SPRINTS

Project Design

SMALL OPERATING SYSTEM

Version 1.0

Prepared by:

Hazem Ashraf

Mohamed Abdelsalam

Abdelrhman Walaa

May 2023

Table of Content

Small OS Design

1. Project Introduction

This project involves implementation of a small operating system, in which tasks are periodic and non-preemptive.

1.1. System Requirements

1.1.1. Hardware Requirements

1. **ATmega32** microcontroller
2. **PBUTTON0** to stop the SOS
3. **PBUTTON1** to run the SOS
4. **Two LEDs** to be toggled

1.1.2. Software Requirements

1. Implement an application that calls SOS module and use 2 tasks
2. Task_1: Toggle LED_0 (Every 300 Millisecond)
3. Task_2: Toggle LED_1 (Every 500 Millisecond)
4. Make sure these tasks occur periodically and forever
5. When pressing BUTTON 0, the SOS will stop
6. When pressing BUTTON 1, the SOS will run

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.

Middleware Layer (*MWL*) is usually the software layer that mediates between application software and the kernel or device driver software.

2.1.2. Layered Architecture

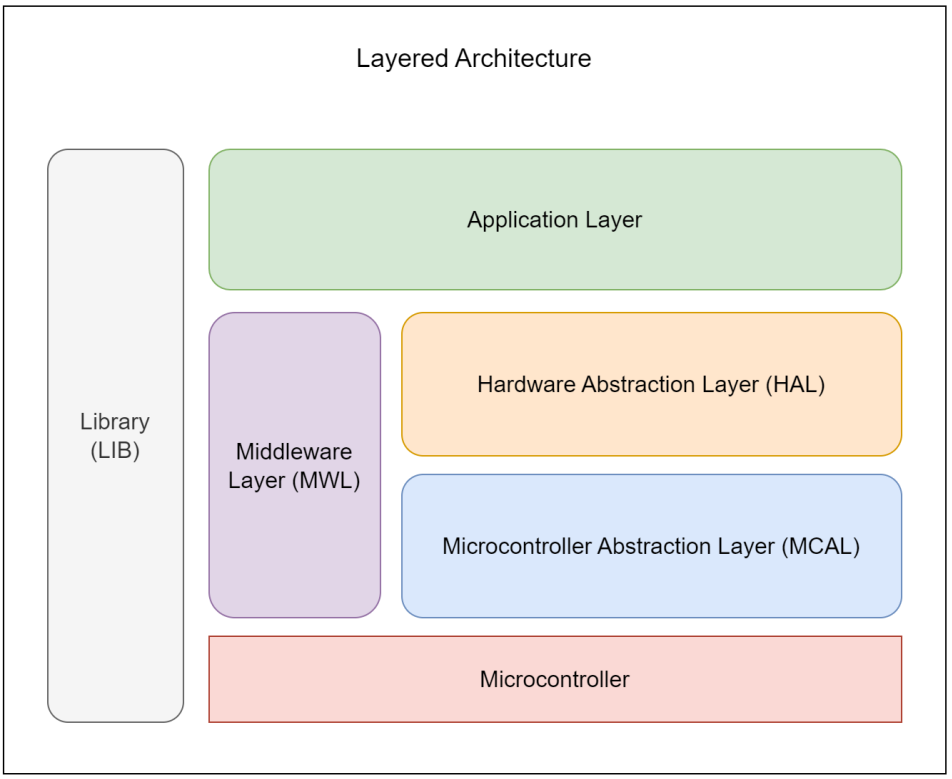


Figure 1. Layered Architecture Design

2.1.3. Project Circuit Schematic

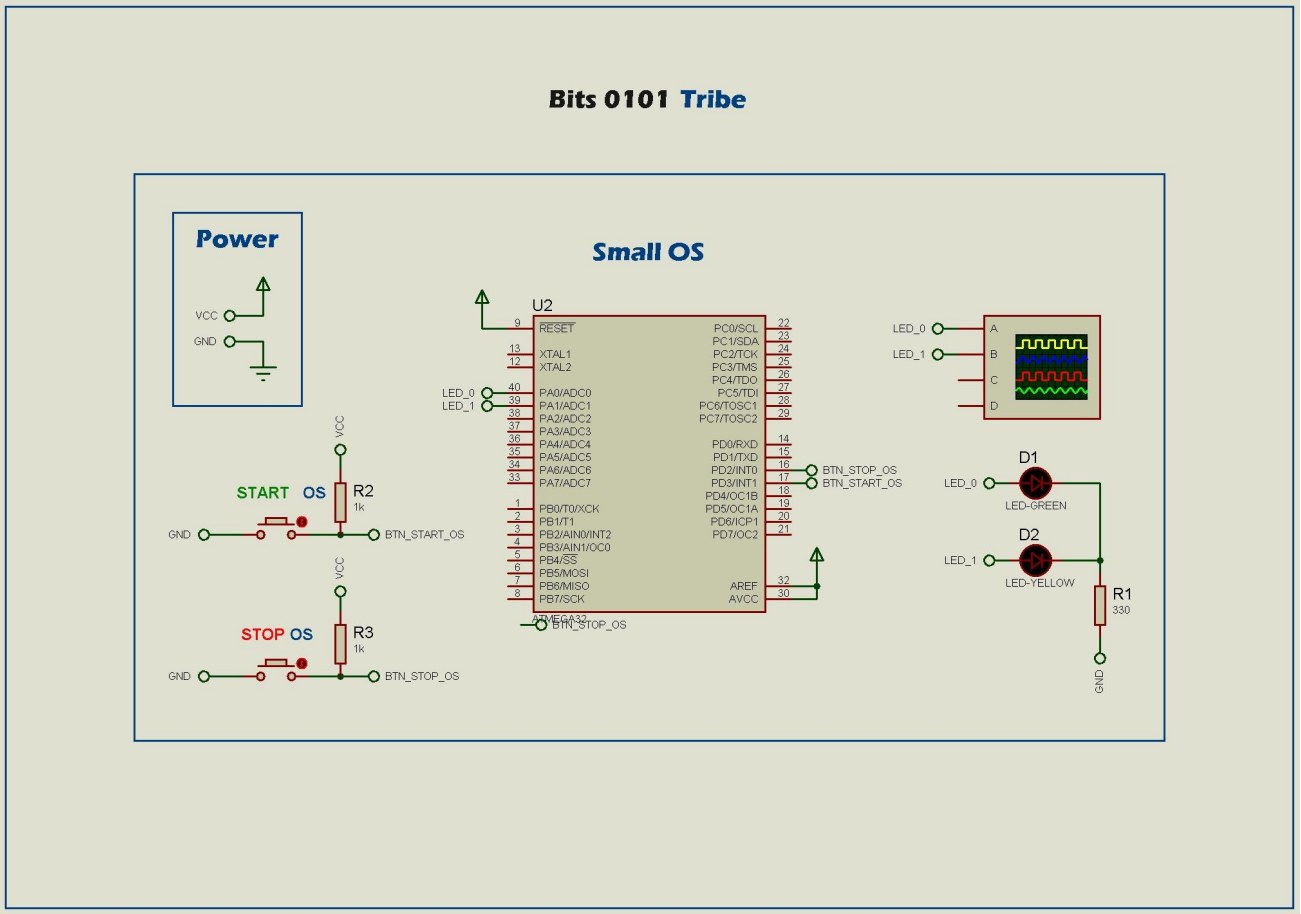


Figure 2. Project Circuit Schematic

2.2. Block Diagram

2.2.1. Definition

A *Block Diagram* (*Figure 3*) 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

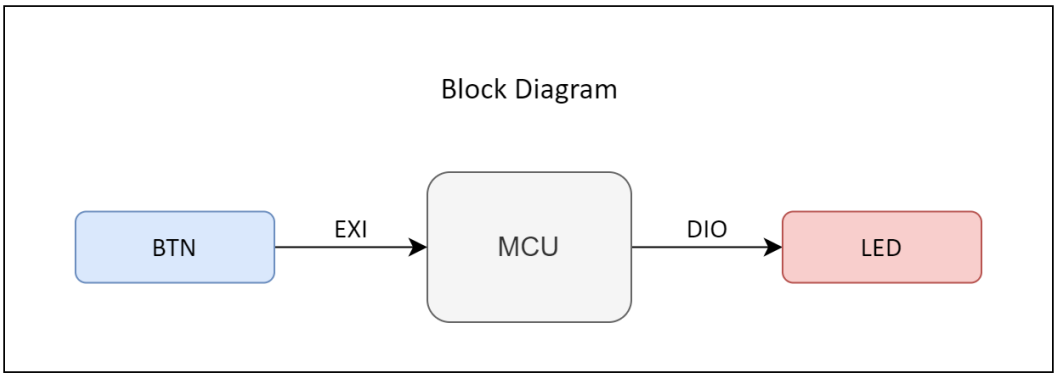


Figure 3. Block Diagram Design

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

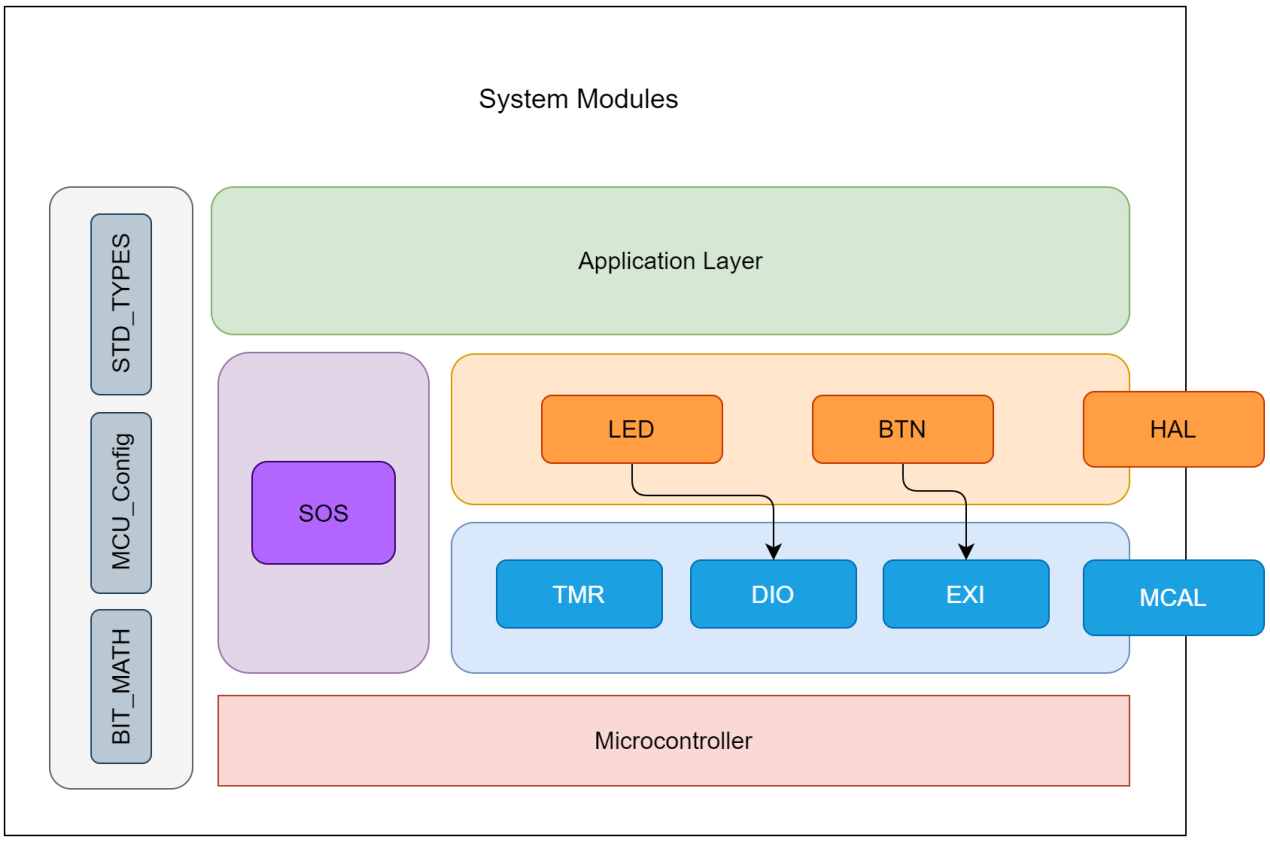


Figure 4. System Modules Design

2.4. Modules Description

2.4.1. DIO Module

The *DIO* (Digital Input/Output) module is responsible for reading input signals from the system's sensors (such as buttons) and driving output signals to the system's actuators (such as *LEDs*). It provides a set of APIs to configure the direction and mode of each pin (input/output, pull-up/down resistor), read the state of an input pin, and set the state of an output pin.

2.4.2. EXI Module

The *EXI* (External Interrupt) module is responsible for detecting external events that require immediate attention from the microcontroller, such as a button press. It provides a set of APIs to enable/disable external interrupts for specific pins, set the interrupt trigger edge (rising/falling/both), and define an interrupt service routine (*ISR*) that will be executed when the interrupt is triggered.

2.4.3. TMR Module

It is a module which is responsible for calculations and configurations of timer zero.
It provides an API to start counting till the setted time in ms then stop counting after finishing.

2.4.4. 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.

2.4.5. 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.

2.4.6. SOS Module

SOS (Small Operating System) is the design of a small OS with a priority based on Non preemptive scheduler based on time triggered Used to executing multiple tasks at different time intervals by design simple scheduler a scheduler can be viewed as a simple operating system that allows tasks to be called periodically or (less commonly) on a one-shot basis.
Also a scheduler can be viewed as a single timer interrupt service routine that is shared between many different tasks. As a result, only one timer needs to be initialized, and any changes to the timing generally require only one function to be altered.

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. DIO Driver APIs

```
| Name: DIO_init
| Input: en PortNumber, en PinNumber, and en PinDirection
| Output: void
| Description: Function to initialize Pin direction.
|
void DIO_init (EN_DIO_PortNumber en_a_portNumber, EN_DIO_PINNumber
en_a_pinNumber, EN_DIO_PINDirection en_a_pinDirection)

| Name: DIO_write
| Input: en PortNumber, en PinNumber, and en PinValue
| Output: void
| Description: Function to set Pin value.
|
void DIO_write (EN_DIO_PortNumber en_a_portNumber, EN_DIO_PINNumber
en_a_pinNumber, EN_DIO_PINValue en_a_pinValue)

| Name: DIO_read
| Input: en PortNumber, en PinNumber, and Pointer to u8 ReturnedData
| Output: void
| Description: Function to get Pin value.
|
void DIO_read (EN_DIO_PortNumber en_a_portNumber, EN_DIO_PINNumber
en_a_pinNumber, u8 *pu8_a_returnedData)
```

```
| Name: DIO_toggle
| Input: en portNumber and en PinNumber
| Output: void
| Description: Function to toggle Pin value.
|
void DIO_toggle (EN_DIO_PortNumber en_a_portNumber, EN_DIO_PINNumber
en_a_pinNumber)

| Name: DIO_setPortDirection
| Input: en PortNumber and en PortDirection
| Output: void
| Description: Function to set Port direction.
|
void DIO_setPortDirection (EN_DIO_PortNumber en_a_portNumber, u8
u8_a_portDirection)

| Name: DIO_setPortValue
| Input: en PortNumber and u8 PortValue
| Output: void
| Description: Function to set Port value.
|
void DIO_setPortValue (EN_DIO_PortNumber en_a_portNumber, u8 u8_a_portValue)

| Name: DIO_getPortValue
| Input: en PortNumber and Pointer to u8 ReturnedPortValue
| Output: void
| Description: Function to get Port value.
|
void DIO_getPortValue (EN_DIO_PortNumber en_a_portNumber, u8
*pu8_a_returnedPortValue)

| Name: DIO_setHigherNibble
| Input: en portNumber and u8 Data
| Output: void
| Description: Function to set Higher Nibble of Port.
|
void DIO_setHigherNibble (EN_DIO_PortNumber en_a_portNumber, u8 u8_a_data)

| Name: DIO_setLowerNibble
| Input: en PortNumber and u8 Data
| Output: void
| Description: Function to set LOWER Nibble of Port.
|
void DIO_setLowerNibble (EN_DIO_PortNumber en_a_portNumber, u8 u8_a_data)
```

2.5.2.2. EXI Driver APIs

Precompile and Linking Configurations Snippet:

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

```
| Name: EXI_enablePIE
| Input: u8 InterruptId and u8 SenseControl
| Output: u8 Error or No Error
| Description: Function to enable and configure Peripheral Interrupt Enable (PIE),
|              by setting relevant bit for each interrupt in GICR register, then
|              configuring Sense Control in MCUCR (case interrupt 0 or 1) or MCUCSR
|              (case interrupt 2) registers.
|
u8 EXI_enablePIE (u8 u8_a_interruptId, u8 u8_a_senseControl)

| Name: EXI_disablePIE
| Input: u8 InterruptId
| Output: u8 Error or No Error
| Description: Function to disable Peripheral Interrupt Enable (PIE), by clearing
|              relevant bits for each interrupt in the GICR register.
|
u8 EXI_disablePIE (u8 u8_a_interruptId)

| Name: EXI_intSetCallback
| Input: u8 InterruptId and Pointer to function INTInterruptAction that takes void and
|        returns void
| Output: u8 Error or No Error
| Description: Function to receive an address of a function (in APP Layer) to be
|              called back in ISR function of the passed Interrupt (InterruptId), the
|              address is passed through a pointer to function (INTInterruptAction),
|              and then pass this address to the ISR function.
|
u8 EXI_intSetCallback (u8 u8_a_interruptId,
void(*vpf_a_intInterruptAction)(void))
```

2.5.2.3. TMR Driver APIs

```
| Name: TMR_initialization
| Input: Pointer to st LinkConfig
| Output: en Error or No Error
| Description: Function to initialize timer with the required configuration.
|
TMR_enErrorState_t TMR_initialization (TMR_stLinkConfig_t *pst_a_linkConfig)

| Name: TMR_setTimer
| Input: u16 Time
| Output: void
| Description: Function to set the timer value.
|
void TMR_setTimer (u16 u16_a_time)

| Name: TMR_clearTimer
| Input: void
| Output: void
| Description: Function to clear the timer value.
|
void TMR_clearTimer (void)

| Name: TMR_stopTimer
| Input: void
| Output: void
| Description: Function to stop the timer.
|
void TMR_stopTimer (void)

| Name: TMR_resumeTimer
| Input: void
| Output: void
| Description: Function to resume the timer.
|
void TMR_resumeTimer (void)

| Name: TMR_ovfSetCallBack
| Input: Pointer to function INTInterruptAction that takes void and
|        returns void
| Output: en Error or No Error
| Description: Function to set timer overflow callback function.
|
TMR_enErrorState_t TMR_ovfSetCallback (void(*vpf_a_intInterruptAction)(void))
```

```
| Name: TMR_compASetCallBack
| Input: Pointer to function INTInterruptAction that takes void and
|        returns void
| Output: en Error or No Error
| Description: Function to set timer A compare callback function.
|
TMR_enErrorState_t TMR_compASetCallback
(void(*vpf_a_intInterruptAction)(void))

| Name: TMR_compBSetCallBack
| Input: Pointer to function INTInterruptAction that takes void and
|        returns void
| Output: en Error or No Error
| Description: Function to set timer B compare callback function.
|
TMR_enErrorState_t TMR_compBSetCallback
(void(*vpf_a_intInterruptAction)(void))
```

2.5.3. HAL APIs

2.5.3.1. BTN Driver APIs

```
| Name: BTN_initializationNMLMode
| Input: u8 DIOPortId and u8 DIOPinId
| Output: u8 Error or No Error
| Description: Function to initialize BTN pin in NML Mode.
|
u8 BTN_initializationNMLMode (u8 u8_a_DIOPortId, u8 u8_a_DIOPinId)

| Name: BTN_initializationEXIMode
| Input: u8 EXIId, u8 EXISenseControl, and Pointer to Function that takes
|        void and returns void
| Output: u8 Error or No Error
| Description: Function to initialize BTN pin in EXI Mode.
|
u8 BTN_initializationEXIMode (u8 u8_a_EXIId, u8 u8_a_EXISenseControl, void
(*pf_a_EXIAction)(void))

| Name: BTN_getBTNState
| Input: u8 DIOPortId, u8 DIOPinId and Pointer to u8 ReturnedBTNState
| Output: u8 Error or No Error
| Description: Function to get BTN state.
|
u8 BTN_getBTNState (u8 u8_a_DIOPortId, u8 u8_a_DIOPinId, u8
*pu8_a_returnedBTNState)
```

2.5.3.2. LED Driver APIs

```
| Name: LED_initialization
| Input: u8 LedId
| Output: u8 Error or No Error
| Description: Function to initialize LED peripheral.
|
u8 LED_initialization (u8 u8_a_ledId)

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

2.5.4. MWL APIs

2.5.4.1. SOS Driver APIs

```
| Name: SOS_initialization
| Input: void
| Output: en Error or No Error
| Description: Function to initialize SOS database.
|
SOS_enErrorState_t SOS_initialization (void)

| Name: SOS_deinitialization
| Input: void
| Output: en Error or No Error
| Description: Function to reset the SOS database to invalid values.
|
SOS_enErrorState_t SOS_deinitialization (void)

| Name: SOS_createTask
| Input: Pointer to en Task
| Output: en Error or No Error
| Description: Function to create a new task and add it to the SOS database.
|
SOS_enErrorState_t SOS_createTask (SOS_enTask_t *pen_a_task)

| Name: SOS_deleteTask
| Input: Pointer to en Task
| Output: en Error or No Error
| Description: Function to delete a task from SOS database.
|
SOS_enErrorState_t SOS_deleteTask (SOS_enTask_t *pen_a_task)

| Name: SOS_modifyTask
| Input: Pointer to en Task
| Output: en Error or No Error
| Description: Function to  modify a task parameters in SOS database.
|
SOS_enErrorState_t SOS_modifyTask (SOS_enTask_t *pen_a_task)

| Name: SOS_enableScheduler
| Input: void
| Output: void
| Description: Function to enable the scheduler.
|
void SOS_enableScheduler (void)
```

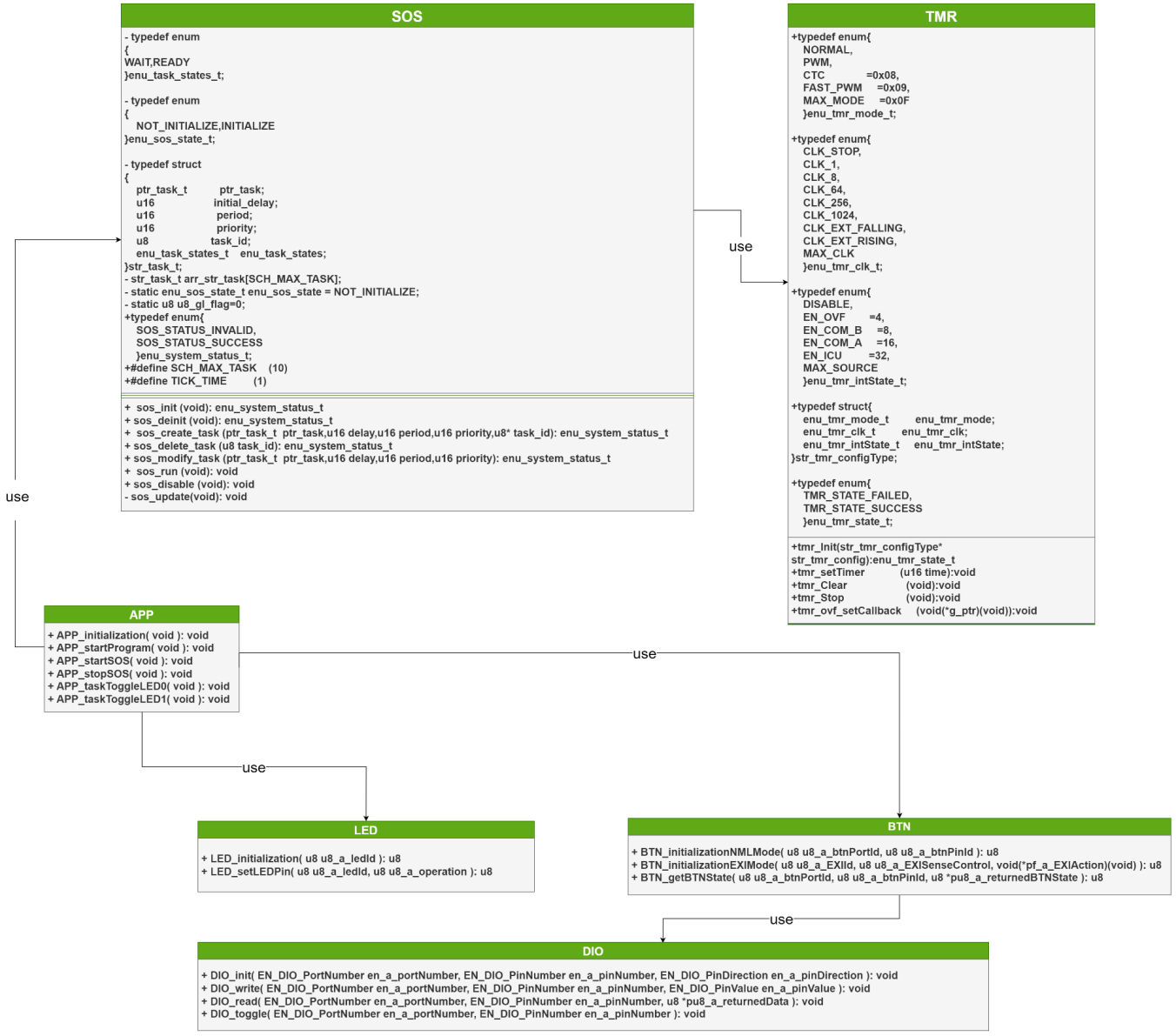
```
| Name: SOS_disableScheduler
| Input: void
| Output: void
| Description: Function to disable the scheduler.
|
void SOS_disableScheduler (void)
```

2.5.5. APP APIs

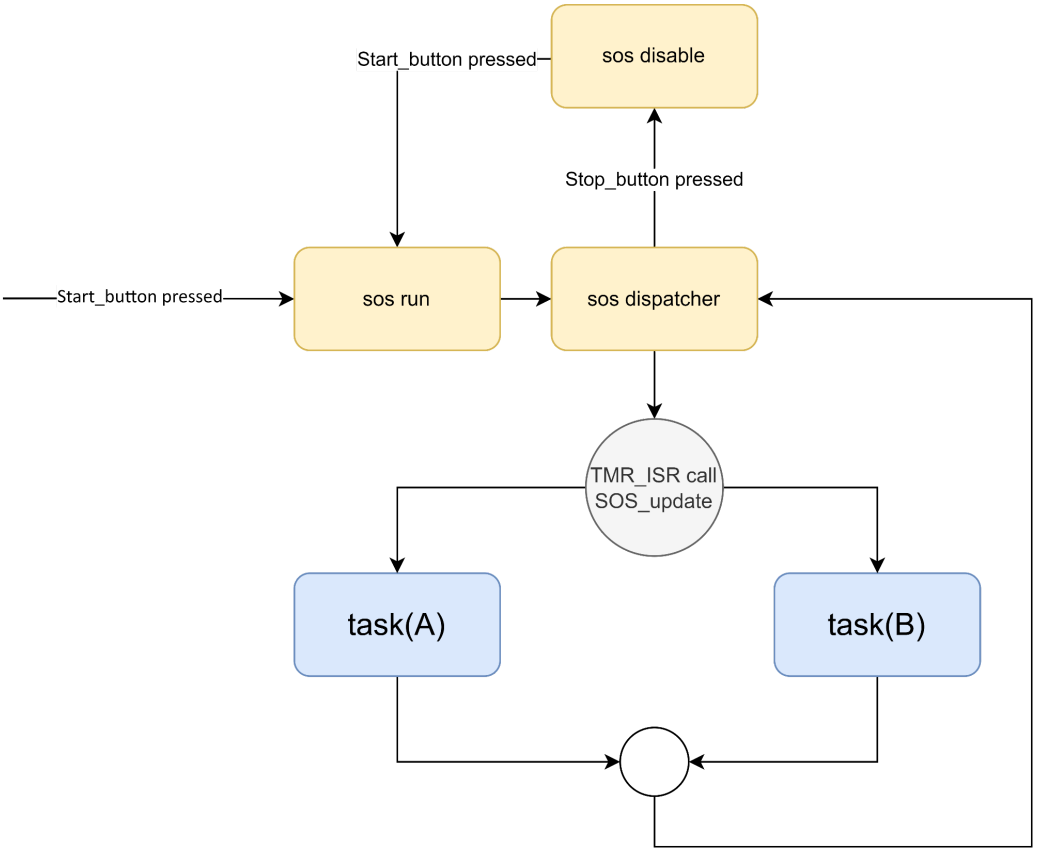
```
| Name: APP_initialization
| Input: void
| Output: void
| Description: Function to Initialize the Application.
|
void APP_initialization (void)

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

| Name: APP_startSOS
| Input: void
| Output: void
| Description: Function to Start the Operating System.
|
void APP_startSOS (void)

| Name: APP_stopSOS
| Input: void
| Output: void
| Description: Function to Stop the Operating System.
|
void APP_stopSOS (void)

| Name: APP_taskToggleLED0
| Input: void
| Output: void
| Description: Function to implement task 1 logic.
|
void APP_taskToggleLED0 (void)

| Name: APP_taskToggleLED1
| Input: void
| Output: void
| Description: Function to implement task 2 logic.
|
void APP_taskToggleLED1 (void)
```

2.6. UML

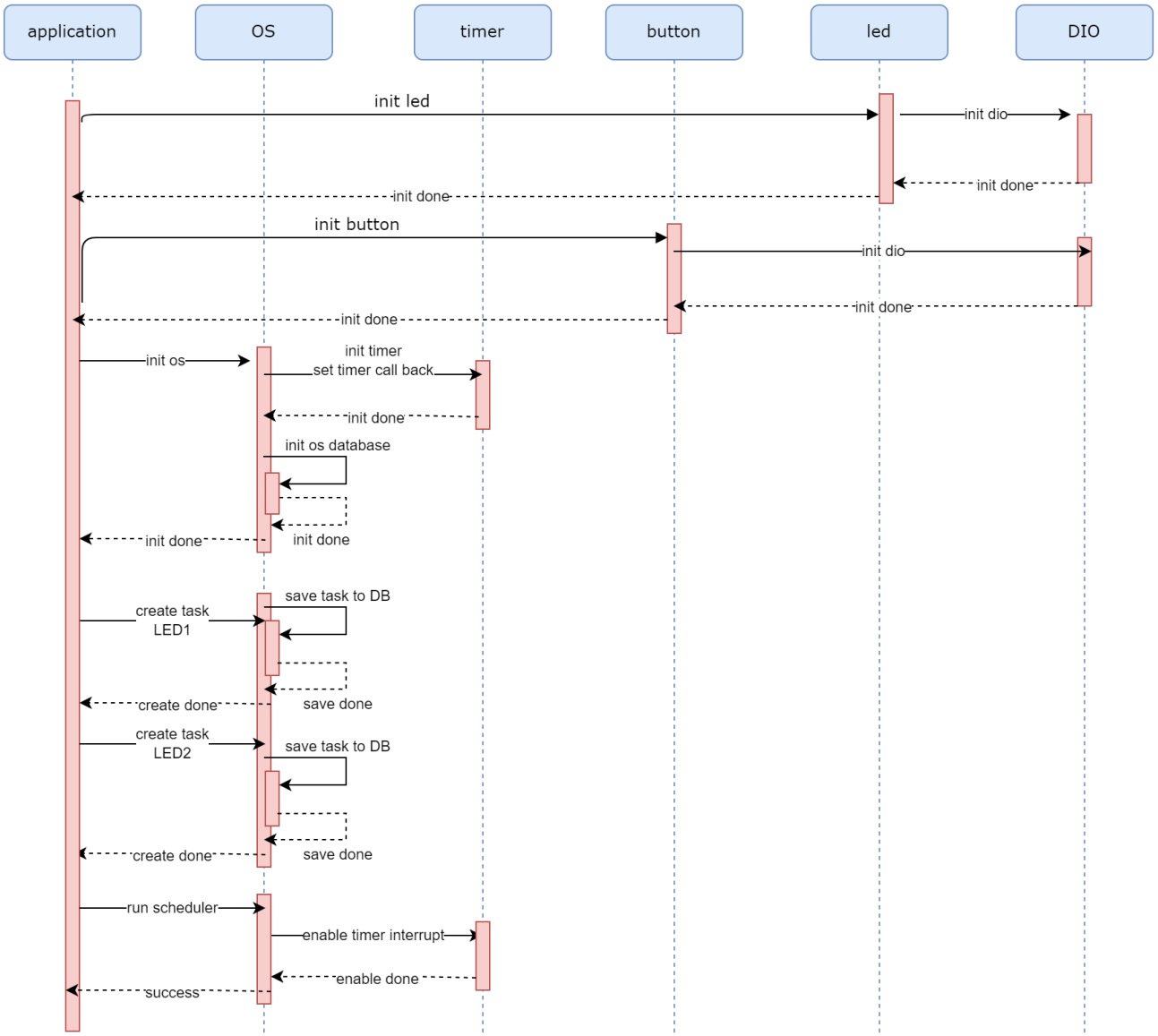
2.6.1. Class Diagram



2.6.2. State Machine Diagram

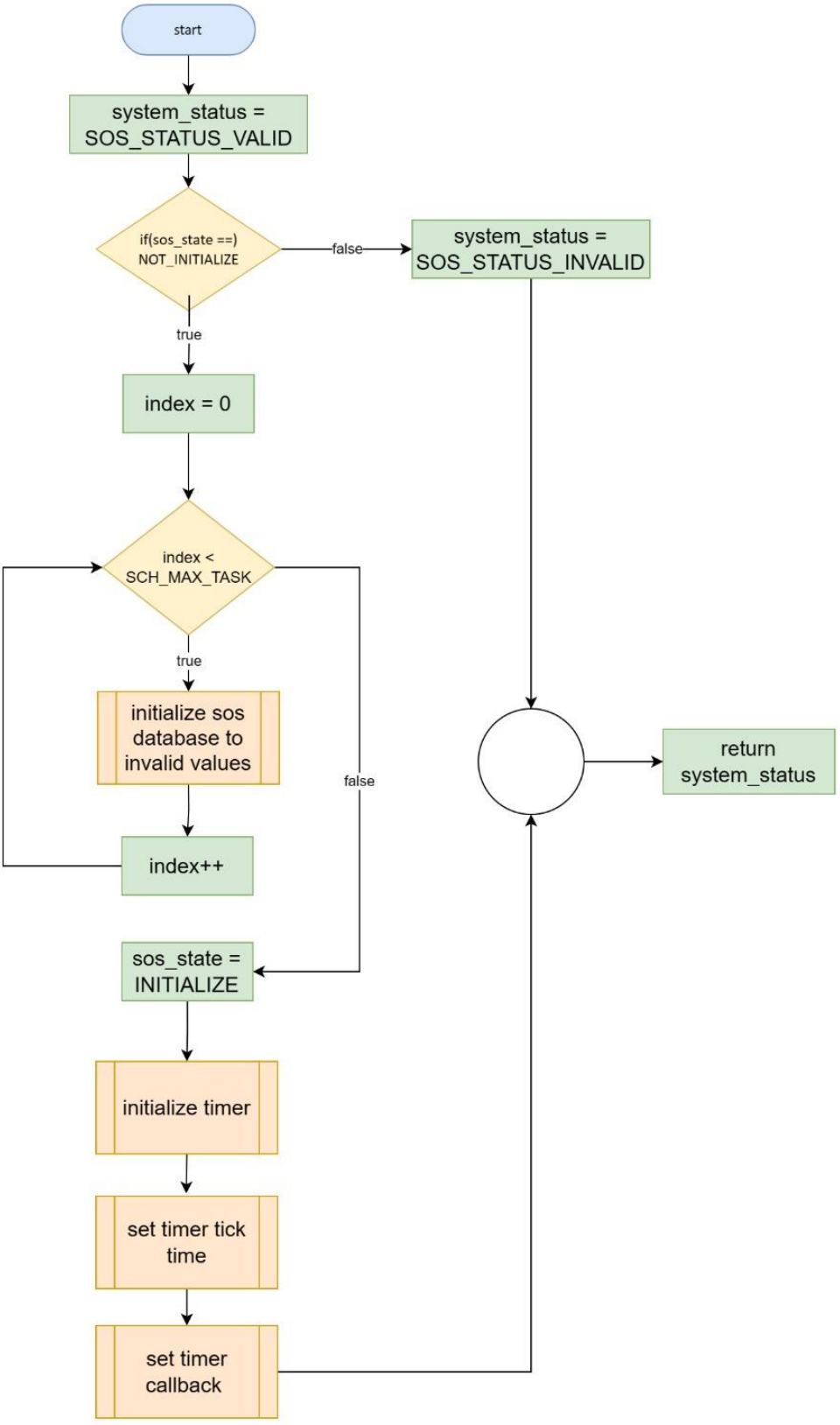


2.6.3. Sequence Diagram

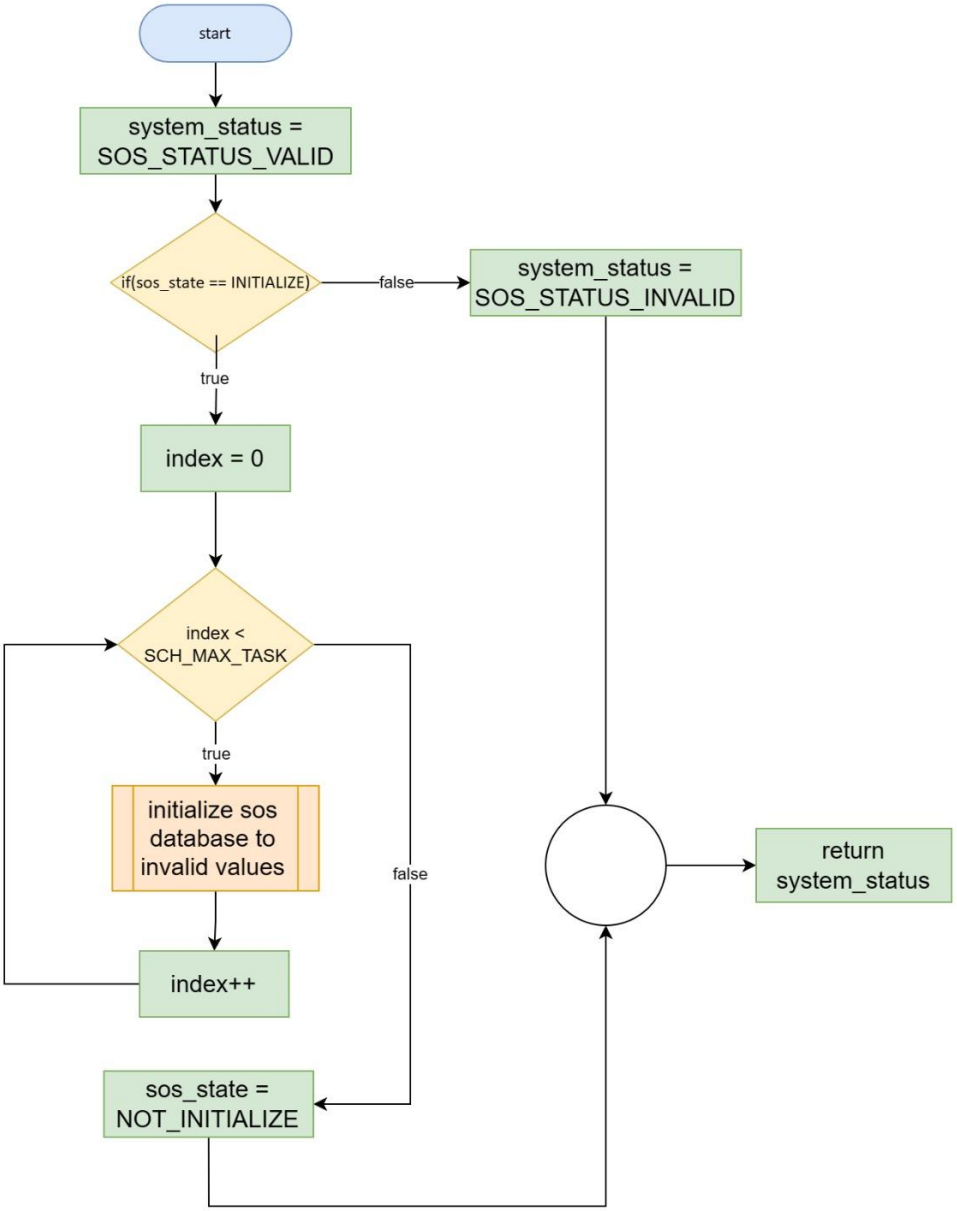


2.6.4. Flowchart Diagram

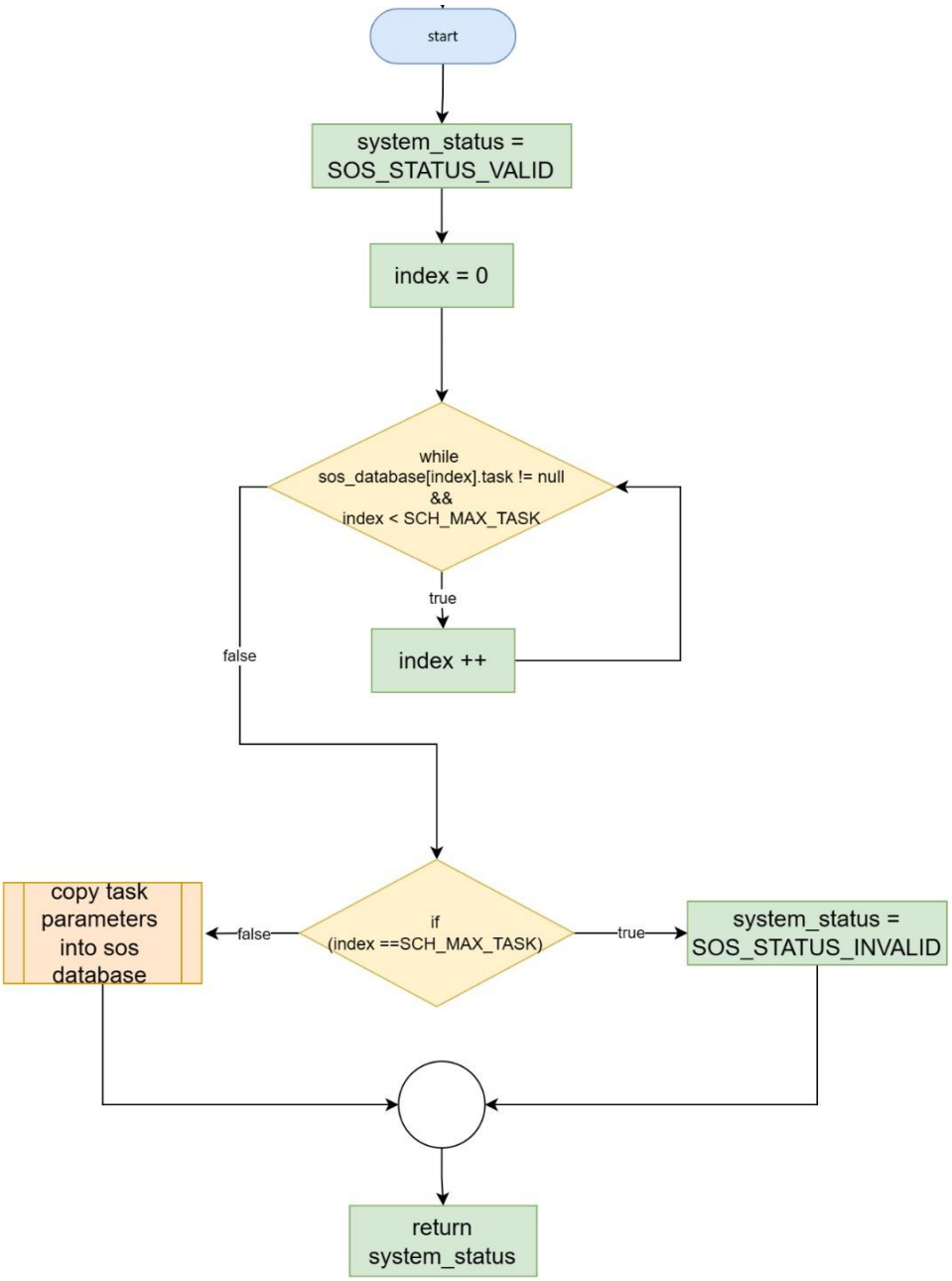
A. SOS_initialization



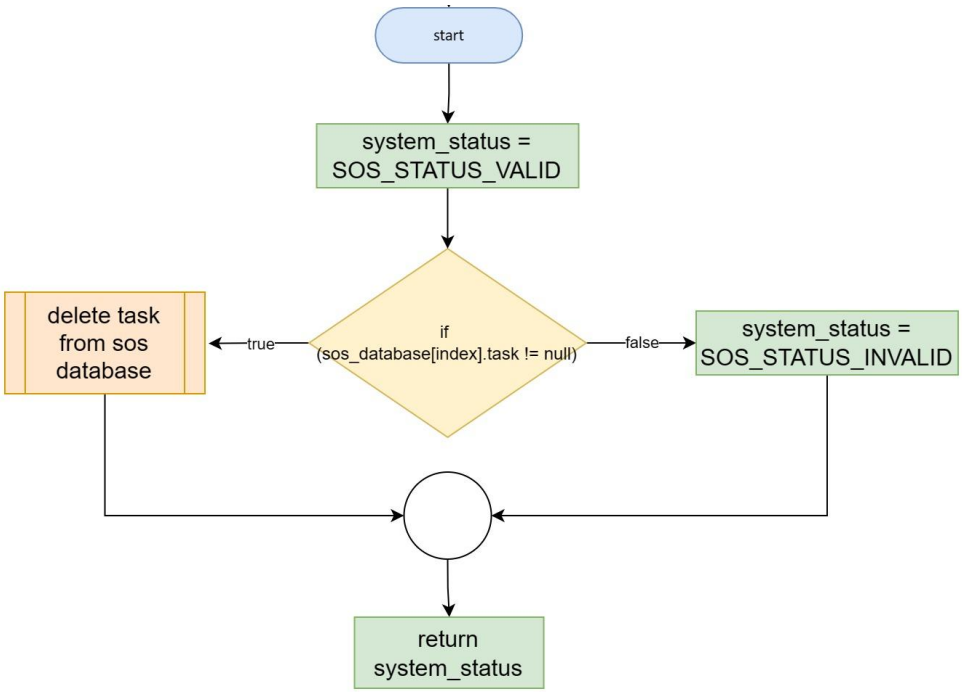
B. SOS_deinitialization



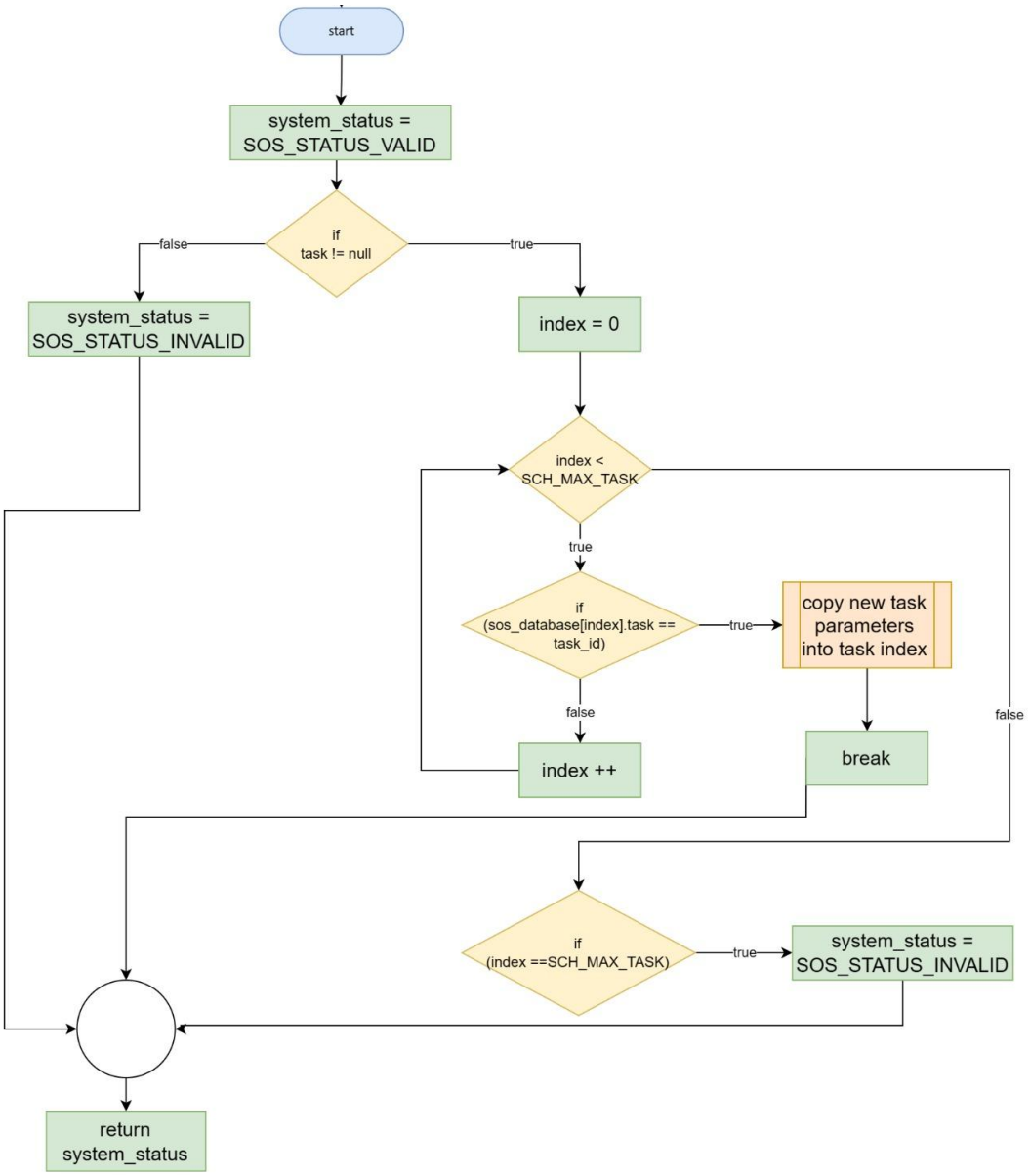
C. SOS_createTask



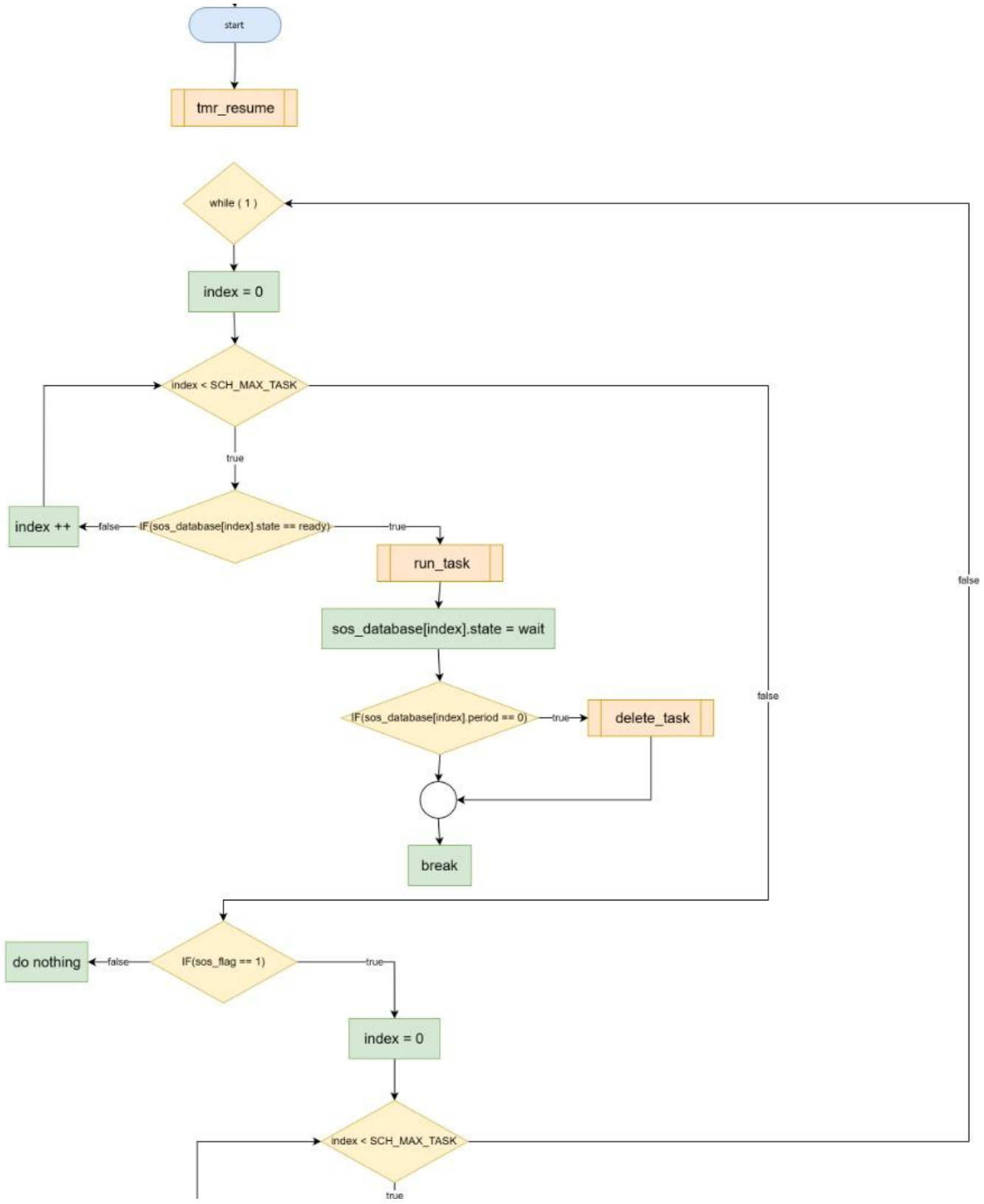
D. SOS_deleteTask



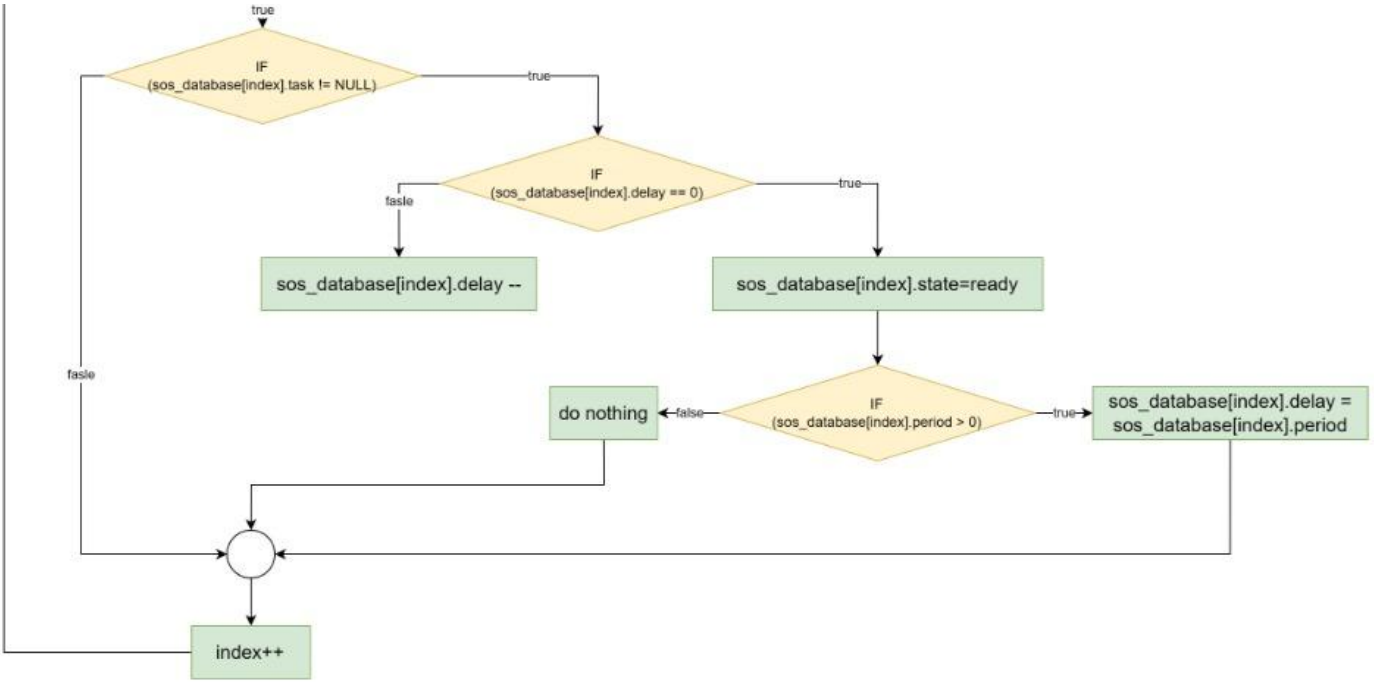
E. SOS_modifyTask



F. SOS_enableScheduler



Continue the flowchart on the next page


true
IF
(sos_database[index].task != NULL)
true
IF
(sos_database[index].delay == 0)
fasle
true
sos_database[index].delay --
sos_database[index].state=ready
fasle
do nothing
false
IF
(sos_database[index].period > 0)
true
sos_database[index].delay =
sos_database[index].period
index++

3. Development Issues

3.1. 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 .
Secondly, due to the ***team-shift*** process done earlier, we faced ***integration issues*** with drivers previously implemented before the team-shift. Thus, integrating or sometimes redeveloping some APIs led to consuming more time than expected as well.

4. 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. *Guide to Embedded Systems Architecture - Part 1: Defining middleware - EE Times*
7. *Top Five Differences Between Layers And Tiers | Skill-Lync*
8. *What is a module in software, hardware and programming?*
9. *Embedded Basics – API's vs HAL's*
10. *Using Push Button Switch with Atmega32 and Atmel Studio*