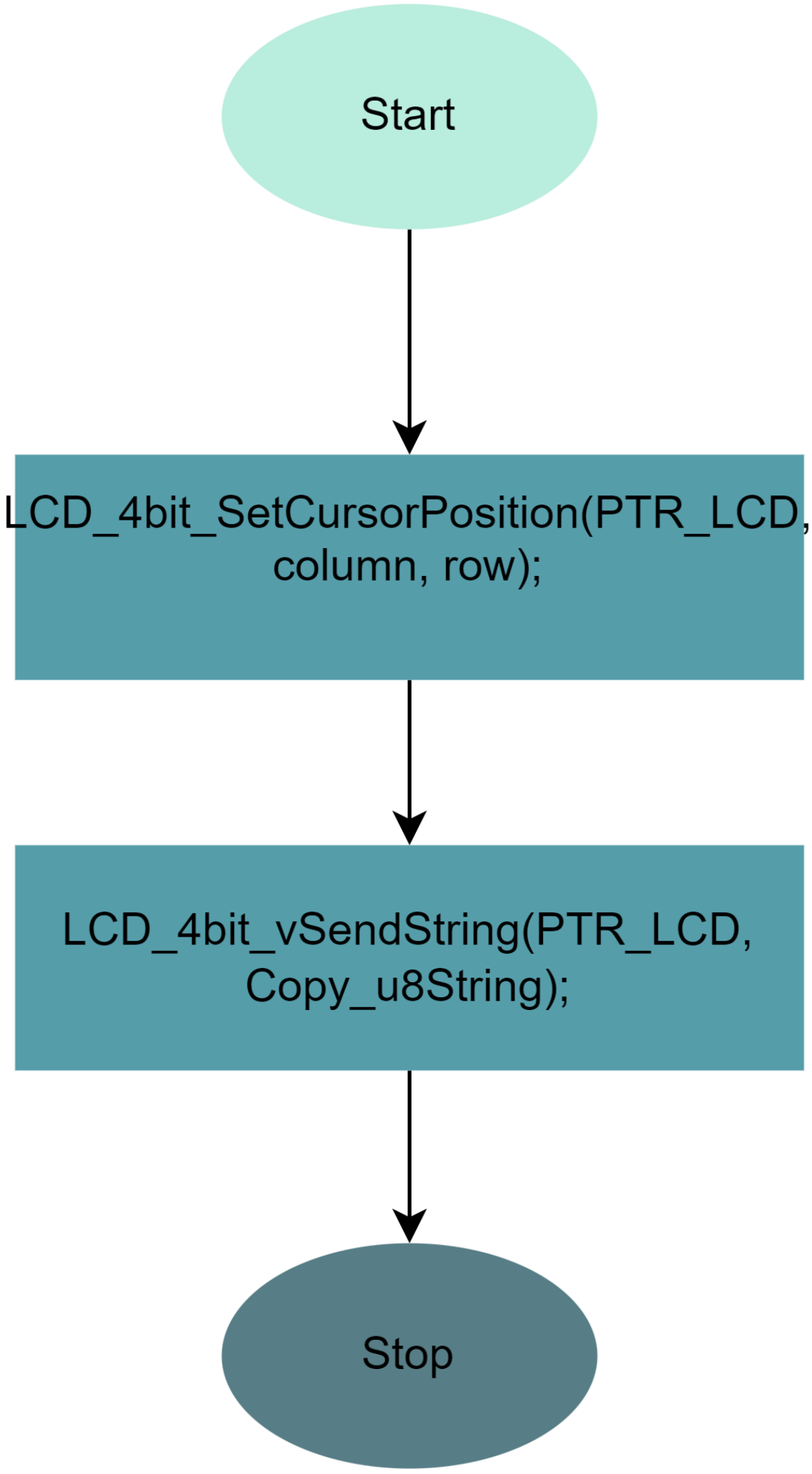

Start
for(int StrIndex=0;Copy_u8String[StrIndex] != '\0';StrIndex++){
NO
YES
LCD_4bit_vSendChar(PTR_LCD, (u8) (Copy_u8String[StrIndex]));
Stop

/*
Function Name: LCD_4bit_vSendString_pos

Function Description: The LCD_4bit_vSendString_pos function is used to send a string of characters to a specific position on an LCD in 4-bit mode.

Function Parameters:

PTR_LCD: a pointer to an LCD_4bit_cfg structure containing the pin assignments and configuration settings for the 4-bit mode LCD.
Copy_u8String: a pointer to an array of 8-bit unsigned integers that contains the string to be sent to the LCD.
row: an 8-bit unsigned integer that specifies the row (0 or 1) of the LCD where the string should be displayed.
column: an 8-bit unsigned integer that specifies the column (0 to 15) of the LCD where the string should start.
Function Return Type: void
*/
void LCD_4bit_vSendString_pos(const LCD_4bit_cfg *PTR_LCD, u8 *Copy_u8String, u8 row, u8 column);


Start
LCD_4bit_SetCursorPosition(PTR_LCD, column, row);
LCD_4bit_vSendString(PTR_LCD, Copy_u8String);
Stop

/*

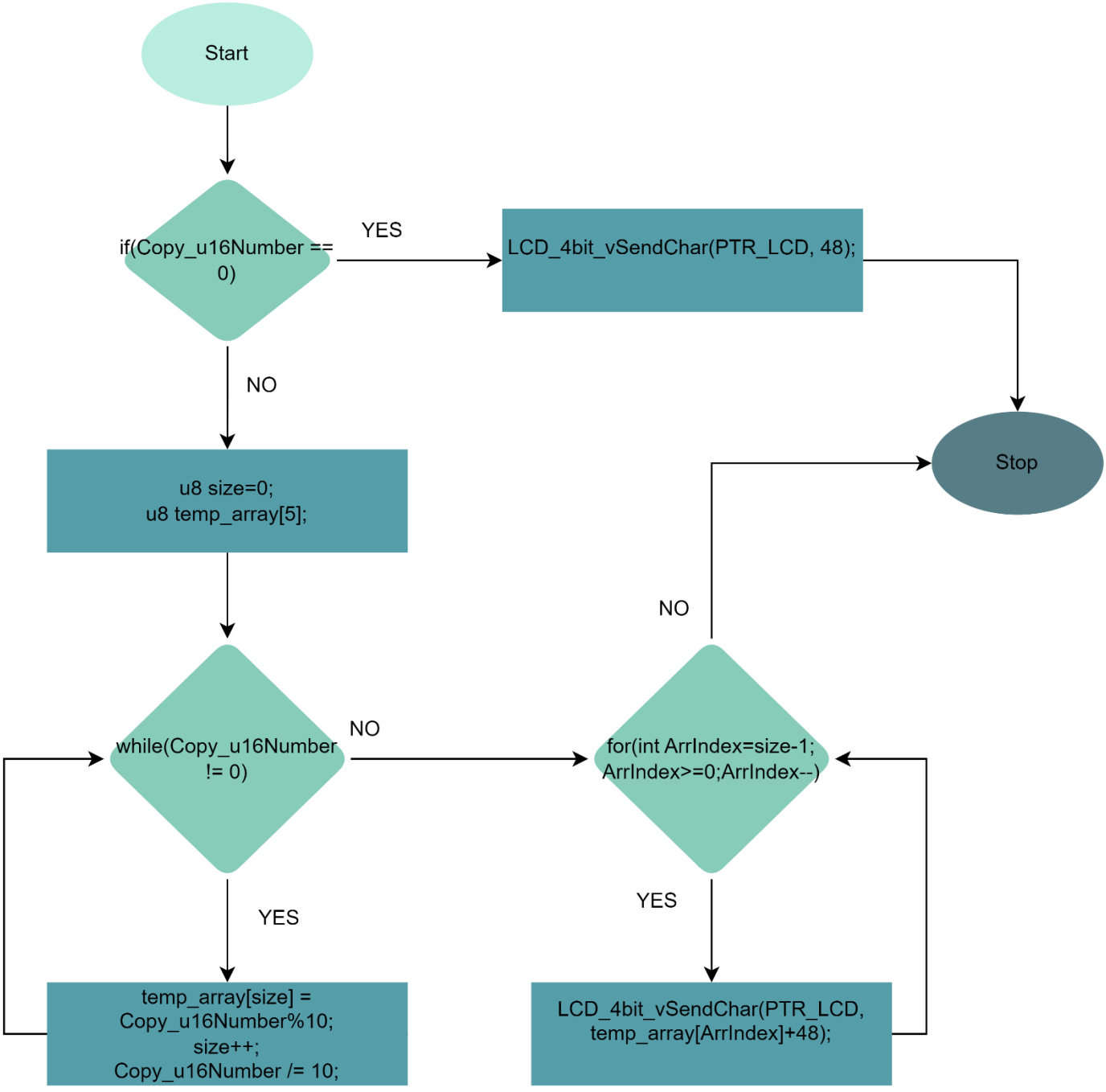
Function Name: LCD_4bit_vSendNumber

Function Description: The LCD_4bit_vSendNumber function is used to send a 16-bit unsigned integer as a decimal number to an LCD in 4-bit mode.

Function Parameters:

PTR_LCD: a pointer to an LCD_4bit_cfg structure containing the pin assignments and configuration settings for the 4-bit mode LCD.
Copy_u16Number: a 16-bit unsigned integer that contains the decimal number to be sent to the LCD.
Function Return Type: void
*/

void LCD_4bit_vSendNumber(const LCD_4bit_cfg *PTR_LCD, u16 Copy_u16Number);

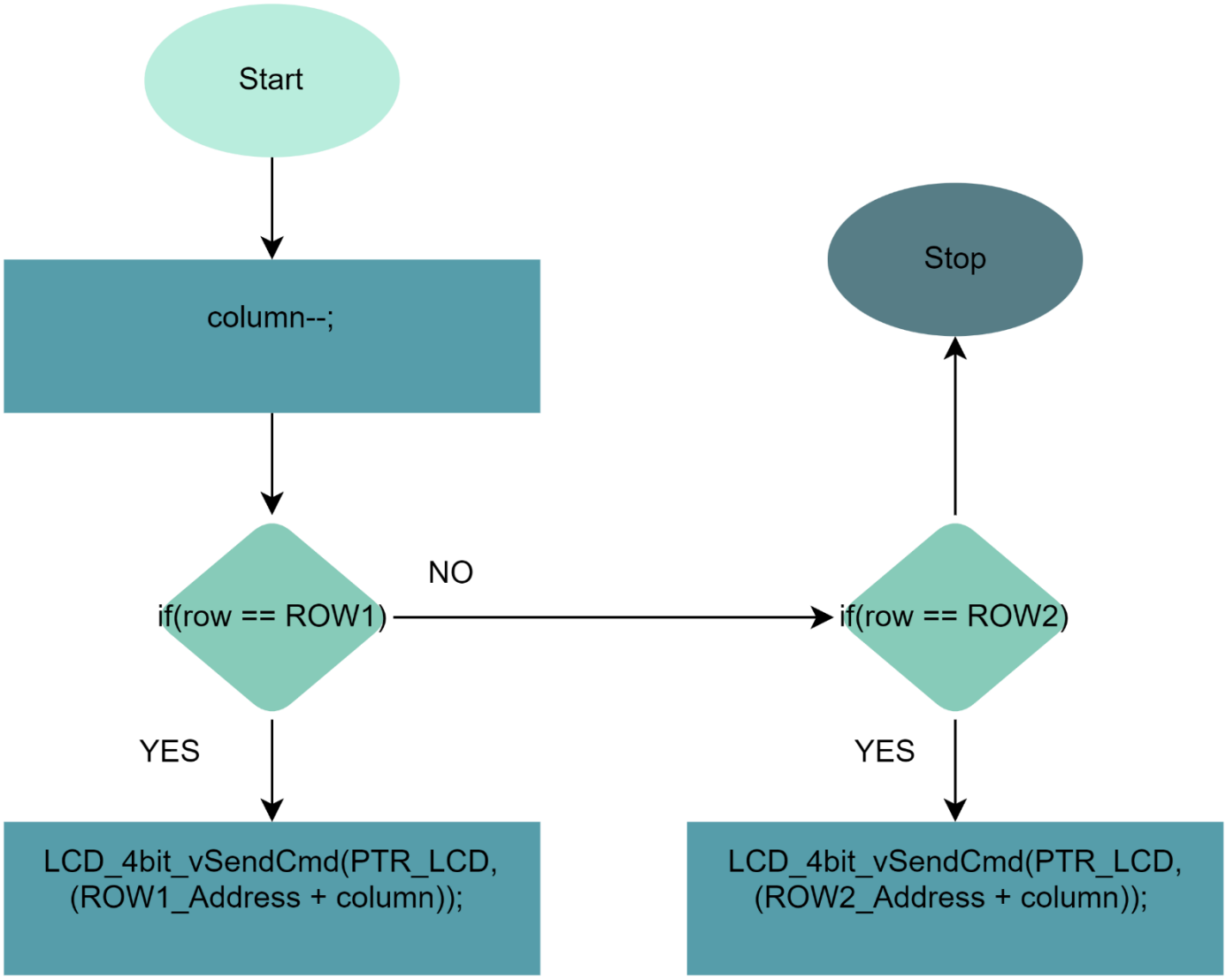


/*
Function Name: LCD_4bit_SetCursorPosition

Function Description: The LCD_4bit_SetCursorPosition function is used to set the cursor position of an LCD in 4-bit mode.

Function Parameters:

PTR_LCD: a pointer to an LCD_4bit_cfg structure containing the pin assignments and configuration settings for the 4-bit mode LCD.
column: an 8-bit unsigned integer that specifies the column (0 to 15) of the LCD where the cursor should be moved.
row: an 8-bit unsigned integer that specifies the row (0 or 1) of the LCD where the cursor should be moved.
Function Return Type: void
*/
void LCD_4bit_SetCursorPosition(const LCD_4bit_cfg *PTR_LCD, u8 column, u8 row);



4 - Keypad

/*
Struct: ST_KEYPAD_cfg_t

Description: A structure that contains the configuration settings for a keypad or matrix keyboard with a specific number of rows and columns.

Members:
- KEYPAD_arrRow[KEYPAD_ROWS_SIZE] : An array of ST_pin_config_t structs that contains the configuration settings for the pins used as the rows of the keypad. The size of the array is specified by the KEYPAD_ROWS_SIZE macro.
- KEYPAD_arrColumns[KEYPAD_COLUMNS_SIZE]: An array of ST_pin_config_t structs that contains the configuration settings for the pins used as the columns of the keypad. The size of the array is specified by the KEYPAD_COLUMNS_SIZE macro.

Overall, the ST_KEYPAD_cfg_t structure provides a structured and organized way to configure the pins used by a keypad or matrix keyboard. By using this structure, the software can ensure that the pins are configured correctly for their intended use, which can help to simplify the code and make it easier to understand. The use of arrays of ST_pin_config_t structs allows for the configuration of multiple pins in a single structure, reducing the amount of code needed to configure the keypad or matrix keyboard.
*/

```
typedef struct
{
        ST_pin_config_t KEYPAD_arrRow[KEYPAD_ROWS_SIZE];
        ST_pin_config_t KEYPAD_arrColumns[KEYPAD_COLUMNS_SIZE];
}ST_KEYPAD_cfg_t;
```

/*
Function : KEYPAD_init

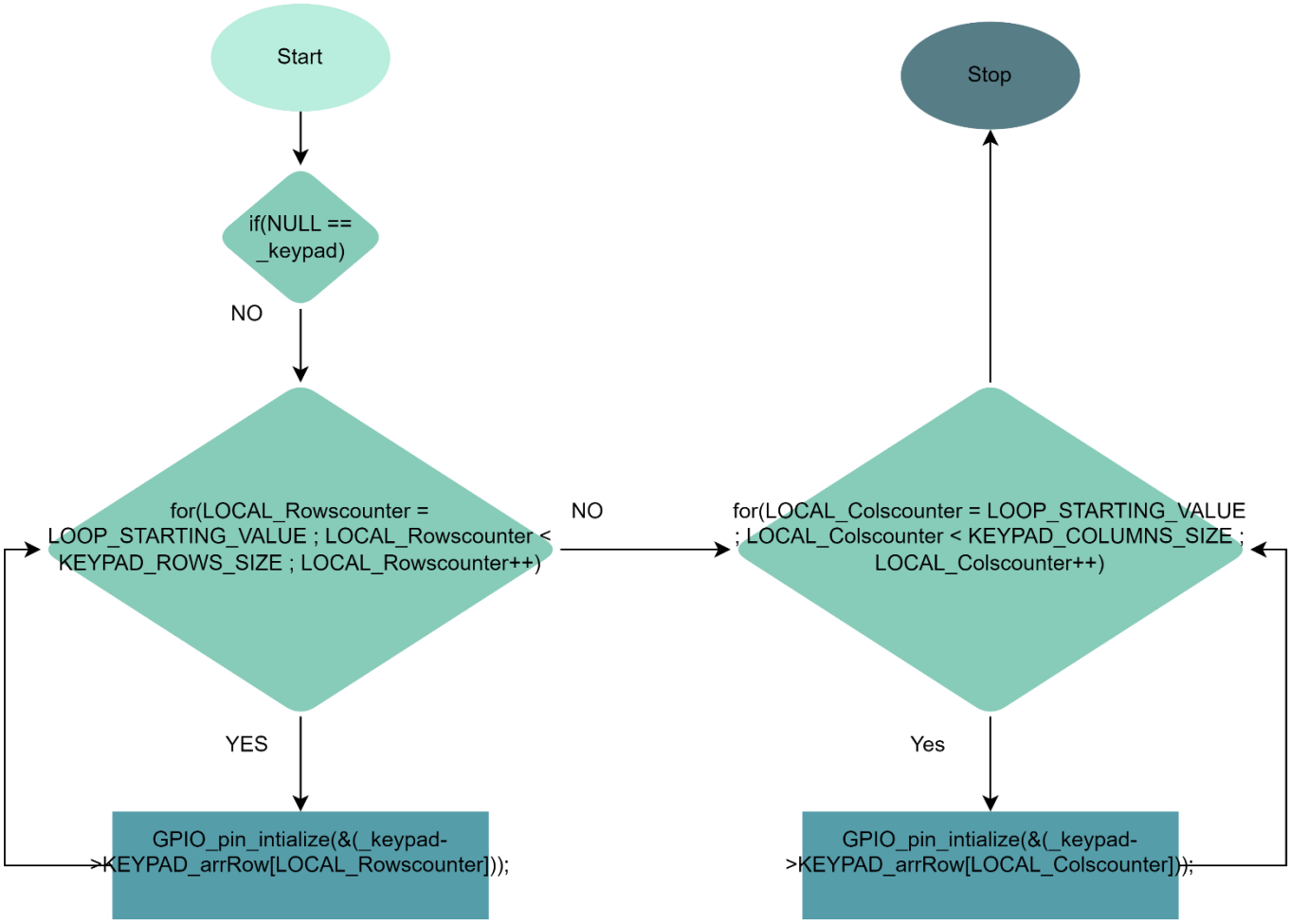
Description : Initializes a keypad or matrix keyboard based on the configuration settings specified in the input parameter.

Parameters :
- _keypad : A pointer to an ST_KEYPAD_cfg_t struct that contains the configuration pin settings for the keypad or matrix keyboard.

Return Type: Std_ReturnType. This is a standard type used in AUTOSAR (Automotive Open System Architecture) software development to indicate the success or failure of a function call.
The possible return values for this function are:
- E_OK : The function has completed successfully.
- E_NOT_OK : The function has encountered an error and could not complete successfully.

Overall, the KEYPAD_init function provides a way to initialize a keypad or matrix keyboard based on the specified configuration settings. By using this function, the software can ensure that the pins are configured correctly and the keypad or matrix keyboard is ready for use.
*/

```
Std_ReturnType KEYPAD_init(ST_KEYPAD_cfg_t *_keypad);
```


Start
if(NULL == _keypad)
NO
for(LOCAL_Rowscounter = LOOP_STARTING_VALUE ; LOCAL_Rowscounter < KEYPAD_ROWS_SIZE ; LOCAL_Rowscounter++)
YES
GPIO_pin_intialize(&(_keypad->KEYPAD_arrRow[LOCAL_Rowscounter]));
NO
for(LOCAL_Colscounter = LOOP_STARTING_VALUE ; LOCAL_Colscounter < KEYPAD_COLUMNS_SIZE ; LOCAL_Colscounter++)
Yes
GPIO_pin_intialize(&(_keypad->KEYPAD_arrRow[LOCAL_Colscounter]));
Stop

/*
Function : KEYPAD_scaningPressedBtn

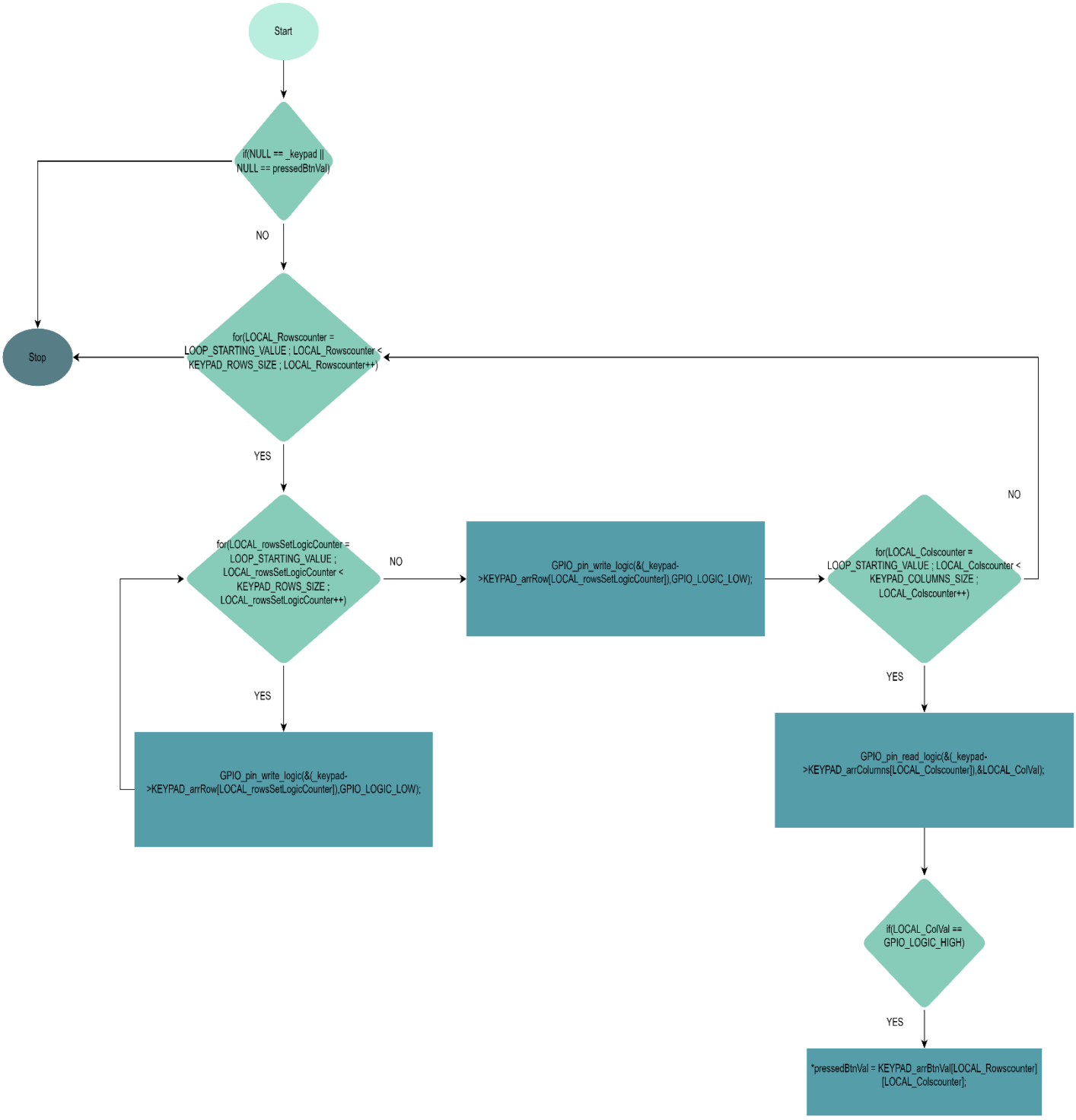
Description : Scans the keypad or matrix keyboard for any pressed buttons and returns the value of the pressed button.

Parameters :
- _keypad : A pointer to an ST_KEYPAD_cfg_t struct that contains the configuration pin settings for the keypad or matrix keyboard.
- pressedBtnVal : A pointer to an 8-bit unsigned character where the value of the pressed button will be stored.

Return Type : Std_ReturnType. This is a standard type used in AUTOSAR (Automotive Open System Architecture) software development to indicate the success or failure of a function call.
The possible return values for this function are:
- E_OK : The function has completed successfully.
- E_NOT_OK : The function has encountered an error and could not complete successfully.

Overall, the KEYPAD_scaningPressedBtn function provides a way to scan the keypad or matrix keyboard for any pressed buttons and retrieve the value of the pressed button. By using this function, the software can detect which button has been pressed and take appropriate action based on the button value.
*/

Std_ReturnType KEYPAD_scaningPressedBtn(ST_KEYPAD_cfg_t *_keypad , Uchar8_t *pressedBtnVal);


Start
if(NULL == _keypad ||
NULL == pressedBtnVal)
NO
Stop
for(LOCAL_Rowscounter =
LOOP_STARTING_VALUE ; LOCAL_Rowscounter <
KEYPAD_ROWS_SIZE ; LOCAL_Rowscounter++)
YES
for(LOCAL_rowsSetLogicCounter =
LOOP_STARTING_VALUE ;
LOCAL_rowsSetLogicCounter <
KEYPAD_ROWS_SIZE ;
LOCAL_rowsSetLogicCounter++)
NO
YES
GPIO_pin_write_logic(&(_keypad-
>KEYPAD_arrRow[LOCAL_rowsSetLogicCounter]),GPIO_LOGIC_LOW);
GPIO_pin_write_logic(&(_keypad-
>KEYPAD_arrRow[LOCAL_rowsSetLogicCounter]),GPIO_LOGIC_LOW);
for(LOCAL_Colscounter =
LOOP_STARTING_VALUE ; LOCAL_Colscounter <
KEYPAD_COLUMNS_SIZE ;
LOCAL_Colscounter++)
NO
YES
GPIO_pin_read_logic(&(_keypad-
>KEYPAD_arrColumns[LOCAL_Colscounter]),&LOCAL_ColVal);
if(LOCAL_ColVal ==
GPIO_LOGIC_HIGH)
YES
*pressedBtnVal = KEYPAD_arrBtnVal[LOCAL_Rowscounter]
[LOCAL_Colscounter];