Sprints

SMALL OPERATING SYSTEM

SOS

You are required to design a small OS with a priority based preemptive scheduler based on time-triggered.

Prepared By

Hossam Elwahsh

Sprints

Small Operating System Design

1. Project Introduction

You are required to design a small OS with a priority based preemptive scheduler based on time-triggered.

2. Project Components

- ATmega32 microcontroller
- Two Buttons:
  - BUTTON0: start
  - BUTTON1: stop
- Two LEDs

3. System Layered Architecture

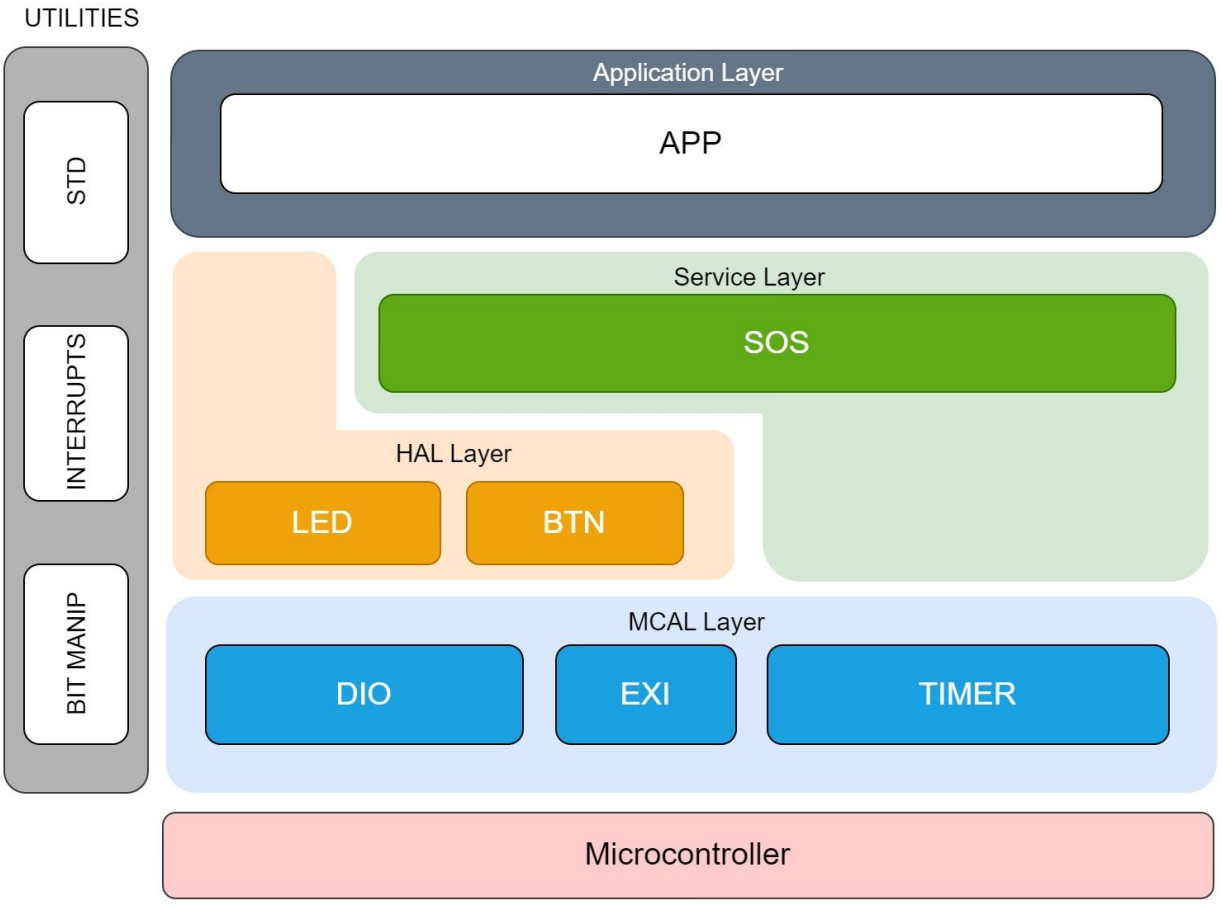


Figure 1. Layered Architecture Design

4. Modules Description

4.1. DIO (Digital Input/Output) Module

The *DIO* 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.

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.

4.3. TIMER Module

The *TIMER* module is responsible for generating timing events that are used by other modules in the system. It provides a set of APIs to configure the timer clock source and prescaler, set the timer mode (count up/down), set the timer period, enable/disable timer interrupts, and define an ISR that will be executed when the timer event occurs.

4.4. BTN Module

The *BTN* (Button) module is responsible for reading the state of the system's buttons. It provides a set of APIs to enable/disable button interrupts, set the button trigger edge (rising/falling/both), and define an ISR that will be executed when a button press is detected.

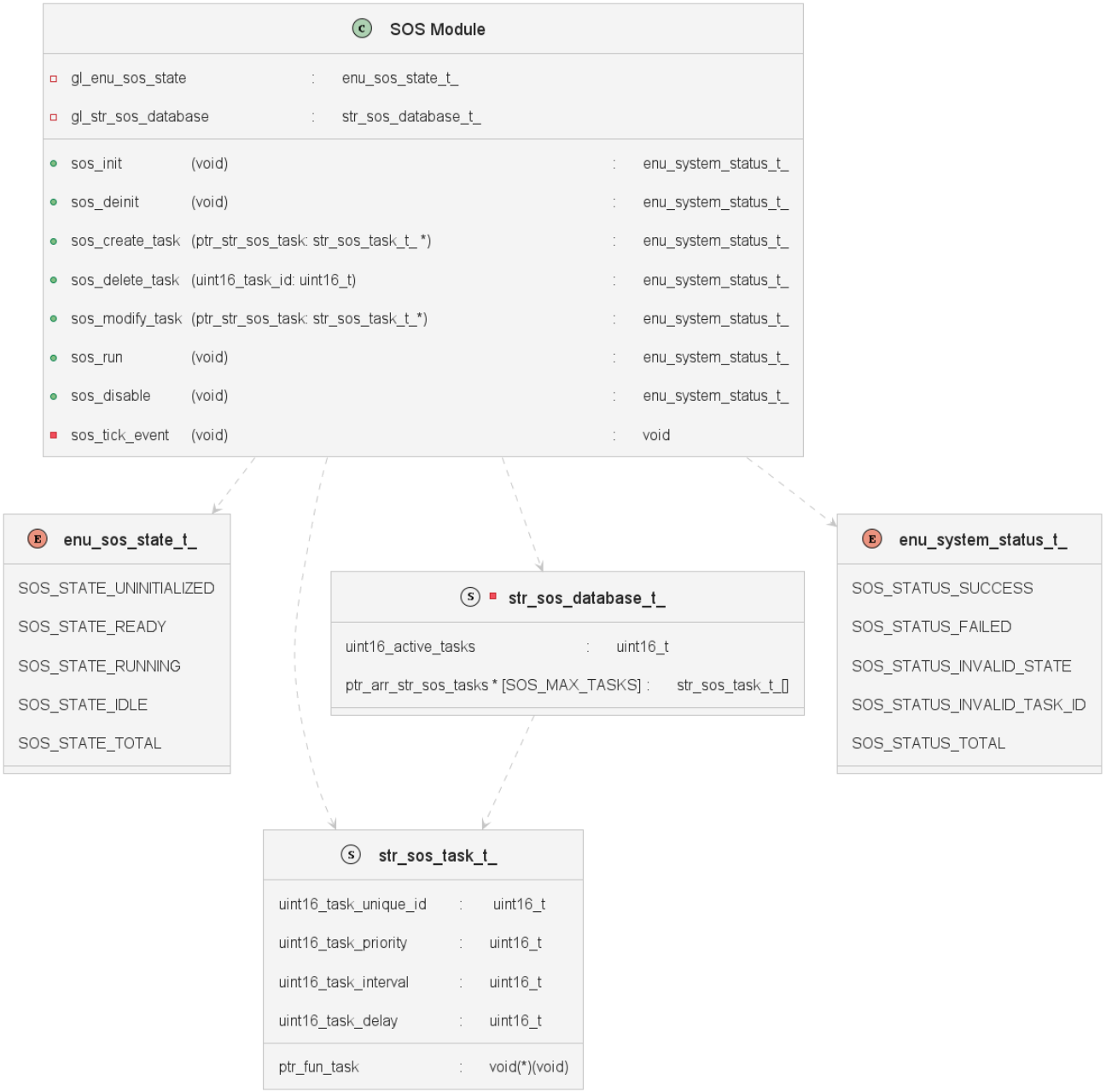
4.5. LED Module

The *LED* (Light Emitting Diode) module is responsible for controlling the state of the system's *LEDs*. It provides a set of APIs to turn on/off each *LED* and toggle its state.

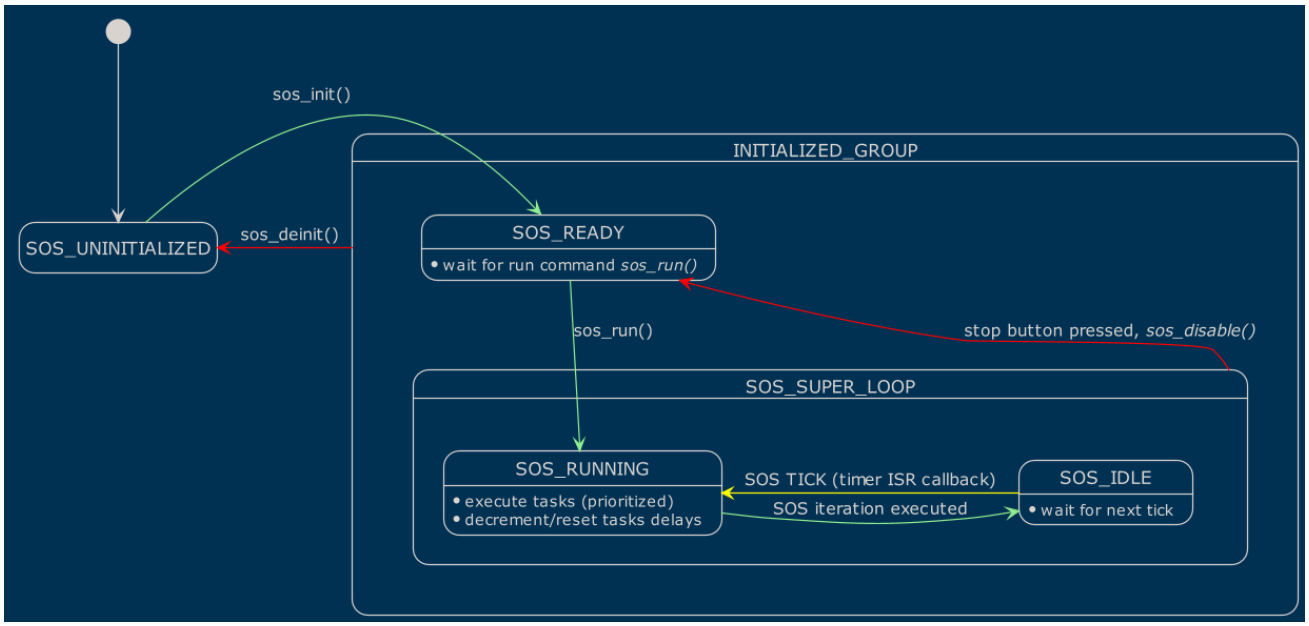
4.6. SOS Module

The SOS (Small Operating System) module is a lightweight operating system designed to provide essential functionality for embedded systems and resource-constrained devices. It offers a minimalistic and efficient approach to managing hardware resources and executing applications in environments where limited memory, processing power, and storage are available.

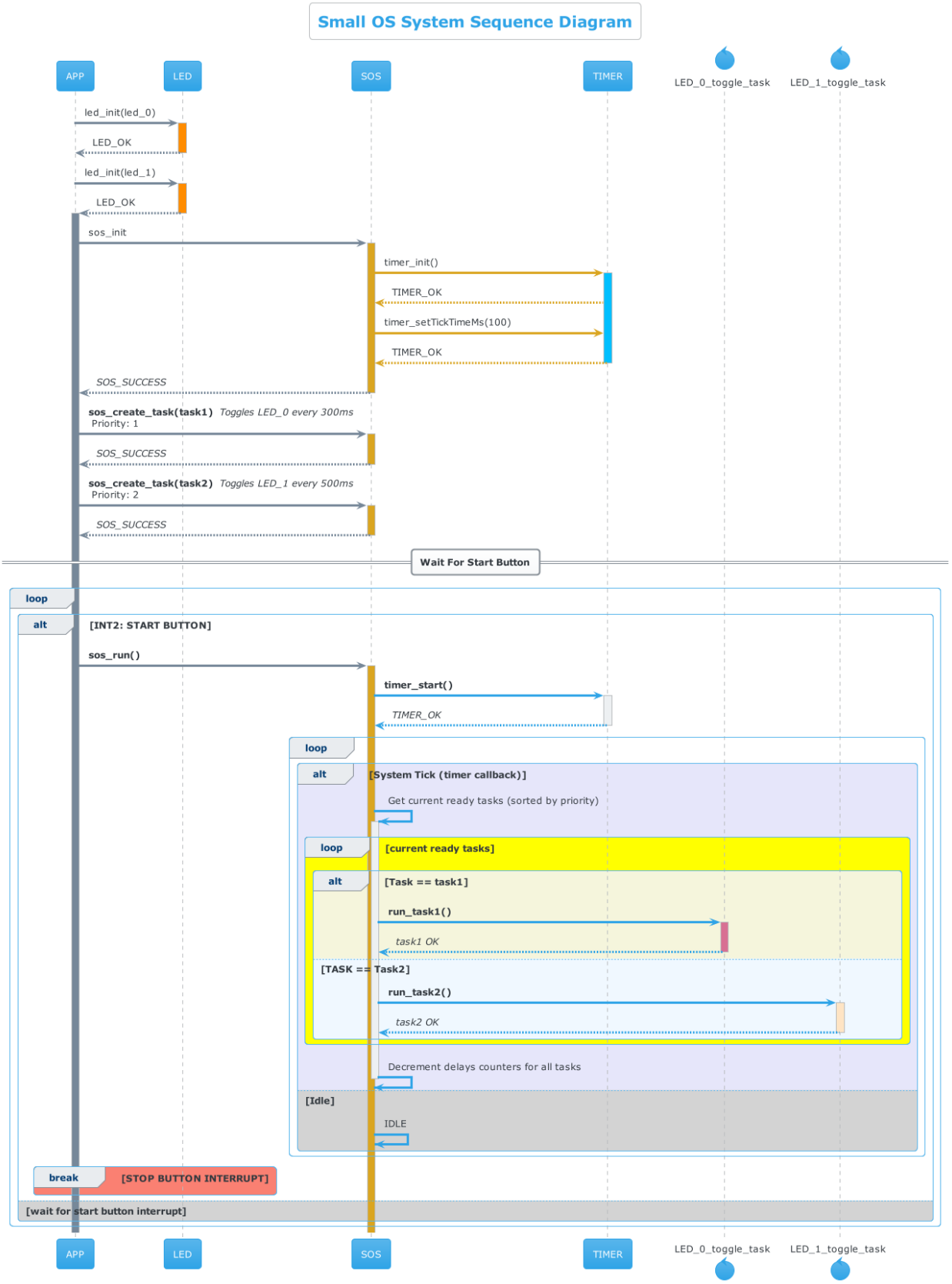
5. SOS Module Class Diagram



6. SOS Module State Machine



7. System Sequence Diagram *(click for HQ)*



8. SOS module header files

8.1. SOS Interface

```
/**
 * @file    :   sos_interface.h
 * @author  :   Hossam Elwahsh - https://github.com/HossamElwahsh
 * @brief   :   Header File contains all the types and status code for SOS
 * @version :   1.0
 * @date    :   2023-05-29
 *
 * @copyright Copyright (c) 2023
 */

#ifndef SOS_INTERFACE_H_
#define SOS_INTERFACE_H_

#include "sos_preconfig.h"
#include "../std.h"

/* -------------------- ENUMS -------------------- */

typedef enum
{
    SOS_STATUS_SUCCESS          =   0       ,
    SOS_STATUS_FAILED                       ,
    SOS_STATUS_INVALID_STATE                ,
    SOS_STATUS_INVALID_INVALID_TASK_ID      ,
    SOS_STATUS_TOTAL
}enu_system_status_t_;

typedef enum
{
    /** SOS is yet to be initialized */
    SOS_STATE_UNINITIALIZED =   0   ,

    /** SOS initialized and ready to run */
    SOS_STATE_READY                 ,

    /** SOS is running */
    SOS_STATE_RUNNING               ,

    /** SOS is idle */
    SOS_STATE_IDLE                  ,

    /** SOS is stopped */
```

```
    SOS_STATE_STOPPED               ,

    SOS_STATE_TOTAL
}enu_sos_state_t_;

/* --------------------- STRUCTS --------------------- */

typedef struct
{
    void        (* ptr_fun_task)(void)      ;
    uint16_t_   uint16_task_unique_id       ;
    uint16_t_   uint16_task_priority        ;
    uint16_t_   uint16_task_interval        ;
    uint16_t_   uint16_task_delay           ;
}str_sos_task_t_;

typedef struct
{
    uint16_t_           uint16_active_tasks             ;
    str_sos_task_t_     str_sos_tasks[SOS_MAX_TASKS]    ;
}str_sos_database_t_;

/* --------------------- Functions' Prototypes --------------------- */

/**
 * @brief                               :   Initializes SOS
 *
 * @return  SOS_STATUS_SUCCESS          :   In case of Successful Operation
 *          SOS_STATUS_INVALID_STATE    :   In case the sos is already
initialized
 */
enu_system_status_t_    sos_init            (void);


/**
 * @brief                               :   De-initializes SOS
 *
 * @return  SOS_STATUS_SUCCESS          :   In case of Successful Operation
 *          SOS_STATUS_INVALID_STATE    :   In case the sos is already
de-initialized or was not initialized previously
 */
enu_system_status_t_    sos_deinit          (void);
```

```
/**
 * @brief                               :   Creates a new task
 *
 * @param ptr_str_sos_task              :   Pointer to task to be created
 *
 * SOS_STATUS_SUCCESS                  :   In case of Successful Operation
(created)
 * SOS_STATUS_FAILED                   :   In case of Failed Operation
(scheduler full / not init)
 */
enu_system_status_t_    sos_create_task     (str_sos_task_t_ *
ptr_str_sos_task);

/**
 * @brief                               :   Deletes a task by it's ID
 *
 * @param uint16_task_unique_id         :   Task ID for the task to be deleted
 *
 * SOS_STATUS_SUCCESS                  :   In case of Successful Operation
(deleted)
 * SOS_STATUS_FAILED                   :   In case of Failed Operation
(failed to delete)
 * SOS_STATUS_INVALID_INVALID_TASK_ID  :   In case of Failed Operation (ID
doesn't exist)
 */
enu_system_status_t_    sos_delete_task     (uint16_t_ uint16_task_unique_id);

/**
 * @brief                               :   Modifies an existing task by it's
ID
 *
 * @param ptr_str_sos_task              :   Pointer to new task values
including the task ID to be modified
 *
 * SOS_STATUS_SUCCESS                  :   In case of Successful Operation
(modified)
 * SOS_STATUS_FAILED                   :   In case of Failed Operation
(failed to modify)
 * SOS_STATUS_INVALID_INVALID_TASK_ID  :   In case of Failed Operation (ID
doesn't exist)
 */
enu_system_status_t_    sos_modify_task     (str_sos_task_t_ *
ptr_str_sos_task);
```

```
/**
 * @brief                               :   Runs/Starts the scheduler
 * @return  SOS_STATUS_SUCCESS          :   In case of Successful Operation
(started)
 *          SOS_STATUS_FAILED           :   In case of failing to start the
scheduler
 *          SOS_STATUS_INVALID_STATE    :   In case Failed Operation; (SOS is
already running, not init, was de-init)
 */
enu_system_status_t_    sos_run             (void);


/**
 * @brief                               :   Stops the scheduler
 *
 * @return  SOS_STATUS_SUCCESS          :   In case of Successful Operation
 *          SOS_STATUS_INVALID_STATE    :   In case of Failed Operation; (SOS
is not running, not init, not de-init)
 */
enu_system_status_t_    sos_disable         (void);


/**
 * @brief event called when selected timer ticks (callback)
 *
 */
void sos_tick_event                         (void);


#endif /* SOS_INTERFACE_H_ */
```

8.2. SOS Preconfiguration

```c
/**
 * @file    :   sos_preconfig.h
 * @author  :   Hossam Elwahsh - https://github.com/HossamElwahsh
 * @brief   :   Header File contains all SOS pre-configuration macros
 * @version :   1.0
 * @date    :   2023-05-29
 *
 * @copyright Copyright (c) 2023
 */


#ifndef SOS_PRECONFIG_H_
#define SOS_PRECONFIG_H_

/** MAX number of tasks that SOS can handle */
#define SOS_MAX_TASKS 20

/** SOS system tick time in ms */
#define SOS_SYSTEM_MS_TICK 100


#endif /* SOS_PRECONFIG_H_ */
```