

design & Implement   
A rgb led control V3

Youssef Abbas

Arafa Arafa



Sprints

Contents

[Project Introduction: 1](#_Toc137926563)

[Hardware Requirements: 1](#_Toc137926564)

[Software Requirements: 1](#_Toc137926565)

[High Level Design: 2](#_Toc137926566)

[Layered architecture: 2](#_Toc137926567)

[Module Description 3](#_Toc137926568)

[Driver Documentations 4](#_Toc137926569)

[Button 4](#_Toc137926570)

[LED 5](#_Toc137926571)

[GPIO 6](#_Toc137926572)

[Low Level Design: 8](#_Toc137926573)

[Flowchart 8](#_Toc137926574)

[LED 8](#_Toc137926575)

[BUTTON 12](#_Toc137926576)

[GPIO 14](#_Toc137926577)

[APPLICATION 19](#_Toc137926578)

# Project Introduction:

The project aims to develop a system that utilizes the TivaC board, incorporating both hardware and software components. The hardware requirements include the use of the TivaC board, which serves as the main platform for the system. Additionally, the system utilizes the SW1 button as an input and the RGB LED for visual feedback.

The software requirements outline the expected behavior of the system. Upon initialization, the RGB LED is initially turned off. The system operates based on four distinct states, triggered by the pressing of the SW1 button. Each press of SW1 corresponds to a specific behavior of the Green LED, controlled by duty cycles.

In the first state, upon the first press of SW1, the Green LED turns on with a 30% duty cycle. This duty cycle indicates the ratio of time the LED is on compared to the total time of the PWM signal, which has a predefined duration of 500 milliseconds.

In the second state, upon the second press of SW1, the Green LED remains on, but with an increased duty cycle of 60%. This adjustment results in a brighter illumination of the LED.

The third state is triggered by the third press of SW1, causing the Green LED to shine with a duty cycle of 90%. This setting provides the brightest illumination among the defined states.

Finally, the fourth press of SW1 transitions the system into the fourth state, where the Green LED is turned off entirely. This state represents a state of rest for the LED.

Upon the fifth press of SW1, the system returns to the first state, allowing the user to cycle through the defined states again.

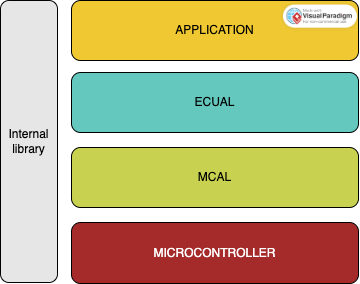
To implement the system, it is necessary to develop drivers, including the General Purpose Timer (GPT) driver. The GPT driver should offer various APIs (Application Programming Interfaces) for different functionalities. These include initialization of the timer, starting the timer with specified durations in microseconds, milliseconds, and seconds, stopping the timer, retrieving the elapsed time, retrieving the remaining time, enabling interrupts, disabling interrupts, and defining a callback function.

By meeting these detailed requirements, the system will effectively control the behavior of the Green LED based on the user's interactions with the SW1 button. The utilization of the TivaC board and the development of the necessary drivers will ensure proper functioning and accurate control of the LED.

# High Level Design:

## Layered architecture:

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



## Module Description

1. Application
2. ECUAL
   1. LED
   2. Button
   3. PWM
3. MCAL
   1. DIO
   2. SYSTICK
   3. NVIC
   4. SYS\_CONTROL
   5. TIMERS
4. Microcontroller

A screenshot of a computer

Description automatically generated with medium confidence

## Driver Documentations

## Button

**Module Description** The push button module provides functions for initializing a push button, reading its state, and configuring its properties. It utilizes the DIO (Digital Input/Output) module for pin configuration and manipulation.

**Macro Declarations** No macro declarations are included in the provided code.

**Macro Function Declarations** No macro function declarations are included in the provided code.

**Data Types Declarations**

* **btn\_enu\_btn\_state\_t**: An enumerated type representing the possible states of the button. It has two values: BUTTON\_RELEASED and BUTTON\_PRESSED.
* **btn\_enu\_\_btn\_active\_t**: An enumerated type representing the active state of the button. It has two values: BUTTON\_ACTIVE\_LOW and BUTTON\_ACTIVE\_HIGH.
* **button\_str\_btn\_config\_t**: A structure representing the configuration of the button. It contains the following members:
  + **port\_name**: An instance of the **dio\_enu\_portx\_t** enum, representing the port name of the button pin.
  + **pin**: An instance of the **dio\_enu\_pinx\_t** enum, representing the pin number of the button.
  + **button\_state**: An instance of the **btn\_enu\_btn\_state\_t** enum, representing the initial state of the button.
  + **button\_active**: An instance of the **btn\_enu\_\_btn\_active\_t** enum, representing the active state of the button.

**Function Declarations**

* **button\_initializa**: Initializes the push button by configuring its pin and setting the initial state. It takes a pointer to a **button\_str\_btn\_config\_t** structure as an argument. Returns a value of type **brn\_enu\_return\_state\_t** indicating the success or failure of the initialization.
* **button\_read\_state**: Reads the state of the push button and updates the **btn\_enu\_btn\_state\_t** variable pointed to by **ptr\_enu\_btn\_state**. It takes a pointer to a **button\_str\_btn\_config\_t** structure and a pointer to a **btn\_enu\_btn\_state\_t** variable as arguments. Returns a value of type **brn\_enu\_return\_state\_t** indicating the success or failure of reading the state.

## LED

The LED module provides functions for initializing an LED, turning it on and off, and toggling its state. It utilizes the DIO (Digital Input/Output) module for pin configuration and manipulation.

**Macro Declarations** No macro declarations are included in the provided code.

**Macro Function Declarations** No macro function declarations are included in the provided code.

**Data Types Declarations**

* **led\_enu\_status\_t**: An enumerated type representing the possible states of the LED. It has two values: LED\_OFF and LED\_ON.
* **led\_str\_led\_config\_t**: A structure representing the configuration of the LED. It contains the following members:
  + **port\_name**: An instance of the **dio\_enu\_portx\_t** enum, representing the port name of the LED pin.
  + **pin**: An instance of the **dio\_enu\_pinx\_t** enum, representing the pin number of the LED.
  + **led\_status**: An instance of the **led\_enu\_status\_t** enum, representing the initial state of the LED.

**Function Declarations**

* **led\_initialization**: Initializes the LED by configuring its pin and setting the initial state. It takes a pointer to a **led\_str\_led\_config\_t** structure as an argument. Returns a value of type **led\_enu\_return\_state\_t** indicating the success or failure of the initialization.
* **led\_turn\_on**: Turns on the LED by setting the appropriate pin to the active state. It takes a pointer to a **led\_str\_led\_config\_t** structure as an argument. Returns a value of type **led\_enu\_return\_state\_t** indicating the success or failure of turning on the LED.
* **led\_turn\_off**: Turns off the LED by setting the appropriate pin to the inactive state. It takes a pointer to a **led\_str\_led\_config\_t** structure as an argument. Returns a value of type **led\_enu\_return\_state\_t** indicating the success or failure of turning off the LED.
* **led\_toggle**: Toggles the state of the LED. If the LED is currently on, it will be turned off, and vice versa. It takes a pointer to a **led\_str\_led\_config\_t** structure as an argument. Returns a value of type **led\_enu\_return\_state\_t** indicating the success or failure of toggling the LED.

## PWM

The Software PWM module provides functions for generating a software-based PWM signal. It allows controlling the duty cycle, frequency, and state of the PWM output. The module utilizes the TIMERS module for precise timing and the DIO module for pin configuration and manipulation.

**Macro Declarations**

No macro declarations are included in the provided code.

**Macro Function Declarations**

No macro function declarations are included in the provided code.

**Data Types Declarations**

* **sw\_pwm\_enu\_return\_state\_t**: An enumerated type representing the possible return states of the Software PWM module. It includes the following values:
  + **SW\_PWM\_NOT\_OK**: Indicates an error or failure.
  + **SW\_PWM\_OK**: Indicates successful execution.
  + **SW\_PWM\_NULL\_PTR**: Indicates a null pointer error.
  + **SW\_PWM\_MAX\_INVALID\_LOAD\_VALUE**: Indicates an invalid maximum load value error.
* **sw\_pwm\_enu\_pwm\_state\_t**: An enumerated type representing the states of the PWM signal. It includes the following values:
  + **PWM\_OFF**: Indicates the PWM signal is turned off.
  + **PWM\_ON**: Indicates the PWM signal is turned on.
* **sw\_pwm\_configuration\_t**: A structure representing the configuration parameters of the Software PWM. It contains the following members:
  + **port\_index**: An instance of the **dio\_enu\_portx\_t** enum, representing the port name of the PWM output pin.
  + **pin\_index**: An instance of the **dio\_enu\_pinx\_t** enum, representing the pin number of the PWM output.
  + **duty\_cycle**: An 8-bit unsigned integer representing the initial duty cycle of the PWM signal.
  + **frequency**: A 32-bit unsigned integer representing the desired frequency of the PWM signal.
  + **t\_on**: A 16-bit unsigned integer representing the "on" time of the PWM signal.
  + **cycle\_duration**: A 16-bit unsigned integer representing the total duration of a PWM cycle.
  + **pwm\_state**: An instance of the **sw\_pwm\_enu\_pwm\_state\_t** enum, representing the initial state of the PWM signal.
  + **pwm\_tick\_ss**: A 64-bit unsigned integer representing the software timer tick value for precise timing.

**Function Declarations**

* **sw\_pwm\_enu\_return\_state\_t pwm\_init(sw\_pwm\_configuration\_t \*ptr\_str\_pwm\_configuration)**: Initializes the Software PWM module by configuring the PWM pin and setting the initial parameters. It takes a pointer to a **sw\_pwm\_configuration\_t** structure as an argument and returns a value of type **sw\_pwm\_enu\_return\_state\_t** indicating the success or failure of the initialization.
* **sw\_pwm\_enu\_return\_state\_t pwm\_change\_duty\_cycle(sw\_pwm\_configuration\_t \*ptr\_str\_pwm\_configuration)**: Changes the duty cycle of the PWM signal. It takes a pointer to a **sw\_pwm\_configuration\_t** structure as an argument and returns a value of type **sw\_pwm\_enu\_return\_state\_t** indicating the success or failure of changing the duty cycle.
* **void pwm\_start\_tick(void)**: Starts the software timer tick for precise timing of the PWM signal. This function is for internal use and should not be called directly.
* **sw\_pwm\_enu\_return\_state\_t pwm\_start(sw\_pwm\_configuration\_t \*ptr\_str\_pwm\_configuration)**: Starts generating the PWM signal. It takes a pointer to a **sw\_pwm\_configuration\_t** structure as an argument and returns a value of type **sw\_pwm\_enu\_return\_state\_t** indicating the success or failure of starting the PWM signal.
* **sw\_pwm\_enu\_return\_state\_t pwm\_checking(sw\_pwm\_configuration\_t \*ptr\_str\_pwm\_configuration)**: Checks the status of the PWM signal. It takes a pointer to a **sw\_pwm\_configuration\_t** structure as an argument and returns a value of type **sw\_pwm\_enu\_return\_state\_t** indicating the current status of the PWM signal.
* **sw\_pwm\_enu\_return\_state\_t pwm\_stop(sw\_pwm\_configuration\_t \*ptr\_str\_pwm\_configuration)**: Stops generating the PWM signal. It takes a pointer to a **sw\_pwm\_configuration\_t** structure as an argument and returns a value of type **sw\_pwm\_enu\_return\_state\_t** indicating the success or failure of stopping the PWM signal.
* **void pwm\_end\_tick(void)**: Ends the software timer tick. This function is for internal use and should not be called directly.

**Usage**

1. Include the necessary header files: **TIMERS\_interface.h**, **DIO\_interface.h**, and **STD\_TYPES.h**.
2. Define a configuration structure of type **sw\_pwm\_configuration\_t** and populate its members with the desired parameters for the PWM signal.
3. Call **pwm\_init()** with a pointer to the configuration structure to initialize the PWM module.
4. Use **pwm\_change\_duty\_cycle()** to update the duty cycle of the PWM signal if needed.
5. Call **pwm\_start()** to start generating the PWM signal.
6. To check the status of the PWM signal, use **pwm\_checking()**.
7. If necessary, call **pwm\_stop()** to stop generating the PWM signal.
8. Remember to clean up by calling **pwm\_end\_tick()** when the PWM module is no longer needed.

**Return States**

The Software PWM module functions return a value of type **sw\_pwm\_enu\_return\_state\_t** to indicate the success or failure of the corresponding operation. The possible return states are:

* **SW\_PWM\_NOT\_OK**: Indicates an error or failure.
* **SW\_PWM\_OK**: Indicates successful execution.
* **SW\_PWM\_NULL\_PTR**: Indicates a null pointer error.
* **SW\_PWM\_MAX\_INVALID\_LOAD\_VALUE**: Indicates an invalid maximum load value error.

**Dependencies**

The Software PWM module depends on the following:

* TIMERS module: Provides precise timing for the PWM signal.
* DIO module: Handles pin configuration and manipulation.
* STD\_TYPES module: Defines standard data types.

Ensure that the necessary dependencies are included and properly configured for the Software PWM module to function correctly.

*Note: The provided documentation assumes the existence and proper configuration of the dependencies and data types mentioned. Please ensure that they are defined and included appropriately in your project.*

## GPIO

The DIO interface module provides functions and data types for configuring and manipulating digital input/output pins. It allows you to enable clock for a specific port, initialize pins, set pin states, toggle pin states, and read pin states.

**Data Types Declarations**

* **dio\_enu\_portx\_t**: An enumerated type representing the available ports. It includes values from DIO\_PORTA to DIO\_PORTF, and MAX\_INVALID\_PORT.
* **dio\_enu\_pinx\_t**: An enumerated type representing the available pins. It includes values from DIO\_PIN\_0 to DIO\_PIN\_7, and MAX\_INVALID\_PIN.
* **dio\_enu\_pin\_state\_t**: An enumerated type representing the possible states of a pin. It has two values: DIO\_PIN\_LOW\_STATE and DIO\_PIN\_HIGH\_STATE.
* **dio\_enu\_pin\_mode\_t**: An enumerated type representing the mode of a pin. It includes values DIO\_PIN\_INPUT, DIO\_PIN\_OUTPUT, DIO\_PIN\_AFM, and DIO\_PIN\_ANALOG.
* **dio\_enu\_output\_type\_t**: An enumerated type representing the output type of a pin. It includes values DIO\_PIN\_OUTPUT\_PUSH\_PULL and DIO\_PIN\_OUTPUT\_OPEN\_DRAIN.
* **dio\_enu\_output\_current\_t**: An enumerated type representing the output current of a pin. It includes values DIO\_PIN\_2MA, DIO\_PIN\_4MA, and DIO\_PIN\_8MA.
* **dio\_str\_output\_type\_and\_speed\_and\_state\_t**: A structure representing the output type, speed, and state of a pin. It contains the following members:
  + **enu\_output\_type**: An instance of the **dio\_enu\_output\_type\_t** enum, representing the output type of the pin.
  + **enu\_output\_current**: An instance of the **dio\_enu\_output\_current\_t** enum, representing the output current of the pin.
  + **enu\_pin\_state**: An instance of the **dio\_enu\_pin\_state\_t** enum, representing the initial state of the pin.
* **dio\_enu\_input\_type\_t**: An enumerated type representing the input type of a pin. It includes values DIO\_PIN\_INPUT\_NO\_PULL\_UP\_NO\_PULL\_DOWN, DIO\_PIN\_INPUT\_PULL\_UP, and DIO\_PIN\_INPUT\_PULL\_DOWN.
* **dio\_un\_input\_output\_type\_t**: A union type representing either the input or output type of a pin. It contains the following members:
  + **str\_output\_type\_and\_speed\_and\_state**: An instance of the **dio\_str\_output\_type\_and\_speed\_and\_state\_t** structure representing the output type, speed, and state of the pin.
  + **enu\_input\_type**: An instance of the **dio\_enu\_input\_type\_t** enum representing the input type of the pin.
* **dio\_str\_pin\_Config\_t**: A structure representing the configuration of a pin. It contains the following members:
  + **enu\_port**: An instance of the **dio\_enu\_portx\_t** enum, representing the port of the pin.
  + **enu\_pin**: An instance of the **dio\_enu\_pinx\_t** enum, representing the pin number.
  + **enu\_pin\_mode**: An instance of the **dio\_enu\_pin\_mode\_t** enum, representing the mode of the pin.
  + **un\_input\_output\_type**: An instance of the **dio\_un\_input\_output\_type\_t** union, representing either the input or output type of the pin.
* **dio\_enu\_return\_state\_t**: An enumerated type representing the return states of the DIO functions. It includes values DIO\_NOT\_OK, DIO\_OK, DIO\_NULL\_PTR, and DIO\_EXCEED\_PORT.

**Function Declarations**

* **dio\_enable\_clock**: Enables the clock for a specific port. It takes a parameter of type **dio\_enu\_portx\_t** representing the port number. Returns a value of type **dio\_enu\_return\_state\_t** indicating the success or failure of enabling the clock.
* **dio\_init\_pin**: Initializes a pin by configuring its mode, input/output type, and state. It takes a pointer to a **dio\_str\_pin\_Config\_t** structure as an argument. Returns a value of type **dio\_enu\_return\_state\_t** indicating the success or failure of pin initialization.
* **dio\_set\_pin**: Sets the state of a pin to either high or low. It takes a pointer to a **dio\_str\_pin\_Config\_t** structure and the desired pin state as arguments. Returns a value of type **dio\_enu\_return\_state\_t** indicating the success or failure of setting the pin state.
* **dio\_toggle\_pin**: Toggles the state of a pin. If the pin is currently high, it will be set to low, and vice versa. It takes a pointer to a **dio\_str\_pin\_Config\_t** structure as an argument. Returns a value of type **dio\_enu\_return\_state\_t** indicating the success or failure of toggling the pin state.
* **dio\_read\_pin**: Reads the current state of a pin and stores it in a variable. It takes a pointer to a **dio\_str\_pin\_Config\_t** structure and a pointer to a **dio\_enu\_pin\_state\_t** variable as arguments. Returns a value of type **dio\_enu\_return\_state\_t** indicating the success or failure of reading the pin state.

## systick

The SysTick interface provides functions for interacting with the SysTick timer. It allows configuring the timer, setting timeouts, enabling/disabling the counter, and retrieving the state of the timer. This module is designed to work with the SysTick timer available on microcontrollers.

Data Types:

* ptr\_void\_func: Function pointer type for callback functions.
* systick\_enu\_reach\_zero\_flag\_t: Enumeration representing the reach-zero flag of the SysTick timer.
  + SYSTICK\_NOT\_REACH\_FLAG: SysTick timer has not reached zero.
  + SYSTICK\_REACH\_FLAG: SysTick timer has reached zero.
* systick\_enu\_clock\_source\_t: Enumeration representing the clock source for the SysTick timer.
  + SYSTICK\_PIOSC\_CLOCK\_PRE\_4: PIOSC (Precision Internal Oscillator) clock divided by 4.
  + SYSTICK\_SYSCLK\_CLOCK: System clock.
* systick\_enu\_int\_state\_t: Enumeration representing the interrupt state for the SysTick timer.
  + SYSTICK\_DISABLE\_INT: Disable interrupts.
  + SYSTICK\_ENABLE\_INT: Enable interrupts.
* systick\_enu\_counter\_state\_t: Enumeration representing the counter state for the SysTick timer.
  + SYSTICK\_DISABLE\_COUNTER: Disable the SysTick counter.
  + SYSTICK\_ENABLE\_COUNTER: Enable the SysTick counter.
* systick\_str\_systick\_config\_t: Structure representing the configuration settings for the SysTick timer.
  + enu\_clock\_source: Clock source for the SysTick timer.
  + enu\_int\_state: Interrupt state for the SysTick timer.
  + enu\_counter\_state: Counter state for the SysTick timer.
* systick\_enu\_return\_state\_t: Enumeration representing the return status of the SysTick functions.
  + SYSTICK\_NOT\_OK: General error or failure.
  + SYSTICK\_OK: Successful operation.
  + SYSTICK\_NULL\_PTR: Null pointer error.
  + SYSTICK\_MAX\_INVALID\_LOAD\_VALUE: Invalid maximum load value error.

Function Prototypes:

* systick\_enu\_return\_state\_t systick\_init(systick\_str\_systick\_config\_t \*ptr\_str\_systick\_config): Initializes the SysTick timer.
* systick\_enu\_return\_state\_t systick\_set\_timeout\_ms(uint32\_t time\_out): Sets the timeout value in milliseconds for the SysTick timer.
* systick\_enu\_return\_state\_t systick\_set\_callback(ptr\_void\_func ptr\_callback\_func): Sets the callback function to be called when the SysTick timer interrupt occurs.
* systick\_enu\_return\_state\_t systick\_counter\_enable(systick\_str\_systick\_config\_t \*ptr\_str\_systick\_config): Enables the SysTick counter with the specified configuration.
* systick\_enu\_return\_state\_t systick\_counter\_disable(systick\_str\_systick\_config\_t \*ptr\_str\_systick\_config): Disables the SysTick counter with the specified configuration.
* systick\_enu\_return\_state\_t systick\_get\_state(systick\_enu\_reach\_zero\_flag\_t \*ptr\_enu\_reach\_zero\_flag): Retrieves the state of the SysTick timer (reach-zero flag).

Note: This interface assumes the presence of additional header files, such as "STD\_TYPES.h", for data type definitions.

Dependencies:

* "../../STD\_LIBRARIES/STD\_TYPES.h": Header file defining standard data types used by the interface.

Usage:

1. Include the "systick\_interface.h" header file in your code.
2. Configure the systick\_str\_systick\_config\_t structure with the desired settings for the SysTick timer.
3. Initialize the SysTick timer using the systick\_init function with the configuration structure.
4. Set the timeout value using the systick\_set\_timeout\_ms

## NVIC

The NVIC module provides functions for enabling and disabling interrupts for various peripherals in the microcontroller. It utilizes the NVIC (Nested Vectored Interrupt Controller) to manage and prioritize interrupts.

**Macro Declarations**

No macro declarations are included in the provided code.

**Data Types Declarations**

* **nvic\_enu\_interupt\_index\_t**: An enumerated type representing the indices of different peripherals or interrupt sources in the microcontroller. It includes the following values:
  + **GPIO\_A** to **GPIO\_E**: Represents GPIO ports A to E.
  + **UART\_0** and **UART\_1**: Represents UART modules 0 and 1.
  + **SSI\_0**: Represents Synchronous Serial Interface (SSI) module 0.
  + **I\_2C\_0**: Represents I2C module 0.
  + **PWM0\_Fault**, **PWM0\_Generator\_0**, **PWM0\_Generator\_1**, and **PWM0\_Generator\_2**: Represents PWM (Pulse Width Modulation) faults and generators of PWM0.
  + **QEI\_0**: Represents Quadrature Encoder Interface (QEI) module 0.
  + **ADC0\_Sequence\_0** to **ADC0\_Sequence\_3**: Represents ADC0 sequences 0 to 3.
  + **Watchdog\_Timers\_0\_1**: Represents Watchdog Timers 0 and 1.
  + **Timer\_0\_A**, **Timer\_0\_B**, **Timer\_1\_A**, **Timer\_1\_B**, **Timer\_2\_A**, and **Timer\_2\_B**: Represents Timer modules and their corresponding capture/compare channels.
  + **Analog\_Comparator\_0** and **Analog\_Comparator\_1**: Represents Analog Comparators 0 and 1.
  + **NVIC\_MAX\_INVALID\_INT**: Indicates an invalid interrupt index.
* **nvic\_enu\_return\_state\_t**: An enumerated type representing the return states of the NVIC module functions. It includes the following values:
  + **NVIC\_NOT\_OK**: Indicates an error or failure.
  + **NVIC\_OK**: Indicates successful execution.
  + **NVIC\_NULL\_PTR**: Indicates a null pointer error.
  + **NVIC\_MAX\_INVALID\_LOAD\_VALUE**: Indicates an invalid maximum load value error.

**Function Declarations**

* **nvic\_enu\_return\_state\_t nvic\_EnablePeripheral(nvic\_enu\_interupt\_index\_t copy\_enu\_interupt\_index)**: Enables the specified peripheral interrupt. It takes an **nvic\_enu\_interupt\_index\_t** value as an argument and returns a value of type **nvic\_enu\_return\_state\_t** indicating the success or failure of enabling the interrupt.
* **nvic\_enu\_return\_state\_t nvic\_DisablePeripheral(nvic\_enu\_interupt\_index\_t copy\_enu\_interupt\_index)**: Disables the specified peripheral interrupt. It takes an **nvic\_enu\_interupt\_index\_t** value as an argument and returns a value of type **nvic\_enu\_return\_state\_t** indicating the success or failure of disabling the interrupt.

**Usage**

1. Include the necessary header file: **NVIC\_interface.h**.
2. Use the **nvic\_EnablePeripheral()** function to enable the interrupt of a specific peripheral by providing the appropriate **nvic\_enu\_interupt\_index\_t** value.
3. Use the **nvic\_DisablePeripheral()** function to disable the interrupt of a specific peripheral by providing the appropriate **nvic\_enu\_interupt\_index\_t** value.

**Return States**

The NVIC module functions return a value of type **nvic\_enu\_return\_state\_t** to indicate the success or failure of the corresponding operation. The possible return states are:

* **NVIC\_NOT\_OK**: Indicates an error or failure.
* **NVIC\_OK**: Indicates successful execution.
* **NVIC\_NULL\_PTR**: Indicates a null pointer error.
* **NVIC\_MAX\_INVALID\_LOAD\_VALUE**: Indicates an invalid maximum load value error.

## SYS\_CONTROL

The Sys Control module provides functions for setting the clock system in the microcontroller. It allows configuring the clock source and prescaler settings.

Macro Declarations

No macro declarations are included in the provided code.

Data Types Declarations

sys\_control\_enu\_OSCSRC\_t: An enumerated type representing the different clock sources available for the microcontroller. It includes the following values:

SYS\_CONTROL\_MOSC: Main Oscillator (MOSC) as the clock source.

SYS\_CONTROL\_PIOSC: Precision Internal Oscillator (PIOSC) as the clock source.

SYS\_CONTROL\_PIOSC\_4: Precision Internal Oscillator divided by 4 (PIOSC/4) as the clock source.

SYS\_CONTROL\_LFIOSC: Low Frequency Internal Oscillator (LFIOSC) as the clock source.

sys\_control\_str\_config\_t: A structure representing the configuration of the clock system. It contains the following members:

prescaler: A 4-bit field representing the prescaler value for the clock system.

enu\_OSCSRC: An instance of the sys\_control\_enu\_OSCSRC\_t enum, representing the clock source selection.

Function Declarations

void sys\_control\_set\_clock\_system(sys\_control\_str\_config\_t \*ptr\_str\_sys\_control\_config): Sets the clock system configuration based on the provided sys\_control\_str\_config\_t structure. It takes a pointer to the configuration structure as an argument.

Usage

Include the necessary header file: SYS\_CONTROL\_interface.h.

Create an instance of the sys\_control\_str\_config\_t structure and initialize its members:

Set the desired prescaler value in the prescaler member.

Select the clock source by assigning the appropriate value from the sys\_control\_enu\_OSCSRC\_t enum to the enu\_OSCSRC member.

Call the sys\_control\_set\_clock\_system() function, passing the address of the configuration structure as the argument.

## TIMER

The Timers module provides functions for initializing, configuring, and controlling timers in the microcontroller. It supports various timer modes and counting modes, allowing for a wide range of timer applications.

**Macro Declarations**

No macro declarations are included in the provided code.

**Data Types Declarations**

* **ptr\_void\_func**: A function pointer type representing a callback function without any arguments and return value.
* **timers\_enu\_timer\_counting\_mode\_t**: An enumerated type representing the counting mode of the timer. It includes the following values:
  + **TIIMER\_COUNT\_DOWN**: Timer counts down from the specified value.
  + **TIIMER\_COUNT\_UP**: Timer counts up from 0.
* **timers\_enu\_timer\_mode\_t**: An enumerated type representing the mode of operation for the timer. It includes the following values:
  + **TIIMER\_ONE\_SHOT**: Timer operates in one-shot mode, where it triggers an interrupt or callback function once and stops.
  + **TIMER\_PERIODIC**: Timer operates in periodic mode, where it triggers an interrupt or callback function repeatedly at regular intervals.
  + **TIMER\_CAPTURE**: Timer operates in capture mode, where it captures the value of an external signal on each event.
* **timers\_enu\_timer\_a\_b\_t**: An enumerated type representing the timer unit within a timer module. It includes the following values:
  + **TIMER\_A**: Timer A unit.
  + **TIMER\_CONCATINATED**: Concatenated timer unit (used for certain timers).
  + **TIMER\_B**: Timer B unit.
* **timers\_enu\_timer\_bit\_resolution\_t**: An enumerated type representing the bit resolution of the timer. It includes the following values:
  + **TIMER\_32\_64**: 32/64-bit timer mode.
  + **TIMER\_RTC\_32\_64**: 32/64-bit real-time clock timer mode.
  + **TIMER\_16\_32**: 16/32-bit timer mode.
* **timers\_enu\_timer\_index\_t**: An enumerated type representing the index of the timer module. It includes values for different timers available in the microcontroller.
* **timers\_enu\_timer\_int\_number\_t**: An enumerated type representing the interrupt numbers associated with specific timers. It includes values for different timer interrupts.
* **timers\_str\_timer\_config\_t**: A structure representing the configuration of a timer. It contains the following members:
  + **timer\_index**: An instance of the **timers\_enu\_timer\_index\_t** enum, representing the index of the timer module.
  + **timer\_bit\_resolution**: An instance of the **timers\_enu\_timer\_bit\_resolution\_t** enum, representing the bit resolution of the timer.
  + **timer\_mode**: An instance of the **timers\_enu\_timer\_mode\_t** enum, representing the mode of operation for the timer.
  + **timer\_a\_b**: An instance of the **timers\_enu\_timer\_a\_b\_t** enum, representing the timer unit within the timer module.
  + **timer\_counting\_mode**: An instance of the **timers\_enu\_timer\_counting\_mode\_t** enum, representing the counting mode of the timer.
* **timers\_enu\_return\_state\_t**: An enumerated type representing the return state of the timer functions. It includes the following values:
  + **TIMERS\_NOT\_OK**: Timer operation encountered an error.
  + **TIMERS\_OK**: Timer operation was successful.
  + **TIMERS\_NULL\_PTR**: Null pointer argument was passed to a timer function.
  + **TIMERS\_MAX\_INVALID\_LOAD\_VALUE**: Invalid load value was specified for the timer.

**Function Declarations**

* **timers\_enu\_return\_state\_t timers\_init(timers\_str\_timer\_config\_t \*ptr\_str\_timer\_config)**: Initializes the timer module based on the provided configuration structure. It takes a pointer to the configuration structure as an argument.
* **timers\_enu\_return\_state\_t timers\_set\_timeout\_ms(timers\_str\_timer\_config\_t \*ptr\_str\_timer\_config, uint32\_t copy\_u32\_time\_out)**: Sets the timeout value in milliseconds for the specified timer. It takes a pointer to the configuration structure and the timeout value as arguments.
* **timers\_enu\_return\_state\_t timers\_set\_timeout\_us(timers\_str\_timer\_config\_t \*ptr\_str\_timer\_config, uint32\_t copy\_u32\_time\_out)**: Sets the timeout value in microseconds for the specified timer. It takes a pointer to the configuration structure and the timeout value as arguments.
* **timers\_enu\_return\_state\_t timers\_set\_callback(timers\_str\_timer\_config\_t \*ptr\_str\_timer\_config, ptr\_void\_func ptr\_callback\_func)**: Sets the callback function to be executed when the timer triggers an interrupt or event. It takes a pointer to the configuration structure and the function pointer of the callback function as arguments.
* **timers\_enu\_return\_state\_t timers\_enable(timers\_str\_timer\_config\_t \*ptr\_str\_timer\_config)**: Enables the specified timer. It takes a pointer to the configuration structure as an argument.
* **timers\_enu\_return\_state\_t timers\_disable(timers\_str\_timer\_config\_t \*ptr\_str\_timer\_config)**: Disables the specified timer. It takes a pointer to the configuration structure as an argument.

**Usage**

1. Include the necessary header file: **TIMERS\_interface.h**.
2. Create an instance of the **timers\_str\_timer\_config\_t** structure and initialize its members:
   * Set the desired timer index in the **timer\_index** member.
   * Select the appropriate timer bit resolution using the **timer\_bit\_resolution** member.
   * Choose the timer mode using the **timer\_mode** member.
   * Specify the timer unit within the module using the **timer\_a\_b** member.
   * Set the counting mode using the **timer\_counting\_mode** member.
3. Call the timer functions as needed:
   * Use **timers\_init()** to initialize the timer module with the provided configuration.
   * Use **timers\_set\_timeout\_ms()** or **timers\_set\_timeout\_us()** to set the timeout value for the timer.
   * Use **timers\_set\_callback()** to assign a callback function to be executed when the timer triggers an interrupt or event.
   * Use **timers\_enable()** to enable the timer.
   * Use **timers\_disable()** to disable the timer.

# Low Level Design:

## Flowchart

## LED

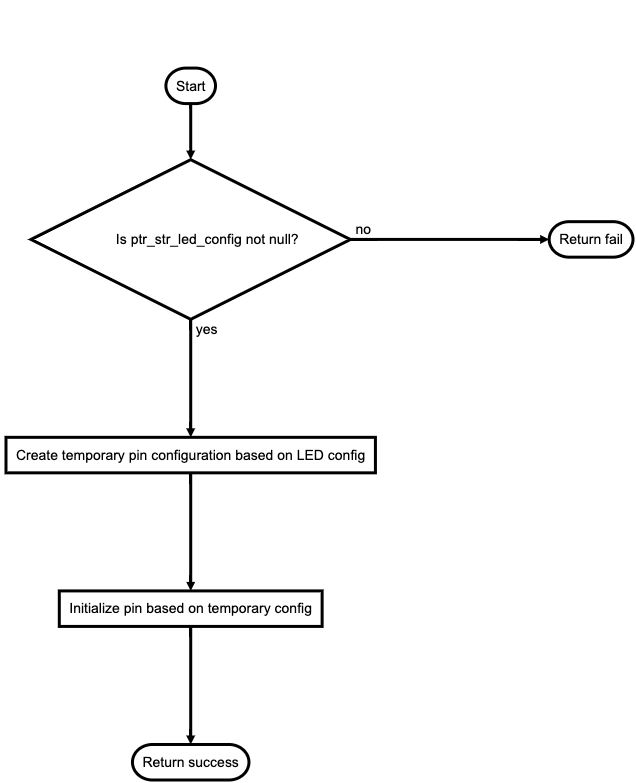


Figure 1 led\_initialization

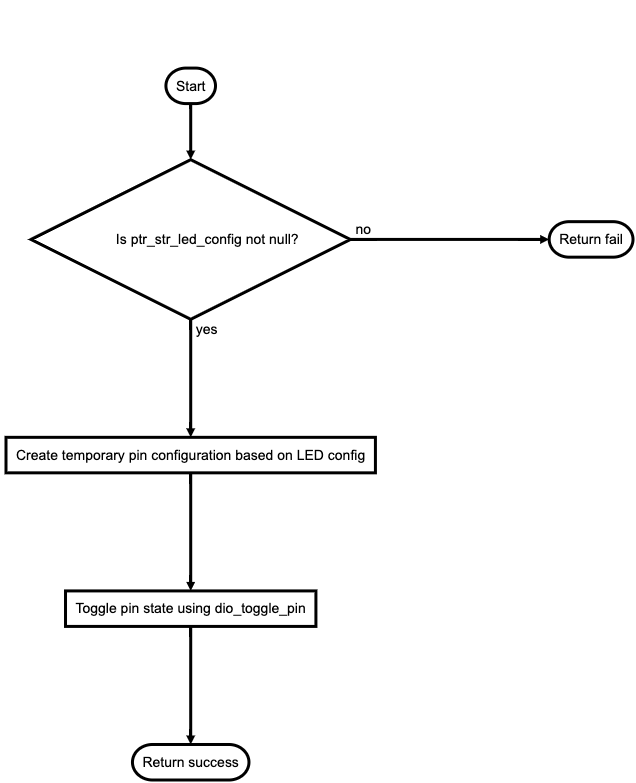


Figure 2 led\_turn\_on

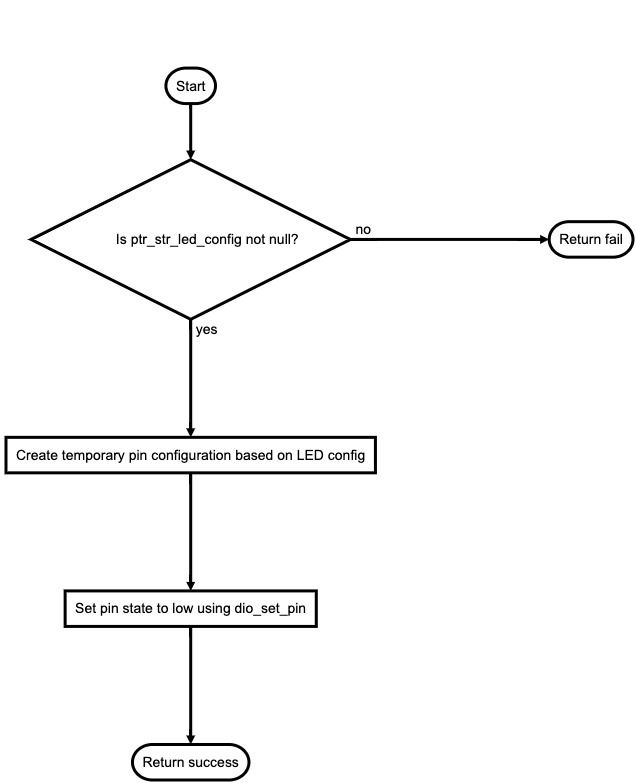


Figure 3 led\_turn\_off

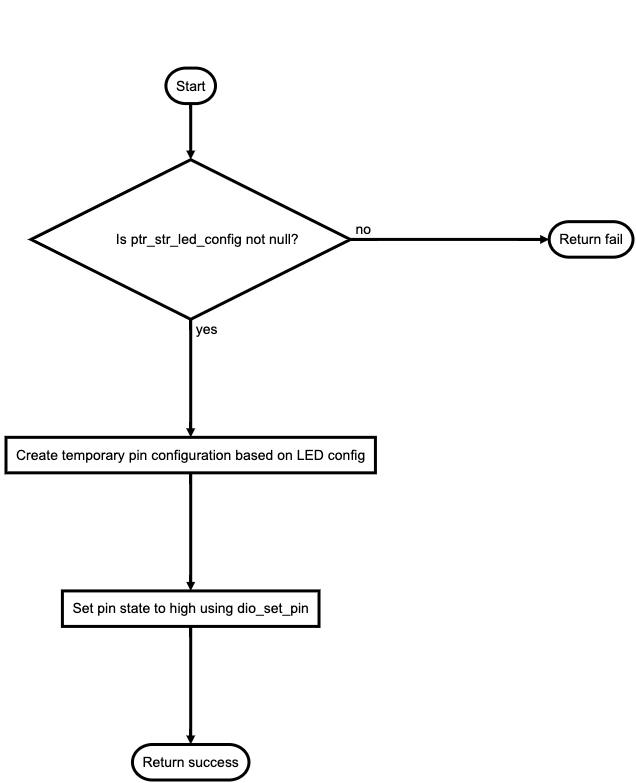


Figure 4 led\_toggle

## BUTTON

A screenshot of a computer

Description automatically generated with medium confidence

Figure 5 button\_initialization

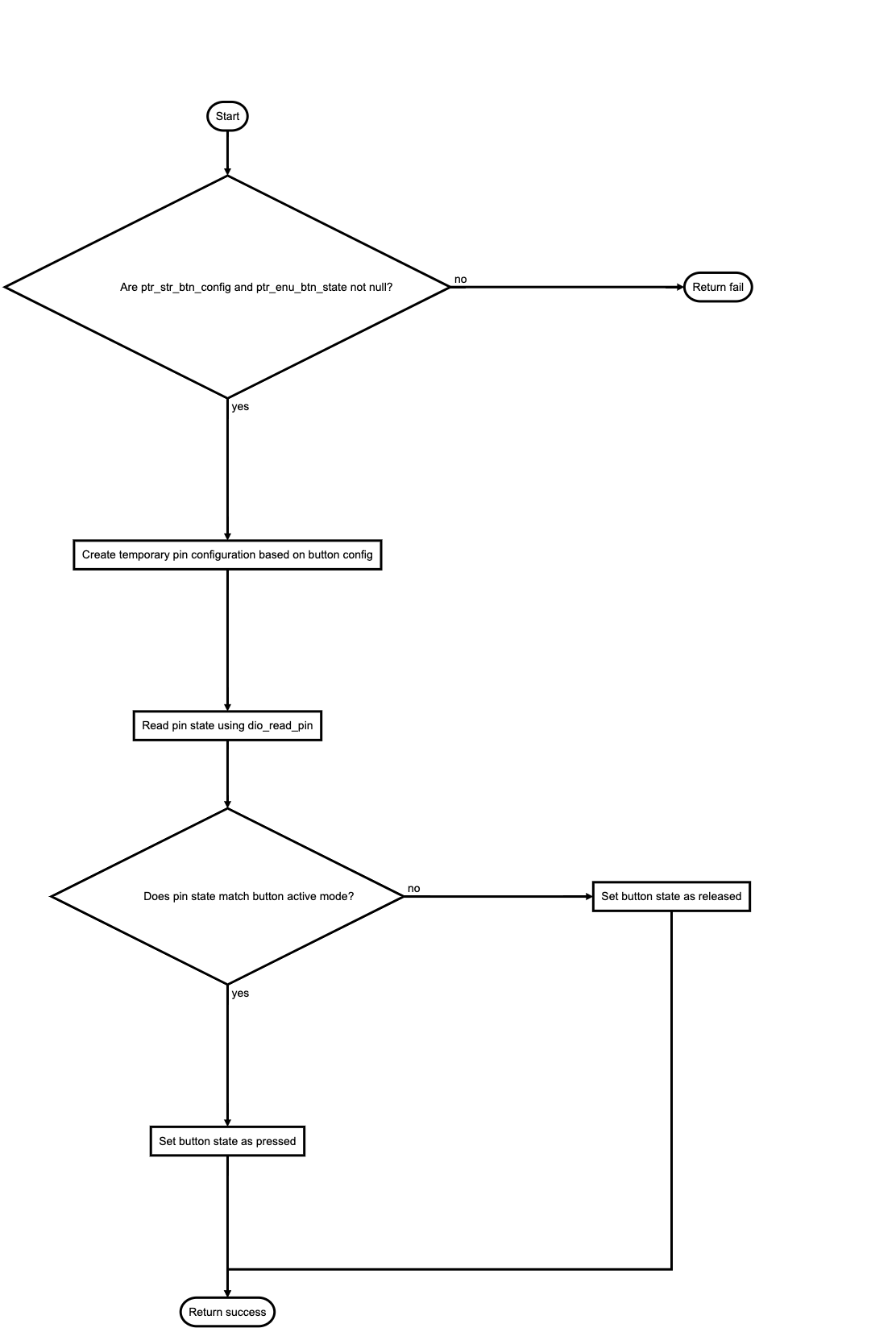


Figure 6 button\_read\_state

## GPIO

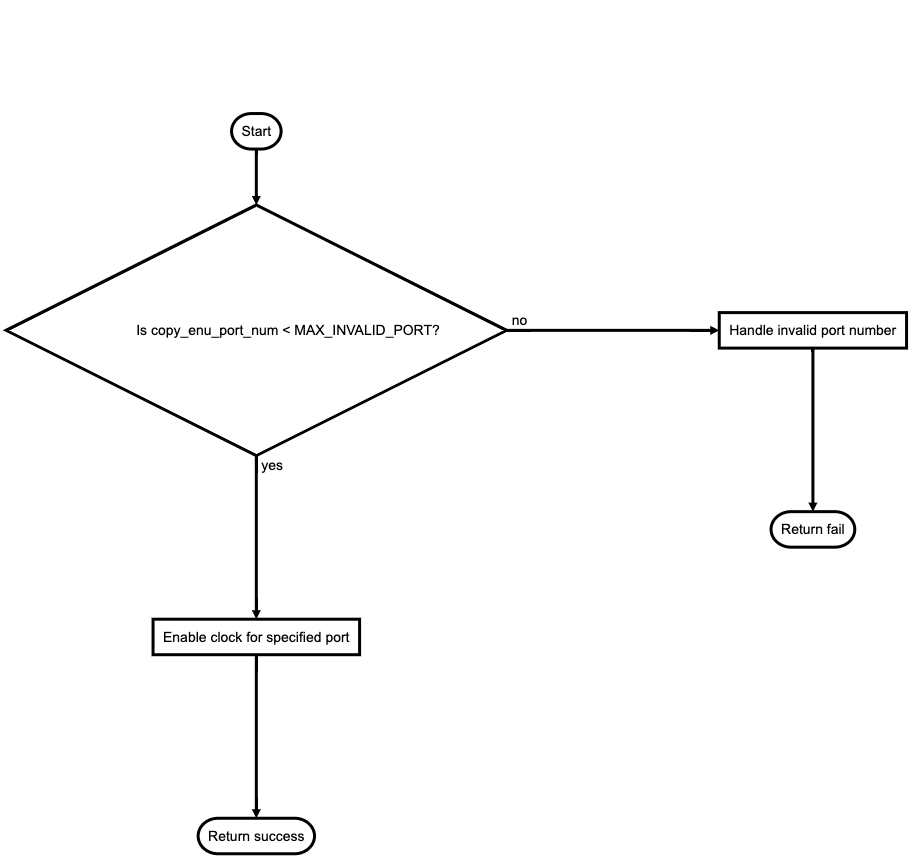


Figure 7 dio\_enable\_clock

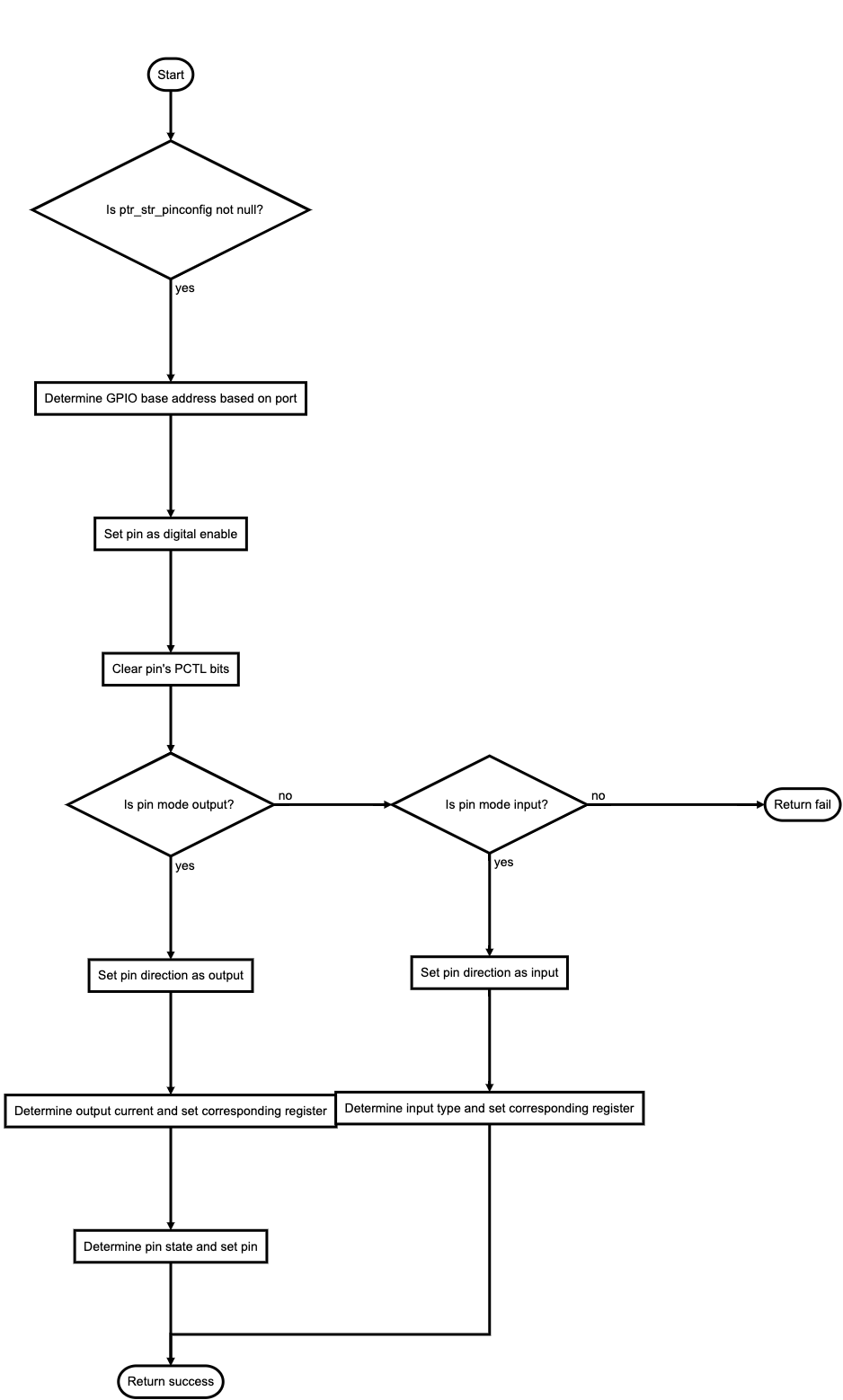


Figure 8 dio\_enable\_clock

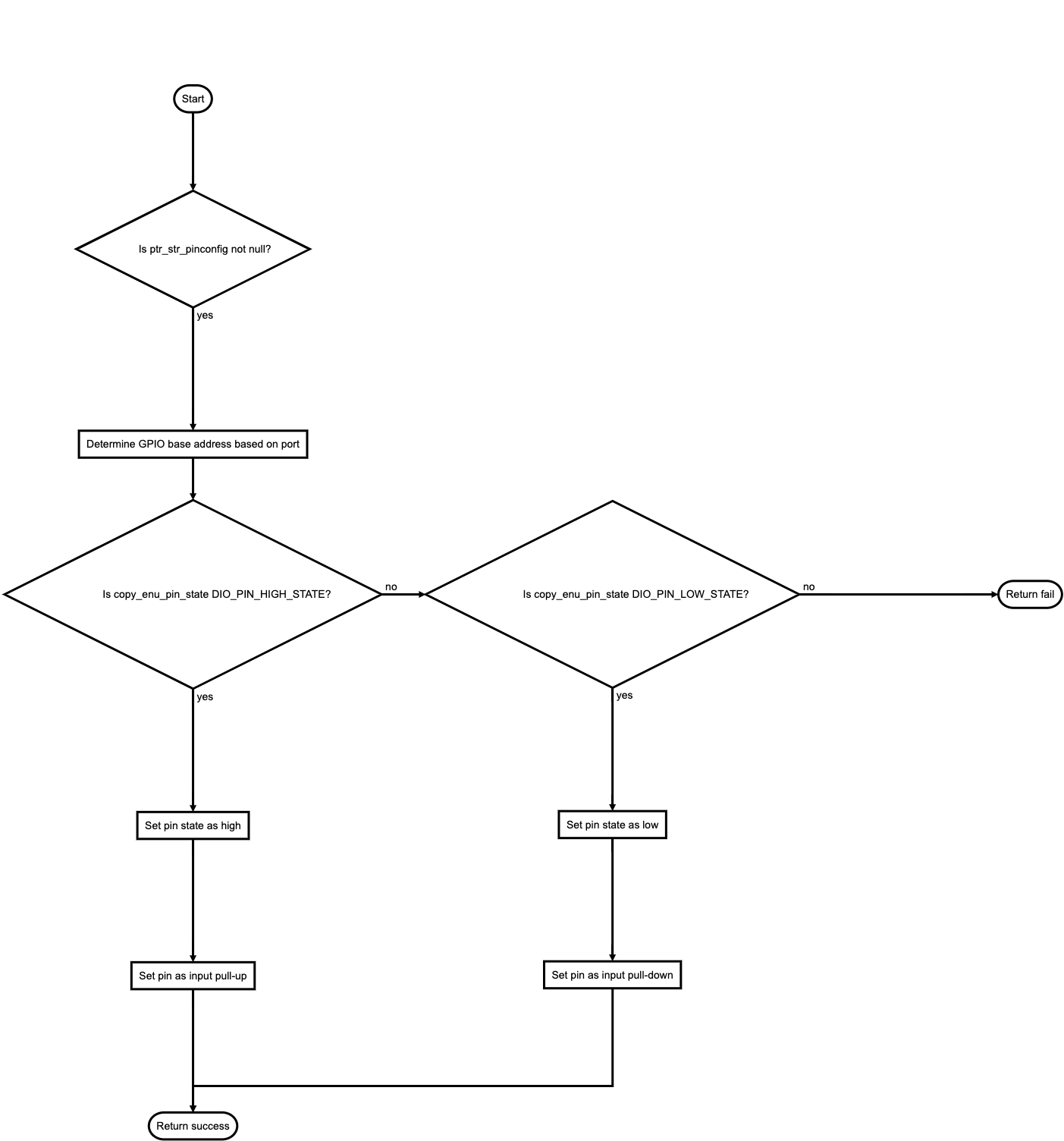


Figure 9 dio\_set\_pin

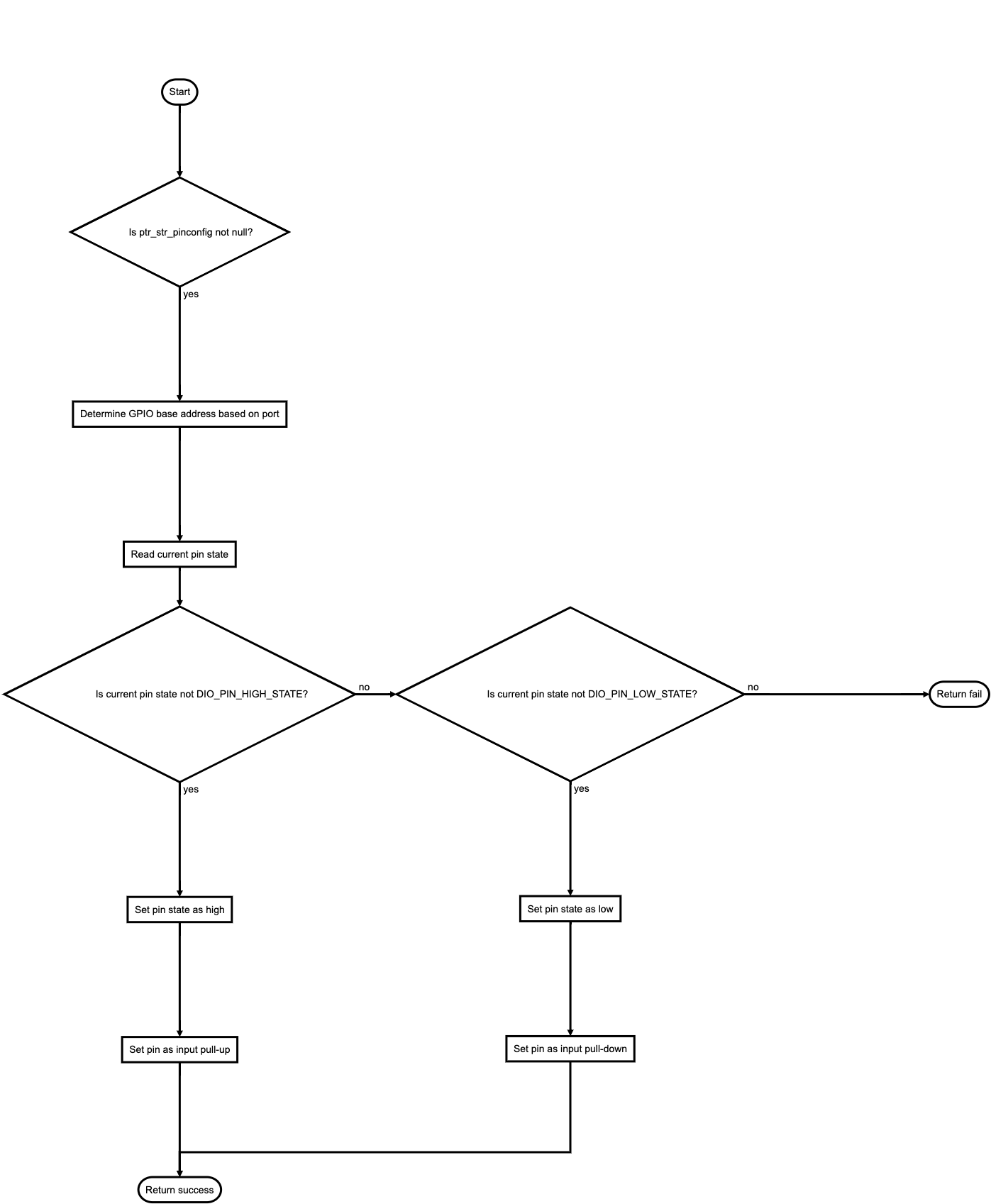


Figure 10 dio\_toggle\_pin

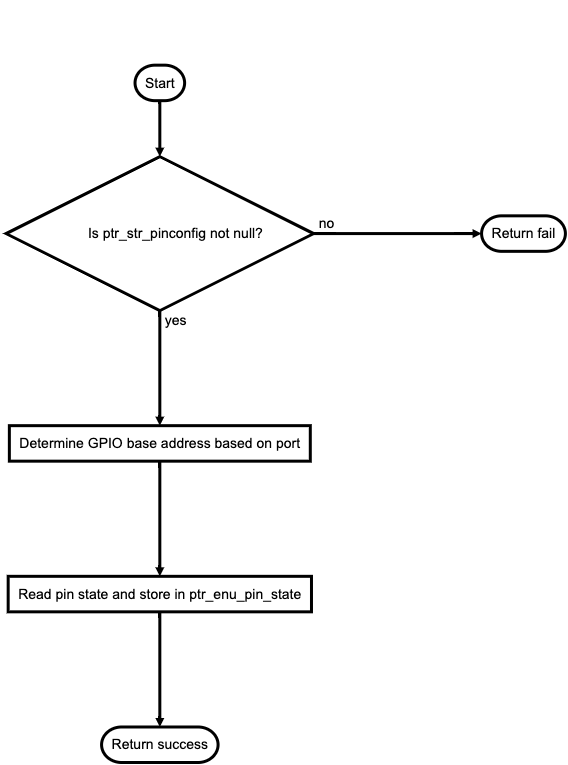


Figure 11 dio\_read\_pin

## APPLICATION