SPRINTS

Project Design

MOVING CAR

Version 2.0

Prepared by:

Hazem Ashraf

Mohamed Abdelsalam

Abdelrhman Walaa

June 2023

Table of Content

Moving Car Design

1. Project Introduction

This project involves developing a robot using *four motors*, *two buttons*, and *four LEDs*, with the control system implemented on a **Tiva C** board. The Tiva C board utilizes **ARM** processors, enabling precise control and efficient programming.

The robot is designed to move in a continuous *rectangular* path, showcasing the programmer's skills in controlling the robot's movements and designing an effective control system. The four motors provide the necessary propulsion for the robot, while the two buttons allow for user interaction and control.

Additionally, the four LEDs can be utilized for status indication or visual feedback. Overall, this project demonstrates the programmer's proficiency in programming robotic systems, utilizing ARM processors, and implementing a control system that enables precise movements and efficient operation

1.1. Car Components

1.1.1. Hardware Requirements

1. **Four** motors **(M1, M2, M3, M4)**
2. **One** button to start **(PB1)**
3. **One** button for stop **(PB2)**
4. **Four** LEDs **(LED1, LED2, LED3, LED4)**

1.1.2. Software Requirements

1. The car *starts* initially from **0** speed.
2. When **PB1** is pressed, the car will *move forward* after **1** second.
3. The car will move forward to create the *longest* side of the rectangle for **3** seconds with **50**% of its maximum speed.
4. After finishing the first *longest* side the car will *stop* for *0.5* seconds, rotate **90** degrees to the *right*, and *stop* for **0.5** second.
5. The car will move to create the *short* side of the rectangle at **30**% of its speed for **2** seconds.
6. After finishing the *shortest* side the car will *stop* for **0.5** seconds, *rotate* **90** degrees to the *right*, and *stop* for **0.5** second.
7. Steps 3 to 6 will be repeated infinitely until you press the *stop* button (**PB2**)
8. **PB2** acts as a sudden break, and it has the highest priority.

2. High Level Design

2.1. System Architecture

2.1.1. Definition

Layered Architecture (*Figure 1*) describes an architectural pattern composed of several separate horizontal layers that function together as a single unit of software.

Microcontroller Abstraction Layer (*MCAL*) is a software module that directly accesses on-chip MCU peripheral modules and external devices that are mapped to memory, and makes the upper software layer independent of the MCU.

Hardware Abstraction Layer (*HAL*) is a layer of programming that allows a computer OS to interact with a hardware device at a general or abstract level rather than at a detailed hardware level.

2.1.2. Layered Architecture

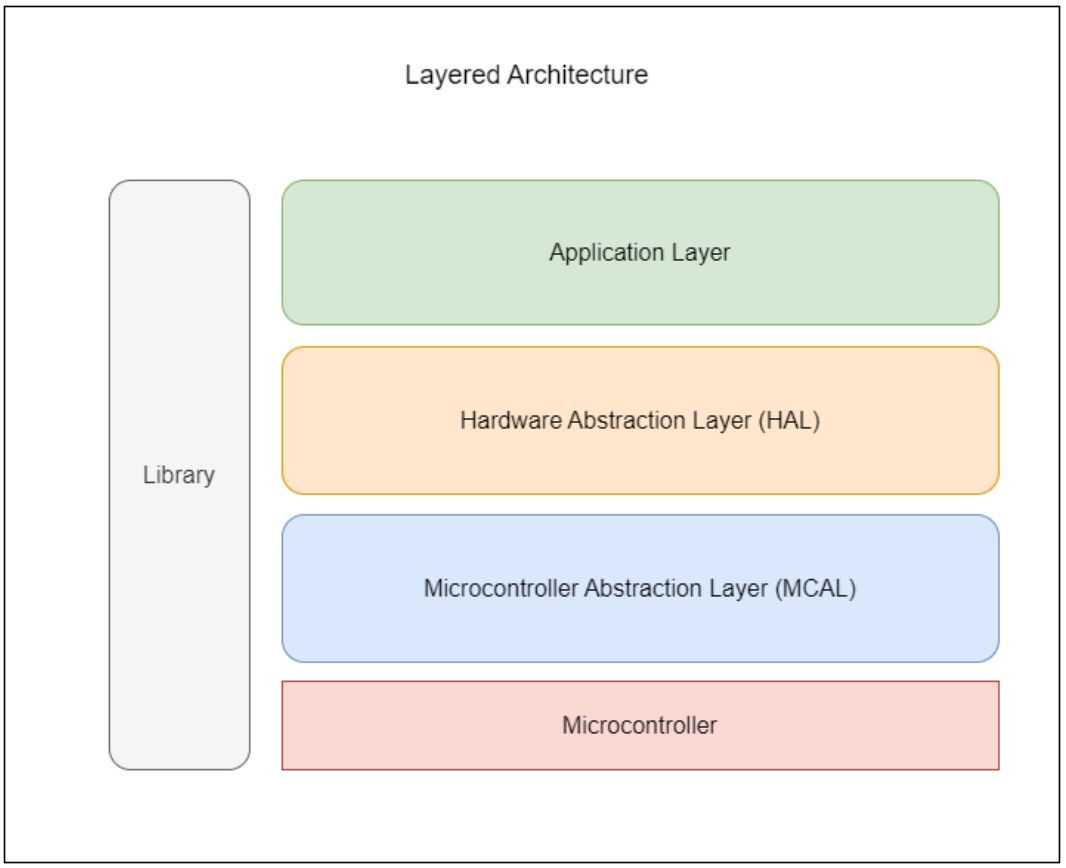


Figure 1. Layered Architecture Design

2.2. Block Diagram

2.2.1. Definition

A *Block Diagram* (*Figure 2*) is a specialized flowchart used to visualize systems and how they interact.

Block Diagrams give you a high-level overview of a system so you can account for major system components, visualize inputs and outputs, and understand working relationships within the system.

2.2.2. Design

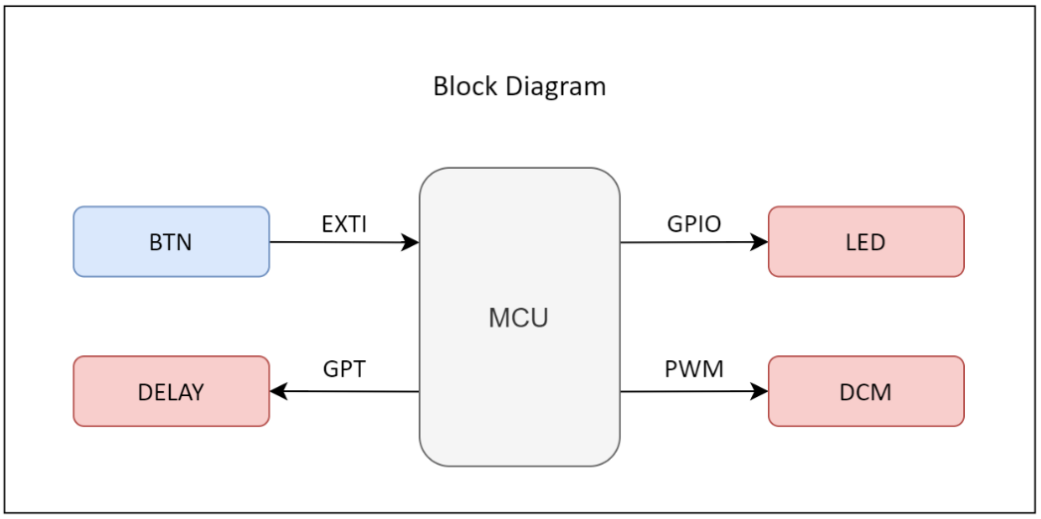


Figure 2. Block Diagram Design

System Input: Blue | *System Output*: Red

2.3. System Modules

2.3.1. Definition

A *Module* is a distinct assembly of components that can be easily added, removed or replaced in a larger system. Generally, a *Module* is not functional on its own.

In computer hardware, a *Module* is a component that is designed for easy replacement.

2.3.2. Design

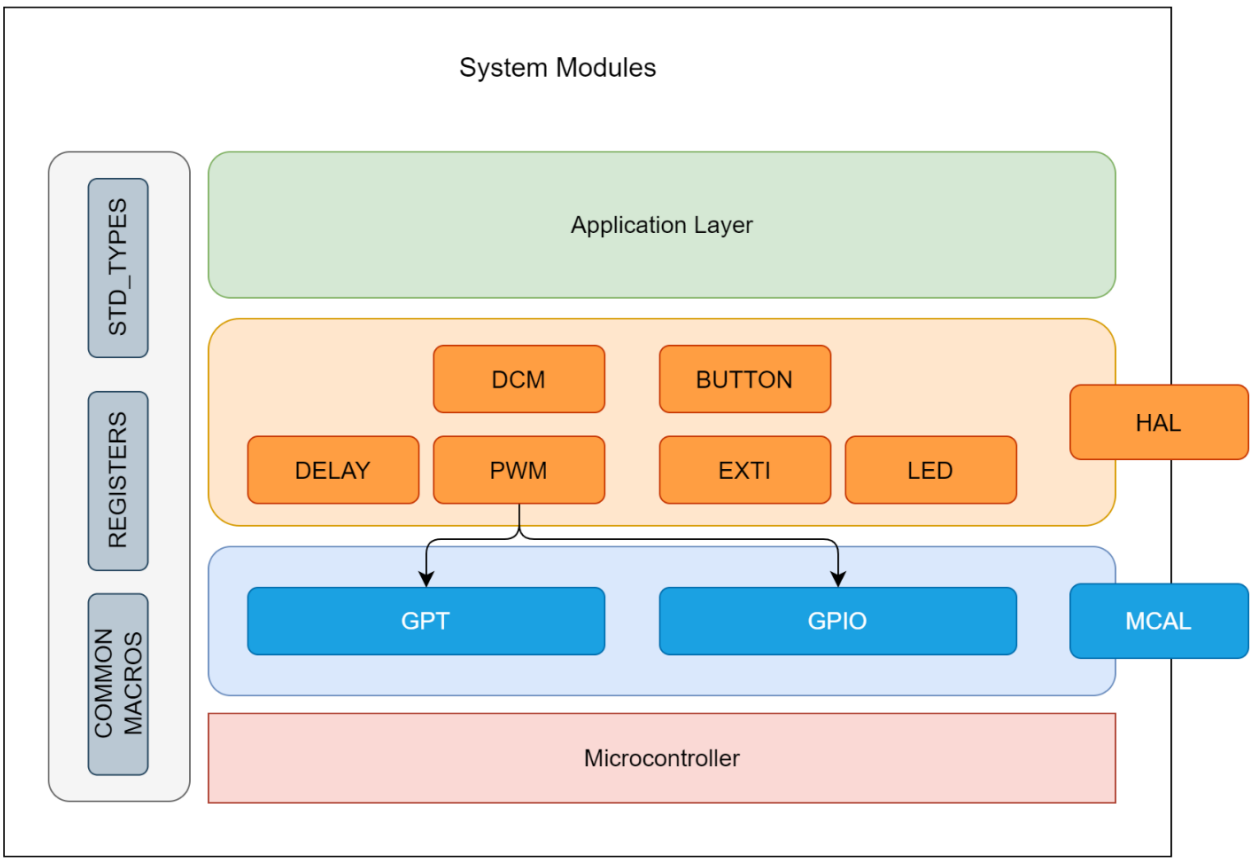


Figure 3. System Modules Design

2.4. Modules Description

2.4.1. GPIO Module

A *GPIO* (General Purpose Input/Output) driver is a fundamental component in microcontroller projects, providing the ability to interface with external devices and control digital signals. It serves as an interface between the microcontroller and the outside world, enabling the manipulation of input and output signals for various applications. The *GPIO* driver in a microcontroller project facilitates the control and configuration of the *GPIO* pins available on the microcontroller. These pins can be individually programmed as inputs or outputs to interface with external devices such as sensors, actuators, or communication modules.

2.4.2. GPT Module

The *GPT* (General-Purpose Timer) peripheral is a versatile timer module available in microcontrollers that provides precise timing and event counting capabilities. It is commonly used in embedded systems for a wide range of applications, including timing measurements, event capturing, and generating periodic or one-shot interrupts. The *GPT* peripheral typically consists of multiple timer modules, each with its own set of registers and functionality. These modules can operate independently or be synchronized to perform coordinated timing operations.

2.4.3. LED Module

LED (Light Emitting Diode) is responsible for controlling the state of the systems' LEDs.
It provides a set of APIs to turn off, to turn on, or to toggle the state of each led.

2.4.4. BUTTON Module

In most of the embedded electronic projects you may want to use a *BUTTON* (Push Button) switch to give user inputs to the microcontroller. Push Button is basically a small controlling device that is pressed to operate any electrical device.

2.4.5. PWM Module

The *PWM* (Pulse Width Modulation) peripheral is a widely used feature in microcontrollers that enables precise control of the output signal's pulse width. *PWM* is commonly employed in various applications such as motor control, LED dimming, audio synthesis, and power regulation.

2.4.6. DCM Module

A *DCM* (DC Motor) driver, also known as a motor controller, is an electronic device that is used to control the speed and direction of a *DCM*. The primary function of a *DCM* driver is to regulate the power supplied to the motor. It takes input signals from a microcontroller or other control circuitry and converts them into suitable voltage and current levels to drive the motor effectively. The *DCM* driver ensures that the motor receives the necessary power and protection, preventing damage due to excessive current or voltage fluctuations.

2.4.7. DELAY Module

A *DELAY* driver is a fundamental component in many microcontroller projects. It allows for precise timing control, delays, and synchronization within the software code. This driver is particularly useful when dealing with time-sensitive operations, such as controlling sensor readings, generating accurate timing intervals, or creating software delays.

2.4.8. EXTI Module

An *EXTI* (External Interrupt) driver allows you to respond to external events or signals by interrupting the normal flow of program execution and executing a specific interrupt service routine (ISR). This enables you to handle time-sensitive or critical events in a timely and efficient manner. This can be useful for applications that require quick reactions to specific events, such as button presses, sensor readings, or other time-critical inputs.

2.5. Drivers' Documentation (APIs)

2.5.1 Definition

An *API* is an *Application Programming Interface* that defines a set of *routines*, *protocols* and *tools* for creating an application. An *API* defines the high level interface of the behavior and capabilities of the component and its inputs and outputs.

An *API* should be created so that it is generic and implementation independent. This allows for the API to be used in multiple applications with changes only to the implementation of the API and not the general interface or behavior.

2.5.2. MCAL APIs

2.5.2.1. GPIO Driver APIs

```
| Name: GPIO_init
| Input: str_gpio_config
| Output: enu_gpio_error_state_t
| Description: Function to initialize GPIO specific pin.
|
enu_gpio_error_state_t GPIO_init (const str_gpio_config_t *str_gpio_config)

| Name: GPIO_digitalWrite
| Input: enu_gpio_port_id, enu_gpio_pin_id, enu_gpio_pin_level
| Output: enu_gpio_error_state_t
| Description: Function to set Pin value.
|
enu_gpio_error_state_t GPIO_digitalWrite (enu_gpio_port_id_t enu_gpio_port_id,
enu_gpio_pin_id_t enu_gpio_pin_id, enu_gpio_pin_level_t enu_gpio_pin_level)

| Name: GPIO_digitalRead
| Input: enu_gpio_port_id, enu_gpio_pin_id, P_value
| Output: enu_gpio_error_state_t
| Description: Function to get Pin value.
|
enu_gpio_error_state_t GPIO_digitalRead (enu_gpio_port_id_t enu_gpio_port_id,
enu_gpio_pin_id_t enu_gpio_pin_id, uint8 *P_value)
```

```
| Name: GPIO_togglePin
| Input: enu_gpio_port_id and enu_gpio_pin_id
| Output: enu_gpio_error_state_t
| Description: Function to toggle Pin value.
|
enu_gpio_error_state_t GPIO_togglePin (enu_gpio_port_id_t enu_gpio_port_id,
enu_gpio_pin_id_t enu_gpio_pin_id)

| Name: GPIO_interruptEnable
| Input: enu_interrupt_edge, enu_gpio_port_id and enu_gpio_pin_id
| Output: enu_gpio_error_state_t
| Description: Function to set and enable GPIO pin to interrupt request
|
enu_gpio_error_state_t GPIO_interruptEnable (enu_interrupt_edge_t
enu_interrupt_edge, enu_gpio_port_id_t enu_gpio_port_id, enu_gpio_pin_id_t
enu_gpio_pin_id)

| Name: GPIO_interruptEnable
| Input: enu_gpio_port_id and enu_gpio_pin_id
| Output: enu_gpio_error_state_t
| Description: Function to disable GPIO pin interrupt
|
enu_gpio_error_state_t GPIO_interruptDisable (enu_gpio_port_id_t
enu_gpio_port_id, enu_gpio_pin_id_t enu_gpio_pin_id)

| Name: GPIO_Setcallback
| Input: enu_gpio_port_id,enu_gpio_pin_id,Fptr
| Output: enu_gpio_error_state_t
| Description: Function to set callback for interrupt
|
enu_gpio_error_state_t GPIO_Setcallback (void(*Fptr)(void), enu_gpio_port_id_t
enu_gpio_port_id, enu_gpio_pin_id_t enu_gpio_pin_id)
```

2.5.2.2. GPT Driver APIs

```
| Name: gpt_Init
| Input: str_gpt_config
| Output: enu_timer_error_t
| Description: Function to initialize GPT mode and its preload value
|
enu_timer_error_t gpt_Init (str_gpt_config_t *str_gpt_config)

| Name: gpt_startTimer
| Input: enu_timer_id,u32_time and enu_tick_unit
| Output: enu_timer_error_t
| Description: Function to start timer to count
|
enu_timer_error_t gpt_startTimer (enu_timer_id_t enu_timer_id, uint32
u32_time, enu_tick_unit_t enu_tick_unit)

| Name: gpt_stopTimer
| Input: enu_timer_id
| Output: enu_timer_error_t
| Description: Function to stop the timer
|
enu_timer_error_t gpt_stopTimer (enu_timer_id_t enu_timer_id)

| Name: gpt_enable_notification
| Input: enu_timer_id
| Output: enu_timer_error_t
| Description: Function to enable timer interrupt
|
enu_timer_error_t gpt_enable_notification (enu_timer_id_t enu_timer_id)

| Name: gpt_disable_notification
| Input: enu_timer_id
| Output: enu_timer_error_t
| Description: Function to disable timer interrupt
|
enu_timer_error_t gpt_disable_notification (enu_timer_id_t enu_timer_id)
```

2.5.3. HAL APIs

2.5.3.1. LED Driver APIs

```
| Name: LED_init
| Input: enu_gpio_port_id,enu_gpio_pin_id
| Output: enu_error_state_t
| Description: Function to initialize LED
|
enu_error_state_t LED_init (enu_gpio_port_id_t enu_gpio_port_id,
enu_gpio_pin_id_t enu_gpio_pin_id)

| Name: LED_digitalWrite
| Input: enu_gpio_port_id,enu_gpio_pin_id and enu_gpio_pin_level
| Output: enu_error_state_t
| Description: Function to set LED on or off
|
enu_error_state_t LED_digitalWrite (enu_gpio_port_id_t enu_gpio_port_id,
enu_gpio_pin_id_t enu_gpio_pin_id, enu_gpio_pin_level_t enu_gpio_pin_level)
```

2.5.3.2. BUTTON Driver APIs

```
| Name: BUTTON_init
| Input: enu_gpio_port_id,enu_gpio_pin_id
| Output: enu_error_state_t
| Description: Function to initialize BUTTON
|
enu_error_state_t BUTTON_init (enu_gpio_port_id_t enu_gpio_port_id,
enu_gpio_pin_id_t enu_gpio_pin_id)

| Name: BUTTON_digitalRead
| Input: enu_gpio_port_id,enu_gpio_pin_id and enu_gpio_pin_level
| Output: enu_error_state_t
| Description: Function to read button value high or low
|
enu_error_state_t BUTTON_digitalRead (enu_gpio_port_id_t enu_gpio_port_id,
enu_gpio_pin_id_t enu_gpio_pin_id, uint8 *P_value)
```

2.5.3.3. PWM Driver APIs

```
| Name: PWM_Init
| Input: enu_gpio_port_id,enu_gpio_pin_id,enu_timer_id
| Output: enu_pwm_error_t
| Description: Function to initialize PWM to specific pin
|
enu_pwm_error_t PWM_Init (enu_gpio_port_id_t enu_gpio_port_id,
enu_gpio_pin_id_t enu_gpio_pin_id, enu_timer_id_t enu_timer_id)

| Name: PWM_start
| Input: enu_timer_id,u32_periodic_time,enu_tick_unit and u8_duty_cycle
| Output: enu_pwm_error_t
| Description: Function used to start generate PWM signal
|
enu_pwm_error_t PWM_start (enu_timer_id_t enu_timer_id, uint32
u32_periodic_time, enu_tick_unit_t enu_tick_unit, uint8 u8_duty_cycle)

| Name: PWM_stop
| Input: enu_timer_id
| Output: enu_pwm_error_t
| Description: Function used to stop generate PWM signal
|
enu_pwm_error_t PWM_stop (enu_timer_id_t enu_timer_id)
```

2.5.3.4. DCM Driver APIs

```
| Name: DCM_initialization
| Input: Pointer to st DCMConfig
| Output: en Error or No Error
| Description: Function to Initialize DCM peripheral.
|
enu_error_state_t DCM_initialization (DCM_ST_CONFIG *pst_a_DCMConfig)

| Name: DCM_controlDCM
| Input: Pointer to st DCMConfig and u8 ControlMode
| Output: en Error or No Error
| Description: Function Control DCM with one of DCM Modes.
|
enu_error_state_t DCM_controlDCM (DCM_ST_CONFIG *pst_a_DCMConfig, uint8
u8_a_controlMode)

| Name: DCM_controlDCMSpeed
| Input: u8 SpeedPercentage
| Output: en Error or No Error
| Description: Function Control DCM Speed.
|
enu_error_state_t DCM_controlDCMSpeed (uint8 u8_a_speedPercentage)
```

2.5.3.5. DELAY Driver APIs

```
| Name: delay_us
| Input: enu_timer_id and u32_time
| Output: enu_delay_error_t
| Description: Function to set delay in microsecond
|
enu_delay_error_t delay_us (enu_timer_id_t enu_timer_id, uint32 u32_time)

| Name: delay_ms
| Input: enu_timer_id and u32_time
| Output: enu_delay_error_t
| Description: Function to set delay in millisecond
|
enu_delay_error_t delay_ms (enu_timer_id_t enu_timer_id, uint32 u32_time)

| Name: delay_sec
| Input: enu_timer_id and u32_time
| Output: enu_delay_error_t
| Description: Function to set delay in second
|
enu_delay_error_t delay_sec (enu_timer_id_t enu_timer_id, uint32 u32_time)
```

2.5.3.6. EXTI Driver APIs

```
| Name: INT_init
| Input: enu_gpio_port_id ,enu_gpio_pin_id,enu_interrupt_edge and Fptr_callback
| Output: enu_int_error_t
| Description: Function used to initialize GPIO interrupt
|
enu_int_error_t INT_init (enu_gpio_port_id_t enu_gpio_port_id,
enu_gpio_pin_id_t enu_gpio_pin_id, enu_interrupt_edge_t enu_interrupt_edge,
void (*Fptr_callback)(void))

| Name: INT_Deinit
| Input: enu_gpio_port_id and enu_gpio_pin_id
| Output: enu_int_error_t
| Description: Function used to Deinitialize GPIO interrupt
|
enu_int_error_t INT_Deinit (enu_gpio_port_id_t enu_gpio_port_id,
enu_gpio_pin_id_t enu_gpio_pin_id)

| Name: INT_Deinit
| Input: Fptr_callback,enu_gpio_port_id and enu_gpio_pin_id
| Output: enu_int_error_t
| Description: Function used to set interrupt callback function
|
enu_int_error_t INT_setCallBack (void (*Fptr_callback)(void),
enu_gpio_port_id_t enu_gpio_port_id, enu_gpio_pin_id_t enu_gpio_pin_id)
```

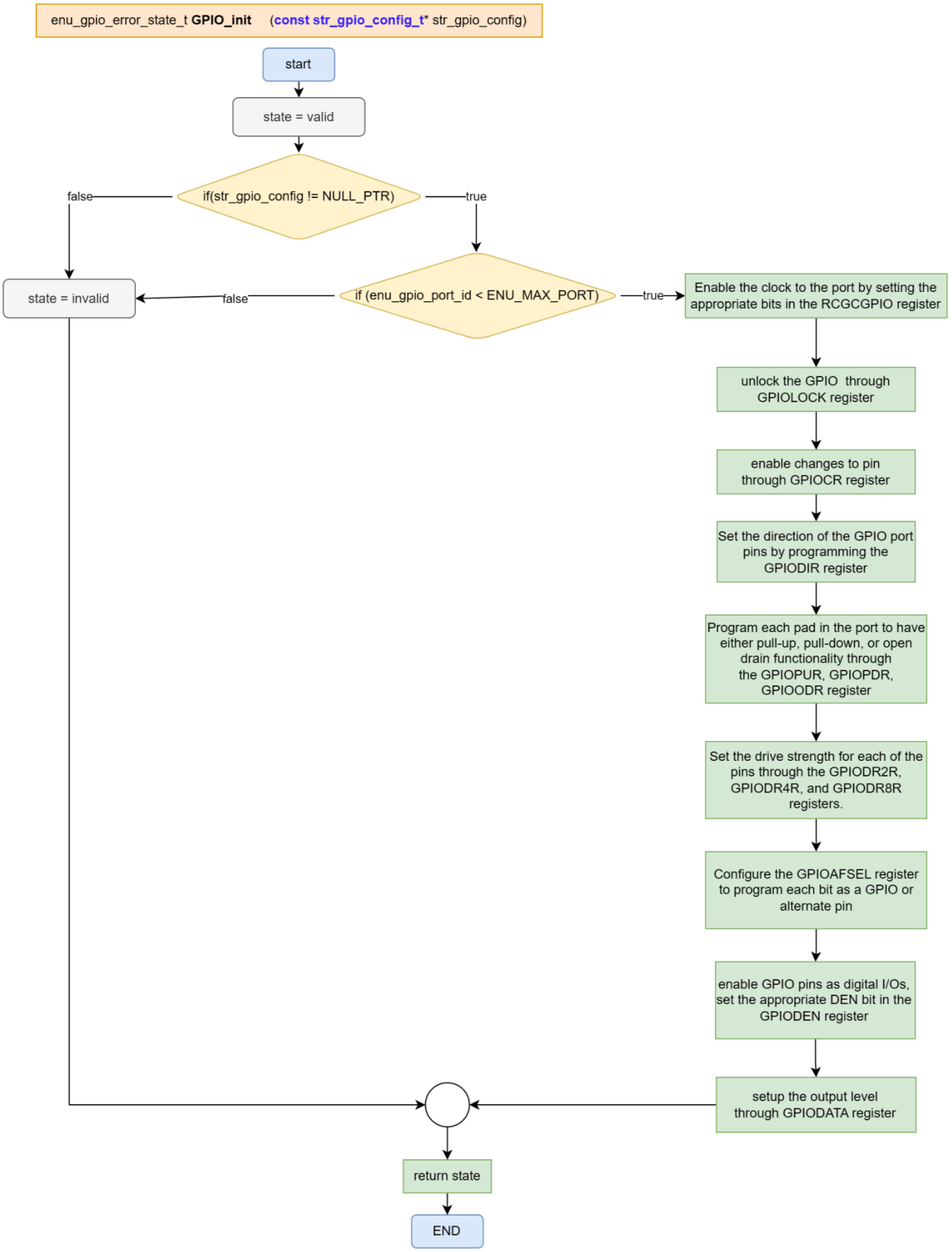
2.5.4. APP APIs

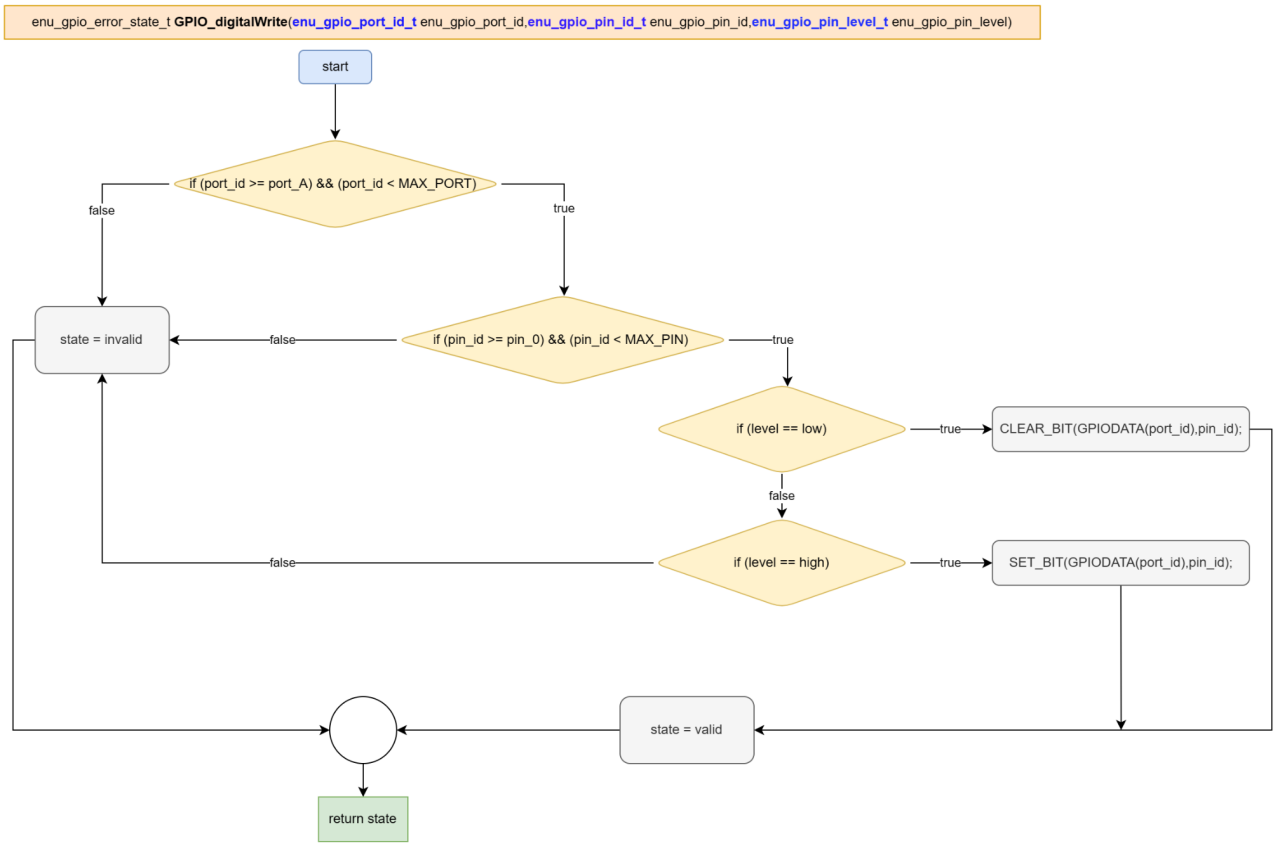
```
| Name: APP_initialization
| Input: void
| Output: void
| Description: Function to Initialize the Application.
|
void APP_initialization (void)

| Name: APP_startProgram
| Input: void
| Output: void
| Description: Function to Start the basic flow of the Application.
|
void APP_startProgram (void)

| Name: APP_stopCar
| Input: void
| Output: void
| Description: Function to Start the car.
|
void APP_startCar (void)

| Name: APP_stopCar
| Input: void
| Output: void
| Description: Function to Stop the car.
|
void APP_stopCar (void)
```

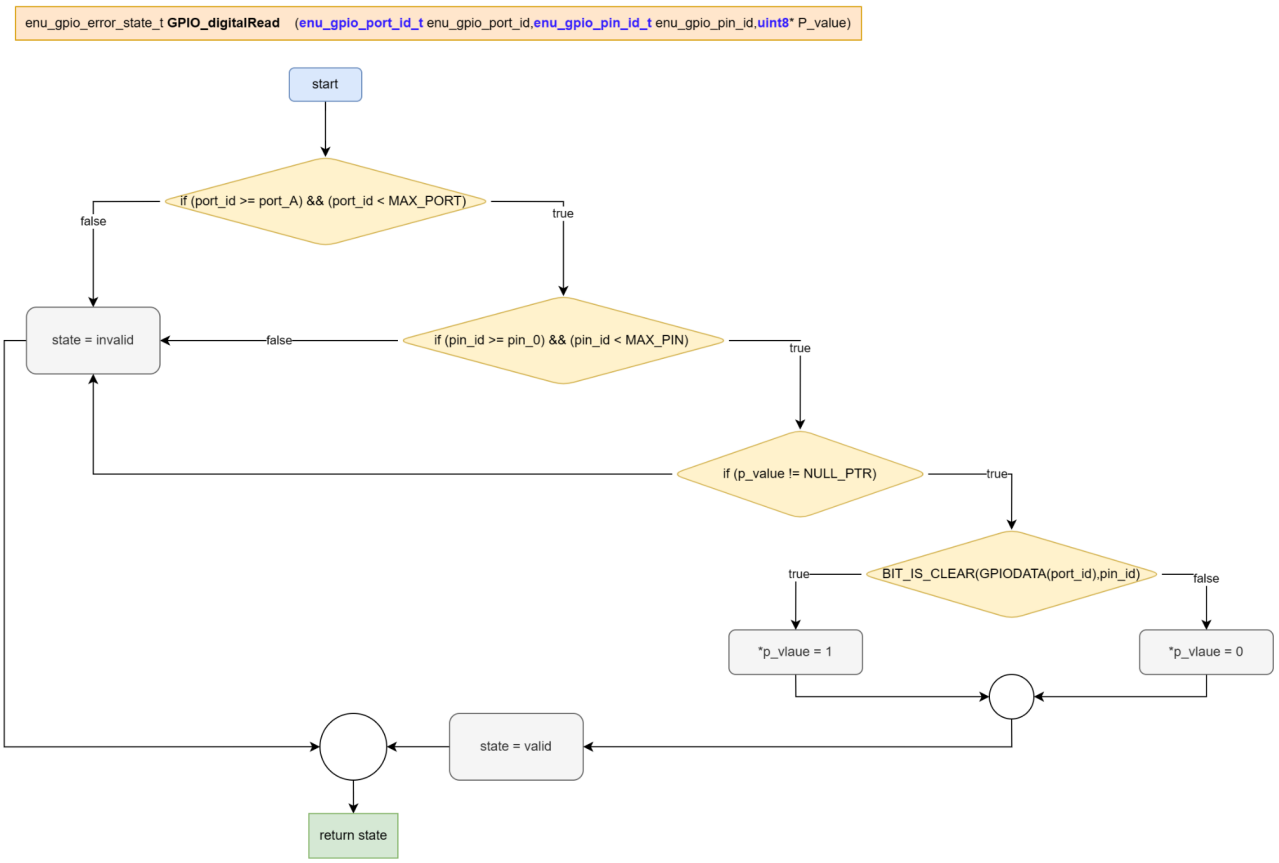
3. Low Level Design

3.1. MCAL Layer

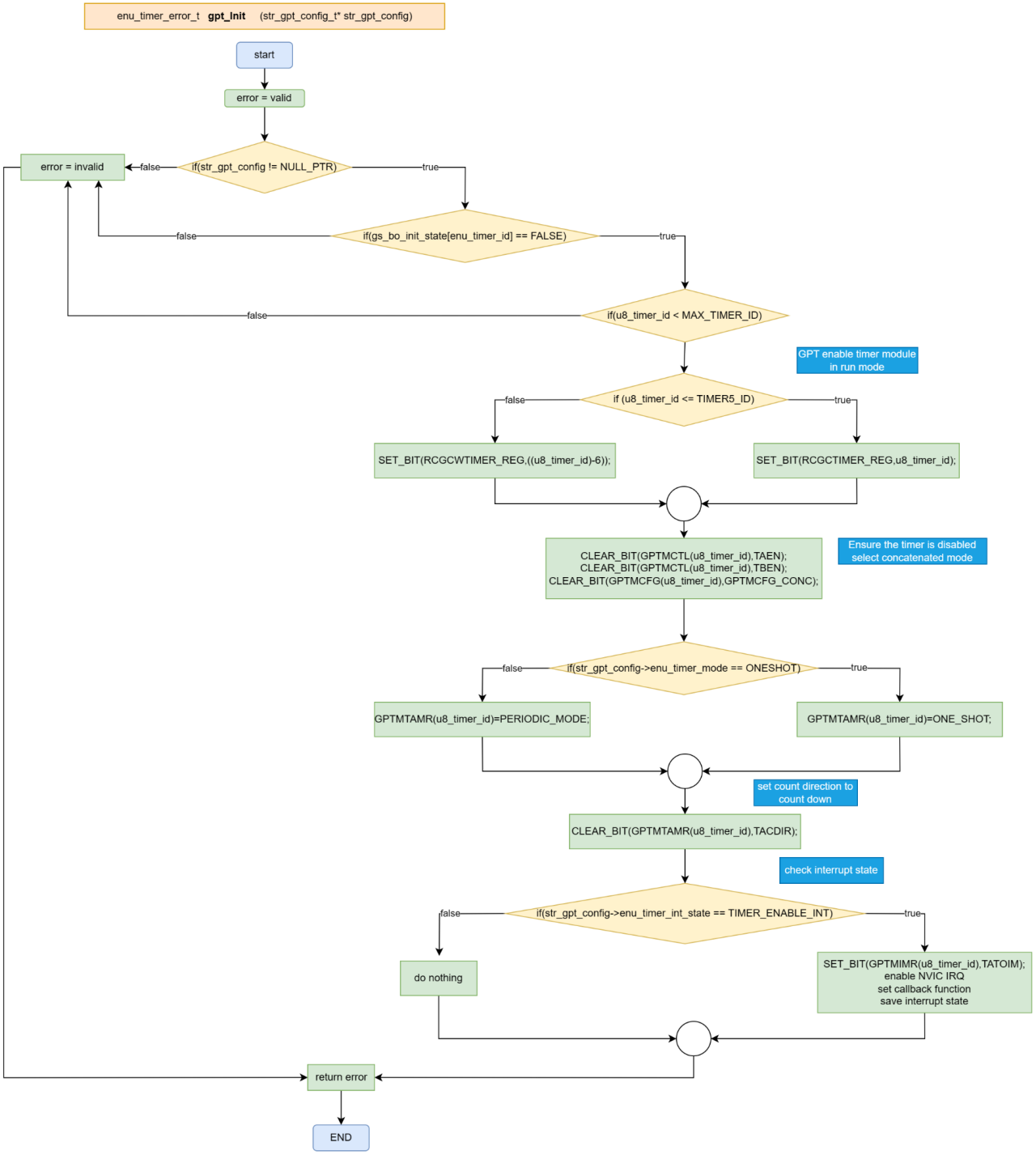
3.1.1. GPIO Module

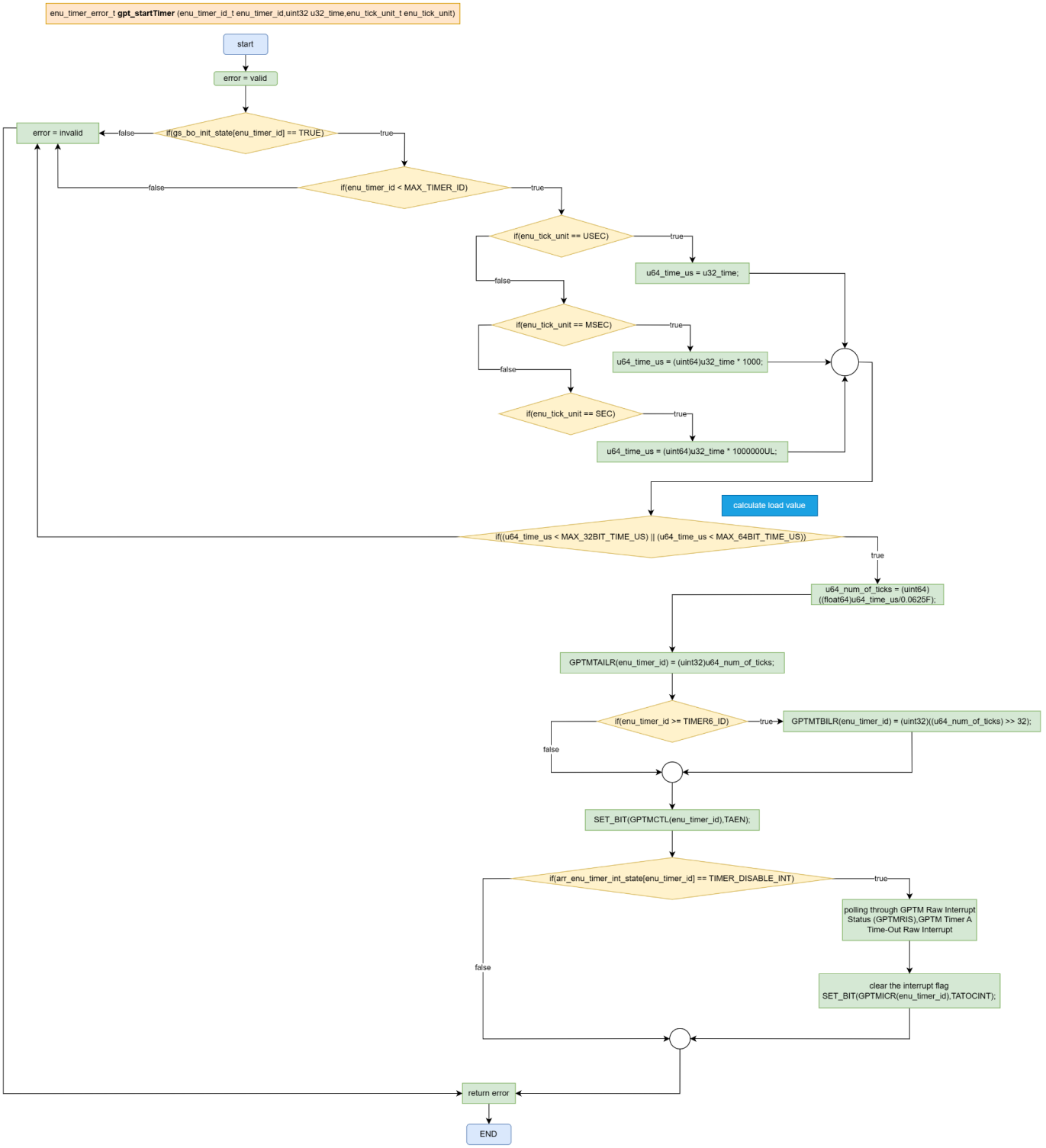


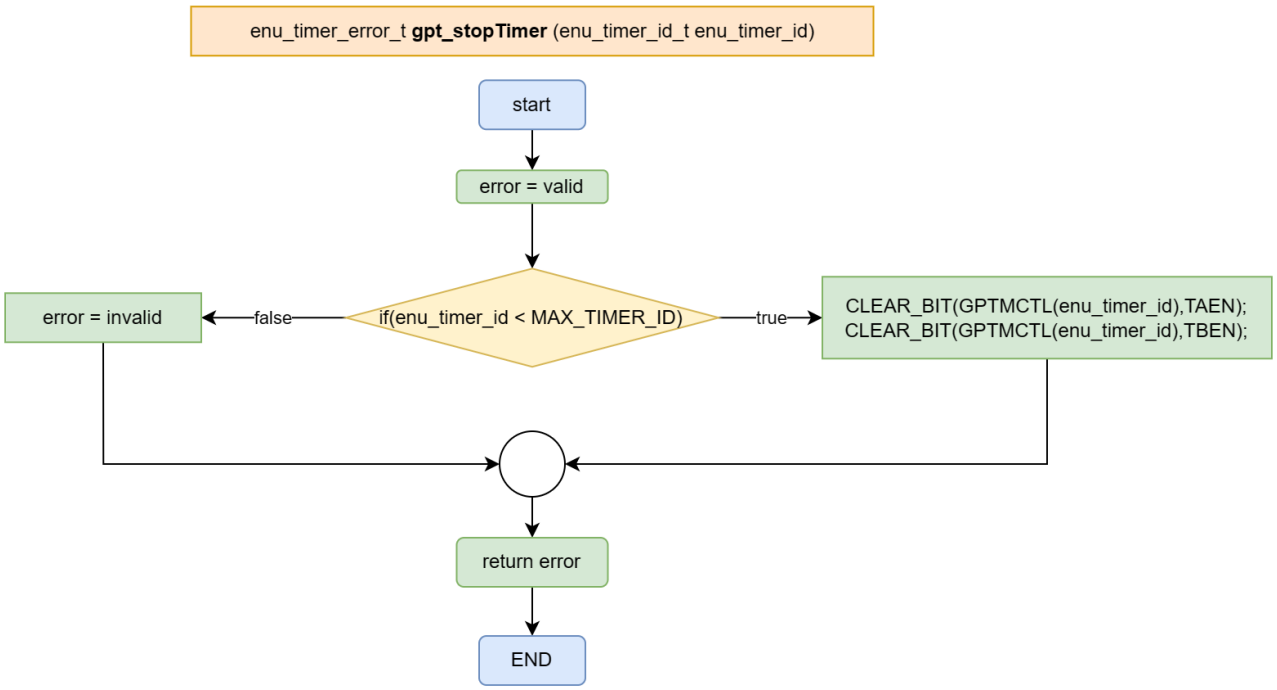

enu_gpio_error_state_t GPIO_digitalWrite(enu_gpio_port_id_t enu_gpio_port_id,enu_gpio_pin_id_t enu_gpio_pin_id,enu_gpio_pin_level_t enu_gpio_pin_level)
start
if (port_id >= port_A) && (port_id < MAX_PORT)
false
true
state = invalid
false
if (pin_id >= pin_0) && (pin_id < MAX_PIN)
true
if (level == low)
true
CLEAR_BIT(GPIODATA(port_id),pin_id);
false
if (level == high)
false
true
SET_BIT(GPIODATA(port_id),pin_id);
state = valid
return state


enu_gpio_error_state_t GPIO_digitalRead (enu_gpio_port_id_t enu_gpio_port_id,enu_gpio_pin_id_t enu_gpio_pin_id,uint8* P_value)
start
if (port_id >= port_A) && (port_id < MAX_PORT)
false
true
state = invalid
false
if (pin_id >= pin_0) && (pin_id < MAX_PIN)
true
if (p_value != NULL_PTR)
true
true
BIT_IS_CLEAR(GPIODATA(port_id),pin_id)
false
*p_vlaue = 1
*p_vlaue = 0
state = valid
return state

3.1.2. GPT Module



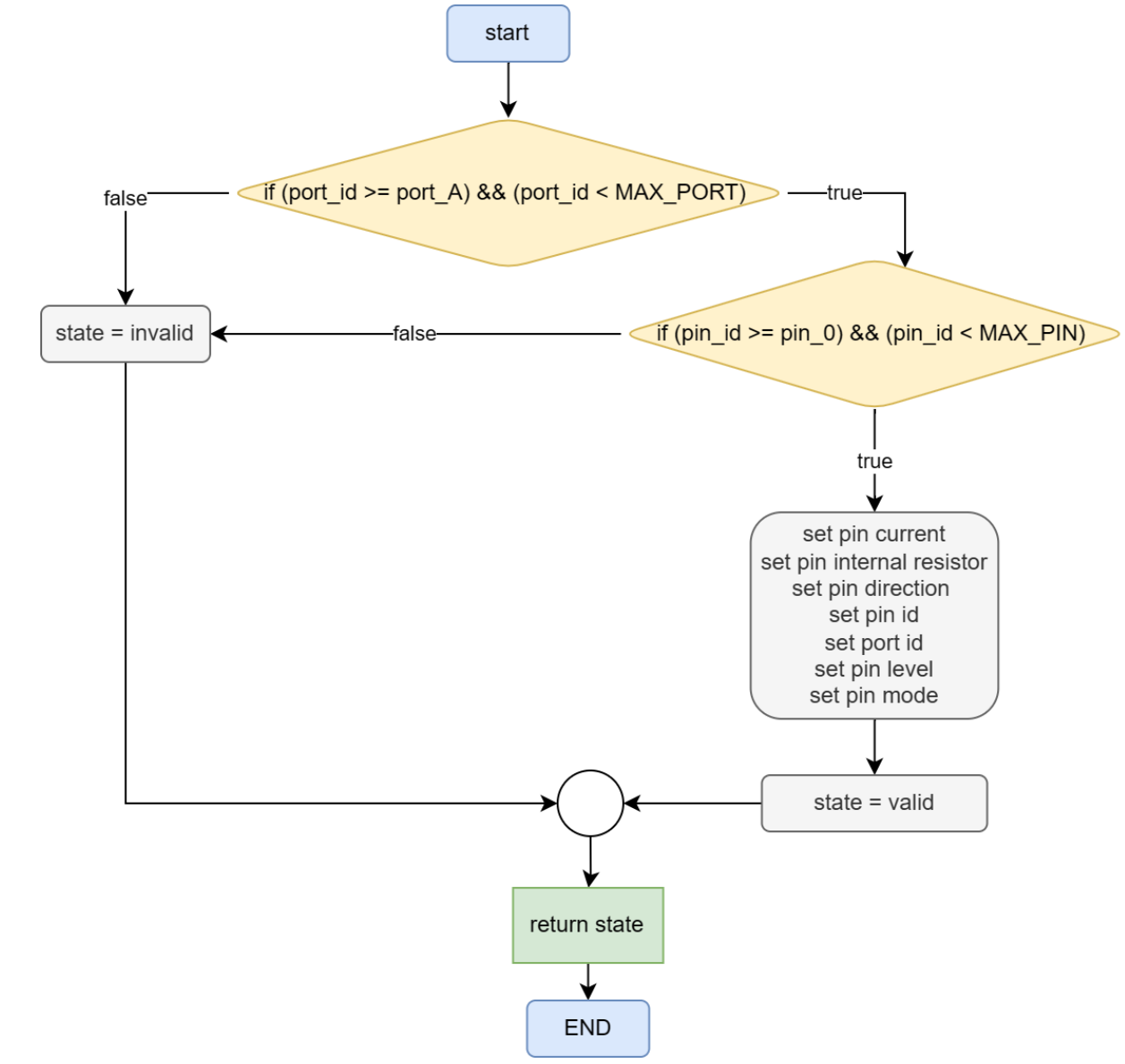

enu_timer_error_t gpt_startTimer (enu_timer_id_t enu_timer_id,uint32 u32_time,enu_tick_unit_t enu_tick_unit)
start
error = valid
if(gs_bo_init_state[enu_timer_id] == TRUE)
false
error = invalid
true
if(enu_timer_id < MAX_TIMER_ID)
false
true
if(enu_tick_unit == USEC)
true
u64_time_us = u32_time;
false
if(enu_tick_unit == MSEC)
true
u64_time_us = (uint64)u32_time * 1000;
false
if(enu_tick_unit == SEC)
true
u64_time_us = (uint64)u32_time * 1000000UL;
calculate load value
if((u64_time_us < MAX_32BIT_TIME_US) || (u64_time_us < MAX_64BIT_TIME_US))
true
u64_num_of_ticks = (uint64)((float64)u64_time_us/0.0625F);
GPTMTAILR(enu_timer_id) = (uint32)u64_num_of_ticks;
if(enu_timer_id >= TIMER6_ID)
true
GPTMTBILR(enu_timer_id) = (uint32)((u64_num_of_ticks) >> 32);
false
SET_BIT(GPTMCTL(enu_timer_id),TAEN);
if(arr_enu_timer_int_state[enu_timer_id] == TIMER_DISABLE_INT)
true
polling through GPTM Raw Interrupt Status (GPTMRIS),GPTM Timer A Time-Out Raw Interrupt
clear the interrupt flag SET_BIT(GPTMICR(enu_timer_id),TATOCINT);
false
return error
END


enu_timer_error_t **gpt_stopTimer** (enu_timer_id_t enu_timer_id)
start
error = valid
if(enu_timer_id < MAX_TIMER_ID)
false
error = invalid
true
CLEAR_BIT(GPTMCTL(enu_timer_id),TAEN);
CLEAR_BIT(GPTMCTL(enu_timer_id),TBEN);
return error
END

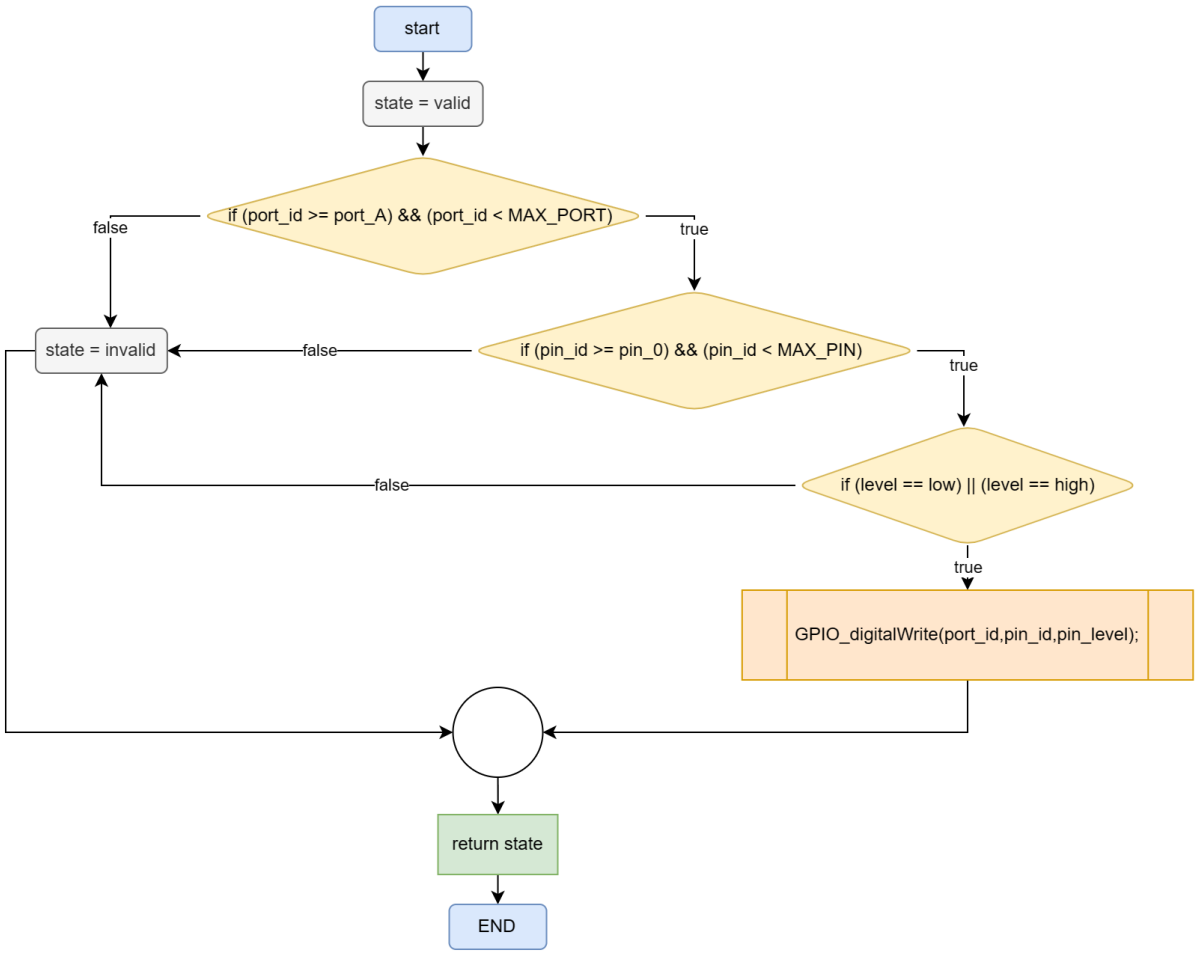
3.2. HAL Layer

3.2.1. LED Module

enu_error_state_t **LED_init**(enu_gpio_port_id_t **enu_gpio_port_id**,enu_gpio_pin_id_t **enu_gpio_pin_id**);

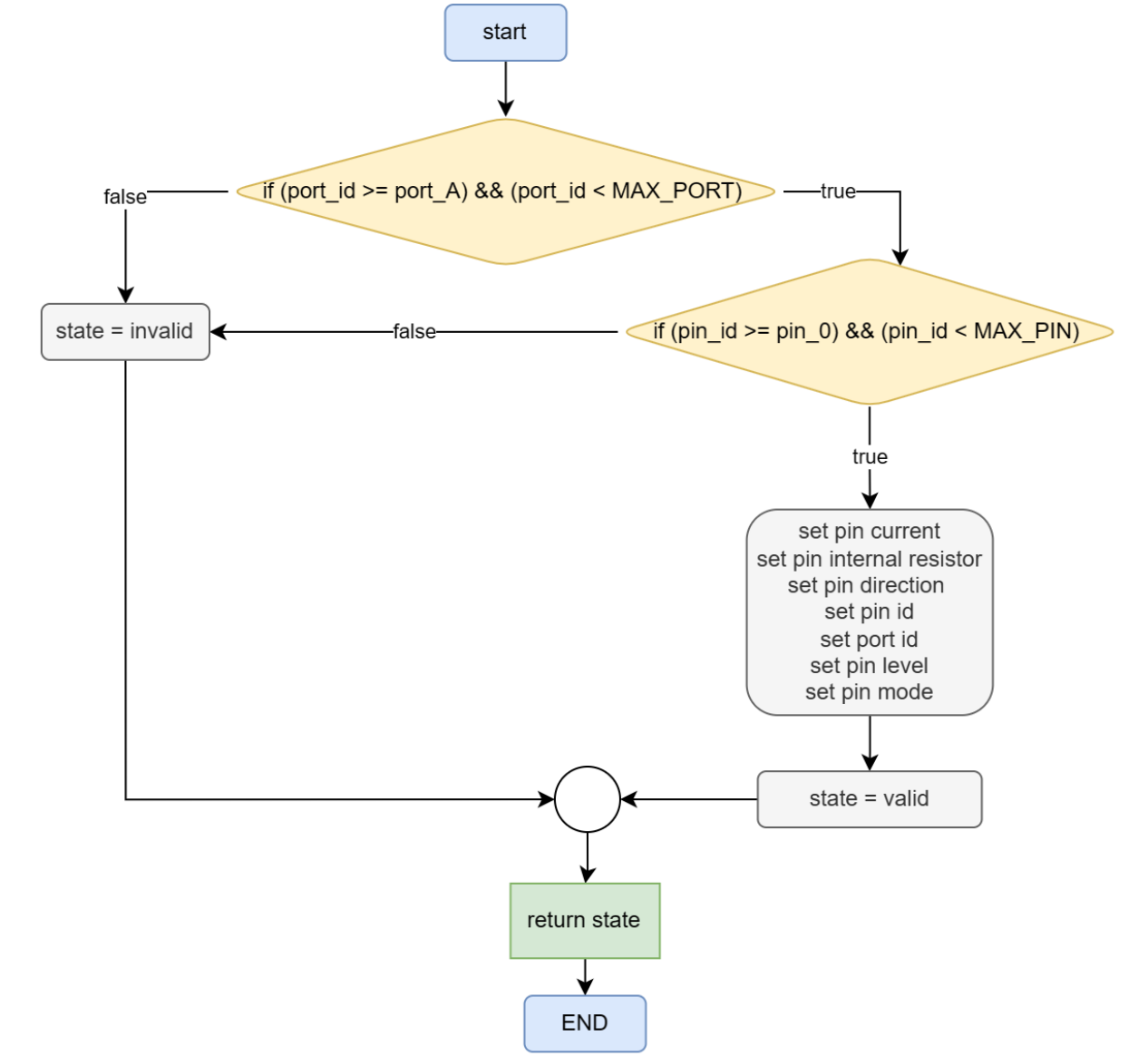


enu_error_state_t **LED_digitalWrite**(enu_gpio_port_id_t **enu_gpio_port_id**,enu_gpio_pin_id_t **enu_gpio_pin_id**,enu_gpio_pin_level_t **enu_gpio_pin_level**);

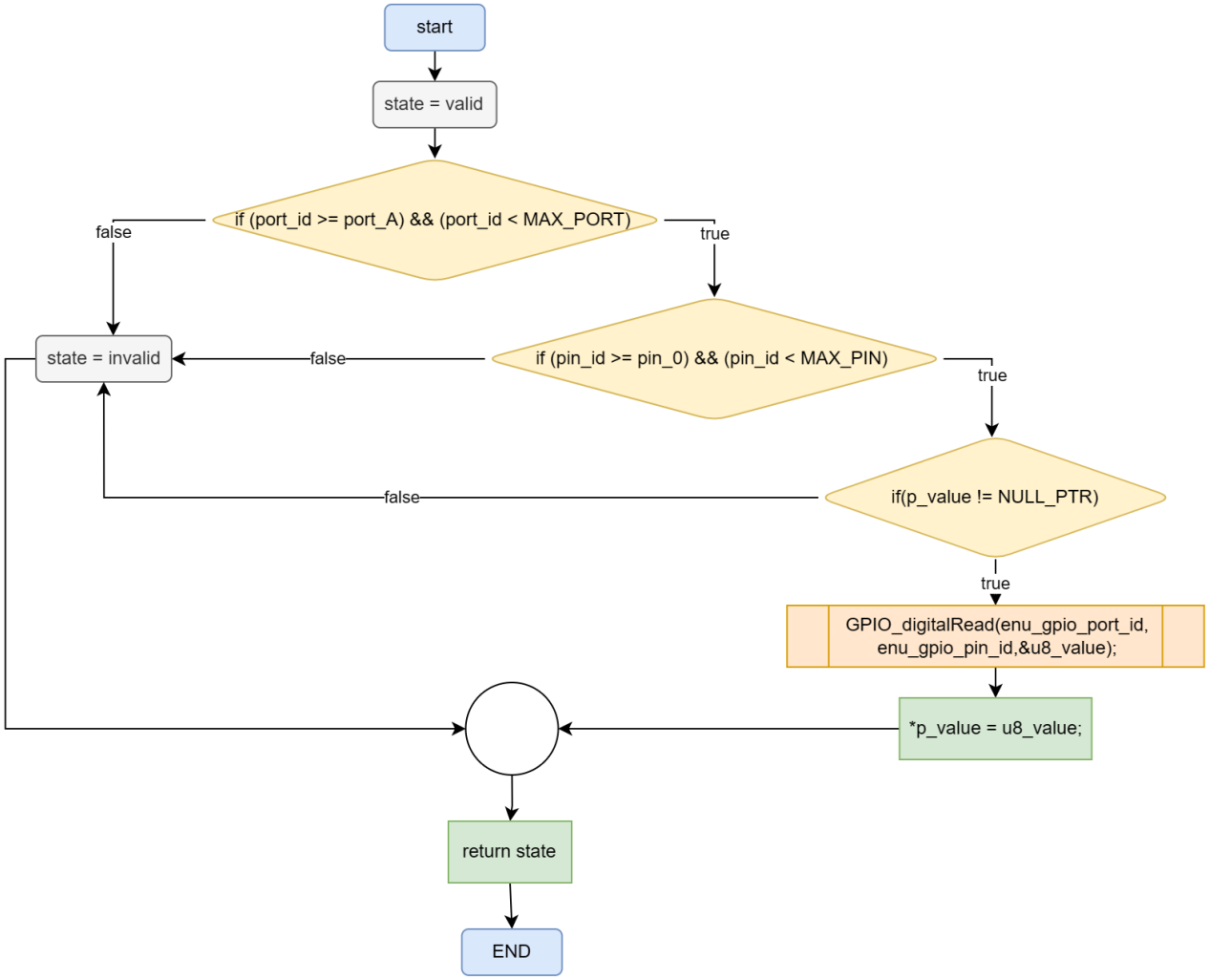


3.2.2. BUTTON Module

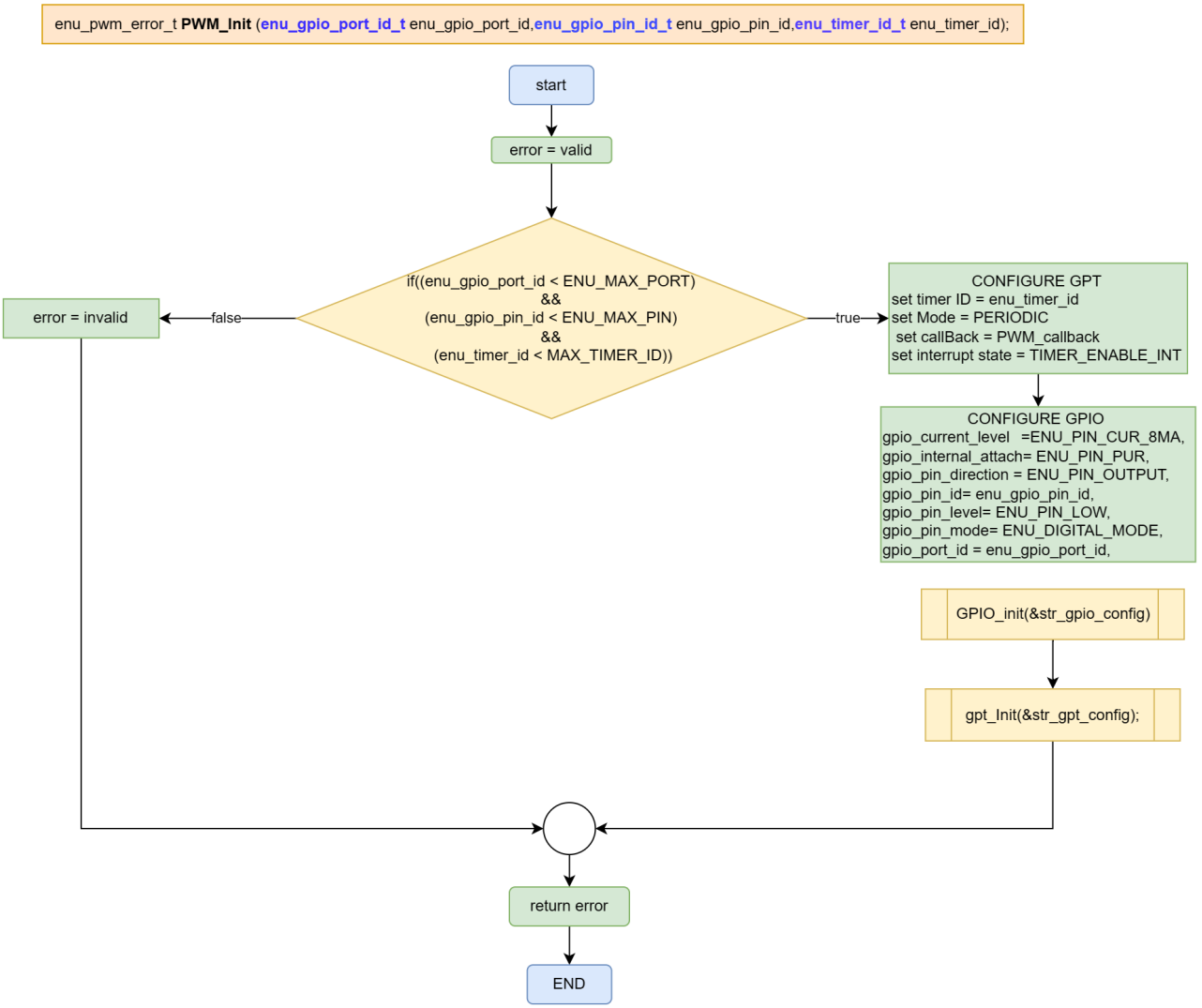
enu_error_state_t **BUTTON _init**(enu_gpio_port_id_t **enu_gpio_port_id**,enu_gpio_pin_id_t **enu_gpio_pin_id**);

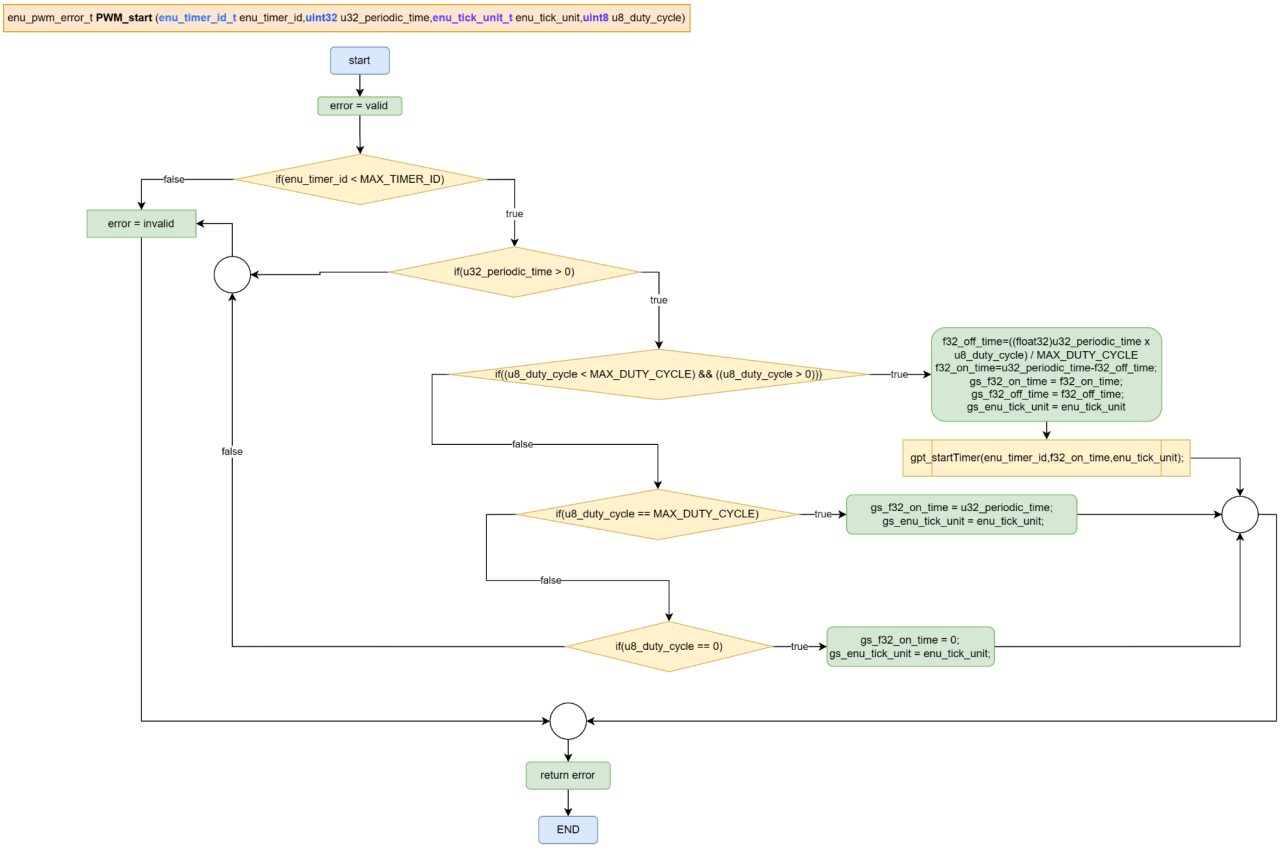


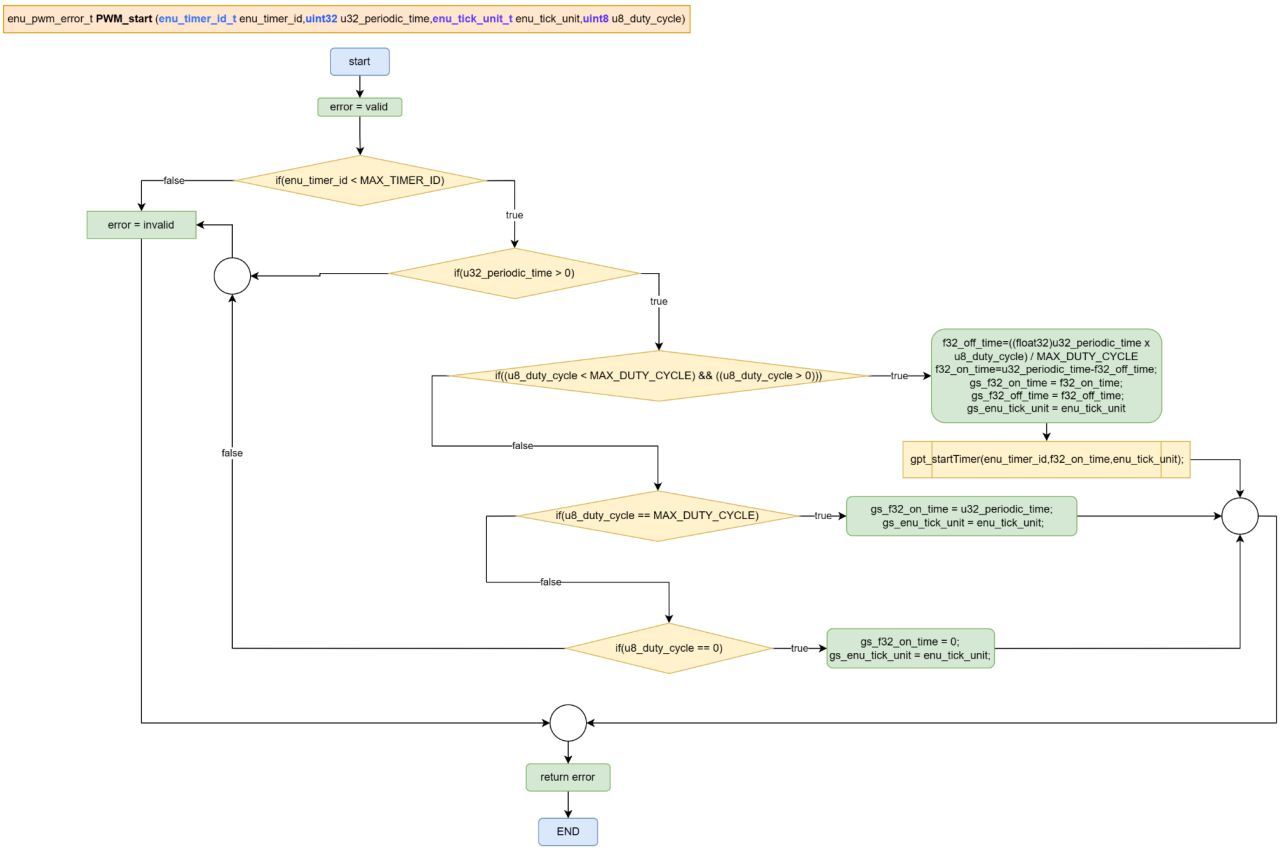
enu_error_state_t **BUTTON_digitalRead**(enu_gpio_port_id_t **enu_gpio_port_id**,enu_gpio_pin_id_t **enu_gpio_pin_id**,uint8* **p_value**);



3.2.3. PWM Module

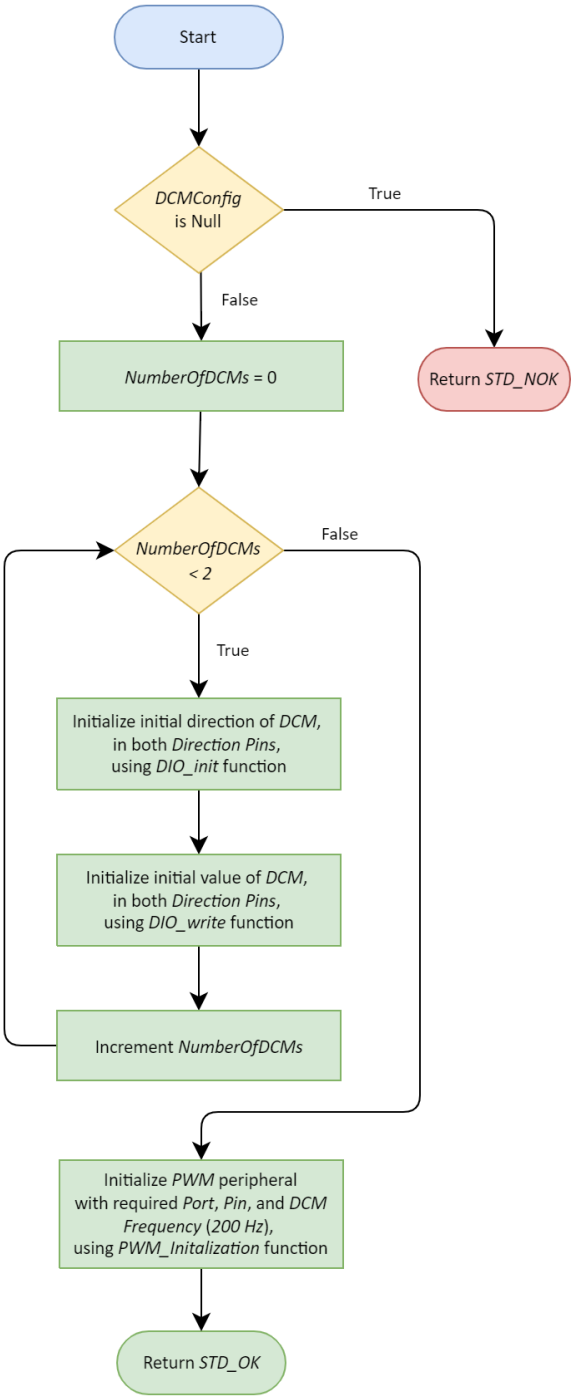



enu_pwm_error_t PWM_start (enu_timer_id_t enu_timer_id,uint32 u32_periodic_time,enu_tick_unit_t enu_tick_unit,uint8 u8_duty_cycle)
start
error = valid
if(enu_timer_id < MAX_TIMER_ID)
false
true
error = invalid
if(u32_periodic_time > 0)
true
false
if((u8_duty_cycle < MAX_DUTY_CYCLE) && ((u8_duty_cycle > 0)))
true
f32_off_time=((float32)u32_periodic_time x u8_duty_cycle) / MAX_DUTY_CYCLE
f32_on_time=u32_periodic_time-f32_off_time;
gs_f32_on_time = f32_on_time;
gs_f32_off_time = f32_off_time;
gs_enu_tick_unit = enu_tick_unit
gpt_startTimer(enu_timer_id,f32_on_time,enu_tick_unit);
false
if(u8_duty_cycle == MAX_DUTY_CYCLE)
true
gs_f32_on_time = u32_periodic_time;
gs_enu_tick_unit = enu_tick_unit;
false
if(u8_duty_cycle == 0)
true
gs_f32_on_time = 0;
gs_enu_tick_unit = enu_tick_unit;
return error
END

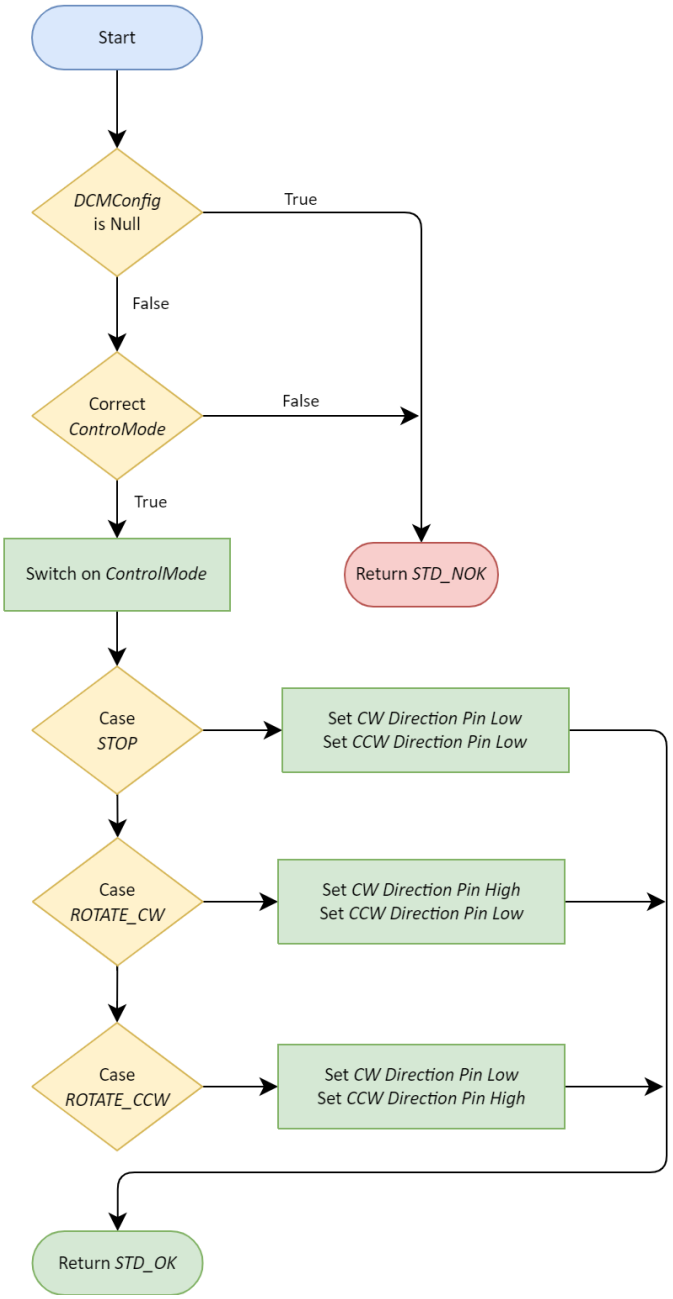

enu_pwm_error_t PWM_start (enu_timer_id_t enu_timer_id,uint32 u32_periodic_time,enu_tick_unit_t enu_tick_unit,uint8 u8_duty_cycle)
start
error = valid
if(enu_timer_id < MAX_TIMER_ID)
false
true
error = invalid
if(u32_periodic_time > 0)
true
false
if((u8_duty_cycle < MAX_DUTY_CYCLE) && ((u8_duty_cycle > 0)))
true
f32_off_time=((float32)u32_periodic_time x u8_duty_cycle) / MAX_DUTY_CYCLE
f32_on_time=u32_periodic_time-f32_off_time;
gs_f32_on_time = f32_on_time;
gs_f32_off_time = f32_off_time;
gs_enu_tick_unit = enu_tick_unit
gpt_startTimer(enu_timer_id,f32_on_time,enu_tick_unit);
false
if(u8_duty_cycle == MAX_DUTY_CYCLE)
true
gs_f32_on_time = u32_periodic_time;
gs_enu_tick_unit = enu_tick_unit;
false
if(u8_duty_cycle == 0)
true
gs_f32_on_time = 0;
gs_enu_tick_unit = enu_tick_unit;
return error
END

3.2.4. DCM Module

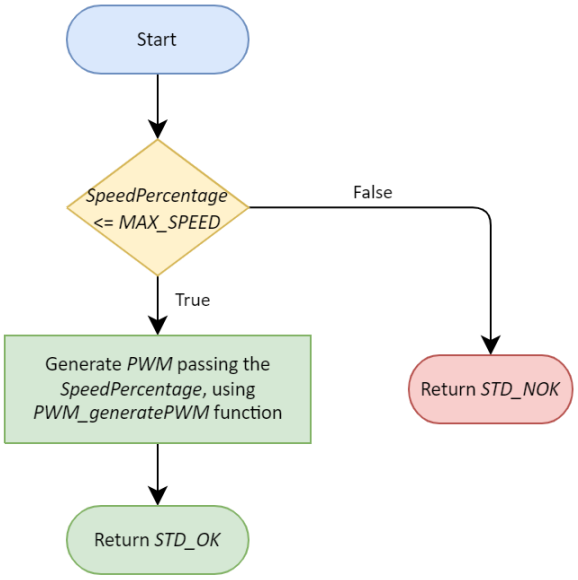
A. DCM_initialization



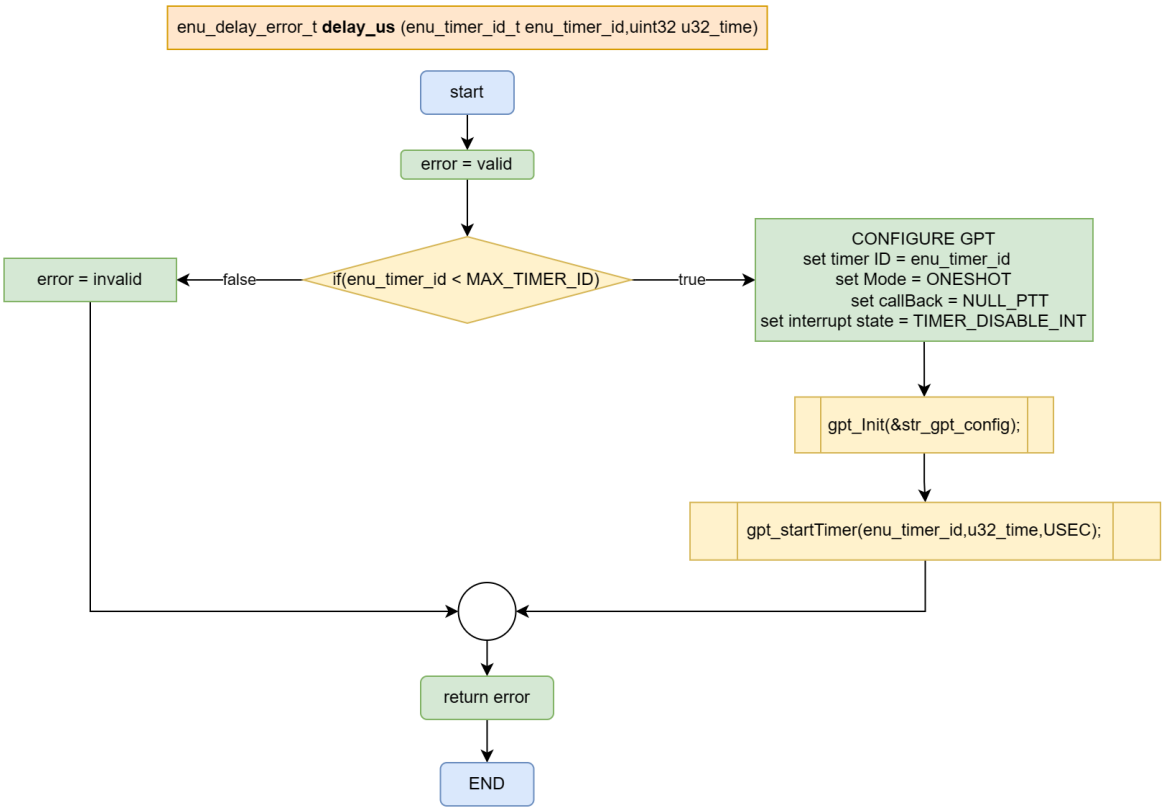
B. DCM_controlDCM

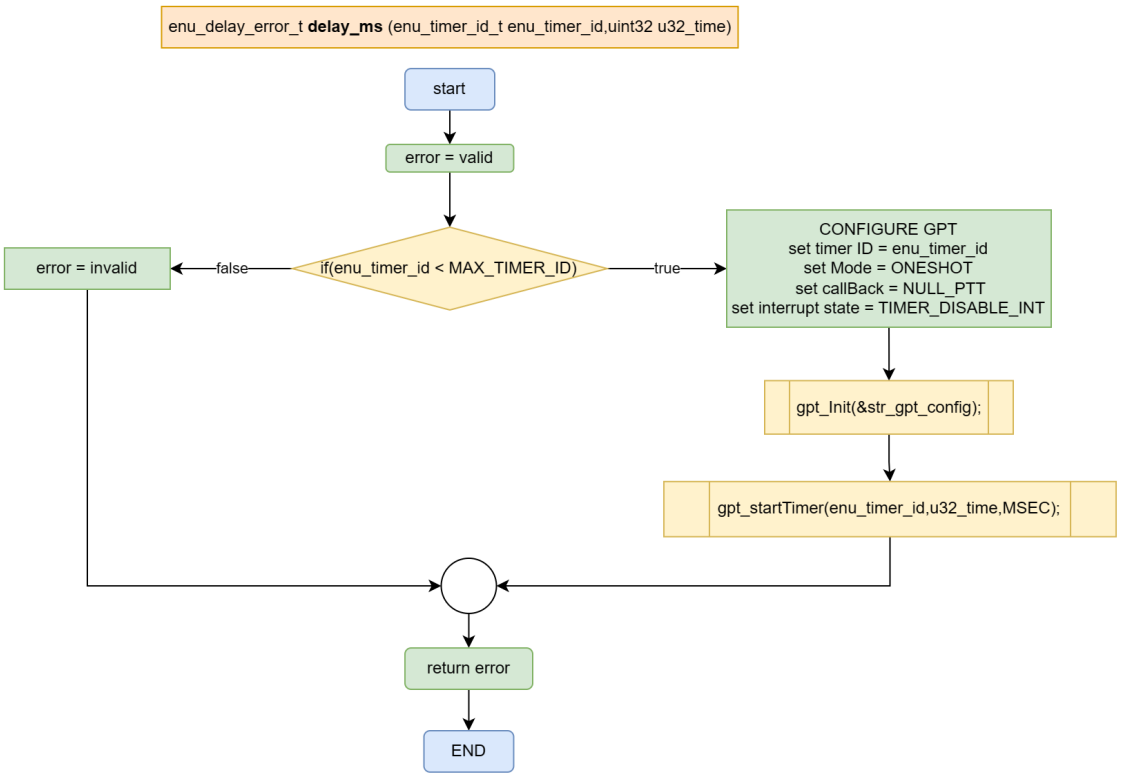


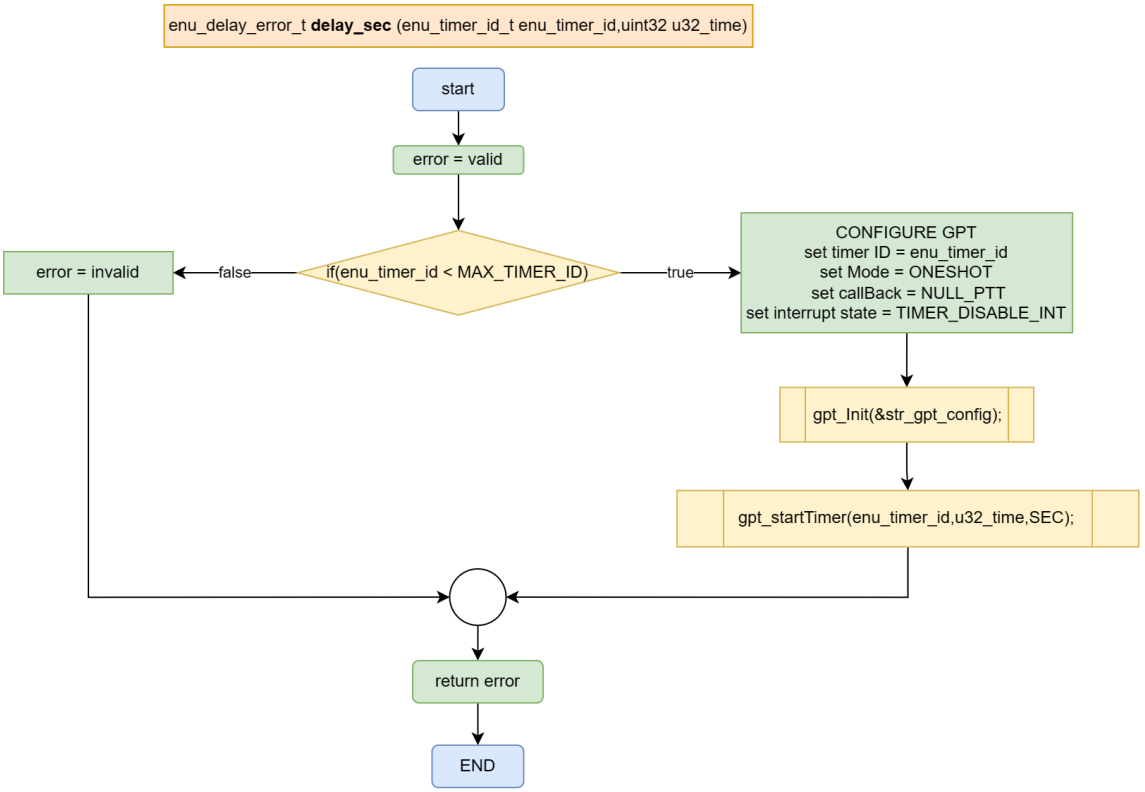
C. DCM_controlDCMSpeed



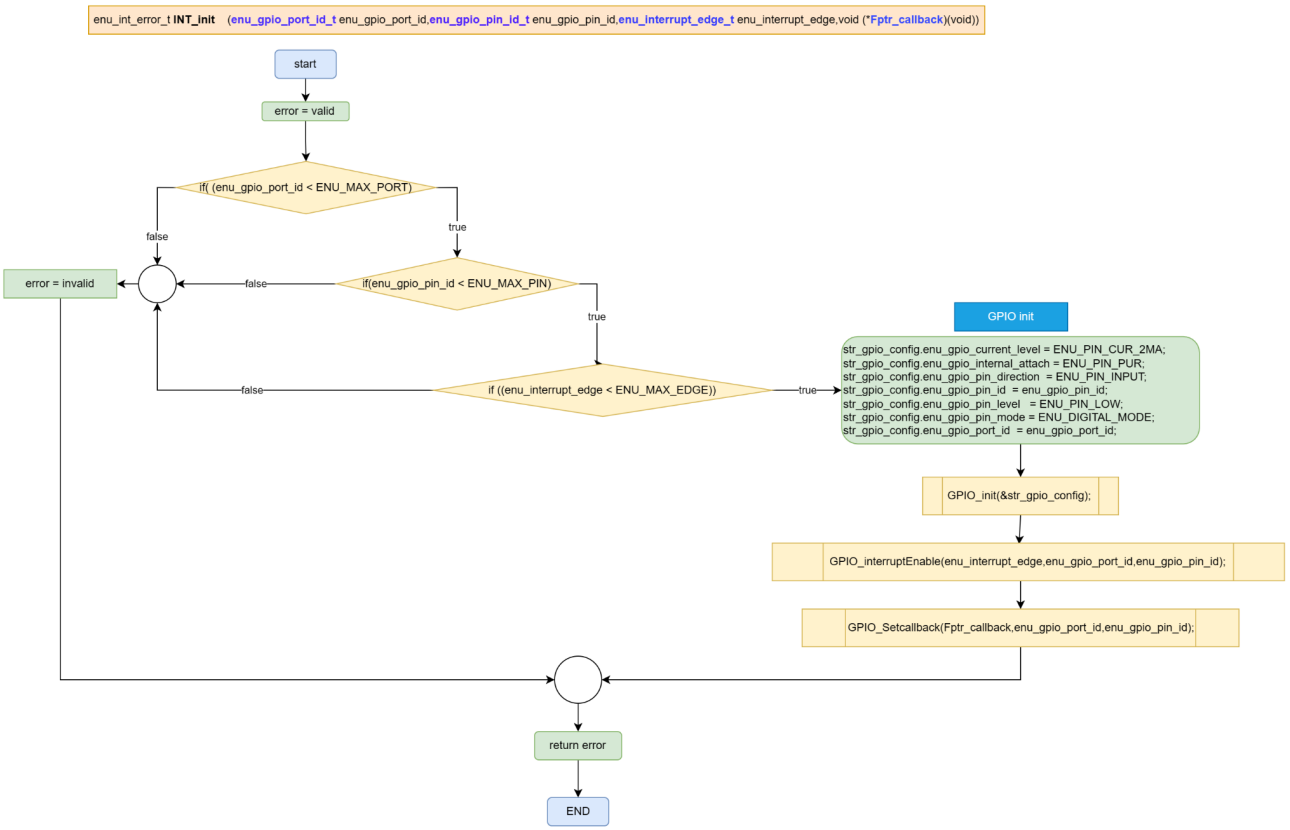
3.2.5. DELAY Module

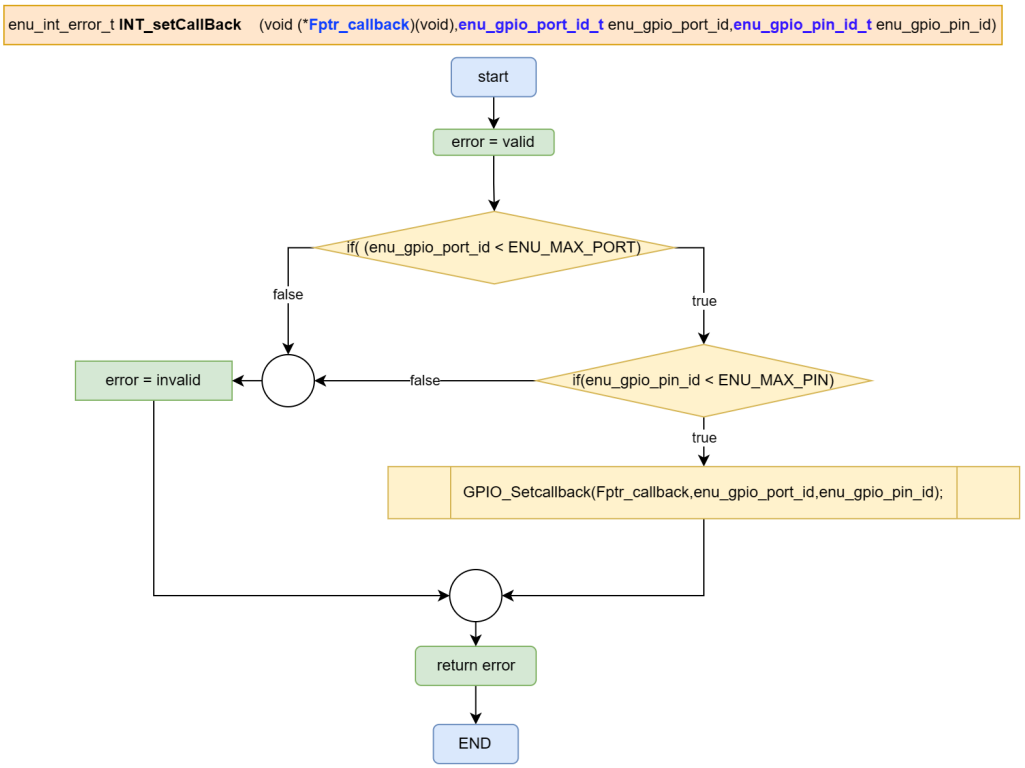


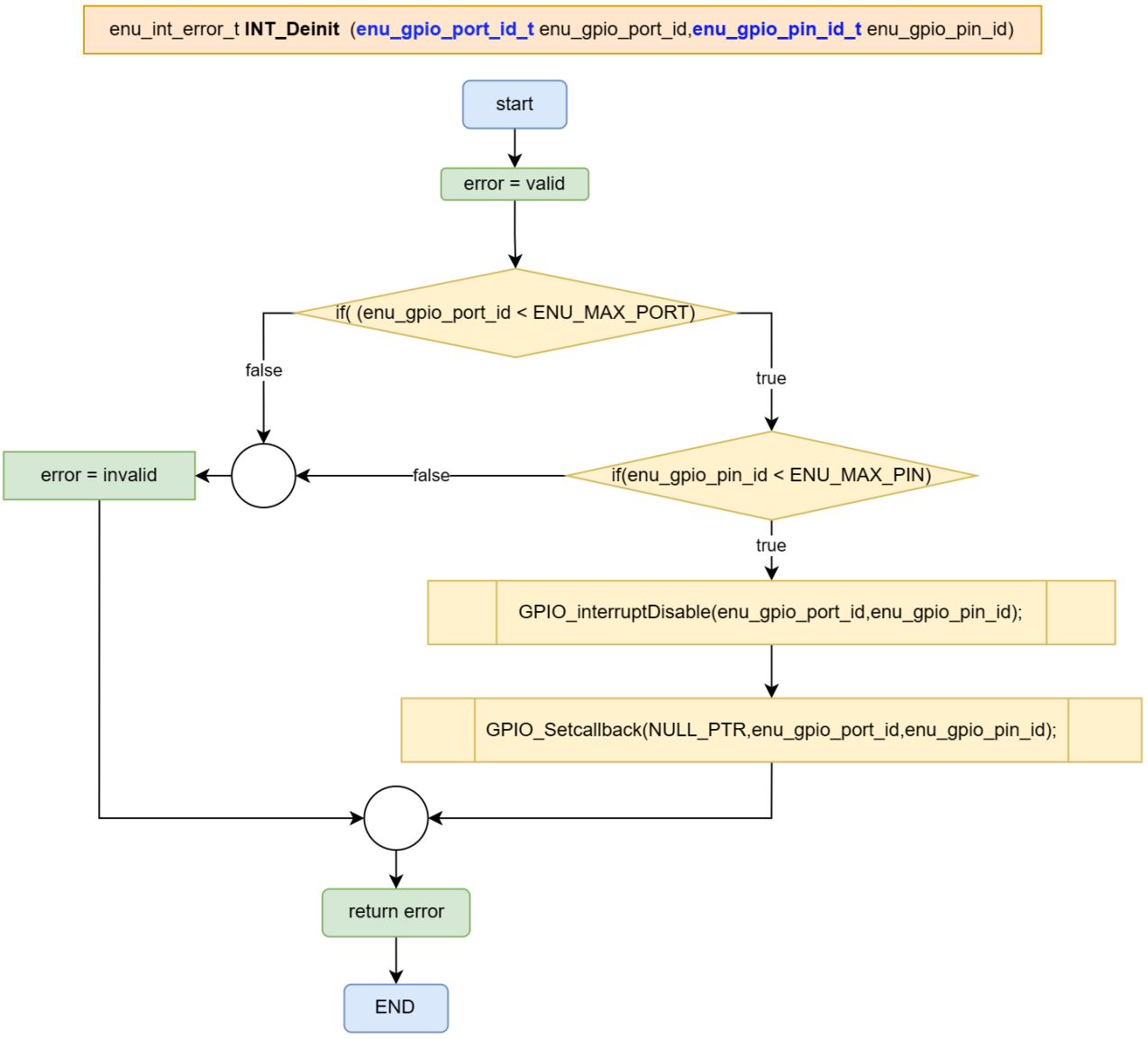



enu_delay_error_t **delay_sec** (enu_timer_id_t enu_timer_id,uint32 u32_time)
start
error = valid
if(enu_timer_id < MAX_TIMER_ID)
false
error = invalid
true
CONFIGURE GPT
set timer ID = enu_timer_id
set Mode = ONESHOT
set callBack = NULL_PTT
set interrupt state = TIMER_DISABLE_INT
gpt_Init(&str_gpt_config);
gpt_startTimer(enu_timer_id,u32_time,SEC);
return error
END

3.2.6. EXTI Module

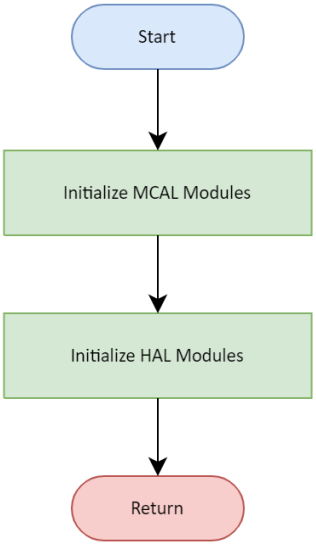




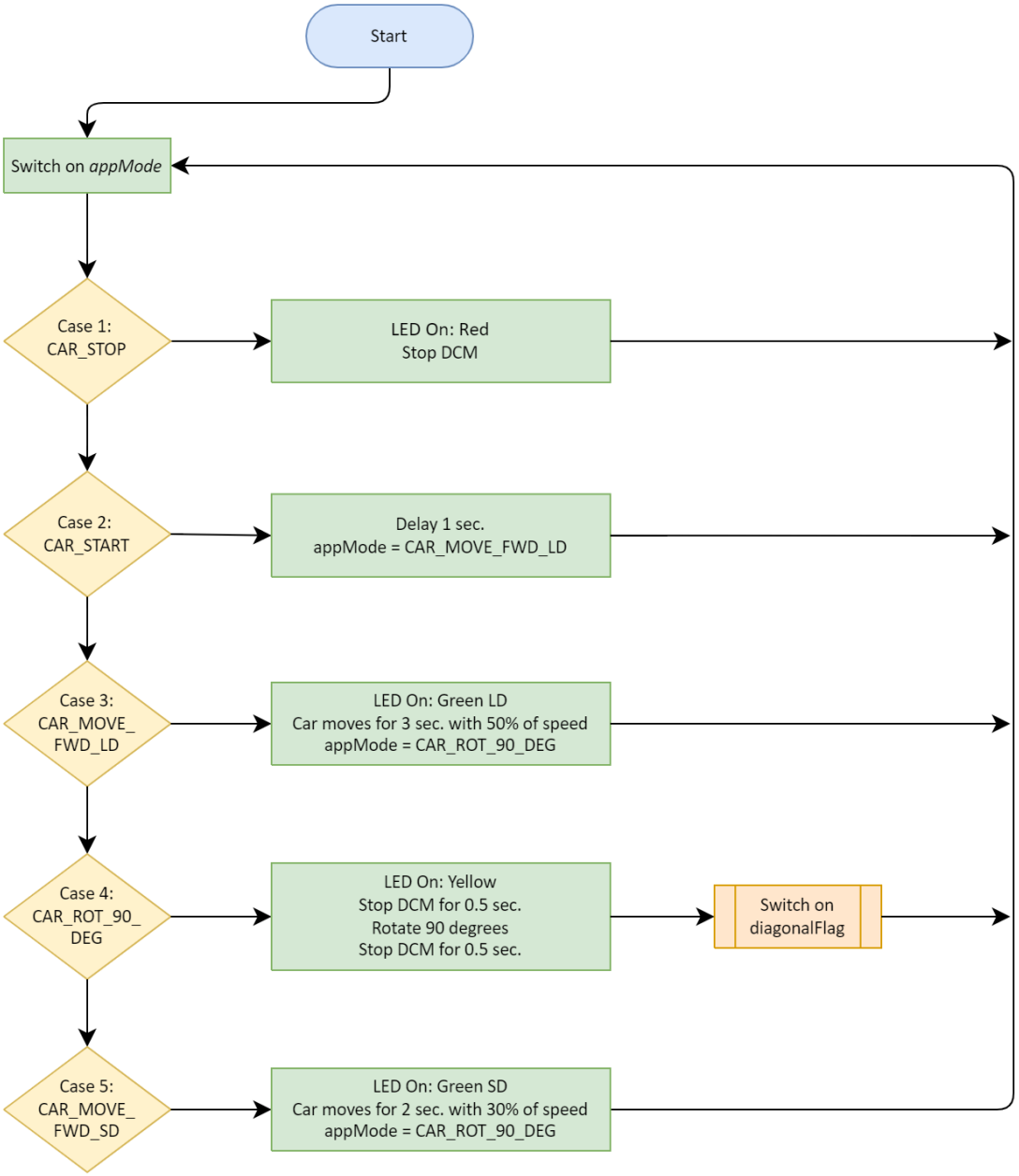

enu_int_error_t INT_Deinit (enu_gpio_port_id_t enu_gpio_port_id,enu_gpio_pin_id_t enu_gpio_pin_id)
start
error = valid
if((enu_gpio_port_id < ENU_MAX_PORT)
false
true
error = invalid
false
if(enu_gpio_pin_id < ENU_MAX_PIN)
true
GPIO_interruptDisable(enu_gpio_port_id,enu_gpio_pin_id);
GPIO_Setcallback(NULL_PTR,enu_gpio_port_id,enu_gpio_pin_id);
return error
END

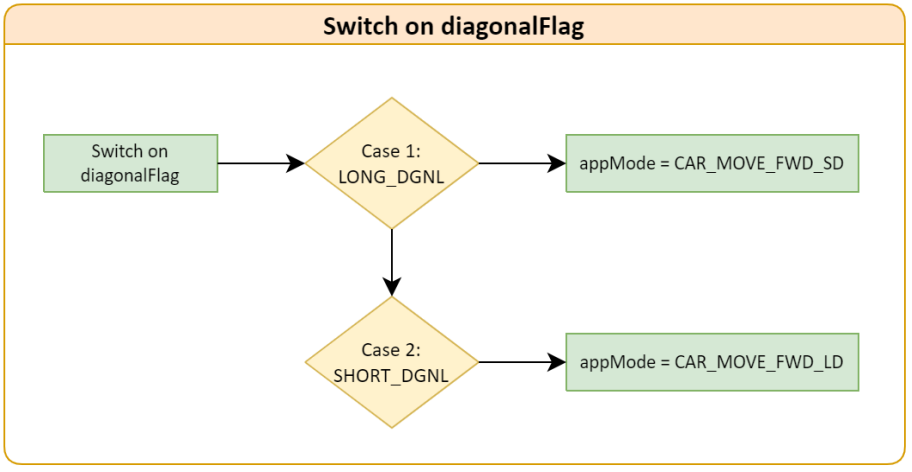
3.3. APP Layer

A. APP_initiailization

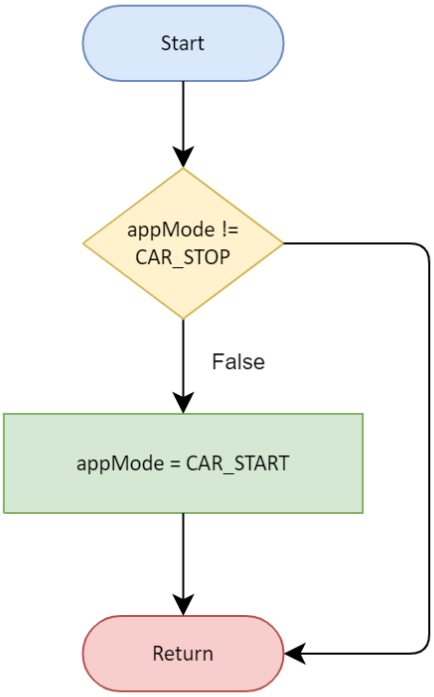


B. APP_startProgram

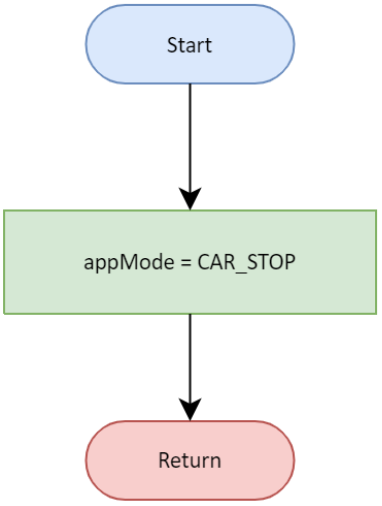




C. APP_startCar



D. APP_stopCar



4. Development Issues

4.1. Team Issues

The first main issue is as a team of ***four*** members who have to engage in completing tasks, we were facing a sort of hardship implementing the drivers and application with only ***three*** members, which led to delay in the project submission.

4.2. Hardware Issues

During the development of this project, we encountered some hardware issues that required us to make certain modifications. Initially, we intended to utilize the LEDs and motor driver L298N module on the connected board (**Sprints** Board) with our Tiva C board, which allowed us to leverage its ARM processors.

However, we faced difficulties in establishing a proper connection between the two boards, resulting in unreliable functionality. As a result, we decided to integrate an external L298N module and utilize the RGB LED available on the Tiva C board instead of the originally planned LEDs. In this configuration, the other board served solely as a power source for the components we mentioned.

Although we had to adjust our initial hardware setup, these adaptations allowed us to overcome the challenges and proceed with the development of the robot, showcasing our programming skills in controlling precise movements and designing an efficient control system.

4.3. Software Issues

4.3.1. System Clock Settings Adjustment

During the course of our project, we encountered timing issues when utilizing the GPT (General-Purpose Timer) for creating delays. After thorough debugging and extensive research in the datasheet, we determined that the root cause of the problem lay in the configuration of the clock settings. The following figures show the configurations that we modified in order to make delay timing correct.

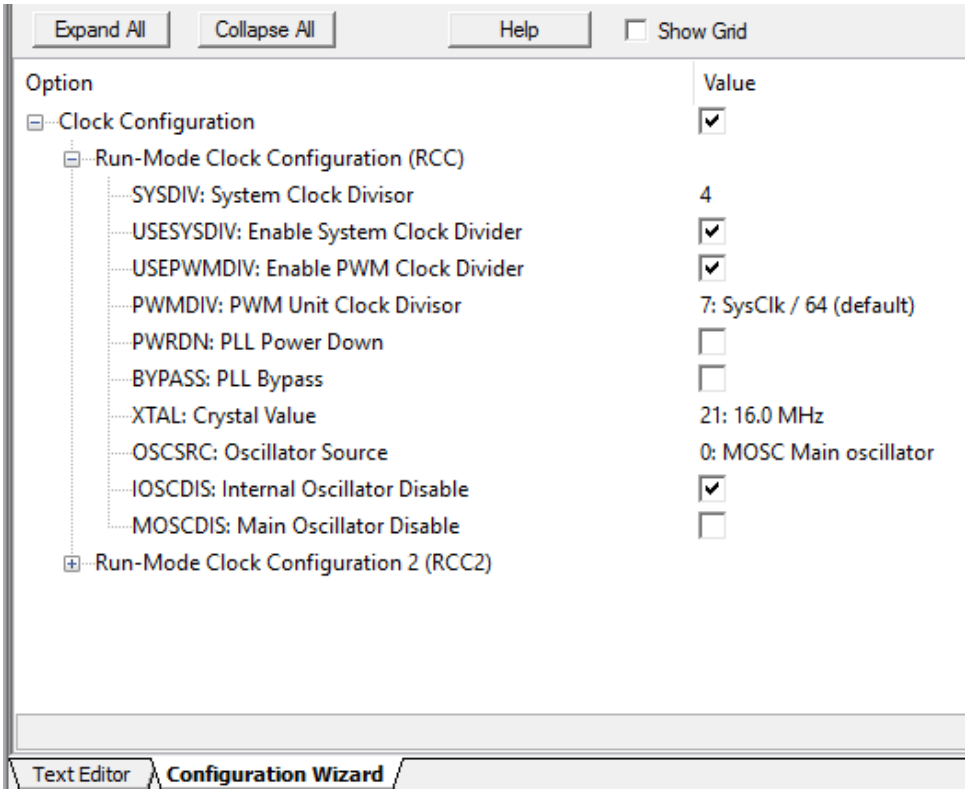


Figure 1. Incorrect Configurations

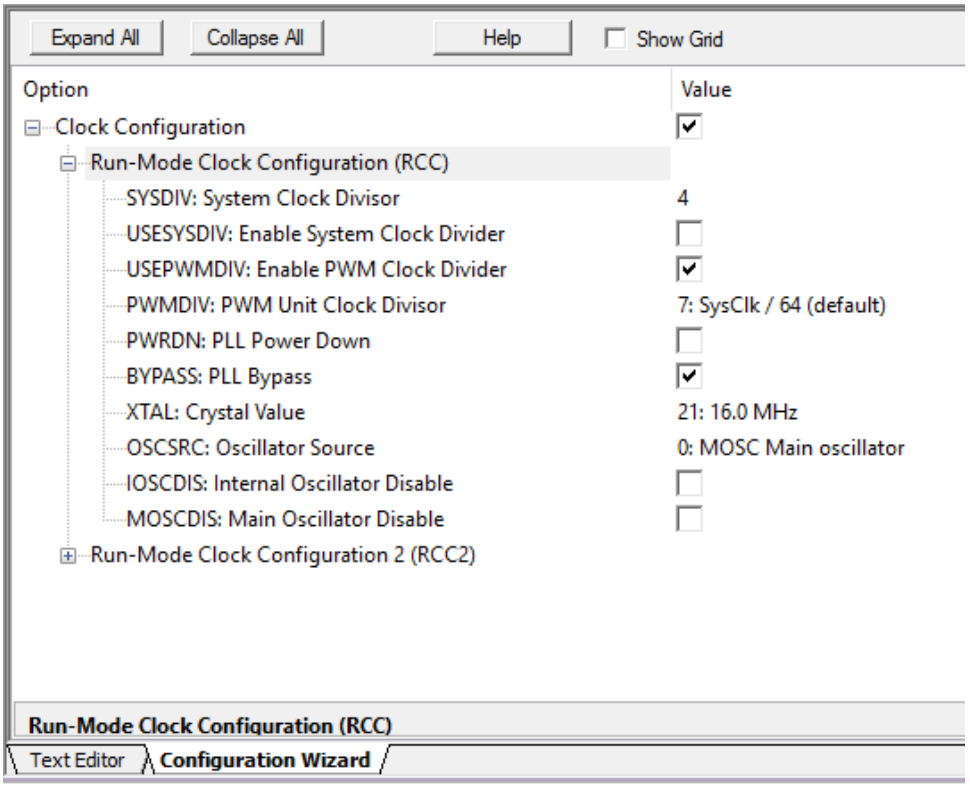


Figure 2. Correct Configurations

4.3.2. GPT counting up/down

During the development process, we faced another challenge related to the PWM module. We noticed unpredictable behavior when updating the duty cycle, which consequently changed the GPT preload value.

After thorough debugging, we discovered that the GPT was waiting for the timer to overflow and start from zero before the new preload value would take effect. To address this issue, we decided to change the count direction from count up to count down.

This adjustment allowed us to synchronize the update of the preload value with the desired timing, ultimately resolving the problem we encountered.

5. References

1. *Draw IO*
2. *Layered Architecture | Baeldung on Computer Science*
3. *Block Diagram Maker | Free Block Diagram Online | Lucidchart*
4. *Microcontroller Abstraction Layer (MCAL) | Renesas*
5. *Hardware Abstraction Layer - an overview | ScienceDirect Topics*
6. *Top Five Differences Between Layers And Tiers | Skill-Lync*
7. *What is a module in software, hardware and programming?*
8. *Embedded Basics – API's vs HAL's*