# SOS

BY: Momen Hassan

# Contents

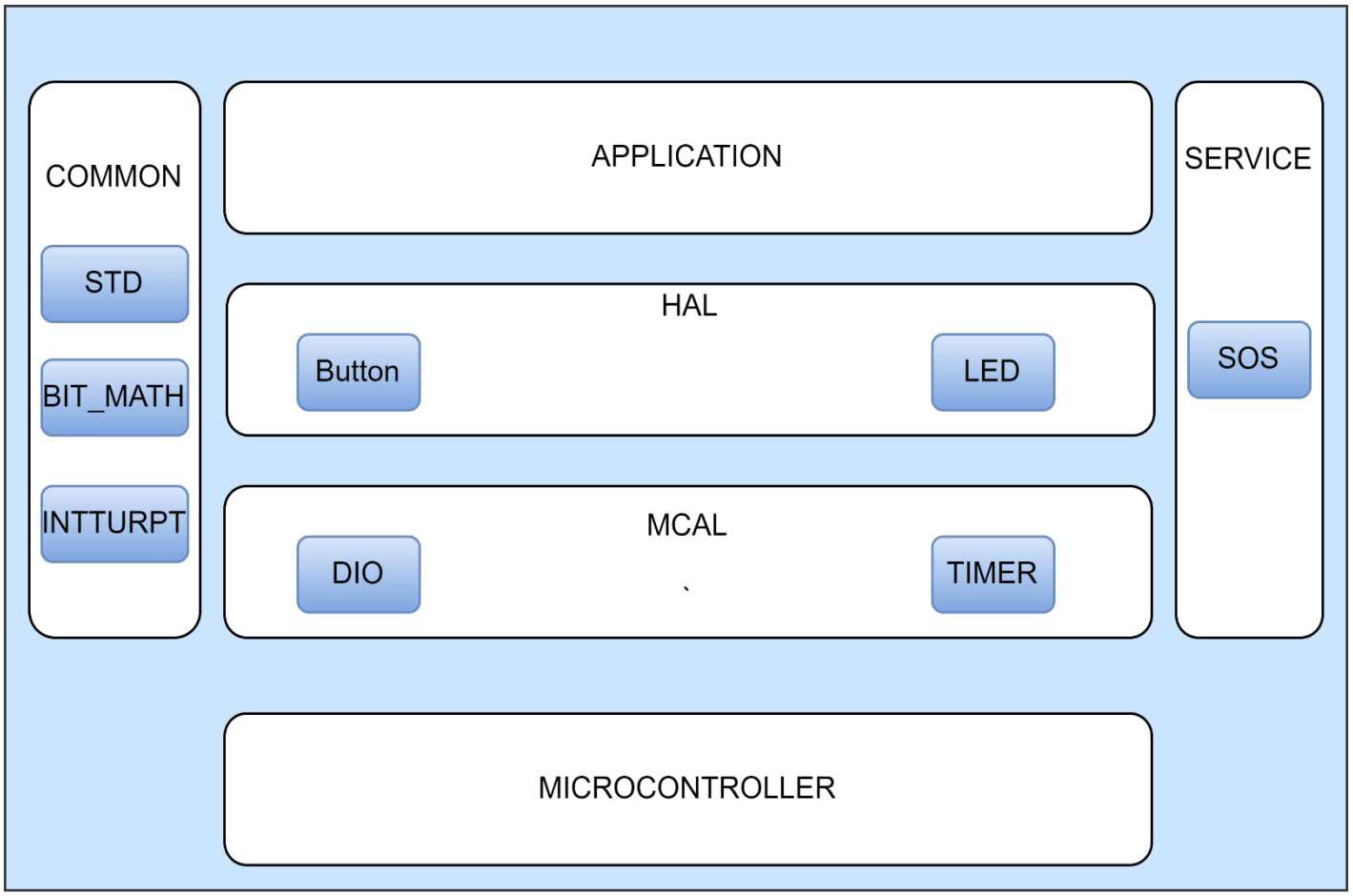
# Introduction

The SOS (Small OS) module is a lightweight operating system designed for embedded systems, specifically targeting devices with limited resources. It provides a priority-based preemptive scheduler that allows for efficient task execution based on their priority and time-triggered events.

**Features**

- **Priority-Based Preemptive Scheduler**: The SOS module implements a priority-based preemptive scheduler, ensuring that higher priority tasks are executed before lower priority tasks. This enables efficient utilization of system resources and responsiveness to time-critical operations.
- **Task Management**: The SOS module allows the creation, deletion, and modification of tasks. Tasks are defined as individual units of work that can be executed periodically or on-demand. Each task is associated with a unique task ID, interval, last execution time, and a function pointer that represents the task's code.
- **Time-Triggered Events**: The SOS module supports time-triggered events, allowing tasks to be executed at specific intervals. Tasks can be scheduled to run periodically or based on custom time intervals defined by the user.
- **Initialization and Deinitialization**: The SOS module provides functions for initializing and deinitializing the SOS database. The initialization function sets up the necessary data structures and initializes the scheduler, while the deinitialization function resets the SOS database to an invalid state.
- **Task Modification**: The SOS module allows for the modification of existing tasks. Users can update the task's interval or function pointer to adapt to changing requirements.
- **Scheduler Control**: The SOS module provides functions to enable and disable the scheduler. Disabling the scheduler halts task execution, while enabling it resumes the execution of tasks.

# LAYERED ARCHITECTURE



# Module, Peripherals, & Supporting Drivers Description

### 2.1.1. 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.

### 2.1.2. BUTTON Module

The BUTTON module is responsible for handling the input from buttons or push buttons in a system. It provides an interface to detect button presses, releases, and other related events. The module typically interacts with the underlying hardware or low-level drivers to monitor the state changes of the buttons and notify the application or other modules accordingly.

### 2.1.3. LED Module

The *LED* Module (driver) for ATmega32 is a compact and versatile solution designed to control LEDs in various applications. With its support for ATmega32 microcontroller, it offersseamless integration and efficient LED management. The module provides easy-to-use functions for controlling individual LEDs, allowing for dynamic lighting effects and customization. Its compact design and optimized code ensure minimal resource utilization while delivering reliable and precise LED control.

### 2.1.4. SOS

*The SOS (Small OS) module is a lightweight operating system designed for embedded systems, specifically targeting devices with limited resources. It provides a priority-based preemptive scheduler that allows for efficient task execution based on their priority and time-triggered events*

### 2.1.5. TIMER

*The Timer module is responsible for managing timers and providing time-related functionalities in the system. It utilizes hardware timers available on the microcontroller to generate time-triggered events and accurately measure time intervals. The Timer module plays a crucial role in the operation of the system, enabling precise timing for various tasks and events.*

# DRIVERS' DOCUMENTATION

## 3.1 DIO

```
/*   Writes a voltage level to a pin of the DIO interface.
* Parameters|
*   [in] pin The pin to write the voltage level to
*   [in] volt The voltage level to write (HIGH or LOW
* Returns
*
*pin represents the pin to write the voltage level to, and volt represents the voltage
*level to write (HIGH or LOW).
*
*
* Parameters|          [in] pin The pin to write the voltage level to
*                      [in] volt The voltage level to write (HIGH or LOW).
* Returns
*       none
*/
void DIO_writepinn   (DIO_Pin_type   pin,DIO_PinVoltage_type volt)


/*   Reads the voltage level from a pin of the DIO interface.
*
*pin represents the pin to read the voltage level from, and volt is a pointer to store
*the read voltage level. The function reads the voltage level from the specified pin
*and stores it in the memory location pointed to by volt.
*
*
* Parameters|          [in] pin The pin to read the voltage level from.
*                      [out] volt Pointer to store the read voltage level.
* Returns
*       none
*/

void DIO_readpinn   (DIO_Pin_type   pin,DIO_PinVoltage_type *volt)

/*   Reads the voltage level from a pin of the DIO interface.
*
*pin represents the pin to read the voltage level from, and volt is a pointer to store
*the read voltage level. The function reads the voltage level from the specified pin
*and stores it in the memory location pointed to by volt.
*
* Parameters|          [in] pin The pin to toggle.
* Returns
*      none
*/
void DIO_togllepin (DIO_Pin_type   pin)
```

```
/*   Initializes a pin of the DIO interface with a given status.
*
*The function DIO_initpinn initializes a pin of the DIO interface with a given status.
*It takes two parameters: pin, which represents the pin number, and status, which
*represents the desired status of the pin (OUTPUT, INFREE, or INPULL).
*
 *  Parameters
 *    [in] pin The The pin number.
 *  [in] status The status of the pin (OUTPUT, INFREE, or INPULL).
* Returns
*  none
 */
void DIO_initpinn   (DIO_Pin_type   pin,DIO_PinStatus_type status)
```

## 3.2 LED

```
/**
 * Initializes LED on given port & pin
 * @param ledPort [in] LED Port
 * @param ledPin [in] LED Pin number in ledPort
 */
void LED_init( Uchar8_t ledPin)
```

```
/**
 * Turns on LED at given port/pin
 * @param ledPort [in] LED Port
 * @param ledPin [in] LED Pin number in ledPort
 */
void LED_on(Uchar8_t ledPin);
```

```
/**
 * Turns off LED at given port/pin
 * @param ledPort [in] LED Port
 * @param ledPin [in] LED Pin number in ledPort
*/
void LED_off(Uchar8_t ledPin);
```

```
/**
 * Toggles LED at given port/pin
 * @param ledPort [in] LED Port
 * @param ledPin [in] LED Pin number in ledPort
 */
void LED_toggle(Uchar8_t ledPin);
```

## 3.3 BUTTON

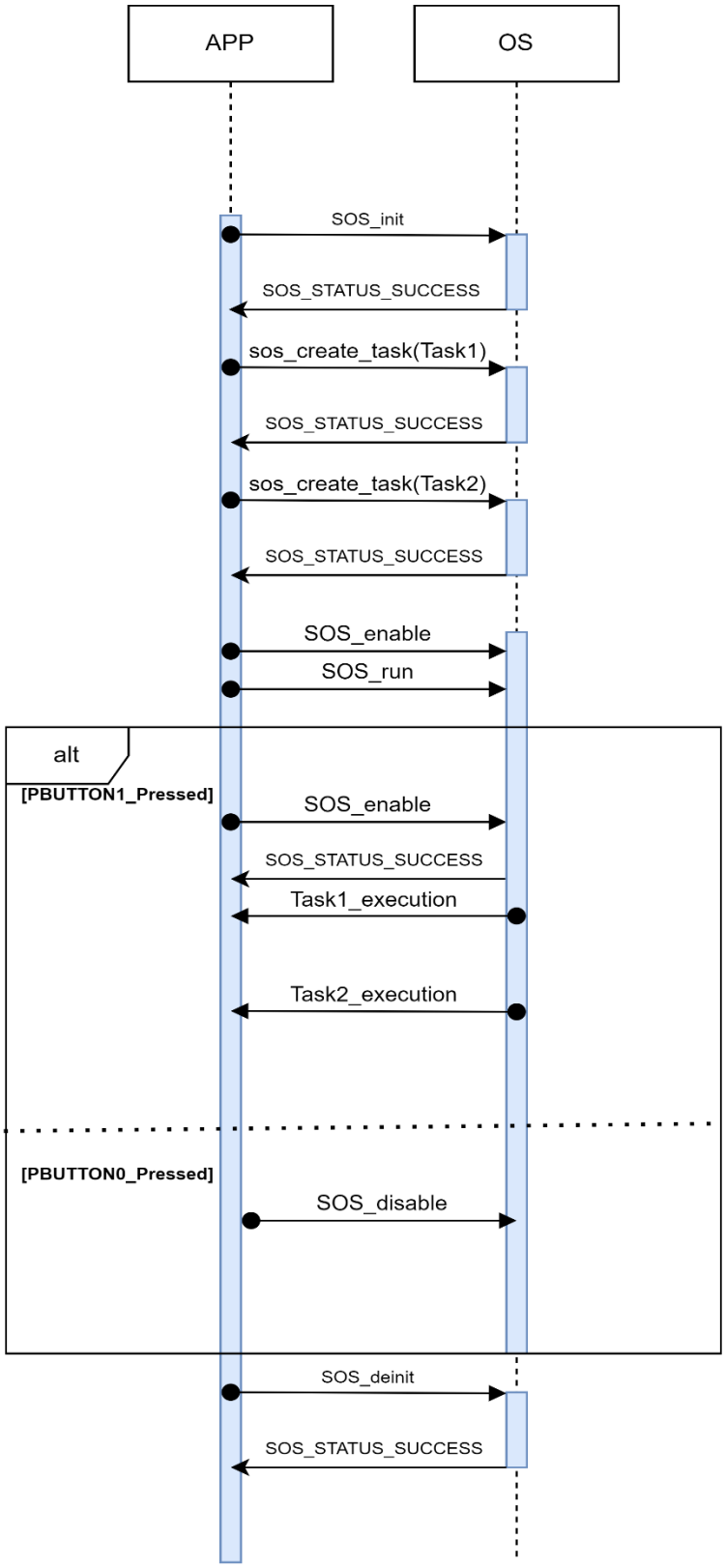
```
/*
*Description: Initializes a push button based on the configuration settings specified in the input
*parameter.
*Parameters:
*- btn : A pointer to an ST_PUSH_BTN_t struct that contains the configuration settings for the push
*button.
*
*Return Type: Std_ReturnType. This is a standard type used in AUTOSAR (Automotive Open System
*Architecture)
*               software development to indicate the success or failure of a function call.
*The possible return values for this function are:
*- E_OK               : The function has completed successfully.
*- E_NOT_OK : The function has encountered an error and could not complete successfully.
*/
void PUSH_BTN_intialize();
/*
Description  : Reads the current en_g_state of a push button and returns its value.
*
*Parameters:
*- btn          : A pointer to an ST_PUSH_BTN_t struct that contains the configuration settings and
*current en_g_state
*                       information for the push button.
*- btn_state : A pointer to an EN_PUSH_BTN_state_t enum where the current en_g_state of the push
*button
*                       will be stored.
*Return Type  : Std_ReturnType. This is a standard type used in AUTOSAR (Automotive Open System
*Architecture)
*                       software development to indicate the success or failure of a function call.
*The possible return values for this function are:
*- E_OK               : The function has completed successfully.
*- E_NOT_OK  : The function has encountered an error and could not complete successfully.
*/
void PUSH_BTN_read_state(Uchar8_t btnNumber, EN_PUSH_BTN_state_t *btn_state);
```

## 3.4 TIMER

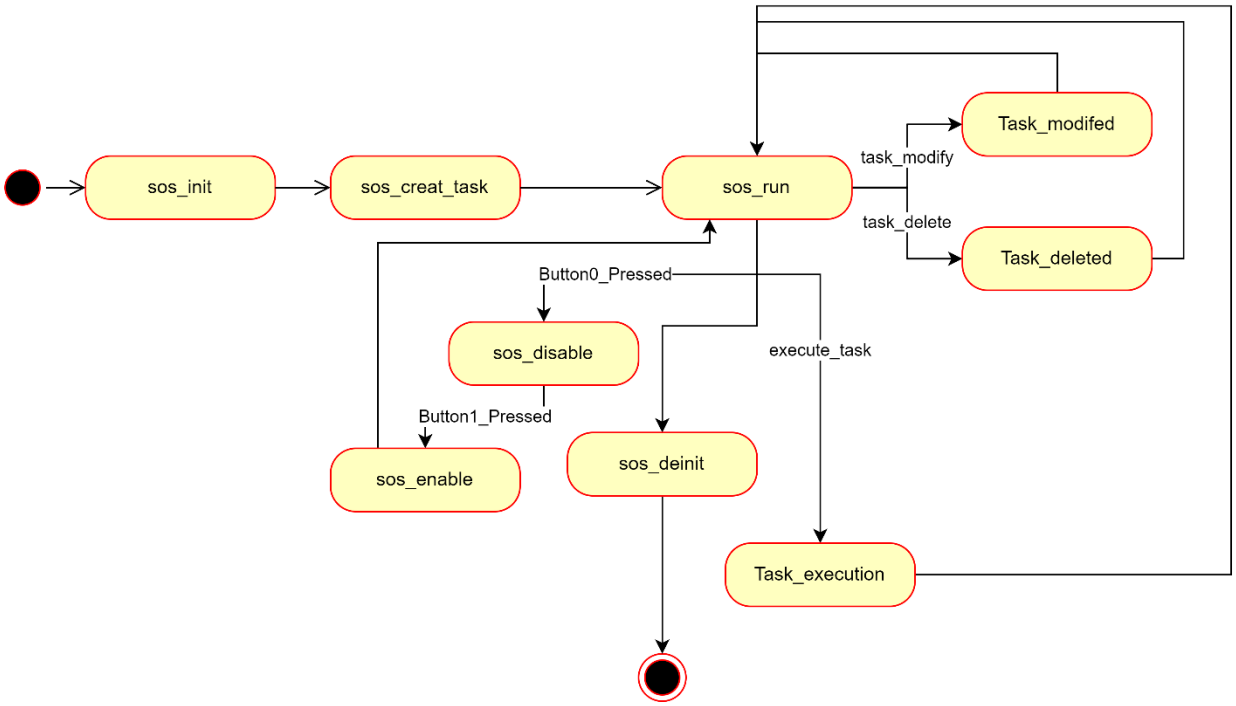
```
/* Initialises timer2 at normal mode
*
* This function initialises/selects the timer_2 normal mode for the
*  timer, and enables the ISR for this timer.
* Parameters
*            [in] en_a_interrputEnable value to set the interrupt
*                                      bit for timer_2 in the TIMSK reg.
* Return
*            An EN_TIMER_ERROR_T value indicating the success or failure of
*            the operation (TIMER_OK if the operation succeeded, TIMER_ERROR
*            otherwise)
*/
EN_TIMER_ERROR_T
TIMER_timer2NormalModeInit(EN_TIMER_ERROR_T   en_a_interrputEnable);


| Start the timer by setting the desired prescaler.
|
| This function sets the prescaler for timer_2.
| Parameters
|            [in] u16_a_prescaler value to set the desired prescaler.
| Return
|            An EN_TIMER_ERROR_T value indicating the success or failure of
|            the operation
|             (TIMER_OK if the operation succeeded, TIMER_ERROR otherwise)
|
EN_TIMER_ERROR_T TIMER_timer2Start(Uint16_t u16_a_prescaler);

| Stop the timer by setting the prescaler to be 000--> timer is stopped.
|
| This function clears the prescaler for timer_2.
|
| Return
|      void
|
void TIMER_timer2Stop(void);
```

# UML

## 4.1 SEQUENCE DIAGRAM



## 4.2 STATE MACHINE



# 4.3 CLASS DIGRAM