SPRINTS

RGB LED Control Design

Version 1.0

June 2023

Presented by
Abdelrhman Walaa

Presented to
Sprints

Table Of Content

RGB LED Control V1.0 Design

1. Project Introduction

The RGB LED Control project aims to demonstrate the capabilities of the Tiva C TM4C123GXL LaunchPad board by implementing a simple user interface using the SW1 button. The project enables users to control the color and behavior of an RGB LED by pressing the SW1 button on the board. By combining the power of the Tiva C microcontroller and the versatility of the RGB LED, this project offers an interactive and visually engaging experience.

1.1. System Requirements

1.1.1. Hardware Requirements

1. **TivaC** board
2. **One** input button **SW1**
3. **One** RGB **LED**

1.1.2. Software Requirements

1. The RGB **LED** is **off** initially.
2. Pressing **SW1**:

    a. After the **first** press, the **Red** led is **on**.
    b. After the **second** press, the **Green** Led is **on**.
    c. After the **third** press, the **Blue** led is **on**.
    d. After the **fourth** press, all **LEDs** are **on**.
    e. After the **fifth** press, should disable all **LEDs**.
    f. After the **sixth** press, **repeat** steps from 1 to 6.

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

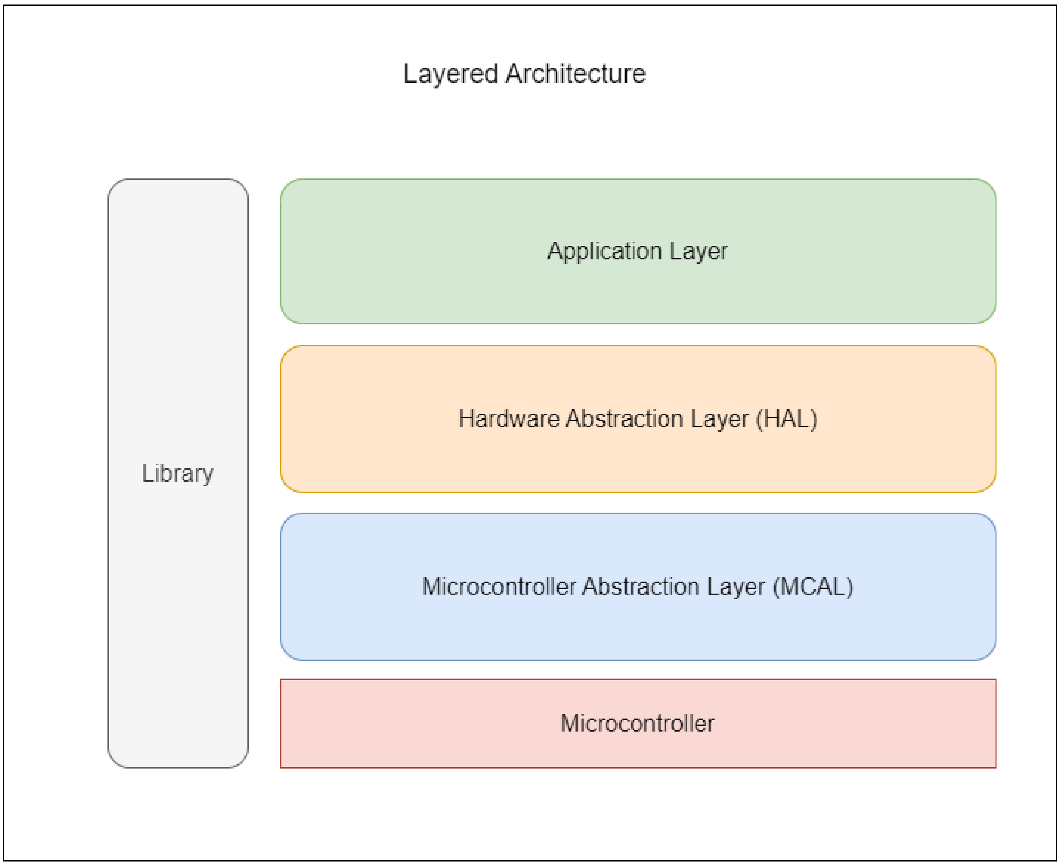


Figure 1. Layered Architecture Design

2.2. System Modules

2.2.1. 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.

2.2.2. Design

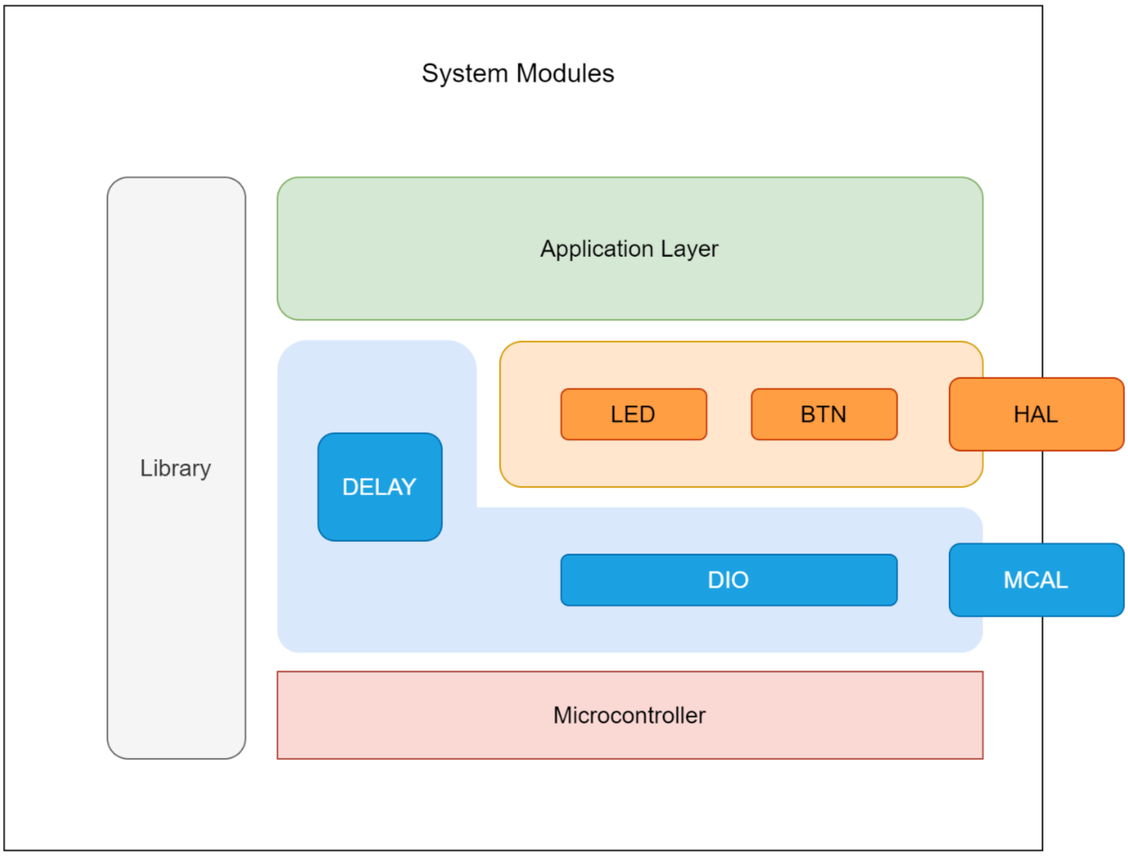


Figure 2. System Modules Design

2.3. Modules Description

2.3.1. 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.

2.3.2. DELAY Module

A simple *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.

2.3.3. LED Module

An *LED* (Light Emitting Diode) driver is a crucial component in microcontroller projects that involve controlling *LED*s. It provides a means to control the brightness, color, and behavior of *LEDs*, allowing for dynamic visual effects and signaling. A well-designed *LED* driver simplifies the interfacing process and enables efficient and precise control over *LED* operations.

2.3.4. BTN Module

In most of the embedded electronic projects you may want to use a *BTN* (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. Drivers' Documentation (APIs)

2.4.1 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.

2.4.2. MCAL APIs

2.4.2.1. GPIO Driver APIs

```
Precompile and LinKing Configurations:

/* GPIO Port Ids */
typedef enum
{
  GPIO_EN_PORTA = 0,
  GPIO_EN_PORTC,
  GPIO_EN_PORTD,
  GPIO_EN_PORTE,
  GPIO_EN_PORTF,
  GPIO_EN_INVALID_PORT_ID
} GPIO_enPortId_t;

/* GPIO Port Bus Ids */
typedef enum
{
  GPIO_EN_APB = 0,
  GPIO_EN_AHB,
  GPIO_EN_INVALID_BUS_ID
} GPIO_enPortBusId_t;

/* GPIO Port Mode */
typedef enum
{
  GPIO_EN_RUN_MODE = 0,
  GPIO_EN_SLEEP_MODE,
  GPIO_EN_DEEP_SLEEP_MODE,
  GPIO_EN_INVALID_PORT_MODE
} GPIO_enPortMode_t;
```

```c
/* GPIO Pin Ids */
typedef enum
{
  GPIO_EN_PIN0 = 0,
  GPIO_EN_PIN1,
  GPIO_EN_PIN2,
  GPIO_EN_PIN3,
  GPIO_EN_PIN4,
  GPIO_EN_PIN5,
  GPIO_EN_PIN6,
  GPIO_EN_PIN7,
  GPIO_EN_INVALID_PIN_ID
} GPIO_enPinId_t;

/* GPIO Pin Mode */
typedef enum
{
  GPIO_EN_DISABLE_PIN = 0,
  GPIO_EN_ENABLE_PIN,
  GPIO_EN_INVALID_PIN_MODE
} GPIO_enPinMode_t;

/* GPIO Pin Direction */
typedef enum
{
  GPIO_EN_INPUT_DIR = 0,
  GPIO_EN_OUTPUT_DIR,
  GPIO_EN_INVALID_PIN_DIR
} GPIO_enPinDirection_t;

/* GPIO Pin Value */
typedef enum
{
  GPIO_EN_PIN_LOW,
  GPIO_EN_PIN_HIGH,
  GPIO_EN_INVALID_PIN_VALUE
} GPIO_enPinValue_t;

/* GPIO Pin Pad */
typedef enum
{
  GPIO_EN_DISABLE_PIN_PAD =0,
  GPIO_EN_ENABLE_PULL_UP,
  GPIO_EN_ENABLE_PULL_DOWN,
  GPIO_EN_ENABLE_OPEN_DRAIN,
  GPIO_EN_INVALID_PIN_PAD
} GPIO_enPinPad_t;
```

```c
/* GPIO Pin Alternate Function Mode */
typedef enum
{
  GPIO_EN_PIN_GPIO_MODE = 0,
  GPIO_EN_PIN_ALT_MODE,
  GPIO_EN_INVALID_PIN_ALT_MODE
} GPIO_enPinAlternateMode_t;

/* GPIO Pin Drive */
typedef enum
{
  GPIO_EN_DISABLE_PIN_DRIVE = 0,
  GPIO_EN_2_MA_DRIVE,
  GPIO_EN_4_MA_DRIVE,
  GPIO_EN_8_MA_DRIVE,
  GPIO_EN_INVALID_PIN_DRIVE
} GPIO_enPinDrive_t;

/* GPIO Interrupt Modes */
typedef enum
{
  GPIO_EN_DISABLE_INT = 0,
  GPIO_EN_ENABLE_INT,
  GPIO_EN_INVALID_INT_MODE
} GPIO_enPinInterruptMode_t;

/* GPIO Interrupt Action */
typedef void (*GPIO_vpfInterruptAction_t) (void);

/* GPIO Port Linking Configurations Structure */
typedef struct
{
  GPIO_enPortId_t      en_a_portId;
  GPIO_enPortBusId_t   en_a_portBusId;
  GPIO_enPortMode_t    en_a_portMode;
} GPIO_stPortLinkConfig_t;

/* GPIO Pin Linking Configurations Structure */
typedef struct
{
  GPIO_enPortId_t              en_a_portId;
  GPIO_enPortBusId_t           en_a_portBusId;
  GPIO_enPinId_t               en_a_pinId;
  GPIO_enPinMode_t             en_a_pinMode;
  GPIO_enPinDirection_t        en_a_pinDirection;
  GPIO_enPinValue_t            en_a_pinValue;
  GPIO_enPinAlternateMode_t    en_a_pinAlternateMode;
  GPIO_enPinDrive_t            en_a_pinDrive;
  GPIO_enPinPad_t              en_a_pinPad;
  GPIO_enPinInterruptMode_t    en_a_pinInterruptMode;
```

```
  GPIO_vpfInterruptAction_t   en_a_pinInterruptAcrion;
} GPIO_stPinLinkConfig_t;

| Name: GPIO_initalization
| Input: Pointer to Array of st PortLinkConfig and u8 NumberOfPorts
| Output: en Error or No Error
| Description: Function to initialize GPIO Port peripheral using Linking
|              Configurations.
|
GPIO_enErrorState_t GPIO_initalization (const GPIO_stPortLinkConfig_t
*past_a_portsLinkConfig, u8 u8_a_numberOfPorts)

| Name: GPIO_configurePin
| Input: Pointer to Array of st PinLinkConfig and u8 NumberOfPins
| Output: en Error or No Error
| Description: Function to configure GPIO Pin peripheral using Linking
|              Configurations.
|
GPIO_enErrorState_t GPIO_configurePin (const GPIO_stPinLinkConfig_t
*past_a_pinsLinkConfig, u8 u8_a_numberOfPins)

| Name: GPIO_setPinValue
| Input: en PortId, en BusId, and en PinId
| Output: en Error or No Error
| Description: Function to set Pin value.
|
GPIO_enErrorState_t GPIO_setPinValue (GPIO_enPortId_t en_a_portId,
GPIO_enPortBusId_t en_a_busId, GPIO_enPinId_t en_a_pinId)

| Name: DIO_u8GetPinValue
| Input: u8 PortId, u8 PinId, and Pointer to u8 ReturnedPinValue
| Output: u8 Error or No Error
| Description: Function to get Pin value.
|
GPIO_enErrorState_t GPIO_getPinValue (GPIO_enPortId_t en_a_portId,
GPIO_enPortBusId_t en_a_busId, GPIO_enPinId_t en_a_pinId, GPIO_enPinValue_t
*pen_a_returnedPinValue)

| Name: GPIO_clearPinValue
| Input: en PortId, en BusId, and en PinId
| Output: en Error or No Error
| Description: Function to clear Pin value.
|
GPIO_enErrorState_t GPIO_clearPinValue (GPIO_enPortId_t en_a_portId,
GPIO_enPortBusId_t en_a_busId, GPIO_enPinId_t en_a_pinId)
```

```
| Name: GPIO_togglePinValue
| Input: en PortId, en BusId, and en PinId
| Output: en Error or No Error
| Description: Function to toggle Pin value.
|
GPIO_enErrorState_t GPIO_togglePinValue (GPIO_enPortId_t en_a_portId,
GPIO_enPortBusId_t en_a_busId, GPIO_enPinId_t en_a_pinId)
```

2.4.2.2. DELAY Driver APIs

```
Precompile and Linking Configurations:

There is no need for Precompile Configurations nor Linking Configurations as
the defined APIs below could change the DELAY peripheral during the Runtime.

| Name: DELAY_simpleDelay
| Input: void
| Output: void
| Description: Function to delay a short period.
|
void DELAY_simpleDelay (void)
```

2.4.3. HAL APIs

2.4.3.1. LED Driver APIs

```
Precompile and Linking Configurations:

/* LED IDs Counted from 0 to 7 */
#define LED_U8_0          0
#define LED_U8_1          1
#define LED_U8_2          2
#define LED_U8_3          3
#define LED_U8_4          4
#define LED_U8_5          5
#define LED_U8_6          6
#define LED_U8_7          7

/* LED Operations Counted from 0 to 2 */
#define LED_U8_ON         0
#define LED_U8_OFF        1
#define LED_U8_TOGGLE     2

/* LED Error States */
typedef enum
{
  LED_EN_NOK = 0,
  LED_EN_OK
} LED_enErrorState_t;

| Name: LED_initialization
| Input: u8 LedId
| Output: en Error or No Error
| Description: Function to initialize LED peripheral, by initializing GPIO peripheral.
|
LED_enErrorState_t LED_initialization (u8 u8_a_ledId)

| Name: LED_setLEDPin
| Input: u8 LedId and u8 Operation
| Output: en Error or No Error
| Description: Function to switch LED on, off, or toggle.
|
LED_enErrorState_t LED_setLEDPin( u8 u8_a_ledId, u8 u8_a_operation )
```

2.4.3.2. BTN Driver APIs

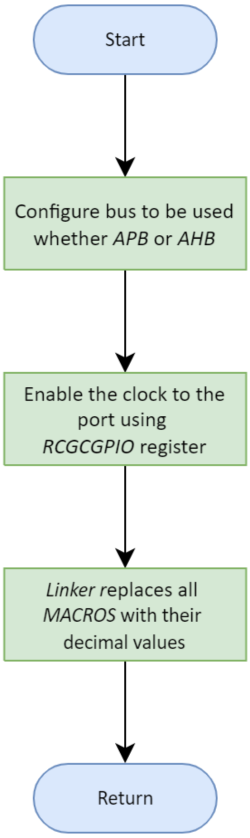
```
Precompile and Linking Configurations Snippet:

/* BTN IDs Counted from 0 to 7 */
#define BTN_U8_0          0
#define BTN_U8_1          1
#define BTN_U8_2          2
#define BTN_U8_3          3
#define BTN_U8_4          4
#define BTN_U8_5          5
#define BTN_U8_6          6
#define BTN_U8_7          7

/* BTN Values Counted from 0 to 1 */
#define BTN_U8_LOW        0
#define BTN_U8_HIGH       1

/* BTN Error States */
typedef enum
{
  BTN_EN_NOK = 0,
  BTN_EN_OK
} BTN_enErrorState_t;

| Name: BTN_initialization
| Input: u8 BTNId
| Output: en Error or No Error
| Description: Function to initialize BTN peripheral, by initializing GPIO peripheral.
|
BTN_enErrorState_t BTN_initialization (u8 u8_a_btnId)

| Name: BTN_getBTNState
| Input: u8 BTNId and Pointer to u8 ReturnedBTNState
| Output: en Error or No Error
| Description: Function to get BTN state.
|
BTN_enErrorState_t BTN_getBTNState (u8 u8_a_btnId, u8 *pu8_a_returnedBTNState)
```

3. Low Level Design

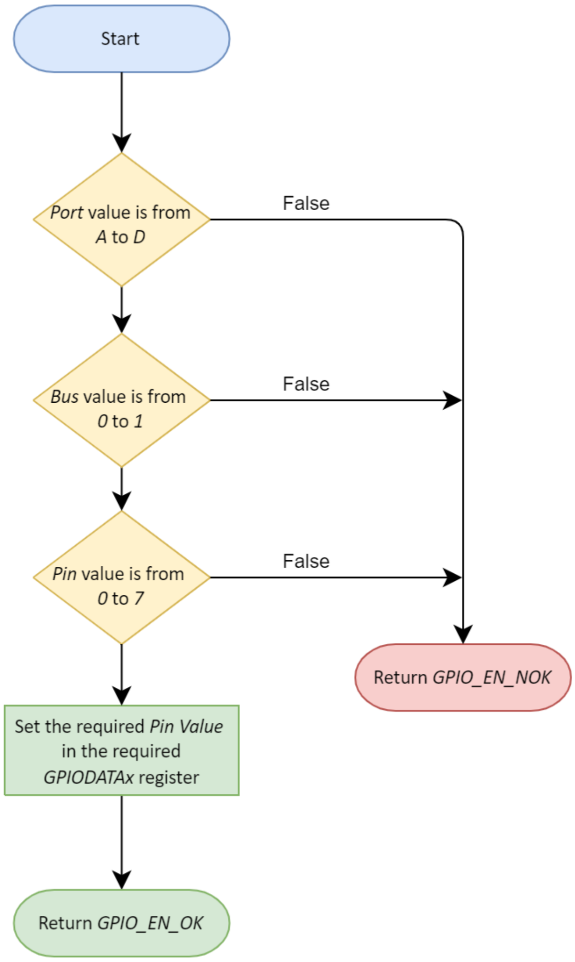
3.1. MCAL Layer

3.1.1. GPIO Module

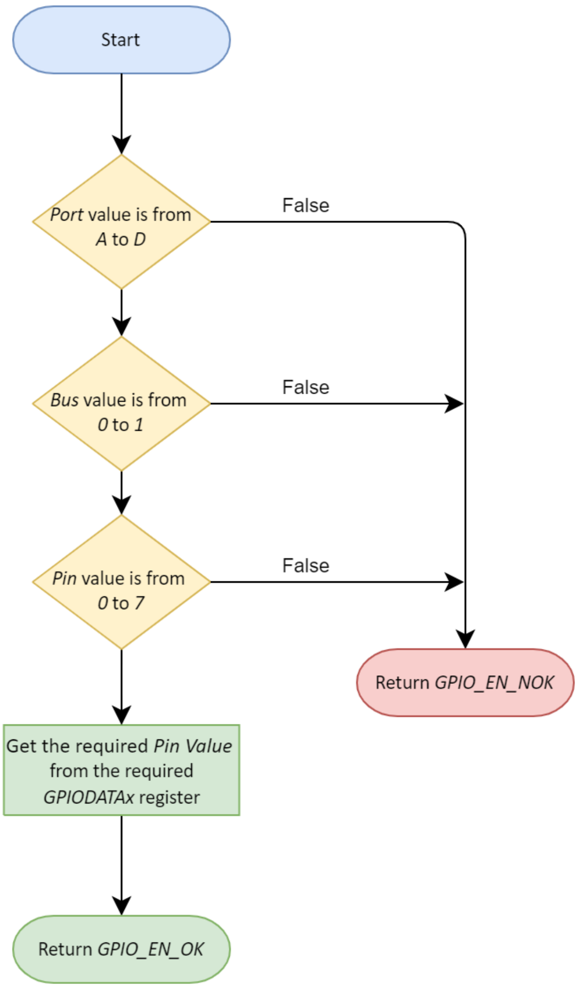
A. GPIO_initialization



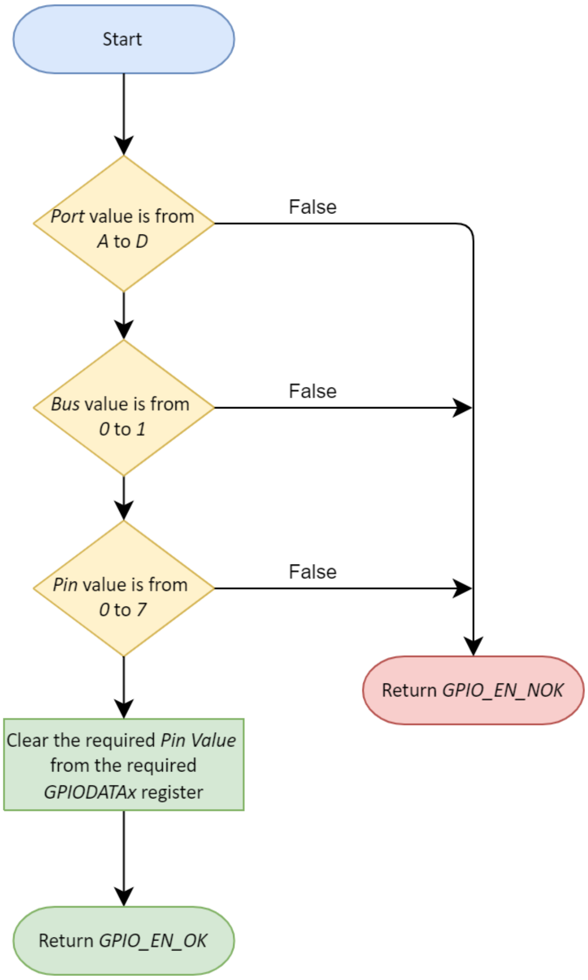
B. *GPIO_setPinValue*



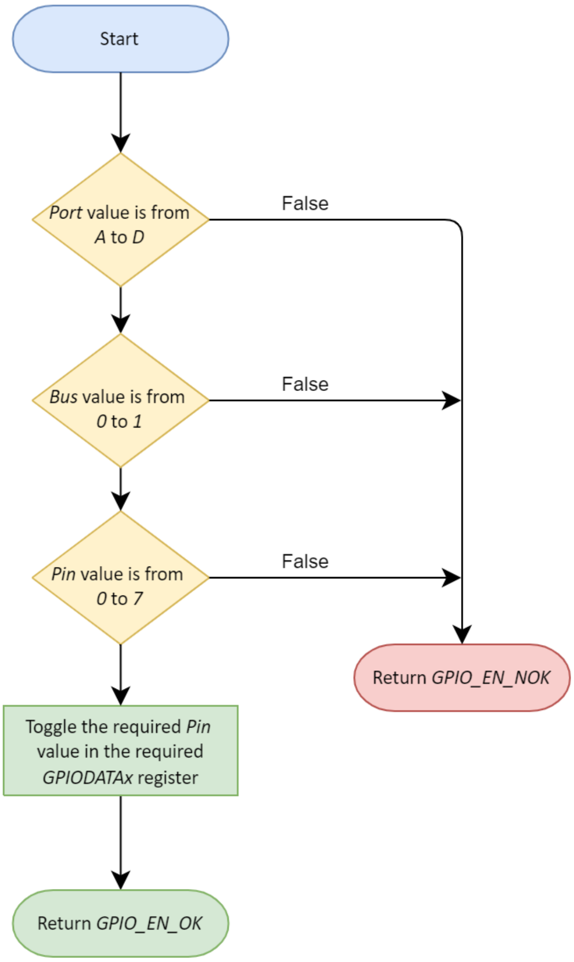
C. GPIO_getPinValue



D. GPIO_clearPinValue



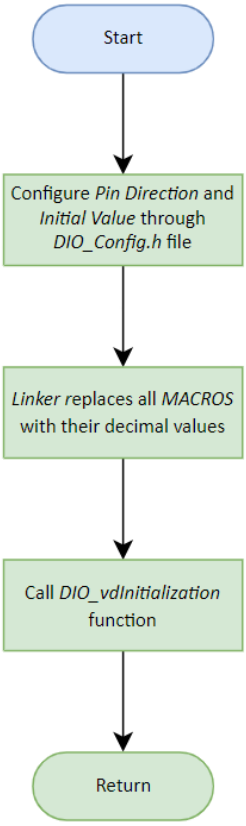
E. GPIO_togglePinValue



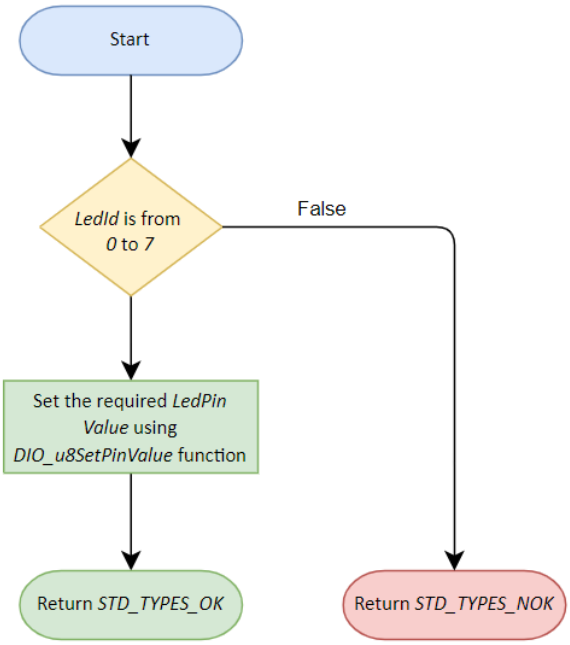
3.2. HAL Layer

3.2.1. LED Module

A. LED_initiailization

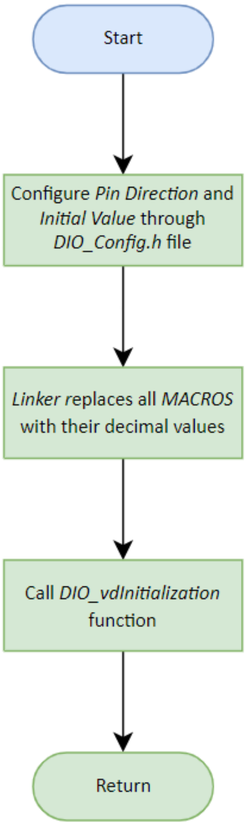


B. LED_setLEDPin

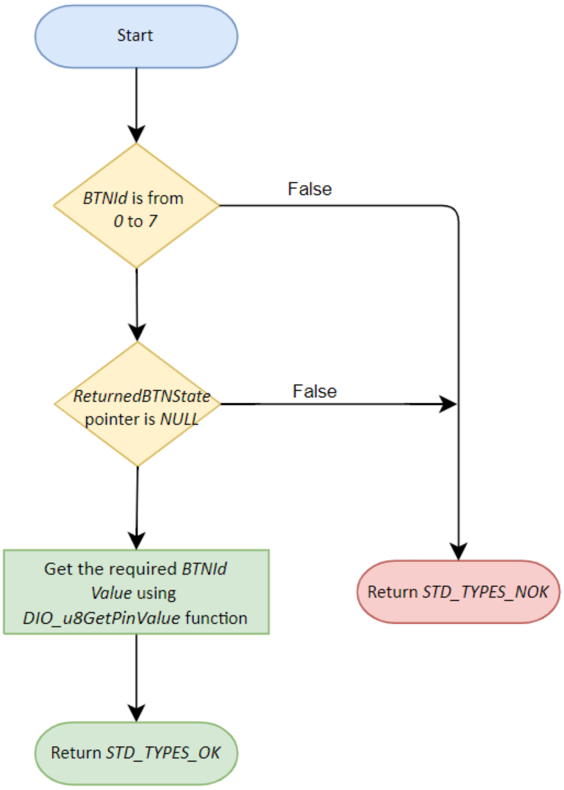


3.2.2. BTN Module

A. BTN_initiailization



B. BTN_getBTNState



4. References

1. *Draw IO*
2. *Layered Architecture | Baeldung on Computer Science*
3. *Microcontroller Abstraction Layer (MCAL) | Renesas*
4. *Hardware Abstraction Layer - an overview | ScienceDirect Topics*
5. *What is a module in software, hardware and programming?*
6. *Embedded Basics – API's vs HAL's*
7. *Using Push Button Switch with Atmega32 and Atmel Studio*