SPRINTS

22 JUNE

2023

RGB LED CONTROL V3.0 REPORT

RGB LED
CONTROL V3.0

Created By :
Momen Hassan
Sherif Ashraf Khadr

1 - Project Introduction

1.1 - Project Description

You are supposed to develop the Timer Driver and use it to control the RGB LED brightness on the TivaC board based on the push button press.

1.2 - Project Components

1.2.2 Hardware Requirements

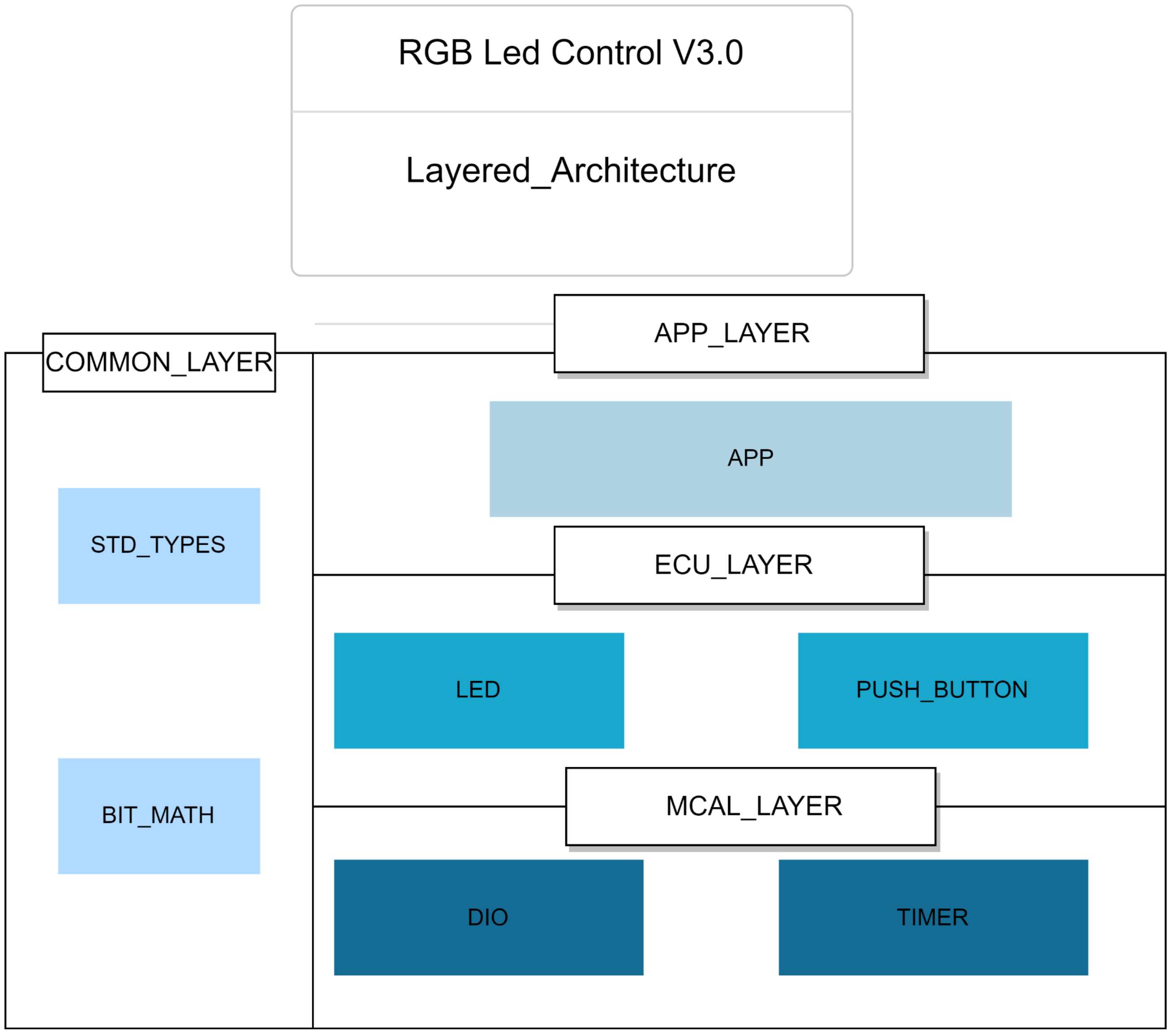
- Use the TivaC board
- Use SW1 as an input button
- Use the RGB LED

1.2.3 Software Requirements

- The RGB LED is OFF initially
- The PWM signal has a 500ms duration
- The system has four states
  - SW1 - First press
    - The Green LED will be on with a 30% duty cycle
  - SW1 - Second press
    - The Green LED will be on with a 60% duty cycle
  - SW1 -Third press
    - The Green LED will be on with a 90% duty cycle
  - SW1 - Fourth press will be off
    - The Green LED will be off
  - On the fifth press, system state will return to state 1

2 - High Level Design

2.1 - Layered Architecture



2.2 - Modules Description

2.2.1 - GPIO Module

A DIO (Digital Input/Output) module is a hardware component that provides digital input and output capabilities to a system. It can be used to interface with digital sensors, switches, and actuators, and typically includes features such as interrupt capability, programmable resistors, overvoltage/current protection, and isolation. Additionally, some DIO modules may include advanced features such as counter/timer functionality or PWM.

2.2.2 - TIMER Module

The timer module in arm is a device that generates and controls timers. It has two counters that can count up or down with different settings. It can use the system clock or an external test clock as input. It has a register that stores the test mode of the counters. It connects to the APB bus.

2.2.3 - LED Module

A LED (Light Emitting Diode) module is a hardware component that provides visual output to a system. It can be controlled by software running on a microcontroller and typically includes features such as brightness control, colour selection, and blinking patterns. The LED module is useful for providing status indicators, displaying data, or as a user interface, and can be interfaced with the microcontroller through different protocols such as I2C, SPI, or digital I/O.

2.2.4 - PUSH_BUTTON Module

A push button is a simple switch mechanism that controls some aspect of a machine or a process. It is usually made of plastic or metal and has a flat or shaped surface that can be easily pressed or pushed. A push button can be either momentary or latching, meaning that it returns to its original state when released or stays in the pushed state until pressed again. Push buttons are used for various purposes, such as turning on or off devices, performing calculations and controlling games.

2.2.5 - APP Module

An app module can use ECU driver modules to handle the flow of a specific application within a system. The ECU driver module provides low-level access to the hardware components of the system, such as sensors and actuators. The app module uses these driver modules to interact with the system and perform its specific tasks, resulting in more efficient development and better code organization.

2.3 - Drivers' documentation

2.3.1 - GPIO Driver

```
/**
 * @brief Initializes the GPIO pin with the given configuration parameters.
 * @param arg_pincfg A pointer to a structure that contains the configuration
parameters for the GPIO pin.
```

 * @return An enumeration value that indicates the system state after the initialization.
 */
ENU_GPIO_systemState_t GPIO_init(ST_dio_pinCfg_t *arg_pincfg);

/**
 * @brief Enables the interrupt for the GPIO port that contains the given pin.
 * @param arg_pincfg A pointer to a structure that contains the configuration parameters for the GPIO pin.
 * @return An enumeration value that indicates the system state after enabling the interrupt.
 */
ENU_GPIO_systemState_t GPIO_interruptPortEnable(ST_dio_pinCfg_t *arg_pincfg);

/**
 * @brief Disables the interrupt for the GPIO port that contains the given pin.
 * @param arg_pincfg A pointer to a structure that contains the configuration parameters for the GPIO pin.
 * @return An enumeration value that indicates the system state after disabling the interrupt.
 */
ENU_GPIO_systemState_t GPIO_interruptPortDisable(ST_dio_pinCfg_t *arg_pincfg);

/**
 * @brief Writes a logic value (high or low) to the GPIO pin.
 * @param arg_pincfg A pointer to a structure that contains the configuration parameters for the GPIO pin.
 * @param arg_logicValue An enumeration value that indicates the logic value to be written to the GPIO pin.
 * @return An enumeration value that indicates the system state after writing the logic value.
 */
ENU_GPIO_systemState_t GPIO_writeLogic(ST_dio_pinCfg_t *arg_pincfg , ENU_GPIO_logic_t arg_logicValue);

/**
 * @brief Reads a logic value (high or low) from the GPIO pin.

 * @param arg_pincfg A pointer to a structure that contains the configuration parameters for the GPIO pin.

 * @param arg_logicValue A pointer to an enumeration value that stores the logic value read from the GPIO pin.

 * @return An enumeration value that indicates the system state after reading the logic value.

 */

ENU_GPIO_systemState_t GPIO_readLogic(ST_dio_pinCfg_t *arg_pincfg , ENU_GPIO_logic_t *arg_logicValue);

/**

 * @brief Toggles a logic value (high or low) of the GPIO pin.

 * @param arg_pincfg A pointer to a structure that contains the configuration parameters for the GPIO pin.

 * @return An enumeration value that indicates the system state after toggling the logic value.

 */

ENU_GPIO_systemState_t GPIO_toggleLogic(ST_dio_pinCfg_t *arg_pincfg);

2.3.2 - TIMER Driver

/**
@brief Initializes all the configured Gpt channels with the given configuration pointer.
@param Config Pointer to the configuration structure
@return GPT_OK if the initialization is successful, or GPT_ERROR if there is an error */

Gpt_State Gpt_init(Gpt_ConfigType *Config);

/**
@brief Enables the interrupt notification for the specified channel.
@param channel_id The channel identifier
@return GPT_OK if the operation is successful, or GPT_ERROR if there is an error */

Gpt_State GPT_enable_intturpt(Gpt_ChannelType channel_id);

/**
@brief Disables the interrupt notification for the specified channel.
@param channel_id The channel identifier
@return GPT_OK if the operation is successful, or GPT_ERROR if there is an error */

Gpt_State GPT_disable_intturpt(Gpt_ChannelType channel_id);

/**
@brief Initializes the PWM module with the given period and duty cycle parameters.
@param ui32Period The PWM period in clock cycles
@param ui8DutyCycle The PWM duty cycle in percentage
@return PWM_OK if the initialization is successful, or PWM_ERROR if there is an error */

Gpt_State PWM_Init(uint32_t ui32Period, uint8_t ui8DutyCycle);

/**
@brief Starts the selected timer channel with the defined target time.
@param channel_id The channel identifier
@param u32_time The target time in clock cycles
@return GPT_OK if the operation is successful, or GPT_ERROR if there is an error */

Gpt_State GPT_Start_Timer(Gpt_ChannelType channel_id,uint32_t u32_time);

/**
@brief Stops the selected timer channel.
@param channel_id The channel identifier
@return GPT_OK if the operation is successful, or GPT_ERROR if there is an error */

Gpt_State GPT_STOP_Timer(Gpt_ChannelType channel_id);

/**
@brief Returns the time elapsed for channel which is referenced.
@param channel_id The channel identifier
@param timer_use The timer number
@param time Pointer to store the elapsed time in clock cycles
@return GPT_OK if the operation is successful, or GPT_ERROR if there is an error */

Gpt_State GPT_GetElapsedTime(Gpt_ChannelType channel_id,Timer_Use timer_use,uint32_t *time);

/**
@brief Returns the timer value remaining until the target time will be reached next time.
@param channel_id The channel identifier
@param timer_use The timer number
@param time Pointer to store the remaining time in clock cycles
@return GPT_OK if the operation is successful, or GPT_ERROR if there is an error */

Gpt_State GPT_GetRemainingTime(Gpt_ChannelType channel_id,Timer_Use timer_use,uint32_t *time);

2.3.3 - LED Driver

```
/**
 * @brief Initializes the LED with the given pin configuration parameters.
 * @param led A pointer to a structure that contains the pin configuration parameters for the LED.
 * @return An enumeration value that indicates the system state after the initialization.
 */
ENU_LED_systemState_t LED_initialize(const ST_led_pinCfg_t *led);

/**
 * @brief Turns on the LED by writing a high logic value to the pin.
 * @param led A pointer to a structure that contains the pin configuration parameters for the LED.
 * @return An enumeration value that indicates the system state after turning on the LED.
 */
ENU_LED_systemState_t LED_turnOn(const ST_led_pinCfg_t *led);

/**
 * @brief Turns off the LED by writing a low logic value to the pin.
 * @param led A pointer to a structure that contains the pin configuration parameters for the LED.
 * @return An enumeration value that indicates the system state after turning off the LED.
 */
ENU_LED_systemState_t LED_turnOff(const ST_led_pinCfg_t *led);

/**
 * @brief Toggles the LED by writing the opposite logic value to the pin.
 * @param led A pointer to a structure that contains the pin configuration parameters for the LED.
 * @return An enumeration value that indicates the system state after toggling the LED.
 */
ENU_LED_systemState_t LED_toggle(const ST_led_pinCfg_t *led);
```

2.3.4 - PUSH_BOTTON Driver

```
/**
 * @brief Initializes the push button with the given pin configuration parameters.
 * @param btn A pointer to a structure that contains the pin configuration parameters for the push button.
 * @return An enumeration value that indicates the system state after the initialization.
 */
```

ENU_PUSH_BTN_systemState_t PUSH_BTN_intialize(const ST_PUSH_BTN_pinCfg_t *btn);

```
/**
 * @brief Reads the current state of the push button (pressed or released).
 * @param btn A pointer to a structure that contains the pin configuration parameters for the push button.
 * @param btn_state A pointer to an enumeration value that stores the current state of the push button.
 * @return An enumeration value that indicates the system state after reading the push button state.
 */
```

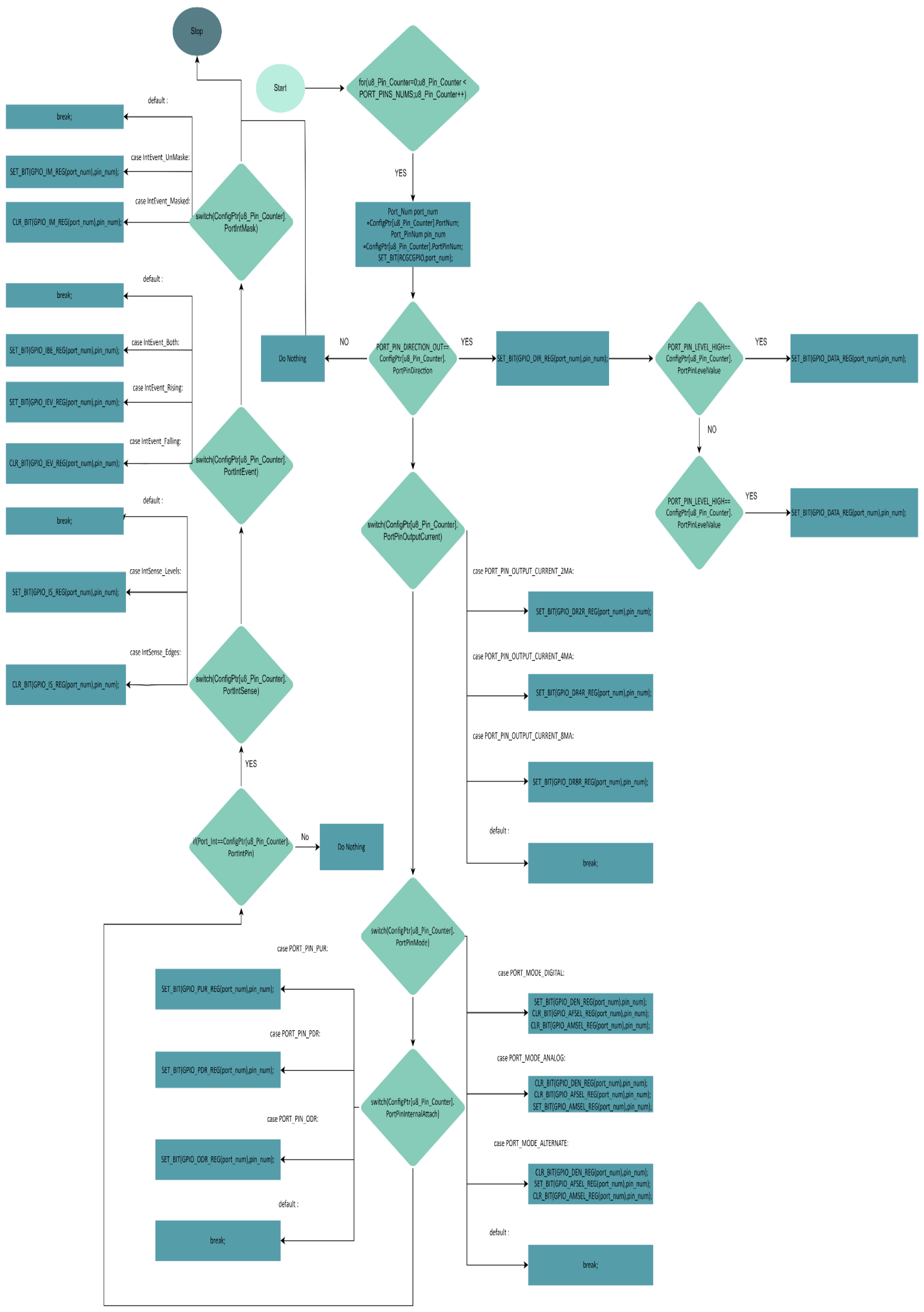
ENU_PUSH_BTN_systemState_t PUSH_BTN_read_state(const ST_PUSH_BTN_pinCfg_t *btn , ENU_PUSH_BTN_state_t *btn_state);

3 - Low-Level Design

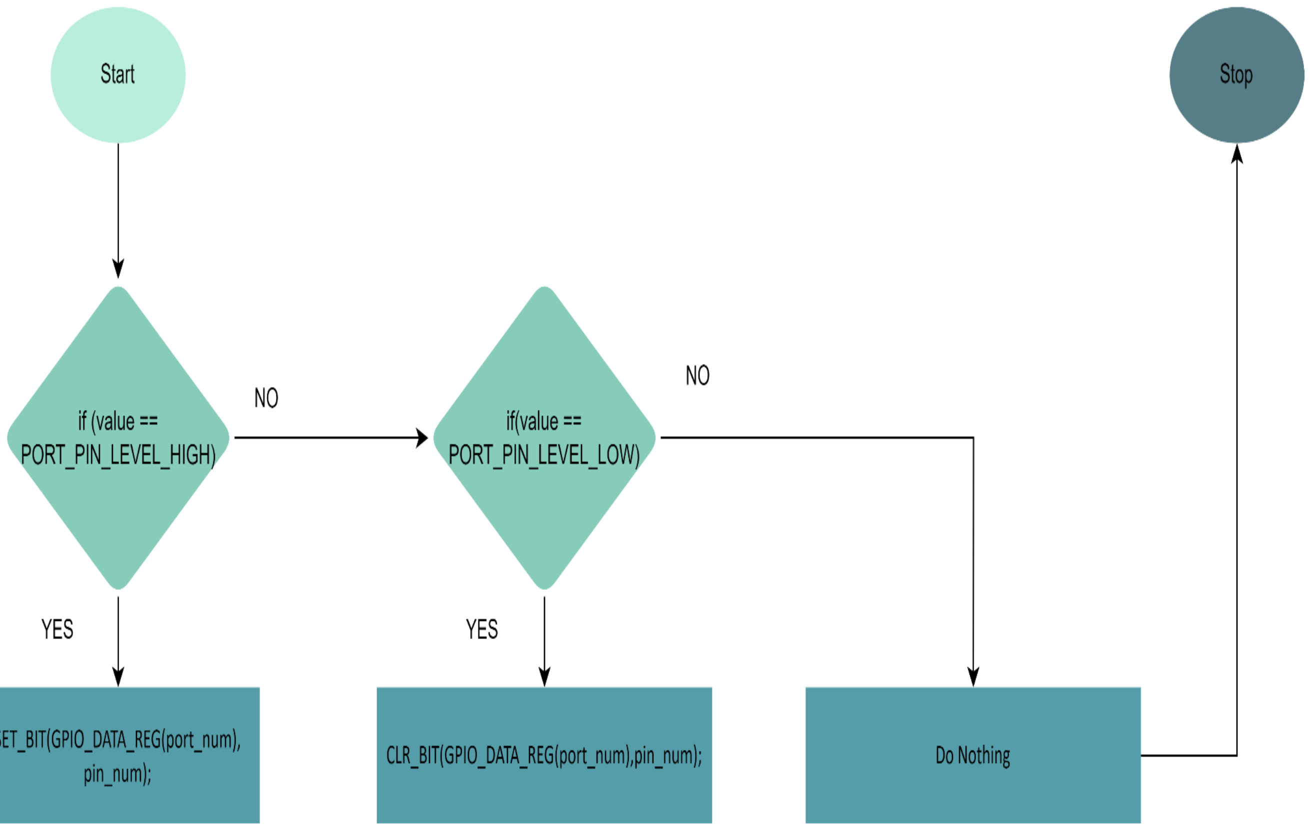
3.1 - Module Flow Charts

3.1.1 - GPIO Flow Charts

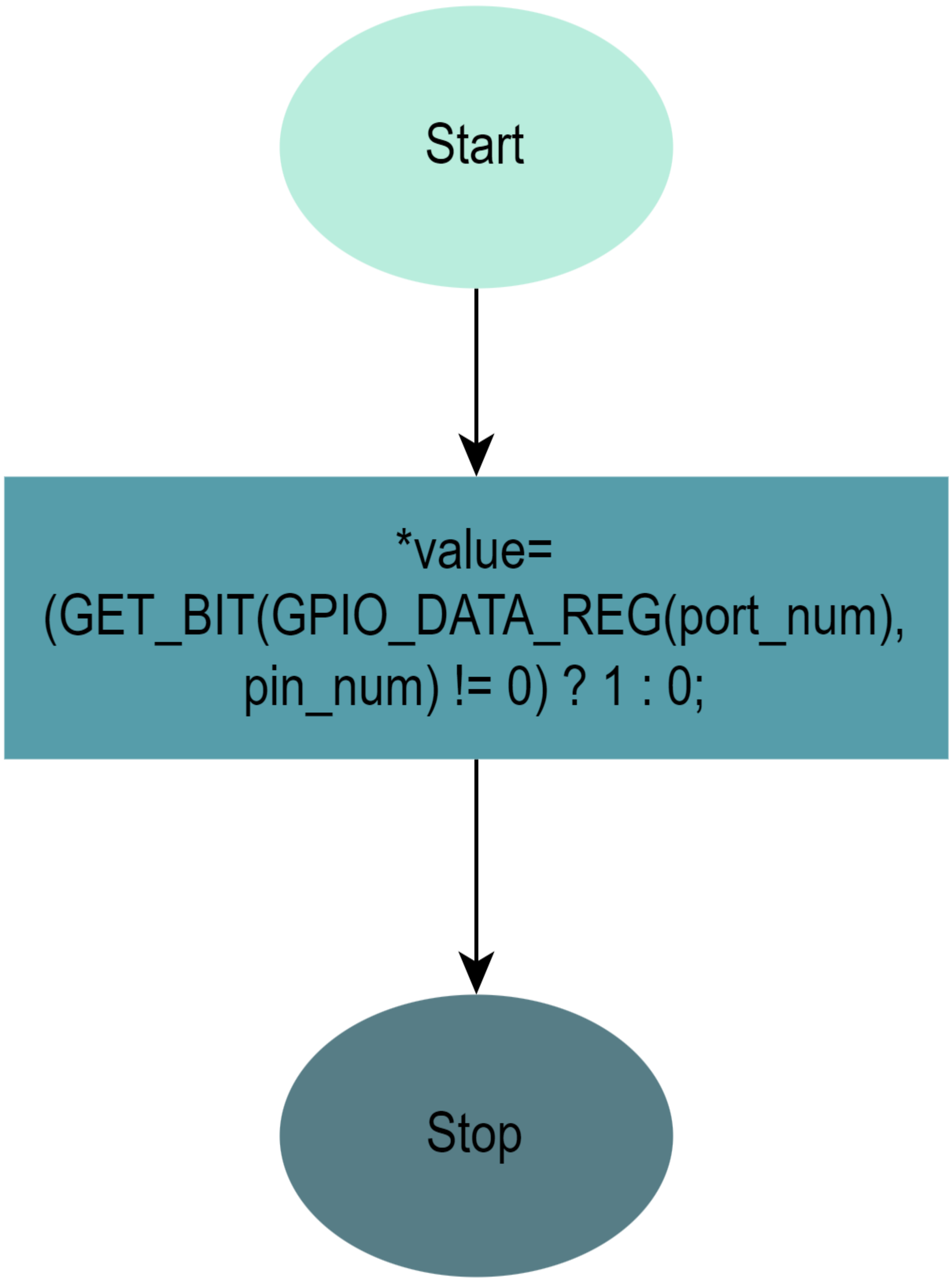
3.1.1.1 - GPIO_INIT


Stop
Start
for(u8_Pin_Counter=0;u8_Pin_Counter < PORT_PINS_NUMS;u8_Pin_Counter++)
YES
Port_Num port_num =ConfigPtr[u8_Pin_Counter].PortNum; Port_PinNum pin_num =ConfigPtr[u8_Pin_Counter].PortPinNum; SET_BIT(RCGCGPIO,port_num);
PORT_PIN_DIRECTION_OUT== ConfigPtr[u8_Pin_Counter]. PortPinDirection
NO
Do Nothing
YES
SET_BIT(GPIO_DIR_REG(port_num),pin_num);
PORT_PIN_LEVEL_HIGH== ConfigPtr[u8_Pin_Counter]. PortPinLevelValue
YES
SET_BIT(GPIO_DATA_REG(port_num),pin_num);
NO
PORT_PIN_LEVEL_HIGH== ConfigPtr[u8_Pin_Counter]. PortPinLevelValue
YES
SET_BIT(GPIO_DATA_REG(port_num),pin_num);
switch(ConfigPtr[u8_Pin_Counter]. PortPinOutputCurrent)
case PORT_PIN_OUTPUT_CURRENT_2MA:
SET_BIT(GPIO_DR2R_REG(port_num),pin_num);
case PORT_PIN_OUTPUT_CURRENT_4MA:
SET_BIT(GPIO_DR4R_REG(port_num),pin_num);
case PORT_PIN_OUTPUT_CURRENT_8MA:
SET_BIT(GPIO_DR8R_REG(port_num),pin_num);
default :
break;
switch(ConfigPtr[u8_Pin_Counter]. PortPinMode)
case PORT_MODE_DIGITAL:
SET_BIT(GPIO_DEN_REG(port_num),pin_num); CLR_BIT(GPIO_AFSEL_REG(port_num),pin_num); CLR_BIT(GPIO_AMSEL_REG(port_num),pin_num);
case PORT_MODE_ANALOG:
CLR_BIT(GPIO_DEN_REG(port_num),pin_num); CLR_BIT(GPIO_AFSEL_REG(port_num),pin_num); SET_BIT(GPIO_AMSEL_REG(port_num),pin_num);
case PORT_MODE_ALTERNATE:
CLR_BIT(GPIO_DEN_REG(port_num),pin_num); SET_BIT(GPIO_AFSEL_REG(port_num),pin_num); CLR_BIT(GPIO_AMSEL_REG(port_num),pin_num);
default :
break;
switch(ConfigPtr[u8_Pin_Counter]. PortPinInternalAttach)
case PORT_PIN_PUR:
SET_BIT(GPIO_PUR_REG(port_num),pin_num);
case PORT_PIN_PDR:
SET_BIT(GPIO_PDR_REG(port_num),pin_num);
case PORT_PIN_ODR:
SET_BIT(GPIO_ODR_REG(port_num),pin_num);
default :
break;
if(Port_Int==ConfigPtr[u8_Pin_Counter]. PortIntPin)
No
Do Nothing
YES
switch(ConfigPtr[u8_Pin_Counter]. PortIntSense)
case IntSense_Levels:
SET_BIT(GPIO_IS_REG(port_num),pin_num);
case IntSense_Edges:
CLR_BIT(GPIO_IS_REG(port_num),pin_num);
default :
break;
switch(ConfigPtr[u8_Pin_Counter]. PortIntEvent)
case IntEvent_Both:
SET_BIT(GPIO_IBE_REG(port_num),pin_num);
case IntEvent_Rising:
SET_BIT(GPIO_IEV_REG(port_num),pin_num);
case IntEvent_Falling:
CLR_BIT(GPIO_IEV_REG(port_num),pin_num);
default :
break;
switch(ConfigPtr[u8_Pin_Counter]. PortIntMask)
case IntEvent_UnMaske:
SET_BIT(GPIO_IM_REG(port_num),pin_num);
case IntEvent_Masked:
CLR_BIT(GPIO_IM_REG(port_num),pin_num);
default :
break;

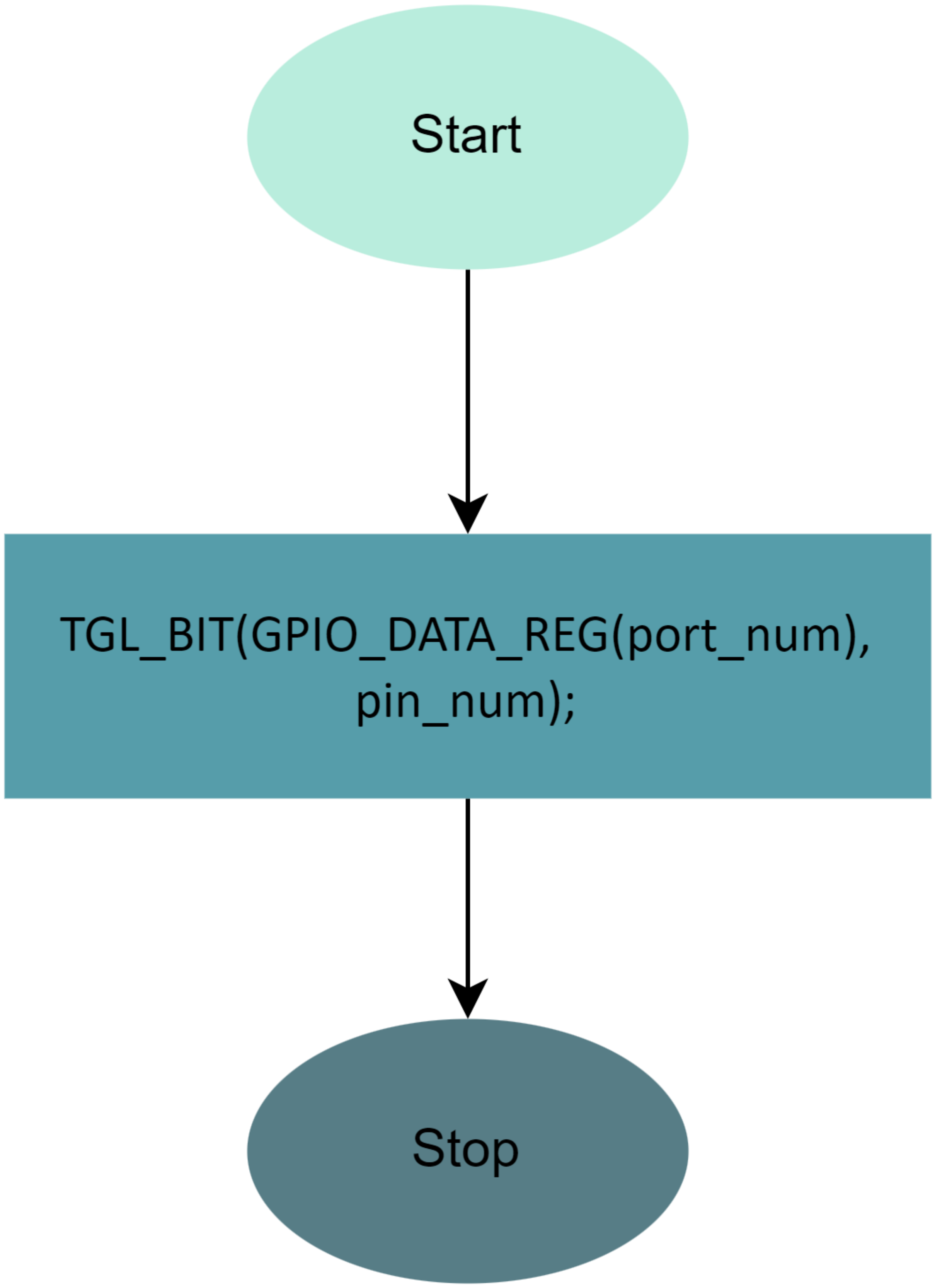
3.1.1.2 - GPIO_WRITE


Start
if (value == PORT_PIN_LEVEL_HIGH)
NO
if(value == PORT_PIN_LEVEL_LOW)
NO
YES
YES
SET_BIT(GPIO_DATA_REG(port_num), pin_num);
CLR_BIT(GPIO_DATA_REG(port_num),pin_num);
Do Nothing
Stop

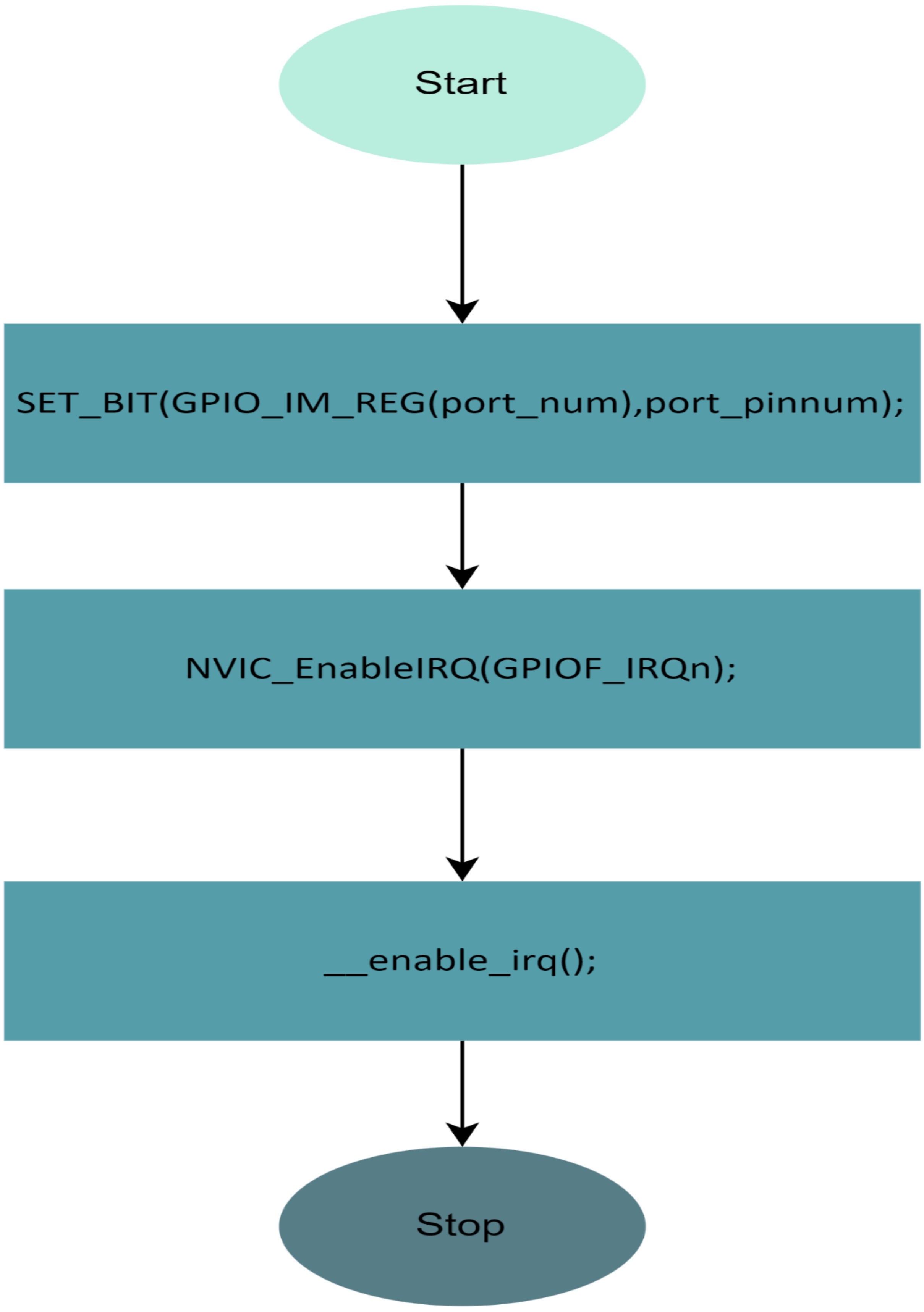
3.1.1.3 - GPIO_READ



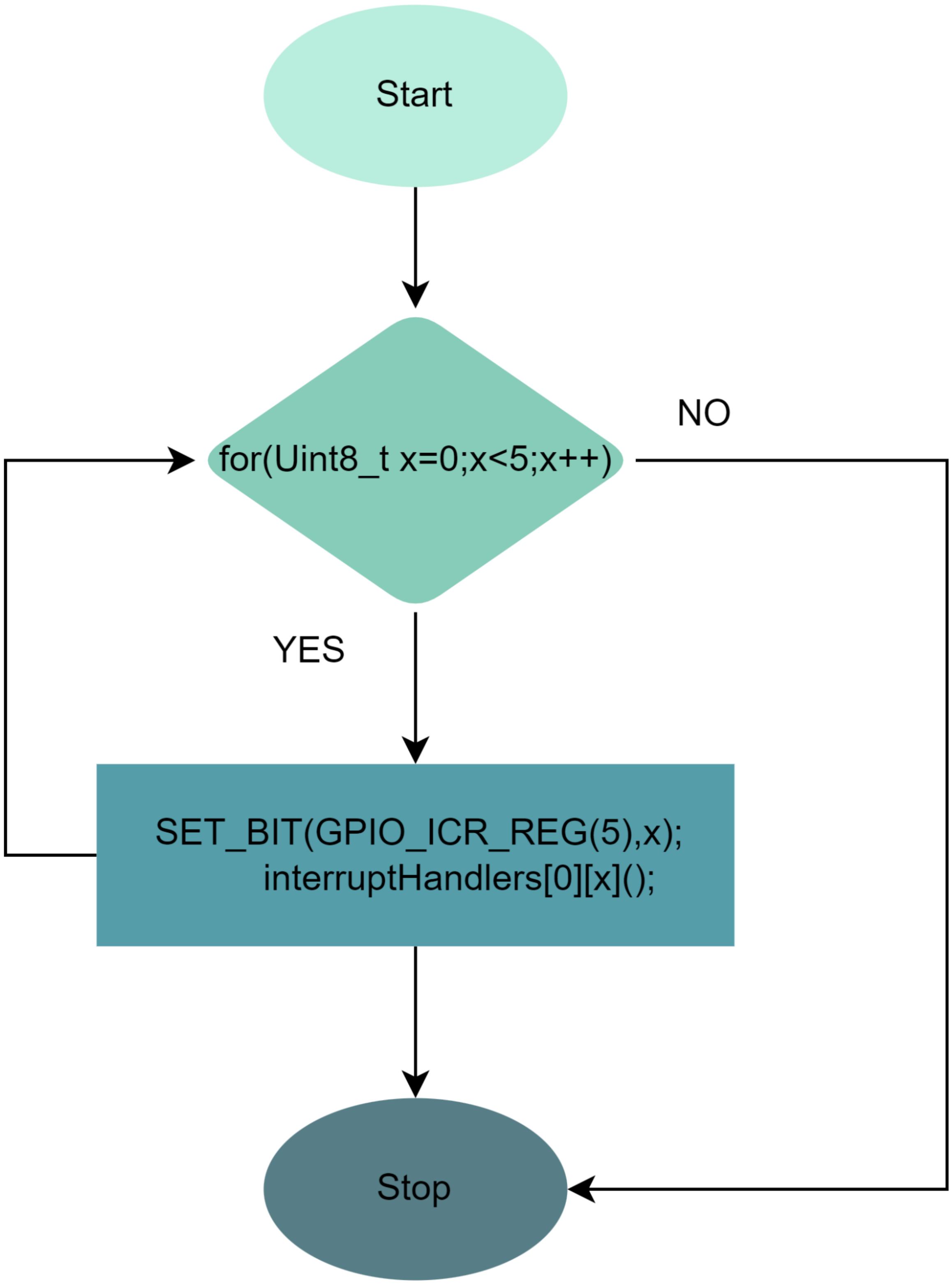
3.1.1.4 - GPIO_TOGGLE



3.1.1.5 - GPIO_ENABLE_INTERRUPT

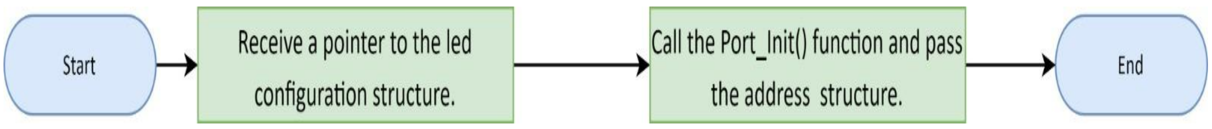

Start
SET_BIT(GPIO_IM_REG(port_num),port_pinnum);
NVIC_EnableIRQ(GPIOF_IRQn);
__enable_irq();
Stop

3.1.1.6 - GPIO_HANDLER

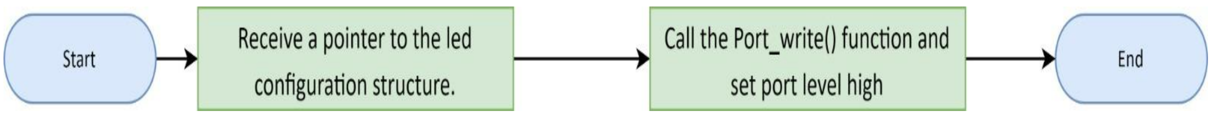


3.1.2 - LED FLOW Charts

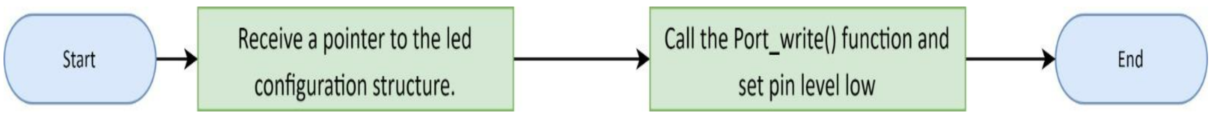
3.1.2.1 - LED_INIT


Start
Receive a pointer to the led configuration structure.
Call the Port_Init() function and pass the address structure.
End

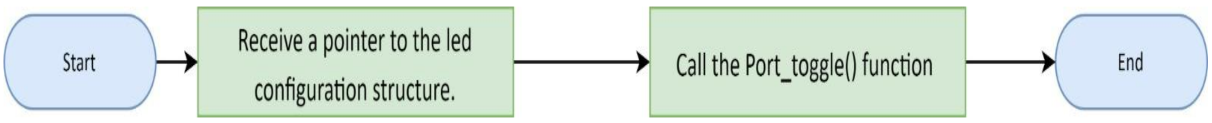
3.1.2.2 - LED_TURNON


Start
Receive a pointer to the led configuration structure.
Call the Port_write() function and set port level high
End

3.1.2.3 - LED_TURNOFF

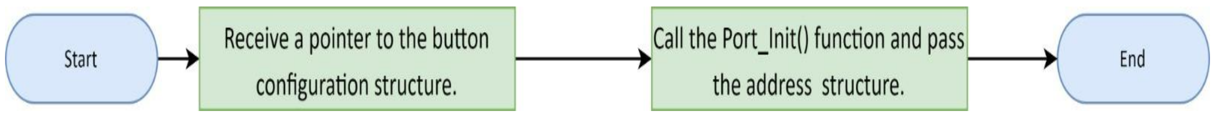

Start
Receive a pointer to the led configuration structure.
Call the Port_write() function and set pin level low
End

3.1.2.4 - LED_TOGGLE

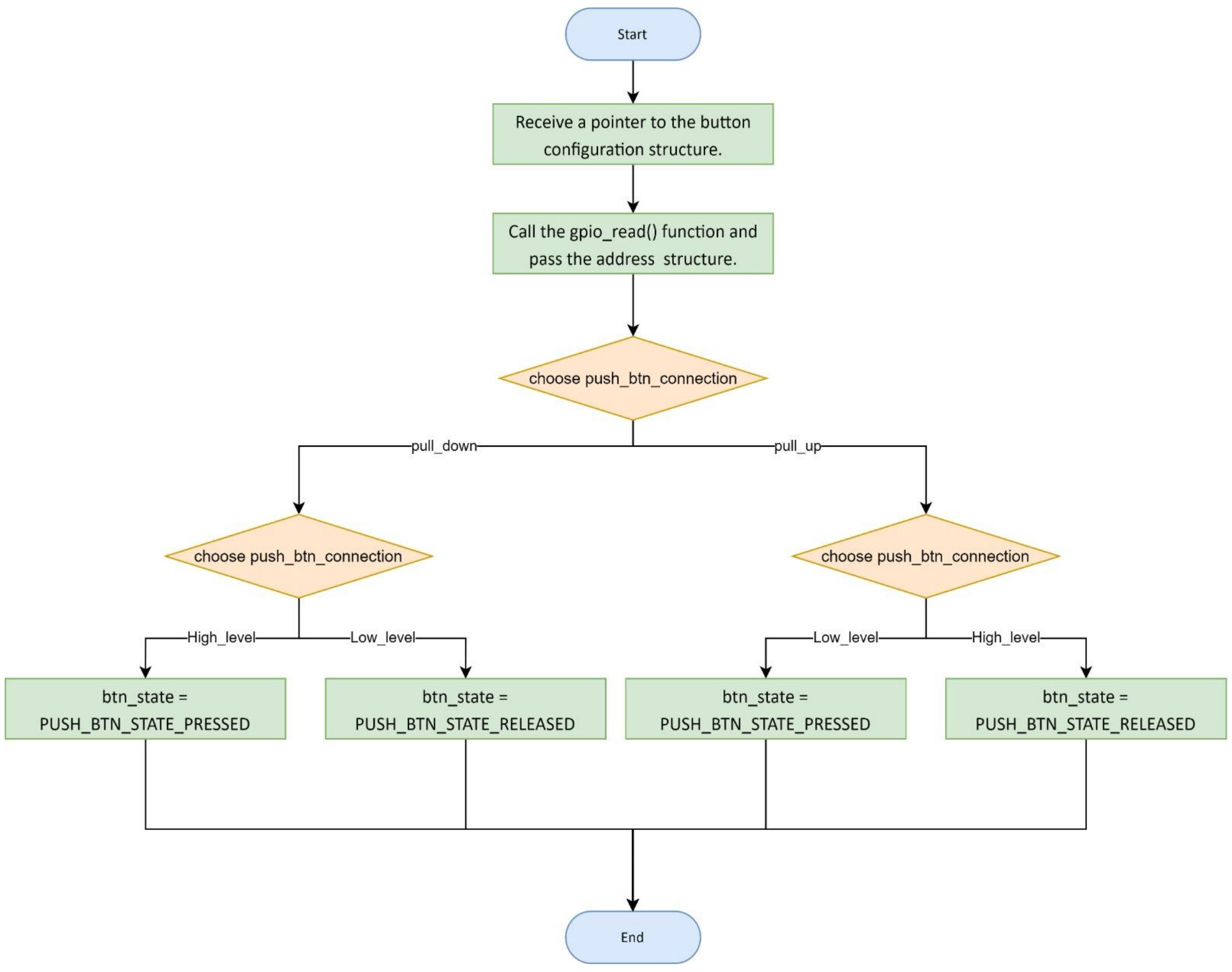

Start
Receive a pointer to the led configuration structure.
Call the Port_toggle() function
End

3.1.3 - PUSH_BUTTON Flow Charts

3.1.3.1 - PUSH_BUTTON_INIT

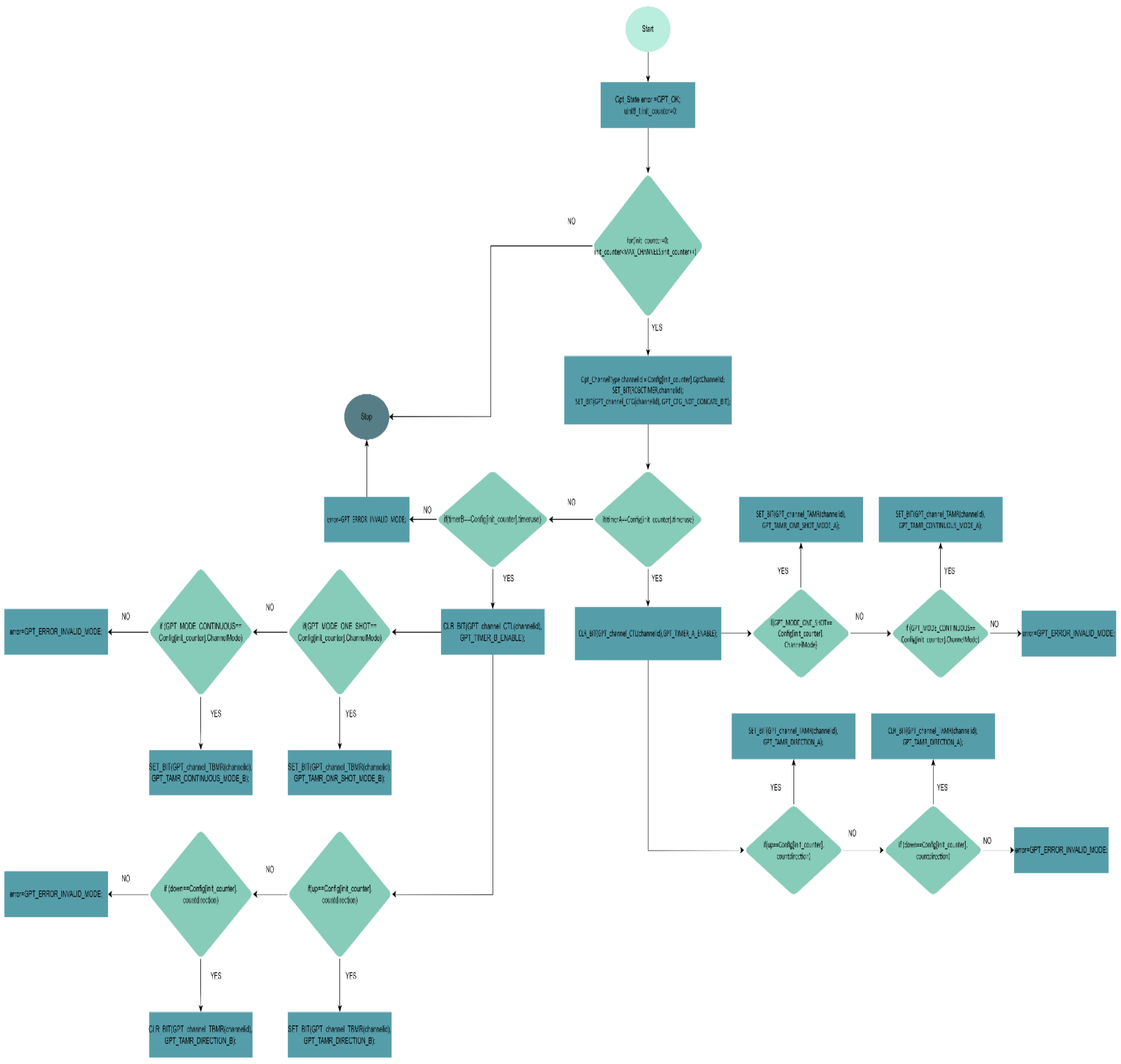

Start
Receive a pointer to the button configuration structure.
Call the Port_Init() function and pass the address structure.
End

3.1.3.2 - PUSH_BUTTON_READ

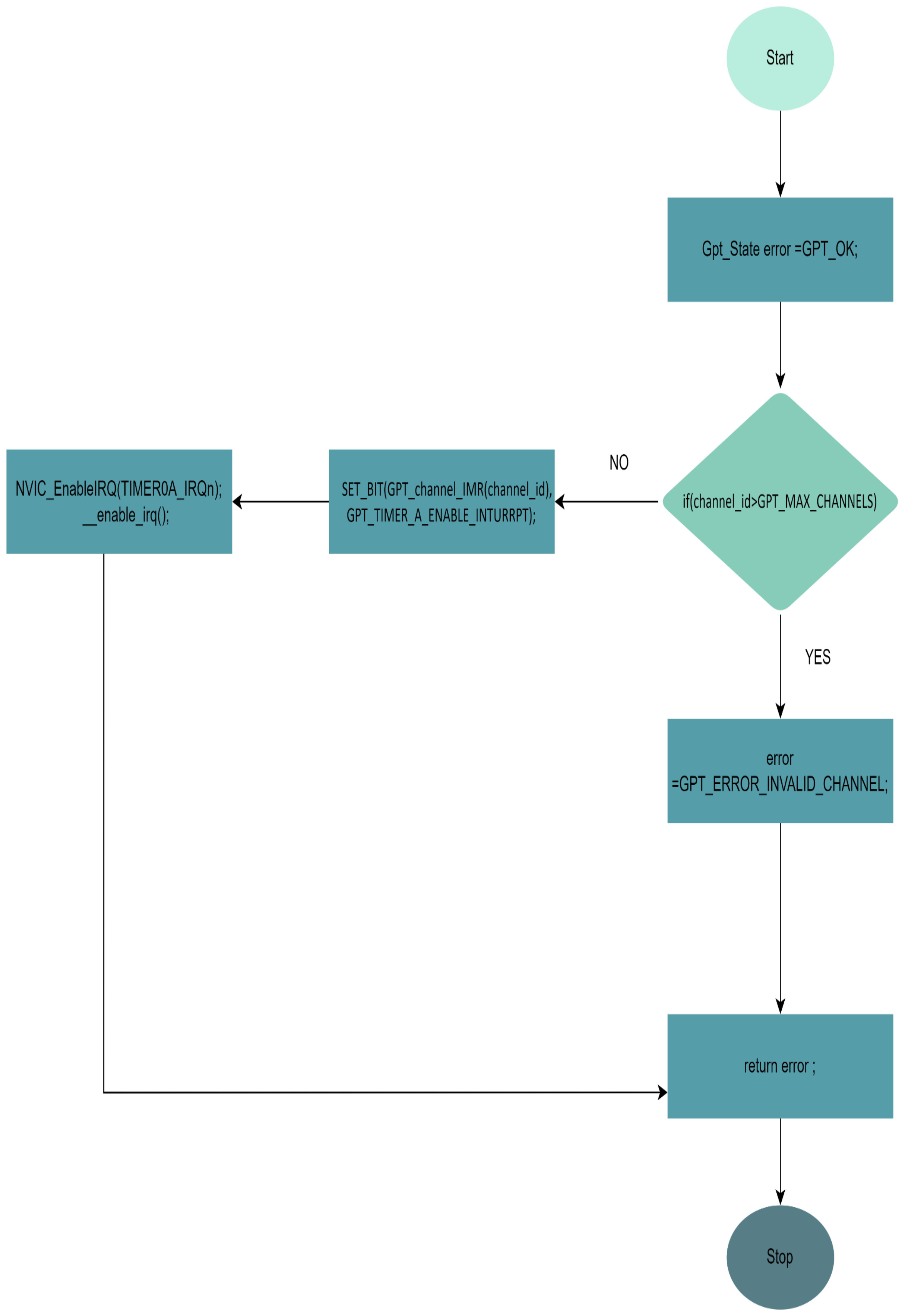

Start
Receive a pointer to the button configuration structure.
Call the gpio_read() function and pass the address structure.
choose push_btn_connection
pull_down
pull_up
choose push_btn_connection
choose push_btn_connection
High_level
Low_level
Low_level
High_level
btn_state = PUSH_BTN_STATE_PRESSED
btn_state = PUSH_BTN_STATE_RELEASED
btn_state = PUSH_BTN_STATE_PRESSED
btn_state = PUSH_BTN_STATE_RELEASED
End

3.1.4 - TIMER Flow Chart

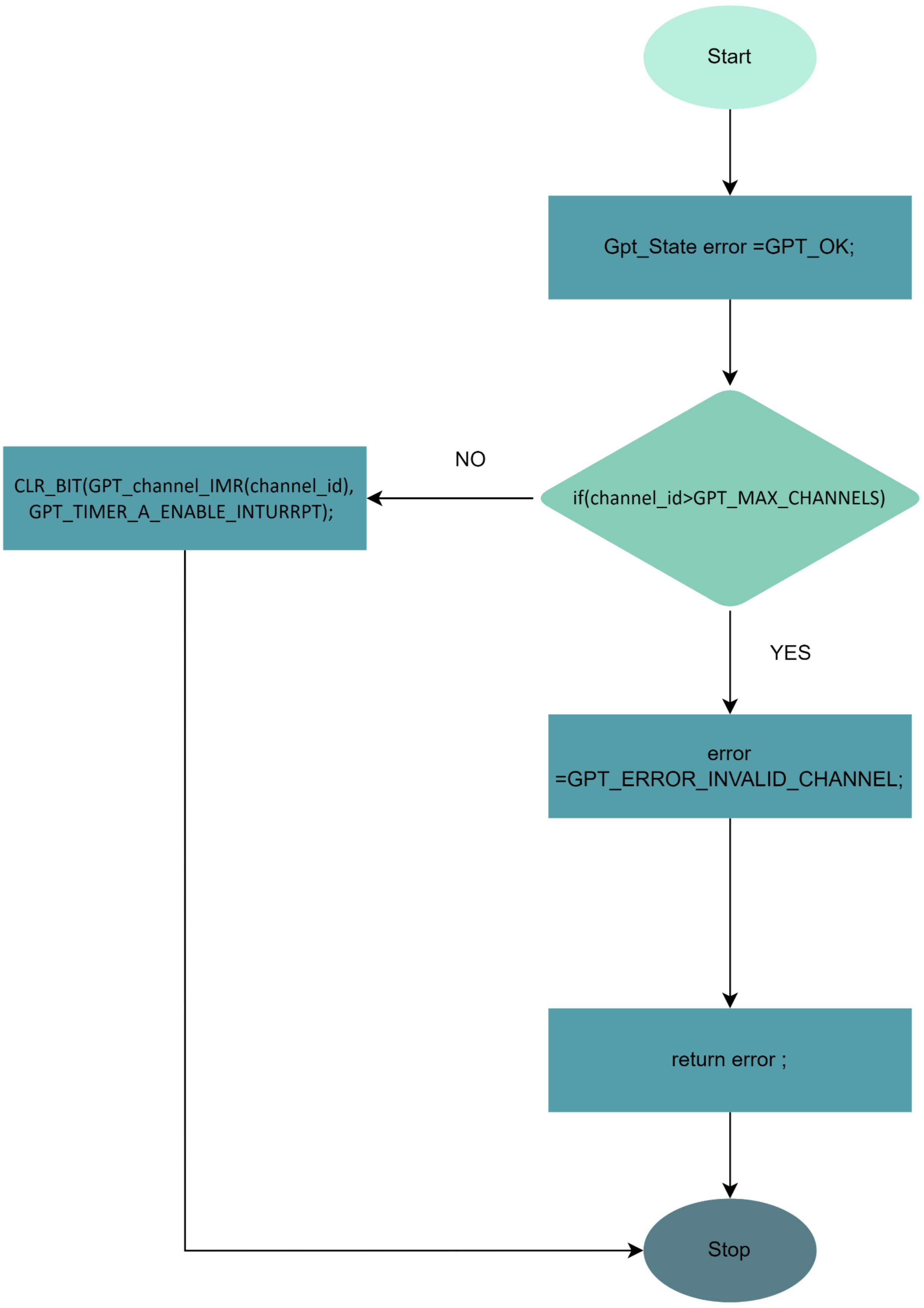
3.1.4.1 - Gpt_init


Start
Gpt_State error =GPT_OK;
uint8_t init_counter=0;
for(init_counter=0;
init_counter<MAX_CHANNELS;init_counter++)
NO
YES
Gpt_ChannelType channelid = Config[init_counter].GptChannelId;
SET_BIT(RCGCTIMER,channelid);
SET_BIT(GPT_channel_CFG(channelid), GPT_CFG_NOT_CONCATE_BIT);
Stop
if(timerA==Config[init_counter].timeruse)
NO
YES
if(timerB==Config[init_counter].timeruse)
NO
YES
error=GPT_ERROR_INVALID_MODE;
CLR_BIT(GPT_channel_CTL(channelid),GPT_TIMER_A_ENABLE);
if(GPT_MODE_ONE_SHOT==
Config[init_counter].
ChannelMode)
YES
SET_BIT(GPT_channel_TAMR(channelid),
GPT_TAMR_ONR_SHOT_MODE_A);
NO
if (GPT_MODE_CONTINUOUS==
Config[init_counter].ChannelMode)
YES
SET_BIT(GPT_channel_TAMR(channelid),
GPT_TAMR_CONTINUOUS_MODE_A);
NO
error=GPT_ERROR_INVALID_MODE;
if(up==Config[init_counter].
countdirection)
YES
SET_BIT(GPT_channel_TAMR(channelid),
GPT_TAMR_DIRECTION_A);
NO
if (down==Config[init_counter].
countdirection)
YES
CLR_BIT(GPT_channel_TAMR(channelid),
GPT_TAMR_DIRECTION_A);
NO
error=GPT_ERROR_INVALID_MODE;
CLR_BIT(GPT_channel_CTL(channelid),
GPT_TIMER_B_ENABLE);
if(GPT_MODE_ONE_SHOT==
Config[init_counter].ChannelMode)
YES
SET_BIT(GPT_channel_TBMR(channelid),
GPT_TAMR_ONR_SHOT_MODE_B);
NO
if (GPT_MODE_CONTINUOUS==
Config[init_counter].ChannelMode)
YES
SET_BIT(GPT_channel_TBMR(channelid),
GPT_TAMR_CONTINUOUS_MODE_B);
NO
error=GPT_ERROR_INVALID_MODE;
if(up==Config[init_counter].
countdirection)
YES
SET_BIT(GPT_channel_TBMR(channelid),
GPT_TAMR_DIRECTION_B);
NO
if (down==Config[init_counter].
countdirection)
YES
CLR_BIT(GPT_channel_TBMR(channelid),
GPT_TAMR_DIRECTION_B);
NO
error=GPT_ERROR_INVALID_MODE;

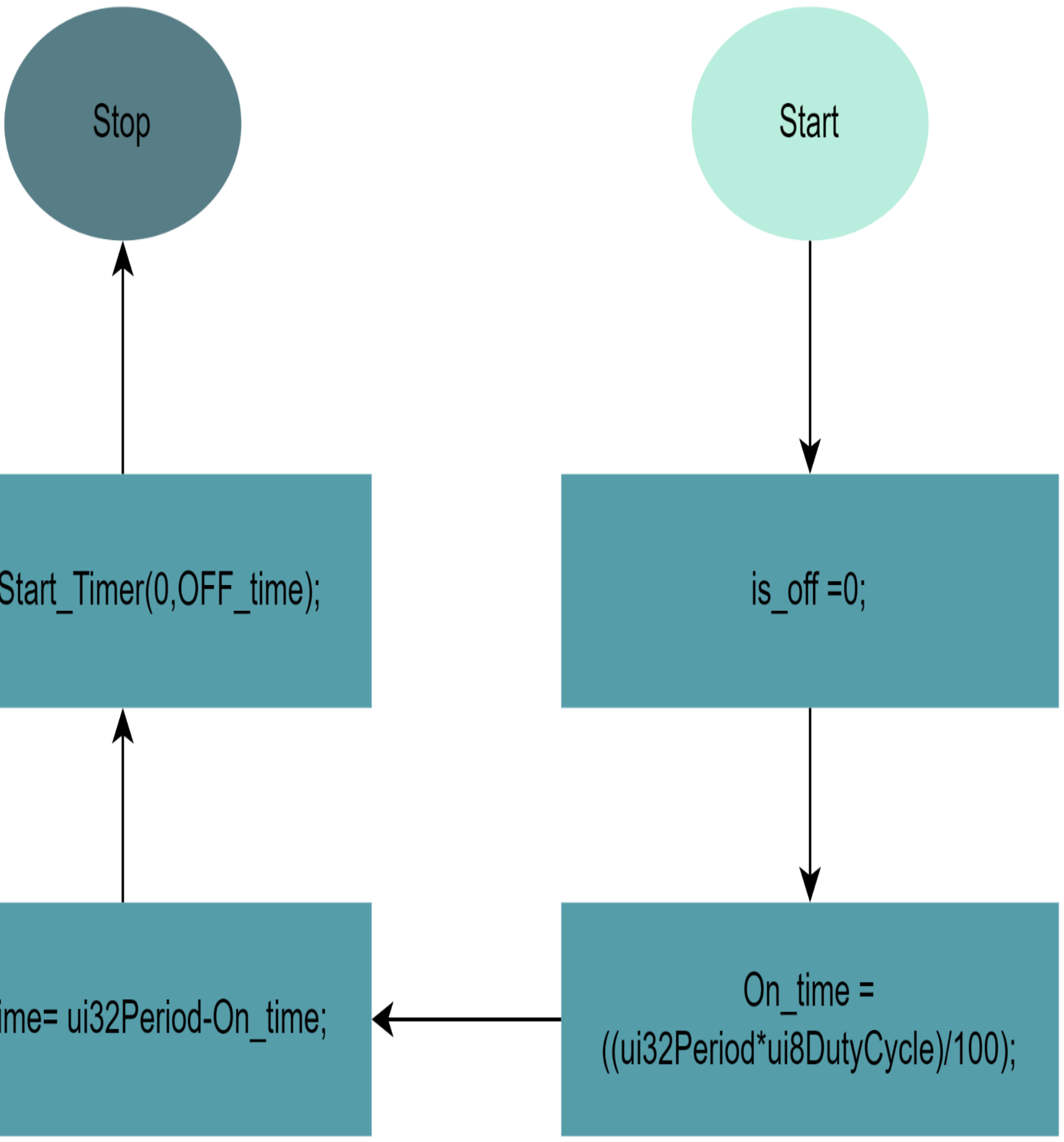
3.1.4.2 - GPT_enable_intturpt


Start
Gpt_State error =GPT_OK;
if(channel_id>GPT_MAX_CHANNELS)
NO
SET_BIT(GPT_channel_IMR(channel_id), GPT_TIMER_A_ENABLE_INTURRPT);
NVIC_EnableIRQ(TIMER0A_IRQn); __enable_irq();
YES
error =GPT_ERROR_INVALID_CHANNEL;
return error ;
Stop

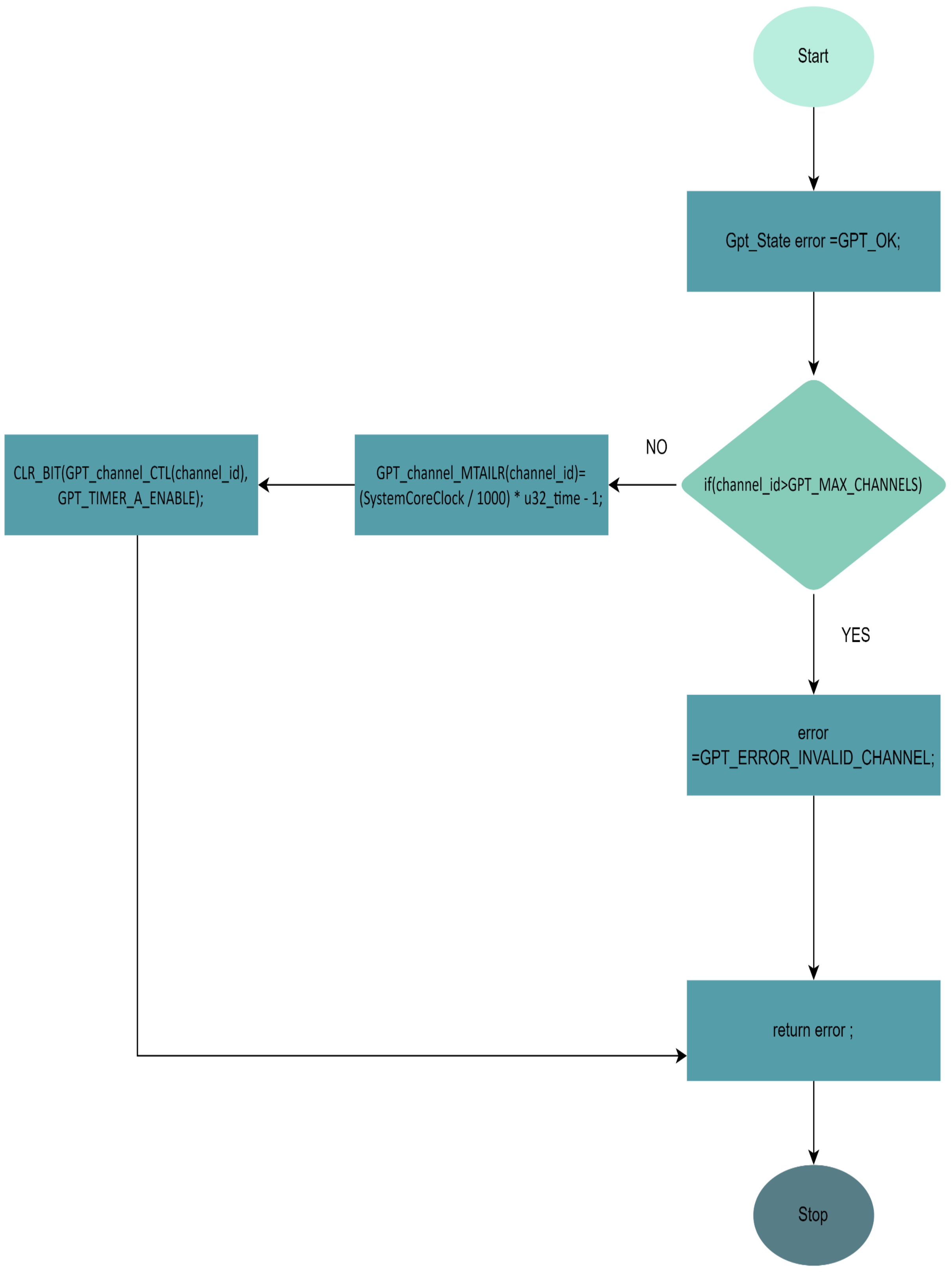
3.1.4.3 - GPT_disable_intturpt


Start
Gpt_State error =GPT_OK;
if(channel_id>GPT_MAX_CHANNELS)
NO
CLR_BIT(GPT_channel_IMR(channel_id), GPT_TIMER_A_ENABLE_INTURRPT);
YES
error =GPT_ERROR_INVALID_CHANNEL;
return error ;
Stop

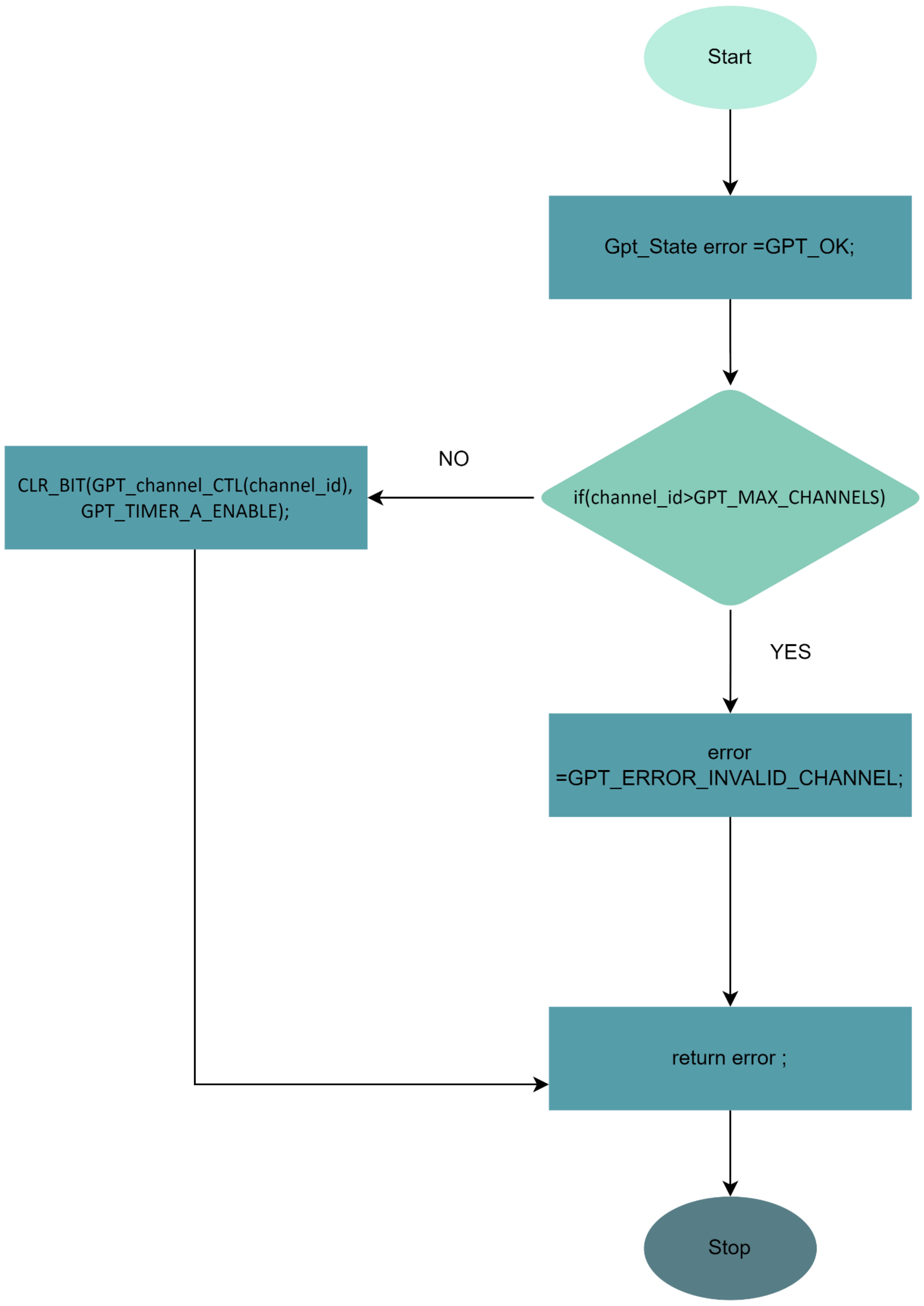
3.1.4.4 - PWM_Init


Start
is_off =0;
On_time = ((ui32Period*ui8DutyCycle)/100);
OFF_time= ui32Period-On_time;
GPT_Start_Timer(0,OFF_time);
Stop

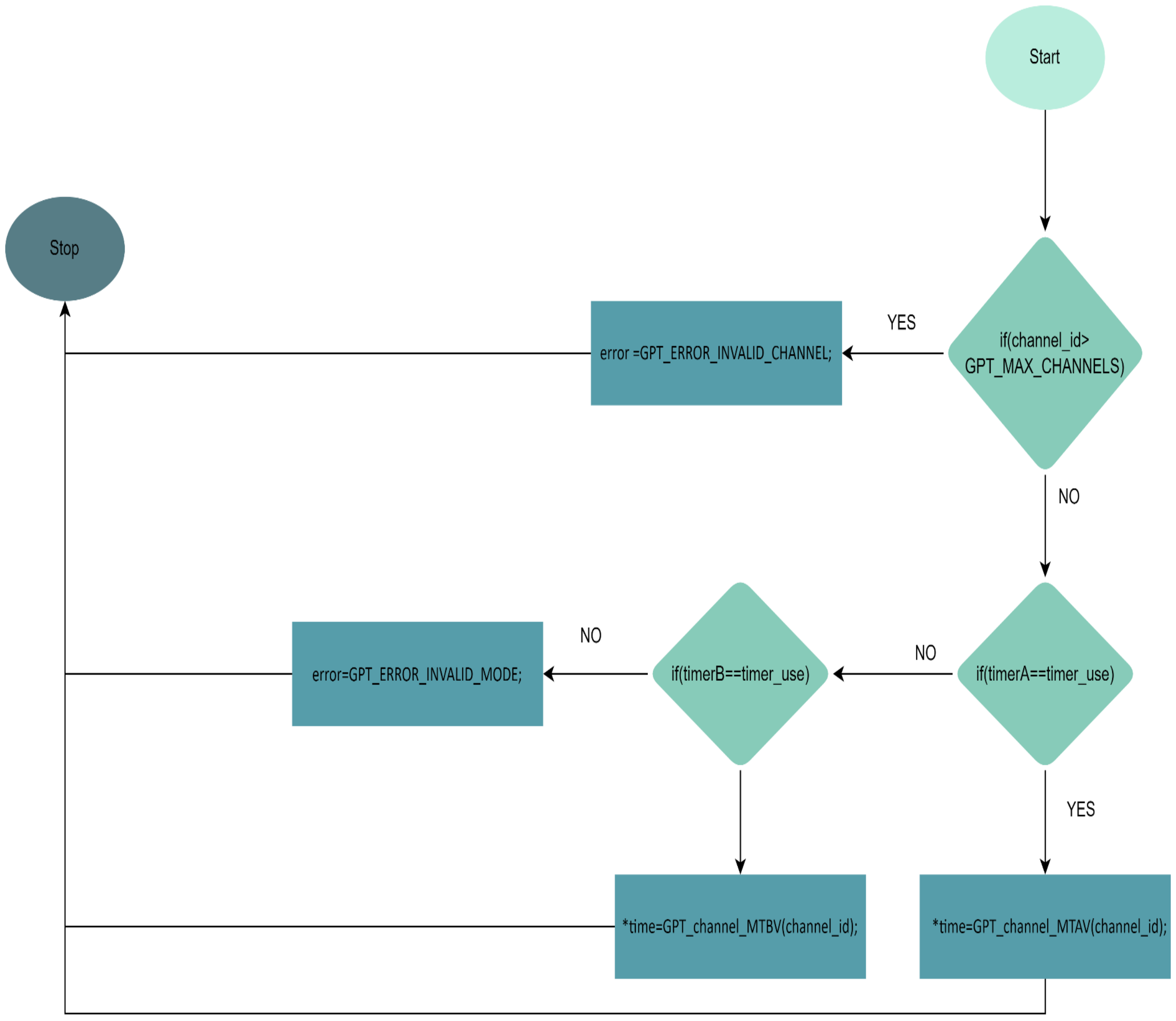
3.1.4.5 - GPT_Start_Timer


Start
Gpt_State error =GPT_OK;
if(channel_id>GPT_MAX_CHANNELS)
NO
GPT_channel_MTAILR(channel_id)= (SystemCoreClock / 1000) * u32_time - 1;
CLR_BIT(GPT_channel_CTL(channel_id), GPT_TIMER_A_ENABLE);
YES
error =GPT_ERROR_INVALID_CHANNEL;
return error ;
Stop

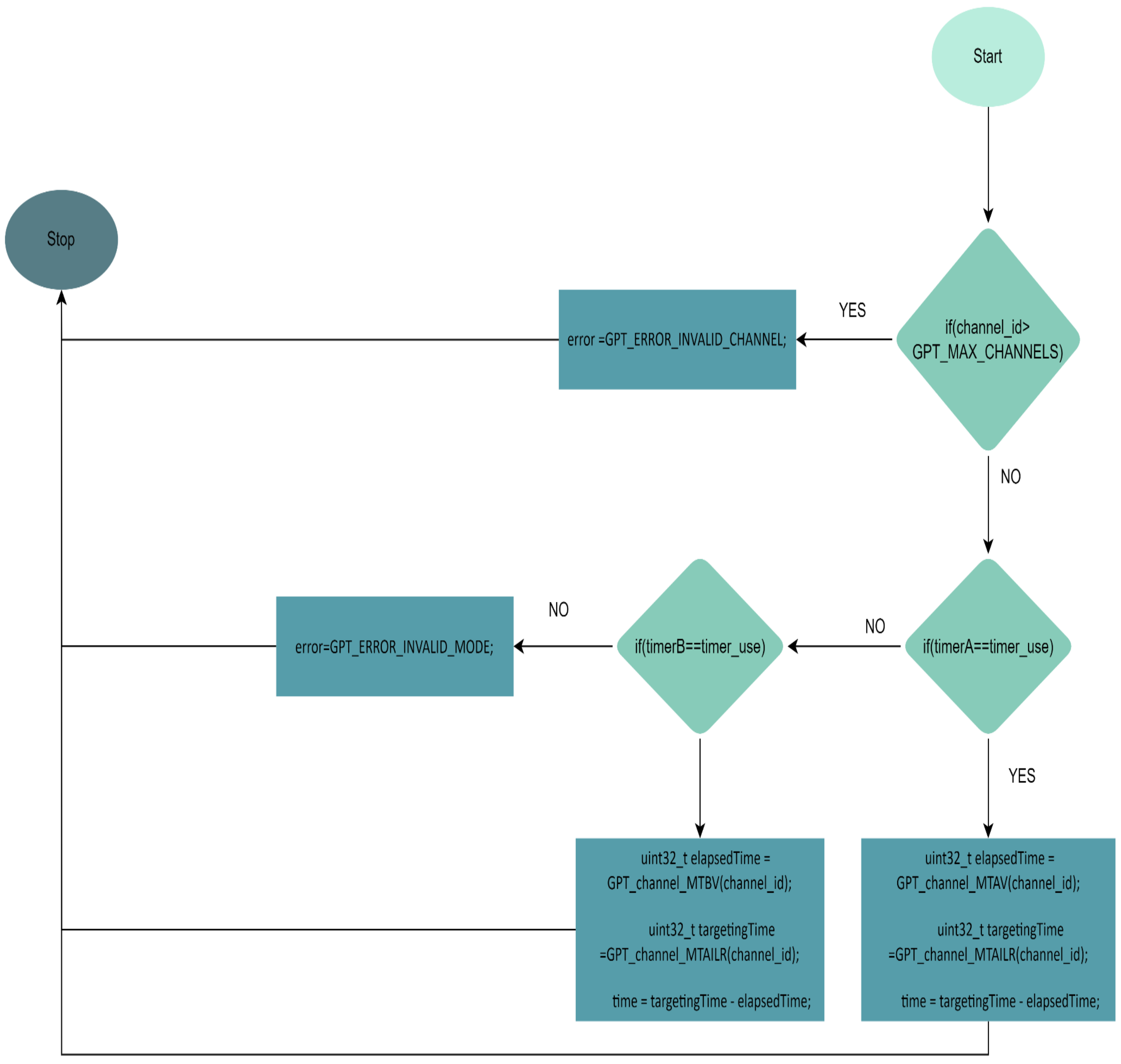
3.1.4.6 - GPT_STOP_Timer


Start
Gpt_State error =GPT_OK;
if(channel_id>GPT_MAX_CHANNELS)
NO
CLR_BIT(GPT_channel_CTL(channel_id), GPT_TIMER_A_ENABLE);
YES
error =GPT_ERROR_INVALID_CHANNEL;
return error ;
Stop

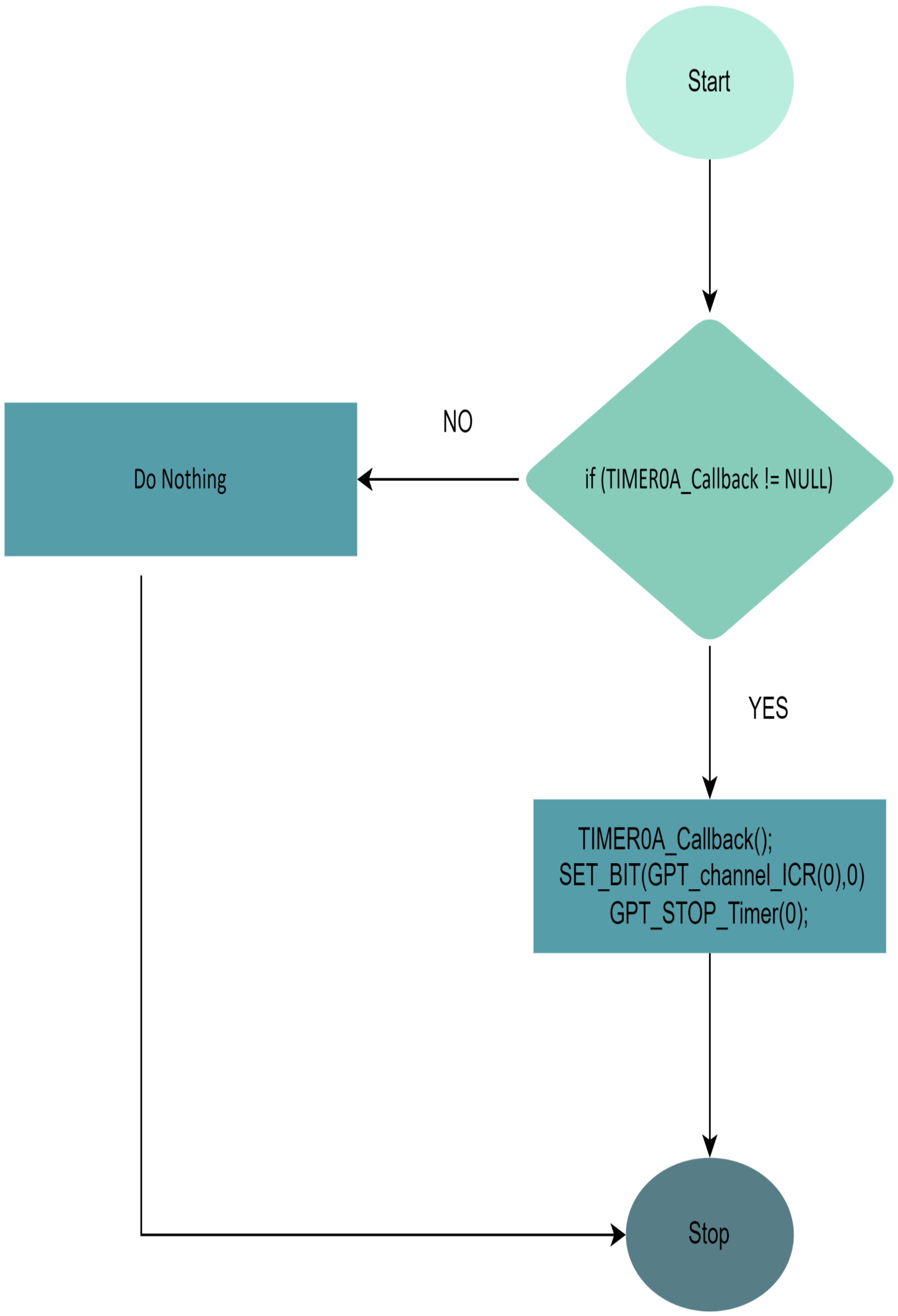
3.1.4.7 - GPT_GetElapsedTime


Start
if(channel_id> GPT_MAX_CHANNELS)
YES
error =GPT_ERROR_INVALID_CHANNEL;
Stop
NO
if(timerA==timer_use)
NO
if(timerB==timer_use)
NO
error=GPT_ERROR_INVALID_MODE;
YES
*time=GPT_channel_MTBV(channel_id);
*time=GPT_channel_MTAV(channel_id);

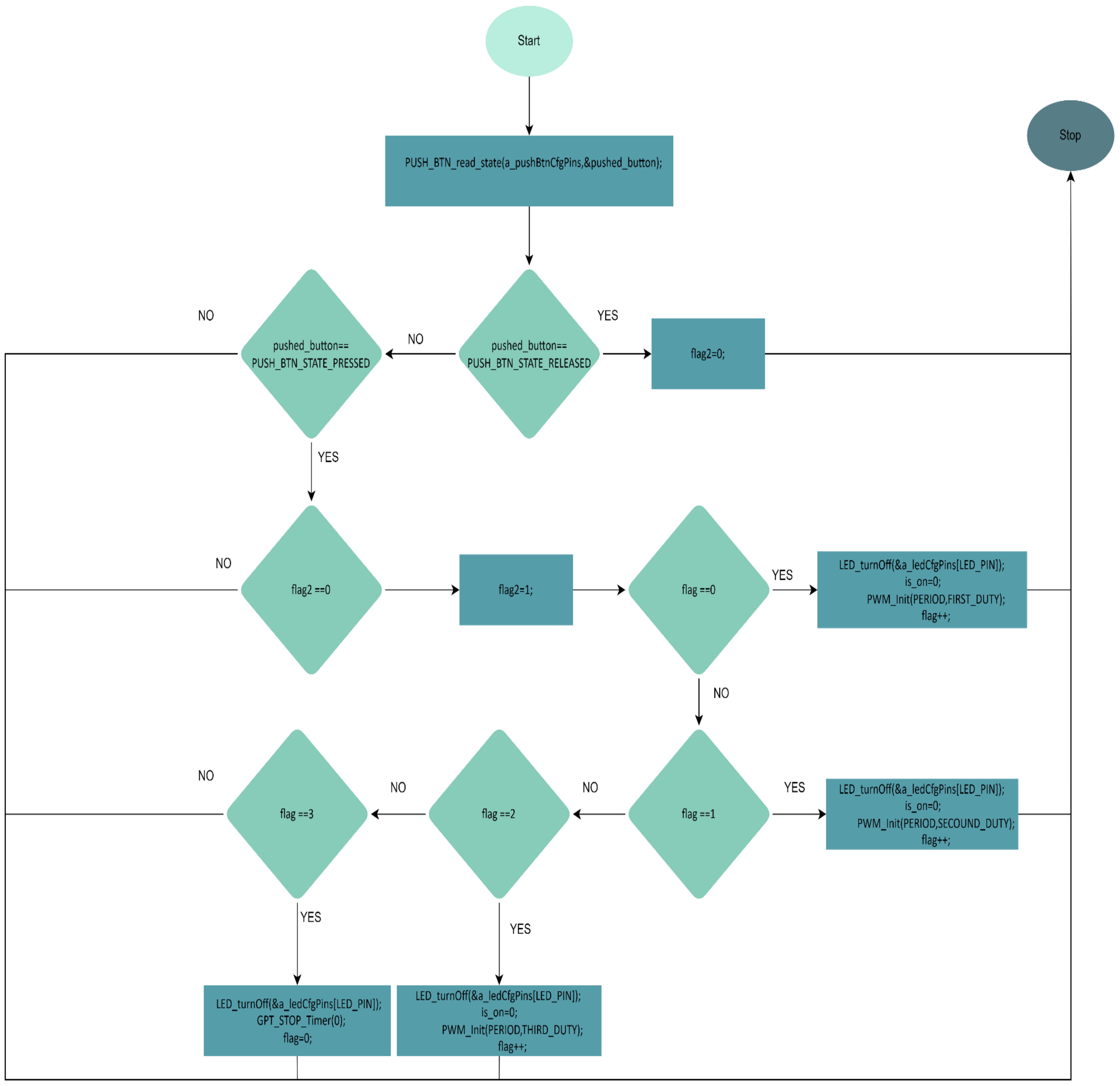
3.1.4.8 - GPT_GetRemainingTime


Start
if(channel_id> GPT_MAX_CHANNELS)
YES
error =GPT_ERROR_INVALID_CHANNEL;
NO
if(timerA==timer_use)
NO
if(timerB==timer_use)
NO
error=GPT_ERROR_INVALID_MODE;
YES
uint32_t elapsedTime = GPT_channel_MTBV(channel_id);
uint32_t targetingTime =GPT_channel_MTAILR(channel_id);
time = targetingTime - elapsedTime;
uint32_t elapsedTime = GPT_channel_MTAV(channel_id);
uint32_t targetingTime =GPT_channel_MTAILR(channel_id);
time = targetingTime - elapsedTime;
Stop

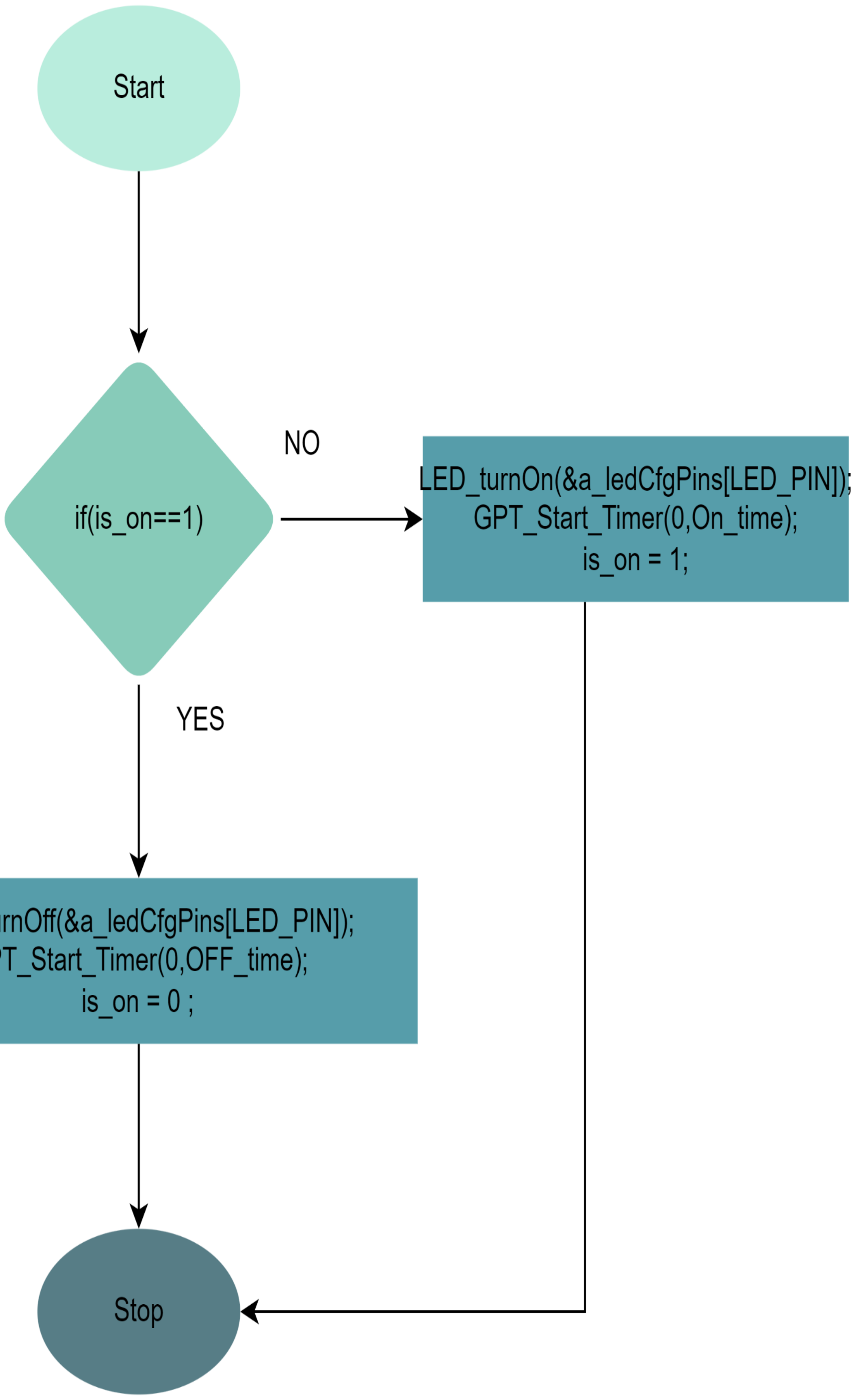
3.1.4.9 - TIMER0A_HANDLER



3.1.5 - MAIN Flow Charts


Start
PUSH_BTN_read_state(a_pushBtnCfgPins,&pushed_button);
Stop
NO
pushed_button==
PUSH_BTN_STATE_PRESSED
NO
pushed_button==
PUSH_BTN_STATE_RELEASED
YES
flag2=0;
YES
NO
flag2 ==0
flag2=1;
flag ==0
YES
LED_turnOff(&a_ledCfgPins[LED_PIN]);
is_on=0;
PWM_Init(PERIOD,FIRST_DUTY);
flag++;
NO
NO
flag ==3
NO
flag ==2
NO
flag ==1
YES
LED_turnOff(&a_ledCfgPins[LED_PIN]);
is_on=0;
PWM_Init(PERIOD,SECOUND_DUTY);
flag++;
YES
YES
LED_turnOff(&a_ledCfgPins[LED_PIN]);
GPT_STOP_Timer(0);
flag=0;
LED_turnOff(&a_ledCfgPins[LED_PIN]);
is_on=0;
PWM_Init(PERIOD,THIRD_DUTY);
flag++;

3.1.6 - TIMER0A_CALLBACK Flowchart


Start
if(is_on==1)
NO
LED_turnOn(&a_ledCfgPins[LED_PIN]);
GPT_Start_Timer(0,On_time);
is_on = 1;
YES
LED_turnOff(&a_ledCfgPins[LED_PIN]);
GPT_Start_Timer(0,OFF_time);
is_on = 0 ;
Stop

3.2 - Pre Compiling Files

3.2.1 - GPIO Driver

```
#define PORT_PINS_NUMS 1
```

3.2.2 - LED Driver

```
#define  LED_PIN_CFG_ARRAY_SIZE    3
```

3.2.3 - PUSH_BUTTON Driver

```
#define  PUSH_BTN_PIN_CFG_ARRAY_SIZE    1
```

3.2.4 - TIMER Driver

```
#define MAX_CHANNELS 1
```

3.3 - Pre Linking Configuration

3.3.1 - GPIO Driver

None

3.3.2 - LED Driver

```
ST_led_pinCfg_t a_ledCfgPins[LED_PIN_CFG_ARRAY_SIZE] =
{
                    {PORTF , PIN_1 , PORT_PIN_LEVEL_LOW},
                    {PORTF , PIN_2 , PORT_PIN_LEVEL_LOW},
                    {PORTF , PIN_3 , PORT_PIN_LEVEL_LOW}

};
```

3.3.3 - PUSH_BUTTON Driver

```
ST_PUSH_BTN_pinCfg_t a_pushBtnCfgPins[PUSH_BTN_PIN_CFG_ARRAY_SIZE] =
{
  {PORTF , PIN_4 , PORT_PIN_PUR}
};
```

3.3.4 - TIMER Driver

```
Gpt_ConfigType STR_GPTConfig[MAX_CHANNELS]=
{
{GPT_CHANNEL_0,GPT_MODE_ONE_SHOT,timerA,down}
};
```

FOR FLOW CHART WITH HIGH QUALITY IT IS ON GITHUB