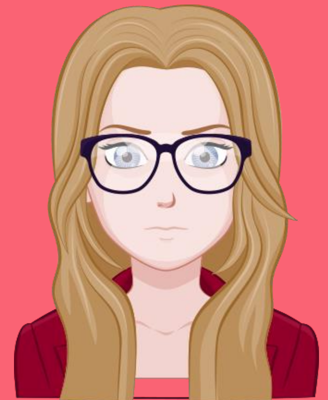
INTRODUCTION TO SMART SYSTEMS

Designing, Developing and Testing an Embedded Open-loop Controller



By Kayleigh Lamb
(14th Dec 2015)

Table of Contents

Introduction to Smart Systems....0
Designing, Developing and Testing an Embedded Open-loop Controller....0
1. Introduction....2
2. Design & Development....4
Real World Application....4
Input & Output Sensors....5
Setup design of the thermistor....6
Thermistor Pin and Bit Mapping....6
Port F Setup....6
Adjustment the to the Multiplex Pins on Port F....6
ADC setup and initialisation....6
Converting the ADC Values....7
Interpolation formula to assume missing y values....7
Setup and use of the onboard timers....8
Timers Used....8
Pre-scaler and timer intervals....8
Timer Priorities....8
Setup and use of the Real Time Clock Module (RTC)....8
3. Testing....9
4. Program code listings....10
Room_Temperature_Sensor.c....10
initialiseADC.h....12
initialiseTimers.h....12
getDateandTime.h....13
getTemperature.h....14
ASCII_MATRIX.h....14
Header files used a libraries for my project....15
LCD_LibraryFunctions_1281.h....15
TWI_Master_For_RTC_ModuleSLAVE_1281_C.h....19
5. Critical evaluation and conclusion....25

1. Introduction

I have chosen to build a thermostat that will display the temperature of a room on an LCD display. Also on the LCD display, the wall clock time and date will be visible. For my system, I will use the following custom sensors:

1. 16 x 2 Intelligent LCD Module

2. Thermistor – Temperature sensor

3. Real Time Clock module

The system will update the temperature reading from the thermistor sensor every 60 seconds and display this on the LCD display. The system will also update the wall clock time on the LCD Display every 10 seconds.

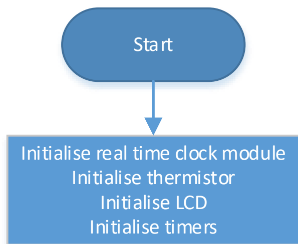
For better accuracy of temperature readings, we will take 3 temperature readings spaced 4 seconds apart, find the average and use this as the final reading.

The system will make use of two interrupts, one for reading the temperature and one for updating the time and date on the LCD display. The real time clock module will be used to keep track of the wall clock date and time whilst the system is turned off and will provide better accuracy.

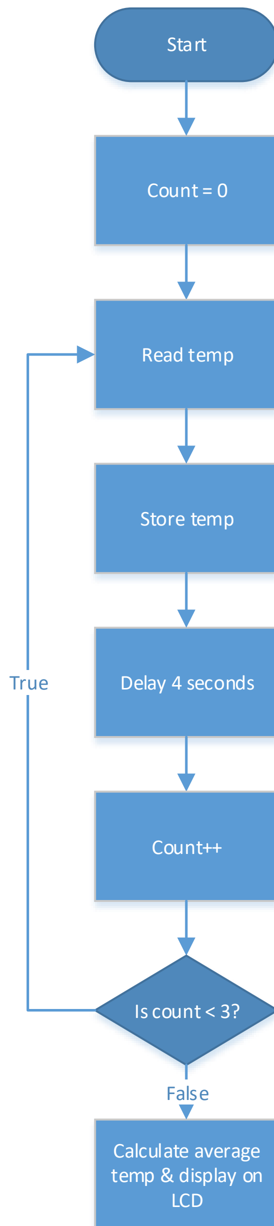
We will make use of timers TC4 & TC5 as they are both 16 bit which makes it easier to time longer intervals of time. Both timers can generate interrupts, which can be used for the temperature readings and retrieving the wall clock time from the real time clock module.

In order for the Dragon debugger and the thermistor to work together, I will use jumper cables to map bit 6 of port F.

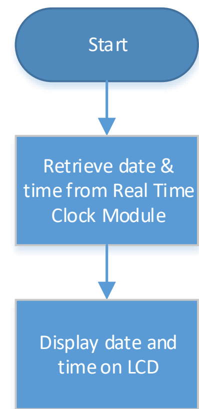
Main Program



Temperature Interrupt



Wall Clock Date & Time Interrupt



2. Design & Development

Real World Application

After looking more into my smart system, I realise that “thermostat” is the wrong name for it, as it is only able to sense the temperature of a room and not control it. Instead I am going to call it a Room Temperature Sensor with accurate “Wall clock” date and time. The intended use of the system is to relate the temperature of the room it is in and provide the user with the current date and time.

The system is a Synchronous system as all of the operations are co-ordinated under the control of the 1 Mhz fixed rate clock pulses, which are used by the timers for counting. When the required count value has been reached, an interrupt is generated, and the count starts again.

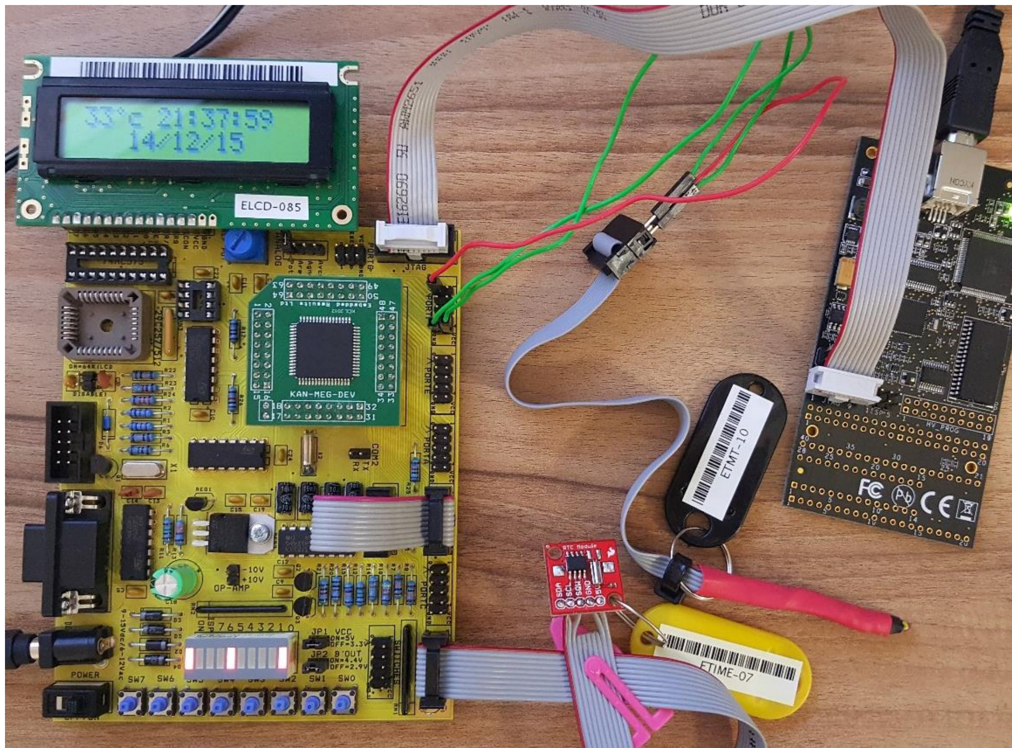


Figure 1 - Full system setup

Input & Output Sensors

Figure 2 – LCD

The LCD screen will be used to output the date, time and temperature values.

This requires the Data Direction Register for Ports A, C & G be set for output. I will also use the following header file as a library:

```
#include "LCD_LibraryFunctions_1281.h"
```

Figure 3 – Thermistor

The thermistor will be connected to the ADC on Port F and will input an analog signal which will be converted to a digital one. See the thermistor setup section of this report for more information on the bit mapping.

Figure 4 - Real Time Clock

The RTC requires that pull up resistors are set on Port D and the DDR for Port D is set for input. See the RTC setup section of this report for more information.

Figure 5 - Onboard LED's

The onboard LED's will be used as an output connected to Port B. The LED's have been included for diagnostic purposes during the development and implementation of the Room Temperature Sensor.

This requires the DDR for Port B is set for output.

Setup design of the thermistor

Thermistor Pin and Bit Mapping

The thermistor uses the analog to digital converter (ADC) on Pin 6 of Port F. Unfortunately, the Dragon JTAG debugger board also uses this pin, and without bit mapping, I will be unable to get an accurate thermistor measurement.

In order to solve this problem, I have used three male to female jumper wires and have connected the thermistor's Pin 6, to Pin 0 on Port F and have adjusted the ADC initialization to convert from Pin 0 instead of Pin 6. This leaves Pin 6 free for use by the Dragon debugger board and means that there will now be no interference with my analog to digital conversion.

Port F Setup

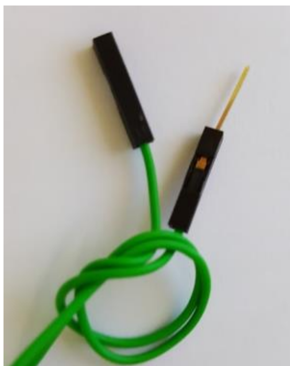


Figure 6 – M to F Jumper

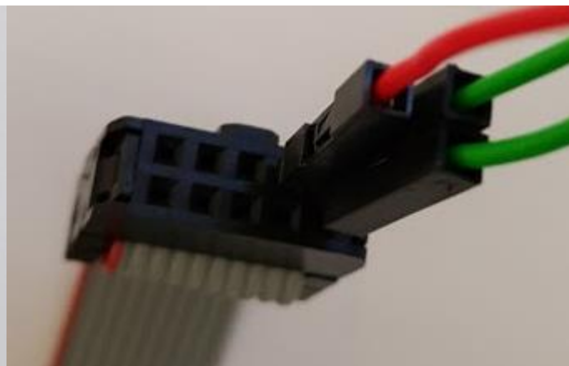


Figure 7 - Thermistor bit 6 (red) & vcc and gnd

Figure 8 - Bit 6 jumper connected to bit 0 on PORT F

Adjustment the to the Multiplex Pins on Port F

Altering the following line of code, tells the ADC to convert the input from pin 0 instead of pin 6:

```
ADMUX = 0b01100110;  // Use AVCC as voltage ref, Convert channel 6 (Thermistor connected to bit 6)
ADMUX = 0b01100000;  // Use AVCC as voltage ref, Convert channel 0 (Thermistor connected to bit 0
with jumper wires)
```

ADC setup and initialisation

`ADCSRA = 0b10101110;`	Enable the ADC, Auto trigger and interrupt with pre-scaler of 64
`ADCSRB = 0b00000000;`	Clear bits 0, 1, 2, & 3 – Free running mode
`DIDR0 = 0b01000000;`	Disable digital input on bit 6
`DIDR2 = 0b11111111;`	Disable digital input on all bits
`ADCSRA |= 0b01000000;`	Read ADCSRA and OR with this value to reset the flag so conversion can begin again

I chose to leave the ADC in free running mode as it was easier for diagnostic purposes to have constant updated ADCH values for the interpolation formula.

Converting the ADC Values

I will need to read the ADC value and convert this into a temperature reading. I have been given the following typical ADC values for the lower and upper bound range 0 degrees to 100 degrees (taken from Richard's sample thermistor and ADC readings):

	Y - Temperature Degrees Celcius	X - ADC reading in decimal
1	0	079
2	100	239

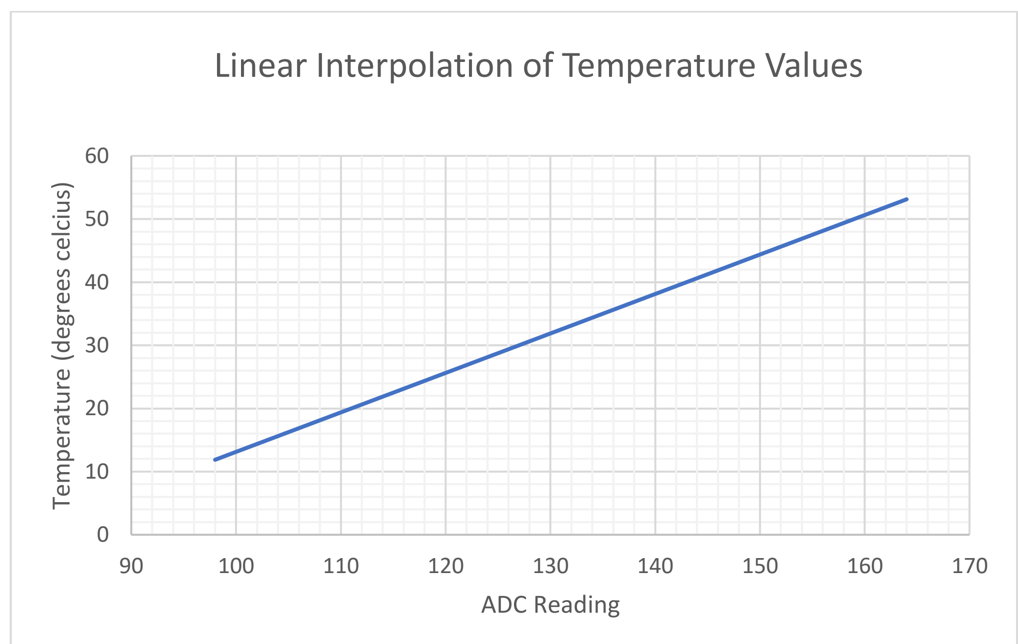
Interpolation formula to assume missing y values

Unfortunately, there is no datasheet available for the thermistors in the lab, but after contacting and taking advice from my lecturer Richard Anthony, I did some research into interpolation and came up with the following formula:

$$y = \frac{((x - x_1) \times (y_2 - y_1))}{(x_2 - x_1)} + y_1$$

Using this formula, I get the following result set and linear graph:

ADC output	Temperature
98	11.875
99	12.5
100	13.125
101	13.75
102	14.375
…	…
127	30
128	30.625
129	31.25
130	31.875
131	32.5
132	33.125
133	33.75
134	34.375
135	35
…	…
175	60
176	60.625
177	61.25
178	61.875
179	62.5
180	63.125
181	63.75
182	64.375
183	65



For example, when the ADC value reads as 111:

$$y = \frac{((111 - 79) \times (100 - 0))}{(239 - 79)} + 0$$
$$y = 20$$

The new interpolated temperature value is calculated as $20°c$. The formula assumes a linear relationship between the highest and lowest variables and because of this, I can estimate what the missing values might have been. Although this means that my temperature readings will not be accurate, I am happy that they are accurate enough for their intended purpose. I will not display the decimal places of the temperature in my application.

Setup and use of the onboard timers

Timers Used

Onboard timers 4 and 5 will be used as counters with interrupts. I will use the first timer to retrieve the date and time from the Real Time Clock Module (RTC) and the latter to sample the temperature and display this on the LCD every second.

Pre-scaler and timer intervals

In order to achieve a one second interval, with a 1 Mhz clock and a 1024 pre-scaler, the timer will need to count 1,000,000/1024 clock cycles which is equal to 976 in decimal or 0x3D0 in hexadecimal. My initial plan had the timers generating interrupts at 4 and 10 second intervals, but using the equipment in the lab, I see it's much more convenient to shorten these.

The timers use the output compare registers (High end and Low end) to see if enough clock cycles have been counted to generate the interrupt. In order to use interrupts, I will include the `<avr/interrupt.h>` library as a header file and enable global interrupts in my program initialisation with `sei();`.

Timer Priorities

On the AtMega1281, the priorities of each timer are fixed in the interrupt vectors number order. So the higher the number, the less priority the interrupt is given. This means that Timer 4 has priority over Timer 5, which is why I designed it so that Timer 4 would handle the "Wall Clock Time" as this value changes more often than the temperature value of a room. Therefore, it makes sense that this should be given precedence over the temperature.

Setup and use of the Real Time Clock Module (RTC)

The RTC will be programmed with the date and time data only once (one off code will be commented out of the application so that the date and time values are not reset each time program the board), as it has its own power source and continues counting even when the power is removed from STK300 board. The RTC has a quartz crystal which means it keeps time more accurately than the on-board timer/counters and is well suited for my system where the user would expect the time displayed to be accurate.

I will include the following header file for use as a library: `#include "TWI_Master_For_RTC_ModuleSLAVE_1281_C.h"`
I have made only a small change to the `void RTC_module_ReadAllSevenDataRegisters()` function by adding a call to two of my own functions:

```
displayDateOnLCD(uRTC_Address_DATE_TENS, uRTC_Address_DATE_UNITS, uRTC_Address_MONTH_TENS,
uRTC_Address_MONTH_UNITS, uRTC_Address_YEAR_TENS, uRTC_Address_YEAR_UNITS);

displayTimeOnLCD(uRTC_Address_HOURS_TENS, uRTC_Address_HOURS_UNITS, uRTC_Address_MINUTES_TENS,
uRTC_Address_MINUTES_UNITS, uRTC_Address_SECONDS_TENS, uRTC_Address_SECONDS_UNITS);
```

I have also removed all the functions that related to using the SERIAL USART connection and displaying message on HyperTerminal when connected to a PC as this was not relevant to my application.

3. Testing

As this is only a room temperature sensor, I will test it against extreme cold and extreme heat.

Test No.	Test	Expected Result	Actual Result	Pass/Fail?
1	Touch the thermistor on ice	The temp on the display should drop to 0 degrees	It takes a few seconds, but the temperature does gradually drop down to 0 - below this though the system isn't designed to support temp values less than 0, so we get unexpected behavior.	Pass
2	Dip the thermistor in boiling water	The temperature should rise up to 99 degrees	It takes a few seconds, but the temperature does gradually increment up to 99 degrees – above this though the system isn't designed to support temp values greater than 99, so we get unexpected behavior.	Pass
3	Touch the thermistor on ice and see how long it takes to return to room temperature (about 31 degrees)	As the interrupts read and display the temperature every second, I expect it will take about 30 seconds	The thermistor took 32 seconds, which shows that it reacts to extreme changes quite quickly.	Pass

4. Program code listings

Room_Temperature_Sensor.c

```
/*
 *  Room_Temperature_Sensor.c
 *     Application to read thermistor temperature and display it on an LCD
 *     Interpolation formula developed for temperature values
 *     based on known values:
 *     Temp = ((adc - adclow)*(temphigh - templow))/(adchigh - adclow)+ templow
 *  Created: 09/12/2015 05:40:01
 *  Author: Kayleigh Lamb - 000773715 University of Greenwich, Introduction to Smart Systems
 */


#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>            // Add this header so the _delay_ms() function can be used
#include "LCD_LibraryFunctions_1281.h"
#include "initialiseADC.h"
#include "getTemperature.h"
#include "initialiseTimers.h"
#include "ASCII_MATRIX.h"
#include "TWI_Master_For_RTC_ModuleSLAVE_1281_C.h"
#include "getDateandTime.h"


//Declare Functions
void initialiseADC();
void initialiseGeneralProgram();
void initialiseTimersFourAndFive();
void displayTempOnLCD();

// RTC Library functions
void RTC_Module_Initialisation();
void RTC_Module_PointTo_Seconds_Register();
void RTC_module_ReadAllSevenDataRegisters();
// TWI related function prototypes
void TWI_MASTER_SETUP();
void TWI_SendSTART();
void TWI_Send_STOP();
void TWI_Send_SLA_W(char cTWI_7bit_RecipientAddress);
void TWI_Send_SLA_R(char cTWI_7bit_RecipientAddress);
void TWI_MASTER_CheckStatusRegisterValueAgainstExpectedValue(unsigned char cExpectedStatusCode);
void WaitFor_TWINT_FLAG();



int main(void)
{
       initialiseGeneralProgram();
       initialiseADC();
       TWI_MASTER_SETUP();
       //RTC_Module_Initialisation(); - Method only called once to set the RTC date and time, 
commented out of program to prevent reset each time program is loaded onto the board.
       initialiseTimersFourAndFive();

    while(1)
    {
        //Two interrupts used so no need for loop code

    }
}

void initialiseGeneralProgram()
```

```
{
	DDRA = 0xFF;			// Configure PortA direction Output
	DDRC = 0xFF;			// Configure PortC direction Output
	DDRG = 0xFF;			// Configure PortG direction Output
	DDRB = 0xFF;			// Configure PortB direction for Output
	PORTB = 0xFF;			// Set all LEDs initially off (inverted on the board, so '1' = 
off)
	DDRD = 0x00;			// Configure PortD direction for Input
	DDRD &=  0b11111100; // clear bits 1,0 (set direction input)
	PORTD |= 0b00000011; // Set bits 1,0 (set pullup resistors)
	sei();					// Enable interrupts at global level, set Global Interrupt 
Enable (I) bit
}

ISR(ADC_vect) // ADC Interrupt Handler
{
//Free running mode means that the ADC interrupt starts the next conversion when the last conversion 
has ended

}

ISR(TIMER4_COMPA_vect)			// TIMER4_CompareA_Handler (Interrupt Handler for Timer 4)
{
	getTemperature();

}

ISR(TIMER5_COMPA_vect)			// TIMER5_CompareA_Handler (Interrupt Handler for Timer 5)
{
	RTC_Module_PointTo_Seconds_Register();
	RTC_module_ReadAllSevenDataRegisters();

}
```

initialiseADC.h

```
/*
 * Initialise the adc and start conversion functions
 *
 * Created: 09/12/2015 08:56:14
 *  Author: Kayleigh
 */


void initialiseADC()
{
	//	Set ADC multiplexer selection register (ADMUX)
	ADMUX = 0b01100000;		// Use AVCC as voltage ref, Convert channel 0 (Thermistor
connected to bit 0 with jumper wires)

	ADCSRA = 0b10101110; // ADC enabled, Auto trigger, Interrupt enabled, Prescaler = 64
	ADCSRB = 0b00000000; // clear bits 3,2,1,0 (Free running mode)

	// DIDR0 - Digital Input Disable Register 0
	// Bit 7:0 – ADC7D:ADC0D: ADC7:0 Digital Input Disable
	DIDR0 = 0b01000000;  // Disable digital input on bit 6

	// DIDR2 – Digital Input Disable Register 2
	// Bit 7:0 – ADC15D:ADC8D: ADC15:8 Digital Input Disable
	DIDR2 = 0b11111111;  // Disable digital input on all bits (64-pin version of ATmega1281 does
not even have these inputs)
	// Start the ADC Conversion (start first sample, runs in 'free run' mode after)
	//	bit 6 ADSC (ADC Start Conversion) = 1 (START)
	//	Read ADSCSR and OR with this value to set the flag without changing others
	ADCSRA |= 0b01000000;
}
```

initialiseTimers.h

```
/*
 *
 * Created: 09/12/2015 09:09:27
 *  Author: Kayleigh
 */

void initialiseTimersFourAndFive()
{
	TCCR4A = 0b00000000; // Normal port operation (OC4A, OC4B, OC4C), Clear Timer on 'Compare
Match' (CTC) waveform mode)
	TCCR4B = 0b00001101; // CTC waveform mode, use prescaler 1024
	TCCR4C = 0b00000000;

	// For 1 MHz clock (with 1024 prescaler) for an approx 1 second interval:
	// Need to count 1 million clock cycles (but already divided by 1024)
	// So actually need to count to (1,000,000 / 1024 =) 976 decimal, = 3D0 Hex
	OCR4AH = 0x03;				// Output Compare Registers (16 bit) OCR1BH and OCR1BL
	OCR4AL = 0xD0;
	TCNT4H = 0b00000000; // Timer/Counter count/value registers (16 bit) TCNT4H and TCNT4L
	TCNT4L = 0b00000000;
	TIMSK4 = 0b00000010; // bit 1 OCIE4A		Use 'Output Compare A Match' Interrupt, i.e.
generate an interrupt
	// when the timer reaches the set value (in the OCR4A register)

	TCCR5A = 0b00000000; // Normal port operation (OC5A, OC5B, OC5C), Clear Timer on 'Compare
Match' (CTC) waveform mode)
	TCCR5B = 0b00001101; // CTC waveform mode, use prescaler 1024
	TCCR5C = 0b00000000;
```

```
        // For 1 MHz clock (with 1024 prescaler) for an approx 4 second interval:
        // Need to count 1 million clock cycles (but already divided by 1024)
        // So actually need to count to (1,000,000 / 1024 =) 976 decimal, = 3D0 Hex
        OCR5AH = 0x03;                   // Output Compare Registers (16 bit) OCR5BH and OCR5BL
        OCR5AL = 0xD0;
        TCNT5H = 0b00000000; // Timer/Counter count/value registers (16 bit) TCNT5H and TCNT5L
        TCNT5L = 0b00000000;
        TIMSK5 = 0b00000010; // bit 1 OCIE5A            Use 'Output Compare A Match' Interrupt, i.e.
generate an interrupt
        // when the timer reaches the set value (in the OCR5A register)

}
```

getDateandTime.h

```
void displayDateOnLCD(unsigned char dayTens, unsigned char dayUnits, unsigned char monthTens,
unsigned char monthUnits, unsigned char yearTens, unsigned char yearUnits)
{

        unsigned char dayMSB = matchWithASCII(dayTens); // get hex codes for ASCII characters
        unsigned char dayLSB = matchWithASCII(dayUnits);

        unsigned char monthMSB = matchWithASCII(monthTens);
        unsigned char monthLSB = matchWithASCII(monthUnits);

        unsigned char yearMSB = matchWithASCII(yearTens);
        unsigned char yearLSB = matchWithASCII(yearUnits);

        lcd_SetCursor(0x44);        // 0b01000101 Set cursor position to line 2, col 4
        lcd_WriteChar(dayMSB);             // Print the day
        lcd_WriteChar(dayLSB);
        lcd_WriteChar(0x2F);        // print the "/" character
        lcd_WriteChar(monthMSB);    // print the month
        lcd_WriteChar(monthLSB);
        lcd_WriteChar(0x2F);
        lcd_WriteChar(yearMSB);            // print the year
        lcd_WriteChar(yearLSB);
}

void displayTimeOnLCD(unsigned char hourTens, unsigned char hourUnits, unsigned char minTens,
unsigned char minUnits, unsigned char secTens, unsigned char secUnits){

                unsigned char hourMSB = matchWithASCII(hourTens); // get hex codes for ASCII
characters
                unsigned char hourLSB = matchWithASCII(hourUnits);

                unsigned char minMSB = matchWithASCII(minTens);
                unsigned char minLSB = matchWithASCII(minUnits);

                unsigned char secMSB = matchWithASCII(secTens);
                unsigned char secLSB = matchWithASCII(secUnits);

                lcd_SetCursor(0x06);        // 0b01000101 Set cursor position to line 1 col 12
                lcd_WriteChar(hourMSB);            // display the hour
                lcd_WriteChar(hourLSB);
                lcd_WriteChar(0x3A);        // write the character ":"
                lcd_WriteChar(minMSB);             // display the minutes
                lcd_WriteChar(minLSB);
                lcd_WriteChar(0x3A);
                lcd_WriteChar(secMSB);             // display seconds
                lcd_WriteChar(secLSB);

}
```

getTemperature.h

```
/*
 *
 * Created: 09/12/2015 09:03:17
 *  Author: Kayleigh
 */


	//Declare Global Values
	unsigned char Temperature;


	void getTemperature()
	{
		//Declare values for interpolation formula
		int adc = ADCH;
		int tempLow = 0x0;   //in dec = 0 degrees
		int tempHigh= 0x64;  //in dec = 100 degrees
		int adcLow = 0x4F;   //in dec = 79 adc reading at 0 degrees
		int adcHigh = 0xEF;  //in dec = 239 adc reading at 100 degrees
		//int adjust = 0x18;  // - 20 adjustment for calibration
		Temperature = ((adc - adcLow)*(tempHigh - tempLow))/(adcHigh - adcLow) - tempLow;
		PORTB = ~Temperature; //send the calculated temperature to the LEDS but invert first
as leds are inverted on the board
		displayTempOnLCD(Temperature);
	}

	void displayTempOnLCD(int Temperature)
	{
		//Split integer into tens and units
		int units = Temperature % 0x0A;				//modulus by ten leaves
remaining units
		int tens = (Temperature - units)/ 0x0A;		//minus units and divide by ten

		unsigned char digit1 = matchWithASCII((unsigned char)tens);
		unsigned char digit2 = matchWithASCII((unsigned char)units);

		lcd_Clear();				// Clear the display
		lcd_StandardMode();			// Set Standard display mode
		lcd_on();					// Set the display on
		lcd_CursorOff();			// Set the cursor display off (underscore)
		lcd_CursorPositionOff();	// Set the cursor position indicator off (flashing square)
		lcd_SetCursor(0x01);		// Set the cursor to row 1 col
		lcd_WriteChar(digit1);			// display digit1
		lcd_WriteChar(digit2);			// display digit2
		lcd_WriteChar(0xB2);		// print the degrees character
		lcd_WriteChar(0x63);		// print the letter c
	}
```

ASCII_MATRIX.h

```
/*
 *  function to match the digit with the correct Hex code for the
 *  relevant ASCII character
 *  Created: 09/12/2015 08:56:14
 *  Author: Kayleigh
 */

matchWithASCII(int data){
		unsigned char matrix;
```

```
		if(data == 0x00){
			matrix = 0x30;
			}else if(data == 0x01){
			matrix = 0x31;
			}else if(data == 0x02){
			matrix = 0x32;
			}else if(data == 0x03){
			matrix = 0x33;
			}else if(data == 0x04){
			matrix = 0x34;
			}else if(data == 0x05){
			matrix = 0x35;
			}else if(data == 0x06){
			matrix = 0x36;
			}else if(data == 0x07){
			matrix = 0x37;
			}else if(data == 0x08){
			matrix = 0x38;
			}else if(data == 0x09){
			matrix = 0x39;
		}
		return matrix;
	}
```

Header files used a libraries for my project

LCD_LibraryFunctions_1281.h

```
//*********************************************************************************
// Project		LCD Library Functions header (Embedded C)
// Target		ATMEL ATmega1281 micro-controller on STK300 board
// Program		LCD_LibraryFunctions_1281.h
// Author		Richard Anthony
// Date			11th October 2013 (8535 version 8th October 2011)

// Function		Provides low-level functions to configure the LCD

// Ports		Uses the 14-pin SIL LCD interface connection on the STK300

//				Uses port A for Data and Command values (8 bits, Output)
//				The LCD 'device Busy' flag also uses bit 7 of port A
//				So sometimes this bit has to be set for Input

//				Uses port C for control (register select 'RS' bit 6 Output)
//									(device enable 'ENABLE' bit 7
Output)

//				Uses port G for control (write 'WR' bit 0 Output)
//									(read 'RD' bit 1 Output)
//*********************************************************************************

//Function declarations
void lcd_wait();				//Check if the LCD device is busy, if so wait.
void lcd_WriteFunctionCommand();	//Output the function command in LCDreg to the LCD:
void lcd_ReadFunctionCommand();		//Read the function command from the LCD into LCDreg
void lcd_Clear();					//Clear the LCD display and set the cursor to the
'home' position
void lcd_StandardMode();			//Set LCD to 8-bit-mode, display window freeze and auto
cursor increment
void lcd_SetCursor(unsigned char cursorPosition);	//Set cursor to a specific display position
void lcd_WriteChar(unsigned char cValue);		//Write the char in LCDreg at current cursor
position and increment cursor
void lcd_WriteString(unsigned char []);
void lcd_on();						//Set LCD display on, cursor on and blink on
void lcd_CursorOn();				//Set Cursor on
```

```
void lcd_CursorOff();                           //Set Cursor Off
void lcd_DisplayOn();                           //Set Display on
void lcd_DisplayOff();                          //Set Display off
void lcd_CursorPositionOff();             //Set Cursor Position Indicator Off
void lcd_BarGraph(unsigned char bar1, unsigned char bar2);    //Display 2-bar bar graph on LCD
display
void lcd_ShiftLeft();       // shift LCD display one place to the left
void lcd_ShiftRight();      // shift LCD display one place to the right
void lcd_OneLineMode();     // configure LCD one-line mode
void lcd_TwoLineMode();     // configure LCD two-line mode
void lcd_WriteVariable_withValueAndPositionParameters(unsigned char row, unsigned char column,
unsigned char data); // Display a variable IN RAW format at a user-specified position
void lcd_WriteVariable_withValueAndPositionParameters_SingleDecimalDigit(unsigned char row, unsigned
char column, unsigned char data); // Display a single-digit DECIMAL value (0-9)

// Declare global variables required for the LCD library
unsigned char  LCDreg;

// PORTA control bits
//     BF=7
// PORTC control bits
//     RS=6
//     ENABLE=7
// PORTG control bits  (for 1281)
//     WR=0
//     RD=1

void lcd_Wait()             // Check if the LCD device is busy, if so wait
{
       PORTC &= 0b00111111; // Clear enable (bit 7), clear Register select (bit 6)
                                            // (Set RS low (command register) so can read busy flag)
       PORTG &= 0b11111101; // Clear RD (bit 1) for read operation
       PORTG |= 0b00000001; // Set WR (bit 0) for read operation
       DDRA &= 0b01111111;  // High bit of port A is Busy Flag (BF, bit 7), set to input
       PORTC |= 0b10000000; // Set enable (bit 7)

       while(PINA & 0b10000000); // Wait here until busy flag is cleared

       PORTC &= 0b01111111; // Clear enable (bit 7)
       PORTG &= 0b11111110; // Clear WR (bit 0) for write operation
       PORTG |= 0b00000010; // Set RD (bit 1) for write operation
       DDRA |= 0b10000000;  // Restore bit 7 of port A to output (for Data use)
}

void lcd_WriteFunctionCommand()    // Output the function command in LCDreg to the LCD
{
       lcd_Wait();          // Wait if the LCD device is busy
       PORTC &= 0b10111111; // Clear Register select (bit 6)
       PORTC |= 0b10000000; // Set enable (bit 7)
       PORTA = LCDreg;      // Send function command to command register, via port A
       PORTC &= 0b01111111; // Clear enable (bit 7)
}

void lcd_ReadFunctionCommand()     // Read the function command from the LCD into LCDreg
{
       lcd_Wait();          // Wait if the LCD device is busy
       PORTC &= 0b10111111; // Clear Register select (bit 6)
       PORTC |= 0b10000000; // Set enable (bit 7)
       LCDreg = PORTA;      // Read the command register via port A
       PORTC &= 0b01111111; // Clear enable (bit 7)
}

void lcd_Clear()     // Clear the LCD display and set the cursor to the 'home' position
{
       LCDreg = 0x01;                              // The 'clear' command
       lcd_WriteFunctionCommand(); // Output the command to the LCD
}
```

```
void lcd_StandardMode()     // Set the LCD to 8-bit-mode, display window freeze and
                                    // automatic cursor increment (standard mode)
{
        LCDreg = 0b00111000;                       // Command for 8-Bit-transfer
        lcd_WriteFunctionCommand();        // Output the command to the LCD
        LCDreg = 0b00000110;                       // Command for Increment, display freeze
        lcd_WriteFunctionCommand();        // Output the command to the LCD
        LCDreg = 0b00010000;                       // Command for Cursor move, not shift
        lcd_WriteFunctionCommand();        // Output the command to the LCD
}

void lcd_SetCursor(unsigned char cursorPosition)// Set cursor on the LCD to a specific display
position
{
        LCDreg = cursorPosition | 0b10000000;     // Set bit 7 of the byte
        lcd_WriteFunctionCommand();        // Output the command to the LCD
}

void lcd_WriteChar(unsigned char cValue)  // Write the character in LCDreg to the display at the
current cursor
                                                                              // position (the
position is incremented after write)
{
        lcd_Wait();           // Wait if the LCD device is busy
        PORTC |= 0b11000000;// Set enable (bit 7), Set RS (bit 6) (for data)
        PORTA = cValue;       // Write data character to LCD
        PORTC &= 0b01111111;// Clear enable (bit 7)
}

void lcd_WriteString(unsigned char Text[])
{
        unsigned char Index;
        for(Index = 0;Index < strlen(Text); Index++)
        {
                lcd_WriteChar(Text[Index]);
        }
}

void lcd_on() // Set LCD display on, cursor on and blink on
{
        LCDreg = 0b00001111;                       // Combined command byte
        lcd_WriteFunctionCommand();        // Output the command to the LCD
}

void lcd_CursorOn()// Set Cursor on
{
        lcd_ReadFunctionCommand();  // Input the command register value into the LCDreg
        LCDreg |= 0b00000010;              // Set bit 1 of the position. Command value to set cursor
on
        lcd_WriteFunctionCommand();        // Output the command to the LCD
}

void lcd_CursorOff() // Set Cursor Off
{
        lcd_ReadFunctionCommand();  // Input the command register value into the LCDreg
        LCDreg &= 0b11111101;              // Command value to set cursor off
        lcd_WriteFunctionCommand(); // Output the command to the LCD
}

void lcd_DisplayOn() // Set Display on
{
        lcd_ReadFunctionCommand();  // Input the command register value into the LCDreg
        LCDreg |= 0b00000100;              // Set bit 2 of the position. Command value to set display
on
        lcd_WriteFunctionCommand(); // Output the command to the LCD
}
```

```
void lcd_DisplayOff()       // Set Display off
{
        lcd_ReadFunctionCommand();  // Input the command register value into the LCDreg
        LCDreg &= 0b11111011;               // Command value to set display off
        lcd_WriteFunctionCommand(); // Output the command to the LCD
}

void lcd_CursorPositionOff()        // Set Cursor Position Indicator Off
{
        LCDreg = 0b00001100;                        // Command value to set cursor position indicator
off
        lcd_WriteFunctionCommand();        // Output the command to the LCD
}

void lcd_BarGraph(unsigned char bar1, unsigned char bar2)
// Display 2-bar bar graph on LCD display
// The bar lengths are given by the values in Bar1 and Bar2
{
        unsigned char  BarGraph1 = bar1;
        unsigned char  BarGraph2 = bar2;
        lcd_CursorOff();
        lcd_CursorPositionOff();
        lcd_Clear();
        lcd_SetCursor(0x00);   // Set cursor position to line 1, col 0

        while(BarGraph1 > 0) // Check for 0 bar height
        {
                lcd_WriteChar(0x11); // 0x11 is the best character code for the bar element
                BarGraph1--;
        }

        lcd_SetCursor(0x40);   // Set cursor position to line 2, col 0

        while(BarGraph2 > 0) // Check for 0 bar height
        {
                lcd_WriteChar(0x11); // 0x11 is the best character code for the bar element
                BarGraph2--;
        }
}

void lcd_ShiftLeft()        // Configure LCD shift Left mode
{
        LCDreg = 0b00011000;                        // Command value to set shift Left (see LCD manual
PDF)
        lcd_WriteFunctionCommand();        // Output the command to the LCD
}

void lcd_ShiftRight()       // Configure LCD shift Right mode
{
        LCDreg = 0b00011100;                        // Command value to set shift Right (see LCD manual
PDF)
        lcd_WriteFunctionCommand();        // Output the command to the LCD
}

void lcd_OneLineMode()      // Configure LCD one-line mode
{
        LCDreg = 0b00110100;                        // Command value to set to one-line mode (see LCD
manual PDF)
        lcd_WriteFunctionCommand();        // Output the command to the LCD
}

void lcd_TwoLineMode()      // Configure LCD Two-line mode
{
        LCDreg = 0b00111100;               // Command value to set to two-line mode (see LCD manual
PDF)
        lcd_WriteFunctionCommand();        // Output the command to the LCD
```

```
}

void lcd_WriteVariable_withValueAndPositionParameters(unsigned char row, unsigned char column,
unsigned char data)
{
	// The LCD has 2 rows R = {0 - 1} and 16 columns C = {0 - 15}
	// The cursor position is defined in a single byte as follows: 0b0B00CCCC, so
	// need to mask and shift the passed-in values, before calling lcd_SetCursor()
	column &= 0b00001111;		// Only lower 4 bits are valid, Mask off upper 4 bits
	row &= 0b00000001;		// Only lowest bit is valid, Mask off upper 7 bits
	row *= 64;				// Shift row value 6 places to the left
	lcd_SetCursor(column + row);	// Set the cursor position
	lcd_WriteChar(data); // Write the data byte to the display at current cursor position
}

void lcd_WriteVariable_withValueAndPositionParameters_SingleDecimalDigit(unsigned char row, unsigned
char column, unsigned char data)
{
	// The LCD has 2 rows R = {0 - 1} and 16 columns C = {0 - 15}
	// The cursor position is defined in a single byte as follows: 0b0R00CCCC, so
	// need to mask and shift the passed-in values, before calling lcd_SetCursor()
	column &= 0b00001111;		// Only lower 4 bits are valid, Mask off upper 4 bits
	row &= 0b00000001;		// Only lowest bit is valid, Mask off upper 7 bits
	row *= 64;				// Shift row value 6 places to the left
	lcd_SetCursor(column + row);	// Set the cursor position
	data += '0';			// Add ASCII '0' to the data value (convert a decimal number into
its character representation)
	lcd_WriteChar(data); // Write the data byte to the display at current cursor position
}
```

TWI_Master_For_RTC_ModuleSLAVE_1281_C.h

```
/*******************************************************************************
Project			TWI Master side for reading Real-Time-Clock module (TWI SLAVE)
Target				ATmega1281 on STK300
Program				TWI_Master_For_RTC_ModuleSLAVE_1281_C.c
Author			Richard Anthony
Date			6th October 2013

Fuse settings System clock: Use the internal RC oscillator at 8.0MHz and CKDIV8 fuse programmed,
resulting in 1.0MHz system clock.
				Fuse settings should be: CKSEL = "0010", SUT = "10", CKDIV8 = "0"
				ATmega1281 clock must be set at 1Mhz, otherwise UART clock-rate
configuration must be adjusted

Function		Reads Time / Date values from I2C (TWI) Real-Time-Clock module.
				Sensor type: Dallas Semiconductor DS1307 Real-Time-Clock module (part
number BOB-00099)
				Displays Seconds value on on-board LEDs
				Transmits Time / Date values to PC Hyperterminal, using the
microcontroller's Serial port (USART0)
*******************************************************************************/
// Communication setup of USART (Hyperterminal settings must match these)
//		Tx / Rx rate	Bits per second		9600
//		Data bits						8
//		Parity						None
//		Stop bits						1
//		Flow control				H/W

// PORTS
//				LEDs on PORTB

//				TWI on PortD
//				SCL (Clock)is on Port D bit 0 (need pullup resistor Set)
//				SDA (Data)is on Port D bit 1 (need pullup resistor Set)
```

```
//                             USART on PortE
//                             USART0 RxD is on Port E bit 0 (Input)
//                             USART0 TxD is on Port E bit 1 (Output)



//*****************************************************************
// Real-Time Clock Application logic            START
//*****************************************************************

#define CR 0x0D
#define LF 0x0A

// RTC module register addresses
#define RTC_Address_SECONDS        0x00
#define RTC_Address_MINUTES        0x01
#define RTC_Address_HOURS          0x02
#define RTC_Address_DAY                   0x03
#define RTC_Address_DATE           0x04
#define RTC_Address_MONTH          0x05
#define RTC_Address_YEAR           0x06
#define RTC_Address_CONTROL        0x07

// TWI Status codes (MASTER - R/W are with respect to MASTER)
#define TWI_M_START                0x08
#define TWI_M_Rep_START            0x10
#define TWI_M_SLA_W_ACK            0x18
#define TWI_M_SLA_W_NACK    0x20
#define TWI_M_DATA_Tx_ACK   0x28
#define TWI_M_DATA_Tx_NACK  0x30
#define TWI_M_ARB_Lost             0x38
#define TWI_M_SLA_R_ACK            0x40
#define TWI_M_SLA_R_NACK    0x48
#define TWI_M_DATA_Rx_ACK   0x50
#define TWI_M_DATA_Rx_NACK  0x58

#define TWI_address_RealTimeClock_Module 0x68   // The default address pre-programmed into the RTC
module

#define TWSR_MASK_5MSB_STATUS_BITS 0xF8                // 5 MSBs are the status code
#define TWINT_FLAG_MASK 0b10000000                     // TWCR bit 7 TWINT: TWI Interrupt Flag (low
signifies 'TWI busy').

// void RTC_Module_Initialisation()
// {
// // Each Data Fetch episode causes 10 TWINT FLAG episodes when working correctly
// // Hangs after 2 episodes if sensor not connected correctly
// // Verify behaviour with Hyperterminal connected to serial port
//      TWI_SendSTART();
//      TWI_Send_SLA_W(TWI_address_RealTimeClock_Module);      // Write to Slave
//      TWI_MasterSendDataByte(RTC_Address_SECONDS /*register #0*/);  // ClockHalt (bit7 of register
0) must be set to 0
//      TWI_MasterSendDataByte(0b00000000); // Clear bit 7 CH bit, to ensure the oscillator is
running, Seconds = 0 0
//      TWI_MasterSendDataByte(0b00100000); // Minutes =  2 0
//      TWI_MasterSendDataByte(0b00000000); // 24Hour mode (bit 6 low), Hours = 0 0
//      TWI_MasterSendDataByte(0b00000110); // Day =  (6 designated inn this application to mean
saturday)
//      TWI_MasterSendDataByte(0b00010010); // Date = 1 2th of Month
//      TWI_MasterSendDataByte(0b00010010); // Month = 1 2 (January)
//      TWI_MasterSendDataByte(0b00010101); // Year = 1 5 (2015)
//      TWI_Send_STOP();
// }

void RTC_Module_PointTo_Seconds_Register()
{
// Each Data Fetch episode causes 3 TWINT FLAG episodes when working correctly
```

```
// Hangs after 2 episodes if sensor not connected correctly
// Verify behaviour with Hyperterminal connected to serial port
	TWI_SendSTART();
	TWI_Send_SLA_W(TWI_address_RealTimeClock_Module);	// Write to Slave
	TWI_MasterSendDataByte(RTC_Address_SECONDS /*register #0*/);
	TWI_Send_STOP();
}

void RTC_module_ReadAllSevenDataRegisters()
{
// Each Data Fetch episode causes 9 TWINT FLAG episodes when working correctly
// Hangs after 2 episodes if sensor not connected correctly
// Verify behaviour with Hyperterminal connected to serial port

// Note the RTC_Module_Initialisation() function sets the register pointer in the RTC module to
register 0
// This function call thus reads registers sequentially beginning at register 0
unsigned char uRTC_Address_SECONDS_UNITS;
unsigned char uRTC_Address_SECONDS_TENS;
unsigned char uRTC_Address_MINUTES_UNITS;
unsigned char uRTC_Address_MINUTES_TENS;
unsigned char uRTC_Address_HOURS_UNITS;
unsigned char uRTC_Address_HOURS_TENS;
unsigned char uRTC_Address_DAY;
unsigned char uRTC_Address_DATE_UNITS;
unsigned char uRTC_Address_DATE_TENS;
unsigned char uRTC_Address_MONTH_UNITS;
unsigned char uRTC_Address_MONTH_TENS;
unsigned char uRTC_Address_YEAR_UNITS;
unsigned char uRTC_Address_YEAR_TENS;

	TWI_SendSTART();
	TWI_Send_SLA_R(TWI_address_RealTimeClock_Module);	// Read from specific TWI Slave (the
sensor)

	// Receive first data byte, and send ACK to signal more bytes expected
	TWCR = (1<<TWINT)+(1<<TWEN)+(1<<TWEA);	// Clear TWINT flag by writing a '1' to it
							// TWEA is 0, so automatically sends
NACK after receiving data byte
							// Signal to H/W this device ready to
receive data from prev addressed Slave
	WaitFor_TWINT_FLAG();
	uRTC_Address_SECONDS_UNITS = (TWDR & 0b00001111);
	uRTC_Address_SECONDS_TENS = (TWDR & 0b01110000) / 16;

	// Receive second data byte, and send ACK to signal more bytes expected
	TWCR = (1<<TWINT)+(1<<TWEN)+(1<<TWEA);	// Clear TWINT flag by writing a '1' to it
							// TWEA is 0, so automatically sends
NACK after receiving data byte
							// Signal to H/W this device ready to
receive data from prev addressed Slave
	WaitFor_TWINT_FLAG();
	uRTC_Address_MINUTES_UNITS = (TWDR & 0b00001111);
	uRTC_Address_MINUTES_TENS = (TWDR & 0b01110000) / 16;

	// Receive third data byte, and send ACK to signal more bytes expected
	TWCR = (1<<TWINT)+(1<<TWEN)+(1<<TWEA);	// Clear TWINT flag by writing a '1' to it
							// TWEA is 0, so automatically sends
NACK after receiving data byte
							// Signal to H/W this device ready to
receive data from prev addressed Slave
	WaitFor_TWINT_FLAG();
	uRTC_Address_HOURS_UNITS = (TWDR & 0b00001111);
	uRTC_Address_HOURS_TENS = (TWDR & 0b00110000) / 16;

	// Receive fourth data byte, and send ACK to signal more bytes expected
	TWCR = (1<<TWINT)+(1<<TWEN)+(1<<TWEA);	// Clear TWINT flag by writing a '1' to it
```

```
        // TWEA is 0, so automatically sends NACK after receiving data byte
        // Signal to H/W this device ready to receive data from prev addressed Slave
        WaitFor_TWINT_FLAG();
        uRTC_Address_DAY = TWDR & 0b00000111;

        // Receive fifth data byte, and send ACK to signal more bytes expected
        TWCR = (1<<TWINT)+(1<<TWEN)+(1<<TWEA);    // Clear TWINT flag by writing a '1' to it
        // TWEA is 0, so automatically sends NACK after receiving data byte
        // Signal to H/W this device ready to receive data from prev addressed Slave
        WaitFor_TWINT_FLAG();
        uRTC_Address_DATE_UNITS = (TWDR & 0b00001111);
        uRTC_Address_DATE_TENS = (TWDR & 0b00110000) / 16;

        // Receive sixth data byte, and send ACK to signal more bytes expected
        TWCR = (1<<TWINT)+(1<<TWEN)+(1<<TWEA);    // Clear TWINT flag by writing a '1' to it
        // TWEA is 0, so automatically sends NACK after receiving data byte
        // Signal to H/W this device ready to receive data from prev addressed Slave
        WaitFor_TWINT_FLAG();
        uRTC_Address_MONTH_UNITS = (TWDR & 0b00001111);
        uRTC_Address_MONTH_TENS = (TWDR & 0b00010000) / 16;

        // Receive seventh data byte, and send NACK to signal no more bytes expected
        TWCR = (1<<TWINT)+(1<<TWEN);       // Clear TWINT flag by writing a '1' to it
                                                              // TWEA is 0, so automatically sends
NACK after receiving data byte
                                                              // Signal to H/W this device ready to
receive data from prev addressed Slave
        WaitFor_TWINT_FLAG();
        uRTC_Address_YEAR_UNITS = (TWDR & 0b00001111);
        uRTC_Address_YEAR_TENS = (TWDR & 0b11110000) / 16;

        TWI_Send_STOP();
        displayDateOnLCD(uRTC_Address_DATE_TENS, uRTC_Address_DATE_UNITS, uRTC_Address_MONTH_TENS,
uRTC_Address_MONTH_UNITS, uRTC_Address_YEAR_TENS, uRTC_Address_YEAR_UNITS);
        displayTimeOnLCD(uRTC_Address_HOURS_TENS, uRTC_Address_HOURS_UNITS,
uRTC_Address_MINUTES_TENS, uRTC_Address_MINUTES_UNITS, uRTC_Address_SECONDS_TENS,
uRTC_Address_SECONDS_UNITS);
}

//**************************************************************
// Real-Time Clock Application logic            END
//**************************************************************


//**************************************************************
// TWI functionality       START
//**************************************************************
void TWI_MASTER_SETUP()
{
        // Set TWI Bit Rate Register (TWBR)
        // SCL  (SCLf) = CPU Clock frequency (CPUf) / (16 + (2TWBR * TWPS)) - thus fastest SCL is
System Clock / 16 with TWBR = 0
        // (where TWPS is one of 1,4,16,64)
        // Slave CPUf must be at least 64 SCLf (from errata sheet, 16 in 8535 manual)
        // In a system in which Microcontrollers run at CPUf = 1MHz, an SCLf of 15KHz is ideal
        // The lower rate allows greater bus length and reduces impacts of noise and bus capacitance
        //      For CPUf = 1MHz, TWBR = 25, TWPS = 1 (bit value 00): gives SCLf = 15151.51 Hz
        //      For CPUf = 1MHz, TWBR = 123, 4^TWPS = 4 (bit value 01): gives SCLf = 1KHz
        //      For CPUf = 1MHz, TWBR = 42, 4^TWPS = 1 (bit value 00): gives SCLf = 10KHz
        //      For CPUf = 1MHz, TWBR = 17, 4^TWPS = 1 (bit value 00): gives SCLf = 20KHz
        //      For CPUf = 1MHz, TWBR = 5, 4^TWPS = 1 (bit value 00): gives SCLf = 38.5KHz
        //      For CPUf = 1MHz, TWBR = 0, 4^TWPS = 1 (bit value 00): gives SCLf = 1MHz / 16 = 62.5KHz
        TWBR = 0;     // SCLf = 62.5KHz

        // Set TWI Status Register (TWSR)
        // bit 7,6,5,4,3 TWS: TWI Status (5 Most Significant bits) (Read only)
        // bit 2 reserved
```

```
	// bit 1,0 TWPS1, TWPS0: TWI Prescaler (increases in powers of 4)
	//	00 = Prescaler 1,   01 = Prescaler 4,    10 = Prescaler 16,   11 = Prescaler 64
	TWSR = 0;	// Use prescaler 1

	// TWI Control Register (TWCR)
	// bit 7 TWINT: TWI Interrupt Flag (low signifies 'TWI busy').
	//		Set by HW when TWI has finished current work and expects SW response.
	//		Reset (to '1') in software, should be done at the end of the interrupt handler
	// bit 6 TWEA: TWI Enable Ack Bit. (1 = enable the device to participate, 0 = virtually
disconnect the device).
	// bit 5 TWSTA: TWI Start Condition Bit.
	//		Set to 0 initially.
	//		Set to 1 when the device wants to become the Bus MASTER. Generates START
	//		condition on the bus if it is free. Otherwise waits until a STOP condition is
	//		detected and then generates the START condition.
	//		Must be reset in SW after the START condition has been generated.
	// bit 4 TWSTO: TWI STOP Condition Bit.
	//		Normally set to 0.
	//		MASTER mode: set to 1 generates a STOP condition
	//		SLAVE mode: set to 1 resets TWI bus to a well-defined state - used in error
recovery
	// bit 3 TWWC: TWI Write Collision Flag (Read only)
	//		'1' indicates attempt to write to TWI Data Reg when TWINT is low
	//		(i.e. TWI still busy with previous operation)
	//		Cleared when a subsequent to write to TWI Data Reg occurs when TWINT is high.
	// bit 2 TWEN: TWI Enable Bit (Enables TWI)
	//		Set to 1 TWI enabled, Port I/O alternate functions SCL and SDA enabled
	//		Set to 0 TWI disabled, Port I/O alternate functions SCL and SDA disabled
	// bit 1 reserved
	// bit 0 TWIE: TWI Interrupt Enable
	//		When set to 1, The TWI interrupt request will be activated while TWINT is high
	//		*** MAY NEED TO DISABLE THE INT, IF ITS CONTINUOUSLY GENERATED ***
	TWCR = (1<<TWEA)+(1<<TWEN)+(1<<TWIE);

	// TWI Data Register (TWDR)
	//		For START, 7 MSBs contain Slave Address, LSB = Read (1) / Write (0) flag
	//		No need to initialise this until ready to send a specific data value
}

void TWI_SendSTART() // Implies sender becomes bus Master
{
	TWCR = (1<<TWINT)+(1<<TWEN)+(1<<TWSTA);
	WaitFor_TWINT_FLAG();
	TWI_MASTER_CheckStatusRegisterValueAgainstExpectedValue(TWI_M_START); // Expected Value
passed in
}

void TWI_Send_STOP()
{
	TWCR = (1<<TWINT)+(1<<TWEN)+(1<<TWSTO); // Send STOP condition and release the bus
}

void TWI_Send_SLA_W(char cTWI_7bit_RecipientAddress)
{
	TWDR = (cTWI_7bit_RecipientAddress * 2)/*shift left into upper 7 bits*/ +0/*Write*/;	//
Slave address  = ssss sss + LSB = Write (0)
	TWCR = (1<<TWINT)+(1<<TWEN);
						// Send SLA (Slave Address) + W (Write flag)
	WaitFor_TWINT_FLAG();
	TWI_MASTER_CheckStatusRegisterValueAgainstExpectedValue(TWI_M_SLA_W_ACK); // Expected Value
passed in
}

void TWI_Send_SLA_R(char cTWI_7bit_RecipientAddress)
{
	TWDR = (cTWI_7bit_RecipientAddress * 2)/*shift left into upper 7 bits*/ +1/*Read*/;
	// Slave address  = ssss sss + LSB = Read (1)
```

```
	TWCR = (1<<TWINT)+(1<<TWEN);
						// Send SLA (Slave Address) + R (Read flag)
	WaitFor_TWINT_FLAG();
	TWI_MASTER_CheckStatusRegisterValueAgainstExpectedValue(TWI_M_SLA_R_ACK); // Expected Value 
passed in
}

void TWI_MasterSendDataByte(char cData)
{
	TWDR = cData;					// *** DATA ***
	TWCR = (1<<TWINT)+(1<<TWEN);	// Send Data byte
	WaitFor_TWINT_FLAG();
	TWI_MASTER_CheckStatusRegisterValueAgainstExpectedValue(TWI_M_DATA_Tx_ACK); // Expected Value 
passed in
}

void TWI_MASTER_CheckStatusRegisterValueAgainstExpectedValue(unsigned char cExpectedStatusCode)
{
	unsigned char TWI_SR;
	TWI_SR = (TWSR & TWSR_MASK_5MSB_STATUS_BITS);  // Get status code from TWSR
	if(cExpectedStatusCode != TWI_SR)
	{
		TWI_Send_STOP();
	}
}

void WaitFor_TWINT_FLAG()   // Wait until START condition has been sent
{
	while(!(TWCR & TWINT_FLAG_MASK));  // TWCR bit 7 TWINT: TWI Interrupt Flag (low signifies 
'TWI busy').
	// Wait until it is set to a '1' before proceeding
}
//****************************************************************
// TWI functionality       END
//****************************************************************
//****************************************************************
```

5. Critical evaluation and conclusion

I believe that overall the application has turned out very well, as I managed to make it display the date, time and temperature as initially planned, without the full support of a lab partner.

Unfortunately, I did not implement the initial idea that the temperature should be sampled 3 times spaced 4 seconds apart and the average of the values displayed. The reason for this was that in trying to implement it, it really over complicated my code, and although only taking one sample may be less accurate, by the time I was testing and implementing pieces of code, one sample seemed to be accurate enough for the systems required purpose.

I did make use of my lecturer Richard Anthony's code for use as some of my libraries, but far from seeing this as a bad thing, I think it was the right thing to do rather than rewrite code that already exists. It is after all common practice in programming to make use of libraries and pre-written code. This allowed me more time to concentrate on the problems that hadn't yet been solved and therefore the use of my time more efficient. I did however study the library code very carefully to make sure I understood it, as I believe this to be good practice – How can you use a library function properly if you don't understand exactly what it does?

This is the first C application that I have ever made, and I learned an awful lot about variable types and simply just interfacing with the "bare metal" of the hardware. Previously I had only programmed AVR in assembler and I found C to be a much kinder more intuitive language to use.

One thing that I would like to improve in this application is the fact that I had to make use of a formula to interpolate the values for the temperature based upon the ADCH reading. Although the analog to digital conversion was not a linear process, without the part number or the datasheets for this particular sensor, I was forced to approximate the temperature values based upon a linear model. This means that the room temperature seems a little higher than it should be on the LCD.

In conclusion, I believe that I could have made my application better and perhaps would have had time to implement the temperature samples and the displaying of average value. I've also run out of time when it comes to making the flowcharts and timing diagrams for my program. Despite this though I am pleased with the quality of the application I have produced and how much I have enjoyed solving the problem.