ECE 3731 Microprocessors and Embedded Systems

Experiment 3:

Sam Khalaf

Professor Aws.

X I certify that this is my own work, and the use of chat gpt or any other sources, such as Chegg, was **not used** in this lab report, except for the use of the assigned starter code and lecture materials used as appropriate.

Sam Khalaf

University of Michigan, Dearborn campus

Abstract :

The main idea of this lab was to be able to demonstrate or gain hands-on experience with the concepts of the communications protocols between the microcontroller and other peripheral devices, or even another microcontroller. The simplest communication protocol was the UART, where we connected wires to another point to send or receive data. This is the slowest communication protocol, with the maximum number of devices being only about 2 devices. I^2c was not as complex but a bit more complex than UART because it involved writing more code to enable sending and receiving data. It can approximately handle about 127 devices, but it will be really hard because it will be super complex due to the number of wires, making it very inefficient. It involves the use of multiple masters or slaves. Spi is complex as you increase the number of devices, but it is one of the fastest protocols; it will require 4 wires (MOSI, MISO, SCLK, SS/CS), and it runs on the idea of one master chip while having multiple slaves; I have included a picture on the right.

Objective:

The main objective of this lab was to be able to convert an analog to a digital value using a microprocessor. The microprocessor captured temperature readings, and we had to convert those that were displayed on the putty terminal via serial connection at a certain baud rate. The putty terminal displayed the values in C, which we then had to modify the code to write into Fahrenheit. The program in this lab was written in C++ instead of our last two labs, where we had to write it in assembly language. The reason that in the last two labs we did it in assembly was to be able to understand how the C gets translated into the compiler using assembly language. Off course, in this lab, we had to understand how to use peripheral devices, the general purpose of Io, and how I2C and UART are incorporated in this lab to be able to understand how to send and receive data to a serial terminal.

Modular programming means being able to break software into two distinct or independent modules, and each function should only be able to contain the desired data to be able to run alone. It's like the previous lab, where we initiated the init common and init inputs and initiated the output functions, where the init_input could function entirely on its own without relying on the inti_output. In this lab, we configured the uart to be able to rely completely on its own without relying on the I^2C. The overall idea of this lab was to be able to debug a real operating system, where we write the code and then debug it step by step to see where we made our mistakes. As professors have mentioned before,we test and then debug instead of running the program.

Deliverable in the experiment write up :

Part 1:

The declared items that we have utilized for this lab were the 2 starter code provided, the lab3 starter code and the I^2c start code, along the help files that were posted. We have also used the canvas lecture slides which was very helpful for understanding how to send/ receive information using uart configuration.

We have utilized these slides which were provided to be able to send/ receive data from uart. The way it does this is by doing put a character UARTCharPut, and it sends a character to terminal putty, then it will enter a loop to read it back via echoing.

Example: Continued

```
    // Configure the UART for 115,200, 8-N-1 operation. This function uses SysCtlClockGet() to
get the system clock frequency. This could be also be a variable or hard coded value instead of a
function call.

    UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 115200,
                        (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                         UART_CONFIG_PAR_NONE));

    // Put a character to show start of example.  This will display on the terminal.

    UARTCharPut(UART0_BASE, '!');

    // Enter a loop to read characters from the UART, and write them back (echo).  When a line
end is received, the loop terminates.

    while(1)
    {   // Read a character using the blocking read function.  This function
        // will not return until a character is available.

        cThisChar = UARTCharGet(UART0_BASE);

        // Write the same character using the blocking write function.  This function will not
return until there was space in the FIFO and the character is written.

        UARTCharPut(UART0_BASE, cThisChar);
    }
```

The lab starter part one had the prototypes and the functions such as the conversion to fahrenheit, and the print temps, which we had to modify. We were given the clkcalevalue, and the array which is used to store data reading from the ADC for temperature holdings. The variables were initialized too which were the TempAvg, and the TempValueC. The clock was changed too which is a very important part of this lab because it's like the heart of this program. Without it the program would not know when to sample for the data. We also had the initsolesconsole( ); this is used for displaying things. The UART printf was provided which basically displayed the message to the serial terminal via uart.

```
// Display the setup on the console.
UARTprintf("ECE473 Lab 3\n");
UARTprintf("**********************************************
UARTprintf("Analog Input: Internal Temperature Sensor\n");
```

The while loop was also provided which basically sampled the data forever, and displayed it to the console. The function provided a way to sample 4 data of temperature and then store then average them out based on the reading to convert the raw data to temp celsius which we were asked to convert into fahrenheit. And then it made a delay which basically offered a delay for the means of generating a constant length.

```
{
    // Trigger the ADC conversion.
    ADCProcessorTrigger(ADC0_BASE, 1);

    // Wait for conversion to be completed.
    while(!ADCIntStatus(ADC0_BASE, 1, false))
    {
    }

    // Clear the ADC interrupt flag.
    ADCIntClear(ADC0_BASE, 1);

    // Read ADC Values.
    ADCSequenceDataGet(ADC0_BASE, 1, ui32ADC0Value);

    //Average the 4 Samples
    ui32TempAvg = (ui32ADC0Value[0] + ui32ADC0Value[1] + ui32ADC0Val

    //Convert Raw Data to Temp Celsius
    ui32TempValueC = (1475 - ((2475 * ui32TempAvg)) / 4096)/10;


    // Display the temperature value on the console.
    PrintTemps (ui32TempValueC);

    //
    // This function provides a means of generating a constant lengt
    // delay.  The function delay (in cycles) = 3 * parameter.  Dela
    // 250ms arbitrarily.
    //
    SysCtlDelay(SysCtlClockGet() / clkscalevalue);
}
```

Below that we were given the function which converted celsius to fahrenheit, and we had to complete it. Additionally, we had a function to print the temperature back into the serial terminal as fahrenheit instead of celsius.

The void init console was also prodiveded to display information as the lab was running. It enabled the GPIO port A which is used for UART and then it had the configuration of the pin mixing on the ports. After that it enabled the clock by using the function sysctlperipheralenable, and uartclcoksourceset, and additionally it enabled the uart alternate function, while initializing

the uart for the console. The ADCInit function is responsible for configuring the ADC0 peripheral. It enables the clock for ADC0, sets up a sequence to perform four samples when triggered by the processor, and enables an interrupt flag to indicate when sampling is done. Additionally, it clears the ADC0 interrupt status flag to ensure it's in a known state before starting the sampling process

For the I^2 we were given the same file but there were alot of other files that were included, added for the setup of the I^2 configurations. We were also given the helper files which contained the instructions and the commands for a previous senior design project that configured the lab 3 code to enable the I2c configuration.

We were given this code and circuit configuration which have helped guide us on how to develop the I^2 configuration with the accelerator, these function were enabling the I^2 modulus, resetting the peripheral of the system and gpio, it also configured the the sda, and the scl which are the clk and the data. Additionally we “hold our breath” and waited for 5 seconds to allow reading. We had to enable the pe4 as scl, and then do some configuration for the master, to initiate the I^2c. The professor explained that false means writing in I2c master slave address set function.

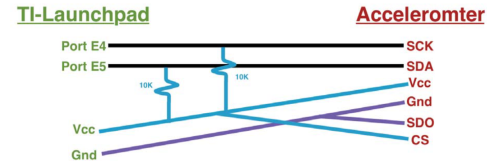
```
static  void  Task1 (void *p_arg)
    {
     uint8_t SLAVE_ADDRESS = 0x18;
     uint8_t SLAVE_ADDRESS_READ =  0x31;

     uint32_t first_byte, second_byte, temperature, result;


//Enable the I2C Module
SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C2);

//reset perhoipheral
SysCtlPeripheralReset(SYSCTL_PERIPH_I2C2);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);

//Wait at least 5 clock cycles
SysCtlDelay(2);

//Configure SDA and SCL
GPIOPinConfigure(GPIO_PE4_I2C2SCL);

GPIOPinConfigure(GPIO_PE5_I2C2SDA);

//Wait at least 5 clock cycles
SysCtlDelay(2);

//Set PE4 as SCL
GPIOPinTypeI2CSCL(GPIO_PORTE_BASE, GPIO_PIN_4);

//Set PE5 as SDA
GPIOPinTypeI2C(GPIO_PORTE_BASE, GPIO_PIN_5);

//Configure Master,
I2CMasterInitExpClk(I2C2_BASE, SysCtlClockGet(), false);

 //I2CSlaveInit(I2C2_BASE, SLAVE_ADDRESS);

//WRITE TO CONTROL REG 0x87
I2CMasterSlaveAddrSet(I2C2_BASE, SLAVE_ADDRESS, false); //false means writing

I2CMasterDataPut(I2C2_BASE, 0x20);
```

The next step was the while (1) loop which basically looped forever until the false means writing, where looping until the bus is no longer busy, true means reading due to the logic of 1. In the while loop, we had a slave address set function to enter our slave address so we can read from that specific accelerometer. Additionally the control and the add functions where provided for the I^2c which helped us

```
341.
342.
343.
344.        while(1){
345.
346.        I2CMasterSlaveAddrSet(I2C2_BASE, SLAVE_ADDRESS, false); //false means writing
347.
348.        I2CMasterDataPut(I2C2_BASE, 0x29);
349.
350.        I2CMasterControl(I2C2_BASE, I2C_MASTER_CMD_SINGLE_SEND);
351.
352.        while(I2CMasterBusy(I2C2_BASE)){}; //Loop until the bus is no longer busy
353.
354.        I2CMasterSlaveAddrSet(I2C2_BASE, SLAVE_ADDRESS, true ); //true means reading
355.
356.        I2CMasterControl(I2C2_BASE, I2C_MASTER_CMD_SINGLE_RECEIVE);
357.
358.        while(I2CMasterBusy(I2C2_BASE)){}; //Loop until the bus is no longer busy
359.
360.        first_byte = I2CMasterDataGet(I2C2_BASE);
361.
362.        UARTprintf(" %d ", first_byte);
363.
364.               }
365.
366.
367.
368.
369.
```

configure it so declare a first byte and send a byte using UARTprintf.

This was also very helpful because it helped us understand where and how to connect our wires with the tiva ware software to better understand what each wire does. The sck was connected on portE4, and the sda the data wire connected to the portE5, while additionally the vcc was connected to the 3.3v, and the gnd was simply connected to ground, the cs was also connected to the Vcc, additionally the sdo was connected to the ground. This diagram explains how to connect the wires in an efficient way so it was very self explanatory.



PART 2:

The first part of the task was relatively straightforward, involving the display of characters of our choice on the PuTTY terminal. To accomplish this, we needed to declare a variable that would allow us to send data to the serial terminal within the PuTTY environment. In C, we use the 'char' data type to declare such a variable, indicating to the compiler that it will store characters.

```
//
int main(void)
{
    char cThisChar;
```

As shown below, initializing the variable is a crucial step. It's both simple and essential because it enables the compiler to recognize this as a variable in the C programming language
The while loop was provided in the class during lecture video, where we had to initiate the “cthischar = uartcharget(UARt0_base)”, we had to put the character into the base, and then the variable that we have previously declared that in the beginning of main, to echo it back.

```
while(1)
{
    // Trigger the ADC conversion.
    ADCProcessorTrigger(ADC0_BASE, 1);
   cThisChar = UARTCharGet(UART0_BASE);
    // Wait for conversion to be completed.
    UARTCharPut(UART0_BASE, cThisChar);
    // Wait for conversion to be completed.
    while(!ADCIntStatus(ADC0_BASE, 1, false))
    {
    }

    // Clear the ADC interrupt flag.
    ADCIntClear(ADC0_BASE, 1);

    // Read ADC Values.
    ADCSequenceDataGet(ADC0_BASE, 1, ui32ADC0Value);

verage the 4 Samples
```

University of Michigan, Dearborn campus

We have used the same while loop as the ADC conversion but the idea is simple. We needed to simply enter a character to show the start of the example and then enter a loop to read from the UART, and then write back to hear the echo. The function this cthischar, is defined as this function will not return until it will capture an echo or a character. This was a very simple example of sending any character of our choice into the uart where we had to visually see it in the putty and then probe it out which will give us a binary equivalent.

Below is the ascii value retrieved which have verified our result because the hex value of 62, and the equivalent is is 01100010.we had used the pc calculator for ease of conversion from the binary values of the oscilloscope, and to convert it to binary value. Left picture is the calculator values and to the right is the oscillation.

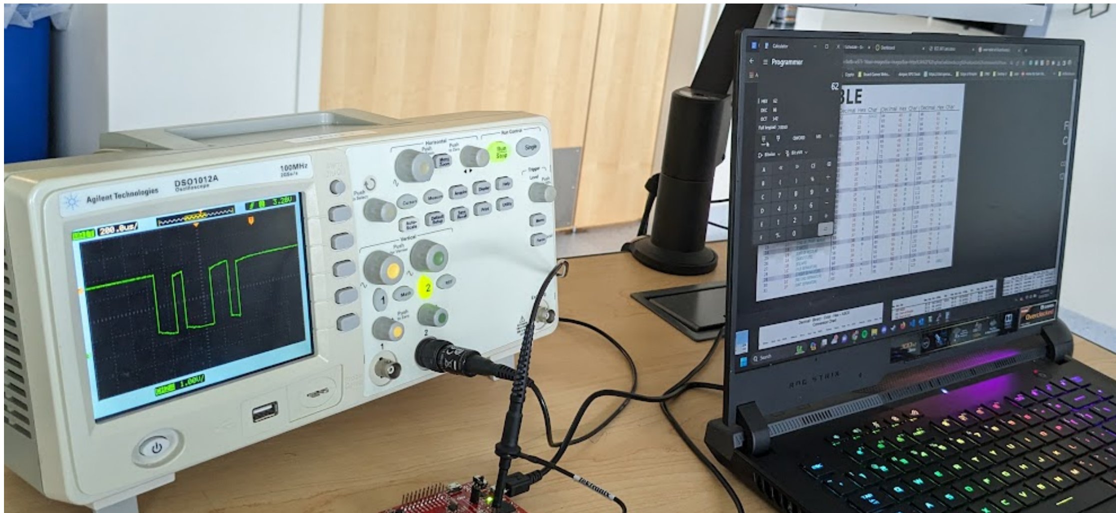
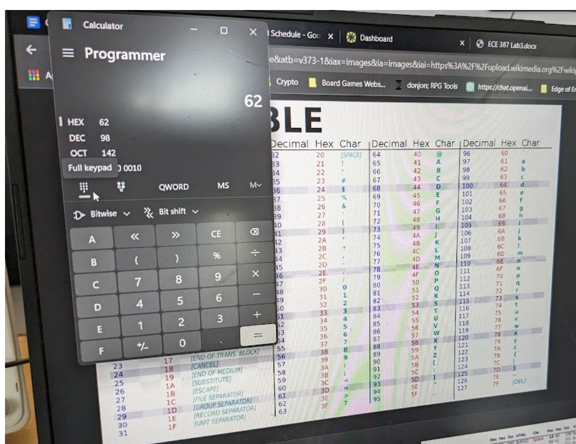


Figure 4: Oscilloscope Output



Part 3:

We have utilized google for assistance with the conversion formula from celsius to fahrenheit. While basically the temperature in fahrenheit is equivalent to the (°C × 9/5) + 32, we have used the function as it was stated that we need to fill it out. Uint 32_t explains that it is an integer value and then we had to use the conversion formula to return the values, that why even the function was not a void rather it was an integer return.

```
uint32_t convertToFahrenheit ( uint32_t TempC)
{
//Fill out this function
uint32_t TempF=0;

    // Conversion formula: Fahrenheit = (Celsius * 9/5) + 32
    TempF = ((TempC * 9) / 5) + 32;
    return TempF;

return TempF;
}
```

Beyond doing the conversion we were latter asked to make it more interactive basically if the user enters “F” vs “f” the specific activities would make the conversion and return a fahrenheit, and if the user enters a c or C it would leave the values alone as the previously were in celsius.

The way we did this was by utilizing the reading character from the terminal uart input, and then respond to the character based on the if statement which compared if the input char was equivalent to the f or Fwhich was to display the fahrenheit by doing conversion and then displaying it on putty. If the letter c or C was pressed we did not need to perform any function as by default it's already in celsius, so that's why we wrote it in the else if part of the statement which basically responded with the uartprintf to show the values in celsius. We also displayed the result in celsius if neither f or c was displayed as by default it was in c. we have demonstrated the functionality of this program during office hours. Also professor himself have helped us during office hours to setup this code because we had a problem with infinite writing, because it displayed fahrenheit only once and then celsius infinitely, the problem was with our logic.

```
//This function prints the two passed temperatures to the console in an
//easy human readable format.
void PrintTemps(uint32_t TempC)
{
    char inputChar;

    // Wait for incoming characters
    //while (1)
    //{
        // Check if a character is available in the UART input
        if (UARTCharsAvail(UART0_BASE))
        {
            // Read the character
            inputChar = UARTCharGet(UART0_BASE);

            // Respond to character inputs
            if (inputChar == 'F' || inputChar == 'f')
            {
                // Respond with the temperature value in Fahrenheit
                uint32_t TempF = convertToFahrenheit(TempC);
                UARTprintf("Temperature = %3d*F\n", TempF);
            }
            else if (inputChar == 'C' || inputChar == 'c')
            {
                // Respond with the temperature value in Celsius
                UARTprintf("Temperature = %3d*C\n", TempC);
            }
            else
            {
                // Unrecognized character, display an error message or take appropriat
                UARTprintf("%c\n", inputChar); //Unrecognized command:
            }
        //}
    }
}
```

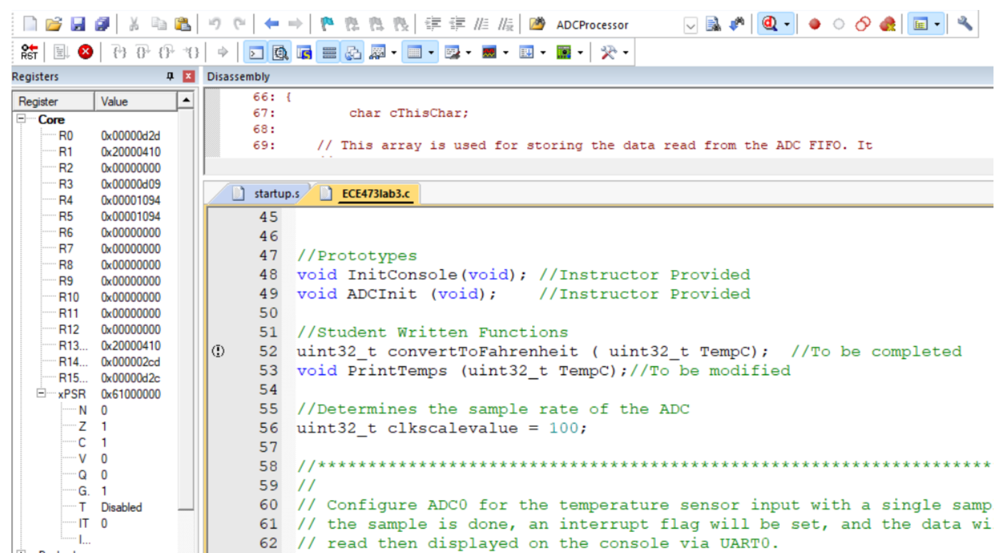
Part4:

Part three asked us to investigate the debugging window feature which basically asked us to use the debugging function and the disassembly features to investigate the c compiler and demonstrate that we actually know how the compiler translates the c++ code into the assembly code by utilizing the breakpoints and to analyze the window.

We had to analyze how the compiler will read the at the beginning of the conversion to the fahrenheit function, which basically we had to observe what the functionality or variables were saved into the stack to perform this conversion. \

Register 13 is accountable for the allocation of the program runtime/memory allocation. The reason why the link register is important is because it holds the values of the return address where it left off in the main, after the completion of the conversion function. This is very important because if it does not return to the proper place it will crash the program.

Professor has specifically highlighted how to analyze the assembly because it will be used in the future labs where the crash will happen in the assembly compared to where interrupts will crash during load assembly.

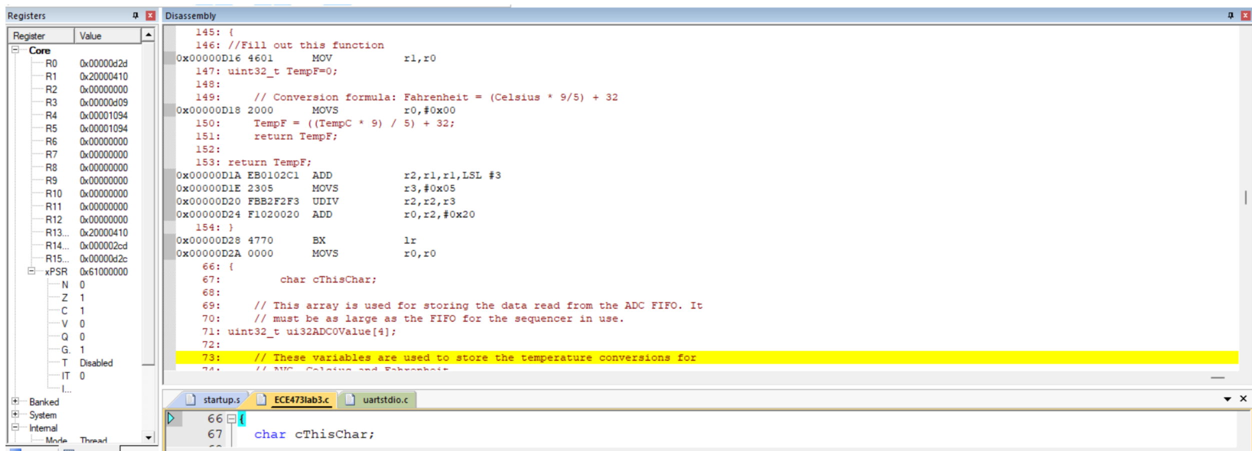


The **bx lr** was also a critical component as it branched with a link register, which helped facilitate this whole operation of doing the function and returning properly without crashing the program.it helped tell the compiler understand where to return to.

It was super cool seeing the operation of the c language being translate back into the assembly language as that will help the compiler translate into the machine language, which computer understands the machine language, and machine language is a bunch of zeros and once. It's inefficient for humans to write in the machine language because it will take so long to perform operations, thus we use c for higher abstraction than even assembly language.

Below is the complete window of the assembly language where the conversion have occurred, as moving and adding, and bx lr, and the movs which checked the status of the flags, as you can view that the operation was not negative, it was zero, and there was a carry, interrupt was disabled in this.

Figure 7: Debug Window



Part 5:

The next part was very complex because we needed to setup the I2c which was super complex as we needed to setup the accelerometer to read values from it, we needed to write to it and set it up so we can retrieve data from the accelerometer which will displayed on a while loop in putty then we will use the data to turn on an led when the value is positive, and another led when the values become negative. We had to demonstrate this to the lab instructor to show that the data actually contributed to the control of the led. We had a lot of difficulty in this part because we did not know why our code only provided one data even though it was in the whole one loop .

```
void I2CInit(void)
{
// setup:
//
//
// Enable the I2C Module
SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C2);
//Wait for the I2C0 module to be ready.

// Reset the peripheral
SysCtlPeripheralReset(SYSCTL_PERIPH_I2C2);

// Enable GPIO Port E
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);


// Wait at least 5 clock cycles
SysCtlDelay(2);

// Configure SDA and SCL
GPIOPinConfigure(GPIO_PE4_I2C2SCL);
GPIOPinConfigure(GPIO_PE5_I2C2SDA);

// Wait at least 5 clock cycles
SysCtlDelay(2);

// Set PE4 as SCL
GPIOPinTypeI2CSCL(GPIO_PORTE_BASE, GPIO_PIN_4);

// Set PE5 as SDA
GPIOPinTypeI2C(GPIO_PORTE_BASE, GPIO_PIN_5);
```

```
82
83
84  // Configure the I2C module as a Master
85  I2CMasterInitExpClk(I2C2_BASE, SysCtlClockGet(), false);
86
87  // Set the slave address for writing
88  I2CMasterSlaveAddrSet(I2C2_BASE, SLAVE_ADDRESS, false);
89
90  // Write data to the control register (0x20)
91  I2CMasterDataPut(I2C2_BASE, 0x20);
92
93  // Perform I2C communication
94  I2CMasterControl(I2C2_BASE, I2C_MASTER_CMD_BURST_SEND_START); // Start the transmission
95  while(I2CMasterBusy(I2C2_BASE)){}; // Wait until the bus is no longer busy
96
97  // Set the slave address for reading
98  I2CMasterSlaveAddrSet(I2C2_BASE, SLAVE_ADDRESS, true); // 'true' means we're reading
99
100 // Send data (0x87 in this case)
101 I2CMasterDataPut(I2C2_BASE, 0x87);
102
103 // Finish the transmission
104 I2CMasterControl(I2C2_BASE, I2C_MASTER_CMD_BURST_RECEIVE_FINISH);
105 while(I2CMasterBusy(I2C2_BASE)){}; // Wait until the bus is no longer busy
106
107
108
109
110
111 }
```

We defined the slave address, then we needed to define the first byte that we will receive from the accelerometer. Also, we had to initiate the I^2c as a void function because we needed to set it up as professor aws has taught me. The slave address was 0X18, then the professor instructed me to set up the voice i2c int to be able to set up the i2c module. Which basically enabled the I2c module, and then reseted the peripherals, and enabled gpios on porte, then we had to wait for about 5 seconds for reading capture. We had to set pe4 as scl, and pe5 as the sda line. We need to configure the master, and then enable I2c communication to perform set and receive data, to perform the transmission.

```
18  //#define SLAVE_ADDRESS 0x19
19  //#define first_byte 0
20
21  //Prototypes
22  void InitConsole(void); //Instructor Provided
23  void ADCInit (void);    //Instructor Provided
24  void I2CInit (void);
25  //Student Written Functions
26  uint32_t convertToFahrenheit ( uint32_t TempC);  //To be completed
27  void PrintTemps (uint32_t TempC);//To be modified
28
29  //Determines the sample rate of the ADC
30  uint32_t clkscalevalue = 100;
31  uint32_t SLAVE_ADDRESS = 0x18;
32  uint32_t first_byte;
33
```

The while one loop allowed you to set the slave address for writing, and then to write to the i2c master, start the communication with only a single send command.we have to wait until idle and then set the next slave address for the reading (true means reading), and then wait for the bus to become idle which means that the compiler have received data, and then we print it over the uart into what we have declared earlier the first byte.

```
while (1) {
    // Set the slave address for writing (false means writing)
    I2CMasterSlaveAddrSet(I2C2_BASE, SLAVE_ADDRESS, false);

    // closed 18, open 19

    // Write data (0x29 in this case)
    I2CMasterDataPut(I2C2_BASE, 0x29);

    // Start the I2C communication with a single send command
    I2CMasterControl(I2C2_BASE, I2C_MASTER_CMD_SINGLE_SEND);

    // Wait for the bus to become idle
    while (I2CMasterBusy(I2C2_BASE)) {};

    // Set the slave address for reading (true means reading)
    I2CMasterSlaveAddrSet(I2C2_BASE, SLAVE_ADDRESS, true);

    // Start the I2C communication with a single receive command
    I2CMasterControl(I2C2_BASE, I2C_MASTER_CMD_SINGLE_RECEIVE);

    // Wait for the bus to become idle
    while (I2CMasterBusy(I2C2_BASE)) {};

    // Read the received data
    first_byte = I2CMasterDataGet(I2C2_BASE);

    // Print the received data over UART
    UARTprintf("%d ", first_byte);
```

Acceleration

The if statement of the reading that we collected was very critical because it basically turned on the red led, and then it turned of the blue led if the value of the byte was larger than the 127, and we had an else condition to show that if the values were less than 127 then turn off the red led and on turn the blue led . This if statement helped configure the setup of the program because it allowed us to visually see what's happening with the reading by the interaction of the led.

```
    if (first_byte >= 127)
        {
            GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, GPIO_PIN_1); // Turn on the red LED
            GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0); // Turn off the blue LED
        }
    else

        {
            GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0); // Turn off the red LED
            GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2); // Turn on the blue LED
        }

    SysCtlDelay(SysCtlClockGet()/clkscalevalue);
```

Figure 11: LED Control

Conclusion :

Throughout the experiment we had set up the uart to the terminal putty to be able to send and receive data for communication to get more of the practice of the protocol. We had to understand how to convert an analog signal to a digital voltage value using the adc, then we needed to understand the c language programming as this lab was the first lab to utilize c, as our previous labs involved assembly language. Then we needed to to understand the uart and the i2c to be able to convert the temperature from fahrenheit to celsius and utilize the if condition to make the program more dynamic. The next part asked us to look at the compiler to see how the code translated into the assembly language. This is critical for next labs for interrupts. In the final part of this lab we set up the i2c to read from the accelerometer and to be able to make it more interactive by setting up an if statement.

We had a lot of challenges with the I^c because we were asked to look over the manual or the guides on how to set up the i2c but it was really confusing because they used a lot of advanced words or terminologies that we are unfamiliar with. Additionally the code needed to performed in a certain procedure to be configured properly, I have made a lot of mistakes with that , as professor have illustrated that we were missing a few lines which was very critical in the setup of the I^C. additionally it was also challenging to wire the pins to tiva c launch board.

In the final aspect of this lab I have learned a lot of important things such as the ability to send information via uart, and then do if statements, and to make code more dynamic, additionally I have learned what the compiler does to translate c code into assembly language. After that I have learned how to set up the I2c code to be able to send and receive information, all that have led to a pleasant experience under the guidance of aws, a phenomenal instructor who does not hesitate to help us with even silly mistakes or things that we should have already known. Aws, I really appreciate you so much for everything you do and continue to help us with.