RTES ECEN-5623

Project Report

Occupant Safety (Airbag control Unit and Seat-Belt Adjustment Unit)

Submitted by

Jithendra H S and Suhas Reddy S

on

05/05/2024

1 Introduction

Occupant Safety (Airbag Deployment and Seat Belt Adjustment) aims to prototype two of the main automotive safety features i.e. Airbag Control Unit (ACU) and Seat-Belt Adjustment Unit (SAU). ACU manages the mechanism to inflate the Airbag and the SAU helps hold the occupant in place. A crash is detected via a push button interrupt and releases the semaphore for ACU and SAU. Additionally, a speed and temperature monitoring system is used to assist the SAU.

1.1 System overview:

Our system architecture is designed with two primary modules: the Sensor side and the Actuator side, each serving distinct functions crucial for the overall functionality of the system.

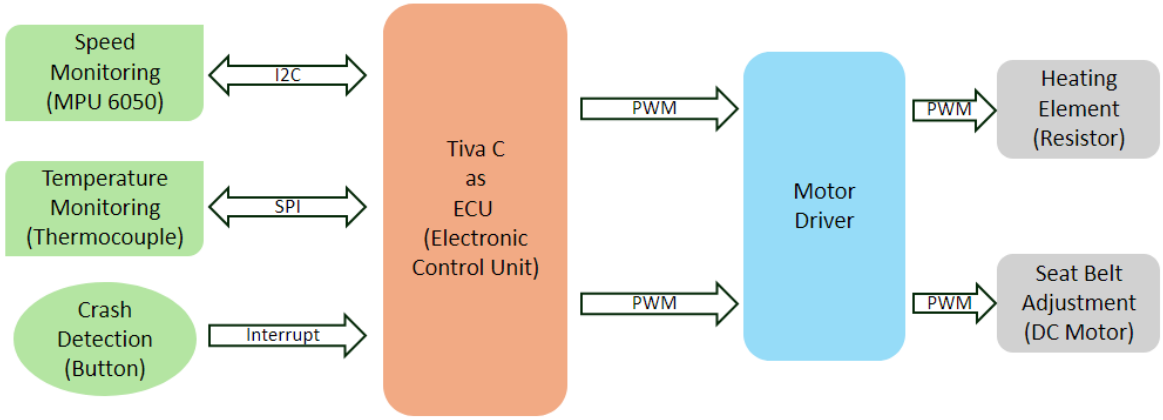


Fig 1.1 Block diagram of Occupant safety

On the Sensor side, we have three key modules:

Speed Monitoring: This module is responsible for continuously monitoring the speed of the vehicle. It utilizes sensors or devices capable of accurately measuring vehicle speed and provides real-time data to the system for analysis and decision-making.

Temperature Monitoring: This module is dedicated to monitoring the temperature of the heating element, which is crucial for post-crash operations. It ensures that the filament reaches and maintains the desired temperature range for optimal performance in seat belt pretension.

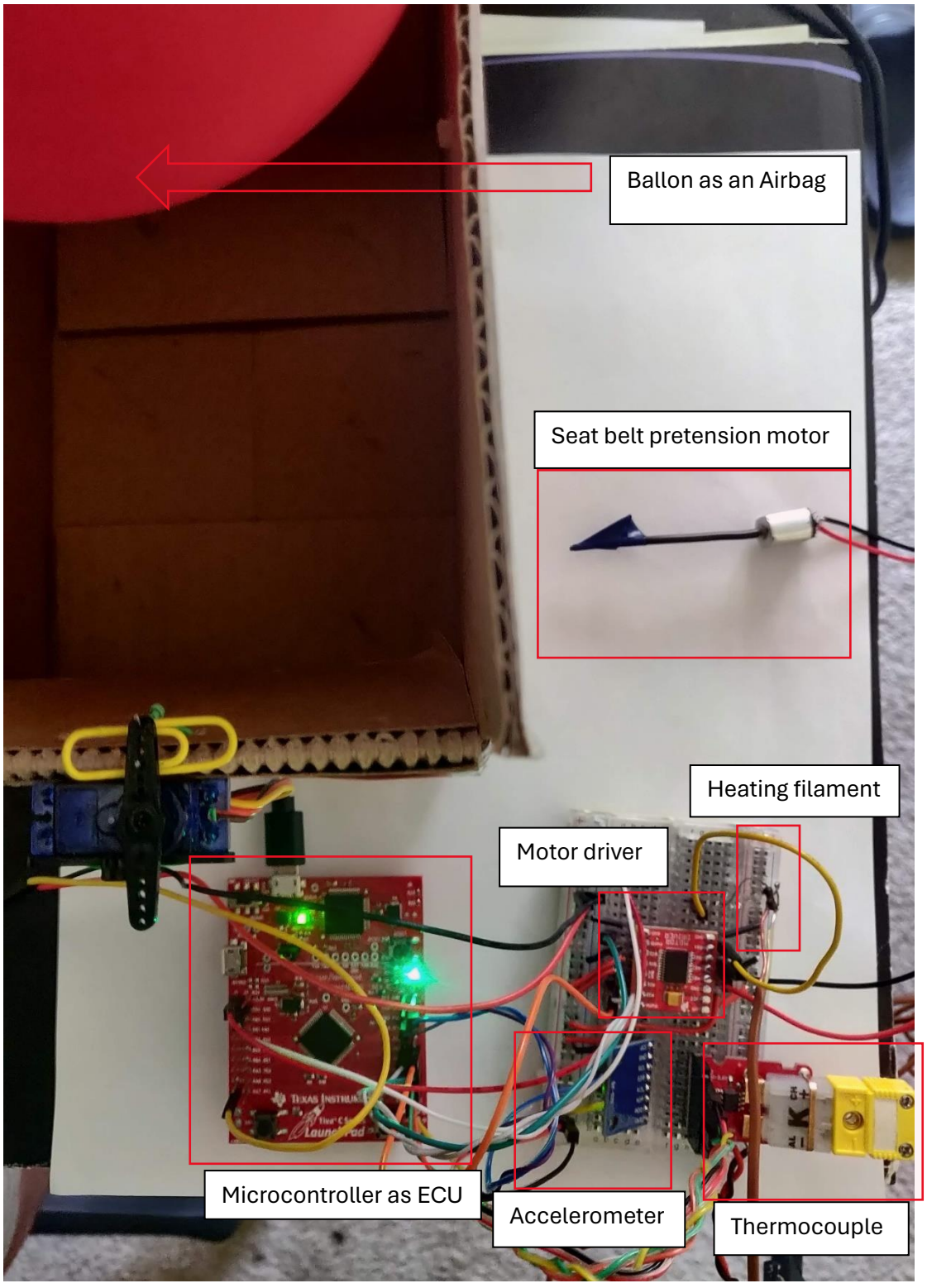
Crash Detection: This module detects crash events using sensors or switches strategically placed within the vehicle. Upon detecting a crash, it triggers the activation of the appropriate safety mechanisms, such as the airbag deployment and seat belt pretensioners.

On the Actuator side, we have two primary modules:

Heating Element: This module controls the heating element, which is essential for seat belt pretension post-crash. It regulates the temperature of the filament based on inputs received from the temperature monitoring module, ensuring optimal performance during operation.

Seat Belt Pretensioners: This module is responsible for deploying the seat belt pretensioners in response to crash events. It receives signals from the crash detection module and coordinates with the heating element module to ensure timely and effective pretension of the seat belts, thereby enhancing occupant safety.

2 Hardware Setup



The above picture depicts the hardware arrangement of components as described in the block diagram. The microcontroller is interfaced with the accelerometer, thermocouple, seat belt pretension motor, and heating filament (resistor) with the help of a motor driver.

3 Functional Requirements

Before the Crash:

Speed Monitoring: The system must monitor and compute the vehicle speed using acceleration data obtained from the MPU6050 accelerometer.

Temperature Monitoring: The system must pre-heat the heater resistor and maintain a specified temperature threshold by continuously monitoring the temperature using the temperature sensor.

Crash Detection: The system must be capable of simulating a crash event via a GPIO interrupt generated by a push button, triggering the activation of safety mechanisms.

After the Crash:

Seat-Belt Control Unit: The system must adjust the PWM load value of the seat belt motor to tighten the seat belt post-crash, enhancing occupant safety.

Airbag Deployment: The system must increase the temperature of the heater resistor to facilitate airbag deployment post-crash. It should monitor and maintain the temperature within a specified range for a duration of 20ms following crash detection, ensuring timely and effective deployment of the airbag system.

4 Real time requirements

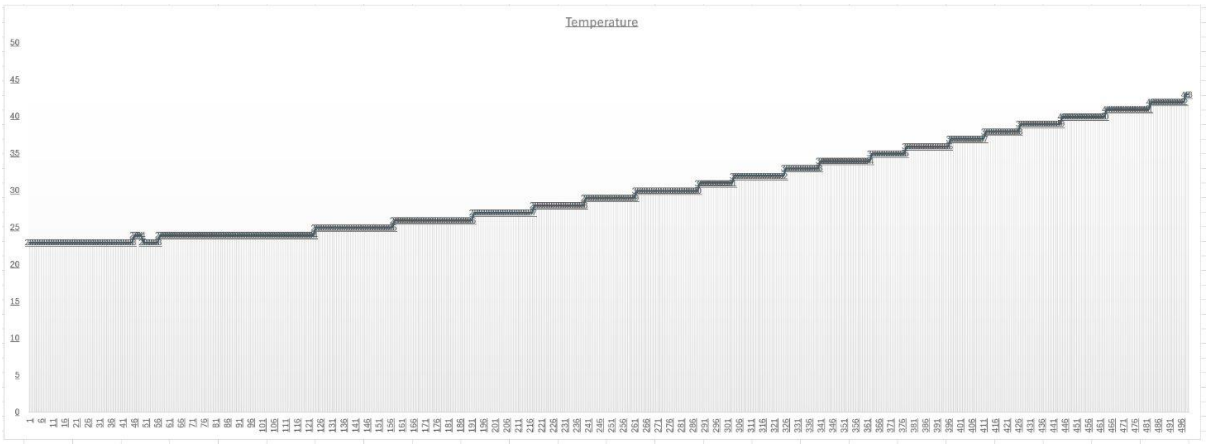
Service 1 Speed Monitoring:

https://www.maxxecu.com/webhelp/settings-configuration-ecu_logging_settings.html

The above link from an ECU-Electronic Control Unit manufacturer states that the data logging can be configured to a rate of 1KHz. We have chosen this to have the latest speed data when a crash happens.

Service 2 Speed Monitoring:

This service is required in our case as the response time for the heater resistor is too large, as seen in the image here. For this reason, we need to pre-heat the resistor to meet the deadline after a crash happens and we have also scaled the temperature values to simulate the same.



Service 3 Seat-Belt Adjustment:

(PDF) ANALYSIS OF PROPERTIES OF OPERATION OF THE SUPPORTING EQUIPMENT FOR THE SEAT BELTS (researchgate.net)

impacts of the seat belts on a man in a car. After implementing the pretensioners and force limiters some effects improving the effectiveness of the seat belts, and reducing possibility of human injury, were obtained:

– getting the belt tension effect in a very short time, after about 15 – 20 ms, with the force of 100 – 300 daN;

Service 4 Airbag Deployment:

Study regarding the influence of airbag deployment time on the occupant injury level during a frontal vehicle collision (matec-conferences.org)

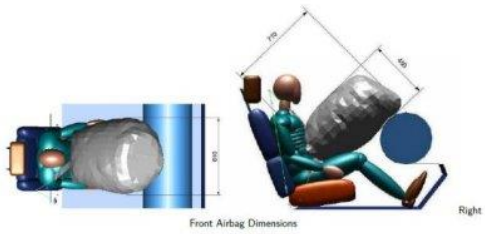


Fig. 3. Passenger airbag technical specifications [16].

The inflation dimensions of the passenger airbag are 450 mm in length, 610 mm in width and 710 mm in height. During inflation, the massflow goes from 0 to 3.3 kg/s in the first 20 ms, then it drops to almost 0 in the next 80 ms. The mass of the airbag expands from 0 to 100 g in 100 ms. It can be observed that the passenger airbag is 2 time

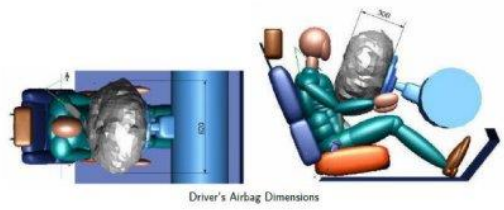


Fig. 2. Driver airbag technical specifications [16].

The inflation dimensions of the airbag are 300 mm in length and 620 mm in width. The total duration of inflation is 25 ms, during this time, the massflow reaches 1.8 kg/s in the first 10 ms, then it drops to 0 in the next 40 ms. The mass of the airbag expands from 0 to 30 g in 50 ms.

Before the Crash:

Speed Monitoring: Speed must be monitored every 1ms, with a deadline of 508.21us.

Temperature Monitoring: Temperature must be monitored every 2ms, with a deadline of 1.917ms.

Crash Detection: Crash detection must occur within 3.04us upon activation of the push button interrupt.

After the Crash:

Seat-Belt Control Unit: The system must adjust the PWM load value for the motor, which is computed based on the speed, within 8.94ms.

Airbag Deployment Unit: The system must increase the temperature of the heater resistor and monitor it for 20ms since crash detection, ensuring timely and effective deployment of the airbag system.

5 Functional design

5.1 State diagram:

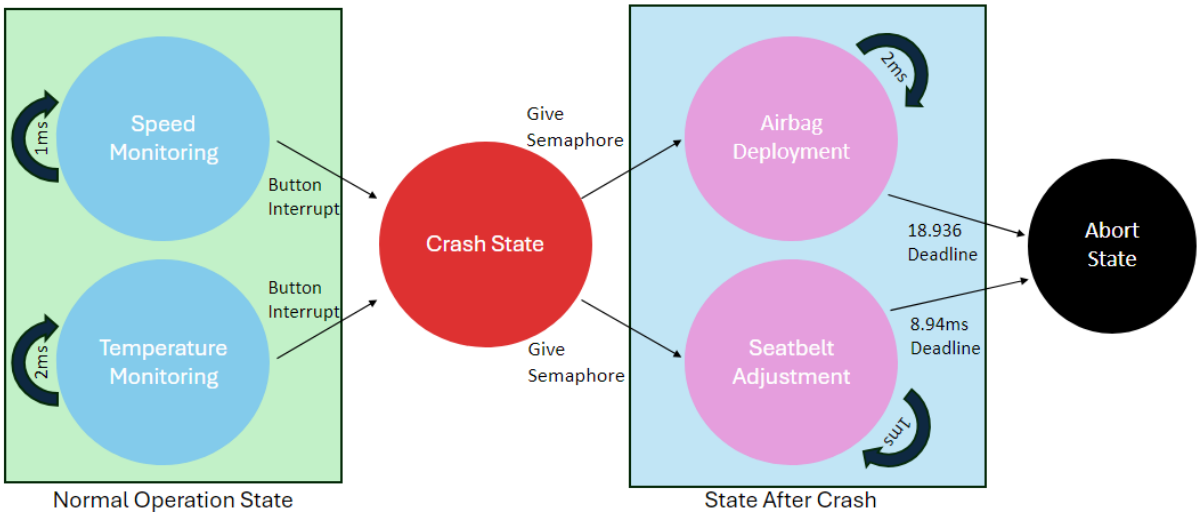


Fig 5.1 State diagram

In the normal operation state, the system schedules tasks S1 and S2 to monitor speed and temperature, respectively, at regular 1ms and 2ms intervals. Upon pressing the button to trigger a crash event, the system releases semaphores for tasks S3 and S4, responsible for seat-belt control and airbag deployment, respectively. Task S3 adjusts the PWM signal for the motor based on vehicle speed noted during normal operation. Task S4 increases the temperature of the heating element to facilitate airbag inflation. Both tasks S3 and S4 continue their operations until 20ms have elapsed since the crash event. After 20ms, both tasks are aborted, and the system returns to the idle state, ready for further operation.

5.2 Sequence (UML):

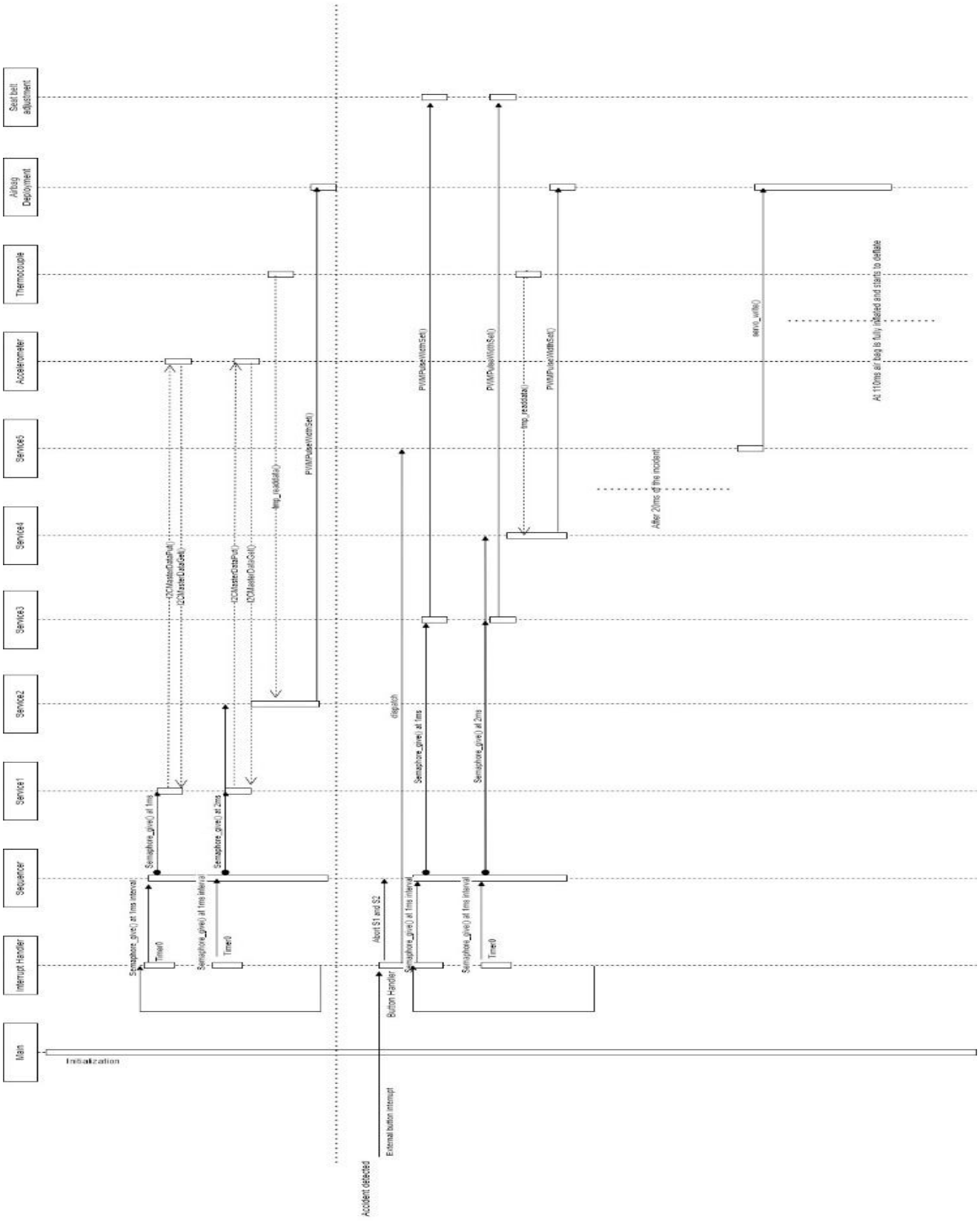


Fig 5.2 UML Sequence diagram

Here's a breakdown of the sequence flow based on the provided steps:

1. Initialization of Modules:
   Modules like Timer0, I2C, SPI, and PWM are initialized to interface with sensors and actuators.
2. Timer0 Interrupt Handling:
   Timer0 is configured to generate interruptions every 1ms.
   Upon each interruption, a semaphore is released to trigger the sequencer task.
3. Sequencer Task:
   The sequencer task schedules S1 for every 1ms and S2 for every 2ms.
4. Service 1 (S1):
   S1 reads acceleration using I2C communication by requesting data from a specific I2C address and register.
5. Service 2 (S2):
   S2 reads temperature data from a thermocouple sensor using temp_readdata() function.
   It adjusts the PWM signal to maintain the temperature at 250°C.
6. Crash Detection:
   Once a crash is detected, Timer0 interrupts resume.
7. Timer0 Interrupt Handling (After Crash):
   Timer0 is configured to generate interruptions every 1ms again.
   A semaphore is released upon each interruption to trigger the sequencer task.
8. Sequencer Task (After Crash):
   The sequencer task schedules S3 for every 1ms and S4 for every 2ms.
9. Service 3 (S3):
   S3 updates PWM signals based on the speed noted before the crash for a certain interval of time.
10. Service 4 (S4):
    S4 reads temperature data from a thermocouple sensor using temp_readdata() function.
    It adjusts the PWM signal to maintain the temperature above 270°C.
11. Airbag Deployment:
    Once 20ms is reached after the crash, S3 and S4 are aborted, allowing S5 to run.
12. Service 5 (S5):
    S5 inflates the airbag and resets the device.

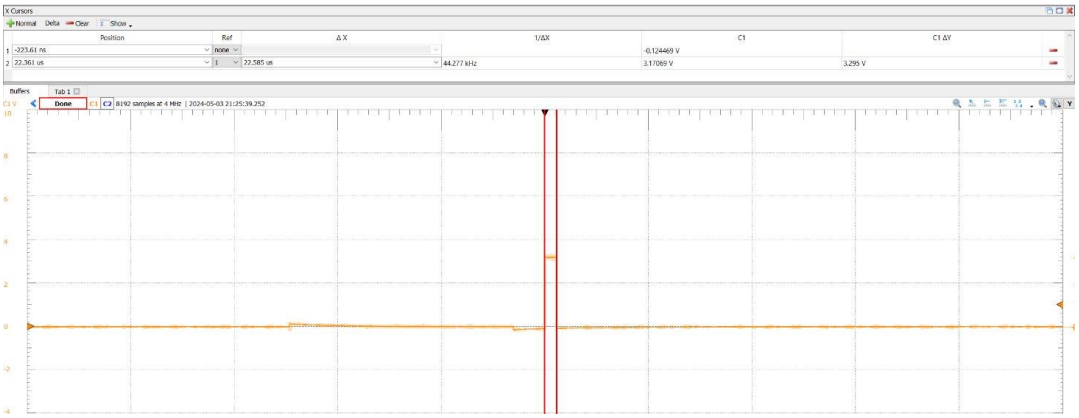
This sequence flow illustrates the sequence of operations performed by different tasks and services in response to events such as timer interruptions and crash detection.

6 Real time Analysis

Time profiling of accelerometer input latency:

Noted around 431.35us required to read data from the accelerometer due to I2C communication.

Time profiling of Temperature sensor input latency:

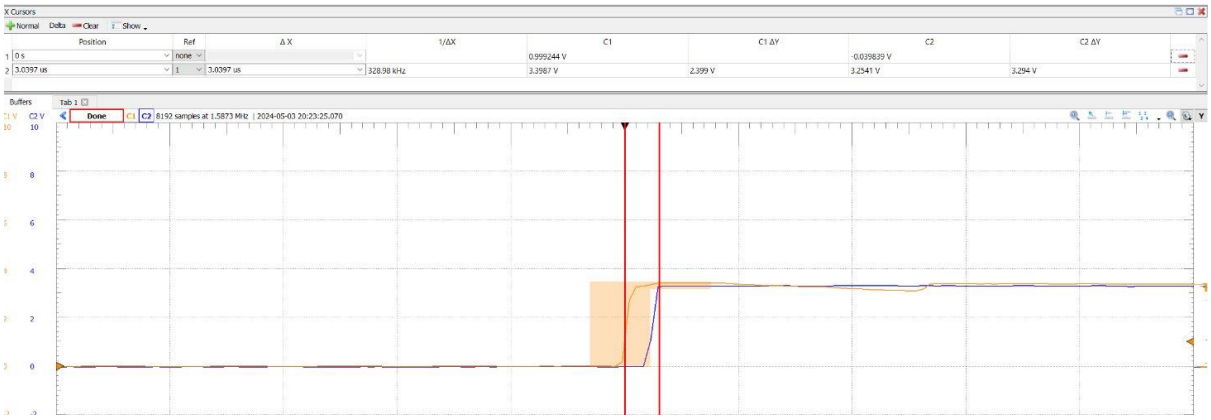


Noted around 22.585us required to read data from temperature Senor due to SPI communication.

Context Switch Latency:

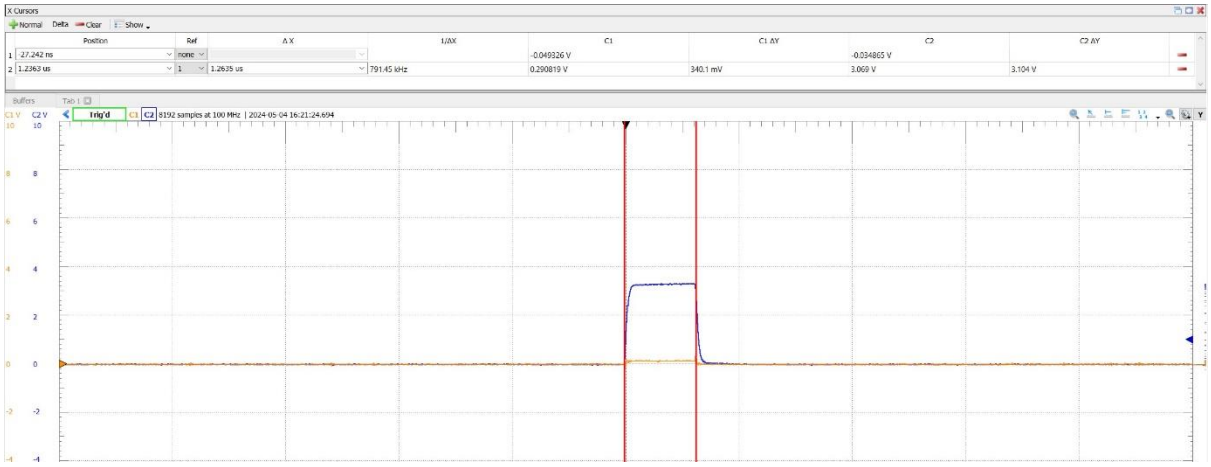
Noted around 60.44us required to do context switch from one task to another.

Button Interrupt Latency:

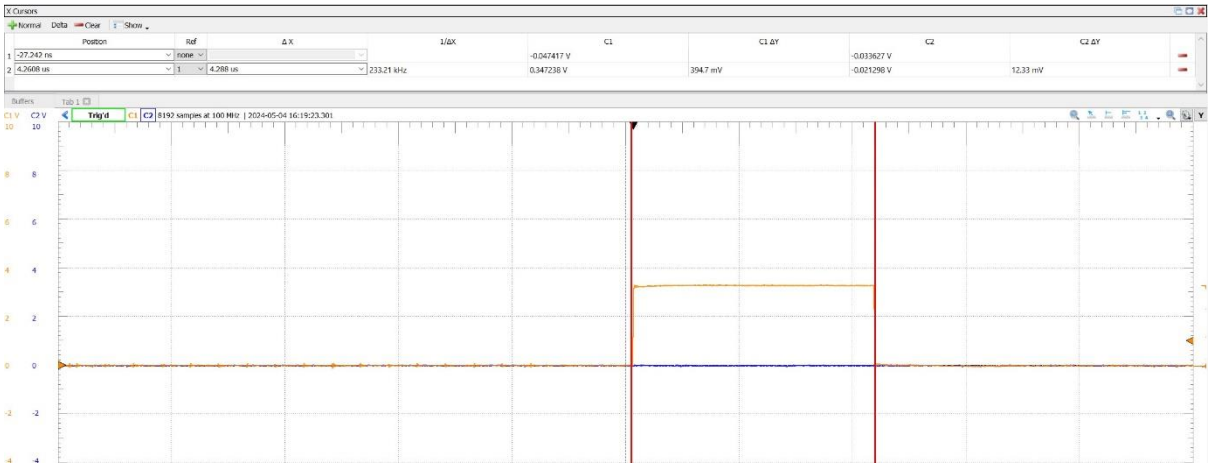


Noted around 3us required to enter into interrupt handler after push button is pressed.

S1 execution time:



S2 execution time:



Computation time for service set after crash (S3, S4):

S3 performs seat belt adjustment for a certain iteration which depends on the speed value before the crash, this entire execution becomes the load for S3. In the worst case scenario where a deadline is not missed the iteration count comes up to 8, and during each iteration executes a fixed number of instructions which runs for 8.576 us. From this we get a WCET of 68.608 us for S3.

Similarly, S2 adjusts the PWM value of the heating element and monitors its temperature for up to 20 ms. Temperature is monitored for about every 2 ms which takes up 87.308 us including input latency and context switch. This can happen for utmost 10 times in 2ms giving WCET as 873.08 us.

Using all the data noted, derived response time for S1, S2, S3 and S4:

Response Timeline for Service Set Before Crash:

Output latencies for these services will be 0 as it won't involve actuating.

Speed Monitoring Service Response Timeline:

The above timeline shows input latency while reading 2 bytes of data from the accelerometer via 12c at 400KHz, context switch latency and execution time to compute speed.

Temperature Monitoring Service Response Timeline:

The above timeline shows input latency while reading 1 byte of data from the accelerometer via SPI at 1MHz, context switch latency and execution time to adjust PWM load.

Response Timeline for Service Set after Crash:

Seat-Belt Adjustment Service

The above timeline shows input latency of the button interrupt, context switch latency, execution time to adjust the pretensioner motor and output latency of the actuator(motor).

Airbag Adjustment Service

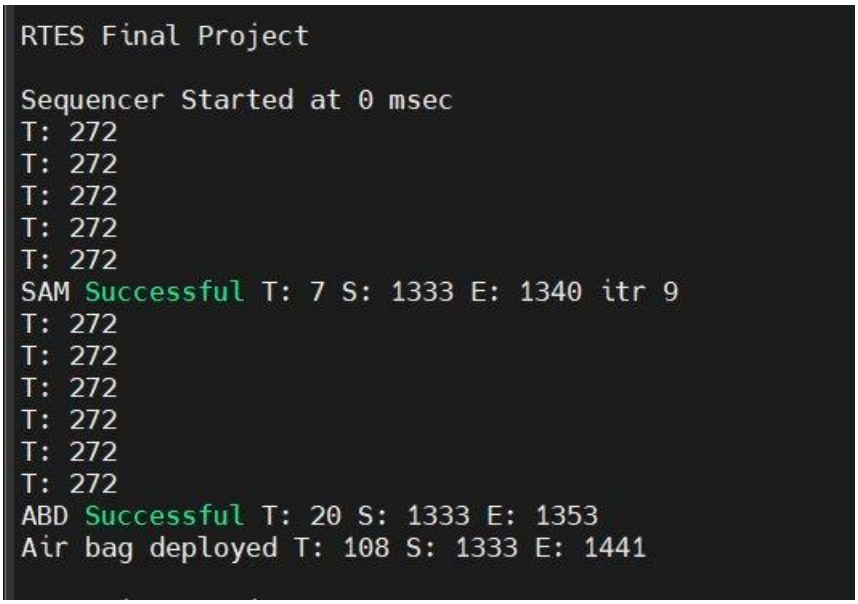
The above timeline shows input latency of the button interrupt, context switch latency, execution time required to maintain the temperature of the heater resistor and output latency of the resistor.

7 Proof of concept

Selection of Hardware and Software: We chose the Tiva C series microcontroller as the core hardware platform and implemented FreeRTOS as the scheduler to manage task scheduling and execution efficiently.

Integration of Sensors and Actuators: We successfully integrated the necessary sensors, including the accelerometer and temperature sensor, to acquire essential data regarding vehicle speed and post-crash temperature. Additionally, we implemented control mechanisms for the actuators, such as the seat belt pretensioner and heating element, to ensure precise and timely response in critical situations.

Implementation of Crash Detection: To simulate crash detection, we utilized push button interrupts as a trigger mechanism. This allowed us to simulate a crash event and initiate the corresponding safety measures, such as airbag deployment and seat belt pretension, in response to the detected event.



Here, we observe that the scheduler's start time is recorded at the beginning of the program execution. To streamline the logging process and avoid unnecessary data printing, we have selectively included logs only after a crash event occurs. During normal operation, sensor readings are not printed continuously. Instead, logs are generated specifically when the temperature is read every 2ms after a crash event.

Upon detecting a crash, we observe the successful signaling of the seat belt pretensioner after 7ms relative to the crash's start time, noted at 1333ms. Throughout this period, the temperature is maintained at approximately 272°C, meeting the required threshold for airbag deployment.

Continuing the monitoring process for the specified 20ms duration, we confirm the successful completion of the temperature maintenance, indicated by corresponding logs. Finally, the completion message for airbag actuation is received at 108ms post-crash, marking the successful execution of the safety protocol.

8 Verification process

The verification process for our Occupant Safety Project involved several meticulous steps to ensure the functionality, reliability, and compatibility of our prototype. Below is an outline of the verification methods employed:

Prototype Verification with Arduino UNO: Initially, our project prototype was rigorously verified using Arduino UNO. Arduino UNO was chosen for its extensive library support, abundant examples, and user-friendly programming interface. This platform allowed for seamless verification and understanding of sensor functionalities, streamlining testing and troubleshooting processes.

Modular Integration: We adopted a modular integration approach, where each project component was individually tested and verified before integration. This method ensured that each component functioned correctly before being integrated into the overall system. Modular integration facilitated efficient debugging and problem isolation, minimizing potential system-wide issues.

Utilization of UART Logs and LED Indications: During the verification process, UART logs and LED indications served as crucial diagnostic tools. UART logs provided real-time feedback and data logging, enabling detailed analysis of system behavior. LED indicators were employed to provide visual cues for system status, aiding in the identification of any anomalies or errors during testing.

Profiling with Oscilloscope and TIVA GPIO Pins: System profiling was conducted using an oscilloscope and TIVA GPIO pins. This involved monitoring electrical signals and waveforms generated by the system components to assess performance and timing. Analysis with the oscilloscope and GPIO pins allowed for the identification and resolution of any deviations or inconsistencies in system operation.

Demo Setup: A meticulously arranged demo setup was prepared to showcase the functionality and performance of our project. The setup simulated real-world scenarios, ensuring all connections and peripherals were appropriately configured.

9 Conclusion and Future scope

In conclusion, this project has provided us with a valuable opportunity to develop a prototype aimed at addressing a significant real-world problem concerning occupant safety. Throughout the project, we

have gained insights into profiling services, input latency, and execution time, underscoring the critical importance of sensor and actuator response times in ensuring the efficacy of safety mechanisms.

Looking ahead, the future scope of this project involves exploring avenues to integrate more responsive sensors and actuators, thereby enhancing the system's overall performance. Additionally, we aim to refine our understanding by altering the FreeRTOS scheduler clock to achieve more precise timing, enabling finer control over safety feature behavior. By leveraging additional data and insights, we endeavor to further improve the system's response and precision, thereby advancing the cause of occupant safety in automotive applications.

10 References

1. https://www.ti.com/tool/EK-TM4C123GXL
2. Tiva™ C Series TM4C123GH6PM Microcontroller Data Sheet datasheet (Rev. E)
3. https://www.analog.com/media/en/technical-documentation/data-sheets/MAX31855.pdf
4. https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf
5. https://www.sparkfun.com/datasheets/Robotics/TB6612FNG.pdf
6. https://e2e.ti.com/support/microcontrollers/arm-based-microcontrollers-group/arm-based-microcontrollers/f/arm-based-microcontrollers-forum/680730/ccs-ek-tm4c123gxl-problems-interfacing-tiva-to-mpu6050-accelerometer-gyroscope
7. https://github.com/smalik007/Servo-controlled-by-Tiva-C-Series-ARM-Cortex-M3-
8. https://e2e.ti.com/support/microcontrollers/arm-based-microcontrollers-group/arm-based-microcontrollers/f/arm-based-microcontrollers-forum/559552/tiva-tm4c123-side-of-spi-interface
9. Smart Automotive Airbags: Occupant Classification and Tracking 0018-9545/$25.00 © 2007 IEEE
10. ANALYSIS OF PROPERTIES OF OPERATION OF THE SUPPORTING EQUIPMENT FOR THE SEAT BELTS
11. Study regarding the influence of airbag deployment time on the occupant injury level during a frontal vehicle collision

11 Appendix

11.1 Code

```
/*****************************************************************************
**
 * Copyright (C) 2023 by Jithendra and Suhas
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra, Suhas and the University of Colorado are not liable
for
 * any misuse of this material.
 *
*****************************************************************************/
/**
 * @file main.c
 * @brief Occupant safety
 * @author Jithendra and Suhas
 * @date 2024-4-29
 */

// Include necessary header files
#define TARGET_IS_TM4C123_RA1
#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_ints.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "driverlib/timer.h"
#include "driverlib/pwm.h"
#include "utils/uartstdio.h"
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
#include "accelerometer.h"
#include "temperature_sensor.h"
#include "button_interrupt.h"
#include "pwm_control.h"

// Define constants and variables
```

```c
#define ACC_SCALING (2048)
#define SCALE (3)
#define SAM_DEADLINE (10)
#define ABD_DEADLINE (20)

#define SEAT_TIGHT_ITER (3)
#define SPEED_OFFSET (4)
#define ACC_OFFSET (5);

// Function prototypes
static void service1(void *params);
static void service2(void *params);
static void service3(void *params);
static void service4(void *params);
static void service5(void *params);
static void Sequencer_thread(void *params);

// Semaphore handles
xSemaphoreHandle semSched, semS1, semS2, semS3, semS4, semS5;

// Abort flags
volatile bool abortS1 = false, abortS2 = false, abortS3 = false, abortS4 =
false;

// Timing parameters
volatile uint32_t T1 = 1 * SCALE;
volatile uint32_t T2 = 2 * SCALE;
volatile uint32_t T3 = 20 * SCALE;
volatile uint32_t T4 = 20 * SCALE;

volatile uint32_t event_start = 0;

volatile uint32_t speed = 0;

#ifdef DEBUG
void
error(char *pcFilename, uint32_t ui32Line)
{
}

#endif

// Stack overflow hook
void vApplicationStackOverflowHook(xTaskHandle *pxTask, char *pcTaskName)
{
    // Handle stack overflow
    while (1)
    {
```

```
    }
}


/**
 * @func    ConfigureUART
 * @brief   Configures UART0 for communication
 * @param   None
 * @return  None
 * @reference   TM4C123GH6PM Example
 */
void ConfigureUART(void)
{
    // Enable the GPIO peripheral for Port A
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    // Enable the UART peripheral
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

    // Configure PA0 as the UART RX pin
    ROM_GPIOPinConfigure(GPIO_PA0_U0RX);

    // Configure PA1 as the UART TX pin
    ROM_GPIOPinConfigure(GPIO_PA1_U0TX);

    // Set PA0 and PA1 as UART pins
    ROM_GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    // Set the clock source for UART0 to the precision internal oscillator
(PIOSC)
    UARTClockSourceSet(UART0_BASE, UART_CLOCK_PIOSC);

    // Configure UART0 with standard I/O settings
    // Parameters: 0 for the UART instance, baud rate of 230400, and system
clock frequency of 16,000,000
    UARTStdioConfig(0, 230400, 16000000);
}


/**
 * @func    timer0_init
 * @brief   Initializes Timer0 for periodic operation
 * @param   None
 * @return  None
 * @reference   TM4C123GH6PM Example
 */
void timer0_init(){
    // Enable Timer0 peripheral
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
```

```
    // Configure Timer0A to run in periodic mode at 1000Hz
    ROM_TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);
    ROM_TimerLoadSet(TIMER0_BASE, TIMER_A, ROM_SysCtlClockGet() / 1000);

    // Enable Timer0A interrupt
    ROM_IntEnable(INT_TIMER0A);
    ROM_TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    // Enable Timer0A
    ROM_TimerEnable(TIMER0_BASE, TIMER_A);
}


/**
 * @func    Timer0IntHandler
 * @brief   Interrupt handler for Timer0
 * @param   None
 * @return  None
 * @reference   TM4C123GH6PM Datasheet - Timer0 Section
 */
void Timer0IntHandler(void)
{
    // Clear the interrupt flag for Timer A timeout
    ROM_TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    // Release the semaphore to signal the sequencer
    xSemaphoreGive(semSched);
}


/**
 * @func    ButtonHandler
 * @brief   Interrupt handler for button press event
 * @param   None
 * @return  None
 * @reference   TM4C123GH6PM Example
 */
void ButtonHandler(void)
{
    // Disable interrupt for GPIO pin 4 on Port F
    GPIOIntDisable(GPIO_PORTF_BASE, GPIO_PIN_4);

    // Write GPIO pin 1 on Port F to HIGH
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, GPIO_PIN_1);

    // Set abort flags for services 1 and 2
    abortS1 = true;
    abortS2 = true;
```

```
    // Release semaphores for services 3, 4, and 5
    xSemaphoreGive(semS3);
    xSemaphoreGive(semS4);
    xSemaphoreGive(semS5);

    // Record the event start time
    event_start = xTaskGetTickCount();
}


/**
 * @func    Sequencer_thread
 * @brief   Task for managing the sequencing of events
 * @param   params: Pointer to task parameters (unused)
 * @return  None
 * @reference   FreeRTOS API Documentation - Semaphore Management
 */
static void Sequencer_thread(void *params)
{
    // Static variable to keep track of scheduler count
    static volatile uint32_t schedCnt = 0;

    // Print a message indicating that the sequencer has started
    UARTprintf("Sequencer Started at %u msec\n", xTaskGetTickCount());

    // Infinite loop for sequencer operation
    while (1)
    {
        // Wait indefinitely for the scheduler semaphore
        xSemaphoreTake(semSched, portMAX_DELAY);
        // Increment the scheduler count
        schedCnt++;

        // Check if it's time to release semaphore S1
        if (schedCnt % T1 == 0 && !abortS1)
        {
            xSemaphoreGive(semS1);
        }

        // Check if it's time to release semaphore S2
        if (schedCnt % T2 == 0 && !abortS2)
        {
            xSemaphoreGive(semS2);
        }

        // Check if it's time to release semaphore S3 (abort condition)
        if (schedCnt % T1 == 0 && abortS1)
        {
```

```
            xSemaphoreGive(semS3);
        }

        // Check if it's time to release semaphore S4 (abort condition)
        if (schedCnt % T2 == 0 && abortS2)
        {
            xSemaphoreGive(semS4);
        }
    }
}


/**
 * @func    service1
 * @brief   Task for processing data from accelerometer
 * @param   params: Pointer to task parameters (unused)
 * @return  None
 * @reference   FreeRTOS API Documentation - Semaphore Management
 */
static void service1(void *params)
{
    // Static variables to store accelerometer and gyroscope readings
    static volatile uint16_t acc_xh = 0, acc_xl = 0;
    static volatile uint16_t gyro_xh = 0, gyro_xl = 0;

    // Loop until the abortS1 flag is set
    while (!abortS1)
    {
        // Wait indefinitely for semaphore S1
        xSemaphoreTake(semS1, portMAX_DELAY);

        // Turn on an LED (assuming GPIO_PIN_3 is connected to an LED)
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, GPIO_PIN_3);

        // Read accelerometer data
        acc_xh = (read_from_accelerometer(I2C0_BASE, 0x68, 0x3B) << 8);
        acc_xl = read_from_accelerometer(I2C0_BASE, 0x68, 0x3C);
        acc_xh |= acc_xl;

        // Read gyroscope data
        gyro_xh = (read_from_accelerometer(I2C0_BASE, 0x68, 0x43) << 8);
        gyro_xl = read_from_accelerometer(I2C0_BASE, 0x68, 0x44);

        // Processing accelerometer data
        acc_xh /= ACC_SCALING - ACC_OFFSET;
        speed += acc_xh;

        // Turn off the LED
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, 0);
```

```
        // Print accelerometer data
        //UARTprintf("A: %d\n", acc_xh);
    }

    // Print a message indicating that Service 1 is exiting
    //UARTprintf("S1 exit\n");

    // Delete the task
    vTaskDelete(NULL);
}


/**
 * @func    service2
 * @brief   Task for processing data from temperature sensor
 * @param   params: Pointer to task parameters (unused)
 * @return  None
 * @reference   FreeRTOS API Documentation - Semaphore Management
 */
static void service2(void *params)
{
    // Variable to store temperature reading
    uint16_t temp;

    // Initial duty cycle for PWM
    uint16_t duty_cycle = (PWMGenPeriodGet(PWM0_BASE, PWM_GEN_0) / 4);

    // Loop until the abortS2 flag is set
    while (!abortS2)
    {
        // Wait indefinitely for semaphore S2
        xSemaphoreTake(semS2, portMAX_DELAY);

        // Turn on an LED (assuming GPIO_PIN_2 is connected to an LED)
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);

        // Read temperature data
        temp = tmp_readdata() * 2;

        // Adjust PWM duty cycle based on temperature
        if (temp + TEMP_OFFSET >= 260)
        {
            duty_cycle = (PWMGenPeriodGet(PWM0_BASE, PWM_GEN_0) / 4);
            PWMPulseWidthSet(
                PWM0_BASE, PWM_OUT_0,
                duty_cycle);
        }
        else
```

```
        {
            duty_cycle = (PWMGenPeriodGet(PWM0_BASE, PWM_GEN_0) / 3);
            PWMPulseWidthSet(PWM0_BASE, PWM_OUT_0,
                             duty_cycle);
        }

        // Turn off the LED
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0);

        // Print temperature data
        //UARTprintf("\rT: %d\n", temp);
    }
    // Delete the task
    vTaskDelete(NULL);
}


/**
 * @func    service3
 * @brief   Task for managing seat adjustment mechanism
 * @param   params: Pointer to task parameters (unused)
 * @return  None
 * @reference   FreeRTOS API Documentation - Semaphore Management
 */
static void service3(void *params)
{
    // Variable to count iterations
    int itr = 0;

    // Variable to store end time for service 3
    volatile uint32_t s3_end_time = 0;

    // Loop until the abortS3 flag is set
    while (!abortS3)
    {
        // Check if semaphore S3 is taken
        xSemaphoreTake(semS3, portMAX_DELAY);

        // Adjust PWM pulse width for PWM output 1
        PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1,
                         (PWMGenPeriodGet(PWM0_BASE, PWM_GEN_0) / 10) * 9);

        // Increment iteration count
        itr++;

        // Check if iterations exceed a certain threshold
        if (itr > SEAT_TIGHT_ITER)
        {
            // Activate seat tightening mechanism
```

```
            GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, 0);
            GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7, GPIO_PIN_7);
            PWMPulseWidthSet(PWM0_BASE, PWM_OUT_1,
                             (PWMGenPeriodGet(PWM0_BASE, PWM_GEN_0) / 10) *
10);
        }

        // Check if iterations exceed the speed offset
        if (itr > SPEED_OFFSET + speed)
        {
            // Exit the loop if conditions are met
            break;
        }
    }

    // Disable PWM output for PWM0 output 1
    PWMOutputState(PWM0_BASE, PWM_OUT_1_BIT, false);

    // Get the end time for service 3
    s3_end_time = xTaskGetTickCount();

    // Check if SAM deadline is missed
    if ((s3_end_time - event_start) > SAM_DEADLINE)
    {
        UARTprintf("SAM deadline missed T: %d S: %d E: %d itr %d\n",
s3_end_time - event_start, event_start, s3_end_time, itr);
    }
    else
    {
        UARTprintf("SAM Successful T: %d S: %d E: %d itr %d\n", s3_end_time -
event_start, event_start, s3_end_time, itr);
    }

    // Delete the task
    vTaskDelete(NULL);
}

/**
 * @func    service4
 * @brief   Task for managing temperature control mechanism
 * @param   params: Pointer to task parameters (unused)
 * @return  None
 * @reference   FreeRTOS API Documentation - Semaphore Management
 */
static void service4(void *params)
{
    // Variable to store temperature reading
    uint16_t temp;
```

```
    // Duty cycle for PWM output
    uint16_t duty_cycle = (PWMGenPeriodGet(PWM0_BASE, PWM_GEN_0) / 4);

    // Variable to store end time for service 4
    uint32_t s4_end_time = 0;

    // Loop until the abortS4 flag is set
    while (!abortS4)
    {
        // Wait indefinitely for semaphore S4
        xSemaphoreTake(semS4, portMAX_DELAY);

        // Turn on an LED (assuming GPIO_PIN_2 is connected to an LED)
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);

        // Read temperature data
        temp = tmp_readdata() * 2;

        // If temperature is above a certain threshold, adjust PWM output
        if (temp + TEMP_OFFSET >= 285)
        {
            PWMPulseWidthSet(
                PWM0_BASE,
                PWM_OUT_0,
                (PWMGenPeriodGet(PWM0_BASE, PWM_GEN_0) / 5) * 2 -
(PWMGenPeriodGet(PWM0_BASE, PWM_GEN_0) / 10));
        }
        else
        {
            PWMPulseWidthSet(
                PWM0_BASE, PWM_OUT_0,
                ((PWMGenPeriodGet(PWM0_BASE, PWM_GEN_0)) / 10) * 10);
        }

        // Get current time
        s4_end_time = xTaskGetTickCount();

        // If deadline is reached, exit loop
        if ((s4_end_time - event_start) >= ABD_DEADLINE)
        {
            break;
        }

        // Turn off the LED
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0);

        // Print temperature data and execution time
```

```
        UARTprintf("\rT: %d\n", temp);
    }

    // Check if ABD deadline is missed
    if ((s4_end_time - event_start) > ABD_DEADLINE)
    {
        UARTprintf("ABD deadline missed T: %d S: %d E: %d\n", s4_end_time -
event_start, event_start, s4_end_time);
    }
    else
    {
        UARTprintf("ABD Successful T: %d S: %d E: %d\n", s4_end_time -
event_start, event_start, s4_end_time);
    }

    // Delete the task
    vTaskDelete(NULL);
}


/**
 * @func    service5
 * @brief   Task for deploying airbag
 * @param   params: Pointer to task parameters (unused)
 * @return  None
 * @reference   FreeRTOS API Documentation - Semaphore Management
 */
static void service5(void *params)
{
    // Wait indefinitely for semaphore S5
    xSemaphoreTake(semS5, portMAX_DELAY);

    // Deploy airbag by writing to servo
    servo_write(10);

    // Get current time
    uint32_t end_time = xTaskGetTickCount();

    // Print message indicating successful airbag deployment along with
execution time
    UARTprintf("Air bag deployed T: %d S: %d E: %d\n", end_time - event_start,
event_start, end_time);

    // Delete the task
    vTaskDelete(NULL);
}


/**
```

```
 * @func    main
 * @brief   Entry point of the program
 * @param   None
 * @return  None
 * @reference   FreeRTOS API Documentation - Task Management
 */
void main(void)
{
    // Enable lazy stacking for floating-point instructions
    ROM_FPULazyStackingEnable();

    // Configure the system clock to use the main oscillator with a 16MHz
crystal
    ROM_SysCtlClockSet(
            SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN
                    | SYSCTL_XTAL_16MHZ);

    // Configure UART communication
    ConfigureUART();

    // Initialize I2C0
    init_i2c0();

    // Initialize button
    button_init();

    // Initialize temperature sensor
    tmp_sensor_init();

    // Initialize PWM for heating
    heating_pwm_init();

    // Initialize PWM for seat control
    seat_pwm_init();

    // Initialize servo motor
    servo_init();
    servo_write(90);

    // Initialize Timer0 for scheduler
    timer0_init();

    // Print a welcome message
    UARTprintf("\nRTES Final Project\n\n");

    // Initialize semaphores
    semSched = xSemaphoreCreateMutex();
    semS1 = xSemaphoreCreateMutex();
```

```
    semS2 = xSemaphoreCreateMutex();
    semS3 = xSemaphoreCreateMutex();
    semS4 = xSemaphoreCreateMutex();
    semS5 = xSemaphoreCreateMutex();

    // Restrict services to start before the sequencer
    xSemaphoreTake(semS1, portMAX_DELAY);
    xSemaphoreTake(semS2, portMAX_DELAY);
    xSemaphoreTake(semS3, portMAX_DELAY);
    xSemaphoreTake(semS4, portMAX_DELAY);
    xSemaphoreTake(semS5, portMAX_DELAY);

    // Create tasks for sequencer and services
    xTaskCreate(Sequencer_thread, "Sequencer Thread", 128, NULL,
                tskIDLE_PRIORITY + 6, NULL);
    xTaskCreate(service1, "Service 1", 128, NULL, tskIDLE_PRIORITY + 4, NULL);
    xTaskCreate(service2, "Service 2", 128, NULL, tskIDLE_PRIORITY + 3, NULL);
    xTaskCreate(service3, "Service 3", 128, NULL, tskIDLE_PRIORITY + 5, NULL);
    xTaskCreate(service4, "Service 4", 128, NULL, tskIDLE_PRIORITY + 4, NULL);
    xTaskCreate(service5, "Service 5", 128, NULL, tskIDLE_PRIORITY + 3, NULL);

    // Start the FreeRTOS scheduler
    vTaskStartScheduler();

    // Code should never reach here, but keep an infinite loop just in case
    while (1)
        ;
}
```

```
/**************************************************************************
**
 * Copyright (C) 2023 by Jithendra and Suhas
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra, Suhas and the University of Colorado are not liable
for
 * any misuse of this material.
 *
**************************************************************************/
/**
 * @file accelerometer.h
 * @brief Accelerometer initialization and access APIs
```

```
 * @author Jithendra and Suhas
 * @date 2024-4-29
 */
#include <stdint.h>
void init_i2c0(void);
void write_to_accelerometer(uint32_t ui32Base, uint8_t ui8SlaveAddr,
                            uint8_t nargs, ...);
uint32_t read_from_accelerometer(uint32_t ui32Base, uint8_t ui8SlaveAddr,
                                 uint8_t reg);
```

```
/*****************************************************************************
**
 * Copyright (C) 2023 by Jithendra and Suhas
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra, Suhas and the University of Colorado are not liable
for
 * any misuse of this material.
 *
*****************************************************************************/
/**
 * @file accelerometer.c
 * @brief Accelerometer initialization and access APIs
 * @author Jithendra and Suhas
 * @date 2024-4-29
 */
#define TARGET_IS_TM4C123_RA1
#include "accelerometer.h"
#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_ints.h"
#include "inc/hw_i2c.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "driverlib/i2c.h"
#include "stdarg.h"

/**
 * @func    init_i2c0
 * @brief   Initializes I2C0 peripheral and GPIO port B
 * @param   None
```

```
 * @return  None
 * @reference   TivaWare Peripheral Driver Library User's Guide
 */
void init_i2c0(void)
{
    // Enable the I2C0 peripheral and GPIO port B
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C0);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);

    // Configure the pin muxing for I2C0 functions on port B2 (SCL) and B3
(SDA)
    ROM_GPIOPinConfigure(GPIO_PB2_I2C0SCL);
    ROM_GPIOPinConfigure(GPIO_PB3_I2C0SDA);

    // Set pin types for I2C0
    GPIOPinTypeI2CSCL(GPIO_PORTB_BASE, GPIO_PIN_2);
    ROM_GPIOPinTypeI2C(GPIO_PORTB_BASE, GPIO_PIN_3);

    // Initialize the I2C master with the system clock frequency and enable
high-speed mode
    ROM_I2CMasterInitExpClk(I2C0_BASE, SysCtlClockGet(), true);

    // Wake up MPU6050 from sleep by writing to its PWR_MGMT_1 register
    write_to_accelerometer(I2C0_BASE, 0x68, 2, 0x6B, 0x00);
}
/**
 * @func    write_to_accelerometer
 * @brief   Writes data to the specified register of the accelerometer via I2C
communication
 * @param   ui32Base: Base address of the I2C module
 * @param   ui8SlaveAddr: 7-bit slave address of the accelerometer device
 * @param   nargs: Number of arguments to be written, including the register
address
 * @param   ...: Variable number of arguments to be written, starting with the
register address followed by data bytes
 * @return  None
 */
void write_to_accelerometer(uint32_t ui32Base, uint8_t ui8SlaveAddr,
                            uint8_t nargs, ...)
{
    // Initialize variable argument list
    va_list vargs;
    va_start(vargs, nargs);

    // Extract the register address from the variable arguments
    uint8_t regAddress = va_arg(vargs, uint8_t); // First argument is always
the register address
```

```
    // Set the slave address and direction to write
    I2CMasterSlaveAddrSet(ui32Base, ui8SlaveAddr, false);

    // Put the register address into the master data register
    I2CMasterDataPut(ui32Base, regAddress);

    // Initiate a burst send start
    I2CMasterControl(ui32Base, I2C_MASTER_CMD_BURST_SEND_START);

    // Wait until the I2C master is not busy
    while (I2CMasterBusy(ui32Base))
        ;

    // Loop through the remaining arguments
    uint8_t i = 0;
    for (i = 1; i < nargs; i++)
    {
        // Put the next data byte into the master data register
        I2CMasterDataPut(ui32Base, va_arg(vargs, uint8_t));

        // Determine whether to send a finish or continue command based on the
current byte's position
        if (i == nargs - 1)
        {
            I2CMasterControl(ui32Base, I2C_MASTER_CMD_BURST_SEND_FINISH);
        }
        else
        {
            I2CMasterControl(ui32Base, I2C_MASTER_CMD_BURST_SEND_CONT);
        }

        // Wait until the I2C master is not busy
        while (I2CMasterBusy(ui32Base))
            ;
    }

    // End variable argument list
    va_end(vargs);
}

/**
 * @func    read_from_accelerometer
 * @brief   Reads data from the specified register of the accelerometer via
I2C communication
 * @param   ui32Base: Base address of the I2C module
 * @param   ui8SlaveAddr: 7-bit slave address of the accelerometer device
 * @param   reg: Register address to read from
 * @return  Data read from the specified register
```

```
 */
uint32_t read_from_accelerometer(uint32_t ui32Base, uint8_t ui8SlaveAddr,
                                 uint8_t reg)
{
    // Set the slave address and direction to write the register address
    I2CMasterSlaveAddrSet(ui32Base, ui8SlaveAddr, false);

    // Put the register address into the master data register
    I2CMasterDataPut(ui32Base, reg);

    // Initiate a single send command
    I2CMasterControl(ui32Base, I2C_MASTER_CMD_SINGLE_SEND);

    // Wait until the I2C master is not busy
    while (I2CMasterBusy(ui32Base))
        ;

    // Set the slave address and direction to read data
    I2CMasterSlaveAddrSet(ui32Base, ui8SlaveAddr, true);

    // Initiate a single receive command
    I2CMasterControl(ui32Base, I2C_MASTER_CMD_SINGLE_RECEIVE);

    // Wait until the I2C master is not busy
    while (I2CMasterBusy(ui32Base))
        ;

    // Return the data read from the master data register
    return I2CMasterDataGet(ui32Base);
}
```

```
/*****************************************************************************
**
 * Copyright (C) 2023 by Jithendra and Suhas
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra, Suhas and the University of Colorado are not liable
for
 * any misuse of this material.
 *
*****************************************************************************/
/**
 * @file button_interrupt.h
```

```
 * @brief Button initialization
 * @author Jithendra and Suhas
 * @date 2024-4-29
 */
void button_init();
```

```
/*****************************************************************************
**
 * Copyright (C) 2023 by Jithendra and Suhas
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra, Suhas and the University of Colorado are not liable
for
 * any misuse of this material.
 *
*****************************************************************************/
/**
 * @file button_interrupt.c
 * @brief Button initialization
 * @author Jithendra and Suhas
 * @date 2024-4-29
 */
 #define TARGET_IS_TM4C123_RA1
 #include "button_interrupt.h"
 #include <stdbool.h>
 #include <stdint.h>
 #include "inc/hw_memmap.h"
 #include "inc/hw_types.h"
 #include "inc/hw_ints.h"
 #include "driverlib/gpio.h"
 #include "driverlib/pin_map.h"
 #include "driverlib/rom.h"
 #include "driverlib/sysctl.h"

 /**
  * @func    button_init
  * @brief   Initializes the GPIO pins and interrupts for button functionality
  * @param   None
  * @return  None
  * @reference   TM4C123GH6PM examples
  */
 void button_init()
 {
```

```
    // Enable the GPIO Port F peripheral
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);

    // Configure pins 1, 2, and 3 of Port F as GPIO outputs
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_2);
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_3);

    // Configure pin 4 of Port F as a GPIO input
    GPIOPinTypeGPIOInput(GPIO_PORTF_BASE, GPIO_PIN_4);

    // Configure pin 4 of Port F with a weak pull-up resistor
    GPIOPadConfigSet(GPIO_PORTF_BASE, GPIO_PIN_4, GPIO_STRENGTH_2MA,
GPIO_PIN_TYPE_STD_WPU);

    // Enable interrupts for pin 4 of Port F
    GPIOIntEnable(GPIO_PORTF_BASE, GPIO_PIN_4);

    // Configure pin 4 of Port F to trigger interrupts on falling edge
    GPIOIntTypeSet(GPIO_PORTF_BASE, GPIO_PIN_4, GPIO_FALLING_EDGE);

    // Enable the GPIO Port F interrupt
    ROM_IntEnable(INT_GPIOF);

    // Enable the global interrupt flag
    ROM_IntMasterEnable();
 }
```

```
/*****************************************************************************
**
 * Copyright (C) 2023 by Jithendra and Suhas
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra, Suhas and the University of Colorado are not liable
for
 * any misuse of this material.
 *
*****************************************************************************/
/**
 * @file pwm_control.h
 * @brief PWM initialization and control API's
 * @author Jithendra and Suhas
```

```
 * @date 2024-4-29
 */
void heating_pwm_init();
void seat_pwm_init();
void servo_init();
void servo_write(float deg);
```

```
/*****************************************************************************
**
 * Copyright (C) 2023 by Jithendra and Suhas
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra, Suhas and the University of Colorado are not liable
for
 * any misuse of this material.
 *
*****************************************************************************/
/**
 * @file pwm_control.c
 * @brief PWM initialization and control API's
 * @author Jithendra and Suhas
 * @date 2024-4-29
 */
#define TARGET_IS_TM4C123_RA1
#include "pwm_control.h"
#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_ints.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "driverlib/pwm.h"

/**
 * @func    heating_pwm_init
 * @brief   Initializes PWM for heating control
 * @param   None
 * @return  None
 * @reference   TM4C123GH6PM examples
 */
```

```
void heating_pwm_init()
{
    // Set the PWM clock to run at the system clock divided by 1
    SysCtlPWMClockSet(SYSCTL_SYSDIV_1);

    // Enable PWM0 peripheral and GPIO port B
    SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);

    // Configure pin B6 as PWM output
    GPIOPinConfigure(GPIO_PB6_M0PWM0);
    GPIOPinTypePWM(GPIO_PORTB_BASE, GPIO_PIN_6);

    // Configure PWM generator 0 in up-down count mode with no synchronization
    PWMGenConfigure(PWM0_BASE, PWM_GEN_0, PWM_GEN_MODE_UP_DOWN |
PWM_GEN_MODE_NO_SYNC);

    // Set the period of PWM generator 0 to 200000 cycles
    PWMGenPeriodSet(PWM0_BASE, PWM_GEN_0, 200000);

    // Set the initial pulse width to 25% of the period
    PWMPulseWidthSet(PWM0_BASE, PWM_OUT_0, PWMGenPeriodGet(PWM0_BASE,
PWM_GEN_0) / 4);

    // Enable PWM output for PWM0 output 0
    PWMOutputState(PWM0_BASE, PWM_OUT_0_BIT, true);

    // Enable PWM generator 0
    PWMGenEnable(PWM0_BASE, PWM_GEN_0);
}

/**
 * @func    seat_pwm_init
 * @brief   Initializes PWM for seat control
 * @param   None
 * @return  None
 * @reference   TM4C123GH6PM examples
 */
void seat_pwm_init()
{
    // Set the PWM clock to run at the system clock divided by 1
    SysCtlPWMClockSet(SYSCTL_SYSDIV_1);

    // Enable PWM0 peripheral and GPIO port B
    SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);

    // Configure pin B7 as PWM output
```

```
    GPIOPinConfigure(GPIO_PB7_M0PWM1);
    GPIOPinTypePWM(GPIO_PORTB_BASE, GPIO_PIN_7);

    // Configure PWM generator 0 in up-down count mode with no synchronization
    PWMGenConfigure(PWM0_BASE, PWM_GEN_0, PWM_GEN_MODE_UP_DOWN |
PWM_GEN_MODE_NO_SYNC);

    // Set the period of PWM generator 0 to 200000 cycles
    PWMGenPeriodSet(PWM0_BASE, PWM_GEN_0, 200000);

    // Enable PWM output for PWM0 output 1
    PWMOutputState(PWM0_BASE, PWM_OUT_1_BIT, true);

    // Enable PWM generator 0
    PWMGenEnable(PWM0_BASE, PWM_GEN_0);

    // Enable GPIO port C and configure pins C6 and C7 as GPIO outputs
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
    GPIOPinTypeGPIOOutput(GPIO_PORTC_BASE, GPIO_PIN_6);
    GPIOPinTypeGPIOOutput(GPIO_PORTC_BASE, GPIO_PIN_7);

    // Set pin C6 high and pin C7 low
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_6, GPIO_PIN_6);
    GPIOPinWrite(GPIO_PORTC_BASE, GPIO_PIN_7, 0);
}

/**
 * @func    servo_init
 * @brief   Initializes PWM for servo motor control
 * @param   None
 * @return  None
 * @reference   https://github.com/smalik007/Servo-controlled-by-Tiva-C-
Series-ARM-Cortex-M3-
 */
void servo_init()
{
    // Set the PWM clock to the system clock divided by 8
    SysCtlPWMClockSet(SYSCTL_PWMDIV_8);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM1);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    // Configure pin A7 as PWM output
    GPIOPinConfigure(GPIO_PA7_M1PWM3);
    GPIOPinTypePWM(GPIO_PORTA_BASE, GPIO_PIN_7);

    // Configure PWM generator 1 in down count mode with no synchronization
    PWMGenConfigure(PWM1_BASE, PWM_GEN_1, PWM_GEN_MODE_DOWN |
PWM_GEN_MODE_NO_SYNC);
```

```
    // Set the period of PWM generator 1 to 40000 cycles
    PWMGenPeriodSet(PWM1_BASE, PWM_GEN_1, 40000);
}

/**
 * @func    servo_write
 * @brief   Writes the desired angle to the servo motor
 * @param   deg: Desired angle (0 to 180 degrees)
 * @return  None
 * @reference   https://github.com/smalik007/Servo-controlled-by-Tiva-C-
Series-ARM-Cortex-M3-
 */
void servo_write(float deg)
{
    // Calculate the duty cycle based on the desired angle
    float duty = ((deg / 90) + 0.4);
    float ticks = duty * 2;
    float divf = (40 / ticks);
    int divfact = (int) divf;

    // Set the PWM pulse width based on the calculated divisor factor
    PWMPulseWidthSet(PWM1_BASE, PWM_OUT_3, PWMGenPeriodGet(PWM1_BASE,
PWM_GEN_1) / divfact);
    PWMOutputState(PWM1_BASE, PWM_OUT_3_BIT, true);
    PWMGenEnable(PWM1_BASE, PWM_GEN_1);

    // Delay for servo motor to reach the desired position
    SysCtlDelay((SysCtlClockGet() * 0.3) / 3);

    // Disable PWM output after reaching the desired position
    PWMOutputState(PWM1_BASE, PWM_OUT_3_BIT, false);
    PWMGenDisable(PWM1_BASE, PWM_GEN_1);
}
```

```
/*****************************************************************************
**
 * Copyright (C) 2023 by Jithendra and Suhas
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra, Suhas and the University of Colorado are not liable
for
 * any misuse of this material.
```

```
 *
*************************************************************************/
/**
 * @file temperature_sensor.h
 * @brief Temperature sensor initialization and Access API's
 * @author Jithendra and Suhas
 * @date 2024-4-29
 */
#include <stdint.h>
#define TEMP_OFFSET (5)
void tmp_sensor_init();
uint16_t tmp_readdata(void);
```

```
/*************************************************************************
**
 * Copyright (C) 2023 by Jithendra and Suhas
 *
 * Redistribution, modification, or use of this software in source or binary
 * forms is permitted as long as the files maintain this copyright. Users are
 * permitted to modify this and use it to learn about the field of embedded
 * software. Jithendra, Suhas and the University of Colorado are not liable
for
 * any misuse of this material.
 *
*************************************************************************/
/**
 * @file temperature_sensor.c
 * @brief Temperature sensor initialization and Access API's
 * @author Jithendra and Suhas
 * @date 2024-4-29
 */
#define TARGET_IS_TM4C123_RA1
#include "temperature_sensor.h"
#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_ints.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "driverlib/ssi.h"
/**
 * @func    tmp_sensor_init
 * @brief   Initializes the temperature sensor (TMP) via SPI (SSI0)
 * @param   None
```

```
 * @return  None
 * @reference   TMP Sensor Datasheet, TM4C123GH6PM example
 */
void tmp_sensor_init()
{
    // Enable the SSI0 peripheral and GPIO port A
    SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    // Configure pins PA2, PA3, PA4, PA5 as SSI0 pins
    GPIOPinConfigure(GPIO_PA2_SSI0CLK);
    GPIOPinConfigure(GPIO_PA3_SSI0FSS);
    GPIOPinConfigure(GPIO_PA4_SSI0RX);
    GPIOPinConfigure(GPIO_PA5_SSI0TX);
    GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_5 | GPIO_PIN_4 | GPIO_PIN_3 |
GPIO_PIN_2);

    // Configure SSI0 to operate as a master, using Motorola SPI mode 0, with
a clock speed of 1 MHz and 16-bit data frames
    SSIConfigSetExpClk(SSI0_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_0,
SSI_MODE_MASTER, 1000000, 16);

    // Enable the SSI0 module
    SSIEnable(SSI0_BASE);
}

/**
 * @func    tmp_readdata
 * @brief   Reads temperature data from the TMP sensor
 * @param   None
 * @return  uint16_t: Temperature data (in degrees Celsius)
 * @reference   TMP Sensor Datasheet, TM4C123GH6PM example
 */
uint16_t tmp_readdata(void)
{
    uint32_t ui32Data;
    uint16_t tempData;

    // Wait until there is no data in the receive FIFO
    while (SSIDataGetNonBlocking(SSI0_BASE, &ui32Data))
        ;

    // Send a dummy byte (0x00) to initiate the SPI transaction and receive
data from the temperature sensor
    SSIDataPut(SSI0_BASE, 0x00);
    SSIDataGet(SSI0_BASE, &ui32Data);

    // Convert the received data to temperature using the sensor's data format
```

```
    tempData = (uint16_t)(ui32Data >> 2) * 0.25;

    // Return the temperature data
    return tempData;
}
```

```
//*****************************************************************************
*
//
// Startup code for use with TI's Code Composer Studio.
//
// Copyright (c) 2011-2014 Texas Instruments Incorporated.  All rights
reserved.
// Software License Agreement
//
// Software License Agreement
//
// Texas Instruments (TI) is supplying this software for use solely and
// exclusively on TI's microcontroller products. The software is owned by
// TI and/or its suppliers, and is protected under applicable copyright
// laws. You may not combine this software with "viral" open-source
// software in order to form a larger program.
//
// THIS SOFTWARE IS PROVIDED "AS IS" AND WITH ALL FAULTS.
// NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT
// NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. TI SHALL NOT, UNDER ANY
// CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
// DAMAGES, FOR ANY REASON WHATSOEVER.
//
//*****************************************************************************
*

#include <stdint.h>

//*****************************************************************************
*
//
// Forward declaration of the default fault handlers.
//
//*****************************************************************************
*
void ResetISR(void);
static void NmiSR(void);
static void FaultISR(void);
```

```
static void IntDefaultHandler(void);

//*****************************************************************************
*
//
// External declaration for the reset handler that is to be called when the
// processor is started
//
//*****************************************************************************
*
extern void _c_int00(void);

//*****************************************************************************
*
//
// Linker variable that marks the top of the stack.
//
//*****************************************************************************
*
extern uint32_t __STACK_TOP;

//*****************************************************************************
*
//
// External declarations for the interrupt handlers used by the application.
//
//*****************************************************************************
*
extern void ButtonHandler(void);
extern void Timer0IntHandler(void);
extern void xPortPendSVHandler(void);
extern void vPortSVCHandler(void);
extern void xPortSysTickHandler(void);

//*****************************************************************************
*
//
// The vector table.  Note that the proper constructs must be placed on this
to
// ensure that it ends up at physical address 0x0000.0000 or at the start of
// the program if located at a start address other than 0.
//
//*****************************************************************************
*
#pragma DATA_SECTION(g_pfnVectors, ".intvecs")
void (* const g_pfnVectors[])(void) =
{
    (void (*)(void))((uint32_t)&__STACK_TOP),
```

```
                                        // The initial stack pointer
    ResetISR,                           // The reset handler
    NmiSR,                              // The NMI handler
    FaultISR,                           // The hard fault handler
    IntDefaultHandler,                  // The MPU fault handler
    IntDefaultHandler,                  // The bus fault handler
    IntDefaultHandler,                  // The usage fault handler
    0,                                  // Reserved
    0,                                  // Reserved
    0,                                  // Reserved
    0,                                  // Reserved
    vPortSVCHandler,                    // SVCall handler
    IntDefaultHandler,                  // Debug monitor handler
    0,                                  // Reserved
    xPortPendSVHandler,                  // The PendSV handler
    xPortSysTickHandler,                  // The SysTick handler
    IntDefaultHandler,                  // GPIO Port A
    IntDefaultHandler,                  // GPIO Port B
    IntDefaultHandler,                  // GPIO Port C
    IntDefaultHandler,                  // GPIO Port D
    IntDefaultHandler,                  // GPIO Port E
    IntDefaultHandler,                  // UART0 Rx and Tx
    IntDefaultHandler,                  // UART1 Rx and Tx
    IntDefaultHandler,                  // SSI0 Rx and Tx
    IntDefaultHandler,                  // I2C0 Master and Slave
    IntDefaultHandler,                  // PWM Fault
    IntDefaultHandler,                  // PWM Generator 0
    IntDefaultHandler,                  // PWM Generator 1
    IntDefaultHandler,                  // PWM Generator 2
    IntDefaultHandler,                  // Quadrature Encoder 0
    IntDefaultHandler,                  // ADC Sequence 0
    IntDefaultHandler,                  // ADC Sequence 1
    IntDefaultHandler,                  // ADC Sequence 2
    IntDefaultHandler,                  // ADC Sequence 3
    IntDefaultHandler,                  // Watchdog timer
    Timer0IntHandler,                   // Timer 0 subtimer A
    IntDefaultHandler,                  // Timer 0 subtimer B
    IntDefaultHandler,                  // Timer 1 subtimer A
    IntDefaultHandler,                  // Timer 1 subtimer B
    IntDefaultHandler,                  // Timer 2 subtimer A
    IntDefaultHandler,                  // Timer 2 subtimer B
    IntDefaultHandler,                  // Analog Comparator 0
    IntDefaultHandler,                  // Analog Comparator 1
    IntDefaultHandler,                  // Analog Comparator 2
    IntDefaultHandler,                  // System Control (PLL, OSC, BO)
    IntDefaultHandler,                  // FLASH Control
    ButtonHandler,                  // GPIO Port F
    IntDefaultHandler,                  // GPIO Port G
```

```
    IntDefaultHandler,                      // GPIO Port H
    IntDefaultHandler,                      // UART2 Rx and Tx
    IntDefaultHandler,                      // SSI1 Rx and Tx
    IntDefaultHandler,                      // Timer 3 subtimer A
    IntDefaultHandler,                      // Timer 3 subtimer B
    IntDefaultHandler,                      // I2C1 Master and Slave
    IntDefaultHandler,                      // Quadrature Encoder 1
    IntDefaultHandler,                      // CAN0
    IntDefaultHandler,                      // CAN1
    0,                                      // Reserved
    0,                                      // Reserved
    IntDefaultHandler,                      // Hibernate
    IntDefaultHandler,                      // USB0
    IntDefaultHandler,                      // PWM Generator 3
    IntDefaultHandler,                      // uDMA Software Transfer
    IntDefaultHandler,                      // uDMA Error
    IntDefaultHandler,                      // ADC1 Sequence 0
    IntDefaultHandler,                      // ADC1 Sequence 1
    IntDefaultHandler,                      // ADC1 Sequence 2
    IntDefaultHandler,                      // ADC1 Sequence 3
    0,                                      // Reserved
    0,                                      // Reserved
    IntDefaultHandler,                      // GPIO Port J
    IntDefaultHandler,                      // GPIO Port K
    IntDefaultHandler,                      // GPIO Port L
    IntDefaultHandler,                      // SSI2 Rx and Tx
    IntDefaultHandler,                      // SSI3 Rx and Tx
    IntDefaultHandler,                      // UART3 Rx and Tx
    IntDefaultHandler,                      // UART4 Rx and Tx
    IntDefaultHandler,                      // UART5 Rx and Tx
    IntDefaultHandler,                      // UART6 Rx and Tx
    IntDefaultHandler,                      // UART7 Rx and Tx
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    IntDefaultHandler,                      // I2C2 Master and Slave
    IntDefaultHandler,                      // I2C3 Master and Slave
    IntDefaultHandler,                      // Timer 4 subtimer A
    IntDefaultHandler,                      // Timer 4 subtimer B
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
```

```
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    IntDefaultHandler,                      // Timer 5 subtimer A
    IntDefaultHandler,                      // Timer 5 subtimer B
    IntDefaultHandler,                      // Wide Timer 0 subtimer A
    IntDefaultHandler,                      // Wide Timer 0 subtimer B
    IntDefaultHandler,                      // Wide Timer 1 subtimer A
    IntDefaultHandler,                      // Wide Timer 1 subtimer B
    IntDefaultHandler,                      // Wide Timer 2 subtimer A
    IntDefaultHandler,                      // Wide Timer 2 subtimer B
    IntDefaultHandler,                      // Wide Timer 3 subtimer A
    IntDefaultHandler,                      // Wide Timer 3 subtimer B
    IntDefaultHandler,                      // Wide Timer 4 subtimer A
    IntDefaultHandler,                      // Wide Timer 4 subtimer B
    IntDefaultHandler,                      // Wide Timer 5 subtimer A
    IntDefaultHandler,                      // Wide Timer 5 subtimer B
    IntDefaultHandler,                      // FPU
    0,                                      // Reserved
    0,                                      // Reserved
    IntDefaultHandler,                      // I2C4 Master and Slave
    IntDefaultHandler,                      // I2C5 Master and Slave
    IntDefaultHandler,                      // GPIO Port M
    IntDefaultHandler,                      // GPIO Port N
    IntDefaultHandler,                      // Quadrature Encoder 2
    0,                                      // Reserved
    0,                                      // Reserved
    IntDefaultHandler,                      // GPIO Port P (Summary or P0)
    IntDefaultHandler,                      // GPIO Port P1
    IntDefaultHandler,                      // GPIO Port P2
    IntDefaultHandler,                      // GPIO Port P3
    IntDefaultHandler,                      // GPIO Port P4
    IntDefaultHandler,                      // GPIO Port P5
    IntDefaultHandler,                      // GPIO Port P6
    IntDefaultHandler,                      // GPIO Port P7
    IntDefaultHandler,                      // GPIO Port Q (Summary or Q0)
    IntDefaultHandler,                      // GPIO Port Q1
    IntDefaultHandler,                      // GPIO Port Q2
    IntDefaultHandler,                      // GPIO Port Q3
```

```
    IntDefaultHandler,                      // GPIO Port Q4
    IntDefaultHandler,                      // GPIO Port Q5
    IntDefaultHandler,                      // GPIO Port Q6
    IntDefaultHandler,                      // GPIO Port Q7
    IntDefaultHandler,                      // GPIO Port R
    IntDefaultHandler,                      // GPIO Port S
    IntDefaultHandler,                      // PWM 1 Generator 0
    IntDefaultHandler,                      // PWM 1 Generator 1
    IntDefaultHandler,                      // PWM 1 Generator 2
    IntDefaultHandler,                      // PWM 1 Generator 3
    IntDefaultHandler                       // PWM 1 Fault
};

//*****************************************************************************
*
//
// This is the code that gets called when the processor first starts execution
// following a reset event.  Only the absolutely necessary set is performed,
// after which the application supplied entry() routine is called.  Any fancy
// actions (such as making decisions based on the reset cause register, and
// resetting the bits in that register) are left solely in the hands of the
// application.
//
//*****************************************************************************
*
void
ResetISR(void)
{
    //
    // Jump to the CCS C initialization routine.  This will enable the
    // floating-point unit as well, so that does not need to be done here.
    //
    __asm("    .global _c_int00\n"
          "    b.w     _c_int00");
}

//*****************************************************************************
*
//
// This is the code that gets called when the processor receives a NMI.  This
// simply enters an infinite loop, preserving the system state for examination
// by a debugger.
//
//*****************************************************************************
*
static void
NmiSR(void)
{
```

```
    //
    // Enter an infinite loop.
    //
    while(1)
    {
    }
}

//*****************************************************************************
*
//
// This is the code that gets called when the processor receives a fault
// interrupt.  This simply enters an infinite loop, preserving the system
state
// for examination by a debugger.
//
//*****************************************************************************
*
static void
FaultISR(void)
{
    //
    // Enter an infinite loop.
    //
    while(1)
    {
    }
}

//*****************************************************************************
*
//
// This is the code that gets called when the processor receives an unexpected
// interrupt.  This simply enters an infinite loop, preserving the system
state
// for examination by a debugger.
//
//*****************************************************************************
*
static void
IntDefaultHandler(void)
{
    //
    // Go into an infinite loop.
    //
    while(1)
    {
    }
```

```
}
```