University of Colorado Boulder

ECEN 5613 – Embedded System Design (Spring-2019)

Final Project Report

Project Details	*Description*
Created By:	**Rushi James Macwan**
Project Title	**Integrated Sensor Network (For an IIoT Service)**

Credits and Courtesy for this work goes to all the authors of the various resources that I have cited and acknowledged for making this project possible. The citations have been numbered accordingly and can be accessed by clicking the links. The images have been cited accordingly and the source link is provided below every image. I duly acknowledge the original authors of the example source codes at Texas Instruments Inc. whose work has been very helpful to me for making this project possible.

Table of Contents

Table of Figures

Abstract

Air pollution and Global Warming are two of the most concerning issues that are currently shaping our climate. In the coming years, the far and wide effects of climate change may become very visible. At this point of time, we have reached a stage where not only measuring pollution is important but is quintessential to understand how we can reduce the hazardous effects of climate change in the future.

Inspired by various companies and organizations trying to fight climate change today, I learnt how often cities are choking under the blanket of smog in these times [1]. In addition to that, fine particulate air pollution is responsible for the vast majority of the 3 million to 4 million annual deaths attributed to air pollution worldwide [2]. Technologies are moving from static pollution sensing networks to now hyper-local and distributed sensing networks [3]. To accomplish this sort of a task, it often demands mounting sensing units on individual transportation systems and connecting those units to a reliable network. This enables real-time measurement and monitoring of air-pollution across the densely populated roadways which effectively increase the accuracy and effectiveness of the air pollution measurement. This not only enables us to achieve a more sensible view of how much real pollution exists today but also allows us to take strong steps before the climate change turns into an irreversible process.

Introduction

This report focusses on an embedded system design project which is part of an effort to design a prototype for a system that would address the above challenges. The prototype does not account for measuring real pollution data but is instead built to demonstrate some efficient data communication mechanism that goes into building an actual system that can accomplish the massive task of mapping pollution data acquired through tens of thousands of vehicles within a particular geographical area. As such, the prototype in this project, is measuring a few parameters from the environment (e.g. temperature, real-time location, magnetic orientation of the object on which the sensing unit is mounted, etc.) and is putting an effort in reliably communicating meaningful information to an end-user using a publish-and-subscribe system (e.g. e-mail communication). It should be noted that while this prototype is not an ideal demonstration of how a hyper-local and distributed air pollution sensing network works, it is focussed to identify the underlying hardware and firmware/software challenges that arise when building such a system that can eventually be termed as an Industrial Internet-of-Things (IoT) product.

System Overview

Aligned with the above discussion, Figure-1 represents the overall concept of the "Integrated Sensor Network" system. The system is designed while considering that it is eventually an Industrial IoT product.

The system includes the follow features:

1. A robust and low-power Microcontroller Unit (i.e. MSP432 in this case)
2. Sensors that provide valuable information (i.e. temperature sensor, magnetometer, GPS module, RTC, etc.)
3. An on-device display unit (i.e. OLED)
4. A network processor unit (i.e. CC3210 BOOST unit for connecting the system to WiFi)
5. Remote requisition system (i.e. In our case, it is the mobile device connected to the WiFi)

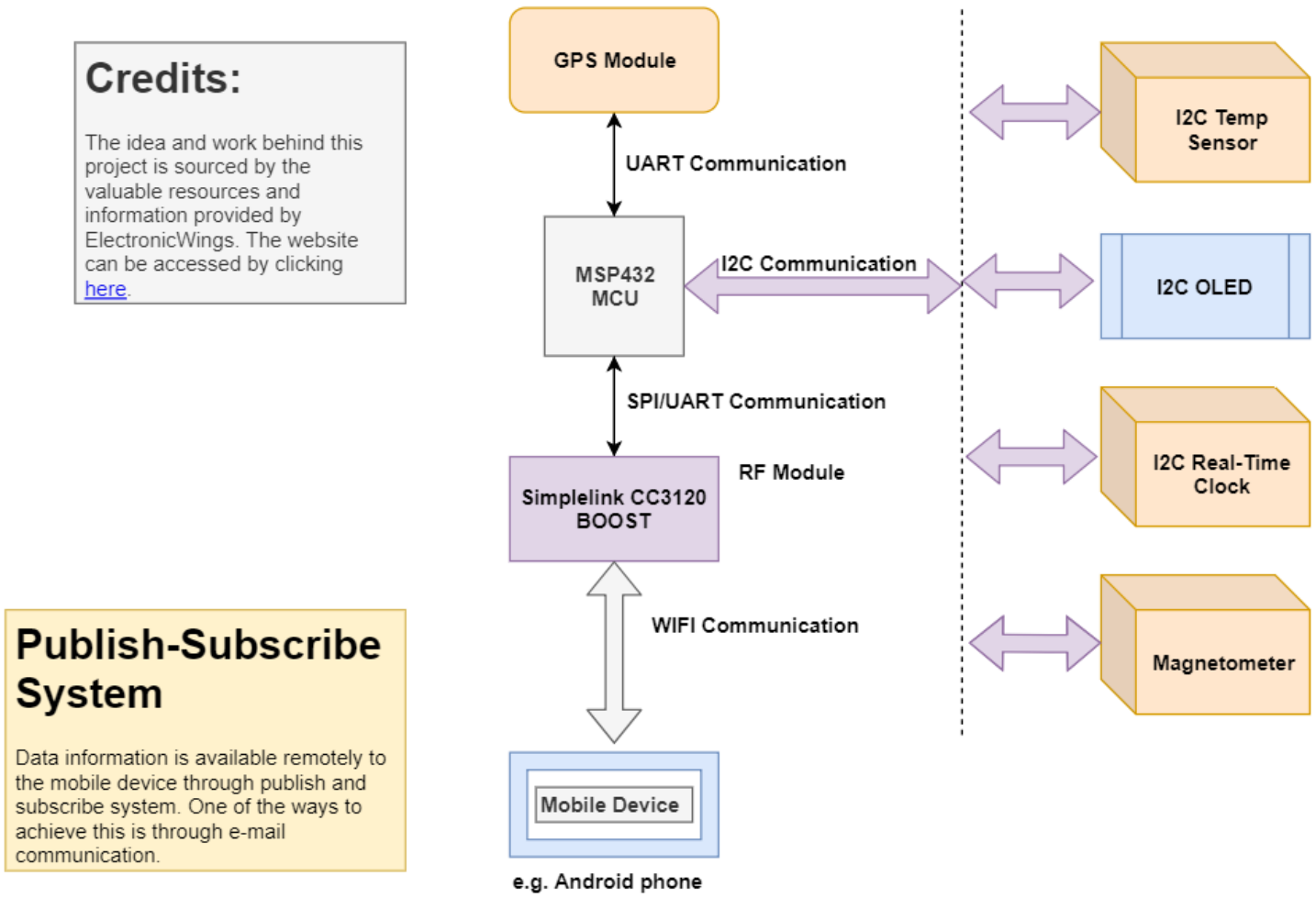


Figure-1: System Concept

Based on the above figure, it is understood that the MSP432 accepts valuable information from the sensors (i.e. temperature sensor, Real-time Clock and Magnetometer) using the I2C communication protocol. While doing so, appropriate sleep routines allows the MCU (MSP432) to sleep as deep and as often as possible. On the other hand, the GPS module provides the real-time location of the system to the MCU using UART communication. Once after a complete data requisition cycle is performed by the MCU for all of the above mentioned sensors, it displays out the data to the OLED using I2C communication. Once that process is served, the MCU attempts to transmit the valuable information to the Mobile Device (which is connected to WiFi) using the Network Processor (Simplelink CC3210 BOOST) which is connected to the WiFi. The Mobile Device has to be authorized to receive the information using the publish-and-subscribe system. The authorization will take place only after an e-mail address is registered with the RF module (Network Processor) that the Mobile Device simultaneously can access. This allows a secure communication between the sensing system and the Mobile Device and thus the prototype serves as an infant model of the actual air pollution measuring system.

Technical Description

Under the topic of technical description, the below sections will provide a detailed overview and explanation of the technical facets of the prototype. The sections outlined are as follows: Hardware Design, Hardware Elements and Device Drivers, Firmware Design and the Application Software.

Hardware (HW) Design

The hardware design of this system essentially involves connecting the MCU with the right pins on the sensors, OLED and the network processor such that successful communication and data exchange can take place within the modules. Figure-2 represents the hardware connections of the modules connected with the MCU.

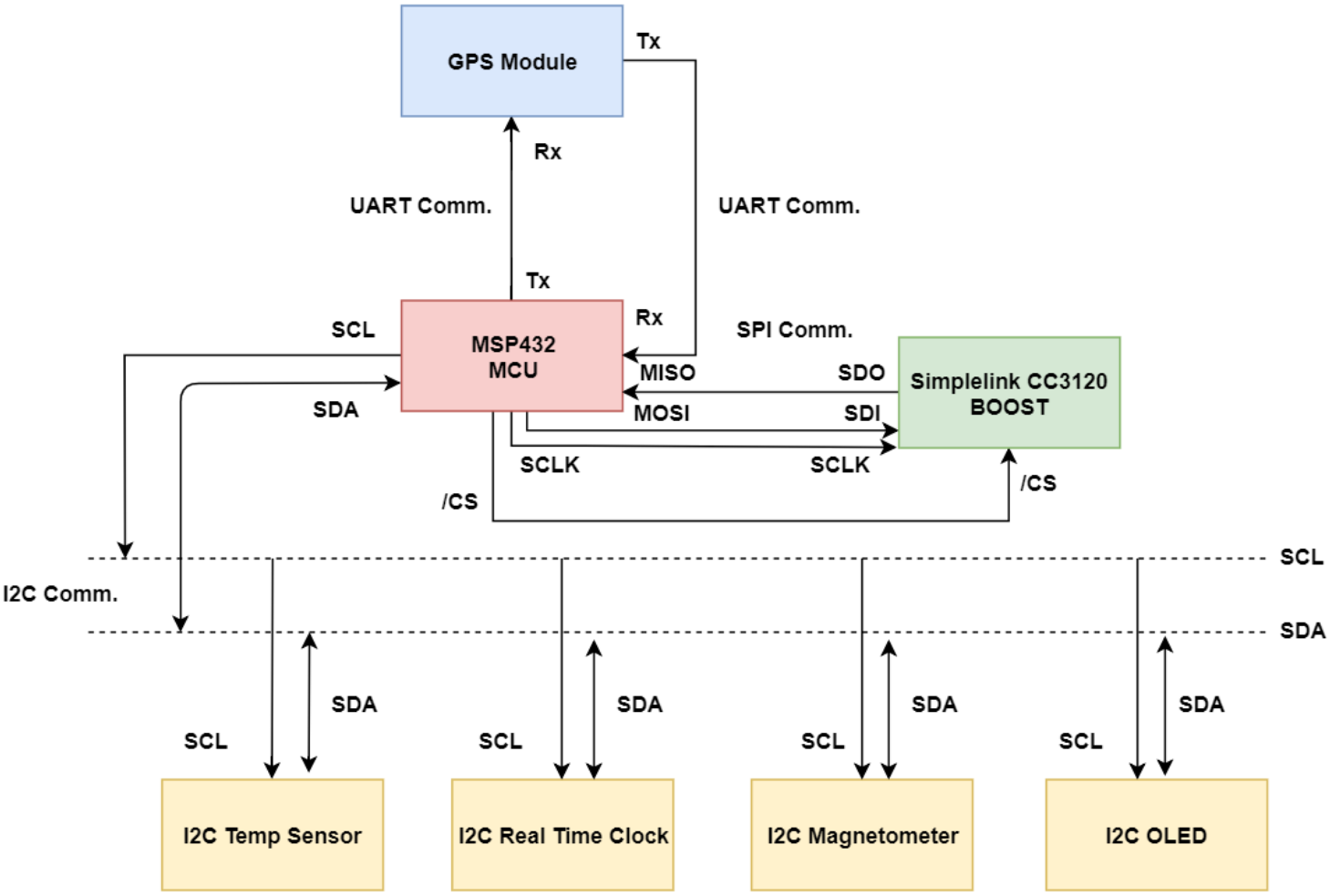


Figure-2: Hardware Layout of the System

On the basis of the above diagram, the following sections will throw more light on the design of the individual modules and how they fit into the system.

Hardware Elements and Device Drivers

In this section, focus will be laid on the general communication architecture of the modules communicating with the MCU so that the discussion can coherently align with the section on firmware design. I2C or UART communication will take place amidst the modules based on the architectural designs of the modules sending data to the MCU. Below are the modules that interact with the MCU:

1. MCP9808 Adafruit Temperature Sensor
2. DIYmall NEO 6M GPS Module
3. HiLetgo Magnetometer
4. DIYmall OLED Module
5. CC3210 BOOST Module – Architecture and set of instructions utilized

To understand the process of the communication between the modules and the MCU, the below section briefly explains how the MCP9808 Adafruit Temperature Sensor will communicate with the MCU. Based on the same, an estimation can be made about other elements that use identical communication mechanisms.

MCP9808 Adafruit Temperature Sensor:

The MCP9808 is a high accuracy digital temperature sensor with a maximum accuracy of +/- 0.5°C. The IC is originally provided by Microchip Technology Inc. and it is further designed in a breakout board format by Adafruit [4]. This temperature sensor converts temperatures between -20°C and +100°C to a digital word with +/- 0.25°C (typical) and +/- 0.5°C (maximum accuracy).

Figure-3 represents the breakout board provided by Adafruit.

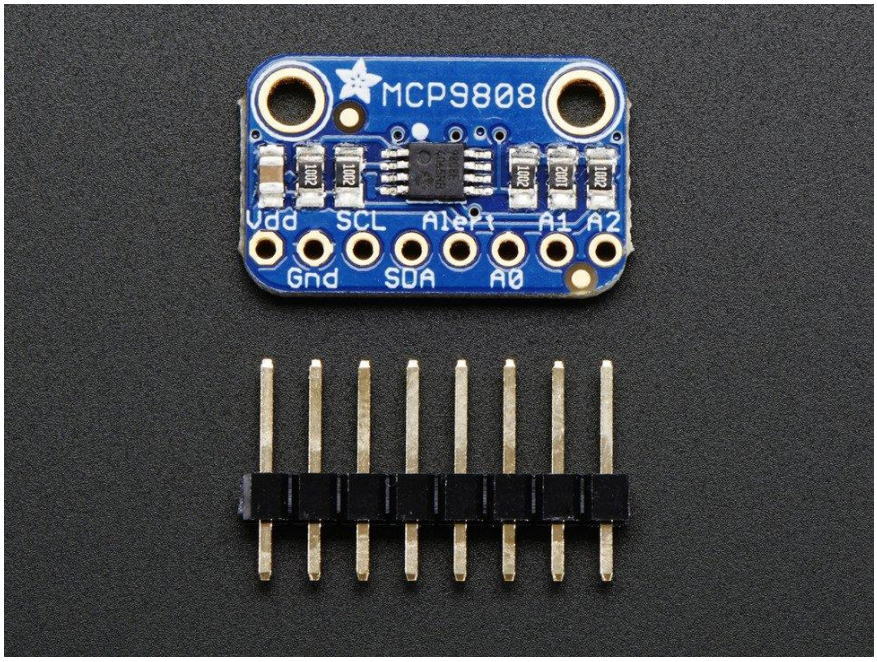
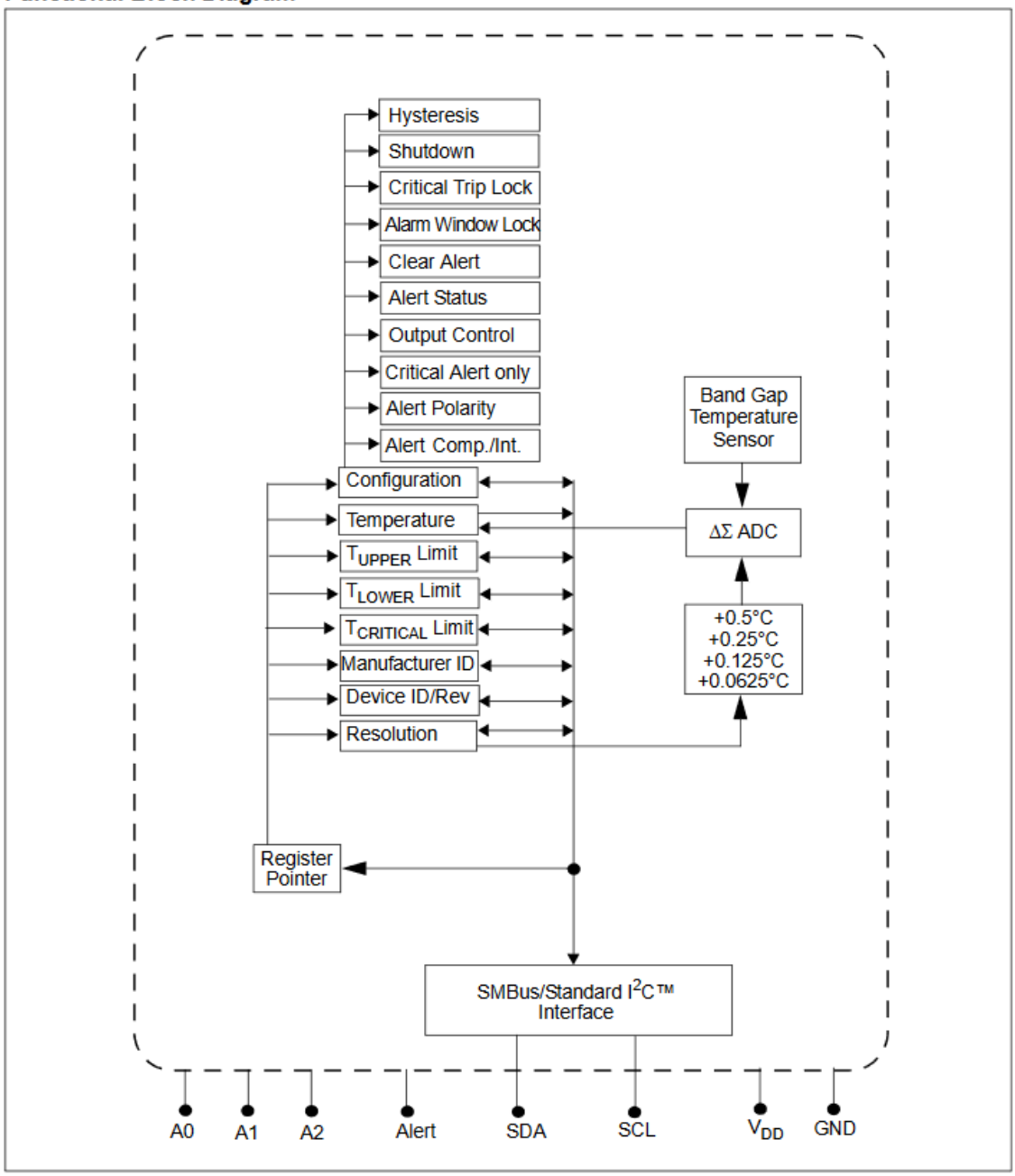


Figure-3: MCP9808 Adafruit Temperature Sensor

Image Source

The MCP9808 chip is powered with the capabilities of user-programmable registers that allow users to flexibly measure temperatures. The registers allow user-selectable settings such as shutdown or low-power modes and the temperature alert window limits produce an alert signal whenever the measured temperature falls outside the range. This sensor has an industry standard 400 KHz, 2-wire I^2C compatible serial bus

Functional Block Diagram



interface which allows up to eight or sixteen sensors to be controlled with a single serial bus. The features allow the MCP9808 to serve as an ideal solution for a robust temperature monitoring application [5].

Figure-4: MCP9808 Block Diagram

Image Source

Figure-4 represents the MCP9808 execution block diagram as provided by Microchip Technology Inc. As per the information provided in the block diagram, up to 8 such devices can be connected on the same I^2C serial bus. This is because A_0, A_1 and A_2 are the three input signals which range from 0-7 (8 combinations) and they govern the fact that 8 such devices can be connected on the same bus.

SDA and SCL are the Serial Data Line and the Serial Clock Line on the I^2C serial bus. These pins are connected to the MCU unit (MSP432 in our case) which will act as a Master and the temperature sensor will act as the Slave on the I^2C serial bus. Further, as can be seen from the block diagram, the user-selectable registers (e.g. T_{UPPER} limit, T_{LOWER} limit and $T_{CRITICAL}$ limit) will decide the execution flow of the temperature sensor. If the temperature that is read from the T_A (Temperature) register meets a certain criteria (e.g. $T_A > T_{UPPER}$ limit), an alert signal may be produced depending on the configurations set by the user for the temperature sensor.

Figure-5 represents the connection of the temperature sensor with the MCU which will depend on the pin function table as provided by Microchip Technology Inc.

DFN	MSOP	Symbol	Pin Function
1	1	SDA	Serial Data Line
2	2	SCL	Serial Clock Line
3	3	Alert	Temperature Alert Output
4	4	GND	Ground
5	5	A2	Slave Address
6	6	A1	Slave Address
7	7	A0	Slave Address
8	8	V_{DD}	Power Pin
9	—	EP	Exposed Thermal Pad (EP); must be connected to GND

Device	Address Code				Slave Address		
	A6	A5	A4	A3	A2	A1	A0
MCP9808	0	0	1	1	x(1)	x	x

Figure-5: MCP9808 Pin Function Table

Image Source

The address code that is used within the sole limits of this project is 0x18 as the address in binary is considered as 0011000 (for A6-A0).

To execute the block diagram flow for this temperature sensor, based on the working principles of the I^2C serial bus interface, to read the temperature from the sensor, the user is first expected to send a start bit and then write to the slave device its address (0x18). After the slave provides an acknowledgement, the user then writes an address (0x05) to the register pointer which will instruct the sensor to decide where a further data read/write has to be performed. After a successful address reception for the register pointer, the slave device (sensor) will send an acknowledgment. Further, the user is expected again to send a start bit once again and the address (0x18) with a read signal to the slave device. The slave acknowledges and sends two bytes of temperature data from the T_A register which is normally C1H (MSB) and 90H (LSB) at a temperature of 25.0°C.

Figure-6, showcases the overall I^2C communication that shall take place with the Temperature Sensor based on the timing diagram of the communication which aligns with the above explanation.

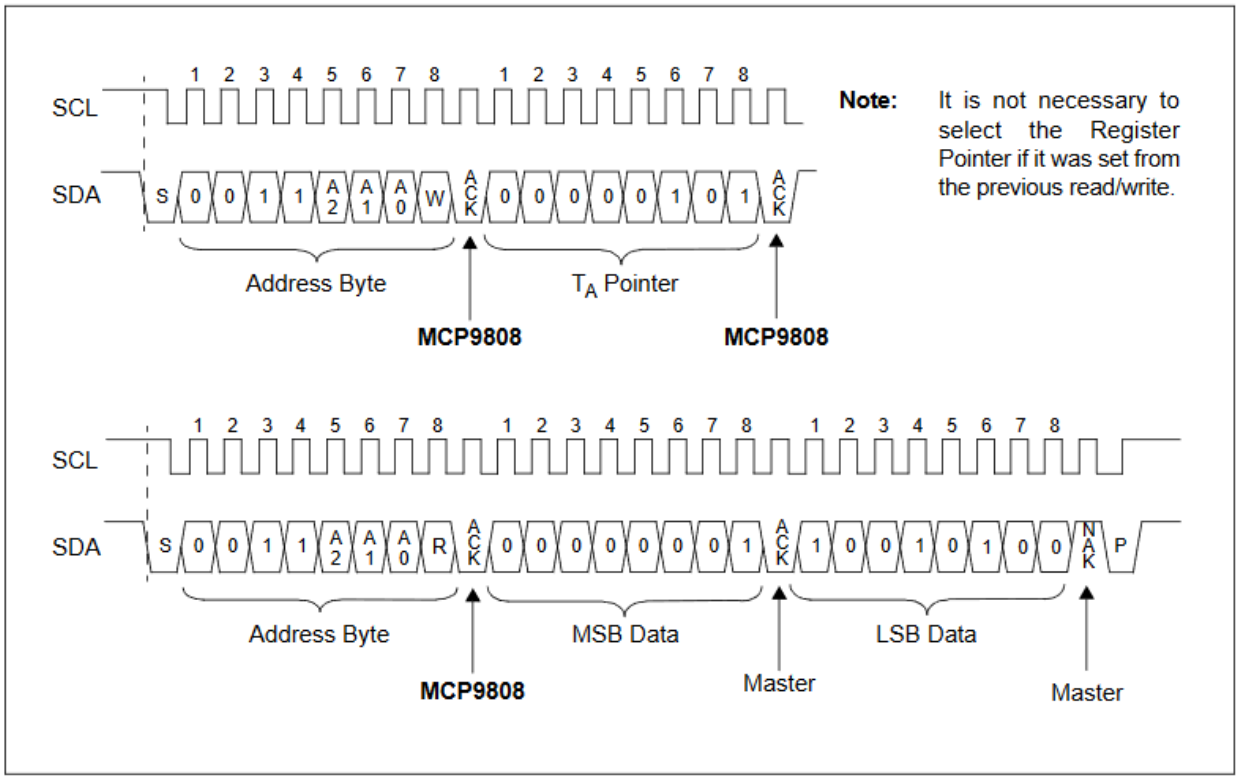


Figure-6: MCP9808 Timing Diagram for reading temperature value of +25.25°C

Image Source

Figure-7 represents the register pointer to which the user initially writes a valid address between 0x00H to 0x08H. This can be clearly understood from the below figure.

REGISTER 5-1: REGISTER POINTER (WRITE-ONLY)

W-0	W-0	W-0	W-0	W-0	W-0	W-0	W-0
—	—	—	—	Pointer bits			
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7-4 **W:** Writable bits
Write '0'.
Bits 7-4 must always be cleared or written to '0'. This device has additional registers that are reserved for test and calibration. If these registers are accessed, the device may not perform according to the specification.

bit 3-0 **Pointer bits**
0000 = RFU, Reserved for Future Use (Read-Only register)
0001 = Configuration register (CONFIG)
0010 = Alert Temperature Upper Boundary Trip register (T_{UPPER})
0011 = Alert Temperature Lower Boundary Trip register (T_{LOWER})
0100 = Critical Temperature Trip register (T_{CRIT})
0101 = Temperature register (T_A)
0110 = Manufacturer ID register
0111 = Device ID/Revision register
1000 = Resolution register
1xxx = Reserved(1)

Note 1: Some registers contain calibration codes and should not be accessed.

Figure-7: MCP9808 Register Pointer

Image Source

When the user writes a value of 0x05 into the register pointer at the beginning of the execution flow, the T_A temperature register is selected which sends in two bytes of data when read from the same slave device. The same can be more clearly understood from figure-8.

REGISTER 5-4: T_A: AMBIENT TEMPERATURE REGISTER (→ ADDRESS '0000 0101'b)(1)

R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
T_A vs. T_{CRIT}(1)	T_A vs. T_{UPPER}(1)	T_A vs. T_{LOWER}(1)	SIGN	2^7 °C	2^6 °C	2^5 °C	2^4 °C
bit 15							bit 8

R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
2^3 °C	2^2 °C	2^1 °C	2^0 °C	2^{-1} °C	2^{-2} °C(2)	2^{-3} °C(2)	2^{-4} °C(2)
bit 7							bit 0

bit 15 **T_A vs. T_{CRIT} bit(1)**
0 = $T_A < T_{CRIT}$
1 = $T_A \geq T_{CRIT}$

bit 14 **T_A vs. T_{UPPER} bit(1)**
0 = $T_A \leq T_{UPPER}$
1 = $T_A > T_{UPPER}$

bit 13 **T_A vs. T_{LOWER} bit(1)**
0 = $T_A \geq T_{LOWER}$
1 = $T_A < T_{LOWER}$

bit 12 **SIGN bit**
0 = $T_A \geq 0°C$
1 = $T_A < 0°C$

bit 11-0 **T_A: Ambient Temperature bits(2)**
12-bit ambient temperature data in two's complement format.

Figure-8: MCP9808 T_A Temperature Register

Image Source

To read the temperature from the T_A temperature register, the MSB is first logic ANDed with 0x10 so that the bit12 (SIGN bit) is obtained. If the result is 0x10 then the temperature is below 0°C and is negative. If the result is 0x00 then the temperature is greater than or equal to 0°C.

Further, if the temperature is negative, the original MSB data is logic ANDed with 0x0F (to receive the bits 11-8) and then fed into the below equation to calculate the temperature in terms of negative degree Celsius and degree Fahrenheit:

```
Temperature (°C)       = 256 - (UpperByte x 16 + LowerByte / 16)          [negative value]

Temperature (°F)       = 32  - ((9.0/5.0)*(Temperature (°C)))             [sign depends on °C]
```

If the temperature was greater than or equal to 0°C, the original MSB data is logic ANDed with 0x0F (to receive the bits 11-8) and then fed into the below equation to calculate the temperature in terms of positive degree Celsius:

```
Temperature (°C)       = (UpperByte x 16 + LowerByte / 16)                [positive value]

Temperature (°F)       = 32  + ((9.0/5.0)*(Temperature (°C)))             [positive value]
```

For an example calculation, if the data bytes read from the slave device is 0xC1H (MSB) and 0x90H (LSB), the upper byte (MSB) is first ANDed with 0x10H and the result is 0x00H which means the temperature is above freezing point of water. Now, the original upper byte (0xC1H) is ANDed with 0x0F and the result is 0x01H. This upper byte value along with the lower byte value (0x90H) is fed to the above equation and the result will be:

```
Temperature (°C)       = ((1)16 x 16 + (90)16 / 16)
                       = ((1)  x 16 + 144 / 16)
                       = (16 + 9)
                       = 25°C

Temperature (°F)       = 32 + ((9.0/5.0)*25.0)
                       = 32 + 45
                       = 77°F
```

This is how the entire process of temperature calculation will take place using the MCP9808 I^2C enabled Temperature Sensor.

Firmware (FW) Design

In this section, an overview of the firmware design has been presented. It includes an introduction to all the elements that have gone into building the source code for the system. This section will discuss the following elements in detail:

1. Hardware Dependent Firmware Layer (HDFW)
2. Hardware Independent Firmware Layer (HIFW)
3. Device Drivers (DDFW)
4. Application Layer (ALFW)

The HDFW is the most crucial segment of the entire FW design of the project. This is because the HDFW decides – when, how and what services the device can support. Although the hardware independent firmware layer is often able to overcome the limitations of the hardware dependent firmware layer through some workarounds, it is mostly crucial to design a robust hardware dependent firmware layer to avoid any execution problems with the device.

The HIFW on the other side, is crucial for integrating several components with the MCU. The HIFW layer mainly focusses on the communication protocols and standards that are incorporated for the system. It mainly includes serial interface standards like UART, I^2C, SPI and RS-232.

The DDFW is the high-level FW layer that brings up the components to life that are interfaced with the MCU. This layer guarantees that a reliable, robust and error-corrected communication exists between the MCU and the devices interfaced with the MCU. As an example, the temperature sensor communicates using the I^2C serial bus interface with the MCU and this layer guarantees that the HIFW protocols and standards meet that of the sensor which have been established in its datasheet.

Finally, the ALFW layer is the top-most layer that directly communicates with the user of the end product, service or system that is under attention of this project. As an example, this layer ensures that a reliable temperature measurement is performed and is displayed on the terminal emulator.

Figure-9 presents a brief overview of the entire FW design of this project. The overview attempts to include most aspects of the Firmware associated with this project however, the figure is not exhaustive.

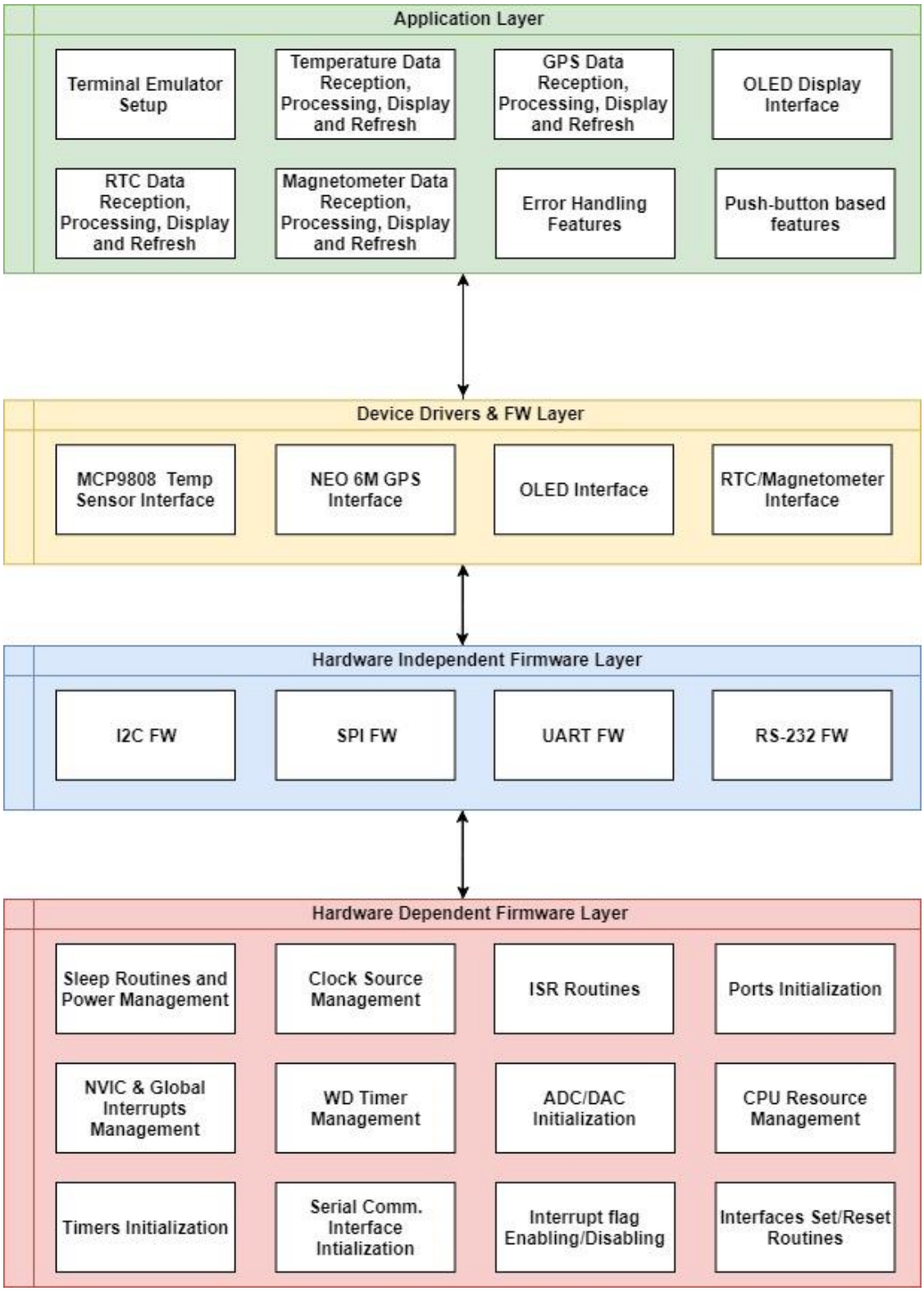


Figure-9: Firmware Design of the Project

Hardware Dependent Firmware Layer (HDFW):

Extending the introduction provided to the FW design of this project, the HDFW layer will mostly involve some or all of the blocks represented in Figure-9. A typical example of HDFW is the ports initialization.

```
void pin_set(void)
{
    // Configure UART pins
    P1->SEL0 |= BIT2 | BIT3 | BIT6 | BIT7;                // set 2-UART pin as
secondary function
    P1->SEL1 = 0;
    P1->DIR |= BIT0;
    P1->OUT |= BIT0;

    P2->DIR |= BIT1; // Setting test LEDs
    P2->OUT |= BIT1; // Setting test LEDs

    // Setting all other ports to a default value
    P3->DIR |= 0xFF; P3->OUT = 0;
    P4->DIR |= 0xFF; P4->OUT = 0;
    P5->DIR |= 0xFF; P5->OUT = 0;
    P6->DIR |= 0xFF; P6->OUT = 0;
    P7->DIR |= 0xFF; P7->OUT = 0;
    P8->DIR |= 0xFF; P8->OUT = 0;
    P9->DIR |= 0xFF; P9->OUT = 0;
    P10->DIR |= 0xFF; P10->OUT = 0;

    // Port-J assignment
    PJ->DIR |= (BIT0| BIT1 | BIT2 | BIT3);
    PJ->OUT &= ~(BIT0 | BIT1 | BIT2 | BIT3);
}
```

Excerpt from the Source Code – Ports Initialization

Here, the pins are set based on the underlying hardware design of the MSP432 MCU. More can be understood from Figure-10 which highlights some of the pin configurations that MSP432 reflects.

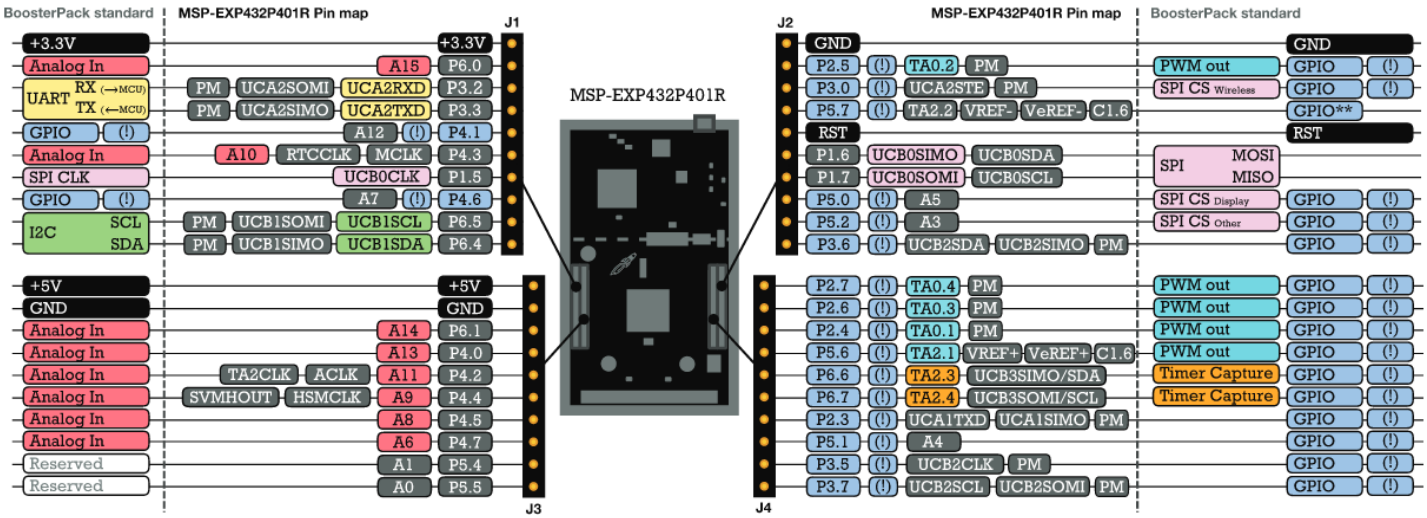


Figure-10: MSP432 Pin Layout

Image Source

More about the HDFW can be understood by looking at the source code and especially the "init.c" file added to the APPENDIX on Source Codes.

Hardware Independent Firmware Layer (HIFW):

Under this segment, some of the source code blocks that are related includes the UART configuration. An excerpt from the Source Code is provided below:

```
void uart_set(void)
{
    CS->KEY = CS_KEY_VAL;                   // Unlock CS module for register access
    CS->CTL0 = 0;                           // Reset tuning parameters
    CS->CTL0 = CS_CTL0_DCORSEL_3;           // Set DCO to 12MHz (nominal, center of
8-16MHz range)
    CS->CTL1 = CS_CTL1_SELA_2 |             // Select ACLK = REFO
            CS_CTL1_SELS_3 |                // SMCLK = DCO
            CS_CTL1_SELM_3;                 // MCLK = DCO
    CS->KEY = 0;                            // Lock CS module from unintended
accesses


    // Configure UART
    EUSCI_A0->CTLW0 |= EUSCI_A_CTLW0_SWRST; // Put eUSCI in reset
    EUSCI_A0->CTLW0 = EUSCI_A_CTLW0_SWRST | // Remain eUSCI in reset
    EUSCI_B_CTLW0_SSEL__SMCLK;      // Configure eUSCI clock source for SMCLK

    // Baud Rate calculation
    // 12000000/(16*9600) = 78.125
    // Fractional portion = 0.125
    // User's Guide Table 21-4: UCBRSx = 0x10
```

```
    // UCBRFx = int ( (78.125-78)*16) = 2

    EUSCI_A0->BRW = 78;

    /* 12000000/16/9600 = 78 (approx.)
     * supports efficient Baud rates between 250,000 (value 3) to 9600 (value 78)
     */

    EUSCI_A0->MCTLW = (2 << EUSCI_A_MCTLW_BRF_OFS) |
            EUSCI_A_MCTLW_OS16;

    EUSCI_A0->CTLW0 &= ~EUSCI_A_CTLW0_SWRST; // Initialize eUSCI
    EUSCI_A0->IFG &= ~EUSCI_A_IFG_RXIFG;    // Clear eUSCI RX interrupt flag
    EUSCI_A0->IE |= EUSCI_A_IE_RXIE;        // Enable USCI_A0 RX interrupt
}
```

Excerpt from the Source Code – UART Setup

In this code, it can be seen that the UART is configured, it is set to operate at 9600 baud rate and the interrupts are enabled for transmission and reception of information. Appropriate clock sources are utilized and set to achieve the required results.

Device Drivers (DDFW):

Under this segment, robust code is written so that every element (e.g. MCP9808 Temperature Sensor) on or off the MCU is able to effectively communicate and meaningful data can be received by the MCU. As an example, an excerpt from the source code is provided for the device driver of the temperature sensor which allows the MCU to continuously receive temperature information from the sensor.

```
1  i2c_temp_sensor_init();
2  i2c_refresh();
3  i2c_temp_meas();
```

Excerpt from the Source Code – Temperature Sensor Device Driver

The first function initializes the temperature sensor by setting its slave address, the baud rate of communication, the information that is to be received, so on and so forth. The second function attempts to refresh the communication so that new data can be received or transmitted as per the need. The third function (which is a part of the interrupt service routine) guarantees that whenever the I^2C communication is refreshed, the temperature measurement takes place by following the routines mentioned under the section on Hardware Design and Device Drivers for the temperature sensor. The communication thus stores valuable information into the MCU that which can be used in the application layer for meaningful reflection of the system to the end-user.

Application Layer (ALFW:

Under this segment, an excerpt from the source code has been presented which receives the temperature measurement information and displays it to the terminal emulator window. This sort of a firmware piece lies under the application layer which is directly related with how the user will interact with the system.

```
void print_temp(void)
{
    if(EUSCI_A0->RXBUF == 't')
    {
        temp_meas();
        print_tempC();
        delay();
        print_tempF();
        new_par();
    }
}
```

Excerpt from the Source Code – Temperature Display on the Terminal Emulator

Under this code section, the temperature measurement takes place in which meaningful temperature conversions are made based on the data bytes received from the temperature sensor through the I^2C serial bus. The temperature then is reflected on the terminal emulator and appropriate spacing is added using functions like "new_par();" which adds some space between each printout of the temperature on the screen.

Application Software

In this section, an overview of the application software has been presented. The section attempts to explain how a piece of information received by the MCU from a particular device interfaced with the MCU using its device driver is acquired, processed, displayed to the user and refreshed every time the MCU receives new information which may or may not be based on the user's request to provide information. As can be understood from the firmware design section, the application software revolves around the following elements which have a notion to communicating directly with the user:

- Temperature Measurement
- GPS Location Measurement
- Magnetometer Measurements
- OLED Display Interface
- Publish-and-Subscribe System Set-up

For understanding how the application software works, an example is provided for the Temperature Measurement which involves four basic steps:

1. Temperature Data Acquisition using I^2C bus driver for the sensor
2. Temperature Data Processing
3. Temperature Display
4. Temperature Refreshing – which all the way repeats steps 1 through 3

An excerpt from the source code has been provided which will explain how these four steps will be met by the MCU.

1. Temperature Data Acquisition

During this step, the temperature data is acquired which can be understood from the below code:

```
void EUSCIB0_IRQHandler(void)
{
    i2c_temp_meas();

    // Wake up on exit from ISR
    SCB->SCR &= ~SCB_SCR_SLEEPONEXIT_Msk;

    // Ensures SLEEPONEXIT takes effect immediately
    __DSB();
}
```

Excerpt from the Source Code – Temperature Data Acquisition

The above excerpt from the source code shows how when an I^2C interrupt occurs, the temperature measurement takes place which is essentially an acquisition of the bytes coming from the temperature sensor over the bus.

2. Temperature Data Processing

During this step, the bytes collected in the above step from the temperature sensor using its device driver from the DDFW layer is processed to form two values: `tempC (Temperature in °C)` and `tempF (Temperature in °F)`. The values are calculated based on the below excerpt from the source code:

```
void temp_meas(void)
{
    MSB = RXData[0];
    LSB = RXData[1];

    MSB = MSB & 0x1F;

    if((MSB & 0x10) ==  0x10)
    {
        s_flag = 1;
        MSB = MSB & 0x0F;
        tempC = (uint8_t) (256.0 - ((MSB*16.0) + (LSB/16.0)));
        tempF = (uint8_t) (32.0 - ((9.0/5.0)*tempC));
    }
    else
    {
        s_flag = 0;
        tempC = (uint8_t) ((MSB*16.0) + (LSB/16.0));
        tempF = (uint8_t) (32.0 + ((9.0/5.0)*tempC));
    }
}
```

Excerpt from the Source Code – Temperature Data Processing

3. Temperature Display

During this step, the temperature information recorded in the previous step is sent out to the terminal emulator. The excerpt from the source code for printing out the temperature in °C is given below:

```
void print_tempC(void)
{
    char text[14] = "Temperature: "; // Text to notify Temperature output
    char temp_print[5] = "XXXC";           // Text to provide Temperature
    int temp_out, text_out;

    //Arranging output into a proper line

    new_line();
    delay();
    car_return();


    // Output of the main text
    for(text_out=0; text_out<=12; text_out++)
    {
        // Main printing part

        delay();
        EUSCI_A0->TXBUF = text[text_out];
        delay();

        // Main printing part ends
```

```
    }

    // Setting array for printing temperature

    if(tempC <= 9)
    {
        temp_print[1] = '0';
        temp_print[2] = (uint8_t) tempC + '0';
    }
    if(tempC >= 10 && tempC <= 99)
    {
        temp_print[1] = ((uint8_t)(tempC/10.0f)) + '0';
        temp_print[2] = (uint8_t) (tempC - ((float) ((uint8_t)(tempC/10.0f)) *
10.0f)) + '0';
    }
    if(s_flag != 0)
    {
        temp_print[0] = '-';
    }
    if(s_flag == 0)
    {
        temp_print[0] = '+';
    }
    else
    {
        temp_print[0] = '*';
        temp_print[1] = '*';
        temp_print[2] = '*';
    }

    // Output of the temperature reading
    for(temp_out=0; temp_out<=3; temp_out++)
    {
        // Main printing part

        delay();
        EUSCI_A0->TXBUF = temp_print[temp_out];
        delay();

        // Main printing part ends
    }
}
```

Excerpt from the Source Code – Temperature Display

4. Temperature Refreshing

This is the last step in the application software part for the temperature sensor. This is basically a repetition of step 1 through 3.

Results & Error Analysis

I did not succeed at integrating all the components together for this project and partly this has been due to some of the below reasons:

1. Tightly coupled firmware development
2. Device drivers interrupting each other's execution
3. Unable to identify the glitches in the system when several elements are talking to the MCU at the same time

To overcome some of the above challenges, I used #define statements wherever necessary during the testing purpose and tested tiny fragments of the source code. Once I was able to make a significant segment of the code to run, I tried integrating one more than one element at the same time and that is where things did not work well. I identify that the reason for this would have been both my coding style and the sharing of resources between processes and elements. Unauthorized or incomplete instruction executions will have resulted into a failure to run every piece of the system simultaneously.

Conclusion

This project has indeed allowed me to understand my weaknesses and strengths as an Embedded Systems Engineer. I have been able to understand the underlying challenges while making the elements of any Embedded System to talk with each other. It has made me more aware about the challenges that Real-time Embedded Systems would be subjected. Through careful design of my firmware elements for this project, I have been able to learn how even a simple communication between two devices over a serial bus interface like I^2C can become quite challenging when many components with different configurations and settings are sharing the same bus.

On the other side, this project journey has enabled me to learn and exhaustively use debugging tools like Logic Analyzer and the debugger in an IDE. I have been able to see where my source code has not failed to meet my expectations and my planning. The tools that I have used in this class has prepared me not only for a bright future in the field of Embedded Systems but has also enabled me to think how systems in general can fail to deliver expected outcomes. As a final outcome of this project journey, I have experienced that whenever I come across any electronic system, I immediately begin analysing on what individual elements would have gone into the makings and working of that system. It is no longer a mystery but a curious and exciting life ahead in the field of Embedded Systems Engineering that has just begun very brightly with the end of this project experience.

Future Motivation & Ideas

I was able to succeed at whatever I could for this project because I used the MSP432 development board. It has been a robust tool and partly why I could meet my goals was because of the well-designed hardware and firmware for the board. This MCU which has a 32-bit Cortex M4F CPU with advanced low-power features and memory capabilities, has made it imaginable to communicate with many devices in real-time.

In the future, I definitely look forward to work on other Embedded Systems Engineering projects that utilize complex firmware elements and incorporate the use of sensors to communicate with the outside world. I am eager to work on connecting my upcoming projects to cloud using reliable wide-area communication technologies so that the system can provide meaningful information to a large amount of users.

Acknowledgements

At the end of my journey in the ECEN 5613 Embedded System Design course, I would sincerely like to thank Prof. McClure for his immense support in delivering unparalleled and highly industry-seasoned knowledge on designing any Embedded System that will work as a backbone for taking other ESE courses.

I would also like to thank both the Teaching Assistants – Tristan Lennertz and Kiran Hegde who have continuously helped me with my queries and concerns over the semester. This would not have been possible without their help.

Last but not the least, I would like to thank and be grateful to all the authors of the various resources that I have cited and acknowledged for making all this possible.

References

Citations

[1]	AI and Big Data vs Air Pollution
[2]	Air Pollution from Vehicles
[3]	Aclima air quality measurement and mapping project with Google
[4]	Adafruit MCP9808 Precision I2C Temperature Sensor
[5]	Microchip Technology Inc. MCP9808

Appendices

APPENDIX – BILL OF MATERIALS

Part Desciption	Source	Cost ($)
Mini Breadboard	Amazon	5.69
Jumper Wires	Amazon	5.78
Adafruit MCP9808 Temp Sensor	Amazon	8.56
DIYmall 6M GPS Module	Amazon	16.5
DIYmall 0.96 Inch Yellow and Blue 128x64 IIC OLED	Amazon	8.99
HiLetgo MPU9250/6500 9-axis Attitude+Accelerometer+ Gyro+Magnetometer	Amazon	8.49
Gikfun RTC	Amazon	4.68
Total		**58.69**

I will be returning some of the components back to Amazon worth around 25-30$.

APPENDIX – SCHEMATICS

Please, refer to Figure-2 for the hardware layout of the system.

APPENDIX – SOURCE CODES

All the source codes have been attested under this section beginning next page.

main.c – source code

```
/* --COPYRIGHT--,BSD_EX
 * Copyright (c) 2014, Texas Instruments Incorporated
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * *  Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * *  Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * *  Neither the name of Texas Instruments Incorporated nor the names of
 *    its contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
 * OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * *  --/COPYRIGHT--*/
/******************************************************************************
 * This code has been created solely by Rushi James Macwan.
 *
 * Various resources available on the MSP-432 Open Source platform have been
 * used for creating this code. This code has been built essentially using
 * the MSP432 Simplelink SDK containing several examples. The same can be
 * accessed by visiting the below given link:
 *
 * Link: http://www.ti.com/general/docs/lit/getliterature.tsp?
 * baseLiteratureNumber=slac698&fileType=zip
 *
 * To summarize explicitly, all the resource examples used for creating this
 * entire project are as under which can be found by accessing the above link:
 *
 * msp432p401x_euscib0_i2c_master_multislave
 * msp432p401x_euscib0_i2c_10
 *
 * This work has been created with due respect for the copyrights of TI and all
 * the authors and contributors at TI who have created the above example codes
```

```
 * which are available at the link provided in this document.
 *
 * I duly acknowledge the original authors for their work which I have modified
 * as per requirement. Credits and Courtesy to the free Open Source platform
 * offered by Texas Instruments and all the individual authors who have
 * provided the example codes.
*******************************************************************************/

/******************   ---INTRODUCTION---    ****************************
 *
 * On pressing the 't' key and running the terminal emulator software at
 * 9600 Baud rate, temperature will be printed out.
 *******************************************************************************/
//******************************************************************************
//  Credits to original author:
//   William Goh (Original Code Resource)
//   Texas Instruments Inc.
//   June 2016 (updated) | June 2014 (created)
//   Built with CCSv6.1, IAR, Keil, GCC
//******************************************************************************


#include "ti/devices/msp432p4xx/inc/msp.h"
#include "Header Files/main.h"
#include <stdint.h>

//////////////////////////////////////////////////////
//TX PART Global Variables
//////////////////////////////////////////////////////
uint8_t TXData[]= {0x05};
uint8_t SlaveAddress[]= {0x18};
uint8_t TXByteCtr;
uint8_t SlaveFlag = 0;

//////////////////////////////////////////////////////
//RX PART Global Variables
//////////////////////////////////////////////////////
uint8_t RXData[2] = {0};
uint8_t RXDataPointer;

//////////////////////////////////////////////////////
//Global Variables
//////////////////////////////////////////////////////
volatile uint32_t i;

//////////////////////////////////////////////////////
//Global Variables - For Printing to Terminal Emulator
//////////////////////////////////////////////////////
uint8_t MSB, LSB, s_flag; // s_flag is the sign-flag
uint8_t tempC, tempF;

//////////////////////////////////////////////////////
//Main Function
```

```
//////////////////////////////////////////////////////

int main(void)
{
    msp_init(); //initializing the MCU

    // Initialize data variable
    RXDataPointer = 0;  //Setting I2C variables

    msp_IRQ_enable();   //Enabling IRQ Services

    i2c_temp_sensor_init(); //Initializing the I2C MCP9808 Temp Sensor

    // Wake up on exit from ISR
    SCB->SCR &= ~SCB_SCR_SLEEPONEXIT_Msk;

    // Ensures SLEEPONEXIT occurs immediately
    __DSB();

    while(1)
    {
        i2c_temp_sensor_init(); //Setting the I2C MCP9808 Temp Sensor
                                // for continuous reception of temp data
        i2c_refresh();

        // Enter LPM0
        __sleep();
        __no_operation();
    }
}

// I2C interrupt service routine
void EUSCIB0_IRQHandler(void)
{
    i2c_temp_meas();            // Measures the temp data

    // Wake up on exit from ISR
    SCB->SCR &= ~SCB_SCR_SLEEPONEXIT_Msk;

    // Ensures SLEEPONEXIT takes effect immediately
    __DSB();
}

void EUSCIA0_IRQHandler(void)
{
    if (EUSCI_A0->IFG & EUSCI_A_IFG_RXIFG)
    {
        // Check if the TX buffer is empty first
       while(!(EUSCI_A0->IFG & EUSCI_A_IFG_TXIFG));
       print_temp();        // Prints the temp data
    }
}
```

main.h – header file

```
#include "ti/devices/msp432p4xx/inc/msp.h"

//////////////////////////////////////////////////////
//Shared global variables across source files
//////////////////////////////////////////////////////
extern volatile uint32_t i;
extern uint8_t mode;

//////////////////////////////////////////////////////
//Shared TX PART Global Variables
//////////////////////////////////////////////////////
extern uint8_t TXData[];
extern uint8_t SlaveAddress[];
extern uint8_t TXByteCtr;
extern uint8_t SlaveFlag;


//////////////////////////////////////////////////////
//Shared RX PART Global Variables
//////////////////////////////////////////////////////
extern uint8_t RXData[2];
extern uint8_t RXDataPointer;

//////////////////////////////////////////////////////
//Global Variables - For Printing to Terminal Emulator
//////////////////////////////////////////////////////
extern uint8_t MSB, LSB, s_flag; // s_flag is the sign-flag
extern uint8_t tempC, tempF;

//////////////////////////////////////////////////////
//Function Prototypes
//////////////////////////////////////////////////////

//Init functions - init.c file
void msp_init(void);        // main msp initialization function
void msp_IRQ_enable(void);  // main msp irq enabling function
void msp_stp_wtd(void);     // watchdog timer function
void pin_set(void);         // pin initialization function
void uart_set(void);        // uart configuration function

//I2C Temperature Sensor functions - i2c_temp.c file
void i2c_temp_sensor_init(void);    //initializating the temp sensor
void i2c_temp_meas(void);           // measuring the temp bytes
void temp_meas(void);               // converting temp to C and F

//I2C functions - i2c_driver.c file
void msp_i2c_reset(uint8_t brate, uint8_t rxbytes, uint8_t slave_address);  //I2C
reset function
void i2c_init(void);                //I2C init function
void i2c_interrupt_enable(void);    //I2C interrupt enable function
```

```
void i2c_refresh();                  //I2C refresh function
void i2c_delay(void);                //I2C delay function
void i2c_slave_select(uint8_t slave_select);    //I2C slave select function
void i2c_reset(void);
void i2c_tx_bytes(uint8_t bytes);   //Setting no bytes of transmission
void i2c_nack_reset(void);          //Setting the no acknowledgement function
void i2c_write(uint8_t write_data); //I2C write function
void i2c_tx_add(uint8_t slave_add, uint8_t bytes);  //I2C slave add function
void i2c_start(void);   //I2C start bit function
void i2c_stop(void);    //I2C stop bit function
void i2c_read(void);    //I2C read function
void i2c_brate(uint8_t brate);  //I2C baud rate function
void i2c_rxbytes(uint8_t rxbytes);  //I2C no of reception bytes function

//Terminal Emulator print functions - print.c file
void delay(void);
void print_temp(void);
void print_tempC(void);
void print_tempF(void);
void new_line(void);        //newline function
void car_return(void);      //carriage return function
void new_par(void);         //new paragraph function
```

init.c – source code

```
#include "ti/devices/msp432p4xx/inc/msp.h"
#include "Header Files/main.h"

//////////////////////////////////////////////////////
//Function Definitions
//////////////////////////////////////////////////////

void msp_init(void)
{
    msp_stp_wtd();
    pin_set();
    uart_set();
}

void msp_IRQ_enable(void)
{
    // Enable global interrupt
    __enable_irq();

    // Enable eUSCIB0 interrupt in NVIC module
    NVIC->ISER[0] = 1 << ((EUSCIB0_IRQn) & 31);
    NVIC->ISER[0] = 1 << ((EUSCIA0_IRQn) & 31); // Enable UART interrupt in NVIC
module
}

void msp_stp_wtd(void)
{
    // volatile uint32_t i;
     WDT_A->CTL = WDT_A_CTL_PW |             // Stop watchdog timer
             WDT_A_CTL_HOLD;
}

void pin_set(void)
{
    // Configure UART pins
    P1->SEL0 |= BIT2 | BIT3 | BIT6 | BIT7;                // set 2-UART pin as
secondary function
    P1->SEL1 = 0;
    P1->DIR |= BIT0;
    P1->OUT |= BIT0;

    P2->DIR |= BIT1;
    P2->OUT |= BIT1;

    // Setting all other ports to a default value
    P3->DIR |= 0xFF; P3->OUT = 0;
    P4->DIR |= 0xFF; P4->OUT = 0;
    P5->DIR |= 0xFF; P5->OUT = 0;
    P6->DIR |= 0xFF; P6->OUT = 0;
    P7->DIR |= 0xFF; P7->OUT = 0;
    P8->DIR |= 0xFF; P8->OUT = 0;
```

```
    P9->DIR |= 0xFF; P9->OUT = 0;
    P10->DIR |= 0xFF; P10->OUT = 0;

    // Port-J assignment
    PJ->DIR |= (BIT0| BIT1 | BIT2 | BIT3);
    PJ->OUT &= ~(BIT0 | BIT1 | BIT2 | BIT3);
}

void uart_set(void)
{
    CS->KEY = CS_KEY_VAL;                   // Unlock CS module for register access
    CS->CTL0 = 0;                           // Reset tuning parameters
    CS->CTL0 = CS_CTL0_DCORSEL_3;           // Set DCO to 12MHz (nominal, center of
8-16MHz range)
    CS->CTL1 = CS_CTL1_SELA_2 |             // Select ACLK = REFO
            CS_CTL1_SELS_3 |                // SMCLK = DCO
            CS_CTL1_SELM_3;                 // MCLK = DCO
    CS->KEY = 0;                            // Lock CS module from unintended
accesses


    // Configure UART
    EUSCI_A0->CTLW0 |= EUSCI_A_CTLW0_SWRST; // Put eUSCI in reset
    EUSCI_A0->CTLW0 = EUSCI_A_CTLW0_SWRST | // Remain eUSCI in reset
    EUSCI_B_CTLW0_SSEL__SMCLK;      // Configure eUSCI clock source for SMCLK

    // Baud Rate calculation
    // 12000000/(16*9600) = 78.125
    // Fractional portion = 0.125
    // User's Guide Table 21-4: UCBRSx = 0x10
    // UCBRFx = int ( (78.125-78)*16) = 2

    EUSCI_A0->BRW = 78;

    /* 12000000/16/9600 = 78 (approx.)
     * supports efficient Baud rates between 250,000 (value 3) to 9600 (value 78)
     */

    EUSCI_A0->MCTLW = (2 << EUSCI_A_MCTLW_BRF_OFS) |
            EUSCI_A_MCTLW_OS16;

    EUSCI_A0->CTLW0 &= ~EUSCI_A_CTLW0_SWRST; // Initialize eUSCI
    EUSCI_A0->IFG &= ~EUSCI_A_IFG_RXIFG;    // Clear eUSCI RX interrupt flag
    EUSCI_A0->IE |= EUSCI_A_IE_RXIE;        // Enable USCI_A0 RX interrupt
}
```

i2c_driver.c – source code

```
#include "ti/devices/msp432p4xx/inc/msp.h"
#include "Header Files/main.h"

//////////////////////////////////////////////////////
//Function Definitions
//////////////////////////////////////////////////////

void i2c_delay(void)
{
    // Delay between transmissions
   for (i = 1000; i > 0; i--);
}

void i2c_slave_select(uint8_t slave_select)
{
    // configure slave address
     EUSCI_B0->I2CSA = slave_select;
}

void i2c_reset(void)
{
    // Ensure stop condition got sent
    while (EUSCI_B0->CTLW0 & EUSCI_B_CTLW0_TXSTP);

    EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TR | // I2C TX
            EUSCI_B_CTLW0_TXSTT;         // Start condition
}

void i2c_tx_bytes(uint8_t bytes)
{
    // Load TX byte counter
    TXByteCtr = bytes;
}

void i2c_nack_reset(void)
{
    if (EUSCI_B0->IFG & EUSCI_B_IFG_NACKIFG)
        {
            EUSCI_B0->IFG &= ~EUSCI_B_IFG_NACKIFG;
            // I2C start condition
            EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TXSTT;
        }
}

void i2c_write(uint8_t write_data)
{
    if (EUSCI_B0->IFG & EUSCI_B_IFG_TXIFG0)
    {
        EUSCI_B0->IFG &= ~EUSCI_B_IFG_TXIFG0;

        // Check TX byte counter
```

```
        if (TXByteCtr)
        {
            // Load TX buffer
            EUSCI_B0->TXBUF = write_data;

            // Decrement TX byte counter
            TXByteCtr--;
        }
        else
        {
            // I2C stop condition
            EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TXSTP;

            // Clear USCI_B0 TX int flag
            EUSCI_B0->IFG &= ~EUSCI_B_IFG_TXIFG;
        }
    }
}

void i2c_tx_add(uint8_t slave_add, uint8_t bytes)
{
    i2c_delay();
    i2c_slave_select(slave_add);
    i2c_tx_bytes(bytes);
    i2c_reset();
}

void i2c_start(void)
{
    // I2C start condition
    EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TXSTT;
}

void i2c_stop(void)
{
    // I2C start condition
    EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TXSTP;
}

void i2c_read(void)
{
    if (EUSCI_B0->IFG & EUSCI_B_IFG_RXIFG0)
    {
        EUSCI_B0->IFG &= ~ EUSCI_B_IFG_RXIFG0;

        // Get RX data
        RXData[RXDataPointer++] = EUSCI_B0->RXBUF;

        if (RXDataPointer > sizeof(RXData))
        {
            RXDataPointer = 0;
        }
    }
```

```
    if (EUSCI_B0->IFG & EUSCI_B_IFG_BCNTIFG)
    {
        EUSCI_B0->IFG &= ~ EUSCI_B_IFG_BCNTIFG;
    }
}

void i2c_brate(uint8_t brate)
{
    EUSCI_B0->BRW = brate;                      // baudrate = SMCLK /30
}

void i2c_rxbytes(uint8_t rxbytes)
{
    EUSCI_B0->TBCNT = rxbytes;                // number of bytes to be received
}

void msp_i2c_reset(uint8_t brate, uint8_t rxbytes, uint8_t slave_address)
{
    i2c_init();

    i2c_brate(brate);

    i2c_rxbytes(rxbytes);

    i2c_slave_select(slave_address);

    i2c_interrupt_enable();

    // Initialize slave flag
    SlaveFlag = 0;
}

void i2c_init(void)
{
    // Configure USCI_B0 for I2C mode
    EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_SWRST; // put eUSCI_B in reset state

    EUSCI_B0->CTLW0 = EUSCI_B_CTLW0_SWRST | // Remain eUSCI_B in reset state
            EUSCI_B_CTLW0_MODE_3 |          // I2C mode
            EUSCI_B_CTLW0_MST |             // I2C master mode
            EUSCI_B_CTLW0_SYNC |            // Sync mode
            EUSCI_B_CTLW0_SSEL__SMCLK;      // SMCLK

    EUSCI_B0->CTLW1 |= EUSCI_B_CTLW1_ASTP_2;// Automatic stop generated
}

void i2c_interrupt_enable(void)
{
    EUSCI_B0->CTLW0 &= ~EUSCI_B_CTLW0_SWRST;// clear reset register

    EUSCI_B0->IE |= EUSCI_B_IE_TXIE0 |      // Enable transmit interrupt
            EUSCI_B_IE_RXIE0 |
            EUSCI_B_IE_BCNTIE |
```

```
            EUSCI_B_IE_NACKIE;              // Enable NACK interrupt
}

void i2c_refresh()
{
    // Arbitrary delay before transmitting the next byte
    for (i = 1000; i > 0; i--);

    // I2C start condition
    EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_TXSTT;
}
```

i2c_temp.c – source code

```
#include "ti/devices/msp432p4xx/inc/msp.h"
#include "Header Files/main.h"

/////////////////////////////////////////////////////
//Function Definitions
/////////////////////////////////////////////////////

void i2c_temp_meas(void)
{
    i2c_nack_reset();
    i2c_write(0x05);
    i2c_read();
}

void i2c_temp_sensor_init(void)
{
    i2c_tx_add(0x18, 1);
    msp_i2c_reset(30, 2, 0x18);
}

void temp_meas(void)
{
    MSB = RXData[0];
    LSB = RXData[1];

    MSB = MSB & 0x1F;

    if((MSB & 0x10) ==  0x10)
    {
        s_flag = 1;
        MSB = MSB & 0x0F;
        tempC = (uint8_t) (256.0 - ((MSB*16.0) + (LSB/16.0)));
        tempF = (uint8_t) (32.0 - ((9.0/5.0)*tempC));
    }
    else
    {
        s_flag = 0;
        tempC = (uint8_t) ((MSB*16.0) + (LSB/16.0));
        tempF = (uint8_t) (32.0 + ((9.0/5.0)*tempC));
    }
}
```

print.c – source code

```
#include "ti/devices/msp432p4xx/inc/msp.h"
#include "Header Files/main.h"

//////////////////////////////////////////////////////
//Function Definitions
//////////////////////////////////////////////////////

void delay(void)
{
    int wait_var;
    for (wait_var=0 ; wait_var <= 750; wait_var++)
    {
        //Do Nothing
    }
}

void print_temp(void)
{
    if(EUSCI_A0->RXBUF == 't')
    {
        temp_meas();
        print_tempC();
        delay();
        print_tempF();
        new_par();
    }
}

void print_tempC(void)
{
    char text[14] = "Temperature: "; // Text to notify Temperature output
    char temp_print[5] = "XXXC";           // Text to provide Temperature
    int temp_out, text_out;

    //Arranging output into a proper line

    new_line();
    delay();
    car_return();


    // Output of the main text
    for(text_out=0; text_out<=12; text_out++)
    {
        // Main printing part

        delay();
        EUSCI_A0->TXBUF = text[text_out];
        delay();

        // Main printing part ends
```

```
    }

    // Setting array for printing temperature

    if(tempC <= 9)
    {
        temp_print[1] = '0';
        temp_print[2] = (uint8_t) tempC + '0';
    }
    if(tempC >= 10 && tempC <= 99)
    {
        temp_print[1] = ((uint8_t)(tempC/10.0f)) + '0';
        temp_print[2] = (uint8_t) (tempC - ((float) ((uint8_t)(tempC/10.0f)) *
10.0f)) + '0';
    }
    if(s_flag != 0)
    {
        temp_print[0] = '-';
    }
    if(s_flag == 0)
    {
        temp_print[0] = '+';
    }
    else
    {
        temp_print[0] = '*';
        temp_print[1] = '*';
        temp_print[2] = '*';
    }

    // Output of the temperature reading
    for(temp_out=0; temp_out<=3; temp_out++)
    {
        // Main printing part

        delay();
        EUSCI_A0->TXBUF = temp_print[temp_out];
        delay();

        // Main printing part ends
    }
}

void print_tempF(void)
{
    char text[14] = "Temperature: "; // Text to notify Temperature output
    char temp_print[5] = "XXXF";           // Text to provide Temperature
    int temp_out, text_out;

    // Arranging output into a proper line

    new_line();
    delay();
```

```
    car_return();

    // Output of the main text
    for(text_out=0; text_out<=12; text_out++)
    {
        // Main printing part

        delay();
        EUSCI_A0->TXBUF = text[text_out];
        delay();

        // Main printing part ends
    }

    // Setting array for printing temperature

    if(tempF <= 9)
    {
        temp_print[1] = '0';
        temp_print[2] = (uint8_t) tempF + '0';
    }
    if(tempF >= 10 && tempF <= 99)
    {
        temp_print[1] = ((uint8_t)(tempF/10.0f)) + '0';
        temp_print[2] = (uint8_t) (tempF - ((float) ((uint8_t)(tempF/10.0f)) *
10.0f)) + '0';
    }
    if(s_flag != 0 && tempC >= 18)
    {
        temp_print[0] = '-';
    }
    if(s_flag == 0)
    {
        temp_print[0] = '+';
    }
    else
    {
        temp_print[0] = '*';
        temp_print[1] = '*';
        temp_print[2] = '*';
    }


    // Output of the temperature reading
    for(temp_out=0; temp_out<=3; temp_out++)
    {
        // Main printing part

        delay();
        EUSCI_A0->TXBUF = temp_print[temp_out];
        delay();

        // Main printing part ends
```

```
    }
}

void new_line(void)
{
    EUSCI_A0->TXBUF = 10;               // New-line function
}

void car_return(void)
{
    EUSCI_A0->TXBUF = 13;               // Carriage return function
}

void new_par(void)
{
    delay();
    new_line();
    delay();
    car_return();
}
```

gps.ino – source code

```
/*This code has been adapted from the original code present in the example folder
 * of the TinyGPS++ library which is available at the below link:
 *
 * http://arduiniana.org/libraries/tinygpsplus/
 *
 * The name of the original code is "Device Example" and the original author of the
 * example code is Mikal Hart. I duly acknowledge the original author based on
whose
 * work I have modified my code accordingly. I also acknowledge that this is not my
 * original work and I have therefore provided credits to Mikal Hart (the original
 * author of the example code). This code has been modified to meet the needs of
 * my project.
 */


#include <TinyGPS++.h>
#include <SoftwareSerial.h>
static const uint32_t GPSBaud = 9600;

// The TinyGPS++ object
TinyGPSPlus gps;

void setup()
{
  Serial.begin(9600);
  ss.begin(GPSBaud);
  Serial.println();
}

void loop()
{
  // Checks if GPS location data is available
  while (ss.available() > 0)
    if (gps.encode(ss.read()))
      display_gps_location();
}

void display_gps_location()
{
  Serial.print(F("Latitude: "));
  if (gps.location.isValid())
  {
    Serial.print(gps.location.lat(), 4);
  }
  else
  {
    Serial.print(F("INVALID"));
  }
  Serial.println();
```

```
Serial.print(F("Longitude: "));
  if (gps.location.isValid())
  {
    Serial.print(gps.location.lng(), 4);
  }
  else
  {
    Serial.print(F("INVALID"));
  }
  Serial.println();
  Serial.println();
}
```

APPENDIX – DATA SHEETS

1.	Microchip Technology Inc. MCP9808
2.	Adafruit OLED SSD1306 Datasheet
3.	u-blox NEO-6 GPS Modules
4.	Invensense MPU9250 Datsheet
5.	Sparkfun DS1307 RTC Datasheet