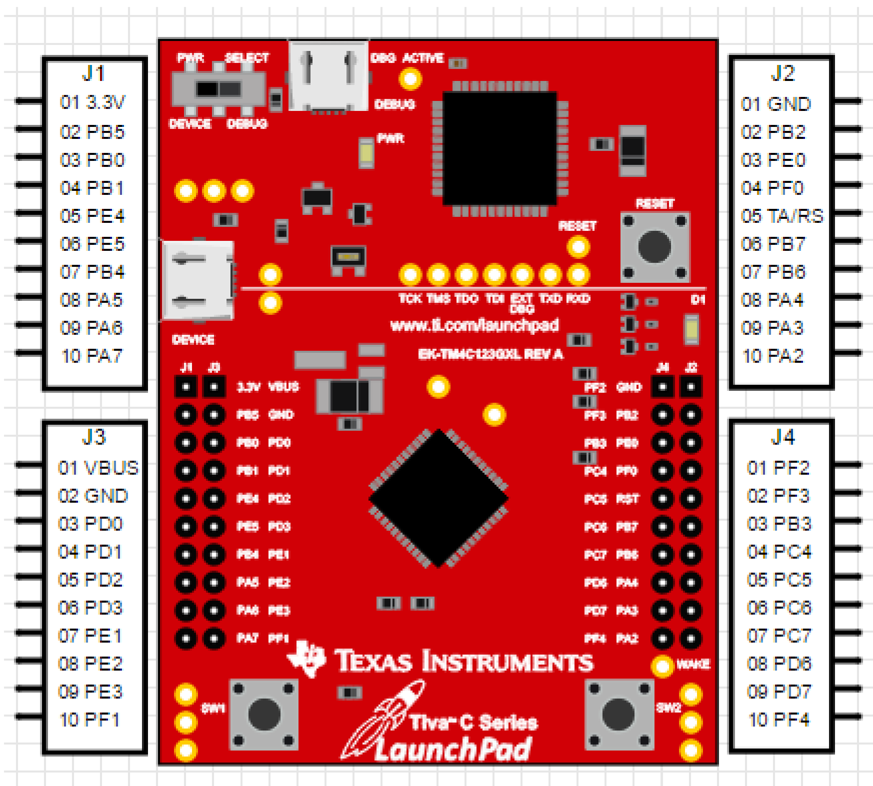
# RGB LED Brightness Control V1.0 Design

**BRIGHTSKIES EMBEDDED SYSTEMS BOOT CAMP PROJECT AT SPRINTS**




By

Mohamed El-Greatly

# Contents

# INTRODUCTION

## RGB LED Brightness Control Design:

-------------------------------------

The RGB LED Brightness Control Design is a crucial element in embedded system applications, providing a sophisticated mechanism for controlling the brightness levels of Red, Green, and Blue (RGB) LEDs. In this design, Software PWM based on timer load and match is employed to precisely control the brightness of each LED, offering a dynamic and visually appealing lighting experience. The control mechanism is initiated through an external button, allowing users to cycle through different brightness levels with each press.

Application Scenario:

-------------------------------------

This scenario illustrates the application of the RGB LED Brightness Control Design, where an embedded system features an RGB LED undergoing changes in brightness levels based on specific application requirements. The control is initiated through an external button, triggering a sequence of brightness levels for each LED, offering a dynamic visual effect.

Scenario Description:

-------------------------------------

This application scenario emphasizes testing the RGB LED Brightness Control Design's functionality, focusing on Software PWM for controlling LED brightness levels. The embedded system allows users to cycle through different brightness levels synchronized with the button press, creating a visually engaging lighting experience.

RGB LED Brightness Control:

-------------------------------------

The RGB LED Brightness Control module utilizes Software PWM based on timer load and match to precisely control the brightness levels of each LED. The button interrupts are employed to cycle through different brightness levels, creating a dynamic and visually appealing lighting effect.

External Button:

-------------------------------------

An external button is programmed to trigger a change in the RGB LED brightness levels. Each button press increments the sequence, cycling through predefined brightness levels for each LED. The sequence includes LED OFF, 30%, 60%, 90%, and repeats, creating an engaging and dynamic lighting pattern.

RGB LED Brightness Control Module:

-------------------------------------

The RGB LED Brightness Control Module, positioned in the Application layer, employs Software PWM based on timer load and match to control the brightness of each LED. This module continues to handle LED initialization, brightness control, and behaviors, ensuring an efficient and visually engaging lighting experience.

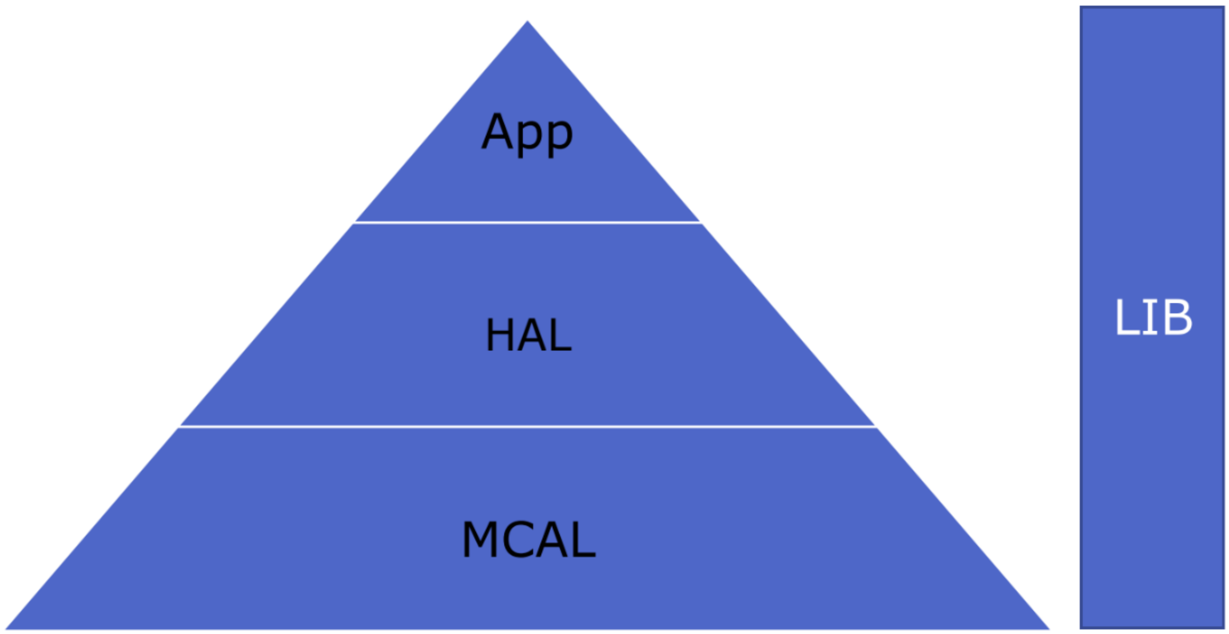
Conclusion:

-------------------------------------

The RGB LED Brightness Control Design, utilizing Software PWM based on timer load and match, provides a sophisticated solution for controlling RGB LED brightness levels. This scenario demonstrates the adaptability of the design in creating dynamic and visually appealing lighting effects in a real-world embedded system context, offering users an interactive and engaging experience through precise brightness control.

# HIGH-LEVEL DESIGN

## Layered architecture:




1. Application Layer (APP):

----------------------------------

- The topmost layer in an embedded software stack.
- Contains application-specific code and logic.
- Implements the primary functionality of the embedded system.

2. Hardware Abstraction Layer (HAL):

-------------------------------------------------

- Abstracts low-level hardware details from the Application and Services Layers.
- Provides a consistent and hardware-independent interface for hardware interaction.
- Eases portability across different microcontroller platforms.

3. Microcontroller Abstraction Layer (MCAL):

----------------------------------------------------------

- Specific to a particular microcontroller family or model.

- Provides low-level access to the microcontroller's hardware features.
- Tightly coupled with the microcontroller's hardware.

4. Library Layer:

----------------------

- Contains reusable software libraries and components.
- Offers functions for common tasks, such as math operations or communication protocols.
- Designed for code reusability and to save development time.

---

## Modules:

---

Tiva C Modules have been extensively utilized in the design, abstracting, and controlling the different hardware components and interfaces needed. The core modules used can be categorized under various layers:

### 1. GPIO (General Purpose Input/Output module):

---------------------------------------------------------------------------

- The GPIO module is part of the Microcontroller Abstraction Layer (MCAL) and aids in abstracting and managing the microcontroller's pins crucial for interfacing with various hardware components, e.g., LEDs, and buttons.
- The module's functions consist of GPIO initialization, determining pin direction (input/output), defining pin drives' strength, setting pull-up/pull-down resistors, enabling and handling pin interrupts and so forth.
- This module provides flexibility and efficiency in controlling and interacting with hardware elements at the raw pin level.

---

### 2. GPT (General-Purpose Timer module):

---------------------------------------------------------------------------

- As an integral component of the Microcontroller Abstraction Layer (MCAL), the General-Purpose Timer (GPT) module in Tiva C offers versatile timing capabilities.
- The GPT module provides functions for initialization, setting timer periods, enabling/disabling interrupts, and configuring various timer features.
- It serves as a fundamental timing tool, allowing precise time tracking, countdowns, and periodic event management in embedded systems.

---

### 3. LED (Light Emitting Diode module):

---------------------------------------------------------------------------

- Part of the Hardware Abstraction Layer (HAL), the LED module provides abstraction to control the LEDs connected to the Tiva C microcontroller.
- The module offers functions to initialize an LED or a group of LEDs, manipulate their states (on/off), and redefine their operating current and active state (active-high or active-low).
- With these functions, controlling the LED indicators within the application layer becomes seamless and intuitive.

---

### 4. BUTTON (Button Module):

---------------------------------------------------------------------------

- Residing in the HAL, the BUTTON module delivers an abstraction layer over the physical buttons connected to the Tiva C microcontroller.
- The functions provided by the BUTTON module include button initialization, reading button status (pressed/released), managing button interrupts, and setting internal/external pull-up/down resistors.
- This layer of abstraction simplifies button interactions, allowing the application layer to handle button-related events effortlessly.

---

5. RGB LED (RGB Light Emitting Diode module):
--------------------------------------------------------------------------------

- The RGB module serves as an extension to the LED module specifically tailored for RGB LEDs.
- It resides in the HAL and interfaces with the microcontroller's GPIOs to control the individual Red, Green, and Blue LEDs of an RGB LED unit.
- The functions included in the RGB module allow the application to easily switch the RGB LED between different colors, thereby enhancing the interaction experience and aesthetic appeal of the system.

---

4. PWM (Pulse Width Modulation) Module using GPT:
--------------------------------------------------------

- The PWM module, residing in the Hardware Abstraction Layer (HAL) and based on the General-Purpose Timer (GPT) in Tiva C, provides a software-based PWM solution for controlling the brightness of LEDs or other devices.
- This module utilizes the GPT's timer functionality to create PWM signals with variable duty cycles, enabling smooth and adjustable brightness levels.

---

Implementing these modules delivers multiple benefits:
--------------------------------------------------------------------------------

- Abstraction relieves the complexity of low-level hardware management and enhances the readability of the source code.
- Standard module interfaces encourage consistent, error-free programming, and increase the maintainability of the application.
- Encapsulation of features within modules helps with debugging as the error scope gets limited, speeding up the debugging process.
- As the hardware specifics are abstracted inside these modules, the same application can be ported to a different microcontroller with minimal changes.

---

# Drivers:

---

## GPIO Driver for Tiva C:

------------------------------

### Introduction:

---------------

The GPIO (General Purpose Input/Output) driver provides a flexible interface for configuring and controlling GPIO pins on Tiva C microcontrollers. It allows users to initialize, configure, and manage GPIO pins efficiently.

### Features:

-----------

- Flexibility: Users can configure GPIO pins individually or in groups, allowing for versatile use in various applications.

- Error Handling: The driver includes error-checking mechanisms to ensure proper usage and handling of errors during GPIO operations.

- Portability: The driver is designed to work with Tiva C microcontrollers, providing a portable solution for GPIO operations.

### Usage:

--------

### 1. Initialization:

------------------

To use the GPIO driver, include the "gpio_interface.h" header file in your project. Before utilizing GPIO functions, initialize the GPIO pins with the 'gpio_init' function.

### Example:

-----------

```
str_gpioPinOrGroupOfPins_t gpioConfig = {

    .enu_gpioPort = GPIO_PORTA,

    .enu_gpioPinOrGroup = GPIO_PIN_0 | GPIO_PIN_5 | GPIO_PIN_7,

    .enu_modeConfig = GPIO_MODE_DIGITAL,

    .enu_direction = GPIO_DIRECTION_OUTPUT,

    .enu_pinDrive = GPIO_OUTPUT_DRIVE_2MA,

    .enu_pullMode = GPIO_FLOATING,

    .enu_interruptMode = GPIO_NO_INTERRUPT

};

gpio_init(&gpioConfig, GPIO_UNLOCK);
```

## 2. Configuration:

-------------------

Configure GPIO pins using the 'str_gpioPinOrGroupOfPins_t' structure, specifying the port, pin or group of pins, mode, direction, drive strength, pull mode, and interrupt mode.

The 'enu_gpioPinOrGroup' field allows you to configure a single pin or a group of pins using the bitwise OR ('|') operator. This is useful for configuring multiple pins simultaneously.

## 3. GPIO Operations:

----------------------

### - Writing to Pins:

-----------------

Use the 'gpio_write' function to set the output level of GPIO pins.

```
gpio_write(&gpioConfig, GPIO_HIGH_LEVEL);
```

### - Reading from Pins:

--------------------

Read the input level of GPIO pins using the 'gpio_read' function.

```
enu_gpioLevelOrValue_t pinValue;
gpio_read(&gpioConfig, &pinValue);
```

### - Toggle Pins:

-------------

Toggle the state of GPIO pins using the 'gpio_toggle' function.

```
gpio_toggle(&gpioConfig);
```

## 4. Callbacks and Interrupts:

-------------------------------

Configure GPIO pins for interrupt handling using the 'gpio_callBackSinglePinInterrupt' function. Provide a callback function to be executed upon interrupt.

```
myCallbackFunction();
gpio_callBackSinglePinInterrupt(GPIO_PORTA, GPIO_PIN_0, &myCallbackFunction);
```

## Error Handling:

-----------------

The GPIO driver includes an error enumeration ('enu_gpioErrorState_t') to handle potential errors during GPIO operations. Always check the return value of GPIO functions to ensure proper execution.

Example:

-----------

```
enu_gpioErrorState_t result = gpio_init(&gpioConfig, GPIO_UNLOCK);

if (result != GPIO_OK)

 {/* Handle error */ }
```

Conclusion:

-------------

The GPIO driver for Tiva C provides a robust and flexible solution for GPIO operations. Utilize the provided functions and structures to configure and control GPIO pins according to your application's requirements.

This updated documentation provides additional clarification on configuring a group of pins using the bitwise OR (‘|’) operator. Adjustments can be made based on specific project requirements and coding standards.

---

# GPT (General-Purpose Timer) Module

----------------------------------------------------------------------

## Introduction:

----------------------

The GPT (General-Purpose Timer) module provides a comprehensive interface for configuring and utilizing the timer functionalities on Tiva C microcontrollers. This module facilitates the initialization, configuration, and control of GPT features, offering both one-shot and periodic modes. The code snippet below illustrates the GPT module's key components and functions.

Features:

---------------

- Versatility:

  - Configure GPT based on specific requirements, allowing seamless integration into various applications.

- Error Handling:

  - The module incorporates error-checking mechanisms to ensure proper usage and effective error management during GPT operations.

- Portability:

  - Designed specifically for Tiva C microcontrollers, the driver provides a portable solution for GPT functionalities.

Usage:

--------------

1. Configuration:

   - Define the GPT base and configuration using the provided enumeration 'enu_gptBase_t' and 'enu_gptConfig_t'.

   enu_gptBase_t enu_gptBase = GPT_BASE_0;

   enu_gptConfig_t enu_gptConfig = GPT_CFG_PERIODIC;

2. Initialization:

   - Initialize the GPT using the 'gpt_config' function before utilizing GPT functionalitie

   gpt_config(enu_gptBase, enu_gptConfig);

3. GPT Operations:

   - Set load values, enable/disable timers, and configure interrupt settings as needed.

```c
gpt_setLoad(enu_gptBase, GPT_TIMER_A, 50000); // Set load value for Timer A
gpt_enable(enu_gptBase, GPT_TIMER_A);        // Enable Timer A
```

4. Interrupt Handling:

   - Register callback functions for handling GPT interrupts.

```c
void myGPTCallbackFunction(void) {  // Your code here }
gpt_interruptRegister(enu_gptBase, GPT_TIMER_A, &myGPTCallbackFunction);
```

5. Error Handling:

   - Check the return value of GPT functions for error management.

```c
enu_gptErrorState_t result = gpt_config(enu_gptBase, enu_gptConfig);
if (result != GPT_OK) { // Handle error }
```

Conclusion:

-------------------

The GPT module for Tiva C microcontrollers provides a robust and adaptable solution for timer functionalities. Utilize the functions and structures provided to configure and control GPT according to your application's requirements.

---

## LED Module:

----------------

## Introduction:

---------------

The LED module provides an interface for controlling and managing LEDs on Tiva C microcontrollers. It offers functions for initialization, deinitialization, changing LED status, toggling LEDs, and more.

## Features:

----------

- Flexibility: Users can configure LED pins individually or in groups, allowing for versatile use in various applications.

- Error Handling: The module includes error-checking mechanisms to ensure proper usage and handling of errors during LED operations.

- Portability: The driver is designed to work with any GPIO with the same interface, providing a portable solution for LEDs operations.

## LED Configuration:

--------------------

1. LED Ports: The module supports LEDs on the following ports:

- LEDS_PORTA, LEDS_PORTB, LEDS_PORTC, LEDS_PORTD, LEDS_PORTE, LEDS_PORTF

2. LED Pins or Groups: The ‘enu_ledPinOrLedsGroup_t’ enum provides options to select individual LED pins or groups of LEDs using bitwise OR (‘|’). For example:

- LED_P0 | LED_P5 | LED_P7

- Additional options include:

    - ‘LEDS_ALL_PINS’: All pins selected.

    - ‘LEDS_P0_TO_P3’: Pins from 0 to 3 selected.

    - ‘LEDS_P4_TO_P7’: Pins from 4 to 7 selected.

3. LED Status: The ‘enu_ledsStatus_t’ enum defines LED status options:

- ‘LEDS_STATUS_MAX_VALUE’: Maximum value for variable LED status.

- ‘LEDS_STATUS_OFF’: LED off state.

- ‘LEDS_STATUS_ON’: LED on state.

4. Active State: The ‘enu_ledsActiveState_t’ enum specifies the active state of LEDs:

- ‘LEDS_ACTIVE_LOW’: Active low state.

- ‘LEDS_ACTIVE_HIGH’: Active high state.

5. Operating Current: The ‘enu_ledsOperatingCurrent_t’ enum determines the operating current for LEDs:

- ‘LEDS_OPERATING_CURRENT_2MA’: 2mA operating current.
- ‘LEDS_OPERATING_CURRENT_4MA’: 4mA operating current.
- ‘LEDS_OPERATING_CURRENT_8MA’: 8mA operating current.

## LED Configuration Structure:

--------------------------------

The ‘str_ledsConfig_t’ structure encapsulates LED configuration parameters, including port, pins or groups, active state, and operating current.

## LED Functions:

-----------------

### 1. Initialization:

------------------

Use the ‘leds_init’ function to initialize LEDs with the provided configuration structure.

Example:

----------

```
str_ledsConfig_t ledConfig =
 {
    .enu_ledsPort = LEDS_PORTA,
    .enu_ledsPinOrGroup = LED_P0 | LED_P5 | LED_P7,
    .enu_ledsActiveState = LEDS_ACTIVE_HIGH,
    .enu_ledsOperatingCurrent = LEDS_OPERATING_CURRENT_2MA
};
leds_init(&ledConfig);
```

### 2. Deinitialization:

---------------------

Use the ‘leds_deinit’ function to deinitialize LEDs.

Example:

----------

leds_deinit(&ledConfig);

### 3. Changing LED Status:

----------------------------

Use the ‘leds_changeStatus’ function to change the status of LEDs.

Example:

----------

leds_changeStatus(&ledConfig, LEDS_STATUS_ON);

### 4. Toggling LEDs:

---------------------

Use the ‘leds_toggle’ function to toggle the state of LEDs.

Example:

----------

leds_toggle(&ledConfig);

### 5. Changing Status of a Single LED:

---------------------------------------

Use the ‘leds_changeSingleLEDStatus’ function to change the status of a single LED.

Example:

----------

leds_changeSingleLEDStatus(LEDS_PORTA, LED_P0, LEDS_ACTIVE_HIGH, LEDS_STATUS_ON);

## Error Handling:

-----------------

The LED module includes an error enumeration (‘enu_ledsErrorState_t’) to handle potential errors during LED operations. Always check the return value of LED functions to ensure proper execution.

Example:

----------

enu_ledsErrorState_t result = leds_init(&ledConfig);

if (result != LEDS_OK)  {// Handle error}

## Conclusion:

--------------

The LED module for Tiva C provides a versatile and error-handling solution for LED operations. Utilize the provided functions and structures to control LEDs based on your application's requirements.

## RGB Module:
-----------------
### Introduction:
---------------

The RGB module provides an interface for controlling RGB LEDs on Tiva C microcontrollers. It supports functions for initialization, deinitialization, and changing the color of the RGB LED.

### Features:
----------

- Error Handling: The module includes error-checking mechanisms to ensure proper usage and handling of errors during RGB operations.

### RGB Color Options:
---------------------

The ‘enu_rgbColorON_t’ enum defines various color options for the RGB LED:

- ‘RGB_TURN_OFF’: Turn off the RGB LED.
- ‘RGB_RED_ON’: Turn on the red component of the RGB LED.
- ‘RGB_GREEN_ON’: Turn on the green component of the RGB LED.
- ‘RGB_BLUE_ON’: Turn on the blue component of the RGB LED.
- ‘RGB_RED_GREEN_ON’: Turn on both red and green components.
- ‘RGB_RED_BLUE_ON’: Turn on both red and blue components.
- ‘RGB_GREEN_BLUE_ON’: Turn on both green and blue components.
- ‘RGB_RED_GREEN_BLUE_ON’: Turn on all components for a full-color display.

### RGB Configuration Structure:
--------------------------------

The RGB module utilizes the ‘str_ledsConfig_t’ structure for configuring the RGB LED. This structure specifies the LED port, pin or group, and operating current.

### RGB Functions:
-----------------

#### 1. Initialization:

------------------

Use the ‘rgb_init’ function to initialize the RGB LED.

Example:

----------

rgb_init();

2. Deinitialization:

-------------------

Use the ‘rgb_deinit’ function to deinitialize the RGB LED.

Example:

----------

rgb_deinit();

3. Changing RGB Color:

-------------------------

Use the ‘rgb_changeColor’ function to change the color of the RGB LED based on the ‘enu_rgbColorON_t’ options.

Example:

----------

rgb_changeColor(RGB_RED_GREEN_BLUE_ON);

Error Handling:

----------------

The RGB module includes an error enumeration (‘enu_rgbErrorState_t’) to handle potential errors during RGB operations. Always check the return value of RGB functions to ensure proper execution.

Example:

----------

enu_rgbErrorState_t result = rgb_init();

if (result != RGB_OK) { // Handle error }

Conclusion:

-------------

The RGB module for Tiva C provides a straightforward solution for RGB LED operations. Utilize the provided functions and structures to control the RGB LED based on your application's requirements.

Button Module:
-------------------

Introduction:
--------------

The Button module provides an interface for handling button-related operations on embedded systems. It includes functions for button initialization, deinitialization, handling button interrupts, and reading button status.

Features:
----------

- Flexibility: Users can configure individual buttons or groups of buttons, allowing for versatile use in different applications.
- Error Handling: The module incorporates error-checking mechanisms to ensure proper usage and handling of errors during button operations.
- Portability: The driver is designed to work with any GPIO with the same interface, providing a portable solution for button operations.

Button Configuration:
------------------------

1. Button Ports: The module supports buttons on the following ports:

   - BUTTONS_PORTA, BUTTONS_PORTB, BUTTONS_PORTC, BUTTONS_PORTD, BUTTONS_PORTE, BUTTONS_PORTF

2. Button Pins or Groups: The ‘enu_buttonPinOrButtonsGroup_t’ enum provides options to select individual button pins or groups of buttons using bitwise OR (‘|’). For example:

   - BUTTON_P0 | BUTTON_P5 | BUTTON_P7

   - Additional options include:

     - ‘BUTTONS_ALL_PINS’: All pins selected.

     - ‘BUTTONS_P0_TO_P3’: Pins from 0 to 3 selected.

     - ‘BUTTONS_P4_TO_P7’: Pins from 4 to 7 selected.

3. Button Status: The ‘enu_buttonsStatus_t’ enum defines button status options:

   - ‘BUTTONS_PIN_STATUS_MAX_VALUE’: Maximum value for variable button status.

   - ‘BUTTONS_PIN_STATUS_LOW’: Button in the low state.

   - ‘BUTTONS_PIN_STATUS_HIGH’: Button in the high state.

4. Pull Mode: The ‘enu_buttonsPullMode_t’ enum specifies the pull mode for buttons:

   - ‘BUTTONS_EXTERNAL_PULL_RES’: External pull resistor.

   - ‘BUTTONS_INTERNAL_PULL_UP’: Internal pull-up resistor.

- ‘BUTTONS_INTERNAL_PULL_DOWN’: Internal pull-down resistor.

5. Interrupt Mode: The ‘enu_buttonsInterruptMode_t’ enum determines the interrupt mode for buttons:

   - ‘BUTTONS_NO_INTERRUPT’: No interrupt.
   - ‘BUTTONS_CHANGE_RISING_EDGE’: Interrupt on rising edge.
   - ‘BUTTONS_CHANGE_FALLING_EDGE’: Interrupt on falling edge.
   - ‘BUTTONS_CHANGE_BOTH_EDGES’: Interrupt on both rising and falling edges.

## Button Configuration Structure:

----------------------------------

The ‘str_buttonsConfig_t’ structure encapsulates button configuration parameters, including port, pins or groups, pull mode, and interrupt mode.

## Button Functions:

--------------------

### 1. Initialization:

------------------

- Use the ‘buttons_init’ function to initialize buttons with the provided configuration structure.

```
str_buttonsConfig_t buttonConfig = {
    .enu_buttonsPort = BUTTONS_PORTA,
    .enu_buttonsPinOrGroup = BUTTON_P0 | BUTTON_P5 | BUTTON_P7,
    .enu_buttonsPullMode = BUTTONS_EXTERNAL_PULL_RES,
    .enu_buttonsInterruptMode = BUTTONS_CHANGE_BOTH_EDGES
};
buttons_init(&buttonConfig);
```

### 2. Deinitialization:

--------------------

- Use the ‘buttons_deinit’ function to deinitialize buttons.

```
buttons_deinit(&buttonConfig);
```

### 3. Button Interrupt Callback:

--------------------------------

- Use the ‘buttons_callBackSingleButtonInterrupt’ function to set a callback function for button interrupts.

```
ptr_Func_buttonsCallBack_t callbackFunction = your_callback_function;
buttons_callBackSingleButtonInterrupt(BUTTONS_PORTA, BUTTON_P0, &callbackFunction);
```

### 4. Reading Button Status:

-----------------------------

- Use the ‘buttons_readStatus’ function to read the status of configured buttons.
  enu_buttonsStatus_t buttonStatus;
  buttons_readStatus(&buttonConfig, &buttonStatus);

## 5. Reading Single Button Status:

-----------------------------------

- Use the ‘buttons_readSingleButtonStatus’ function to read the status of a single configured button.
  enu_buttonsStatus_t singleButtonStatus;
  buttons_readSingleButtonStatus(BUTTONS_PORTA, BUTTON_P0, &singleButtonStatus);

## 6. Reading Single Button Change (Polling):

-----------------------------------------------

- Use the ‘buttons_readSingleButtonChange_Polling’ function to detect a change in the status of a single configured button (polling).
  enu_buttonsStatus_t singleButtonStatus;
  buttons_readSingleButtonChange_Polling(BUTTONS_PORTA, BUTTON_P0, &singleButtonStatus);

## Error Handling:

----------------

The Button module includes an error enumeration (‘enu_buttonsErrorState_t’) to handle potential errors during button operations. Always check the return value of button functions to ensure proper execution.

## Example:

----------

enu_buttonsErrorState_t result = buttons_init(&buttonConfig);
if (result != BUTTONS_OK)  {// Handle error}

## Conclusion:

-------------

The Button module provides a reliable and flexible solution for managing button-related tasks in embedded systems. With its comprehensive functions and error-checking mechanisms, developers can easily integrate and customize button operations based on their application needs.

---

## Software PWM Module:

--------------------------------------------

### Introduction:

----------------------

The Software PWM module provides a flexible and configurable solution for generating PWM signals using General-Purpose Timers (GPT) on Tiva C microcontrollers. This module allows users to control the duty cycle of PWM signals, enabling precise control of connected devices such as LEDs, motors, and more.

### Features:

---------------

- Dynamic PWM Configuration: Supports multiple PWM channels with independent configurations.
- Duty Cycle Control: Adjust the duty cycle of PWM signals to control connected devices.
- Error Handling: The module includes error-checking mechanisms to ensure proper usage and handling of errors during PWM operations.

### PWM Configuration:

--------------------------

1. Channel Enumeration:

---------------------------

The 'enu_pwmIndx_t' enumeration specifies the available PWM channels.

2. Configuration Structure:

------------------------------

The 'str_pwmCfg_t' structure is used for PWM configuration, including the PWM channel and frequency.

```
typedef enum {
    PWM_0A,
    PWM_0B,
    PWM_1A,
    PWM_1B,
    PWM_2A,
    PWM_2B,
```

```
    PWM_3A,
    PWM_3B,
    PWM_4A,
    PWM_4B,
    PWM_5A,
    PWM_5B,
} enu_pwmIndx_t;

typedef struct {
    enu_pwmIndx_t enu_pwmIndx;
    uint32_t uint32_pwmFreq;
} str_pwmCfg_t;
```

## 3. Control Structure:

--------------------------

The 'str_pwmCtrl_t' structure is used for PWM control, including the GPIO port, pin, and callback function.

```
typedef struct {
    enu_gpioPort_t enu_pwmPort;
    enu_gpioPins_t enu_pwmPin;
    ptr_func_gptIRQCallBack_t* ptr_func_pwmCallback;
} str_pwmCtrl_t;
```

4. Initialization:

------------------------

Use the 'pwm_init' function to initialize a PWM channel with the selected configuration, GPIO port, pin, and duty cycle.

```
enu_pwmErrorState_t pwm_init(const str_pwmCfg_t* ptr_str_l_config, uint8_t uint8_l_pwmPort, uint8_t
uint8_l_pwmPin, uint8_t uint8_l_pwmDutyCycle);
```

PWM Operations:

------------------------

5. Start and Stop PWM:

---------------------------

Use the 'pwm_start' and 'pwm_stop' functions to start and stop the PWM signal, respectively.

enu_pwmErrorState_t pwm_start(const str_pwmCfg_t* ptr_str_l_config);

enu_pwmErrorState_t pwm_stop(const str_pwmCfg_t* ptr_str_l_config);

6. Set Duty Cycle:

------------------------

Use the 'pwm_setDutyCycle' function to dynamically change the duty cycle of the PWM signal.

enu_pwmErrorState_t pwm_setDutyCycle(const str_pwmCfg_t* ptr_str_l_config, uint8_t uint8_l_pwmDutyCycle);

Callback Functions:

--------------------------

The module relies on callback functions for handling PWM events. Callback functions for each PWM channel are specified in the 'str_pwmCtrl_t' structure during initialization.

Example:

static void pwm_callback_0A(void) {

// Handle PWM event for channel 0A }

// Other callback functions for different channels.

Conclusion:

-------------------

The Software PWM module for Tiva C provides a versatile and error-handling solution for generating PWM signals. Utilize the provided functions and configurations to implement PWM control based on your application's requirements.

## Error Check Module:

----------------------------

### Introduction:

--------------

The Error_Check module in the LIB (Library) layer provides a robust mechanism for error detection and handling in Tiva C microcontroller projects. It offers a standardized approach for checking and reporting errors within the software.

### Features:

-----------

- Unified Error Enumeration: The module defines an enumeration (‘enu_stdErrorState_t’) with standardized error states (‘STATE_OK’ and ‘STATE_NOK’).

- Error Detection Function: The module includes the ‘ERROR_DETECTED’ function, which, when called, indicates an error occurrence. It captures the file name and line number where the error is detected.

- Macro-Based Error Check: The ‘ERROR_CHECK’ macro simplifies error checking in the code. It takes a boolean expression as an argument and returns ‘STATE_OK’ if true and ‘STATE_NOK’ if false. This macro is especially useful for conditional error handling.

### Error Detection Structure:

-----------------------------

The ‘str_errorDetected_t’ structure encapsulates information about the detected error, including the file name and line number.

### Error Detection Function:

-----------------------------

The ‘ERROR_DETECTED’ function is called to indicate an error occurrence. It captures the file name and line number, triggering a breakpoint (‘bkpt 1’), and returns ‘STATE_NOK’.

### Macro-Based Error Check:

-----------------------------

The ‘ERROR_CHECK’ macro simplifies error checking in the code. It takes a boolean expression as an argument, evaluates it, and returns ‘STATE_OK’ if true and ‘STATE_NOK’ if false.

### Usage:

--------

Example of using the ‘ERROR_CHECK’ macro:

```
enu_stdErrorState_t result = ERROR_CHECK(some_condition);
if (result != STATE_OK) { // Handle error }
```

Example of using the 'ERROR_DETECTED' function:
// Trigger an error detection
ERROR_DETECTED(__FILE__, __LINE__);

Conclusion:

-------------

The Error_Check module offers a standardized and efficient way to handle errors in Tiva C microcontroller projects. Utilize the provided functions and macros to enhance the robustness of your code by detecting and responding to errors appropriately.

# LOW-LEVEL DESIGN

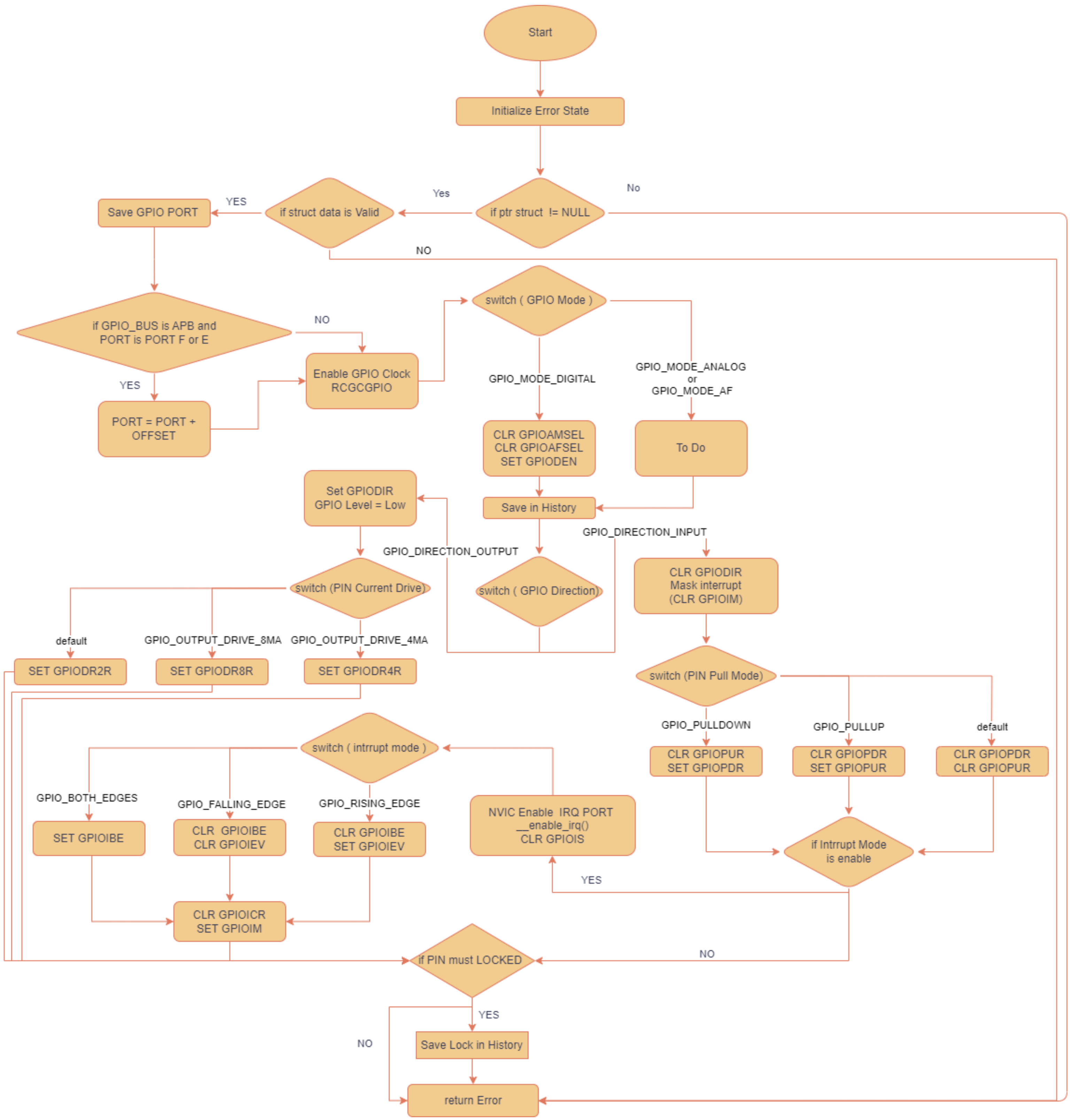
The Low-Level Design phase is a pivotal stage in the development of any project. This phase entails the specification of software components, modules, and their interactions at a granular level. It is dedicated to converting high-level requirements and functionalities into well-defined algorithms, data structures, and code implementations tailored for the Tiva C platform. The primary goal of the Low-Level Design is to ensure the software's efficiency, maintainability, and seamless integration with the underlying Tiva C hardware.

## Functions Flowchart:

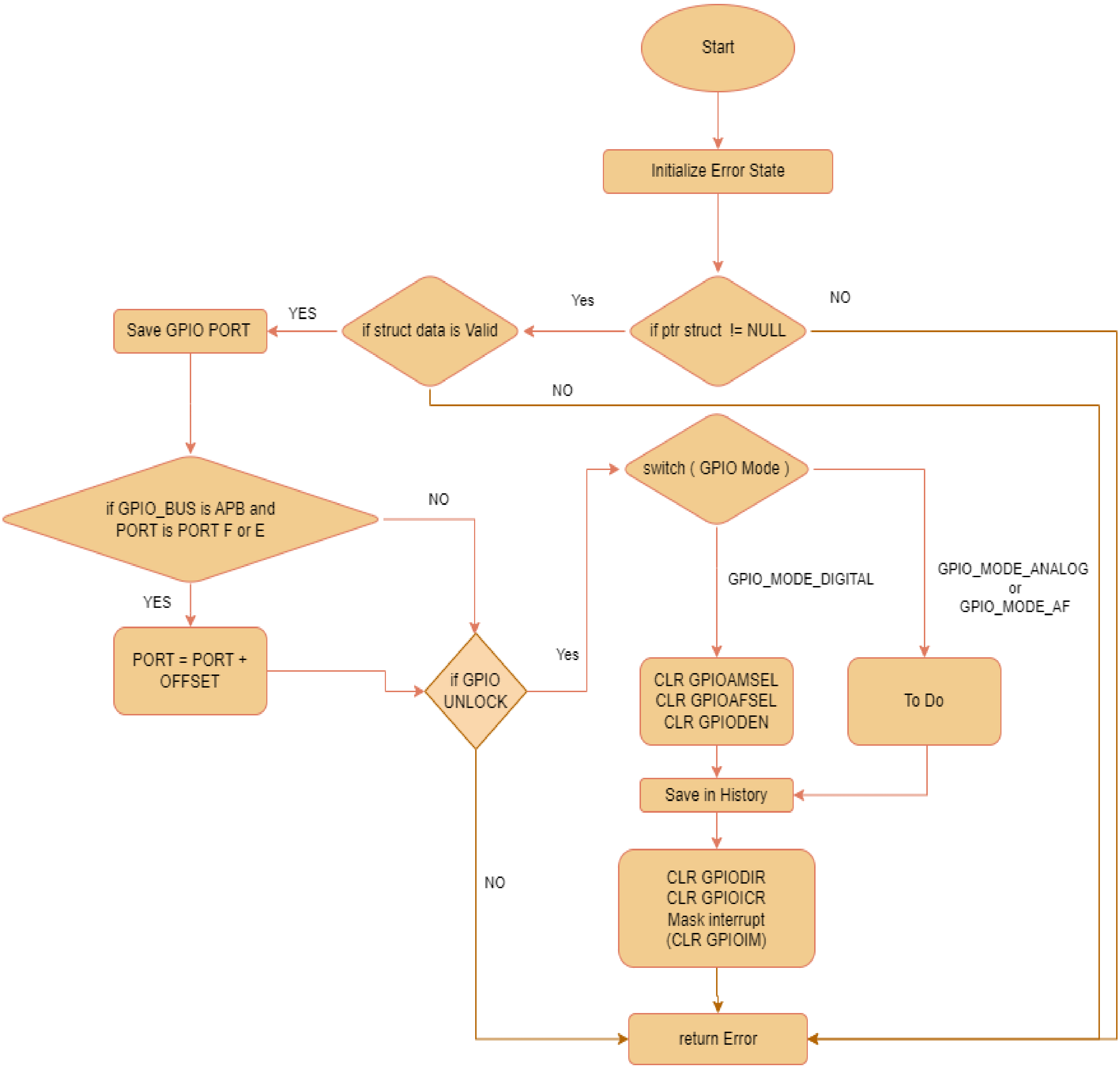
The Functions Flowchart is a visual representation that provides a structured overview of the project's software functions and their interconnections. This design artifact serves as a roadmap for understanding how different software functions and modules interact to achieve the desired behavior of the car robot.

## GPIO Functions

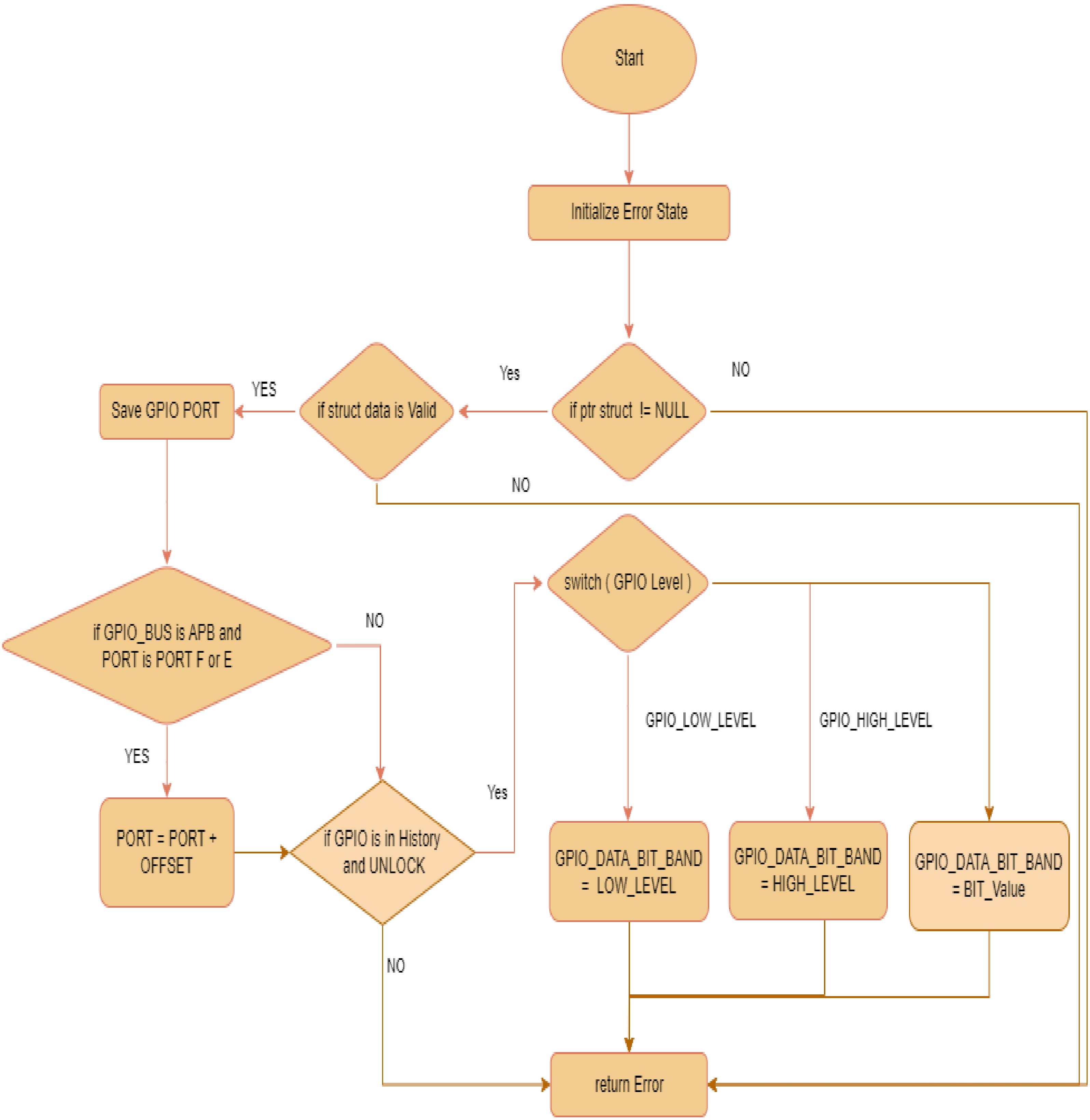
```
enu_gpioErrorState_t gpio_init(const str_gpioPinOrGroupOfPins_t* str_ptr_pinOrGroup ,
enu_gpioPinsLock_t enu_gpioPinsLock ) )
```



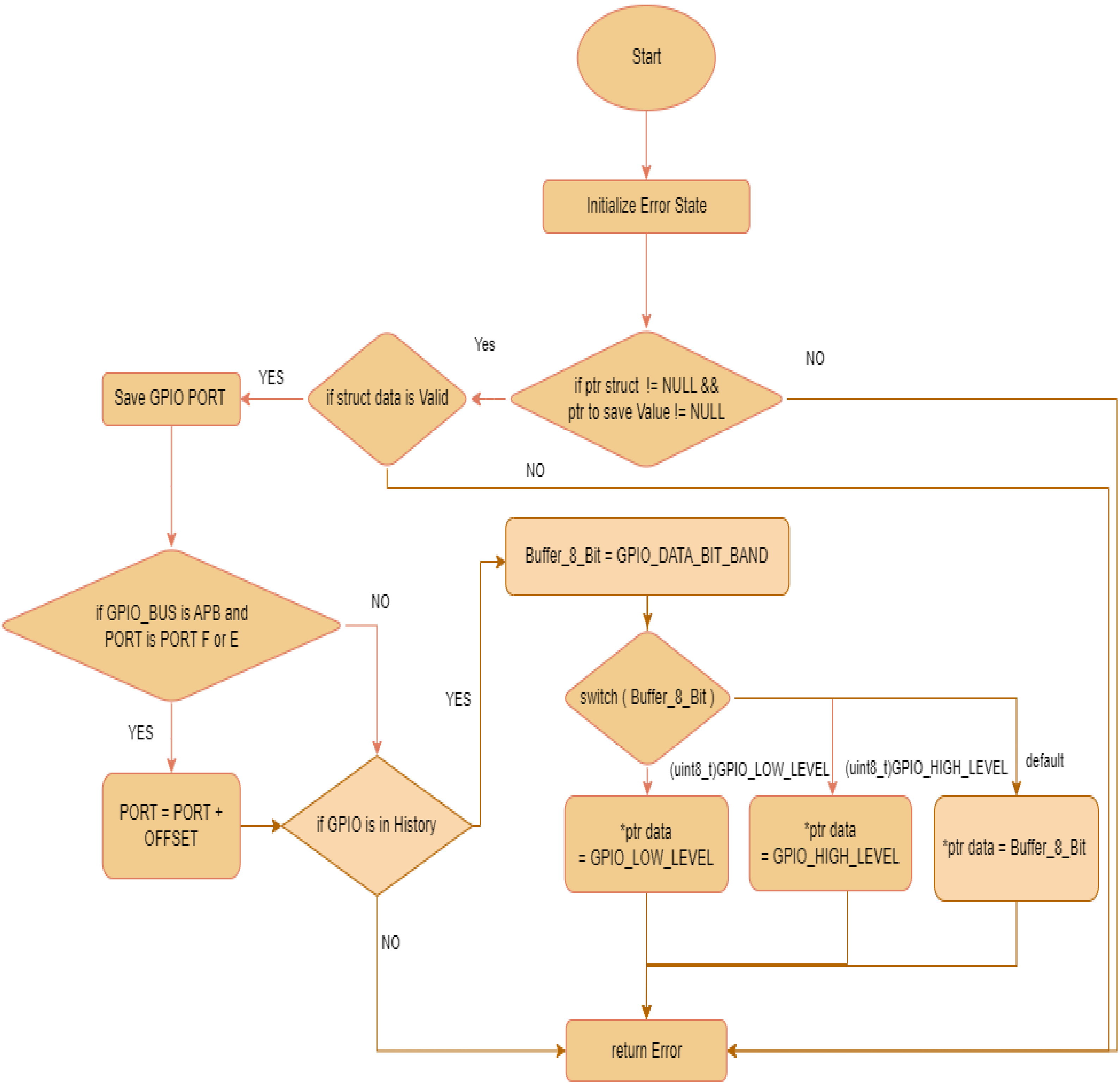
```
enu_gpioErrorState_t gpio_deinit(const str_gpioPinOrGroupOfPins_t* str_ptr_pinOrGroup )
```



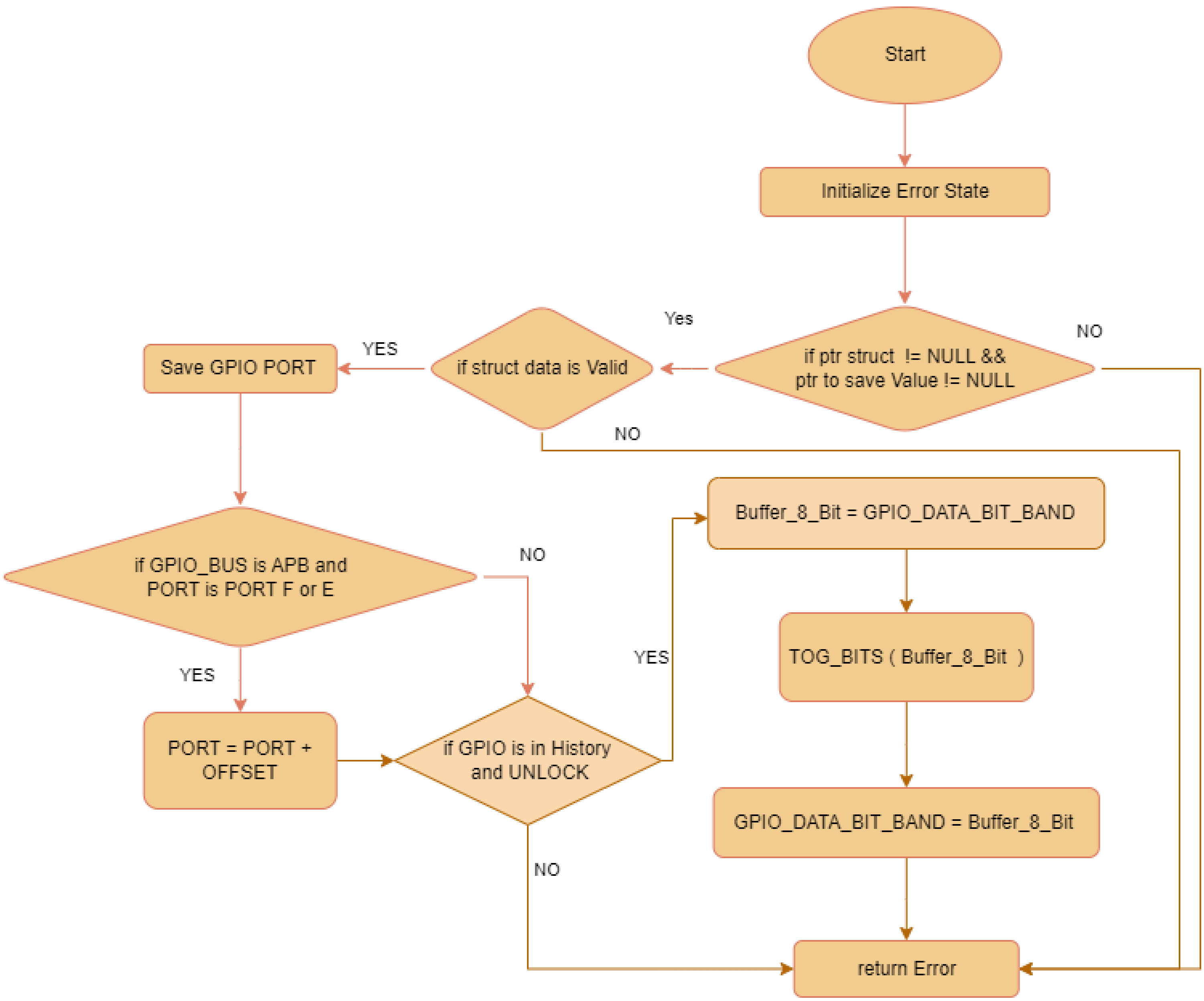
```
enu_gpioErrorState_t gpio_write(const str_gpioPinOrGroupOfPins_t* str_ptr_pinOrGroup ,
enu_gpioLevelOrValue_t  enu_gpioLevelOrValue)
```



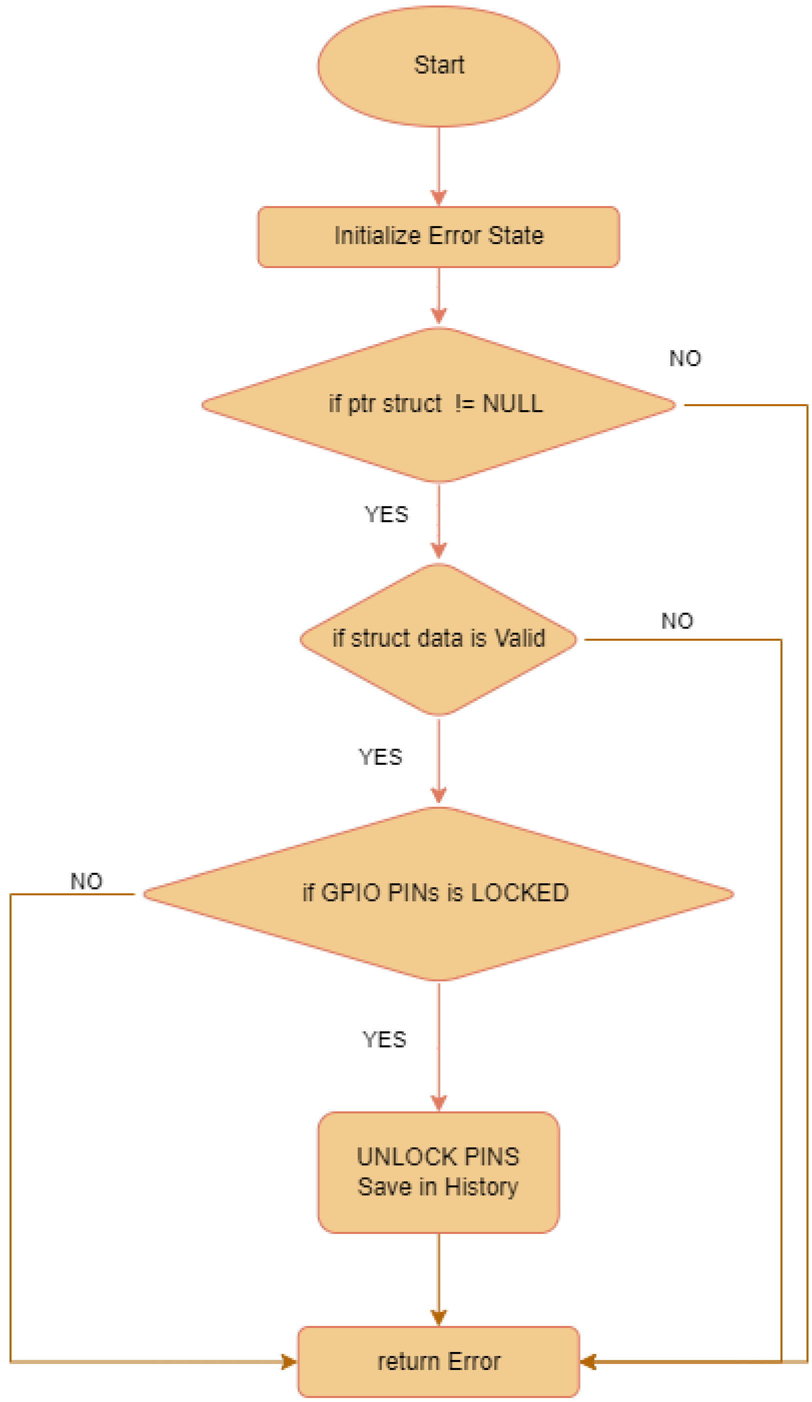
```
enu_gpioErrorState_t gpio_read(const str_gpioPinOrGroupOfPins_t* str_ptr_pinOrGroup ,
enu_gpioLevelOrValue_t  * enu_gpioLevelOrValue)
```



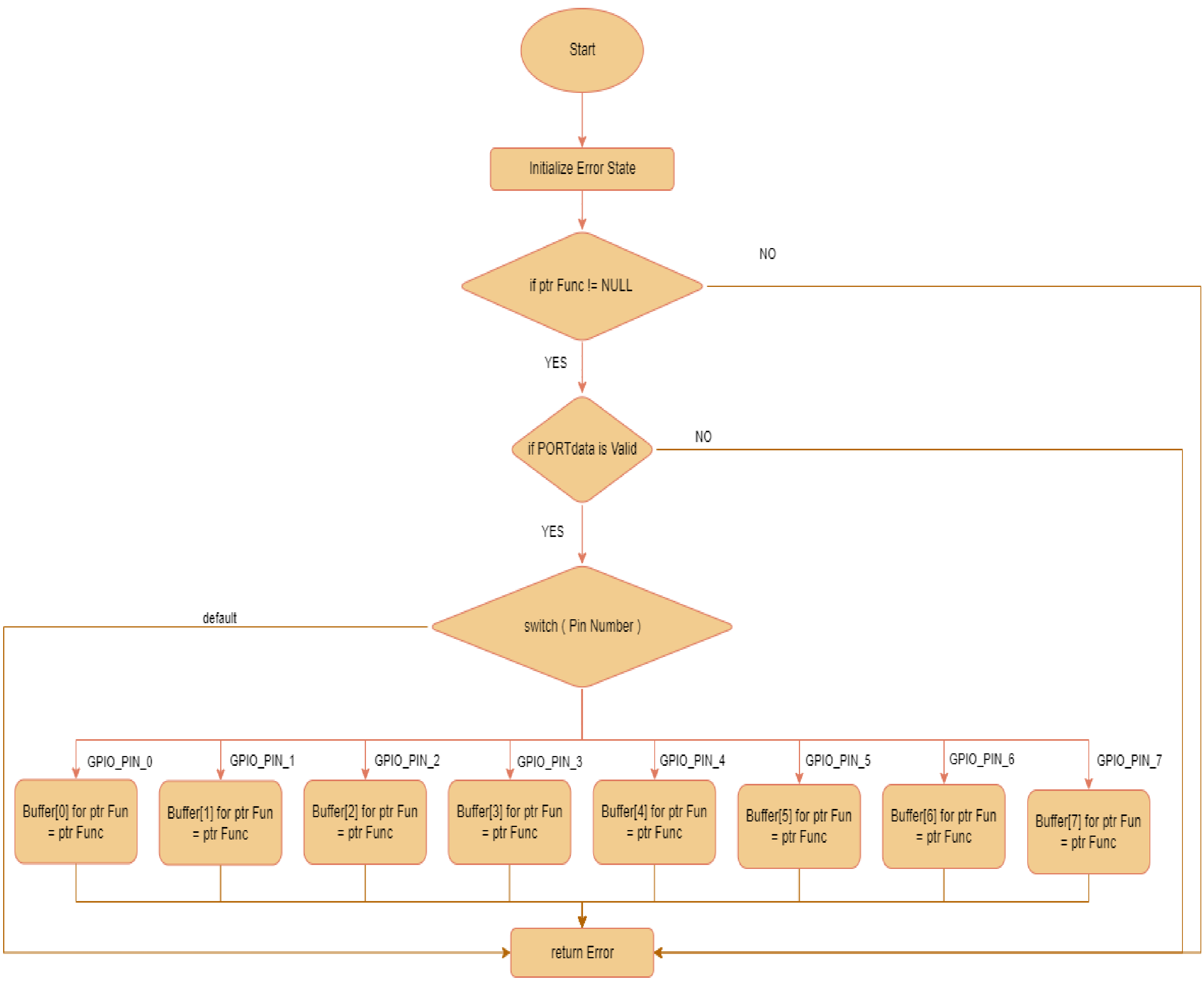
```
enu_gpioErrorState_t gpio_toggle(const str_gpioPinOrGroupOfPins_t* str_ptr_pinOrGroup)
```



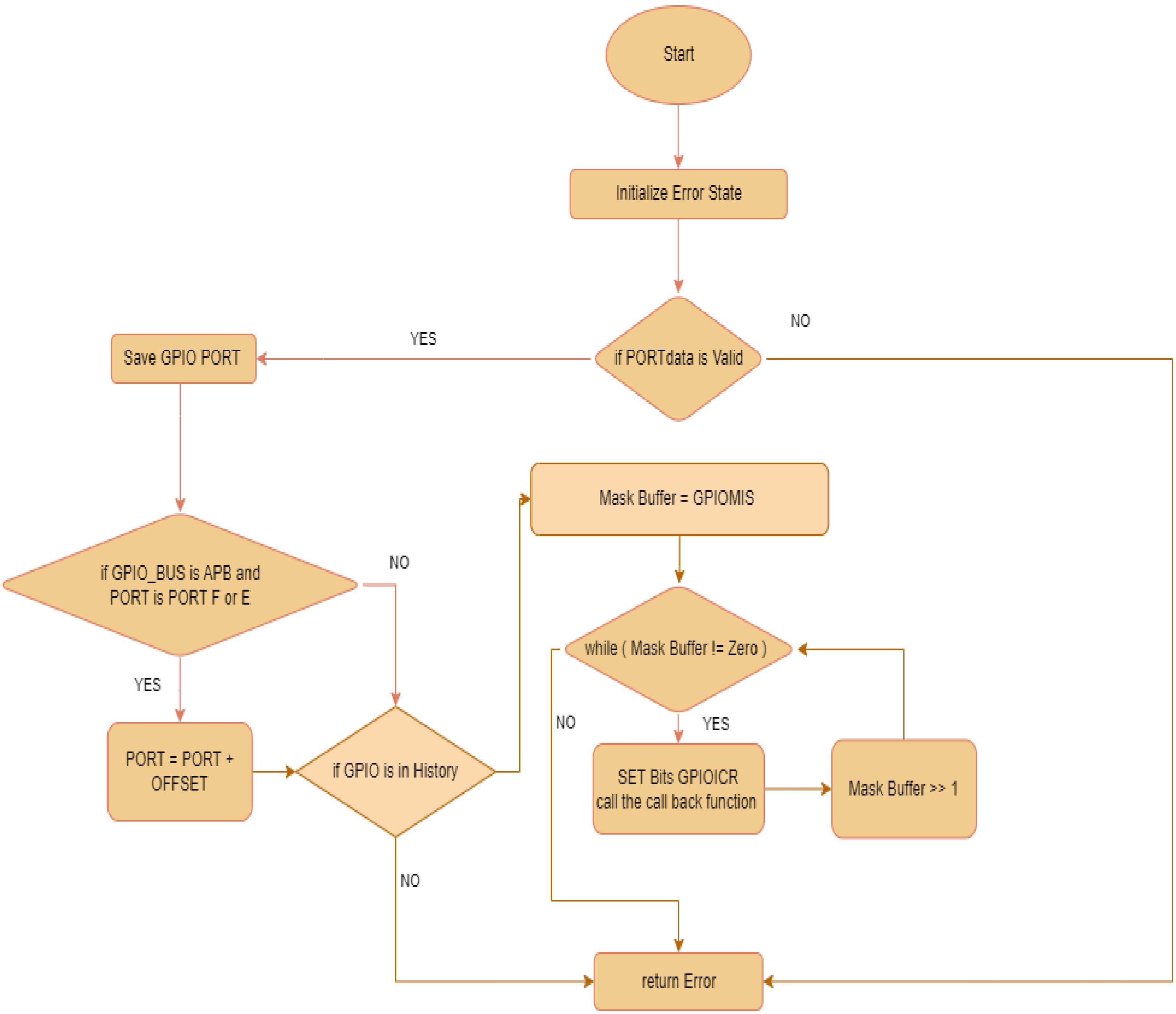
```
enu_gpioErrorState_t gpio_lockAndUnlockPins(const str_gpioPinOrGroupOfPins_t* str_ptr_pinOrGroup
, enu_gpioPinsLock_t enu_gpioPinsLock )
```



```
enu_gpioErrorState_t gpio_callBackSinglePinInterrupt(enu_gpioPort_t enu_gpioPort,enu_gpioPins_t enu_gpioPin,ptr_Func_gpioCallBack_t* ptr_func_interruptCallBack)
```

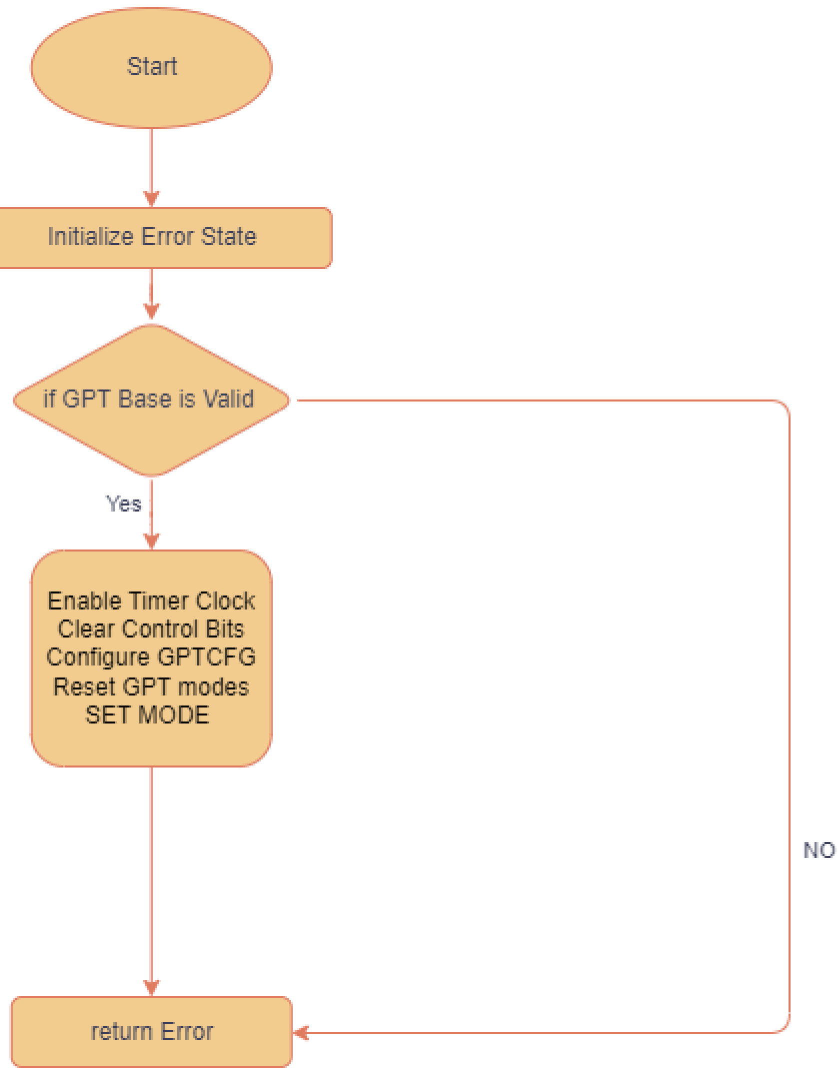


```c
static void gpio_irqHandler(enu_gpioPort_t enu_l_port)
```

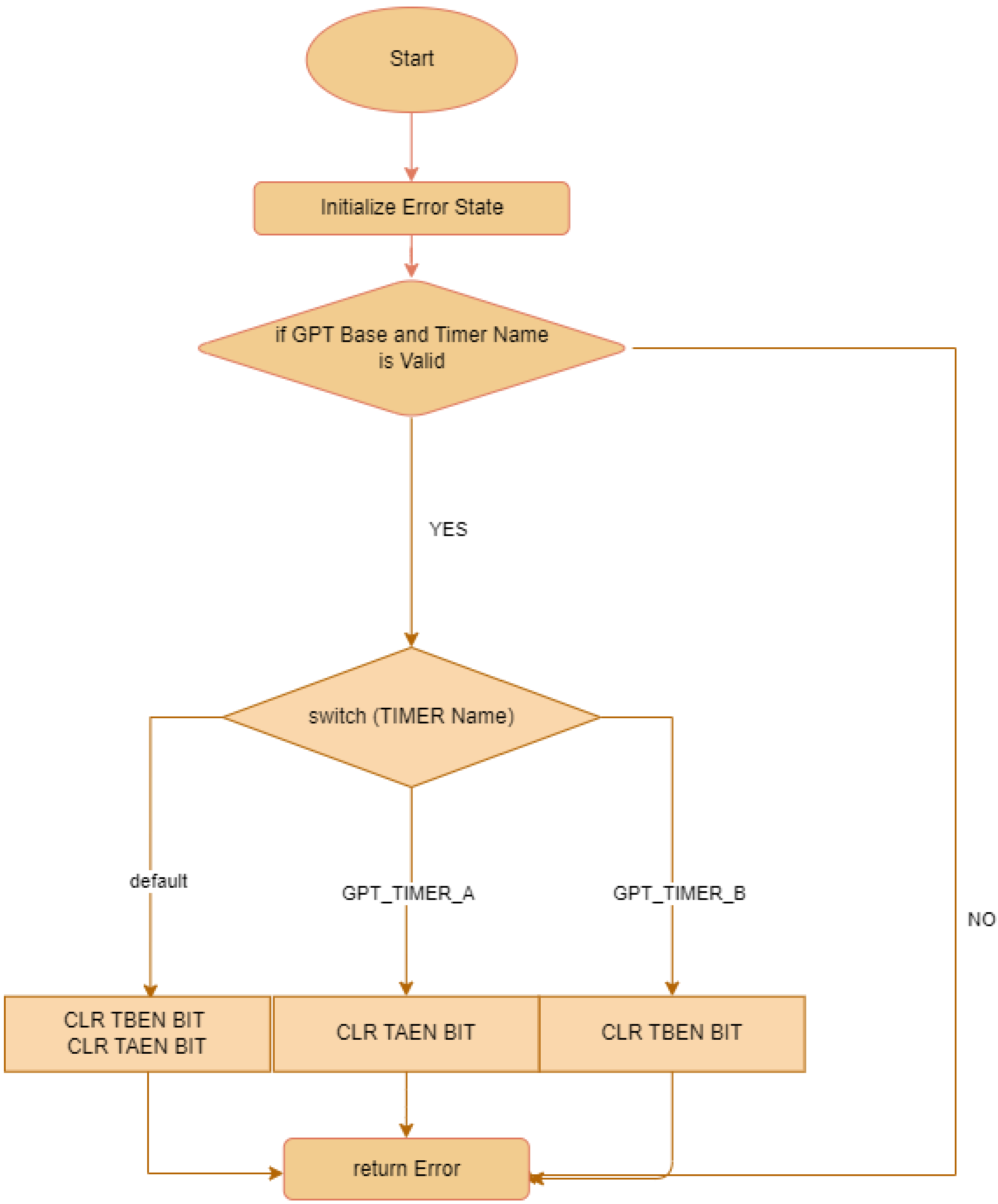


## GPT Functions

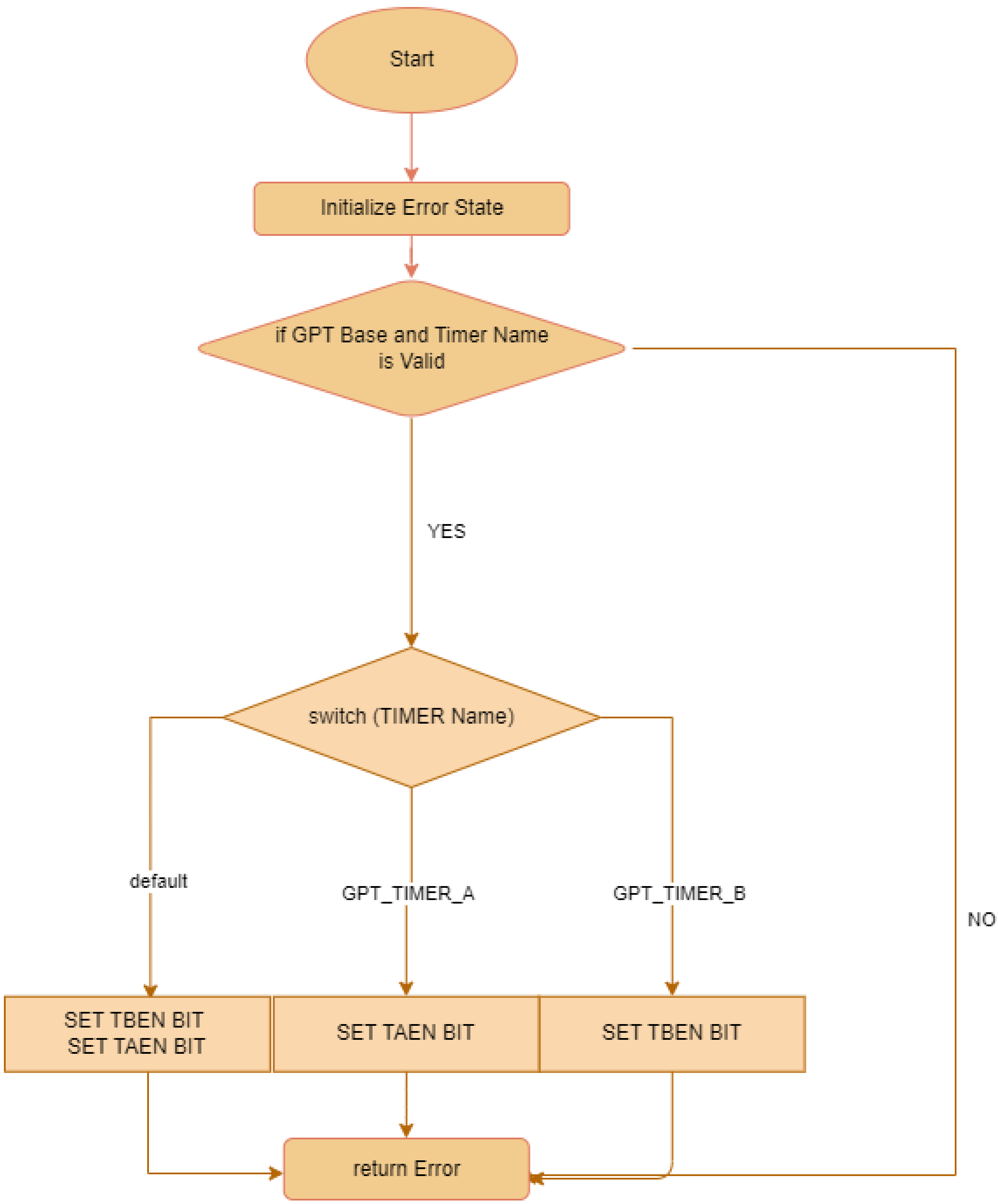
```
enu_gptErrorState_t gpt_config(enu_gptBase_t enu_l_gptBase, enu_gptConfig_t enu_l_gptConfig);
```



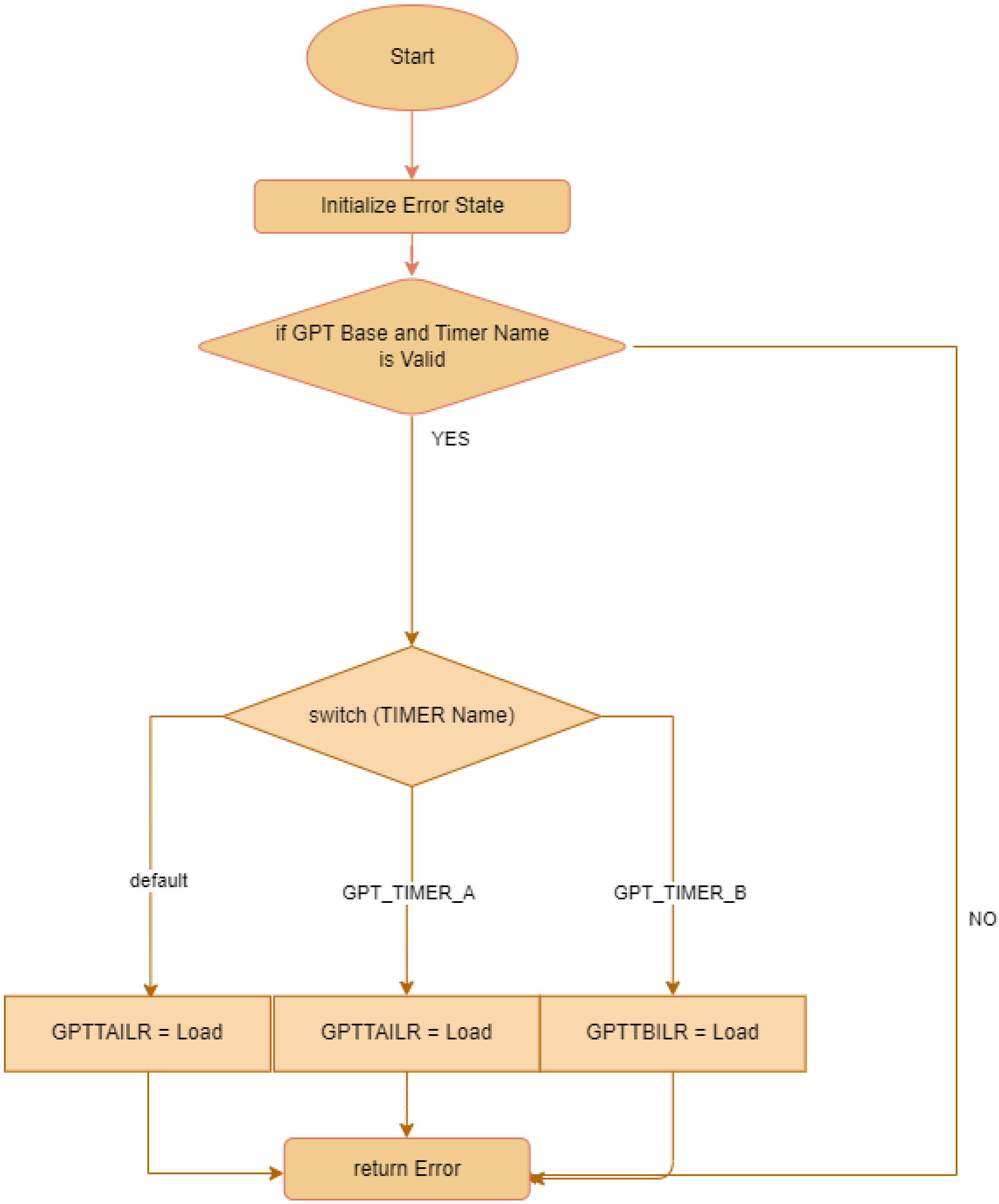
```c
enu_gptErrorState_t gpt_disable(enu_gptBase_t enu_l_gptBase, enu_gptTimerName_t
enu_l_gptTimerName);
```



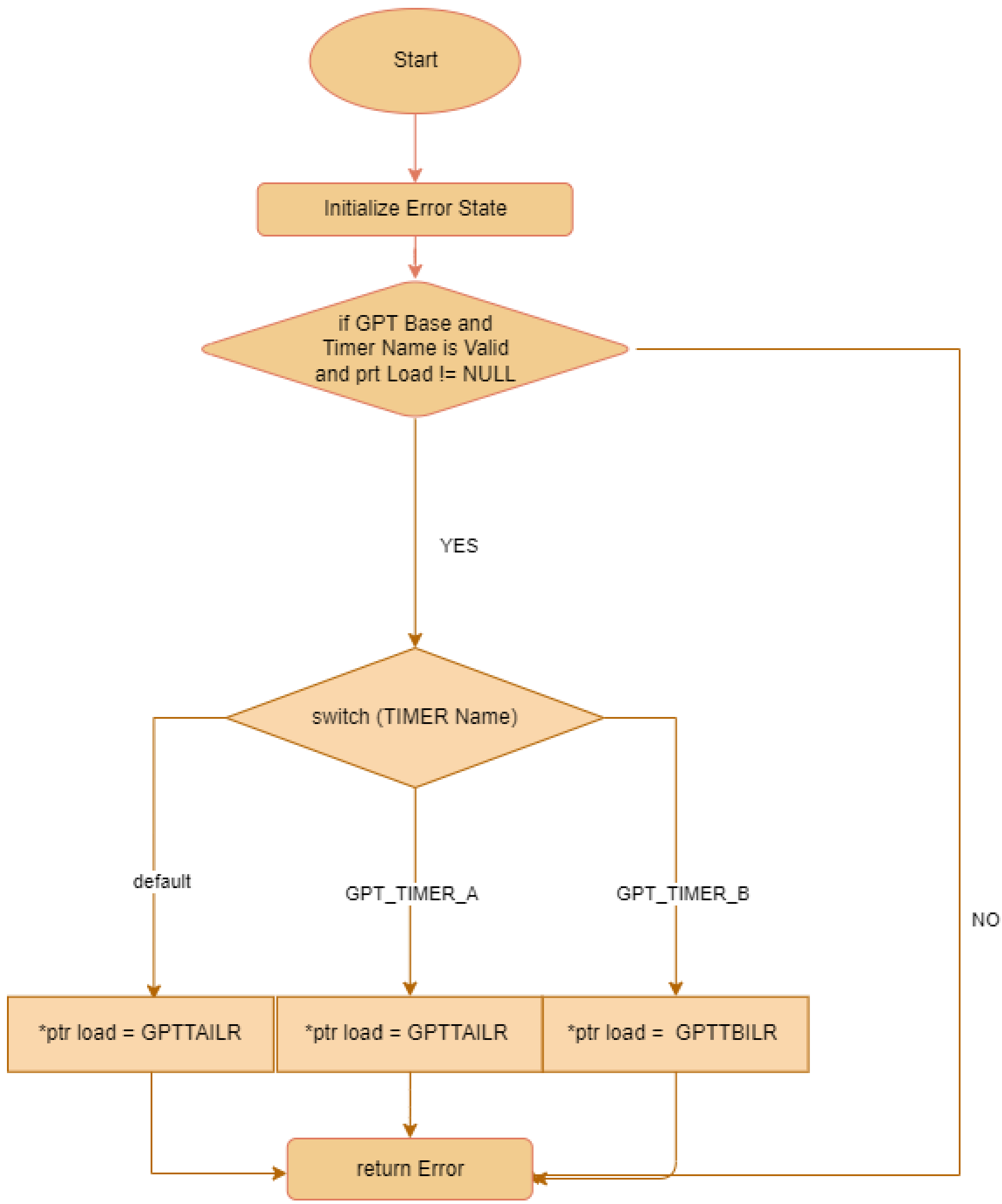
```c
enu_gptErrorState_t gpt_enable(enu_gptBase_t enu_l_gptBase, enu_gptTimerName_t
enu_l_gptTimerName);
```



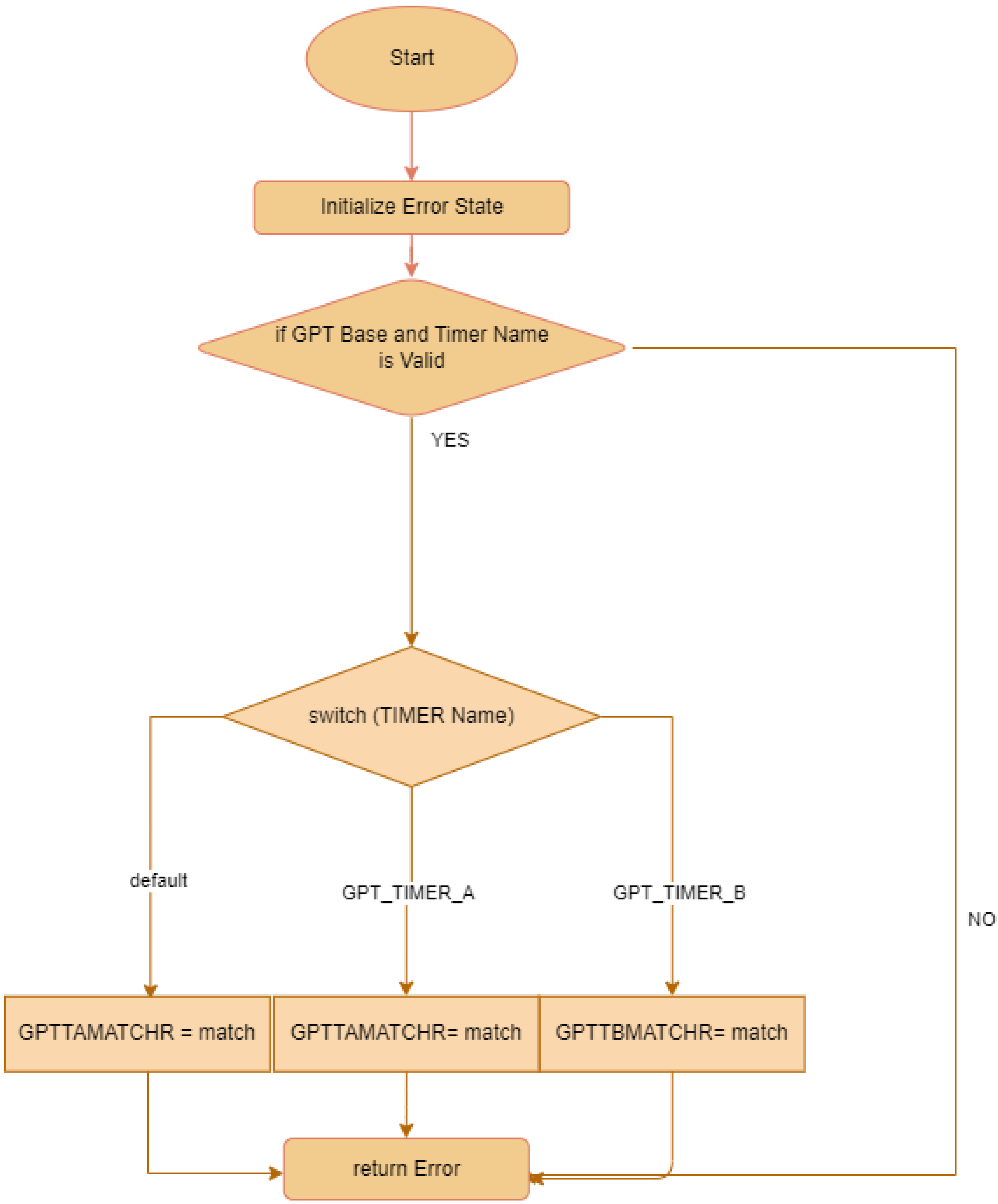
```
enu_gptErrorState_t gpt_setLoad(enu_gptBase_t enu_l_gptBase, enu_gptTimerName_t
enu_l_gptTimerName, uint32_t uint32_l_gptLoadVal);
```



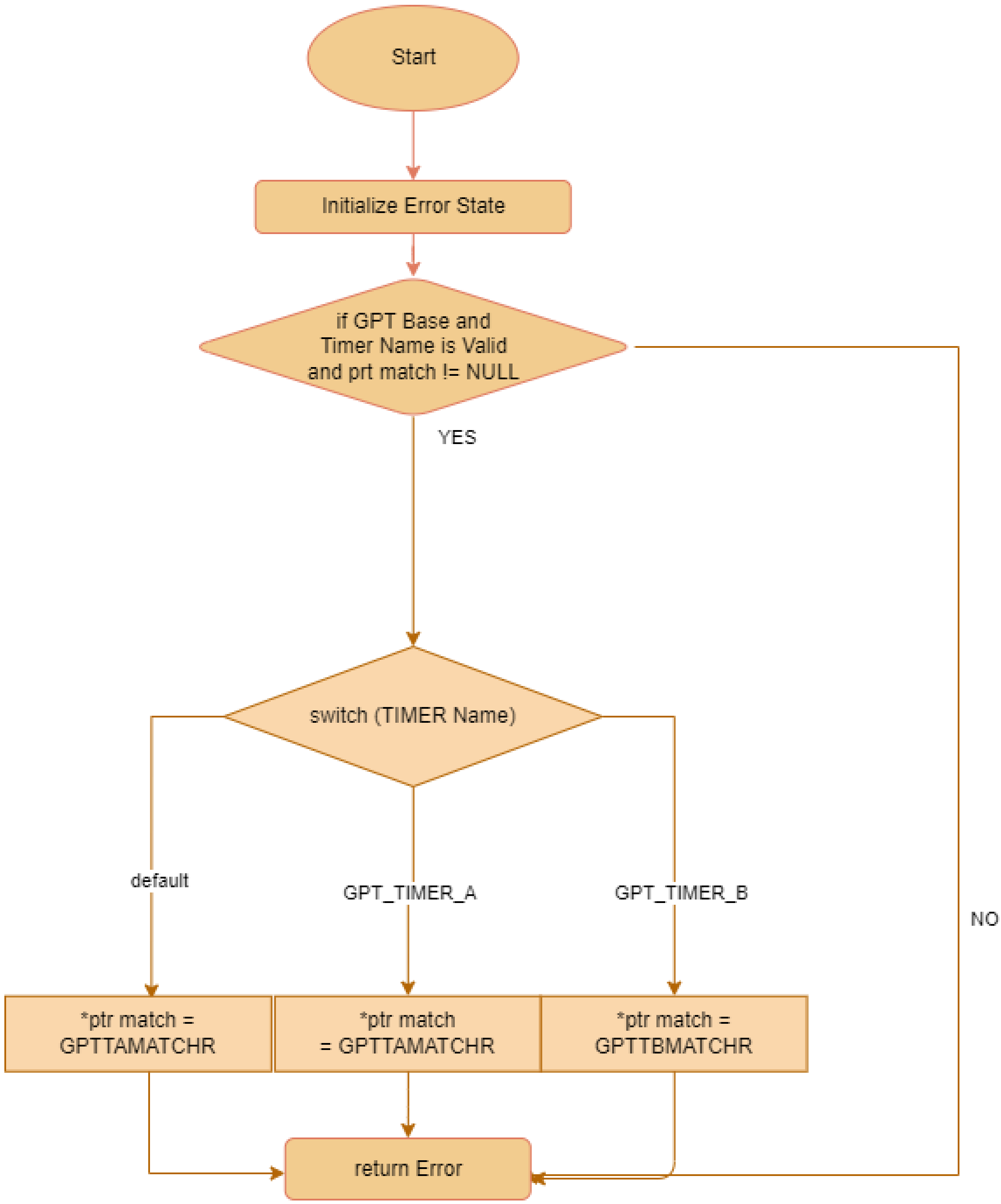
```
enu_gptErrorState_t gpt_getLoad(enu_gptBase_t enu_l_gptBase, enu_gptTimerName_t enu_l_gptTimerName, uint32_t* ptr_uint32_l_gptLoadVal);
```



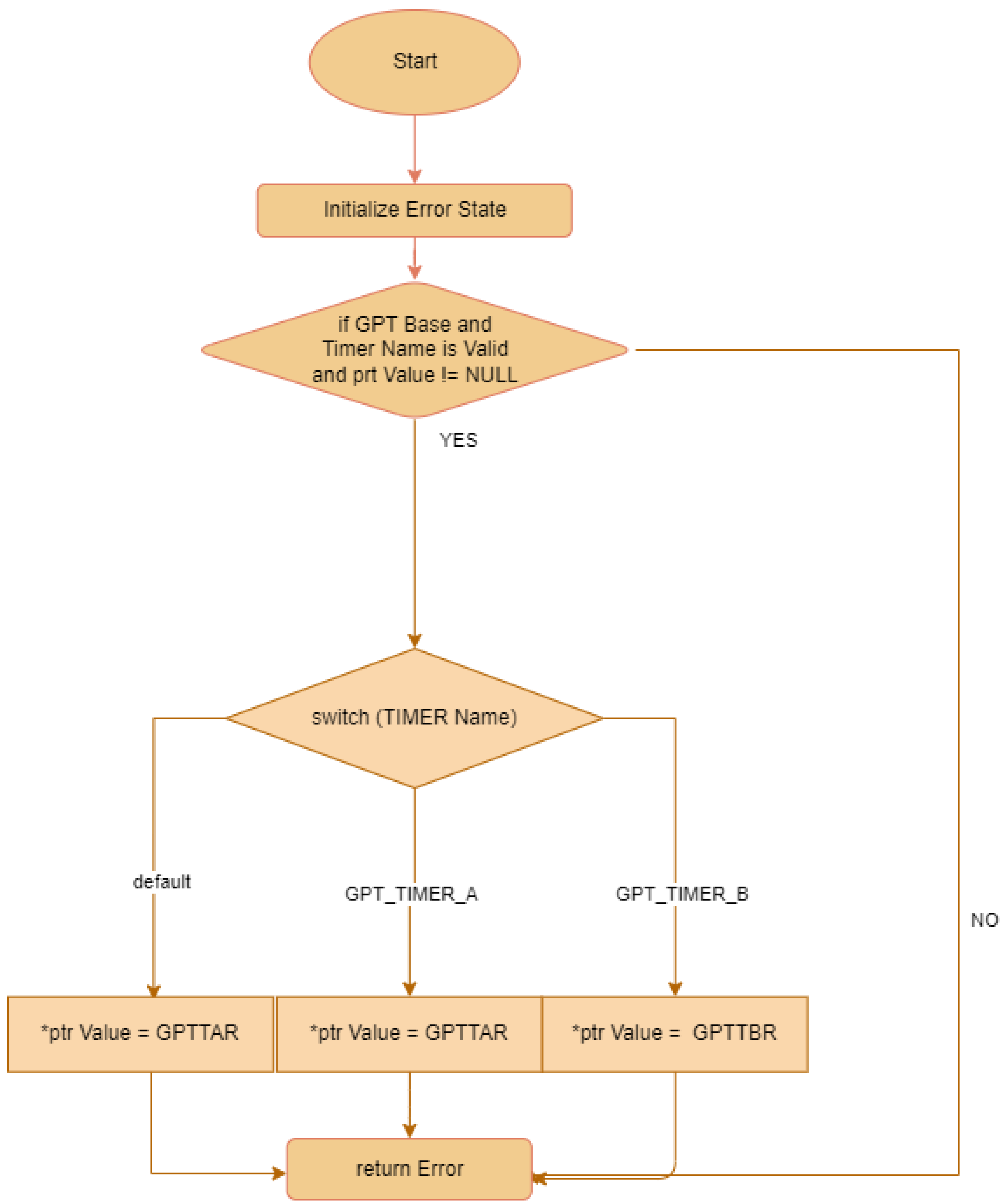
```
enu_gptErrorState_t gpt_setMatch(enu_gptBase_t enu_l_gptBase, enu_gptTimerName_t
enu_l_gptTimerName, uint32_t uint32_l_gptMatchVal);
```



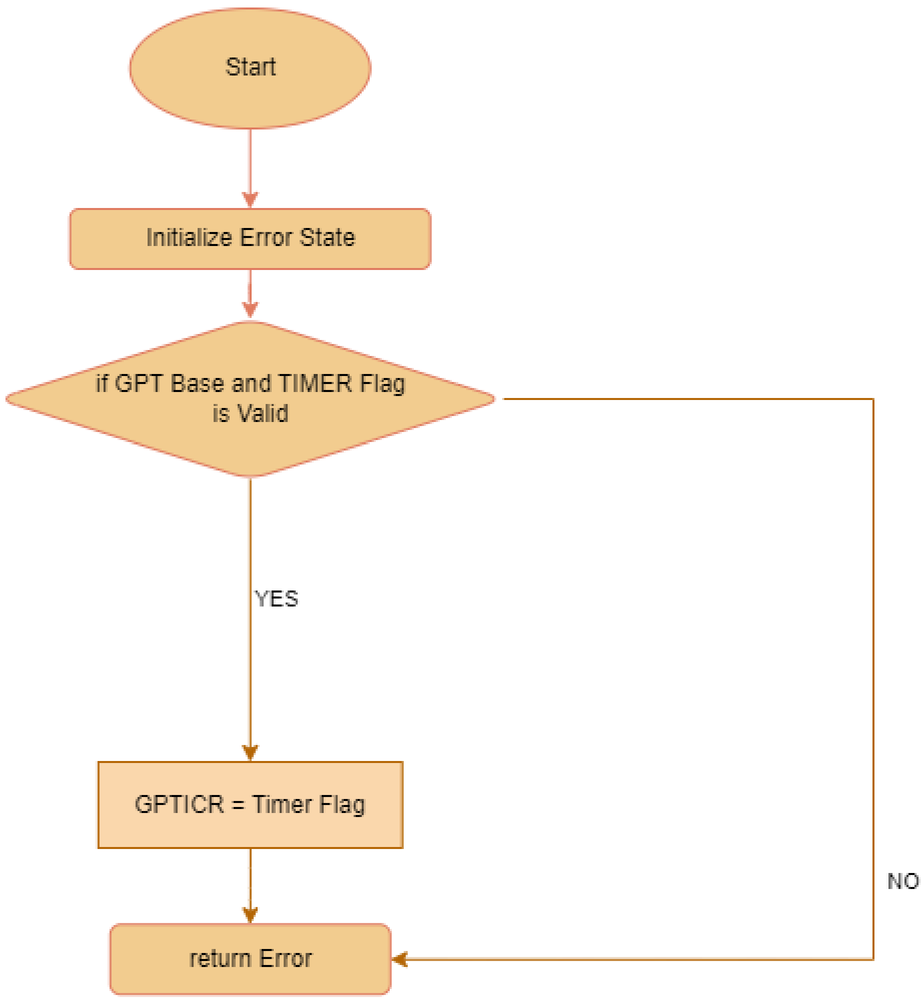
```c
enu_gptErrorState_t gpt_getMatch(enu_gptBase_t enu_l_gptBase, enu_gptTimerName_t
enu_l_gptTimerName, uint32_t* ptr_uint32_l_gptMatchVal);
```



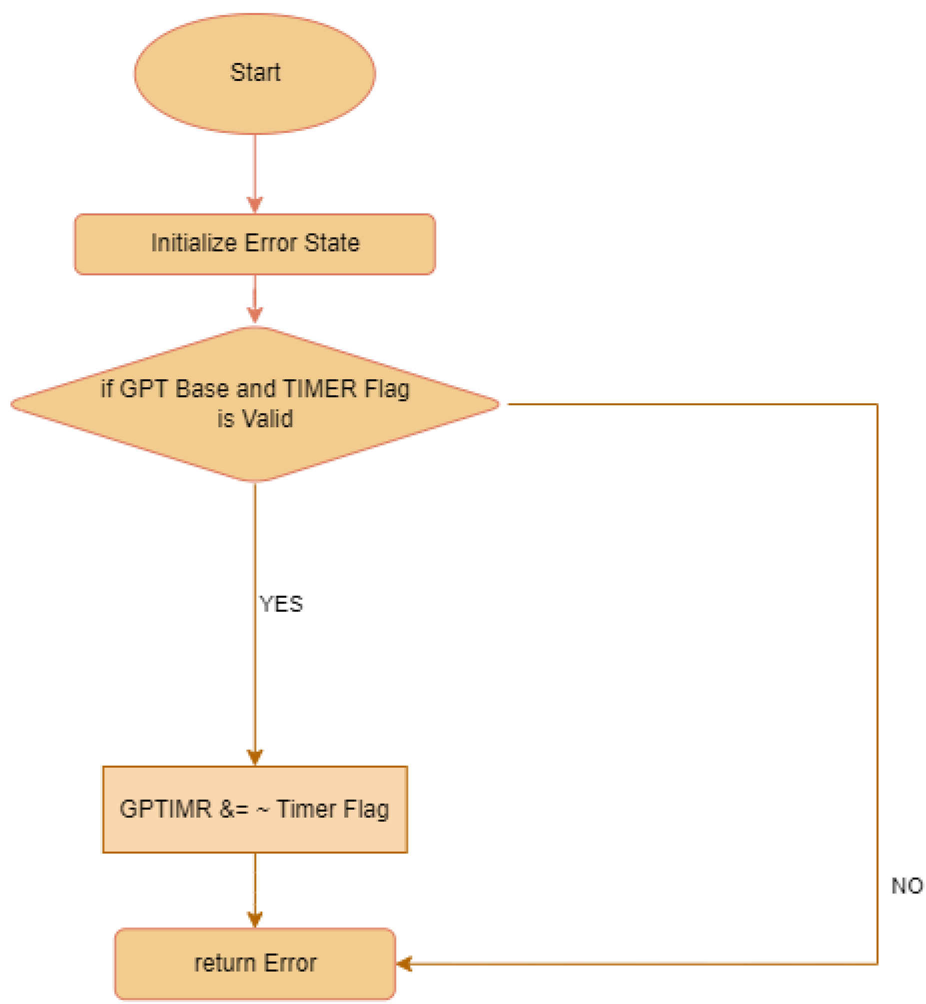
```
enu_gptErrorState_t gpt_getValue(enu_gptBase_t enu_l_gptBase, enu_gptTimerName_t enu_l_gptTimerName, uint32_t* ptr_uint32_l_gptVal);
```



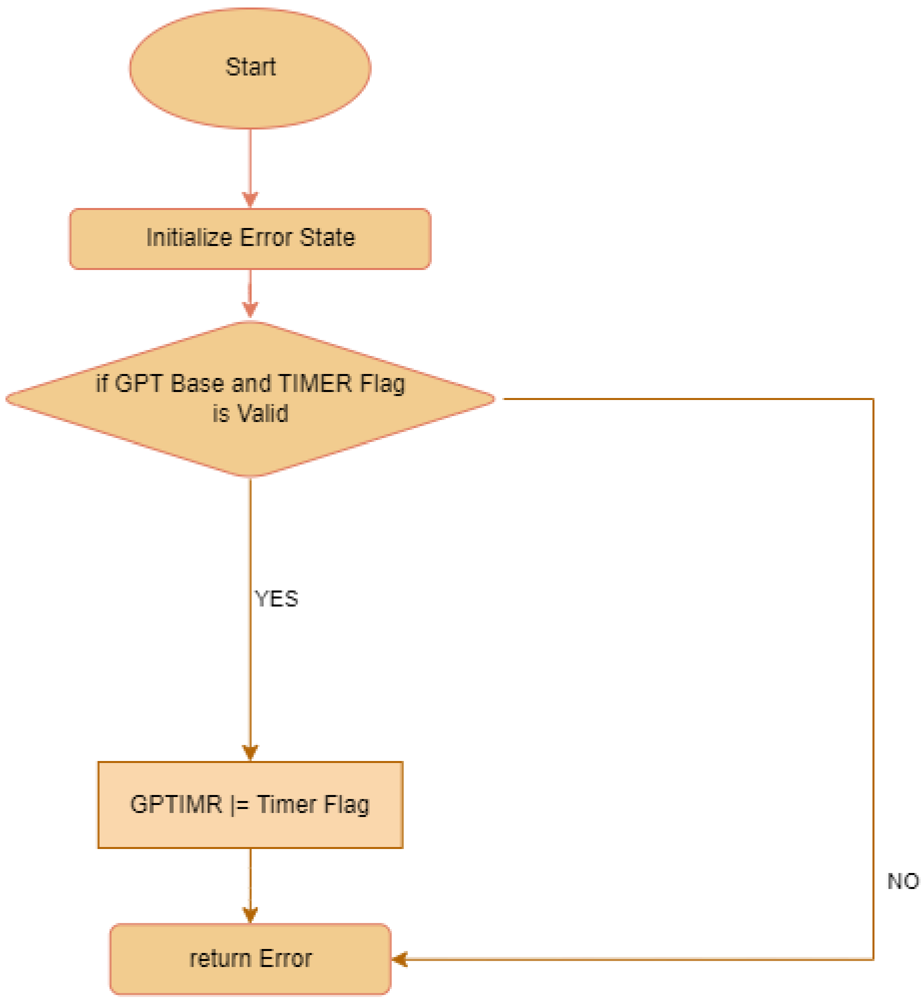
```
enu_gptErrorState_t gpt_interruptClear(enu_gptBase_t enu_l_gptBase, enu_gptInterruptFlags_t enu_l_intFlags );
```



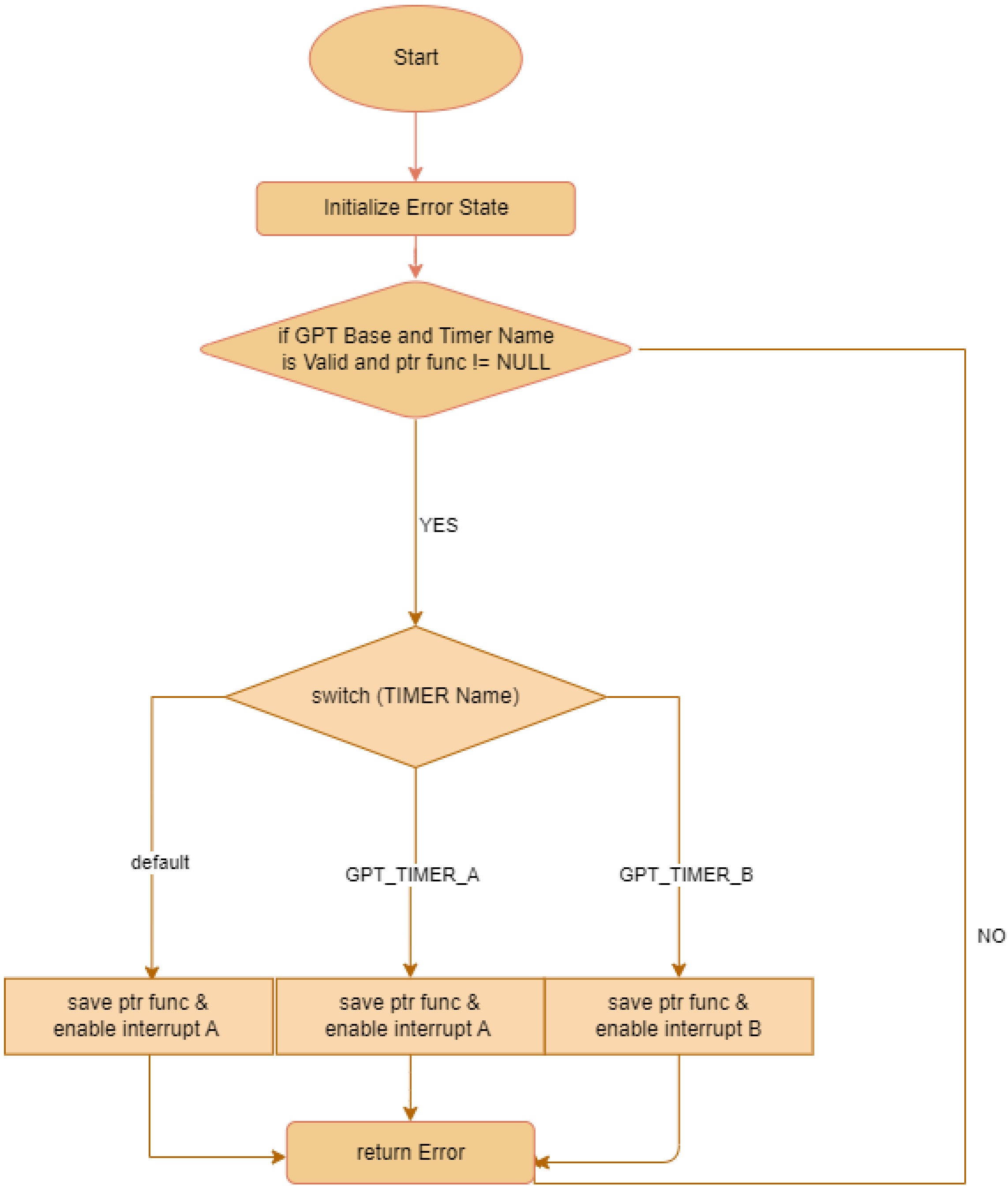
```
enu_gptErrorState_t gpt_interruptDisable(enu_gptBase_t enu_l_gptBase, enu_gptInterruptFlags_t
enu_l_intFlags );
```



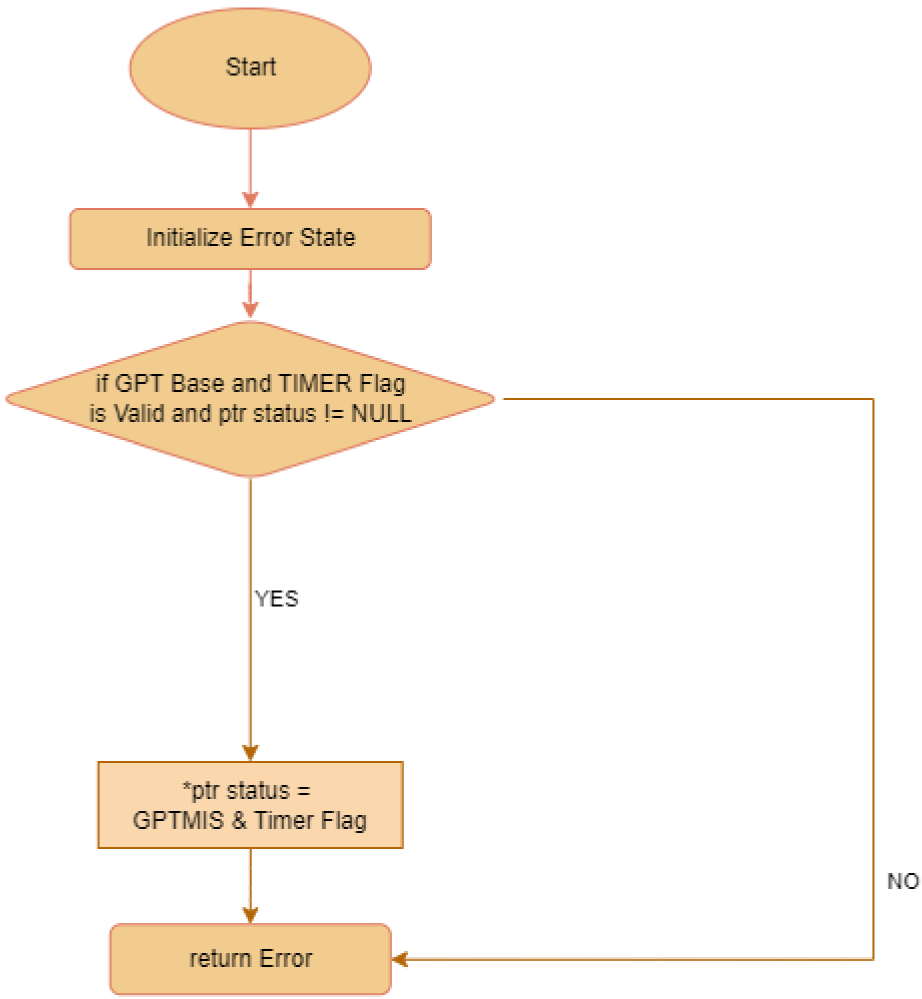
```c
enu_gptErrorState_t gpt_interruptEnable(enu_gptBase_t enu_l_gptBase, enu_gptInterruptFlags_t
enu_l_intFlags );
```



```
enu_gptErrorState_t gpt_interruptRegister(enu_gptBase_t enu_l_gptBase, enu_gptTimerName_t enu_l_gptTimerName, ptr_func_gptIRQCallBack_t* ptr_func_l_handler);
```

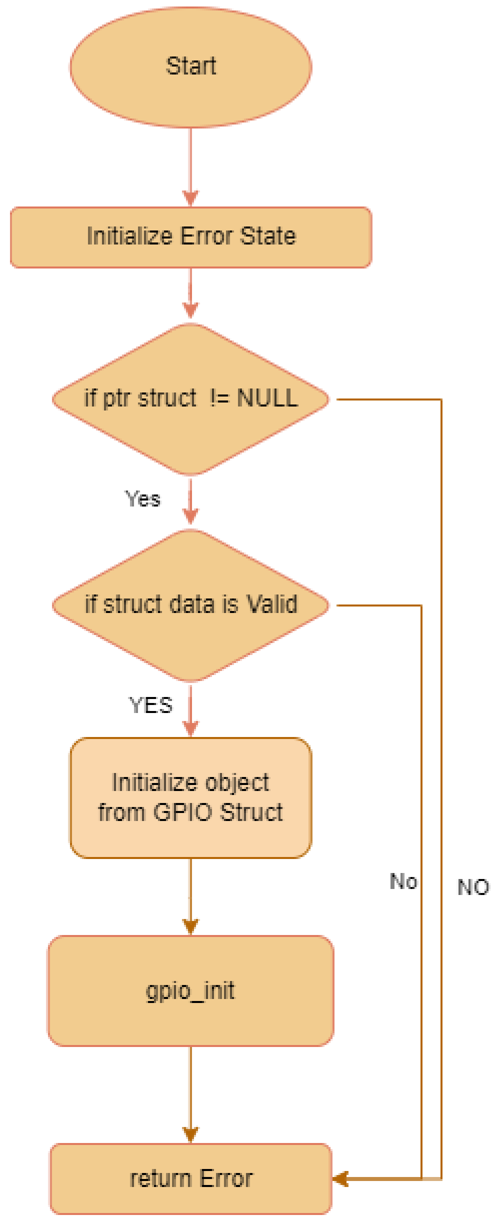


```
enu_gptErrorState_t gpt_interruptStatus(enu_gptBase_t enu_l_gptBase, enu_gptInterruptFlags_t
enu_l_intFlags, uint32_t* ptr_uint32_l_intStatus);
```

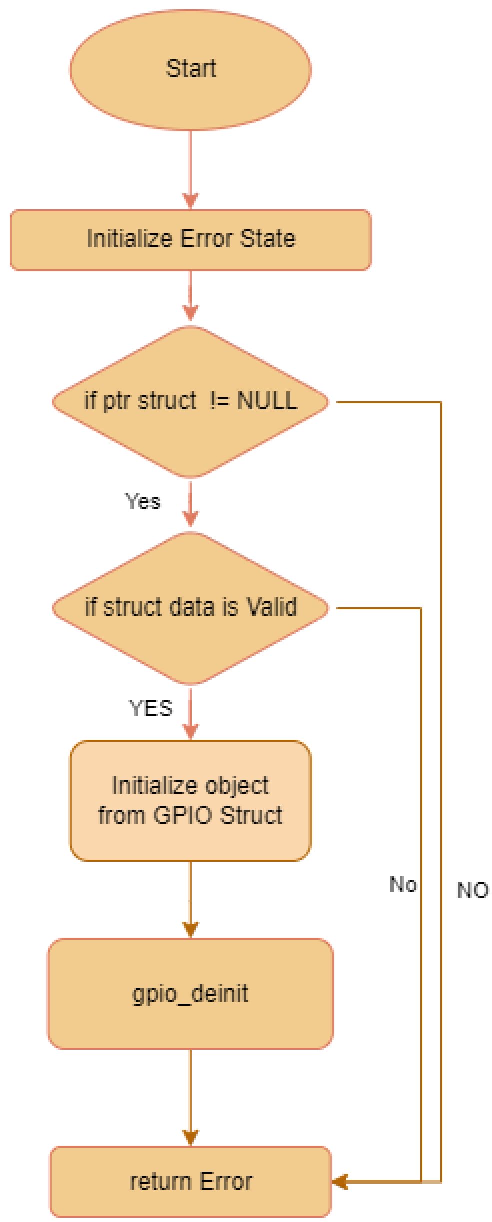


## LED Functions

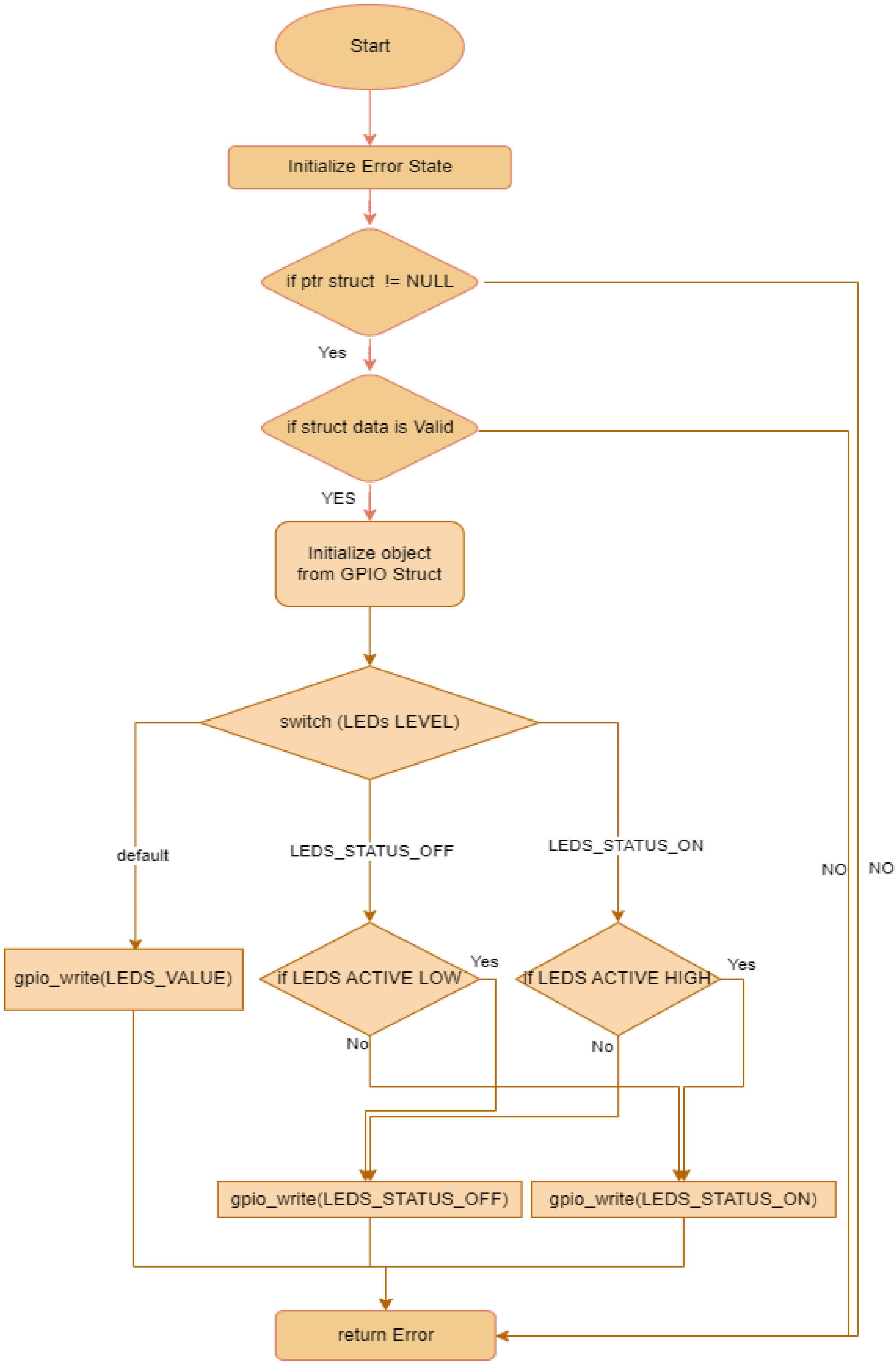
```
enu_gpioErrorState_t gpio_init(const str_gpioPinOrGroupOfPins_t* str_ptr_pinOrGroup ,
enu_gpioPinsLock_t enu_gpioPinsLock ) )
```



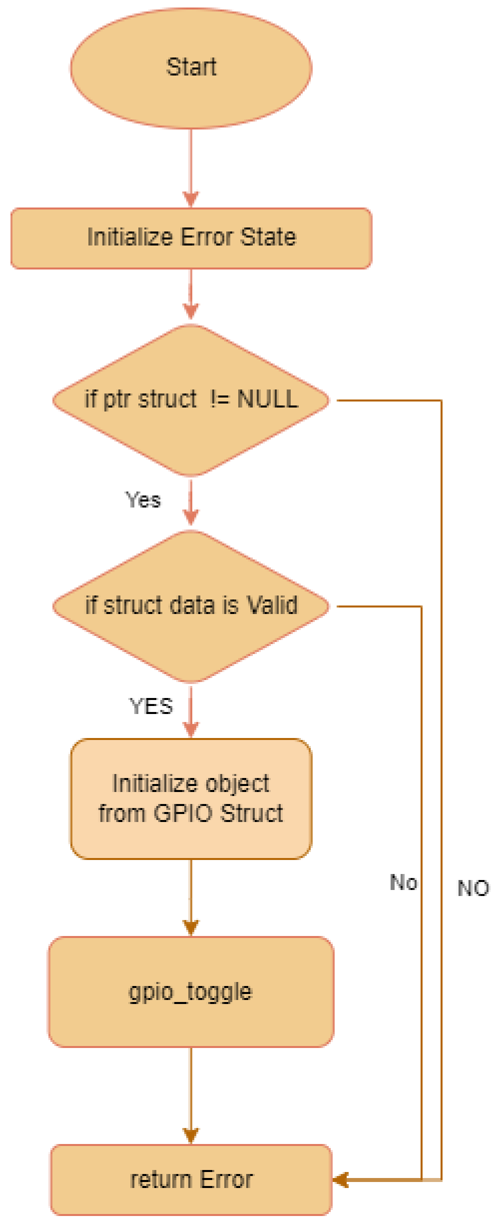
```
enu_ledsErrorState_t leds_deinit(const str_ledsConfig_t * str_ptr_ledOrGroup)
```



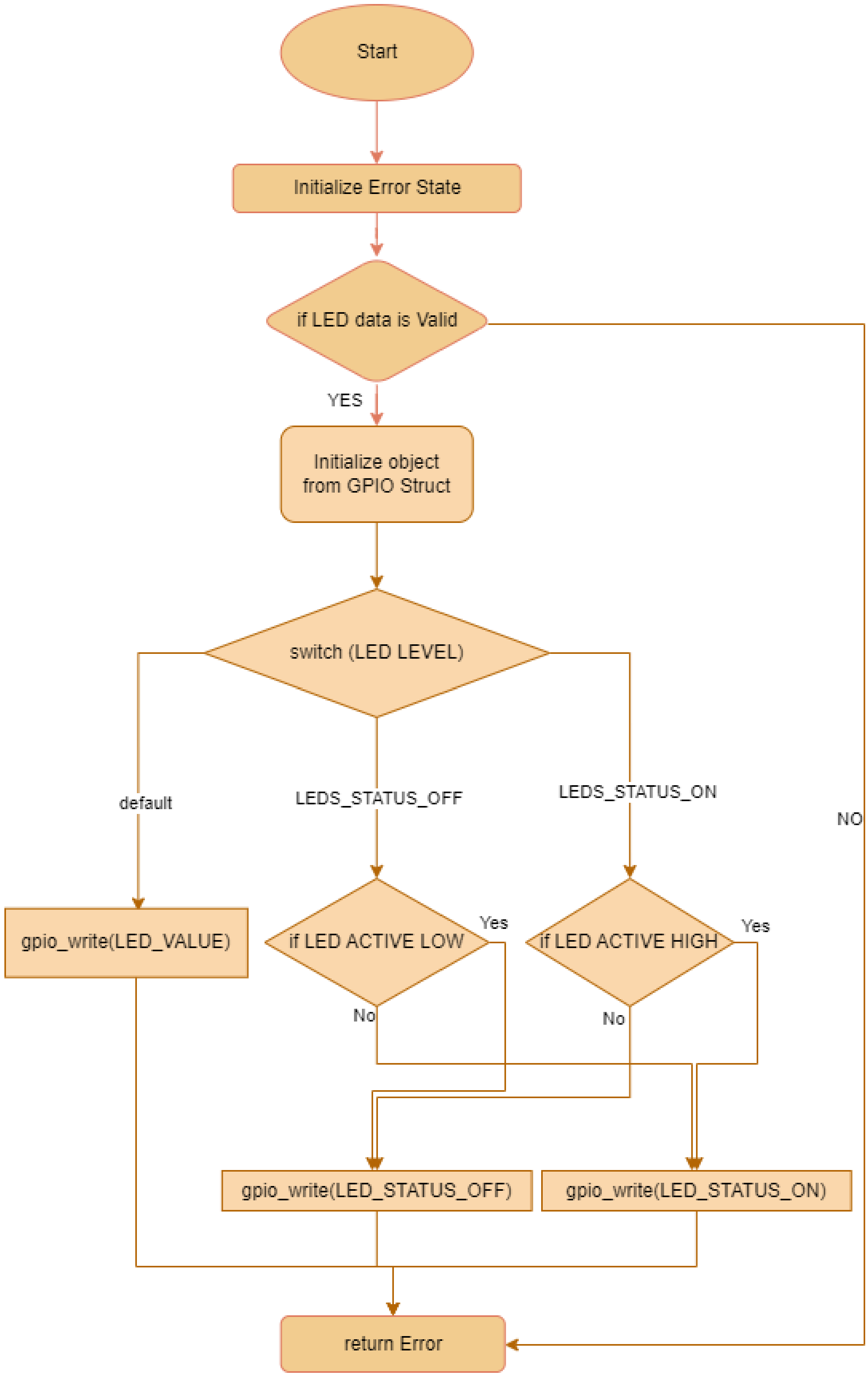
```
enu_ledsErrorState_t leds_changeStatus(const str_ledsConfig_t* str_ptr_ledOrGroup,
enu_ledsStatus_t enu_newLedsStatus)
```



```c
enu_ledsErrorState_t leds_toggle(const str_ledsConfig_t* str_ptr_ledOrGroup)
```

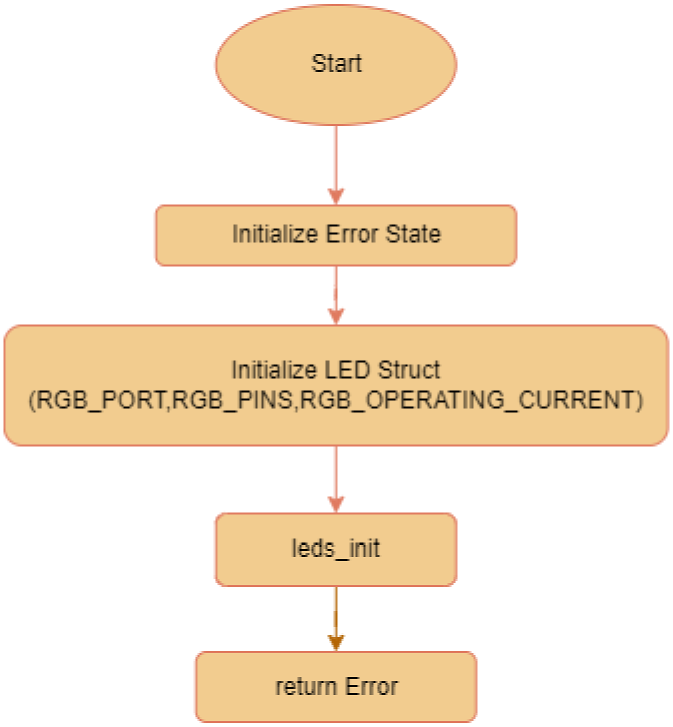


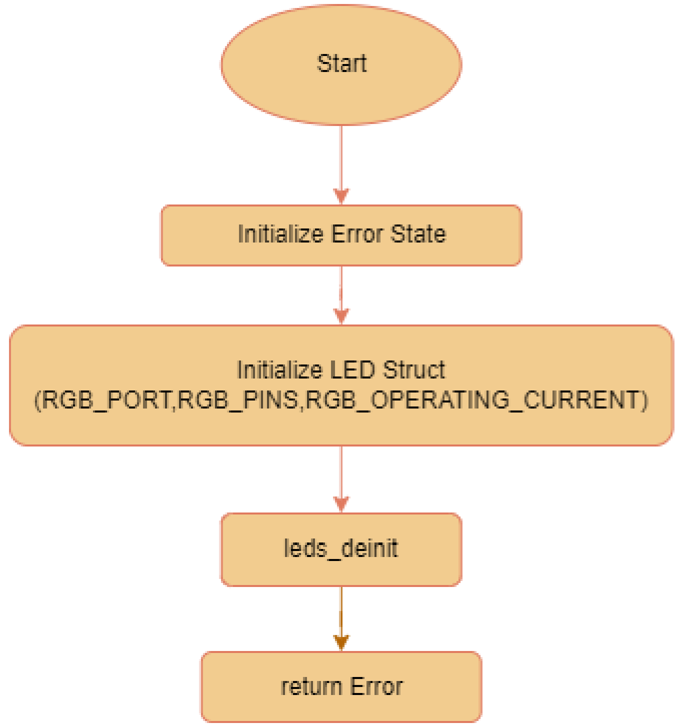
```
enu_ledsErrorState_t leds_changeSingleLEDStatus(enu_ledsPort_t enu_ledPort,
enu_ledPinOrLedsGroup_t enu_ledPin, enu_ledsActiveState_t enu_ledActiveState, enu_ledsStatus_t
enu_newLedStatus)
```



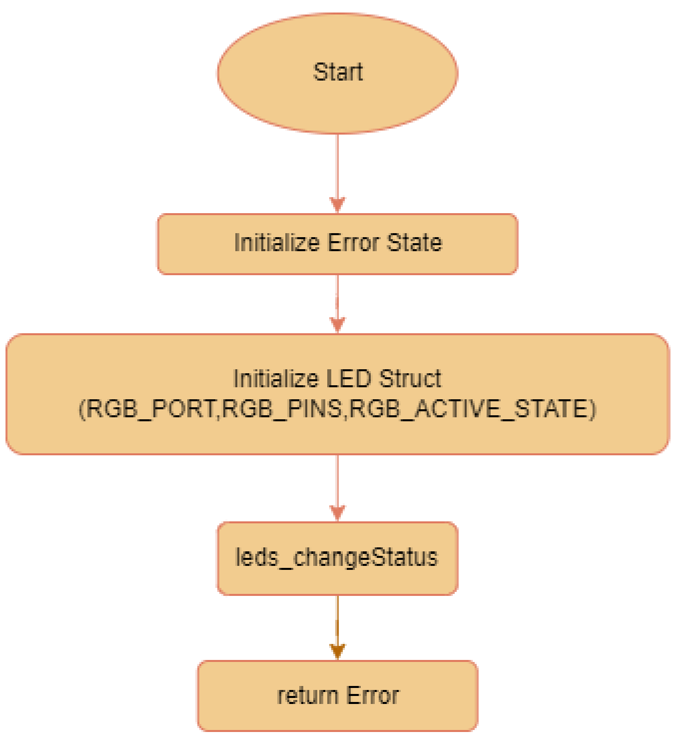
## RGB Functions

`enu_rgbErrorState_t rgb_init(void)` & `enu_rgbErrorState_t rgb_deinit(void)`



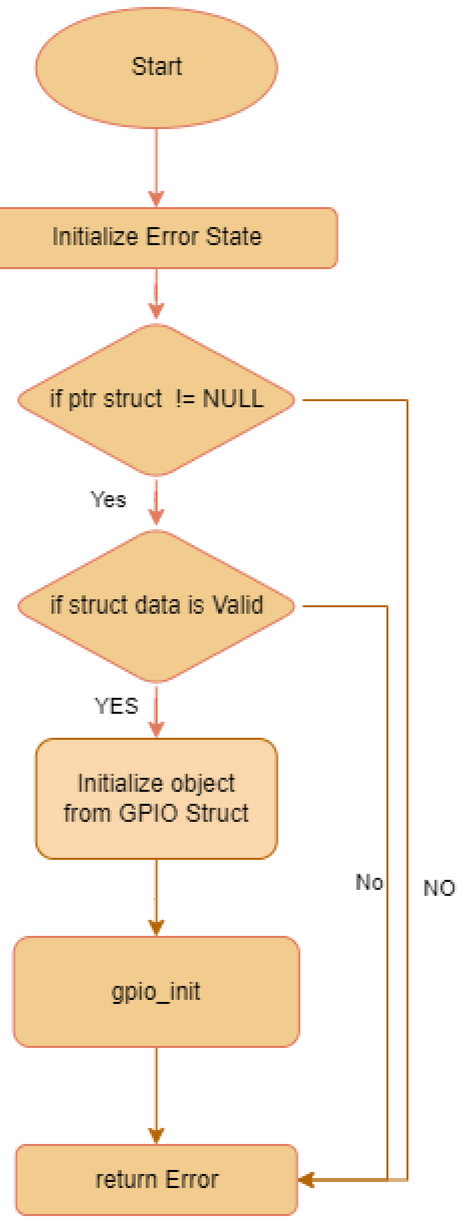




`enu_rgbErrorState_t rgb_changeColor(enu_rgbColorON_t enu_rgbColorON)`

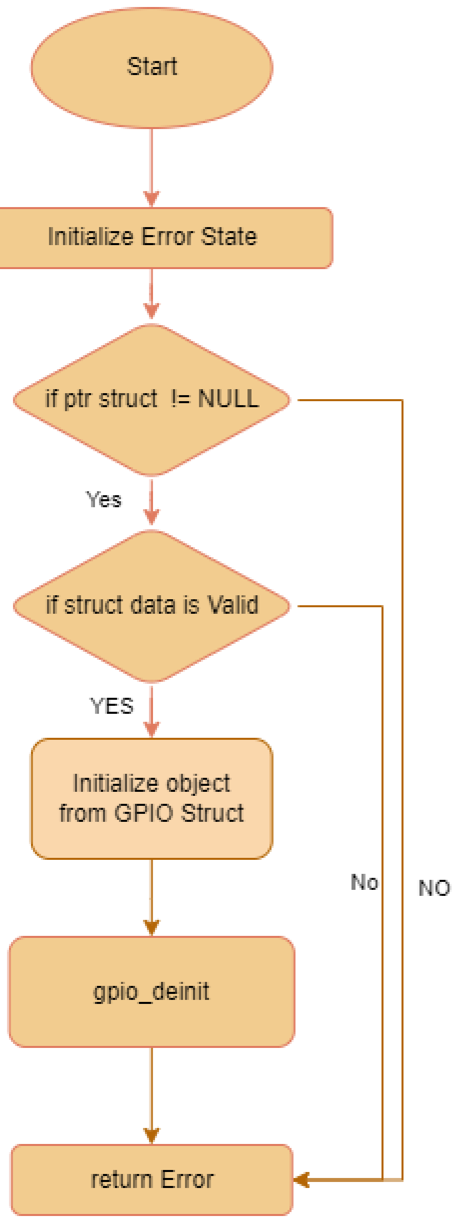


## BUTTON Functions

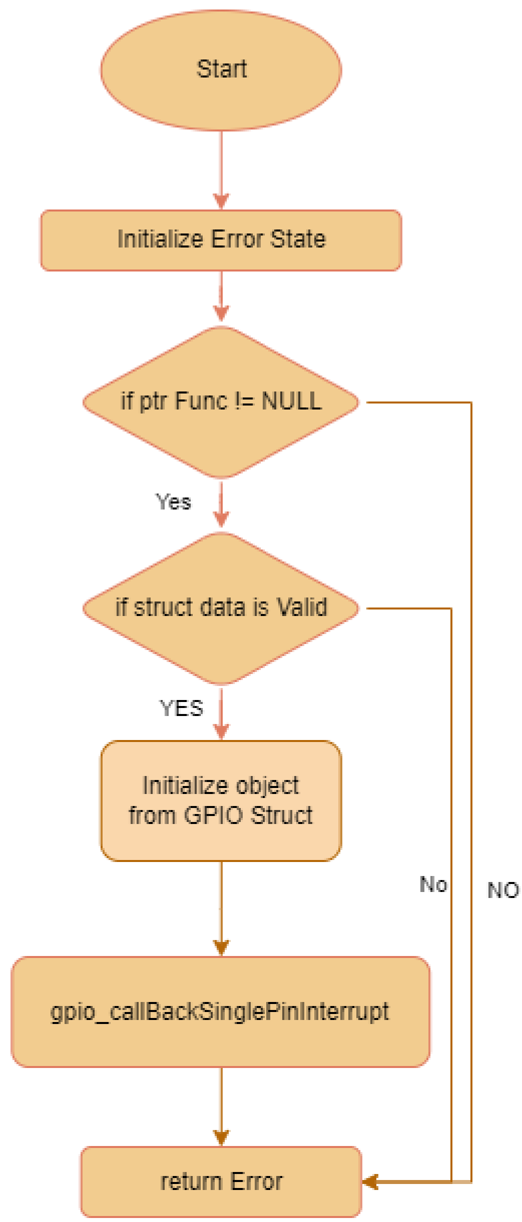
```
enu_buttonsErrorState_t buttons_init(const str_buttonsConfig_t * str_ptr_buttonOrGroup)
```



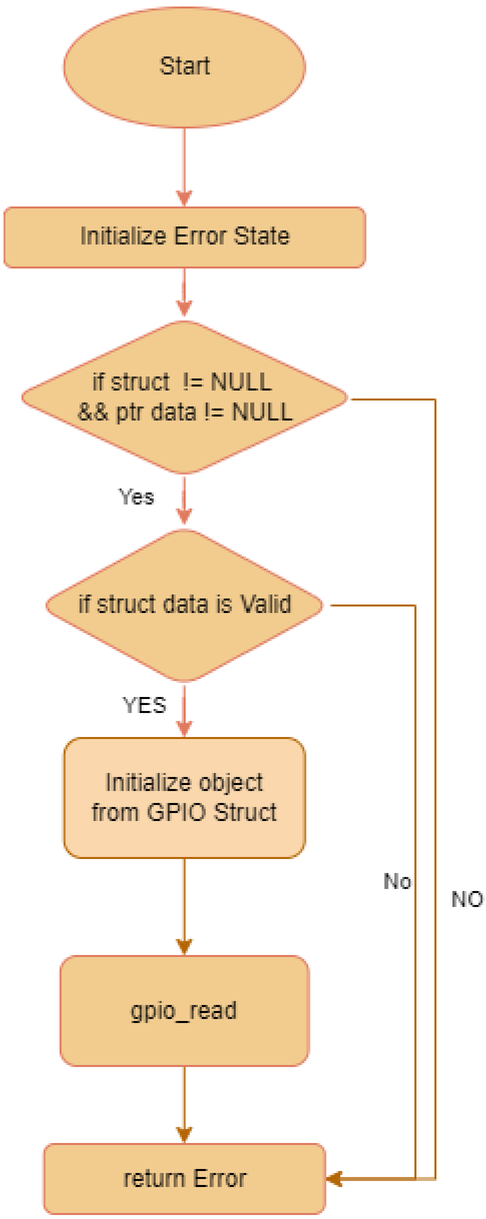
```
enu_buttonsErrorState_t buttons_deinit(const str_buttonsConfig_t * str_ptr_buttonOrGroup)
```



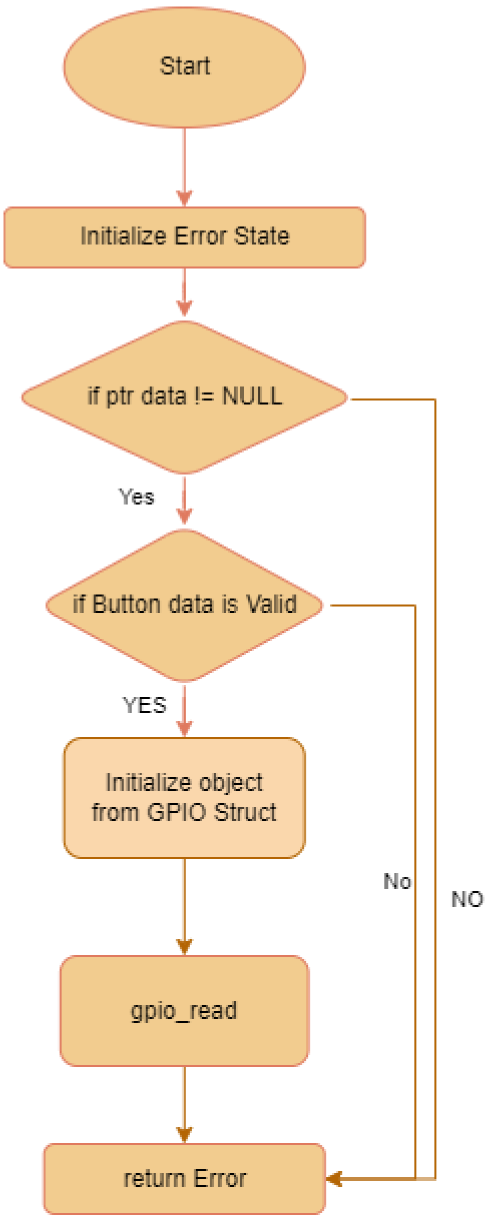
```
enu_buttonsErrorState_t buttons_callBackSingleButtonInterrupt(enu_buttonsPort_t
enu_buttonsPort,enu_buttonPinOrButtonsGroup_t enu_buttonsPin,ptr_Func_buttonsCallBack_t*
ptr_func_interruptCallBack)
```



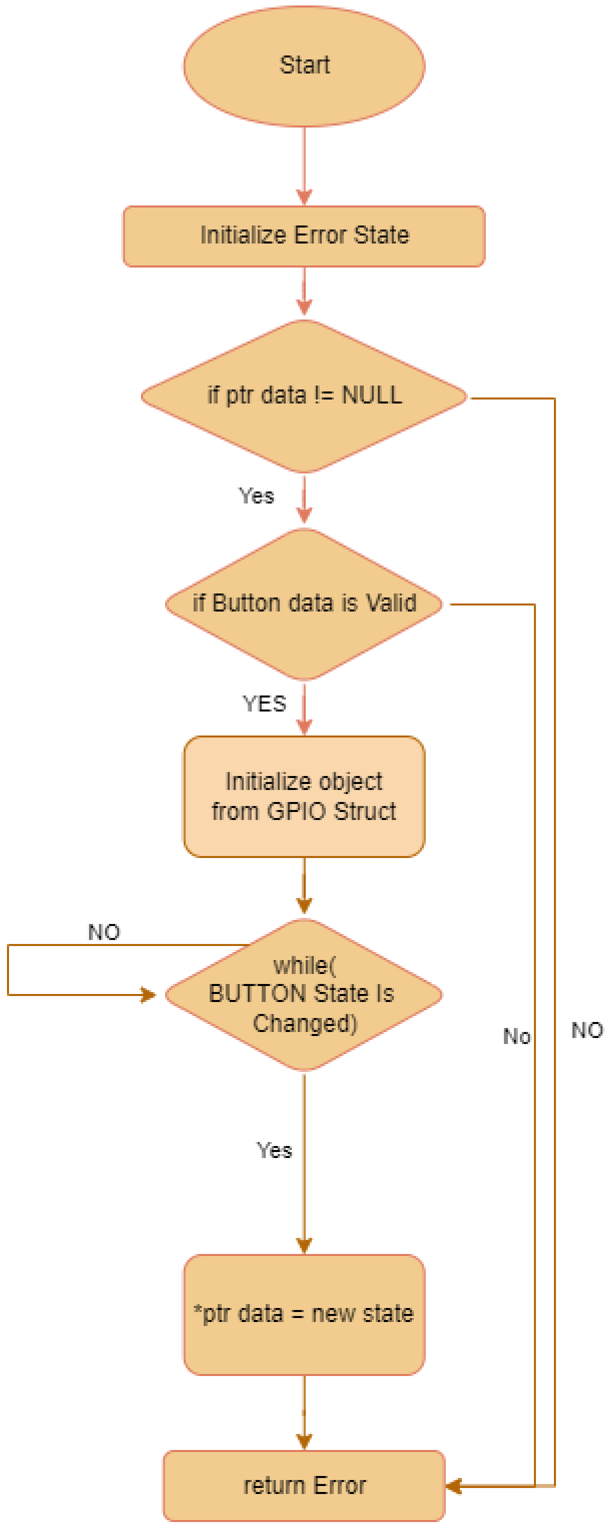
```
enu_buttonsErrorState_t buttons_readStatus(const str_buttonsConfig_t * str_ptr_buttonOrGroup ,
enu_buttonsStatus_t * enu_buttonsStatus)
```



```
enu_buttonsErrorState_t buttons_readSingleButtonStatus(enu_buttonsPort_t
enu_buttonsPort,enu_buttonPinOrButtonsGroup_t enu_buttonsPin, enu_buttonsStatus_t *
enu_buttonsStatus)
```

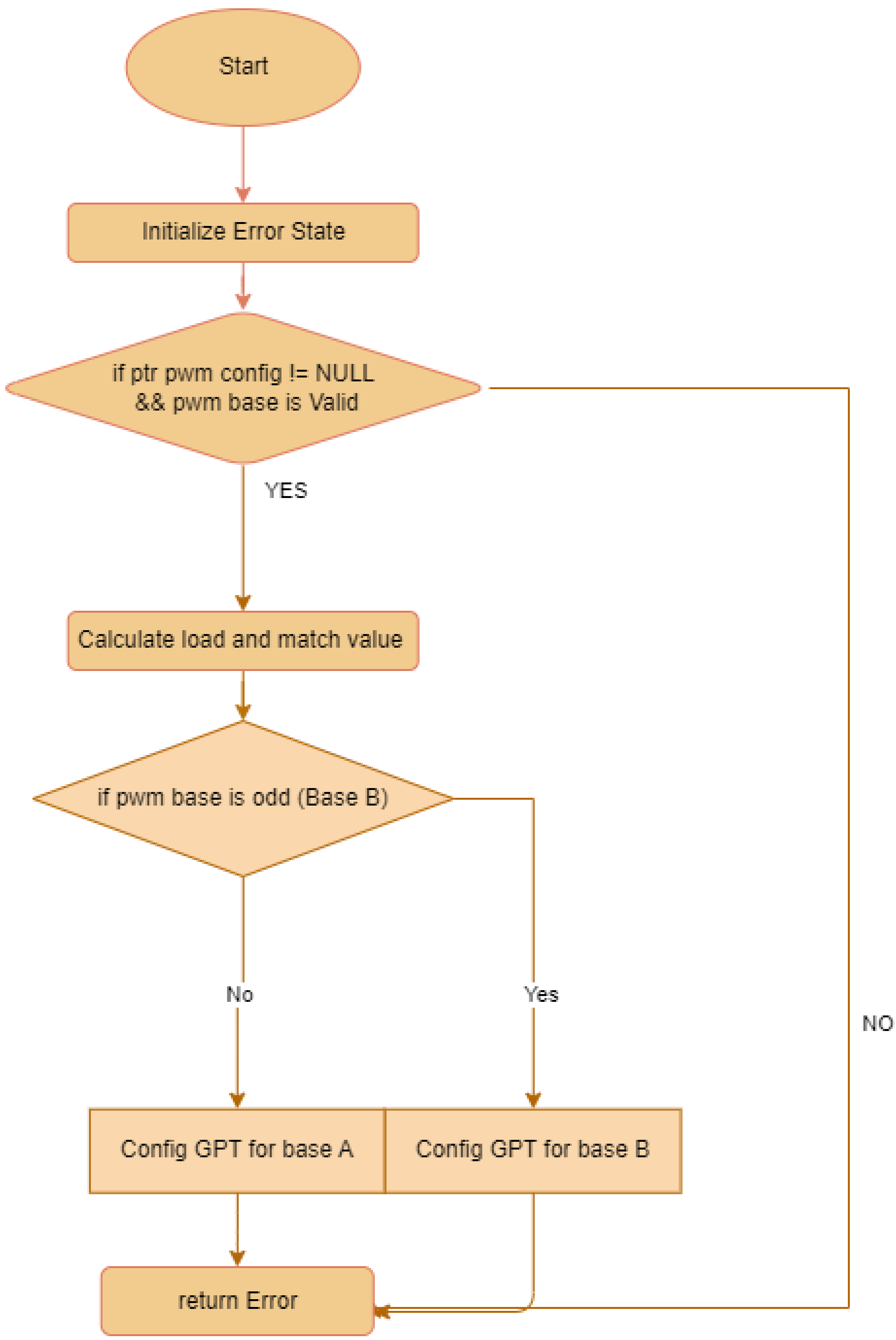


```
enu_buttonsErrorState_t buttons_readSingleButtonChange_Polling(enu_buttonsPort_t
enu_buttonsPort,enu_buttonPinOrButtonsGroup_t enu_buttonsPin, enu_buttonsStatus_t *
enu_buttonsStatus)
```

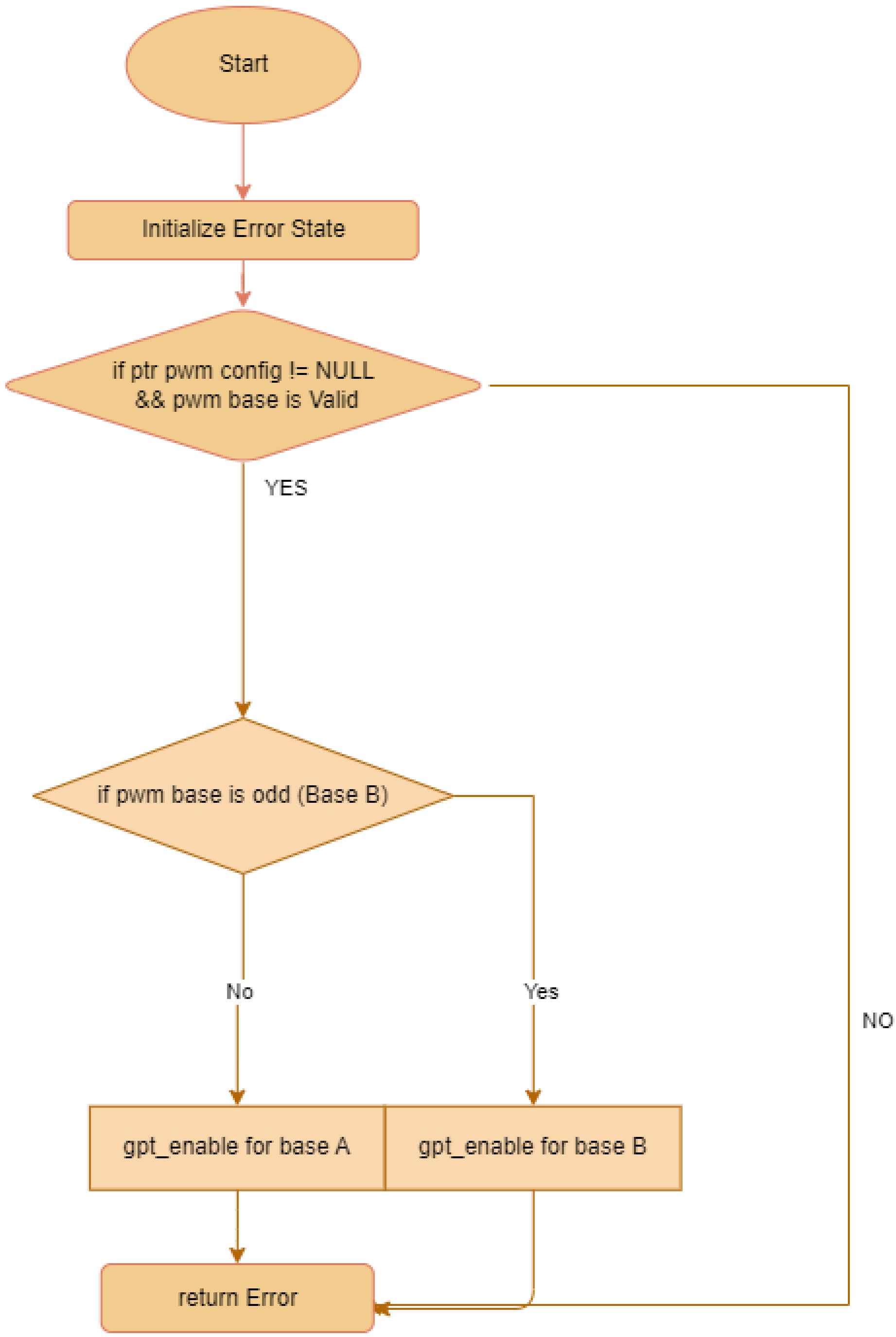


## PWM Functions

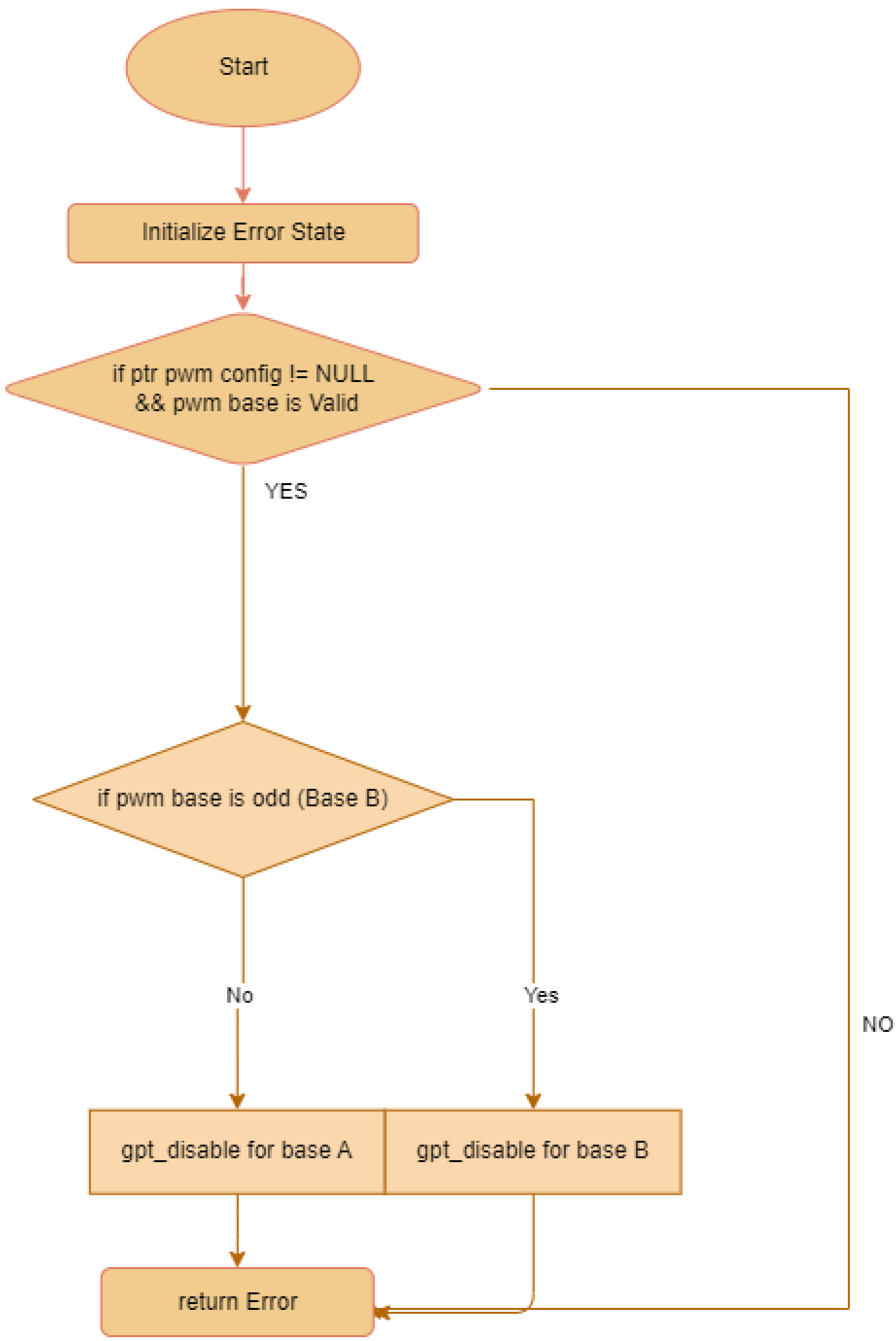
```
enu_pwmErrorState_t pwm_init(const str_pwmCfg_t* ptr_str_l_config, uint8_t enu_pwmPort, uint8_t
enu_pwmPin, uint8_t uint8_l_pwmDutyCycle);
```



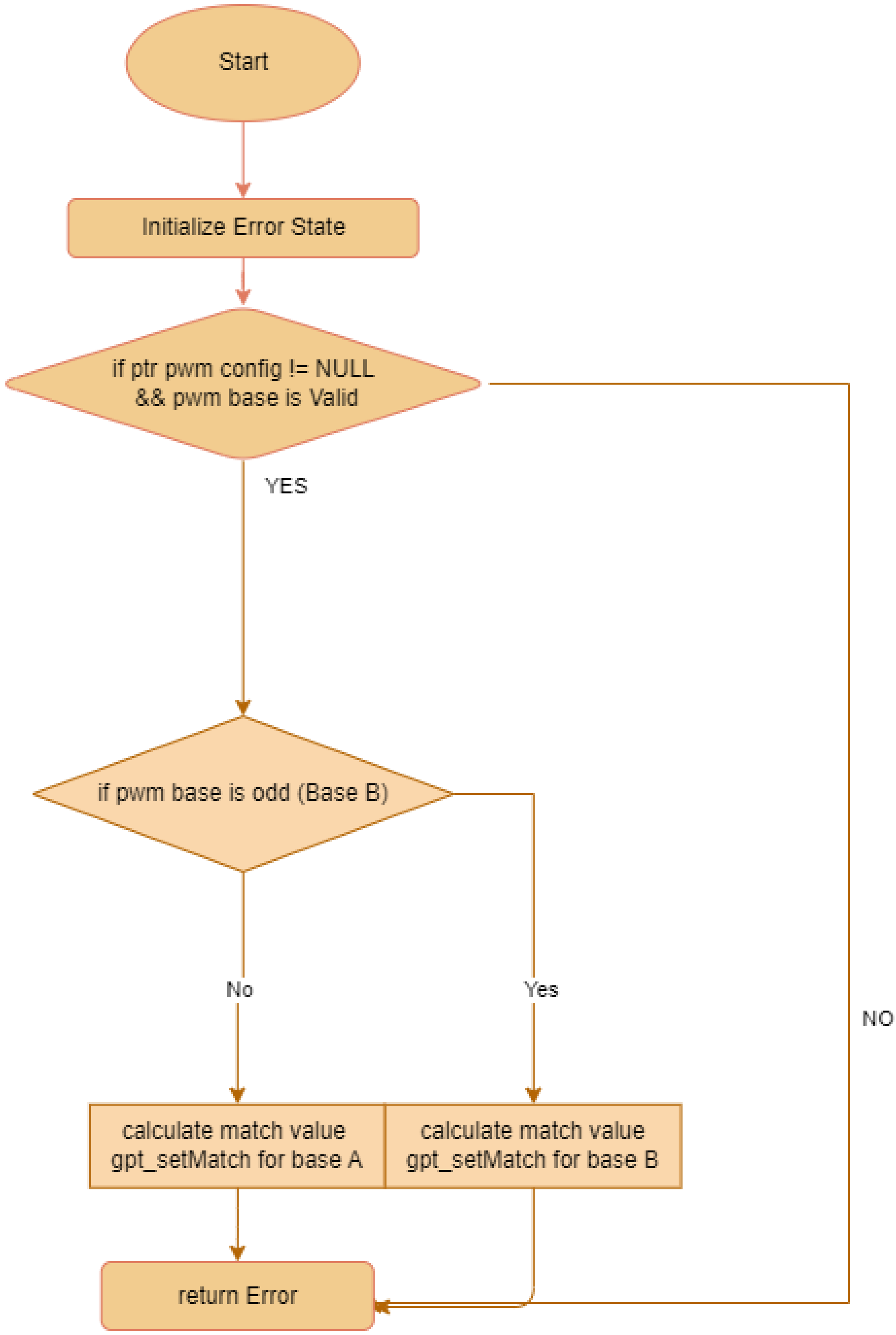
```c
enu_pwmErrorState_t pwm_start(const str_pwmCfg_t* ptr_str_l_config);
```



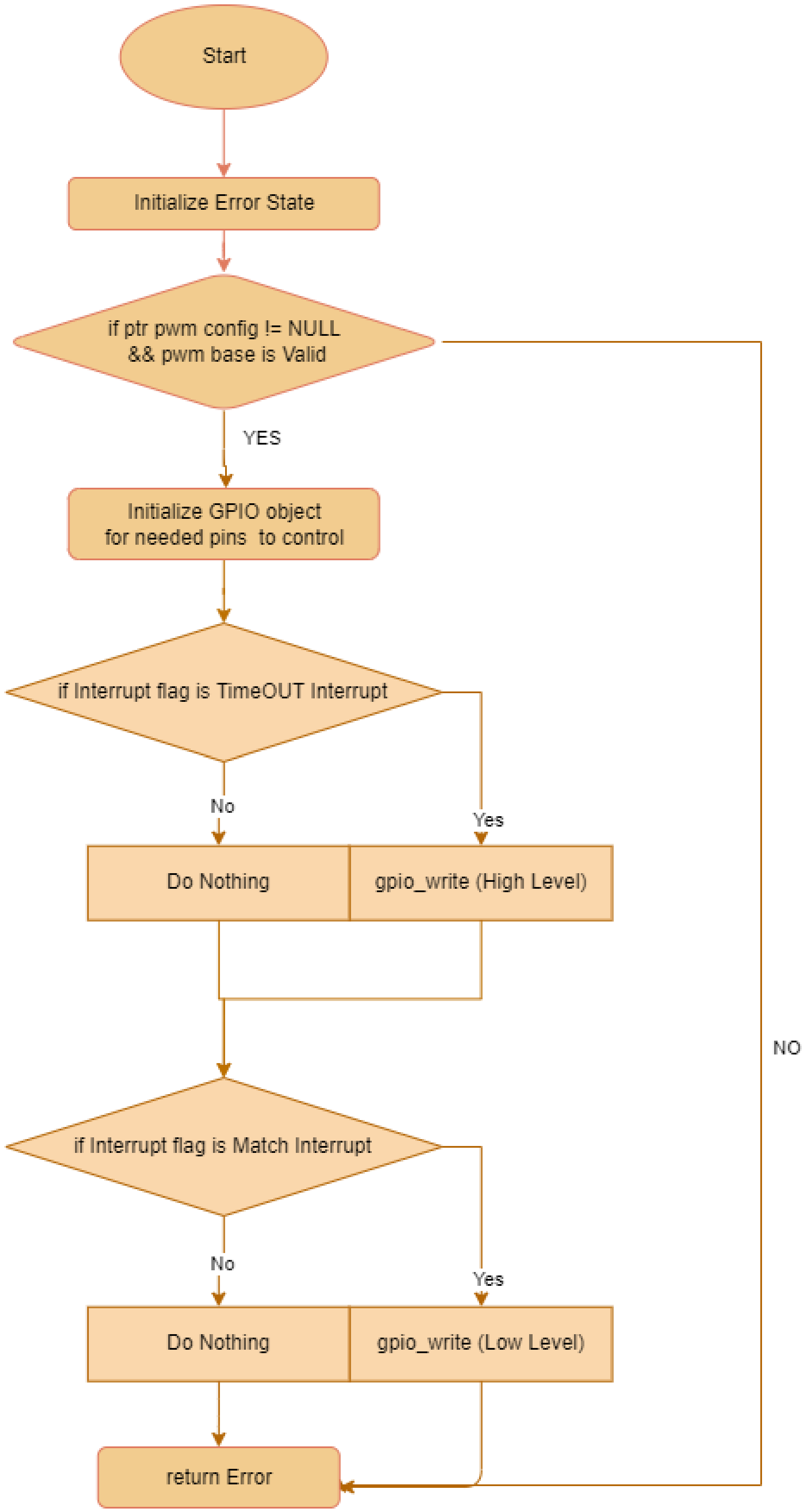
```c
enu_pwmErrorState_t pwm_stop(const str_pwmCfg_t* ptr_str_l_config);
```



```c
enu_pwmErrorState_t pwm_setDutyCycle(const str_pwmCfg_t* ptr_str_l_config, uint8_t
uint8_l_pwmDutyCycle);
```

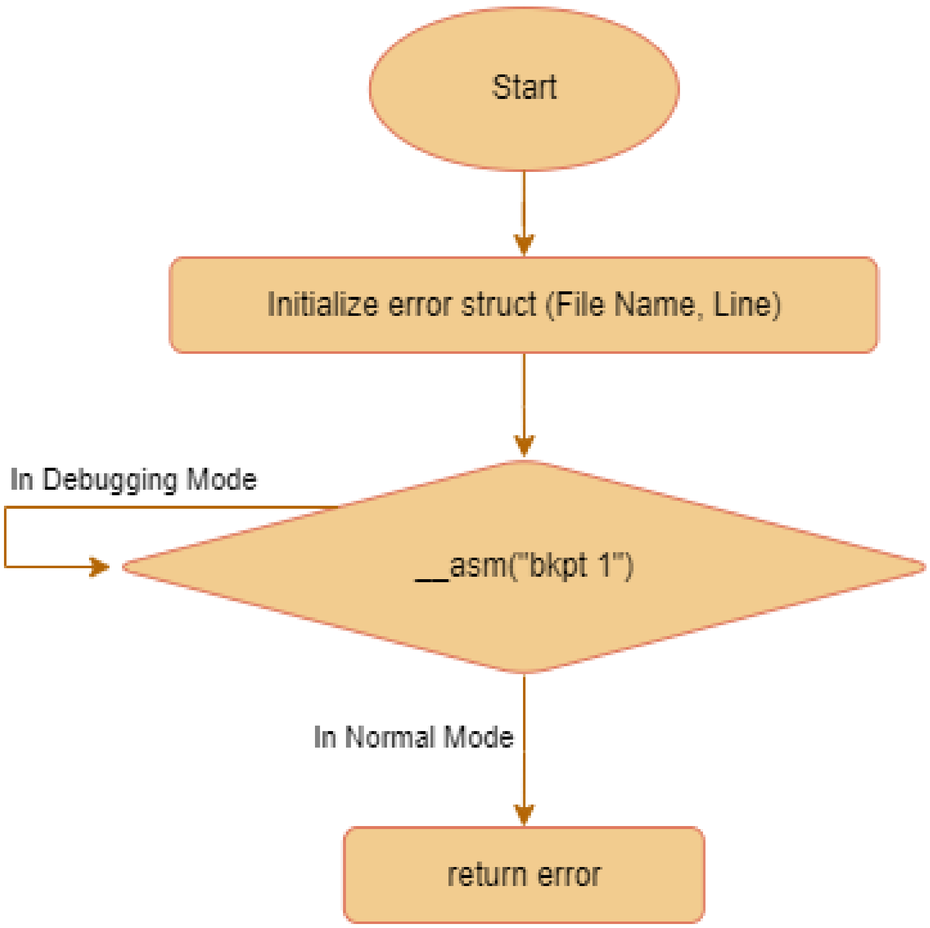


```
static void pwm_callback_0A(void) -> static void pwm_callback_5A(void) -> ( 0B -> 5B )
```



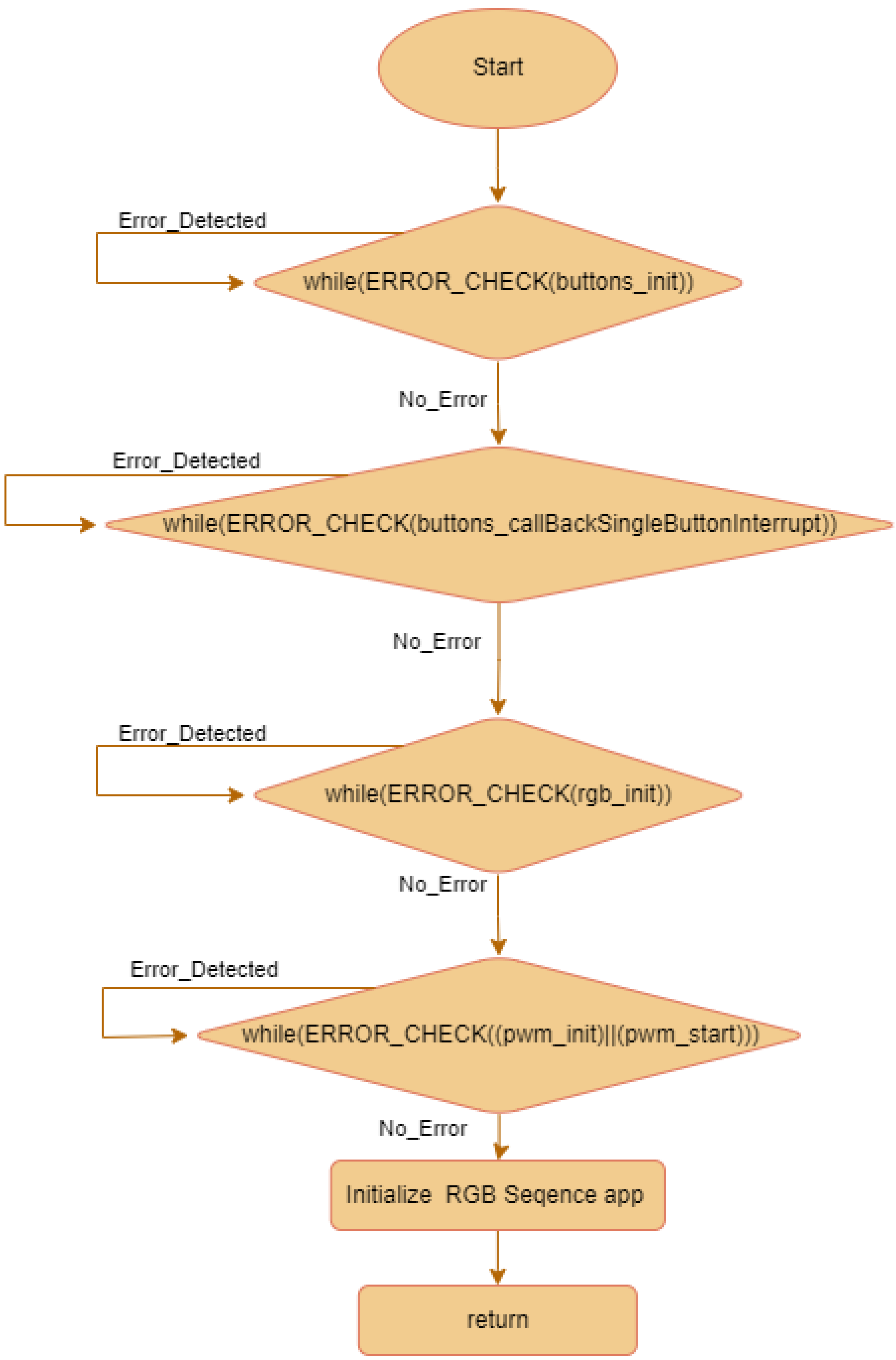
## LIB (ERROR_CHECK) Function

```
enu_stdErrorState_t  ERROR_DETECTED(const uint8_t* uint8_l_flieName, uint32_t uint32_l_line)
```

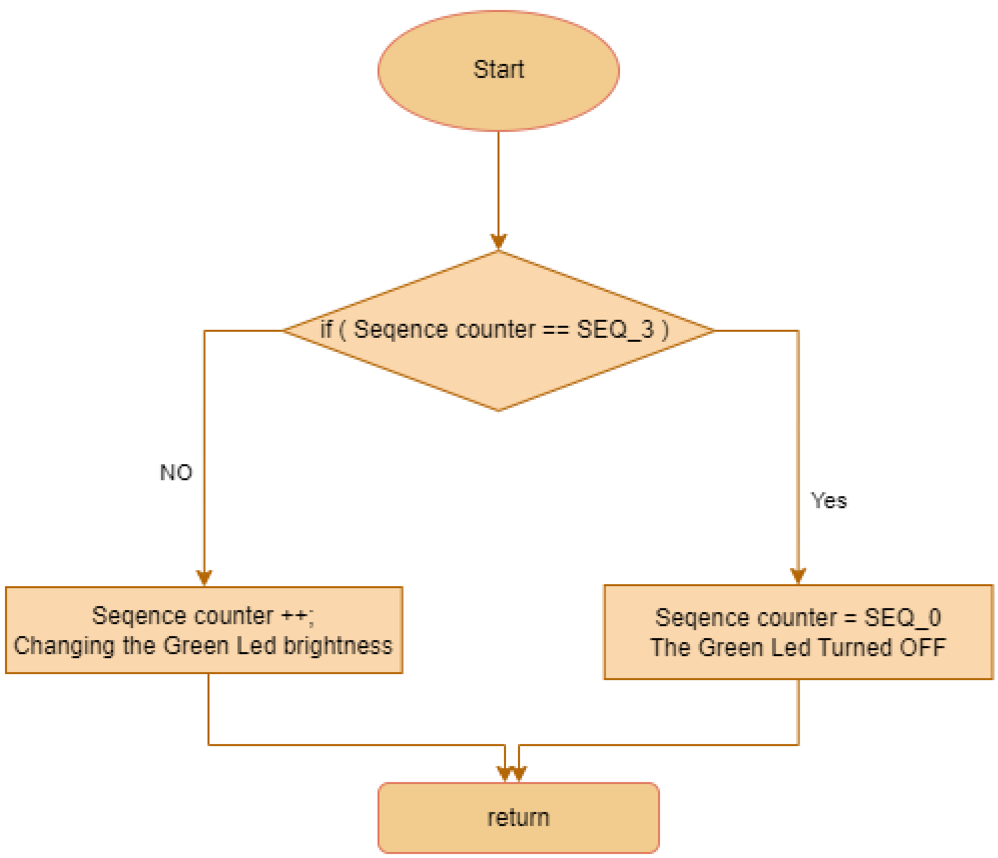


## APP Functions

```
void app_rgbInit(void)
```



```c
static void app_buttonPressed(void)
```



## Configuration

---

RGB_LED:

-------------------------------------------------

Pre-Build Configuration

-------------------------------------------------

```c
#ifndef _RGB_CONFIG_H_
#define _RGB_CONFIG_H_
/**********************************************************************/
/*                           INCLUDE FROM LED                         */
/**********************************************************************/
#include "..\LED\leds_interface.h"
/**********************************************************************/
/*                           Choose Any From                          */
/*LEDS_PORTA, LEDS_PORTB, LEDS_PORTC, LEDS_PORTD, LEDS_PORTE, LEDS_PORTF*/
/**********************************************************************/
#define RGB_PORT                    LEDS_PORTF
/**********************************************************************/
/*                           Choose Any From                          */
/*    LED_P0, LED_P1, LED_P2, LED_P3, LED_P4, LED_P5, LED_P6, LED_P7    */
/**********************************************************************/
#define RED_PIN                      LED_P1
#define GREEN_PIN                    LED_P2
#define BLUE_PIN                     LED_P3
/**********************************************************************/
/*                           Choose Any From                          */
/*       LEDS_OPERATING_CURRENT_2MA, LEDS_OPERATING_CURRENT_4MA       */
/*                    LEDS_OPERATING_CURRENT_8MA                      */
/**********************************************************************/
#define RGB_OPERATING_CURRENT LEDS_OPERATING_CURRENT_2MA
/**********************************************************************/
/*                           Choose Any From                          */
/*                 LEDS_ACTIVE_HIGH, LEDS_ACTIVE_LOW                  */
/**********************************************************************/
#define RGB_ACTIVE_STATE         LEDS_ACTIVE_HIGH
/**********************************************************************/
#endif /* _RGB_CONFIG_H_ */
```

-------------------------------------------------

Linking Configuration

-------------------------------------------------

```c
/*                           INCLUDE FROM LED                         */
#include "..\LED\leds_interface.h"
/**********************************************************************/
typedef enum
{
   RED_PIN   = LED_P1,
   GREEN_PIN = LED_P2,
   BLUE_PIN  = LED_P3,
  RGB_PORT  = LEDS_PORTF,
}enu_rgbInfo_t;
```

```c
str_ledsConfig_t  str_gl_rgbConfig =
{
   .enu_ledsPort = RGB_PORT,
   .enu_ledsPinOrGroup = RED_PIN|GREEN_PIN|BLUE_PIN,
   .enu_ledsActiveState = LEDS_ACTIVE_HIGH,
   .enu_ledsOperatingCurrent = LEDS_OPERATING_CURRENT_2MA,
};
/*************************************************************************/
```

GPT:

--------------------------------------------

Run-Time Configuration

--------------------------------------------

```c
typedef enum
{/*************************************************************************/
    GPT_OK = 0,
    GPT_WRONG_INPUT,
    /*************************************************************************/
} enu_gptErrorState_t;
/*************************************************************************/
typedef enum
{/*************************************************************************/
    GPT_BASE_0 = 0,
    GPT_BASE_1,
    GPT_BASE_2,
    GPT_BASE_3,
    GPT_BASE_4,
    GPT_BASE_5,
    /*************************************************************************/
} enu_gptBase_t;
/*************************************************************************/
typedef enum
{/*************************************************************************/
    GPT_CFG_ONE_SHOT    = 0x00000021,
    GPT_CFG_PERIODIC    = 0x00000022,
    GPT_CFG_SPLIT_PAIR  = 0x04000000,
    GPT_CFG_A_ONE_SHOT  = 0x00000021,
    GPT_CFG_A_PERIODIC  = 0x00000022,
    GPT_CFG_B_ONE_SHOT  = 0x00002100,
    GPT_CFG_B_PERIODIC  = 0x00002200,
    /*************************************************************************/
} enu_gptConfig_t;
/*************************************************************************/
typedef enum
{/*************************************************************************/
    GPT_INT_A_TIMEOUT   = 0x00000001,
    GPT_INT_A_MATCH     = 0x00000010,
    GPT_INT_B_TIMEOUT   = 0x00000100,
    GPT_INT_B_MATCH     = 0x00000800,
} enu_gptInterruptFlags_t;
/*************************************************************************/
typedef enum
{/*************************************************************************/
    GPT_TIMER_A    ,
    GPT_TIMER_B    ,
    GPT_A_AND_B    ,
} enu_gptTimerName_t;
/*************************************************************************/
```

LED:

-------------------------------------------------

Run-Time Configuration

-------------------------------------------------

```
/*********************************************************************/
typedef enum
{
   /*********************************************************************/
   LEDS_OK   =   STATE_OK, /* STATE_OK From LIB  #include "error_check.h" */
   LEDS_WRONG_INPUT_VALUE,
   LEDS_WRONG_INPUT_NULL ,
  LEDS_NOT_INITIALIZED  ,
   /*********************************************************************/
}enu_ledsErrorState_t;
/*********************************************************************/
typedef enum
{
   /*********************************************************************/
   LEDS_PORTA = 0,
   LEDS_PORTB     ,
   LEDS_PORTC     ,
   LEDS_PORTD     ,
   LEDS_PORTE     ,
   LEDS_PORTF     ,
   /*********************************************************************/
}enu_ledsPort_t;
/*********************************************************************/
typedef enum  /* You Can Use It To Choose Group Of LEDs = (LED_P0 | LED_P5 | LED_P7 )
 , Or Only One Led  */
{
   /*********************************************************************/
   LED_NO_PIN_SELECTED = 0,
   /*********************************************************************/
   LED_P0 = 0b1,
   LED_P1 = 0b10,
   LED_P2 = 0b100,
   LED_P3 = 0b1000,
   LED_P4 = 0b10000,
   LED_P5 = 0b100000,
   LED_P6 = 0b1000000,
   LED_P7 = 0b10000000,
   /*********************************************************************/
   LEDS_ALL_PINS = 0xFF,
   LEDS_P0_TO_P3 = 0x0F,
   LEDS_P4_TO_P7 = 0xF0,
   /*********************************************************************/
} enu_ledPinOrLedsGroup_t;
/*********************************************************************/
typedef enum  /* You Can Operate Choosen Leds with Value From 0 to 255  or  Use
On or Off State For All Choosen Leds */
{
   /*********************************************************************/
   LEDS_STATUS_MAX_VALUE = 0x0FF,
   LEDS_STATUS_OFF       = 0x100,
   LEDS_STATUS_ON        = 0x101,
   /*********************************************************************/
}enu_ledsStatus_t;
/*********************************************************************/
```

```c
typedef enum
{
  /*********************************************************************/
  LEDS_ACTIVE_LOW  = 0,
  LEDS_ACTIVE_HIGH    ,
  /*********************************************************************/
} enu_ledsActiveState_t;
/*********************************************************************/
typedef enum
{
  /*********************************************************************/
  LEDS_OPERATING_CURRENT_2MA = 0, //Default as Output
  LEDS_OPERATING_CURRENT_4MA       ,
  LEDS_OPERATING_CURRENT_8MA       ,
  /*********************************************************************/
} enu_ledsOperatingCurrent_t;
/*********************************************************************/
/*************************      LEDS STRUCT       **********************/
/*********************************************************************/
typedef struct
{
  /*********************************************************************/
  enu_ledsPort_t             enu_ledsPort;
  enu_ledPinOrLedsGroup_t    enu_ledsPinOrGroup;
  enu_ledsActiveState_t      enu_ledsActiveState;
  enu_ledsOperatingCurrent_t enu_ledsOperatingCurrent;
  /*********************************************************************/
}str_ledsConfig_t;
#endif /* _LED_INTERFACE_H_ */

/*********************************************************************/
```

BUTTON:

-----------------------------------------------

Run-Time Configuration

-----------------------------------------------

```c
#ifndef _BUTTON_INTERFACE_H_
#define _BUTTON_INTERFACE_H_
/*********************************************************************/
typedef void (ptr_Func_buttonsCallBack_t) (void);
/*********************************************************************/

typedef enum
{
  /*********************************************************************/
  BUTTONS_OK   =   STATE_OK, /* STATE_OK From LIB  #include "error_check.h" */
  BUTTONS_WRONG_INPUT_VALUE,
  BUTTONS_WRONG_INPUT_NULL ,
  BUTTONS_NOT_INITIALIZED   ,
  /*********************************************************************/
}enu_buttonsErrorState_t;
/*********************************************************************/
typedef enum
{
  /*********************************************************************/
  BUTTONS_PORTA = 0,
  BUTTONS_PORTB       ,
  BUTTONS_PORTC       ,
```

```c
   BUTTONS_PORTD         ,
   BUTTONS_PORTE         ,
   BUTTONS_PORTF         ,
   /*************************************************************************/
}enu_buttonsPort_t;
/*************************************************************************/
typedef enum  /* You Can Use It To Choose Group Of BUTTONs = (BUTTON_P0 | BUTTON_P5 |
 BUTTON_P7 ) , Or Only One BUTTON  */
{
   /*************************************************************************/
   BUTTON_NO_PIN_SELECTED = 0,
   /*************************************************************************/
   BUTTON_P0 = 0b1,
   BUTTON_P1 = 0b10,
   BUTTON_P2 = 0b100,
   BUTTON_P3 = 0b1000,
   BUTTON_P4 = 0b10000,
   BUTTON_P5 = 0b100000,
   BUTTON_P6 = 0b1000000,
   BUTTON_P7 = 0b10000000,
   /*************************************************************************/
   BUTTONS_ALL_PINS = 0xFF,
   BUTTONS_P0_TO_P3 = 0x0F,
   BUTTONS_P4_TO_P7 = 0xF0,
   /*************************************************************************/
} enu_buttonPinOrButtonsGroup_t;
/*************************************************************************/
typedef enum  /* You Can Read From Choosen BUTTONs Value From 0 to 255  or  Read Low
or High State Form All Choosen BUTTONs */
{
   /*************************************************************************/
   BUTTONS_PIN_STATUS_MAX_VALUE = 0x0FF,
   BUTTONS_PIN_STATUS_LOW       = 0x100,
   BUTTONS_PIN_STATUS_HIGH      = 0x101,
   /*************************************************************************/
}enu_buttonsStatus_t;
/*************************************************************************/
typedef enum
{
   /*************************************************************************/
   BUTTONS_EXTERNAL_PULL_RES  = 0,
   BUTTONS_INTRTNAL_PULL_UP      ,
    BUTTONS_INTRTNAL_PULL_DOWN    ,
   /*************************************************************************/
} enu_buttonsPullMode_t;
/*************************************************************************/
typedef enum
{
   /*************************************************************************/
   BUTTONS_NO_INTERRUPT    = 0,
   BUTTONS_CHANGE_RISING_EDGE ,
   BUTTONS_CHANGE_FALLING_EDGE,
   BUTTONS_CHANGE_BOTH_EDGES  ,
   /*************************************************************************/
} enu_buttonsInterruptMode_t;

/*************************************************************************/
/**********************      BUTTONS STRUCT       ***********************/
/*************************************************************************/
```

```c
typedef struct
{
   /*************************************************************************/
   enu_buttonsPort_t               enu_buttonsPort;
   enu_buttonPinOrButtonsGroup_t    enu_buttonsPinOrGroup;
   enu_buttonsPullMode_t            enu_buttonsPullMode;
  enu_buttonsInterruptMode_t        enu_buttonsInterruptMode;
   /*************************************************************************/
}str_buttonsConfig_t;
#endif /* _BUTTON_INTERFACE_H_ */
/*************************************************************************/
```

GPIO:

--------------------------------------------

Run-Time Configuration

--------------------------------------------

```c
#ifndef _GPIO_INTERFACE_H__
#define _GPIO_INTERFACE_H__
/*************************************************************************/
typedef void (ptr_Func_gpioCallBack_t) (void);
/*************************************************************************/
typedef enum
{
   /*************************************************************************/
   GPIO_OK = STATE_OK       ,  /* STATE_OK From LIB  #include "error_check.h" */
   GPIO_WRONG_INPUT_VALUE   ,
   GPIO_WRONG_INPUT_NULL    ,
   GPIO_NOT_INITIALIZED_OR_LOCKED,
   /*************************************************************************/
} enu_gpioErrorState_t;
/*************************************************************************/
typedef enum
{
   /*************************************************************************/
   GPIO_UNLOCK ,
   GPIO_LOCK   ,
   /*************************************************************************/
} enu_gpioPinsLock_t;
/*************************************************************************/
typedef enum
{
   GPIO_DIRECTION_INPUT = 0,
   GPIO_DIRECTION_OUTPUT    ,
} enu_gpioPinDirection_t;
/*************************************************************************/
typedef enum
{
   /*************************************************************************/
   GPIO_INPUT_NO_DRIVE    = 0,
   /*************************************************************************/
   GPIO_OUTPUT_DRIVE_2MA = 1, //Default as Output
   GPIO_OUTPUT_DRIVE_4MA     ,
   GPIO_OUTPUT_DRIVE_8MA     ,
   /*************************************************************************/
} enu_gpioPinDrive_t;
/*************************************************************************/
```

```c
typedef enum
{
   /**********************************************************************/
   GPIO_FLOATING = 0 ,
   GPIO_PULLUP       ,
   GPIO_PULLDOWN     ,
   /**********************************************************************/
} enu_gpioPullMode_t;
/**********************************************************************/
typedef enum
{
   /**********************************************************************/
   GPIO_MODE_DIGITAL = 0,
   GPIO_MODE_ANALOG     ,
   GPIO_MODE_AF         ,
   /**********************************************************************/
} enu_gpioPinModeConfig_t;
/**********************************************************************/
typedef enum
{
   /**********************************************************************/
   GPIO_NO_INTERRUPT = 0,
   GPIO_RISING_EDGE     ,
   GPIO_FALLING_EDGE    ,
   GPIO_BOTH_EDGES      ,
   /**********************************************************************/
} enu_gpioInterruptMode_t;
/**********************************************************************/
typedef enum
{
   /**********************************************************************/
   GPIO_PORTA = 0,
   GPIO_PORTB    ,
   GPIO_PORTC    ,
   GPIO_PORTD    ,
   GPIO_PORTE    ,
   GPIO_PORTF    ,
   /**********************************************************************/
} enu_gpioPort_t;
/**********************************************************************/
typedef enum  /* You Can Use It As Group Of Pins = (GPIO_PIN_0 | GPIO_PIN_5 | GPIO_PIN_6)
 And So On */
{
   /**********************************************************************/
   GPIO_NO_PIN     = 0,
   /**********************************************************************/
   GPIO_PIN_0      = 0b1,
   GPIO_PIN_1      = 0b10,
   GPIO_PIN_2      = 0b100,
   GPIO_PIN_3      = 0b1000,
   GPIO_PIN_4      = 0b10000,
   GPIO_PIN_5      = 0b100000,
   GPIO_PIN_6      = 0b1000000,
   GPIO_PIN_7      = 0b10000000,
   /**********************************************************************/
   GPIO_ALL_PINS   = 0xFF,
   GPIO_P0_TO_P3   = 0x0F,
   GPIO_P4_TO_P7   = 0xF0,
   /**********************************************************************/
} enu_gpioPins_t;
/**********************************************************************/
```

```c
typedef enum  /* You Can Use Value From 0 to 255 For Chosen Pins and Can Use Low or High Level For All Chosen Pins */
{
  /*************************************************************************/
  GPIO_MAX_VALUE  = 0x0FF,
  GPIO_LOW_LEVEL  = 0x100,
  GPIO_HIGH_LEVEL = 0x101,
  /*************************************************************************/
} enu_gpioLevelOrValue_t;
/*************************      GPIO STRUCT       ***********************/
typedef struct
{/**************************************************************************/
  enu_gpioPort_t                      enu_gpioPort;
  enu_gpioPins_t                      enu_gpioPinOrGroup;
  enu_gpioPinModeConfig_t   enu_modeConfig;
  enu_gpioPinDirection_t      enu_direction;
  enu_gpioPinDrive_t          enu_pinDrive;
  enu_gpioPullMode_t          enu_pullMode;
  enu_gpioInterruptMode_t   enu_interruptMode;
} str_gpioPinOrGroupOfPins_t;
#endif
/*************************************************************************/
```

PWM :

-------------------------------------------

Run-Time Configuration

-------------------------------------------

```c
typedef enum
{
    /*************************************************************************/
    PWM_OK = STATE_OK, /* STATE_OK From LIB  #include "error_check.h" */
    PWM_WRONG_INPUT,
    /*************************************************************************/
}enu_pwmErrorState_t;

typedef enum
{/*************************************************************************/
    PWM_0A,
    PWM_0B,
    PWM_1A,
    PWM_1B,
    PWM_2A,
    PWM_2B,
    PWM_3A,
    PWM_3B,
    PWM_4A,
    PWM_4B,
    PWM_5A,
    PWM_5B,
    /*************************************************************************/
} enu_pwmIndx_t;
typedef struct
{/*************************************************************************/
    enu_pwmIndx_t               enu_pwmIndx;
    uint32_t                    uint32_pwmFreq;
} str_pwmCfg_t;
    /*************************************************************************/
```