Final Project Library Documentation

Temperature Sensor and visualizer on iLED

Truman Brown (brow5190)

Introduction:

This library was developed to have a PIC24FJ64GA002 microcontroller interface with the MCP9808 High Accuracy I2C Temperature Sensor Breakout Board via I2C and then display those temperature values on an Akizuki/Sitronix Controlled LCD and then visualize that temperature on a blue-red scale on the WS2812 iLED. The library sets up the MCP9808 Temperature sensor to send digital data via I2C to the PIC microcontroller. Then the library contains code that takes that digital temperature data, and then sends it to the LCD to be displayed as a string. There is code that sets up a double-click detector with a button. If that button is double-clicked, the LCD switches from displaying the temperature in Fahrenheit to Celsius and vice-versa. Finally, the library contains code that takes that temperature data and uses it to light up the iLED on a blue-red scale. This means that if the temperature sensor reads higher temperatures, the iLED fades to red, while if the temperature sensor reads lower temperatures, the iLED fades to blue. At room temperature, the iLED is set to a purple (all red and blue) color.

Hardware Description:

There were many components needed to have this library perform correctly in hardware.

- A PIC24FJ64GA002 microcontroller was needed to be programmed with the library (http://ww1.microchip.com/downloads/en/DeviceDoc/39881e.pdf)
- The MCP9808 High Accuracy I2C Temperature Sensor Breakout Board was needed to actually read temperature data and then send that data to the PIC24 Microcontroller (https://cdn-shop.adafruit.com/datasheets/MCP9808.pdf)
- The ST77032 Controller Chip for the Akizuki LCD was used to display the string form of the temperature data.
    - Sitronix Datasheet- https://drive.google.com/file/d/0B5wD_vPJ2ttWTUZlM18teWhjc2s/view)
    - Akizuki Datasheet- https://drive.google.com/file/d/0B5wD_vPJ2ttWSHRLYmdJVUZOZW8/view
- The WS2812 iLED was needed to fade between the red and blue values and a 100-ohm resistor was necessary to connect the iLED to the microcontroller (https://cdn-shop.adafruit.com/datasheets/WS2812.pdf)
- A Pushbutton Switch used to detect a double-click
- Standard Components to use and support the PIC24FJ64GA002 (pull-up, caps, etc.)

Full Documentation (All public Functions Sorted by Library):

Assembly Library:

void delay_100us(void);

This function takes no parameters and returns nothing. This function was a simple global function written in assembly language that repeats the 'nop' call 1592 times which generates a delay that is equal to 100 microseconds.

void delay_1ms(void);

This function takes no parameters and returns nothing. This function was a simple global function written in assembly language that repeats the ‘nop’ call 16000 times which generates a delay that is equal to 1 millisecond. This function will be used in future functions to generate ‘n’ number of millisecond delays.

void write_0(void);

This function takes no parameters and returns nothing. This function was a global function written in assembly language that first sets RA0 high, then delays to create a 0.35 us pulse in a 1.25 us period (RA0 is set to low).

void write_1(void);

This function takes no parameters and returns nothing. This function was a global function written in assembly language that first sets RA0 high, then delays to create a 0.70 us pulse in a 1.25 us period (RA0 is set to low). Both the write_0/1 functions are used later in the iLED library to drive the LED with pulses to RA0.

iLED Library:

void setupLED(void);

This function takes no parameters and returns nothing. The first thing done in the function was to set the clock to 16 MHz and to set all the pins to digital I/O function like is done for every lab. Then TRISA was configured to set pin RA0 as an output port that will send pulses to the iLED. Then RA0 was initialized to be set to high.

void delay(int delay_in_ms);

This function takes an int delay_in_ms and returns nothing. This function simply delays “delay_in_ms” times, which corresponds to some number of ms delays. This function uses the delay_1ms() function from the assembly library. This function is used many times in the main code to avoid debouncing errors, to allow time for the LCD to send/receive data, and more.

void writeColor(int r, int g, int b);

This function takes three integers, ‘r’ ‘g’ ‘b’, and returns nothing. This function uses the write_1() and write_0() functions from the assembly library to send pulses to the iLED which corresponds to a color composed of 0-255 values of red, green, and blue. It works by bit masking and bit shifting to send a serial value of write_0() and write_1() functions. This function is very important for the main code to fade the iLED between red and blue because this function controls the color of the iLED each time it needs to fade.

LCD Library:

void setupLCD(void);

This function took no parameters and returned nothing. The first thing done in the function was to set the clock to 16 MHz and to set all the pins to digital I/O function like is done for every lab. Then, the I2C interface was set up to run at the minimum frequency of 100 kHz. The I2C interrupt flag was reset, and then I2C was enabled.

void lcd_cmd(char command);

This function took a byte of data (char) and included the necessary protocol to write that data to the slave LCD. This includes initiating the start condition, then addressing the slave, then sending the control byte, then sending the data, then setting the stop bit. This function is used in every other function that communicates with the slave. This is because this is the most basic function needed to send whatever data (command) needed to the LCD. After each write to the data port (I2C2TRN), the I2C2 interrupt flag is reset so that the following commands will be sent.

void lcd_init(void);

This function took no parameters and returned nothing. This function used a series of lcd_cmd calls with different bytes to configure the LCD before any characters were written to it. Specifically, the lcd_cmd's were used to set normal and extended instruction mode, to set the adjustable contrast value, and other necessary commands like clearing the display. This function was necessary because it needed to be called before the setCursor, printChar, and printStr functions could be used. If it wasn't called, the LCD display would've never been turned on or initialized correctly.

void lcd_setCursor(char Columnx, char Rowy);

This function took two char parameters corresponding to the column and the row. This function used an lcd_cmd call as well to set the cursor of the LCD. This function relates to the other functions because it was necessary to set the position of the cursor before any character could be displayed on the LCD.

void lcd_printChar(char myChar);

This function took one char parameter corresponding to whatever ASCII character the user wanted to display on the LCD. This function is very similar in structure to the lcd_cmd function, except that the control byte is slightly changed so that the RS bit is high so the PIC24 can write data to the LCD. Then, the structure is the same as lcd_cmd except that the command char in lcd_cmd is now replaced with the "myChar" byte. This function was necessary for the lcd_printStr function to work because the printStr function is simply writing multiple characters onto the LCD using this function

void lcd_printStr(const char s[])

This function took a constant array 's' of characters that contain the characters of a string the user wants to display on the LCD. This function uses the lcd_printChar function to print each char element of the array 's' so that the LCD will display each element and make it appear like a string. This was the final function that was needed in this lab, and it was just using array logic to call lcd_printChar over and over until all the characters were displayed.

In the main code of the final project, all these functions were used in some way. For example, at the beginning of the main function, setupLCD(), lcd_init(), and lcd_setCursor(0,0) were all called to initialize the LCD and set its cursor to the top left. Then, in the while(1) loop, lcd_printStr(str) was repeatedly called to always be printing the most recent string value of the temperature, whether in Celsius or Fahrenheit. The lcd_printStr(const char s[]) contained a call to lcd_printChar(char myChar) so every function in this library was used in the main file of the final project.

Temperature Sensor Library (Also documented in comments in source code);

void i2c_init(void);

This function took no parameters and returned nothing. The first thing done was to clear the control bits in the I2C1CON register and to clear the status bits in the I2C1STAT register by setting them equal to 0x0000. Then, the slew rate control was disabled and the SMBUS was also disabled in the I2C1CON register. Then the baud rate was set to 100 kHz by setting the I2C1BRG register equal to 0x9D. Then, the interrupt flag was clear and the I2C protocol was enabled. Finally, I2C1 was forced idle by setting the send bit equal to 0. This function was necessary in the main code so that all the proceeding I2C data transfer functions would work properly.

void i2c_start(void);

This function took no parameters and returned nothing. The first thing this function does is that it clears the I2C1 interrupt flag. Then the function waits for the idle condition (SEN=0). Then the start condition is set and the function waits for the interrupt flag to be set. Once it is set, the function resets the interrupt flag. This function was necessary because it is used in the main code to start the I2C data transfer.

void i2c_stop(void);

This function took no parameters and returned nothing. The first thing this function does is that it clears the I2C1 interrupt flag. Then the function waits for the idle condition (SEN=0). Then the stop condition is set (PEN=1) and the function waits for the I2C1 interrupt flag to be set. Once it is set, the function resets the interrupt flag. This function was necessary because it is used in the main code to stop the I2C data transfer.

unsigned char i2c_write(unsigned char i2cWriteData);

This function takes an unsigned char variable 'i2cWriteData' and returns an unsigned char that returns if the transmission is acknowledged. The first thing this function does is that it clears the I2C1 interrupt flag. Then the function waits for the idle condition (SEN=0). Then, the I2C1TRN (transmit register) is set equal to the i2cWriteData char variable to be transmitted. Then the function waits for the I2C1 interrupt flag to be set. Once it is set, the flag is subsequently cleared. Finally, the function returns the acknowledge status, meaning that if the transmission is acknowledged, the function returns '1'.

unsigned char i2c_read(unsigned char ack);

This function takes an unsigned char variable 'ack' and returns an unsigned char that returns the digital data from the temperature sensor. The first thing this function does is that it clears the I2C1 interrupt flag. Then the function waits for the idle condition (SEN=0). Then the function sets the RCEN (receive register enable) to high to enable receive mode. Then the function waits for the I2C1 interrupt flag to be set. Once it is set, the flag is subsequently cleared. Then, an unsigned char variable 'i2cReadData' is set equal to I2C1RCV (receive register), which reads the received data from the sensor. Finally, if the parameter 'ack' is high, then the function transmits an Acknowledge signal, otherwise it transmits a Not Acknowledge signal. Finally the function waits for the I2C1 interrupt flag to be set. Once it is set, the flag is subsequently cleared and the function returns the unsigned char variable 'i2cReadData'.

void i2c_config(void);

This function took no parameters and returned nothing. This function contains various calls to i2c_start(), i2c_write(...), and i2c_stop() to send packets of commands that initialize the temperature sensor to work correctly. This function also contains the i2c_write(0b00110000); call that is the slave address byte for the temperature sensor.

void initPushButton(void);

This function took no parameters and returned nothing. This function is not necessarily related to the temperature sensor, but I decided to put in this library because it is used to control whether the temperature will be displayed in Celsius or Fahrenheit. This function sets TRISB15 high so that RB15 will be read as digital input. Then, using the CNPU register, an internal pull-up resistor for RB15 is set high. Then, TMR2 is initialized to have a prescaler of 256:1, and the period to be equal to 1 second. Additionally, the interrupt for TMR2 was initialized with the Interrupt Service Routine in the main code.

volatile unsigned int overflow=0;

void __attribute__((interrupt, auto_psv)) _T2Interrupt(void);

This was the ISR for TMR2, and all it did was that it reset the T2 interrupt flag and then increased the overflow volatile variable each time the interrupt was triggered.

Main Code

This main code begins by calling all the setup functions, including setupLED(), setupLCD(), lcd_init(), lcd_setCursor(0,0), i2c_init(), and initPushButton(). There were calls to delay(1) to delay between each setup to allow enough time for the data to be transmitted via I2C. Then, several variables were created to store temperature related data. Each variable has commented notes next to it describing what the variable is used for. Then, two arrays of size 255 each were initialized. The first array, tempArr, was initialized to store the values {70, 69.85, 69.70, 69.55,.....,(70-i*0.15)}. These were initialized as "checkpoints" for temperatures in a way. This was done so that the latest temperature sensor reading could be compared to each of these values and then the RGB color could be determined based on how far into the array the temperature was less than. The second array, tempArr1, was initialized to store the values {70, 70.15, 70.30, 70.45,.....,(70+i*0.15)}. These were also initialized as "checkpoints" for temperatures.

Then, the while(1) loop that repeats forever began. The variable press was set equal to RB15, which corresponds to 0 if the button is pressed and 1 if the button is not pressed. Then i2c_config() is called. Then, both the UpperByte and LowerByte of the temperature data from the temperature buffer from the sensor are read and an ACK and then NACK signal are sent. Then i2c_stop() is called to end the I2C data transfer during each loop. I included code that detects if the ambient temperature is either equal to T_{crit}, greater than T_{upper}, or lower than T_{lower}. This code is commented out however because T_{crit}, T_{upper}, and T_{lower} are all customizable temperature checkpoints that are set to 0 deg Celsius by default and this code may be uncommented if the user wishes to change these temperature checkpoints. Then, the flag bits from the UpperByte data are cleared using bit masking because the code only wants the temperature data and not the flag data. Then, the UpperByte is checked in two if loops to determine whether the temperature is less than 0 degrees Celsius or greater than 0 degrees Celsius. Within each if statement, the variable 'Temperature' is set equal to the degrees Celsius floating point value that the temperature sensor is getting. Then, the Celsius data is transformed to Fahrenheit data in an equation.

Then, an if statement checks if the button is pressed and the lastPress variable was equal to 'not pressed'. Another nested if statement checks if the double click was less than 0.25 seconds, and if it was, a variable 'TorC' is inverted between 0 and 1. This variable corresponds to whether the temperature should be displayed on the LCD as Celsius or Fahrenheit. Then lastPress is set equal to press to save the most recent press. Then the code is delayed for 2 ms to avoid debouncing errors in the button switch.

Then there is an if and an else statement that checks if TorC equals 1 or 0. If TorC equals 1, then the LCD displays the temperature in Fahrenheit and if TorC equals 0 then the LCD displays the temperature in Celsius. The sprintF() function was used to convert the floating point number to a constant array of characters.

Finally, the temperature in Fahrenheit is compared against the previous tempArr and tempArr1. This was done to compare the most recent temperature reading to an array of checkpoints for temperature. For example, if the temperature sensor reading was 56.5 degrees F, then the temperature would be less than all temperatures [0….90] in tempArr. Then, a variable t1 is set to 90 and writeColor is called with the formula writeColor(255-t1,0,255-t2). To better understand this, consider a temperature sensor reading greater than 150 degrees Fahrenheit. Knowing that the max value stored in tempArr1 is 108.25 degrees F, this means that this temperature reading would be greater than every tempArr1[k]. That means k would be equal to the array size, or 255, and k is equal to t2. Then when you call writeColor(255-t1,0,255-t2), this is equal to writeColor(255,0,0), or a completely red LED, which is what we want.

<u>Basic Usage Example:</u>

A basic usage example of this program would be the following: If you were to blow hot air onto the temperature sensor, you would see the temperature reading on the LCD increase and you would see iLED fade from purple to red. This is because the hot air increases the temperature of the sensor. The sensor then sends that temperature reading to the PIC24. Then the PIC24 microcontroller sends then converts that temperature data into a char array and then sends that data to the LCD. The PIC24 also interprets that temperature data and as the temperature increases, the PIC24 commands the iLED to show less blue color, meaning that the color would fade from bright purple to bright red.

<u>Advanced Usage Example:</u>

An advanced usage example of this program would be the following: If, at first, you were to blow hot air onto the temperature sensor, you would see the temperature reading on the LCD increase and you would see iLED fade from purple to red. Then, however, if you let the temperature sensor rest at room temperature, you would slowly see the temperature reading on the LCD decrease to room temperature and you would see the iLED fade from red back to purple. Then, if you were to put an ice-pack, anything else cold onto the sensor, you would see the temperature reading on the LCD decrease and you would see iLED fade from purple to blue. You could repeat these examples as much as you would like and you would see that the iLED will always fade (not jump) to red if the temperature sensor is reading something hot, fade to blue if the temperature sensor is reading something cold, or fade to purple if the temperature sensor is at room temperature. Additionally, if at any time you want to switch the temperature reading on the LCD from Fahrenheit to Celsius, or Celsius to Fahrenheit, you could double click the button on the circuit to switch the units and the corresponding temperature readings. If you single click the button, nothing will happen as I implemented the button as a double-click detector. Finally, the iLED will perform the correct fading behavior no matter if the LCD is displaying the temperature in Fahrenheit or Celsius.