ROBOT ARM

ECEN 5623 REAL TIME EMBEDDED SYSTEMS

VISHNU DODBALLAPUR

VENKATA JANAKIRAMA PRADYUMNA GUDLURU

Introduction

Objective: A motion controlled robotic arm which moves based on the movement of a hand on a FreeRTOS system.

This Robotic Arm can be used in many applications. According to the robotic arm in this proof of concept, it can be used as a robotic crane based on the input steering from the user. To mimic the motion of the hand with a robotic arm, we plan to use a MPU6050, a gyroscope and accelerometer sensor as the sensory device for sensing the motion of hand and a robotic arm with micro servo motors. The MPU6050 sensor gives the data of position and angle of the hand. This sensor works on I2C communication protocol. The write-read-read process is used to receive data through I2C communication. Using this value, we calibrate the system to a temporary origin. Using the origin as the reference point, we get the hand movement and read the present values from the sensor. By using the computation algorithm, the direction is identified and updated in a message queue.

We will use the Tiva TM4C123 for our processor, the MPU6050 as our accelerometer, and the SNAM1500 4 DOF Wood Robotic Mechanical Arm for our robot arm. They are shown in Figures 1, 2, and 3 below.

Figure 1: Tiva Launchpad (TM4C123)

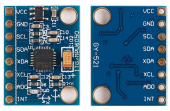

Figure 2: MPU6050

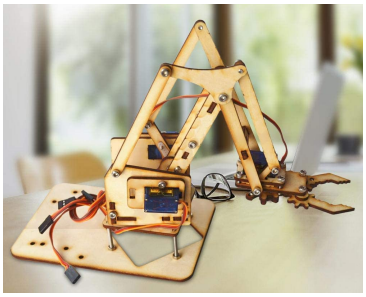

Figure 3: SNAM1500 4 DOF Wood Robotic Mechanical Arm

Functional Requirements

1. **System will run FreeRTOS**

   The Tiva board was chosen to run an RTOS system for this project.

2. **Processor will communicate with accelerometer via I2C**

   An I2C communication driver will have to be written to communicate with the MPU6050 and read the appropriate values.

3. **Processor will set position of robot elements with 50Hz PWM signal**

   The processor will have to enable timers and GPIO pins to output a 50Hz PWM signal to control the servos.

4. **Processor will read XYZ values from the accelerometer**

   The processor will have to read raw XYZ values from the MPU6050.

5. **Processor will convert XYZ values to pitch and roll**

   Using trigonometric functions, the processor will convert the XYZ values read from the accelerometer into pitch and roll values to be used by the robot arm.

6. **Processor will set position of robot base depending on roll**

   The roll value calculated by the processor will set the position of the base of the robot.

7. **Processor will set position of robot arm depending on pitch**

   The pitch value calculated by the processor will set the position of the arm of the robot.

8. **Base of robot will have 180 degrees of motion**

   The base of the robot is capable of 180 degrees of motion, and will be allowed to move for that whole half-circle.

9. **Arm of robot will have 60 degrees of motion**

The arm of the robot does not have the full half-circle range of motion, and will be restricted to 60 degrees of motion.

Real Time Requirements

While trying to find an acceptable response time for our robot, we stumbled across several papers for high-grade robot arms that had response times around 50ms and some even in the single digit ms range. This was not feasible for our robot with plastic and wooden components. Finally, we found a paper on making a robot arm with commercially available components by Christian Smith and Henrik I. Christensen [1] which set the end-to-end response time for the robot arm at 500ms.

With this response time now set, we moved on to figuring out our latencies. The MPU6050 takes 10ms for each new XYZ reading [2]. This is our input latency.

Our servo motor takes 100ms to rotate 60 degrees. Since the most our servo can rotate is 180 degrees, our worst case latency here is 300 ms. This is our output latency.

Knowing our response times and our latencies, we can calculate our effective deadline for our system. Our effective deadline is 500ms - 300ms - 10ms = **190ms**. A diagram detailing these calculations is shown below in Figure 4.

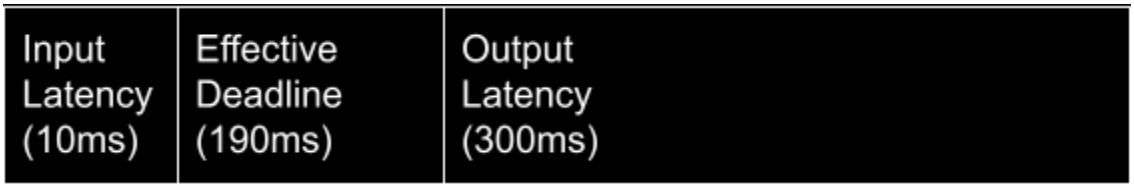

Figure 4: Response Time

Functional Design Overview

In Figures 5, 6, and 7, we will show diagrams detailing the block diagram, the software components, and the software flow.

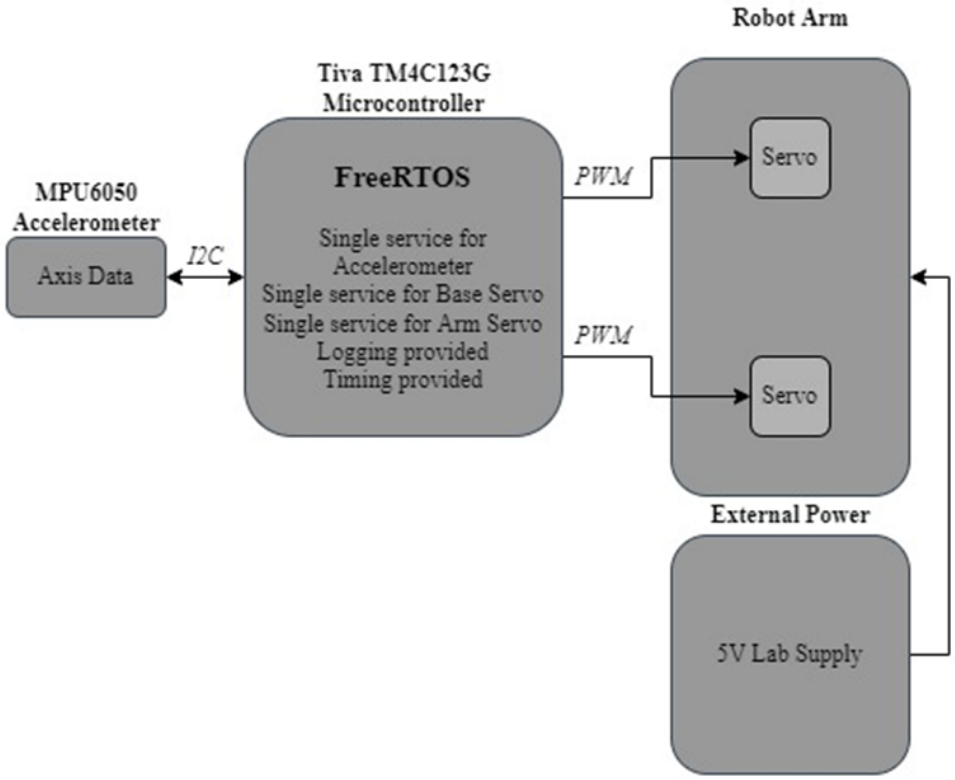

Figure 5: Block Diagram

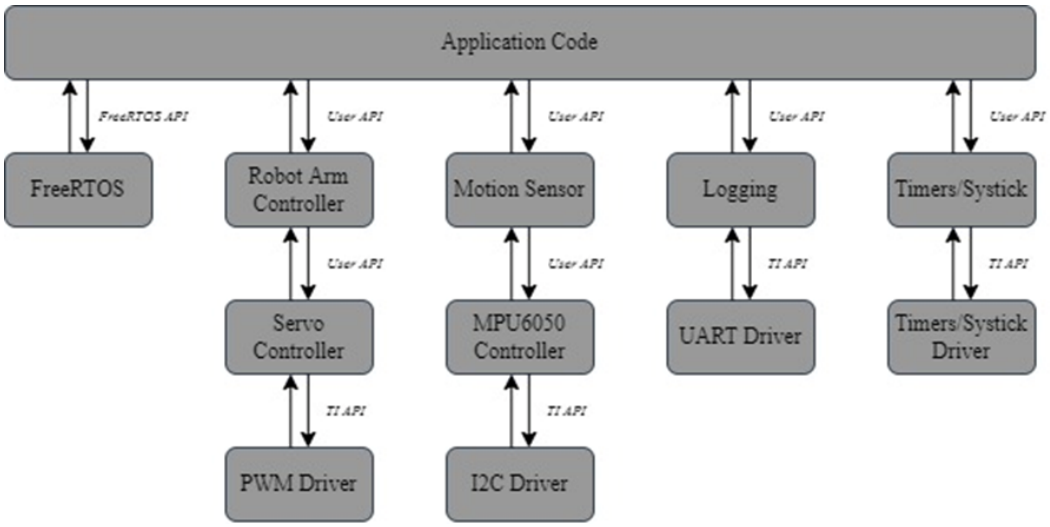

Figure 6: Software Component Diagram

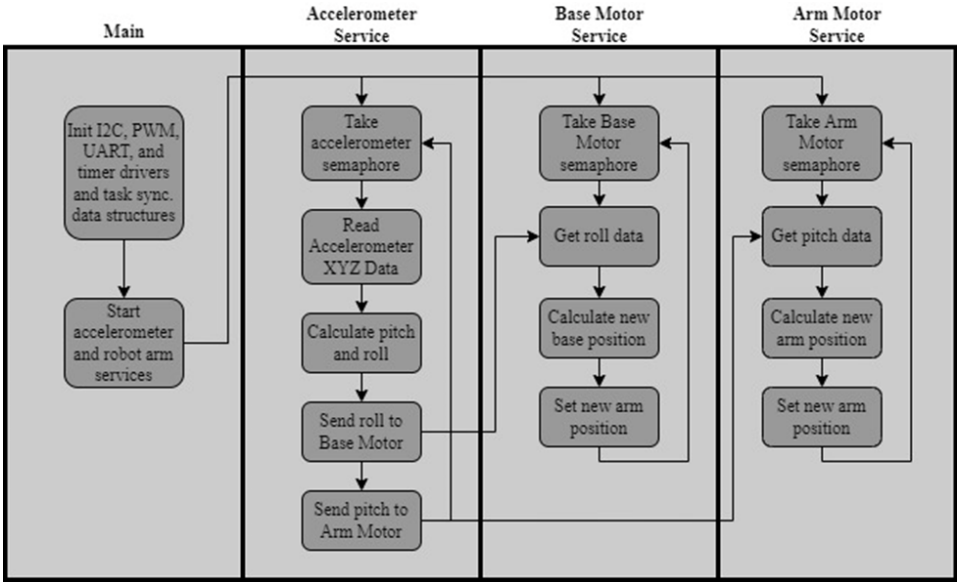

Figure 7: Software Flow Diagram

In our system, the processor uses I2C, PWM, and GPIO drivers to communicate with the peripheral devices - I2C to communicate with the accelerometer, and PWM/GPIO to communicate with the servo motors to set the positions of the robot arm. Additionally, we use the UART driver to log timestamps for validation and use hardware timers to calculate those timestamps. For task synchronization, we use software timers for task timing, semaphores to signal when to begin each task, and message queues to pass data between services.

We created three services for each peripheral we will be controlling - one service for the accelerometer, one for the base motor, and one for the arm motor. As per our functionality requirements we set, we read the raw XYZ values from the accelerometer, and then calculate the pitch and roll. One message queue is used to pass roll data from the accelerometer service to the base motor service, and another message queue is used to pass pitch data from the accelerometer service to the arm motor service.

In the base motor service and the arm motor service, we map the roll and pitch values received from the accelerometer service to a duty cycle that sets the position of the base and arm respectively. All three services now wait until they are signaled via their respective semaphore to run again.

Real Time Analysis and Design

For each of our services, the Ci, Ti, and Di is shown below in Table 1.

Table 1: Ci, Ti, and Di

Service	**Ci**	**Ti**	**Di**
Service 1: Accelerometer Service	2.11ms	25ms	25ms
Service 2: Base Motor Service	134.5µs	50ms	50ms
Service 3: Arm Motor Service	138µs	50ms	50ms

The deadlines for the robot arm services were going to be our restricting services, so we set those to be 50ms to meet our effective deadline that we previously calculated. We set the deadline of the accelerometer service to be half of the motor services (25ms) to find a happy medium so that we were not oversampling and flooding the message queues, but also not missing any readings by sampling too infrequently.

We are using a Rate-Monotonic scheduling policy for our system, which means that Ti = Di. It also means that since the accelerometer service runs at a higher frequency than the motor services, it will have a higher priority than the motor services.

The Ci values listed in Table 1 are the WCETs for each service. We calculated the WCET for the accelerometer service by calculating the time taken to read the raw XYZ values read from the MPU6050, the time taken to calculate pitch and roll from the XYZ values, and the time taken to place the pitch and roll values in the message queues. Since the I2C bus runs at 100000 bps and each byte transaction is 9 bits long (8 bytes of data along with the ACK or NACK), one byte transaction takes 900µs. The total transaction for reading XYZ data is 10 bytes, with about a half byte transaction's worth of time between each byte. The final calculation is shown below.

$$\frac{90\mu s}{Byte} * 10\ bytes\ *\ 1.5\ = 1.35ms$$

To calculate the time taken for the pitch and roll calculations and placing those values into the message queues, we viewed the assembly output in Code Composer Studio by going to the debug window and clicking View ->Window -> Disassembly. We then made an estimate of the number of instructions taken by these operations, and found the number to be 3040. We estimated the CPI to be 4 because each instruction has a different CPI - add instructions have a

CPI of 1, while divide instructions have a max CPI of 12 [4]. We also know that we are running at a clock frequency of 16MHz. The calculation for this is shown below.

$$3040\ instructions\ * \frac{4\ clocks}{cycle} * \frac{16M\ cycles}{second} = 760\mu s$$

So, our final WCET for the accelerometer service is 1.35ms + 760µs = 2.11ms.

Our WCET for the motor services is more straightforward, only needing the instruction count, the estimated CPI, and the clock frequency. The instruction count for the base motor service was found to be 40 instructions, and the instruction count for the arm motor service was found to be 54 instructions. The calculation for the WCET for the base motor service is shown below.

$$40\ instructions\ * \frac{4\ clocks}{cycle} * \frac{16M\ cycles}{second} = 134.5\mu s$$

Similarly, the calculation for the WCET for the arm motor service is shown below.

$$54\ instructions\ * \frac{4\ clocks}{cycle} * \frac{16M\ cycles}{second} = 138\mu s$$

Now that we have our Ci, Ti, and Di, we will use Cheddar to help us determine whether or not our system is feasible. Figure 8 the Cheddar output for our system.

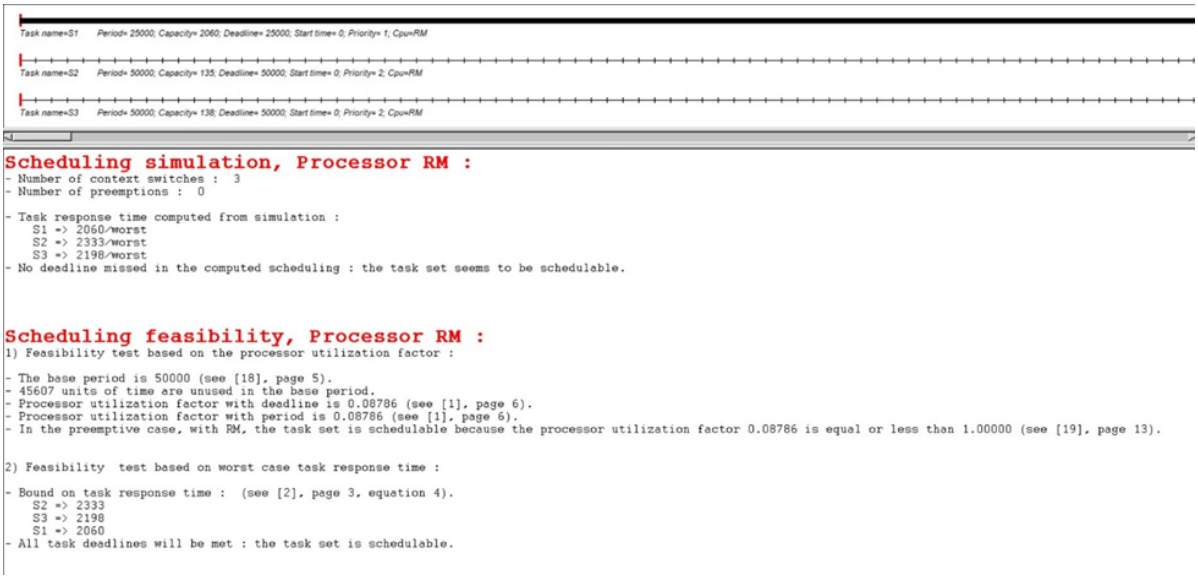

Figure 8: Cheddar Simulation for Proposed System

We can see from the output that no deadlines were missed, and that the set of services appears to be schedulable. To cement our confidence, we will perform the Rate-Monotonic Least Upper Bound test. While this test does not necessarily determine infeasibility if the test fails, it does determine feasibility if the test passes.

The RM LUB test is shown below.

$U < m(2^{\frac{1}{m}} - 1)$, where U is the CPU utilization by the set of services, and m is the number of services. Since we have 3 services, we know our CPU utilization must be less than

$3(2^{\frac{1}{3}} - 1) = 0.7797.$

Our CPU utilization can be calculated with the following equation.

$$U = \sum_{i=1}^{m} \frac{Ci}{Ti}$$

Plugging in our Ci and Ti values, we get

$U = \frac{2.11ms}{25ms} + \frac{134.5\mu s}{50ms} + \frac{138\mu s}{50ms} = 0.08985.$

We can see that 0.08985 is clearly less than 0.7797, so we can say that our set of services is feasible by the RM LUB test.

Proof Of Concept – Example and Tests

For the robot arm, we tried to work on building the arm from scratch. The final result is shown in Figure 9.

Figure 9: Side View of the Robotic Arm - Proof of Concept

This is a side view of the robotic arm, with two degrees of freedom. There are two servo motors connected one for the base rotation which moves based on the roll input from the sensor and the other for the arm movement which moves based on the pitch value obtained from the sensor. The I2C and PWM are configured with GPIO pins for SDA and SCL as PB.2 and PB.3, and the two servo motors with PF.1 and PF.3 of the TIVA board. A 5V DC supply is given to the servo motors through a power supply.

The complete source code for the configuration on Tiva board is available on the following link: https://github.com/vido2373/Robot_Arm_ECEN5623

The video demo is available on the following link: https://drive.google.com/drive/u/1/folders/1jf0fsjtgGdSbq6QL4RqAG5DAA3nsOnGK

Verification/Tests:

For the verification of our robot arm, we identified three methods. One being the logging of timing using the 1microsec resolution timer, second being the logging of accelerometer data and third by manually verifying the angle of roll and pitch using a protractor.

The following screenshot is the logged statemts using UARTPrintf().

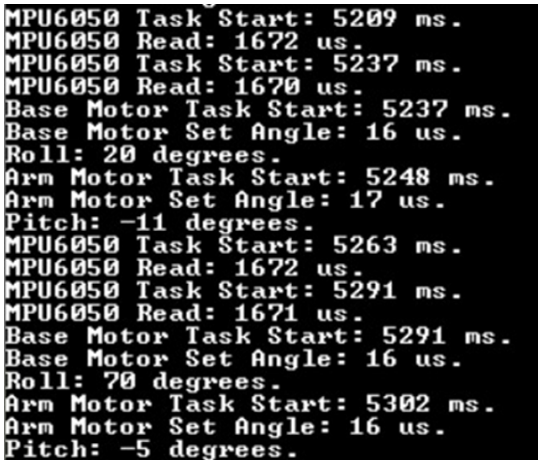

Figure 10: LOG Details of Timing and Angle from Accelerometer

These logs tell us many things. First, we can see that the services are executing with the correct priority - the accelerometer service completes first, and then the motor services complete. We can also see that the angles detected by the accelerometer are within the accepted ranges - the base motor angle goes between -90 and 90 degrees, and the arm motor goes between -30 and 30 degrees.

There are a few observations with regards to timing as well. First, we can see that the start times of each service do not line up exactly with what we had set for their period. The accelerometer service, for example, seems to run every 26ms to 28ms rather than every 25ms. This is likely due to the extra delays that come from logging all of this information - UART logging is slow compared to the rest of these tasks. We could have validated this behavior in another manner, such as toggling a GPIO and tracing the timing of the toggle on an oscilloscope, but there was too much data we wanted to see at one time, and we chose to accept this small amount of jitter caused by our logging.

The second observation is that the execution times are quite a bit less than the WCET that we had previously calculated. The Ci of the accelerometer service appears to be in the area of 1.672ms, the Ci of the base motor service appears to be 16µs, and the Ci of the arm motor service appears to be 17µs. The discrepancy between our measured Ci and our calculated Ci is likely due to the assumptions we made that the average CPI is 4. Many of the instructions executed could have had a CPI less than 4, resulting in a Ci less than the WCET we calculated. However, the measured Ci times do not exceed the WCET we had calculated, and ended up giving our system a margin of error of roughly 26% for our accelerometer service, 841% for our base motor service, and 812% for our arm motor service.

For the robotic angle, we manually measured the angle using a protractor and it is visible from the top view of the robotic arm as in the picture below.

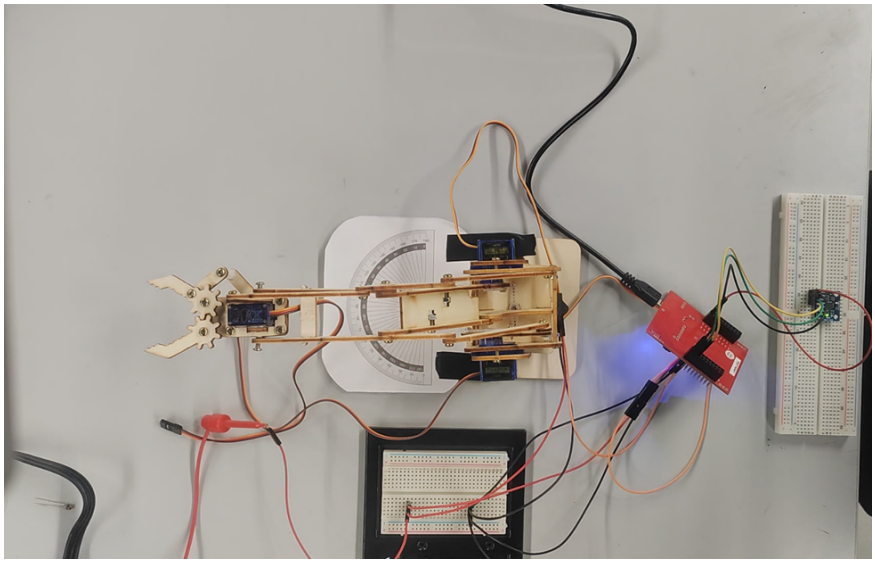

Figure 11: Top View of the Robot Arm with Protractor on Base

The following picture is an example for a +90 degree roll. This is a front view and as visible, the arm is tilted completely towards the right with the sensor going 90 degree vertical from the ground.

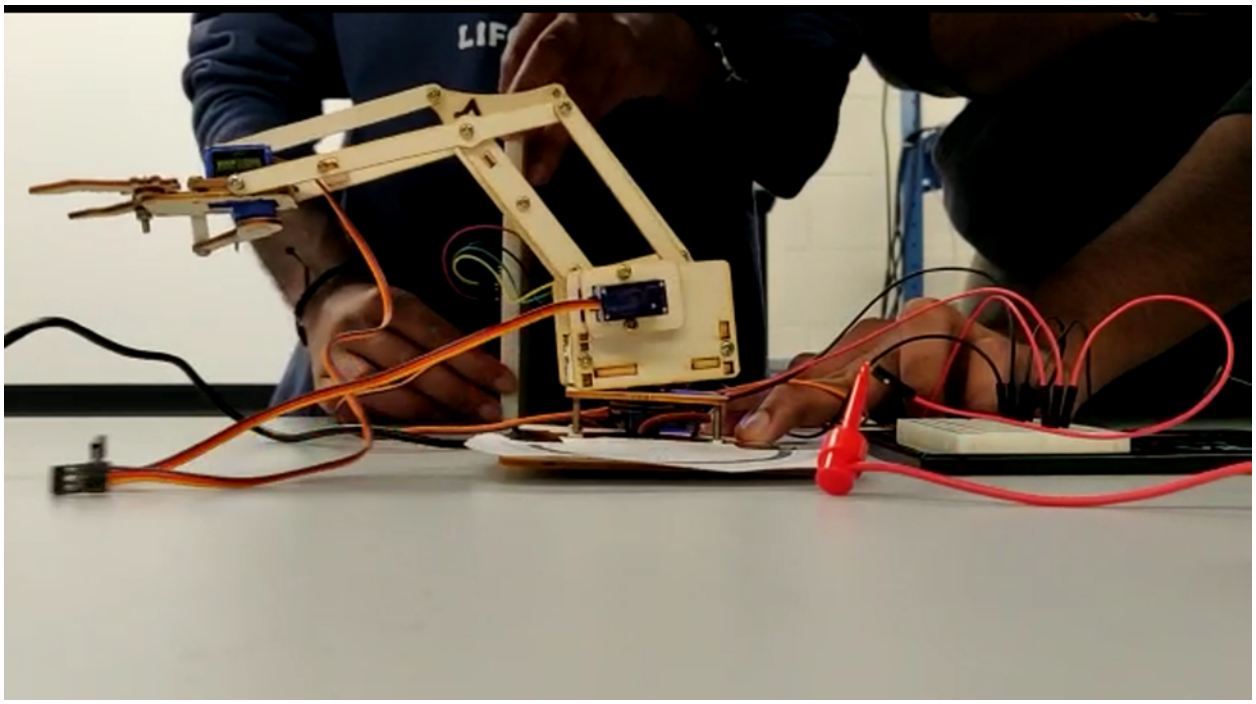

Figure 12: Front View of Robotic Armwith Roll = +90 degrees

Conclusion

The major takeaway from this project is working on the FreeRTOS system and configuring the I2C and PWM drivers using the available library files. Understanding the switching between the tasks, using semaphores and using a hardware timer from accessing data, and working with the PWM to logging the data using UARTPrintf().

References

[1] Smith, C., & Christensen, H. I. (2009, December). Robot Manipulators. Atlanta. Retrieved April 29, 2022.

[2] “MPU-6000 and MPU-6050 Product Specification Revision 3.4 MPU-6000/MPU-6050 Product Specification,” 2013. [Online]. Available: https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf

[3] “SG90 micro servo servo motor for Arduino and raspberry-pi robotics ... - kjell.com.” [Online]. Available: https://www.kjell.com/globalassets/mediaassets/701916_87897_datasheet_en.pdf. [Accessed: 16-Apr-2022].

[4] Morales, M., 2022. *An Introduction to the Tiva C Series Platform of Microcontrollers*. [online] Available at: <https://www.ti.com/lit/wp/spmy010/spmy010.pdf> [Accessed 3 May 2022].

Appendix

```
//*****************************************************************************
//
// Robot_Arm_ECEN5623.c - Motion controlled robot arm system.
// Attribution: freertos_demo project in Tiva SDK
//
// Copyright (c) 2012-2017 Texas Instruments Incorporated.  All rights reserved.
// 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.
//
// This is part of revision 2.1.4.178 of the EK-TM4C123GXL Firmware Package.
//
//*****************************************************************************

#include <robot_system_timers.h>
#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "driverlib/interrupt.h"
#include "driverlib/timer.h"
#include "driverlib/pwm.h"
#include "utils/uartstdio.h"
#include "system_tasks.h"
#include "mpu6050.h"
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
#include "timers.h"

xSemaphoreHandle g_pUARTSemaphore;

//*****************************************************************************
//
// The error routine that is called if the driver library encounters an error.
//
//*****************************************************************************
#ifdef DEBUG
void
__error__(char *pcFilename, uint32_t ui32Line)
{
}
```

```
#endif

//*****************************************************************************
//
// This hook is called by FreeRTOS when an stack overflow error is detected.
//
//*****************************************************************************
void vApplicationStackOverflowHook(xTaskHandle *pxTask, char *pcTaskName)
{
    //
    // This function can not return, so loop forever.  Interrupts are disabled
    // on entry to this function, so no processor interrupts will interrupt
    // this loop.
    //
    while(1);
}

//*****************************************************************************
//
// Configure the UART and its pins.  This must be called before UARTprintf().
//
//*****************************************************************************
void ConfigureUART(void)
{
    //
    // Enable the GPIO Peripheral used by the UART.
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    //
    // Enable UART0
    //
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

    //
    // Configure GPIO Pins for UART mode.
    //
    ROM_GPIOPinConfigure(GPIO_PA0_U0RX);
    ROM_GPIOPinConfigure(GPIO_PA1_U0TX);
    ROM_GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    //
    // Use the internal 16MHz oscillator as the UART clock source.
    //
    UARTClockSourceSet(UART0_BASE, UART_CLOCK_PIOSC);

    //
    // Initialize the UART for console I/O.
    //
    UARTStdioConfig(0, 115200, 16000000);
}

//*****************************************************************************
//
// Configure the PWM for servo motors.
//
//*****************************************************************************
void ConfigurePWM(void)
{
    //Configure PWM Clock divide system clock by 64
    ROM_SysCtlPWMClockSet(SYSCTL_PWMDIV_64);
```

```
    // Enable the peripherals used by this program.
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
     ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM1);  //The Tiva Launchpad has two modules (0 and
1). Module 1 covers the LED pins

    //Configure PF1,PF2,PF3 Pins as PWM
    GPIOPinConfigure(GPIO_PF1_M1PWM5);
    GPIOPinConfigure(GPIO_PF2_M1PWM6);
    GPIOPinConfigure(GPIO_PF3_M1PWM7);
    GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3);

    //Configure PWM Options
    //PWM_GEN_2 Covers M1PWM4 and M1PWM5
    //PWM_GEN_3 Covers M1PWM6 and M1PWM7
    PWMGenConfigure(PWM1_BASE, PWM_GEN_2, PWM_GEN_MODE_DOWN | PWM_GEN_MODE_NO_SYNC);
    PWMGenConfigure(PWM1_BASE, PWM_GEN_3, PWM_GEN_MODE_DOWN | PWM_GEN_MODE_NO_SYNC);

    //Set the Period (expressed in clock ticks)
    PWMGenPeriodSet(PWM1_BASE, PWM_GEN_2, PWM_PERIOD_TICKS);
    PWMGenPeriodSet(PWM1_BASE, PWM_GEN_3, PWM_PERIOD_TICKS);

    //Set PWM duty
    PWMPulseWidthSet(PWM1_BASE, PWM_OUT_5, ((PWM_TICKS_0 + PWM_TICKS_180)/2));
    PWMPulseWidthSet(PWM1_BASE, PWM_OUT_7, ((PWM_TICKS_0 + PWM_TICKS_60)/2));

    // Enable the PWM generator
    PWMGenEnable(PWM1_BASE, PWM_GEN_2);
    PWMGenEnable(PWM1_BASE, PWM_GEN_3);

    // Turn on the Output pins
    PWMOutputState(PWM1_BASE, PWM_OUT_5_BIT | PWM_OUT_6_BIT | PWM_OUT_7_BIT, true);
}

//*****************************************************************************
//
// Configure Timer 5 for timestamping
//
//*****************************************************************************
void ConfigureTimer5(void)
{
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER5);
    TimerClockSourceSet(TIMER5_BASE, TIMER_CLOCK_SYSTEM);
    TimerConfigure(TIMER5_BASE, TIMER_CFG_SPLIT_PAIR | TIMER_CFG_A_PERIODIC);
    TimerPrescaleSet(TIMER5_BASE, TIMER_A, 16);
}

//*****************************************************************************
//
// Initialize FreeRTOS and start the initial set of tasks.
//
//*****************************************************************************
int main(void)
{
    const uint32_t ui32Ms = 1;

    ROM_SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC | SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);

    //
    // Initialize the UART and configure it for 115,200, 8-N-1 operation.
    //
    ConfigureUART();
```

```
    //
    // Initialize PWM for servos
    //
    ConfigurePWM();

    //
    // Initialize Timer5 for timestamping
    //
    ConfigureTimer5();

    //
    // Print demo introduction.
    //
    UARTprintf("Final Project\n");
    UARTprintf("Sys Clock Speed: %d Hz\n", ROM_SysCtlClockGet());

    //
    // Create a mutex to guard the UART.
    //
    g_pUARTSemaphore = xSemaphoreCreateMutex();

    vInitTimers(ui32Ms);

    //
    // Create the Fib 1 Task.
    //
    if(MPU6050TaskInit((void *)0) != 0)
    {
        while(1);
    }

    //
    // Create the Base Motor Task.
    //
    if(BaseMotorTaskInit((void *)0) != 0)
    {
        while(1);
    }

    //
    // Create the Arm Motor Task.
    //
    if(ArmMotorTaskInit((void *)0) != 0)
    {
        while(1);
    }


    //
    // Start the scheduler.  This should not return.
    //
    vTaskStartScheduler();

    //
    // In case the scheduler returns for some reason, print an error and loop
    // forever.
    //

    while(1);
}
```

```
//*****************************************************************************
//
// priorities.h - Priorities for the various FreeRTOS tasks.
//
// Copyright (c) 2012-2017 Texas Instruments Incorporated.  All rights reserved.
// 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.
//
// This is part of revision 2.1.4.178 of the EK-TM4C123GXL Firmware Package.
//
//*****************************************************************************

#ifndef __PRIORITIES_H__
#define __PRIORITIES_H__

//*****************************************************************************
//
// The priorities of the various tasks.
//
//*****************************************************************************
#define PRIORITY_MPU6050_TASK       2
#define PRIORITY_MOTOR_TASK         1


#endif // __PRIORITIES_H__
```

```
//*****************************************************************************
//
// system_tasks.h - Prototypes for the system tasks.
//
// Copyright (c) 2012-2017 Texas Instruments Incorporated.  All rights reserved.
// 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.
//
// This is part of revision 2.1.4.178 of the EK-TM4C123GXL Firmware Package.
//
//*****************************************************************************

#ifndef __SYSTEM_TASKS_H__
#define __SYSTEM_TASKS_H__

#include <stdint.h>

#define PWM_PERIOD_TICKS    (5000)
#define PWM_TICKS_0             (125) // 0 degrees
#define PWM_TICKS_180             (625) // 180 degrees
#define PWM_TICKS_60       (290) // 60 degrees

//*****************************************************************************
//
// Prototypes for the system Tasks.
//
//*****************************************************************************
uint32_t MPU6050TaskInit(void* pvParameters);
uint32_t BaseMotorTaskInit(void* pvParameters);
uint32_t ArmMotorTaskInit(void* pvParameters);

#endif // __SYSTEM_TASKS_H__
```

```
//*****************************************************************************
//
// system_tasks.c - System tasks.
//
// Copyright (c) 2012-2017 Texas Instruments Incorporated.  All rights reserved.
// 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.
//
// This is part of revision 2.1.4.178 of the EK-TM4C123GXL Firmware Package.
//
//*****************************************************************************

#include "system_tasks.h"
#include <stdbool.h>
#include <stdint.h>
#include <math.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "driverlib/timer.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "driverlib/pwm.h"
#include "utils/uartstdio.h"
#include "robot_system_timers.h"
#include "priorities.h"
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
#include "mpu6050.h"

//*****************************************************************************
//
// The stack size for the system tasks.
//
//*****************************************************************************
#define TASK_STACK_SIZE         128         // Stack size in words

#define COUNTS_PER_G            (8192.0)

typedef struct motor_data_s {
    float pitch;
    float roll;
} motor_data_t;


//*****************************************************************************
```

```
//
// The item size and queue size for the Processing message queue.
//
//*****************************************************************************
#define PROCESSING_ITEM_SIZE           sizeof(motor_data_t)
#define PROCESSING_QUEUE_SIZE          128

xSemaphoreHandle g_pMpuSemaphore;
xSemaphoreHandle g_pBaseMotorSemaphore;
xSemaphoreHandle g_pArmMotorSemaphore;

xQueueHandle g_pBaseMotorDataQueue;
xQueueHandle g_pArmMotorDataQueue;

extern xSemaphoreHandle g_pUARTSemaphore;

//*****************************************************************************
//
// MPU6050 Task.
//
//*****************************************************************************
static void MPU6050Task(void *pvParameters)
{
    uint8_t ui8AccelBuffer[6];
    int16_t i16X, i16Y, i16Z;
    motor_data_t xMotorData;
    float fX, fY, fZ;
    float fPrevPitch, fPrevRoll;

    TimerEnable(TIMER5_BASE, TIMER_A);
    vStartMpuTimer();
    vStartBaseMotorTimer();
    vStartArmMotorTimer();
    uint16_t t1, t2;

    while (1)
    {
        xSemaphoreTake(g_pMpuSemaphore, portMAX_DELAY);
        xSemaphoreTake(g_pUARTSemaphore, portMAX_DELAY);
        UARTprintf("MPU6050 Task Start: %d ms.\n", ui32GetTickMs());
        xSemaphoreGive(g_pUARTSemaphore);

        t1 = (uint16_t)TimerValueGet(TIMER5_BASE, TIMER_A);

        vMPU6050_Read(0x3B, &ui8AccelBuffer[0], 6);
        i16X = (uint16_t)(ui8AccelBuffer[0] << 8 | ui8AccelBuffer[1]);
        i16Y = (uint16_t)(ui8AccelBuffer[2] << 8 | ui8AccelBuffer[3]);
        i16Z = (uint16_t)(ui8AccelBuffer[4] << 8 | ui8AccelBuffer[5]);
        fX = i16X / COUNTS_PER_G;
        fY = i16Y / COUNTS_PER_G;
        fZ = i16Z / COUNTS_PER_G;
        xMotorData.pitch = (atan2(fX, sqrt(fY * fY + fZ*fZ))*180.0)/M_PI;
        xMotorData.roll = (atan2(fY, sqrt(fX * fX + fZ*fZ))*180.0)/M_PI;

        if ((xMotorData.roll > fPrevRoll + 5) || (xMotorData.roll < fPrevRoll - 5))
        {
//            UARTprintf("Send to base motor service...\n");
            xQueueSend(g_pBaseMotorDataQueue, &xMotorData, portMAX_DELAY);
        }
        if ((xMotorData.pitch > fPrevPitch + 5) || (xMotorData.pitch < fPrevPitch - 5))
        {
            xQueueSend(g_pArmMotorDataQueue, &xMotorData, portMAX_DELAY);
```

```
        }

        fPrevPitch = xMotorData.pitch;
        fPrevRoll = xMotorData.roll;

        t2 = (uint16_t)TimerValueGet(TIMER5_BASE, TIMER_A);

        xSemaphoreTake(g_pUARTSemaphore, portMAX_DELAY);
        UARTprintf("MPU6050 Read: %d us.\n", (uint16_t)(t1 - t2));
        xSemaphoreGive(g_pUARTSemaphore);
    }
}

//*****************************************************************************
//
// Base Motor Task.
//
//*****************************************************************************
static void BaseMotorTask(void *pvParameters)
{
    uint16_t t1, t2;
    motor_data_t xMotorData;
    uint32_t ui32DutyCycleTicks;

    while (1)
    {
        xSemaphoreTake(g_pBaseMotorSemaphore, portMAX_DELAY);

        xSemaphoreTake(g_pUARTSemaphore, portMAX_DELAY);
        UARTprintf("Base Motor Task Start: %d ms.\n", ui32GetTickMs());
        xSemaphoreGive(g_pUARTSemaphore);

        t1 = (uint16_t)TimerValueGet(TIMER5_BASE, TIMER_A);
        if(xQueueReceive(g_pBaseMotorDataQueue, &xMotorData, 0) == pdPASS)
        {

                        ui32DutyCycleTicks  =  ((xMotorData.roll  *  500)/180)  +  ((PWM_TICKS_0  +
PWM_TICKS_180)/2);
            PWMPulseWidthSet(PWM1_BASE, PWM_OUT_5, ui32DutyCycleTicks);

            t2 = (uint16_t)TimerValueGet(TIMER5_BASE, TIMER_A);

            xSemaphoreTake(g_pUARTSemaphore, portMAX_DELAY);
            UARTprintf("Base Motor Set Angle: %d us.\n", (uint16_t)(t1 - t2));
            UARTprintf("Roll: %d degrees.\n", (int)xMotorData.roll);
            xSemaphoreGive(g_pUARTSemaphore);
        }
    }
}

//*****************************************************************************
//
// Arm Motor Task.
//
//*****************************************************************************
static void ArmMotorTask(void *pvParameters)
{
    uint16_t t1, t2;
    motor_data_t xMotorData;
    uint32_t ui32DutyCycleTicks;

    while (1)
```

```
    {
        xSemaphoreTake(g_pArmMotorSemaphore, portMAX_DELAY);
        xSemaphoreTake(g_pUARTSemaphore, portMAX_DELAY);
        UARTprintf("Arm Motor Task Start: %d ms.\n", ui32GetTickMs());
        xSemaphoreGive(g_pUARTSemaphore);

        t1 = (uint16_t)TimerValueGet(TIMER5_BASE, TIMER_A);

        if(xQueueReceive(g_pArmMotorDataQueue, &xMotorData, 0) == pdPASS)
        {

            if (xMotorData.pitch < -30)
            {
                xMotorData.pitch = -30;
            }
            else if (xMotorData.pitch > 30)
            {
                xMotorData.pitch = 30;
            }

                        ui32DutyCycleTicks  =  ((xMotorData.pitch  *  500)/180)  +  ((PWM_TICKS_0  +
PWM_TICKS_60)/2);
            PWMPulseWidthSet(PWM1_BASE, PWM_OUT_7, ui32DutyCycleTicks);

            t2 = (uint16_t)TimerValueGet(TIMER5_BASE, TIMER_A);

            xSemaphoreTake(g_pUARTSemaphore, portMAX_DELAY);
            UARTprintf("Arm Motor Set Angle: %d us.\n", (uint16_t)(t1 - t2));
            UARTprintf("Pitch: %d degrees.\n", (int)xMotorData.pitch);
            xSemaphoreGive(g_pUARTSemaphore);
        }
    }
}

//*****************************************************************************
//
// Initializes the MPU6050 task.
//
//*****************************************************************************
uint32_t MPU6050TaskInit(void* pvParameters)
{
    //
    // Create a counting semaphore for the MPU6050 task.
    //
    g_pMpuSemaphore = xSemaphoreCreateCounting(1, 1);
    uint8_t ui8InitBuffer[1] = { 0x00 };

    vMPU6050_Init();

    vMPU6050_Write(0x6B, ui8InitBuffer, 1);

    g_pBaseMotorDataQueue = xQueueCreate(PROCESSING_QUEUE_SIZE, PROCESSING_ITEM_SIZE);
    g_pArmMotorDataQueue = xQueueCreate(PROCESSING_QUEUE_SIZE, PROCESSING_ITEM_SIZE);

    //
    // Create the Processing task.
    //
    if(xTaskCreate(MPU6050Task, (const portCHAR *)"MPU6050 Task", TASK_STACK_SIZE, pvParameters,
                   tskIDLE_PRIORITY + PRIORITY_MPU6050_TASK, NULL) != pdTRUE)
    {
        return(1);
    }
```

```
    //
    // Success.
    //
    return(0);
}

//*****************************************************************************
//
// Initializes the Base Motor task.
//
//*****************************************************************************
uint32_t BaseMotorTaskInit(void* pvParameters)
{
    //
    // Create a counting semaphore for the Motor task.
    //
    g_pBaseMotorSemaphore = xSemaphoreCreateCounting(1, 1);

    //
    // Create the Processing task.
    //
        if(xTaskCreate(BaseMotorTask, (const portCHAR *)"Base Motor Task", TASK_STACK_SIZE,
pvParameters,
                   tskIDLE_PRIORITY + PRIORITY_MOTOR_TASK, NULL) != pdTRUE)
    {
        return(1);
    }

    //
    // Success.
    //
    return(0);
}

//*****************************************************************************
//
// Initializes the Arm Motor task.
//
//*****************************************************************************
uint32_t ArmMotorTaskInit(void* pvParameters)
{
    //
    // Create a counting semaphore for the Motor task.
    //
    g_pArmMotorSemaphore = xSemaphoreCreateCounting(1, 1);

    //
    // Create the Processing task.
    //
         if(xTaskCreate(ArmMotorTask, (const portCHAR *)"Arm Motor Task", TASK_STACK_SIZE,
pvParameters,
                   tskIDLE_PRIORITY + PRIORITY_MOTOR_TASK, NULL) != pdTRUE)
    {
        return(1);
    }

    //
    // Success.
    //
    return(0);
}
```

```
/*
 * robot_system_timers.h
 *
 *  Created on: Mar 25, 2022
 *      Author: vishn
 */

#ifndef ROBOT_SYSTEM_TIMERS_H_
#define ROBOT_SYSTEM_TIMERS_H_

#include <stdint.h>

//*****************************************************************************
//
// Initializes timer.
//
//*****************************************************************************
void vInitTimers(uint32_t ui32Ms);

//*****************************************************************************
//
// Get ticks in ms.
//
//*****************************************************************************
uint32_t ui32GetTickMs(void);

//*****************************************************************************
//
// Starts MPU timer.
//
//*****************************************************************************
void vStartMpuTimer(void);

//*****************************************************************************
//
// Stops MPU timer.
//
//*****************************************************************************
void vStopMpuTimer(void);

//*****************************************************************************
//
// Starts Base Motor timer.
//
//*****************************************************************************
void vStartBaseMotorTimer(void);

//*****************************************************************************
//
// Starts Arm Motor timer.
//
//*****************************************************************************
void vStartArmMotorTimer(void);



#endif /* ROBOT_SYSTEM_TIMERS_H_ */
```

```
/*
 * robot_system_timer.c
 *
 *  Created on: Mar 25, 2022
 *      Author: vishn
 */

#include "robot_system_timers.h"
#include <stdbool.h>
#include <stdint.h>
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/rom.h"
#include "driverlib/sysctl.h"
#include "FreeRTOS.h"
#include "timers.h"
#include "semphr.h"

#define MS_MULTIPLIER       (1000)

xTimerHandle g_pTimestampTimerHandle;
xTimerHandle g_pMpuTimerHandle;
xTimerHandle g_pBaseMotorTimerHandle;
xTimerHandle g_pArmMotorTimerHandle;

const uint32_t ui32IdMpuTimer = 1;
const uint32_t ui32IdBaseMotorTimer = 2;
const uint32_t ui32IdArmMotorTimer = 3;

static volatile uint32_t g_ui32Counter = 0;

extern xSemaphoreHandle g_pMpuSemaphore;
extern xSemaphoreHandle g_pBaseMotorSemaphore;
extern xSemaphoreHandle g_pArmMotorSemaphore;

//*****************************************************************************
//
// Timer Callback Function.
//
//*****************************************************************************
void vTimerCallback(TimerHandle_t pxTimer)
{
    uint32_t ui32ID;
    g_ui32Counter++;

    ui32ID = (uint32_t)pvTimerGetTimerID(pxTimer);

    if (ui32ID == ui32IdMpuTimer)
    {
        xSemaphoreGive(g_pMpuSemaphore);
    }
    else if (ui32ID == ui32IdBaseMotorTimer)
    {
        xSemaphoreGive(g_pBaseMotorSemaphore);
    }
    else if (ui32ID == ui32IdArmMotorTimer)
    {
        xSemaphoreGive(g_pArmMotorSemaphore);
    }
}

//*****************************************************************************
```

```
//
// Initializes timer.
//
//*****************************************************************************
void vInitTimers(uint32_t ui32Ms)
{
     g_pTimestampTimerHandle = xTimerCreate("Timestamp Timer", (TickType_t)ui32Ms, pdTRUE, (void
*)0, vTimerCallback);
    g_pMpuTimerHandle = xTimerCreate("MPU Timer", (TickType_t)25, pdTRUE, (void *)ui32IdMpuTimer,
vTimerCallback);
      g_pBaseMotorTimerHandle = xTimerCreate("Base Motor Timer", (TickType_t)50, pdTRUE, (void
*)ui32IdBaseMotorTimer, vTimerCallback);
       g_pArmMotorTimerHandle = xTimerCreate("Arm Motor Timer", (TickType_t)50, pdTRUE, (void
*)ui32IdArmMotorTimer, vTimerCallback);
    xTimerStart(g_pTimestampTimerHandle, 0);
}

//*****************************************************************************
//
// Get ticks in ms.
//
//*****************************************************************************
uint32_t ui32GetTickMs(void)
{
    return g_ui32Counter;
}

//*****************************************************************************
//
// Starts MPU timer.
//
//*****************************************************************************
void vStartMpuTimer(void)
{
    xTimerStart(g_pMpuTimerHandle, 0);
}

//*****************************************************************************
//
// Stop MPU timer.
//
//*****************************************************************************
void vStopMpuTimer(void)
{
    xTimerStop(g_pMpuTimerHandle, 0);
}

//*****************************************************************************
//
// Starts Base Motor timer.
//
//*****************************************************************************
void vStartBaseMotorTimer(void)
{
    xTimerStart(g_pBaseMotorTimerHandle, 0);
}

//*****************************************************************************
//
// Starts Base Motor timer.
//
//*****************************************************************************
```

```
void vStartArmMotorTimer(void)
{
    xTimerStart(g_pArmMotorTimerHandle, 0);
}
```

```
/*
 * mpu6050.h
 *
 *  Created on: Apr 14, 2022
 *      Author: vishn
 */

#ifndef MPU6050_H_
#define MPU6050_H_

#include <stdint.h>

#define SLAVE_ADDRESS       (0x68)

void vMPU6050_Init(void);
void vMPU6050_Write(uint8_t ui8Reg, uint8_t* ui8WrBuff, uint8_t ui8WrBuffLen);
void vMPU6050_Read(uint8_t ui8Reg, uint8_t* ui8RdBuff, uint8_t ui8RdBuffLen);



#endif /* MPU6050_H_ */
```

```
/*
 * mpu6050.c
 *
 *  Created on: Apr 14, 2022
 *      Author: vishn
 */

#include "mpu6050.h"

#include <stdbool.h>
#include <stdint.h>
#include "inc/hw_i2c.h"
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/gpio.h"
#include "driverlib/i2c.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "utils/uartstdio.h"

void vMPU6050_Init(void)
{
    ROM_SysCtlPeripheralDisable(SYSCTL_PERIPH_I2C0);
    ROM_SysCtlPeripheralDisable(SYSCTL_PERIPH_GPIOB);
    ROM_SysCtlPeripheralReset(SYSCTL_PERIPH_I2C0);
    //ROM_SysCtlPeripheralReset(SYSCTL_PERIPH_GPIOB);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C0);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
                                    while(!ROM_SysCtlPeripheralReady(SYSCTL_PERIPH_I2C0)         ||
!ROM_SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOB));

    //configure PB2 to SCL and PB3 SDA
    ROM_GPIOPinConfigure(GPIO_PB2_I2C0SCL);
    ROM_GPIOPinConfigure(GPIO_PB3_I2C0SDA);
    ROM_GPIOPinTypeI2CSCL(GPIO_PORTB_BASE, GPIO_PIN_2);
    ROM_GPIOPinTypeI2C(GPIO_PORTB_BASE, GPIO_PIN_3);

    //init I2C Master with 100kbps
    ROM_I2CMasterInitExpClk(I2C0_BASE, ROM_SysCtlClockGet(), false);
}


void vMPU6050_Write(uint8_t ui8Reg, uint8_t* ui8WrBuff, uint8_t ui8WrBuffLen)
{
    uint8_t ui8I;
    //Set device slave address
    I2CMasterSlaveAddrSet(I2C0_BASE, SLAVE_ADDRESS, false);

    //Send command
    I2CMasterDataPut(I2C0_BASE, ui8Reg);
    I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_SEND_START);
    while(I2CMasterBusy(I2C0_BASE));

    //Send data
    I2CMasterDataPut(I2C0_BASE, ui8WrBuff[0]);
    for (ui8I = 1; ui8I < ui8WrBuffLen; ui8I++)
    {
```

```
        I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_SEND_CONT);
        while(I2CMasterBusy(I2C0_BASE));

        I2CMasterDataPut(I2C0_BASE, ui8WrBuff[ui8I]);
    }

    //Send stop bit
    I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_SEND_FINISH);
    while(I2CMasterBusy(I2C0_BASE));
}


void vMPU6050_Read(uint8_t ui8Reg, uint8_t* ui8RdBuff, uint8_t ui8RdBuffLen)
{
    uint8_t ui8I;
    //Set device slave address
    I2CMasterSlaveAddrSet(I2C0_BASE, SLAVE_ADDRESS, false);

    //Send command
    I2CMasterDataPut(I2C0_BASE, ui8Reg);
    I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_SEND_START);
    while(I2CMasterBusy(I2C0_BASE));

    //Send Sr and read bytes
    I2CMasterSlaveAddrSet(I2C0_BASE, SLAVE_ADDRESS, true);

    if (ui8RdBuffLen == 1)
    {
        I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_SINGLE_RECEIVE);
        while(I2CMasterBusy(I2C0_BASE));

        ui8RdBuff[0] = I2CMasterDataGet(I2C0_BASE);
    }
    else
    {
        I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_RECEIVE_START);
        while(I2CMasterBusy(I2C0_BASE));

        ui8RdBuff[0] = I2CMasterDataGet(I2C0_BASE);

        for (ui8I = 1; ui8I < ui8RdBuffLen - 1; ui8I++)
        {
            I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_RECEIVE_CONT);
            while(I2CMasterBusy(I2C0_BASE));

            ui8RdBuff[ui8I] = I2CMasterDataGet(I2C0_BASE);
        }

        I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_RECEIVE_FINISH);
        while(I2CMasterBusy(I2C0_BASE));

        ui8RdBuff[ui8I] = I2CMasterDataGet(I2C0_BASE);
    }
}
```