CSUN | COLLEGE OF ENGINEERING AND COMPUTER SCIENCE

ECE 425L

Microprocessor Systems Lab

Final Project
3D Printed Blooming Floral Design

Macy Varga
Vedi Vartani
Instructor: Aaron Nanas
Spring 2025

Table of Contents

1. Introduction

This project presents the design and implementation of a 3D-printed flower capable of realistically blooming through precise, vertical petal motion driven by a servo motor. Initially controlled via a manual button input, the system evolves into a more dynamic and autonomous design by incorporating an ambient light sensor that triggers blooming.

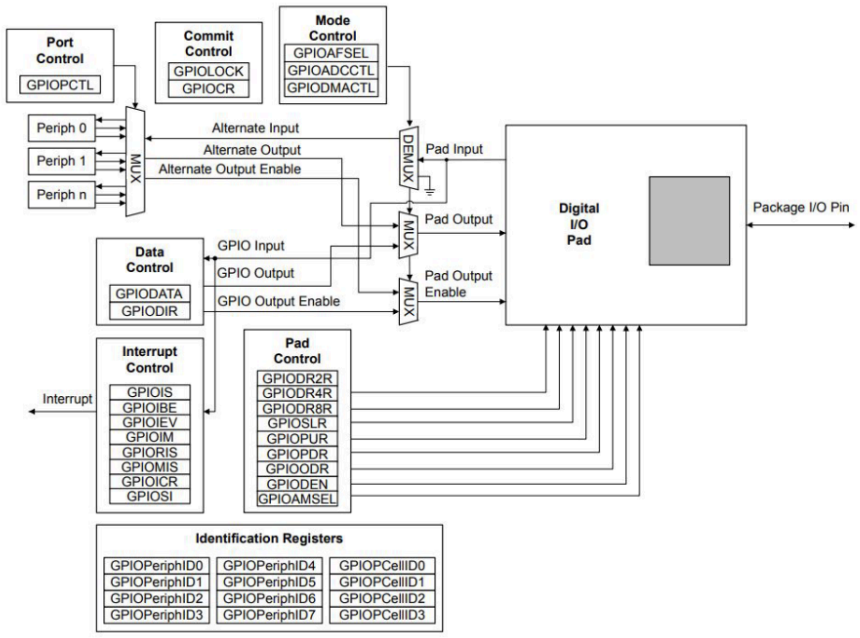
The flower's movement is achieved by converting the servo's rotary motion into vertical displacement of a central axle. This axle manipulates the petal positions: when lowered, the petals converge to represent a closed state; when raised, the petals spread open to simulate blooming. Controlled by the TM4C123 microcontroller, the petal motion is smooth, repeatable, and finely tuned using embedded PWM signals.

The project integrates key embedded systems concepts taught in ECE 425, including GPIO interfacing, interrupt handling, SPI communication, and PWM-based motor control. The final prototype demonstrates both manual and automatic modes, combining mechanical design and microcontroller based control to mimic natural flower behavior.

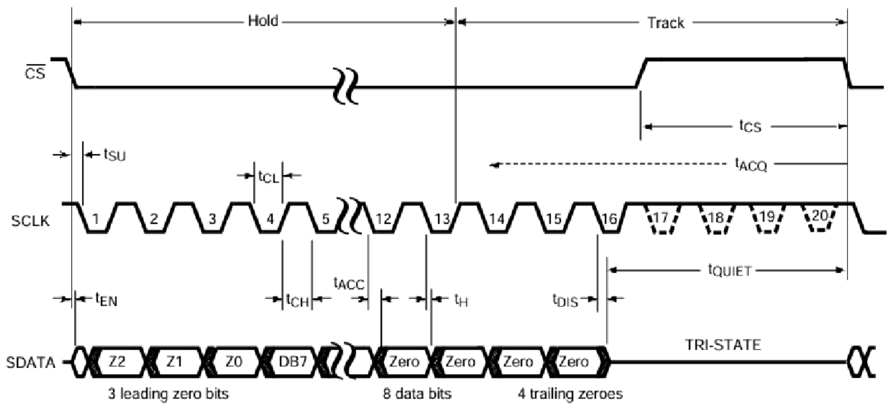
2. Background and Methodology

This project implements several embedded systems concepts learned in ECE 425, including GPIO control, SPI communication, PWM signal generation, and interrupt-driven event handling, to design a 3D-printed flower capable of blooming motion.

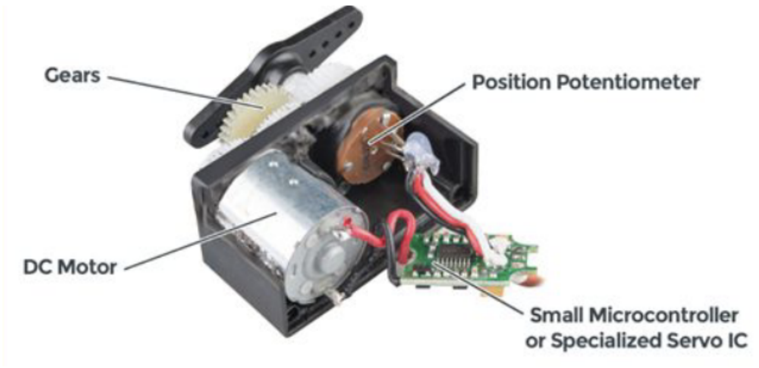
The TM4C123GH6PM microcontroller provides General-Purpose Input/Output (GPIO) functionality to interact with external devices. GPIO Port A pins (PA2–PA5) are configured as digital inputs to read the button states from the PMOD BTN module. Pull-down resistors ensure a known low default state, while interrupts are configured to trigger on rising edges, enabling immediate responses to button presses without needing constant polling.

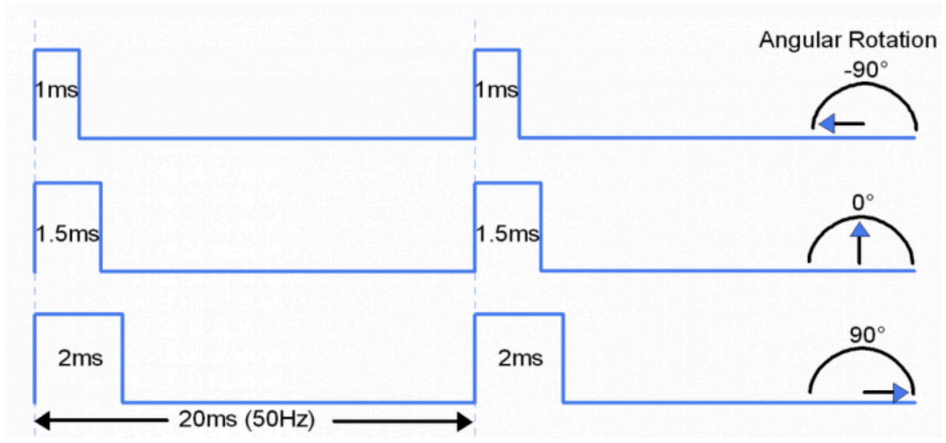


SPI (Serial Peripheral Interface) communication is used to interface with the PMOD ALS ambient light sensor. The TM4C123GH6PM's SSI2 peripheral is configured in SPI Mode 0 (Clock Polarity = 0, Clock Phase = 0), using the PB4 (SCLK), PB5 (CS), and PB6 (MISO) pins. MOSI is unused in this application. The light sensor outputs an 8-bit digital value representing ambient brightness, which is captured through manual control of the chip select (CS) pin.

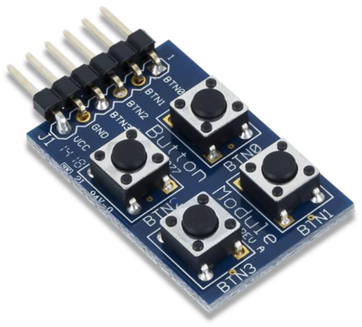


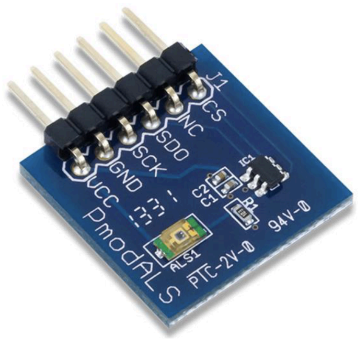
The blooming motion is achieved through a HS-485HB servo motor controlled by PWM signals. PWM Module 1, Generator 3 (M1PWM6 output on PF2), produces a 50Hz signal suitable for servo control. The duty cycle determines the angular position of the servo, with calibrated values ranging from approximately 553 µs (representing 0°) to 2425 µs (representing ~190°). Specific duty cycle values (e.g., 1875 and 3590 in clock ticks) are selected to create repeatable open and closed flower states.





Input handling differs depending on whether manual (PMOD BTN) or automatic (PMOD ALS) mode is selected at compile time using macros (#define USE_PMOD_BTN or #define USE_PMOD_ALS). Manual Mode (Button-Controlled): When a button is pressed, an interrupt is triggered, and a user-defined interrupt service routine (ISR) adjusts the PWM duty cycle based on the button pressed, allowing discrete control of flower positions. Automatic Mode (Light-Controlled): When ambient light exceeds a certain threshold, determined through periodic sampling via the PMOD ALS, the servo duty cycle is adjusted automatically to bloom the flower; otherwise, the flower remains closed.

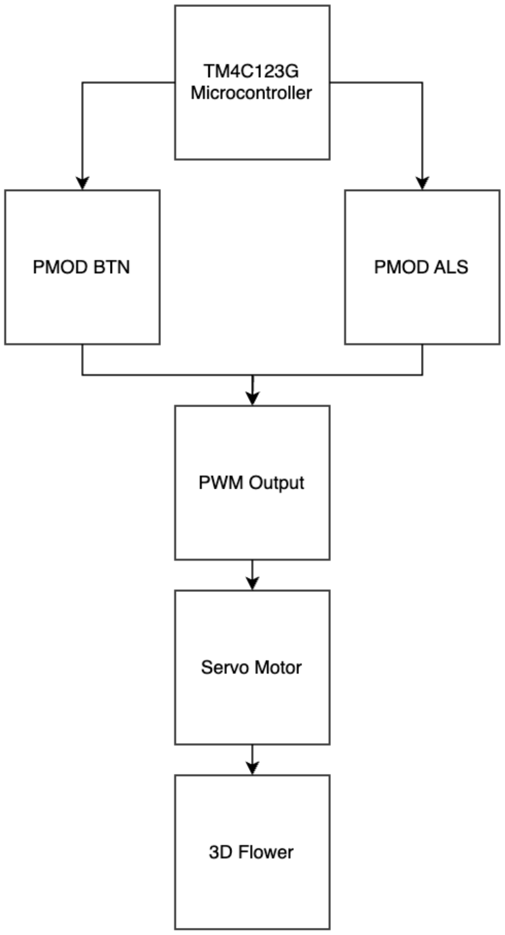




Precise timing throughout the system is managed using the SysTick timer configured for 1-microsecond intervals. Blocking delay functions provide fine-grained control for operations such as LED updates and sensor reads.

Overall, the project demonstrates integration of GPIO manipulation, interrupt-based input handling, SPI-based sensor communication, and PWM-based motor control to achieve a realistic, automatic blooming effect for the 3D-printed flower.

3. Block Diagram



4. Components Used

Description	Quantity	Manufacturer
Tivia C Series TM4C123G LaunchPad	1	Texas Instruments
USB-A to Micro-USB Cable	1	N/A
EduBase Board	1	Trainer4Edu
PMOD ALS (Ambient Light Sensor)	1	Digilent
PMOD BTN Module	1	Diligent
HS-485HB Servo	1	ServoCity

4.1 Software Used

Download Version	Software
MDK-arm / Keil µVision5	arm KEIL
Rhino 8	Rhinoceros

5. Pinout Used

5.1 PMOD BTN Pinout

Ensure the TM4C123G LaunchPad is off. Then connect the PMOD BTN module using a breadboard and jumper wires. Refer to the following pinout.

PMOD BTN Pin	TM4C123G Launchpad Pin
BTN0	PA2
BTN1	PA3
BTN2	PA4
BTN3	PA5
Pin 5 (GND)	GND
Pin 6 (VCC)	VCC (3.3V)

5.2 PMOD ALS Pinout

Ensure the TM4C123G LaunchPad is off. Then connect the PMOD ALS module using a breadboard and jumper wires. Refer to the following pinout.

PMOD ALS Pin	TM4C123G Launchpad Pin
Pin 1 (CS)	PB5
Pin 2 (Not Connected)	Not Connected
Pin 3 (MISO)	PB6
Pin 4 (SCLK)	PB4
Pin 5 (GND)	GND
Pin 6 (VCC)	VCC (3.3V)

5.3 HS-485HB Servo Pinout

Ensure the TM4C123G LaunchPad is off. Make the following connections on a breadboard using jumper wires. Use the 5V output pin from the J5 connector on the EduBase board. Refer to the following pinout.

HS-485HB Servo	TM4C123G Launchpad Pin
Servo PWM Input Signal (Yellow Wire)	PF2
Servo VCC (Red Wire)	5V
Servo GND (Black Wire)	GND

6. Analysis and Results

To develop the blooming flower system, we wrote our main application logic in main.c, which serves as the central control point for either button-based or light-based operation. At the top of the file, we used macros to choose between these two modes: #define USE_PMOD_BTN enables manual control using the PMOD BTN module, while #define USE_PMOD_ALS activates automatic control using the PMOD ALS light sensor. Only one mode should be active at a time to prevent conflicts in the initialization and runtime logic.

After defining the mode, the program begins with initializing essential modules. The SysTick_Delay_Init() function sets up the system timer for microsecond-level delays, which we use throughout the code for timing operations. Next, we initialize PWM output on pin PF2 using PWM1_3_Init(), which creates a 50Hz PWM signal for controlling the servo motor. The initial duty cycle is set to 1875 clock ticks, corresponding to the "closed" flower position. We also initialize the EduBase LEDs with EduBase_LEDs_Init() to provide visual feedback based on input or sensor values.

In manual mode, we call PMOD_BTN_Interrupt_Init() and pass in a handler function that responds to button presses. The buttons are connected to PA2 through PA5, and interrupts are triggered on rising edges. The handler, PMOD_BTN_Handler(), sets the servo angle by adjusting the PWM duty cycle. For example, pressing BTN0 causes the flower to close, while BTN1 makes it bloom to about 90 degrees. In this mode, all interaction happens through interrupts, and the main loop remains idle.

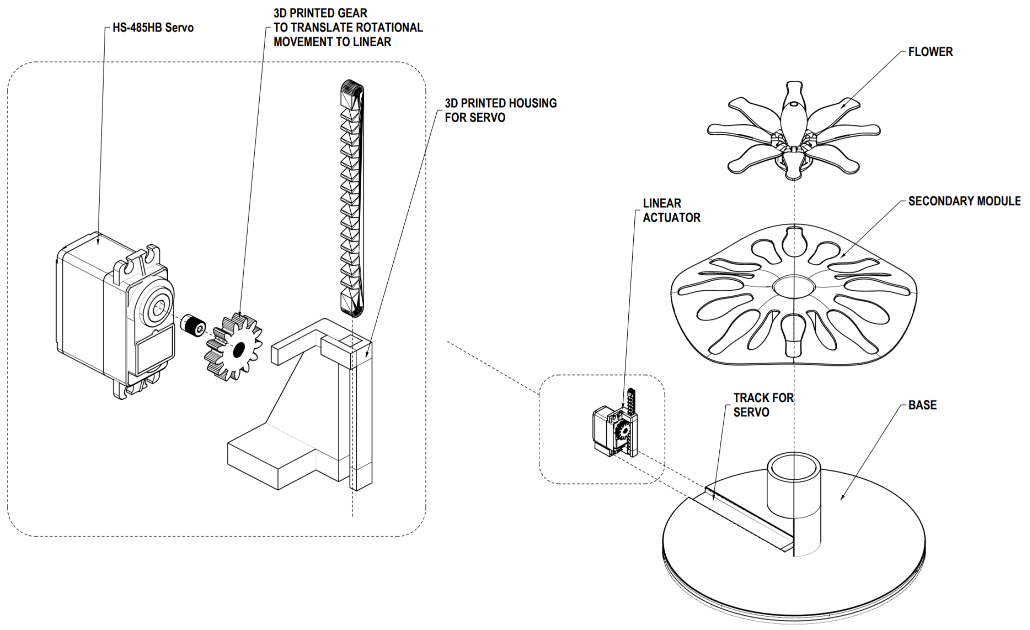
In automatic mode, SSI2_Init() sets up SPI communication with the PMOD ALS sensor. This involves configuring PB4 for the clock (SCLK), PB6 for data input (MISO), and PB5 for chip select (CS). The CS line is controlled manually in our code. Inside the main loop, we repeatedly call PMOD_ALS_Read() every 300 milliseconds to sample the light intensity. The sensor returns an 8-bit value, and if this value exceeds 100, we instruct the servo to open the flower; otherwise, it stays closed. We also output the top 4 bits of the light level to the EduBase LEDs so we can visualize brightness changes directly on the board.

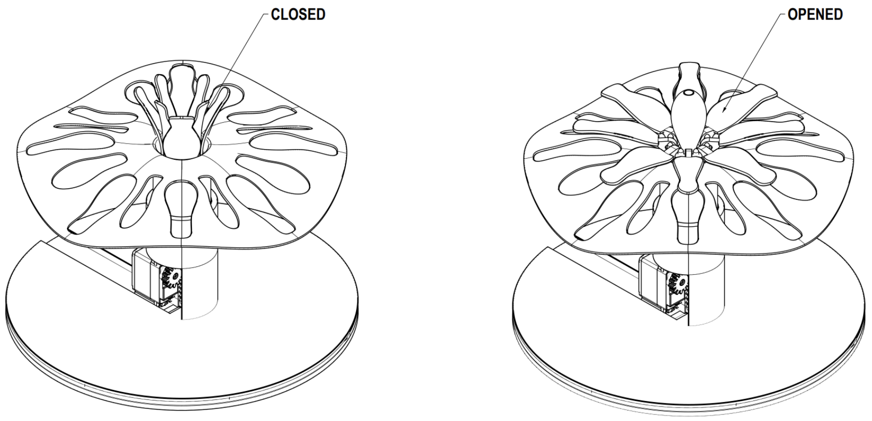
All delays in the code use the SysTick timer through blocking functions like SysTick_Delay1ms() or SysTick_Delay1us(), which are helpful for stabilizing sensor reads or pacing the system. The rest of the project is split into individual source files: PWM1_3.c handles PWM setup and updates, SysTick_Delay.c manages timing, GPIO.c provides LED and button utilities, and PMOD_BTN_Interrupt.c and SSI2.c take care of peripheral interaction. This modular approach makes the system more maintainable and easier to understand.

To run the code, we simply select the desired mode by editing the macro in main.c, compile the project using Keil uVision or another supported IDE, and flash it to the TM4C123G LaunchPad. Depending on the mode, we can then either press buttons or shine a flashlight to trigger the blooming motion. The system demonstrates practical use of GPIO interrupts, SPI communication, and PWM motor control—all of which we learned throughout the semester.

The process of designing the 3D model was very much informed by our intention and needs of the project. Starting from the initial translation of the trigger to a physical and mechanical response. In order to translate the rotary movement of the servo we had to design a linear actuator translator kit to take the rotary movement of the servo and output the desired linear up and down movement required for the flower to “bloom”. We used Rhino 3D modeling software to realize the design and fabrication of the 3D parts, including the housing for the servo, the base for the model and the shaft and piston required for the mechanical flower to function.

The challenges we faced during this process was mainly due to the 3D printing limitations of the printers we have on campus and the time constraints due to outsourcing the fabrication forcing us to improvise to deliver a functioning project but a work in progress for further improvements, including an integrated housing for the micro controller and 3D printed flower portion using high precision resin printers outside of the school.


HS-485HB Servo
3D PRINTED GEAR TO TRANSLATE ROTATIONAL MOVEMENT TO LINEAR
3D PRINTED HOUSING FOR SERVO
FLOWER
LINEAR ACTUATOR
SECONDARY MODULE
TRACK FOR SERVO
BASE



Video Demonstrations:

Automatic mode: https://www.youtube.com/shorts/-aOCNeL3odc

Manual Mode: https://www.youtube.com/shorts/sJubQrR22dA

7. Conclusion

This project gave us a great opportunity to apply what we learned in ECE 425 to something creative and hands-on. By combining 3D printing with embedded systems, we were able to build a flower that could realistically bloom using servo motor control. Starting with manual button activation and later expanding to automatic light-based control helped us explore both GPIO interrupts and SPI communication in real-world applications.

We used PWM signals from the TM4C123 microcontroller to control the servo and designed a custom mechanism to convert its rotation into vertical motion, allowing the petals to open and close. Along the way, we faced challenges like mechanical design adjustments and limited access to high-quality 3D printing, but we managed to adapt and still deliver a working prototype.

Overall, this project helped reinforce core concepts from the course like interrupt handling, sensor communication, and timing control. In addition, it showed us how hardware and software can come together to create something interactive. There's definitely room for future improvements, especially in refining the physical design and housing the electronics, but we're proud of how far we got.

8. Resources

Tiva C Series TM4C123G LaunchPad Evaluation Board User's Guide
https://www.ti.com/lit/pdf/spmu296

Tiva TM4C123GH6PM Microcontroller Datasheet
https://www.ti.com/lit/gpn/TM4C123GH6PM

Getting Started with Keil MDK
http://www.keil.com/gsg

GNU C library Reference Manual
https://www.gnu.org/software/libc/manual/pdf/libc.pdf

PMOD BTN Reference Manual
https://digilent.com/reference/pmod/pmodbtn/reference-manual

HS-485HB Servo-Stock Rotation Datasheet

https://www.servocity.com/hs-485hb-servo/

TEMT6000X01 Ambient Light Sensor Datasheet

https://www.vishay.com/docs/81579/temt6000.pdf

9. Code

Main.c

```
/**
 * @file main.c
 *
 * @brief Main application entry point.
 *
 * This file demonstrates the use of either the PMOD BTN or PMOD ALS (SSI2)
 * module on the TM4C123G LaunchPad to control a servo via PWM. Only one
 * module should be enabled at a time using the corresponding macro.
 *
 * @author Macy Varga
 */

#include "PWM1_3.h"
#include "SysTick_Delay.h"
#include "TM4C123GH6PM.h"
#include "GPIO.h"


// Uncomment one of the following to select the active module:
// dont forget under project ->Target 1-> C/C++-> Define USE_PMOD_ALS or
USE_PMOD_BTN
//#define USE_PMOD_BTN
 #define USE_PMOD_ALS


#ifdef USE_PMOD_BTN
#include "PMOD_BTN_Interrupt.h"
#endif

#ifdef USE_PMOD_ALS
#include "SSI2.h"
#endif
```

```
//(used if PMOD_BTN interrupts are active)
volatile uint8_t move_servo = 0;

#ifdef USE_PMOD_BTN
/**
 * @brief Handler function for PMOD BTN interrupts.
 *
 * This function is triggered when a button on the PMOD BTN module is
pressed.
 * It adjusts the PWM duty cycle based on which button is pressed.
 *
 * @param pmod_btn_status Status of the PMOD BTN input (bits PA2-PA5).
 */
void PMOD_BTN_Handler(uint8_t pmod_btn_status)
{
    switch (pmod_btn_status)
    {
        case 0x04: // BTN0 (PA2)
            PWM1_3_SetDuty(1875);
            break;

        case 0x08: // BTN1 (PA3)
            PWM1_3_SetDuty(3590); // 90 degrees
            break;

        case 0x10: // BTN2 (PA4)
            PWM1_3_SetDuty(0);
            break;

        case 0x20: // BTN3 (PA5)
            // No action defined
            break;

        default:
            break;
    }
}
#endif
```

```
int main(void)
{
    SysTick_Delay_Init();
    PWM1_3_Init(62500, 1875); // 50Hz PWM, start at 0 degrees
    EduBase_LEDs_Init();     // Optional: Visual feedback

#ifdef USE_PMOD_BTN
    PMOD_BTN_Interrupt_Init(&PMOD_BTN_Handler);
#endif

#ifdef USE_PMOD_ALS
    SSI2_Init();


#endif

    while (1)
    {
#ifdef USE_PMOD_ALS
        uint8_t light_level = PMOD_ALS_Read();
                  EduBase_LEDs_Output(light_level >> 4); // Visualize light
level on LEDs


        if (light_level > 100)
        {
            PWM1_3_SetDuty(3590); // 90 degrees
        }
        else
        {
            PWM1_3_SetDuty(1875); // 0 degrees
        }

        SysTick_Delay1ms(300); // Delay 300ms
#endif

#ifdef USE_PMOD_BTN
        // All logic handled in ISR, loop can be idle or extended
#endif
    }
}
```

GPIO.c

```
/**
* @file GPIO.c
*
* @brief Source code for the GPIO driver.
*
* This file contains the function definitions for the GPIO driver.
* It interfaces with the following:
*  - User LED (RGB) Tiva C Series TM4C123G LaunchPad
*  - EduBase Board LEDs (LED0 - LED3)
*  - EduBase Board Push Buttons (SW2 - SW5)
*
* To verify the pinout of the user LED, refer to the Tiva C Series TM4C123G
LaunchPad User's Guide
* Link: https://www.ti.com/lit/pdf/spmu296
*
* @author Aaron Nanas, Edited by Macy Varga
*/


#include "GPIO.h"


// Constant definitions for the user LED (RGB) colors
const uint8_t RGB_LED_OFF      = 0x00;
const uint8_t RGB_LED_RED      = 0x02;
const uint8_t RGB_LED_BLUE     = 0x04;
const uint8_t RGB_LED_GREEN    = 0x08;


// Constant definitions for the EduBase board LEDs
const uint8_t EDUBASE_LED_ALL_OFF = 0x0;
const uint8_t EDUBASE_LED_ALL_ON    = 0xF;


void RGB_LED_Init(void)
```

```
{
   // Enable the clock to Port F
   SYSCTL->RCGCGPIO |= 0x20;


   // Set PF1, PF2, and PF3 as output GPIO pins
   GPIOF->DIR |= 0x0E;

   // Configure PF1, PF2, and PF3 to function as GPIO pins
   GPIOF->AFSEL &= ~0x0E;

   // Enable digital functionality for PF1, PF2, and PF3
   GPIOF->DEN |= 0x0E;

   // Initialize the output of the RGB LED to zero
   GPIOF->DATA &= ~0x0E;
}


void RGB_LED_Output(uint8_t led_value)
{
   // Set the output of the RGB LED
   GPIOF->DATA = (GPIOF->DATA & 0xF1) | led_value;
}


uint8_t RGB_LED_Status(void)
{
   // Assign the value of Port F to a local variable
   // and only read the values of the following bits: 3, 2, and 1
   // Then, return the local variable's value
   uint8_t RGB_LED_Status = GPIOF->DATA & 0x0E;
   return RGB_LED_Status;
}


void EduBase_LEDs_Init(void)
{
   // Enable the clock to Port B
   SYSCTL->RCGCGPIO |= 0x02;
```

```
    // Set PB0, PB1, PB2, and PB3 as output GPIO pins
    GPIOB->DIR |= 0x0F;

    // Configure PB0, PB1, PB2, and PB3 to function as GPIO pins
    GPIOB->AFSEL &= ~0x0F;

    // Enable digital functionality for PB0, PB1, PB2, and PB3
    GPIOB->DEN |= 0x0F;

    // Initialize the output of the EduBase LEDs to zero
    GPIOB->DATA &= ~0x0F;
}


void EduBase_LEDs_Output(uint8_t led_value)
{
    // Set the output of the LEDs
    GPIOB->DATA = (GPIOB->DATA & 0xF0) | led_value;
}


void EduBase_Button_Init(void)
{
    // Enable the clock to Port D
    SYSCTL->RCGCGPIO |= 0x08;

    // Set PD0, PD1, PD2, and PD3 as input GPIO pins
    GPIOD->DIR &= ~0x0F;

    // Configure PD0, PD1, PD2, and PD3 to function as GPIO pins
    GPIOD->AFSEL &= ~0x0F;

    // Enable digital functionality for PD0, PD1, PD2, and PD3
    GPIOD->DEN |= 0x0F;
}


uint8_t Get_EduBase_Button_Status(void)
{
    // Assign the value of Port D to a local variable
    // and only read the values of the following bits: 3, 2, 1, and 0
```

```
   // Then, return the local variable's value
   uint8_t button_status = GPIOD->DATA & 0x0F;
   return button_status;
}
```

SysTick_Delay.c

```
/**
 * @file SysTick_Delay.c
 *
 * @brief Source code for the SysTick_Delay driver.
 *
 * It provides two blocking functions, SysTick_Delay1ms and
SysTick_Delay1us,
 * to create a delay with a busy-wait loop. It uses the SysTick timer with
 * a specified reload value to generate interrupts every 1 us.
 *
 * In addition, it uses the Peripheral Internal Oscillator (PIOSC)
 * as the clock source. The PIOSC provides 16 MHz which is then divided by
4.
 * The timer is used for creating delays in either microseconds or
milliseconds.
 *
 * @author Aaron Nanas, Edited by Macy Varga
 */

#include "SysTick_Delay.h"

// Global variable used to keep track of elapsed time in microseconds
static uint32_t us_elapsed = 0;

// Global variable used to keep track of elapsed time in milliseconds
static uint32_t ms_elapsed = 0;

// Global flag used to indicate if milliseconds delay is active
static uint8_t ms_active = 0;

void SysTick_Delay_Init(void)
{
      // Set the SysTick timer reload value for 1 us intervals
	// Each clock cycle is (1 / 4 MHz) = 0.25 us
	SysTick->LOAD = (4 - 1);

	// Clear the VAL register by writing any value to it
	SysTick->VAL = 0;

	// Enable the SysTick timer and its interrupt
	// with the Peripheral Internal Oscillator (PIOSC) as the clock 
source
	SysTick->CTRL |= 0x03;
}

void SysTick_Delay1us(uint32_t delay_in_us)
{
	// Reset the global variable, us_elapsed
	us_elapsed = 0;

	// Wait until ms_value reaches the specified delay_in_ms
	while (delay_in_us > us_elapsed);
}

void SysTick_Delay1ms(uint32_t delay_in_ms)
{
	// Reset the global variables, us_elapsed and ms_elapsed
	us_elapsed = 0;
	ms_elapsed = 0;

	// Set the ms_active global flag
	ms_active = 0x01;

	// Wait until ms_elapsed reaches the specified delay_in_ms
	while (delay_in_ms > ms_elapsed);

	// Reset the ms_active global flag
	ms_active = 0x00;
}

void SysTick_Handler(void)
{
	// Increment the global variable, us_elapsed
	us_elapsed = us_elapsed + 1;
```

```
      // Check if us_elapsed has reached 1000 (1 millisecond) and if
milliseconds delay is active
      if (us_elapsed == 1000 && (ms_active == 0x01))
      {
            // Reset us_elapsed
            us_elapsed = 0;

            // Increment ms_elapsed to indicate that 1 millisecond has
passed
            ms_elapsed = ms_elapsed + 1;
      }
}
```

PWM1_3.c

```
/**
* @file PWM1_3.c
*
* @brief Source file for the PWM1_3 driver.
*
* This file contains the function implementations for controlling PWM
Generator 3
* on PWM Module 1. It provides functionality to initialize PWM on PF2
(M1PWM6)
* and adjust the duty cycle to control devices like servo motors.
*
* @author Macy Varga
*/


#include "PWM1_3.h"
#include "TM4C123GH6PM.h"


//Initialize the PWM Generator 3 on Module 1 for use with PF2 as output
(M1PWM6)
void PWM1_3_Init(uint16_t period, uint16_t duty)
 {
   if (duty >= period) return;
   //Enable PWM1
   SYSCTL->RCGCPWM |= (1 << 1);
   //Enable clock for PortF
   SYSCTL->RCGCGPIO |= (1 << 5);


   //Use PWM divider
   SYSCTL->RCC |= 0x00100000;
   SYSCTL->RCC &= ~0x000E0000;
   //Sets the PWM clock divider by writing 0x3 to the PWMDIV field (Bits
19-17) in the RCC.
   // That divides the system clock by 64, givinga PWM clock of 3.125 MHz
assuming a base of 200MHz
   SYSCTL->RCC |= 0x00060000;



   // Enables alternate fn mode on PF2 , required for PWM functionality
(M1PWM6).
   //Bit 2 corresponds to PF2
   GPIOF->AFSEL |= (1 << 2);
   //Clears the bits 11:8 in the PCTL for PF2
   //Selects the alt fn for that pin
   GPIOF->PCTL &= ~0x00000F00;
   //sets bits 11:8 in PCTL to 0x5, selecting the M1PWM6 function (PF2's
PWM
   //function) according to the datasheet
   GPIOF->PCTL |= 0x00000500;
   //Digitally enables PF2 so it can output signals, including the PWM wave
   GPIOF->DEN |= (1 << 2);


   // Disable generator
   PWM1->_3_CTL = 0;
   // config the output behavior: output is HIGH on LOAD and LOW on
   //CMPA down count. This sets up a standard PWM waveform
   PWM1->_3_GENA |= 0x000000C8;
   //Sets the total period of the PWM signal (used to get 50Hz). LOAD-1 is
   //required because the timer counts from LOAD to 0
   PWM1->_3_LOAD = period - 1;
   //pulse width (duty cycle). The output will go low when the counter
```

```
    // reaches CMPA
    PWM1->_3_CMPA = duty - 1;
    // Enable generator
    PWM1->_3_CTL |= 0x01;
    // Enables PWM output on M1PWM6 (PF2). Bit 6 corresponds to PWM1
    //output 6.
    PWM1->ENABLE |= (1 << 6);
}


// Updates the duty cycle for PWM without reinitializing the whole PWM
module
void PWM1_3_SetDuty(uint16_t duty)
{
    // duty value doesn't exceed the period; PWM would not function
correctly
    //if this check fails
    PWM1->_3_CMPA = duty - 1;
}
```

PMOD_BTN_Interrupt.c

```
/**
* @file PMOD_BTN_Interrupt.c
*
* @brief Source file for the PMOD_BTN_Interrupt driver.
*
* This file contains the function implementations for configuring the PMOD
BTN
* module using GPIO Port A pins (PA2 - PA5) to trigger interrupts on button
presses.
* It includes functions to initialize interrupts, read button status, and
handle GPIO interrupts.
*
* @author Macy Varga
*/


#ifdef USE_PMOD_BTN
```

```c
#include "PMOD_BTN_Interrupt.h"
// Declare pointer to the user-defined task
void (*PMOD_BTN_Task)(uint8_t pmod_btn_state);


void PMOD_BTN_Interrupt_Init(void(*task)(uint8_t))
{
   // Store the user-defined task function for use during interrupt
handling
   PMOD_BTN_Task = task;

   // Enable the clock to Port A by setting the
   // R0 bit (Bit 0) in the RCGCGPIO register
   SYSCTL->RCGCGPIO |= 0x01;

   // Configure the PA5, PA4, PA3, and PA2 pins as input
   // by clearing Bits 5 to 2 in the DIR register
   GPIOA->DIR &= ~0x3C;

   // Configure the PA5, PA4, PA3, and PA2 pins to function as
   // GPIO pins by clearing Bits 5 to 2 in the AFSEL register
   GPIOA->AFSEL &= ~0x3C;

   // Enable the digital functionality for the PA5, PA4, PA3, and PA2 pins
   // by setting Bits 5 to 2 in the DEN register
   GPIOA->DEN |= 0x3C;

   // Enable the weak pull-down resistor for the PA5, PA4, PA3, and PA2
pins
   // by setting Bits 5 to 2 in the PDR register
   GPIOA->PDR |= 0x3C;

   // Configure the PA5, PA4, PA3, and PA2 pins to detect edges
   // by clearing Bits 5 to 2 in the IS register
   GPIOA->IS &= ~0x3C;

   // Allow the GPIOIEV register to handle interrupt generation
   // and determine which edge to check for the PA5, PA4, PA3, and PA2 pins
   // by clearing Bits 5 to 2 in the IBE register
   GPIOA->IBE &= ~0x3C;
   // Configure the PA5, PA4, PA3, and PA2 pins to detect
   // rising edges by setting Bits 5 to 2 in the IEV register
   // Rising edges on the corresponding pins will trigger interrupts
   GPIOA->IEV |= 0x3C;

   // Clear any existing interrupt flags on the PA5, PA4, PA3, and PA2 pins
   // by setting Bits 5 to 2 in the ICR register
   GPIOA->ICR |= 0x3C;

   // Allow the interrupts that are generated by the PA5, PA4, PA3, and PA2
pins
   // to be sent to the interrupt controller by setting
   // Bits 5 to 2 in the IM register
   GPIOA->IM |= 0x3C;

   // Clear the INTA field (Bits 7 to 5) of the IPR[0] register (PRI0)
   NVIC->IPR[0] &= ~0x000000E0;

   // Set the priority level of the interrupts to 3. Port A has an
Interrupt Request (IRQ) number of 0
   NVIC->IPR[0] |= (3 << 5);

   // Enable IRQ 0 for GPIO Port A by setting Bit 0 in the ISER[0] register
   NVIC->ISER[0] |= (1 << 0);
}


uint8_t PMOD_BTN_Read(void)
{
   // Declare a local variable to store the status of the PMOD BTN
   // Then, read the DATA register for Port A
   // A "0x3C" bit mask is used to capture only the pins used the PMOD BTN
   uint8_t pmod_btn_state = GPIOA->DATA & 0x3C;

   // Return the status of the PMOD BTN module
   return pmod_btn_state;
}


void GPIOA_Handler(void)
```

```
{
   // Check if an interrupt has been triggered by any of
   // the following pins: PA5, PA4, PA3, and PA2
   if (GPIOA->MIS & 0x3C)
   {
       // Execute the user-defined function
       (*PMOD_BTN_Task)(PMOD_BTN_Read());

       // Acknowledge the interrupt from any of the following pins
       // and clear it: PA5, PA4, PA3, and PA2
       GPIOA->ICR |= 0x3C;
   }
}
#endif
```

SSI1.c

```
/**
 * @file SSI2.c
 *
 * @brief Source file for the SSI2 driver.
 *
 * This file contains the function implementations for configuring the SSI2
 * SPI peripheral to communicate with external SPI devices like the PMOD
ALS
 * light sensor module. It includes initialization and sensor data reading
functions.
 *
 * @author Macy Varga
 */

#ifdef USE_PMOD_ALS

#include "SSI2.h"

void SSI2_Init(void)
{
    SYSCTL->RCGCSSI |= (1 << 2);    // Enable SSI2 clock
    SYSCTL->RCGCGPIO |= (1 << 1);   // Enable Port B clock
    volatile int delay = SYSCTL->RCGCGPIO; // small delay for clock ready

    // Configure SCLK (PB4) and MISO (PB6) for alternate function
    GPIOB->AFSEL |= (1 << 4) | (1 << 6);  // Enable alt fn on PB4, PB6
    GPIOB->PCTL &= ~0x0F0F0000;           // Clear PCTL for PB4, PB6
    GPIOB->PCTL |= 0x02020000;           // Set SSI2 function
    GPIOB->DEN |= (1 << 4) | (1 << 6);   // Enable digital for PB4, PB6
    GPIOB->AMSEL &= ~((1 << 4) | (1 << 6)); // Disable analog

    // Configure CS (PB5) as GPIO output (manual control)
    GPIOB->AFSEL &= ~(1 << 5); // Remove alt fn for PB5
    GPIOB->DIR |= (1 << 5);    // Set PB5 as output
    GPIOB->DEN |= (1 << 5);    // Enable digital function for PB5
    GPIOB->DATA |= (1 << 5);   // Deassert CS (active low)

    // Configure SSI2 module
    SSI2->CR1 &= ~0x02;      // Disable SSI during setup
    SSI2->CR1 &= ~0x04;      // Master mode
    SSI2->CC = 0x0;          // Use system clock
    SSI2->CPSR = 100;        // SPI clock = system / 100 = 500 kHz
    SSI2->CR0 = 0x0F;        // 16-bit data, SPI Mode 0 (CPOL=0, CPHA=0)
    SSI2->CR1 |= 0x02;       // Enable SSI2
}

uint8_t PMOD_ALS_Read(void)
{
    // Assert CS (active low)
    GPIOB->DATA &= ~(1 << 5);

    // Write dummy data to generate clock
    while ((SSI2->SR & 0x02) == 0); // Wait until TX FIFO not full
    SSI2->DR = 0xAAAA;              // Dummy data

    // Wait for data to be received
    while ((SSI2->SR & 0x04) == 0); // Wait until RX FIFO not empty
    uint16_t als_raw = SSI2->DR;

    // Deassert CS
    GPIOB->DATA |= (1 << 5);

    // Extract 8-bit light value (bits 5-12)
    uint8_t light_level = (als_raw >> 5) & 0xFF;

    return light_level;
}

#endif // USE_PMOD_ALS
```

GPIO.h

```
/**
* @file GPIO.h
*
* @brief Header file for the GPIO driver.
*
* This file contains the function definitions for the GPIO driver.
* It interfaces with the following:
*   - User LED (RGB) Tiva C Series TM4C123G LaunchPad
*   - EduBase Board LEDs (LED0 - LED3)
*   - EduBase Board Push Buttons (SW2 - SW5)
*
* To verify the pinout of the user LED, refer to the Tiva C Series TM4C123G
LaunchPad User's Guide
* Link: https://www.ti.com/lit/pdf/spmu296
*
* @author Aaron Nanas, Edited by Macy Varga
*/


#include "TM4C123GH6PM.h"
#include "SysTick_Delay.h"


// Constant definitions for the user LED (RGB) colors
extern const uint8_t RGB_LED_OFF;
extern const uint8_t RGB_LED_RED;
extern const uint8_t RGB_LED_BLUE;
extern const uint8_t RGB_LED_GREEN;


// Constant definitions for the EduBase board LEDs
```

```
extern const uint8_t EDUBASE_LED_ALL_OFF;
extern const uint8_t EDUBASE_LED_ALL_ON;


/**
* @brief The RGB_LED_Init function initializes the RGB LED (PF1 - PF3)
*
* This function initializes the following RGB LED pins, configures the
digital functionality for the pins,
* and sets the direction of the pins as output. The RGB LED is off by
default upon initialization.
*   - LED_R     (PF1)
*   - LED_B     (PF2)
*   - LED_G     (PF3)
*
* @param None
*
* @return None
*/
void RGB_LED_Init(void);


/**
* @brief The RGB_LED_Output function sets the output of the RGB LED.
*
* This function sets the output of the RGB LED based on the value of the
input, led_value.
* A bitwise AND operation (& 0xF1) is performed to mask the Bits 1 to 3 of
the GPIOF's DATA register
* to preserve the state of other pins connected to Port F while keeping the
RGB LED pins unaffected.
* Then, a bitwise OR operation is performed with led_value to set the RGB
LED pins to the desired state
* specified by led_value.
*
* @param led_value An 8-bit unsigned integer that determines the output of
the RGB LED. To turn off
*                  the RGB LED, set led_value to 0. The following values
determine the color of the RGB LED:
*
*  Color       LED(s)   led_value
*  Off          ---         0x00
*  Red          R--         0x02
*  Blue         -B-         0x04
*  Green        --G         0x08
*
* @return None
*/
void RGB_LED_Output(uint8_t led_value);


/**
* @brief The RGB_LED_Status function indicates the status of the RGB LED
* located at pins PF1, PF2, and PF3.
*
* @param None
*
* @return uint8_t The value representing the status of the RGB LED.
*
*  Color        LED(s)   led_value
*  Off          ---         0x00
*  Red          R--         0x02
*  Blue         -B-         0x04
*  Green        --G         0x08
*
*/
uint8_t RGB_LED_Status(void);


/**
* @brief The EduBase_LEDs_Init function initializes the EduBase Board LEDs
(LED0 - LED3)
*
* This function initializes the following EduBase Board LEDs, configures
the digital functionality for the pins,
* and sets the direction of the pins as output. The EduBase Board LEDs are
off by default upon initialization.
*   - LED0      (PB0)
*   - LED1      (PB1)
*   - LED2      (PB2)
*   - LED3      (PB3)
*
```

```
* @param None
*
* @return None
*/
void EduBase_LEDs_Init(void);


/**
* @brief The EduBase_LEDs_Output function sets the output of the EduBase
Board LEDs.
*
* This function sets the output of the EduBase Board LEDs based on the
value of the input, led_value.
* A bitwise AND operation (& 0xF0) is performed to mask the lower four bits
(Bits 0 to 3) of the GPIOF's DATA register
* to preserve the state of other pins connected to Port B while keeping the
LED pins unaffected.
* Then, a bitwise OR operation is performed with led_value to set the LED
pins to the desired state
* specified by led_value.
*
* @param led_value An 8-bit unsigned integer that determines the output of
the EduBase Board LEDs.
*
* @return None
*/
void EduBase_LEDs_Output(uint8_t led_value);


/**
* @brief The EduBase_Button_Init function initializes the EduBase Board
buttons (SW2 - SW5).
*
* This function initializes the EduBase Board buttons connected to pins
PD0, PD1, PD2, and PD3.
* It enables digital functionality and configures the pins as GPIO input
pins.
*
* @param None
*
* @return None
```

```c
*/
void EduBase_Button_Init(void);


/**
* @brief The Get_EduBase_Button_Status reads the status of the EduBase
Board buttons (SW2 - SW5) and returns it.
*
* This function reads the status of the EduBase Board buttons connected to
pins PD0, PD1, PD2, and PD3.
* It indicates whether or not the buttons are pressed and returns the
status.
* A bitwise AND operation (& 0xF) is performed to mask the unused bits in
the data register.
*
* @param None
*
* @return Indicates the status of the buttons.
*
* For example:
*   - 0x00: No buttons are pressed
*   - 0x01: SW5 is pressed
*   - 0x02: SW4 is pressed
*   - 0x04: SW3 is pressed
*   - 0x08: SW2 is pressed
*/
uint8_t Get_EduBase_Button_Status(void);
```

SysTick_Delay.h

```c
/**
 * @file SysTick_Delay.h
 *
 * @brief Header file for the SysTick_Delay driver.
 *
 * It provides two blocking functions, SysTick_Delay1ms and
SysTick_Delay1us,
 * to create a delay with a busy-wait loop. It uses the SysTick timer with
 * a specified reload value to generate interrupts every 1 us.
 *
 * In addition, it uses the Peripheral Internal Oscillator (PIOSC)
 * as the clock source. The PIOSC provides 16 MHz which is then divided by
4.
 * The timer is used for creating delays in either microseconds or
milliseconds.
 *
 * @author Aaron Nanas, edited by Macy Varga
 */

#include "TM4C123GH6PM.h"

/**
 * @brief The SysTick_Delay_Init function initializes the SysTick timer to
be used for a blocking delay function.
 *
 * This function configures the SysTick timer and its interrupt with a
specified reload value to
 * generate interrupts every 1 us. It uses the Peripheral Internal
Oscillator (PIOSC) as the clock source.
 * The PIOSC provides 16 MHz which is then divided by 4. The timer is used
for creating delays in either
 * microseconds or milliseconds.
 *
 * @param None
 *
 * @return None
 */
void SysTick_Delay_Init(void);

/**
 * @brief The SysTick_Delay1us function provides a blocking delay in
microseconds using the SysTick timer.
 *
 * This function resets the global variable, us_elapsed, to zero and waits
until us_elapsed reaches
 * the specified delay_in_us.
 *
 * @param delay_in_us The delay time in microseconds.
 *
 * @return None
 */
```

```
void SysTick_Delay1us(uint32_t delay_in_us);

/**
 * @brief The SysTick_Delay1ms function provides a blocking delay in
milliseconds using the SysTick timer.
 *
 * This function clears the global variables, us_elapsed and ms_elapsed, to
zero and sets ms_active flag to 0x01
 * indicating that milliseconds delay is active. It then waits until
ms_elapsed reaches the specified delay_in_ms.
 * After the delay, it clears ms_active flag back to 0x00.
 *
 * @param delay_in_ms The delay time in milliseconds.
 *
 * @return None
 */
void SysTick_Delay1ms(uint32_t delay_in_ms);

/**
 * @brief The SysTick_Handler function is the interrupt service routine for
the SysTick timer.
 *
 * This function is called whenever the SysTick timer generates an
interrupt. It increments the global variable
 * us_elapsed by 1, indicating that 1 microsecond has passed. Additionally,
if us_elapsed reaches 1000 and ms_active
 * flag is set to 0x01, it resets us_elapsed and increments ms_elapsed by
1, indicating that 1 millisecond has passed.
 *
 * @param None
 *
 * @return None
 */
void SysTick_Handler(void);
```

PWM1_3.h

```
/**
* @file PWM1_3.h
*
```

```
* @brief Header file for the PWM1_3 driver.
*
* This file contains function declarations for controlling PWM Generator 3
* on PWM Module 1 for use with a servo motor connected to PF2 (M1PWM6).
* It provides initialization and dynamic duty cycle adjustment functions.
*
* @author Macy Varga
*/


#ifndef _PWM1_3_H_
#define _PWM1_3_H_


#include <stdint.h>


/**
* @brief Initializes PWM Generator 3 on Module 1.
*
* Sets up PF2 for PWM output, configures the PWM clock and generator
behavior.
*
* @param period_constant The total period value for the PWM signal.
* @param duty_cycle The initial duty cycle value for the PWM signal.
*
* @return None
*/
void PWM1_3_Init(uint16_t period_constant, uint16_t duty_cycle);


/**
* @brief Updates the duty cycle of the PWM signal.
*
* @param duty_cycle New duty cycle value to be loaded into the generator.
*
* @return None
*/
void PWM1_3_SetDuty(uint16_t duty_cycle);
```

```
#endif // _PWM1_3_H_
```

PMOD_BTN_Interrupt.h

```
/**
* @file PMOD_BTN_Interrupt.h
*
* @brief Header file for the PMOD_BTN_Interrupt driver.
*
* This file contains the function declarations for the PMOD_BTN_Interrupt
driver.
* It interfaces with the PMOD BTN module using GPIO Port A pins (PA2 -
PA5),
* and sets up interrupts on rising edges to detect button presses.
*
* @author Macy Varga
*/


#ifndef _PMOD_BTN_INTERRUPT_H_
#define _PMOD_BTN_INTERRUPT_H_


#include "TM4C123GH6PM.h"


/**
* @brief Pointer to the user-defined task function called on PMOD BTN
interrupt.
*
* @param pmod_btn_state 8-bit unsigned integer representing button status.
*/
extern void (*PMOD_BTN_Task)(uint8_t pmod_btn_state);


/**
* @brief Initializes the PMOD BTN interrupts on Port A.
*
* Configures PA2-PA5 as GPIO inputs, enables pull-down resistors,
* and sets up interrupts on rising edges. Sets interrupt priority and
```

```
enables IRQ.
*
* @param task Function pointer to the user-defined handler function.
*
* @return None
*/
void PMOD_BTN_Interrupt_Init(void(*task)(uint8_t));


/**
* @brief Reads the current button status of the PMOD BTN module.
*
* @param None
*
* @return uint8_t Current status of buttons (active-high).
*/
uint8_t PMOD_BTN_Read(void);


/**
* @brief Interrupt Service Routine for Port A (PMOD BTN interrupt
handling).
*
* Checks which button triggered the interrupt, calls the user task, and
clears the interrupt flag.
*
* @param None
*
* @return None
*/
void GPIOA_Handler(void);


#endif // _PMOD_BTN_INTERRUPT_H_
```

SSI2.h

```
/**
* @file SSI2.h
*
```

```
* @brief Header file for the SSI2 driver (used with PMOD ALS).
*
* This file provides function declarations for initializing SSI2 peripheral
* and reading light intensity values from the PMOD ALS sensor module via
SPI communication.
*
* @author Macy Varga
*/


#ifndef _SSI2_H_
#define _SSI2_H_


#include "TM4C123GH6PM.h"
#include <stdint.h>


/**
* @brief Initializes the SSI2 module.
*
* Configures GPIO pins PB4 (SCLK), PB5 (CS), PB6 (MISO) for SSI2 operation,
* and sets up SSI2 for 16-bit SPI Mode 0 communication at 1 MHz.
*
* @param None
*
* @return None
*/
void SSI2_Init(void);


/**
* @brief Reads the light intensity value from the PMOD ALS sensor.
*
* @param None
*
* @return uint8_t Light intensity value (8 bits).
*/
uint8_t PMOD_ALS_Read(void);
```

```
#endif // _SSI2_H_
```