Michael Tangy

ECE 381 – Spring 2015

Lab 7

Programmable Thermostat

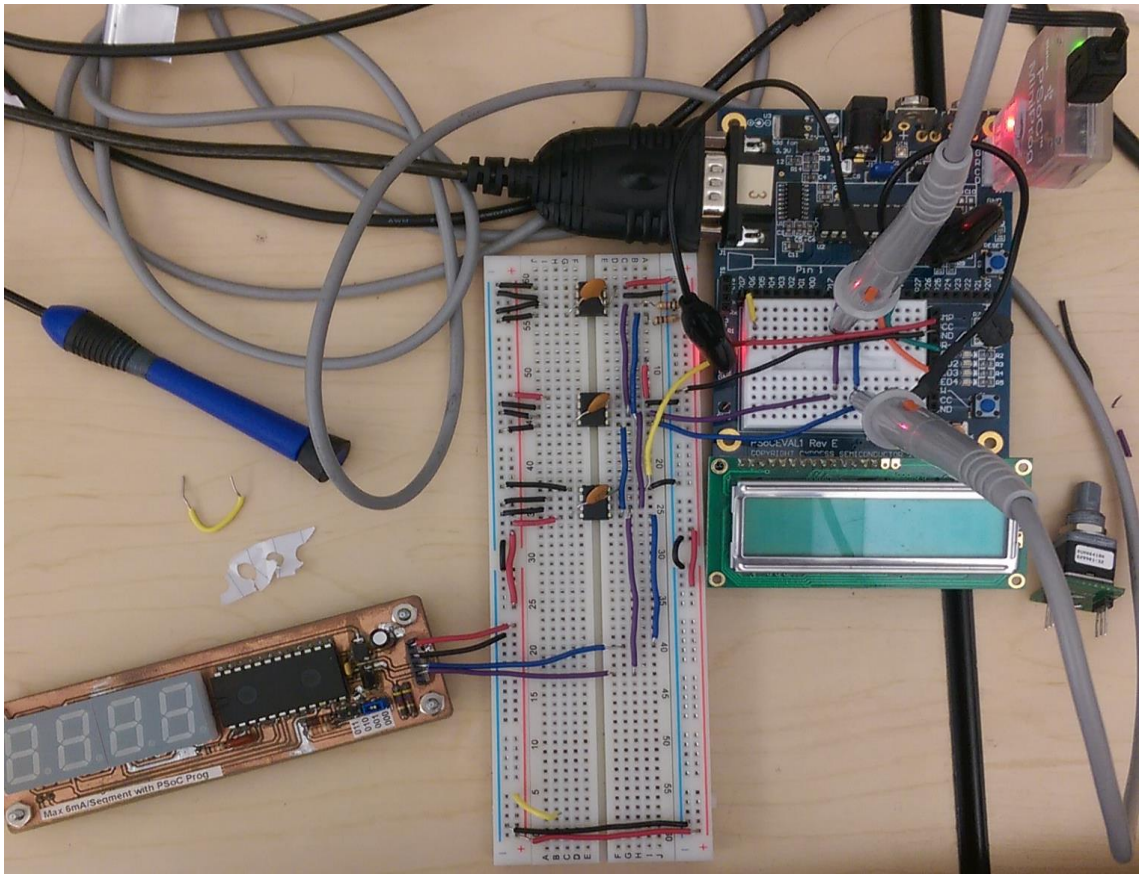
The purpose of this lab was for us to gain further familiarity with the I2C protocol as well as how to use it to communicate/interface with our DS1621 thermostat chip to create a functioning digital thermostat. The thermostat contains a 7-segment display that displays the current temperature as well as the desired temperature set by the user. The thermostat also has an LCD display which tells the user the current mode and operation. All of the digital thermostats setting's will inputted through the PSoC's UART via real term.

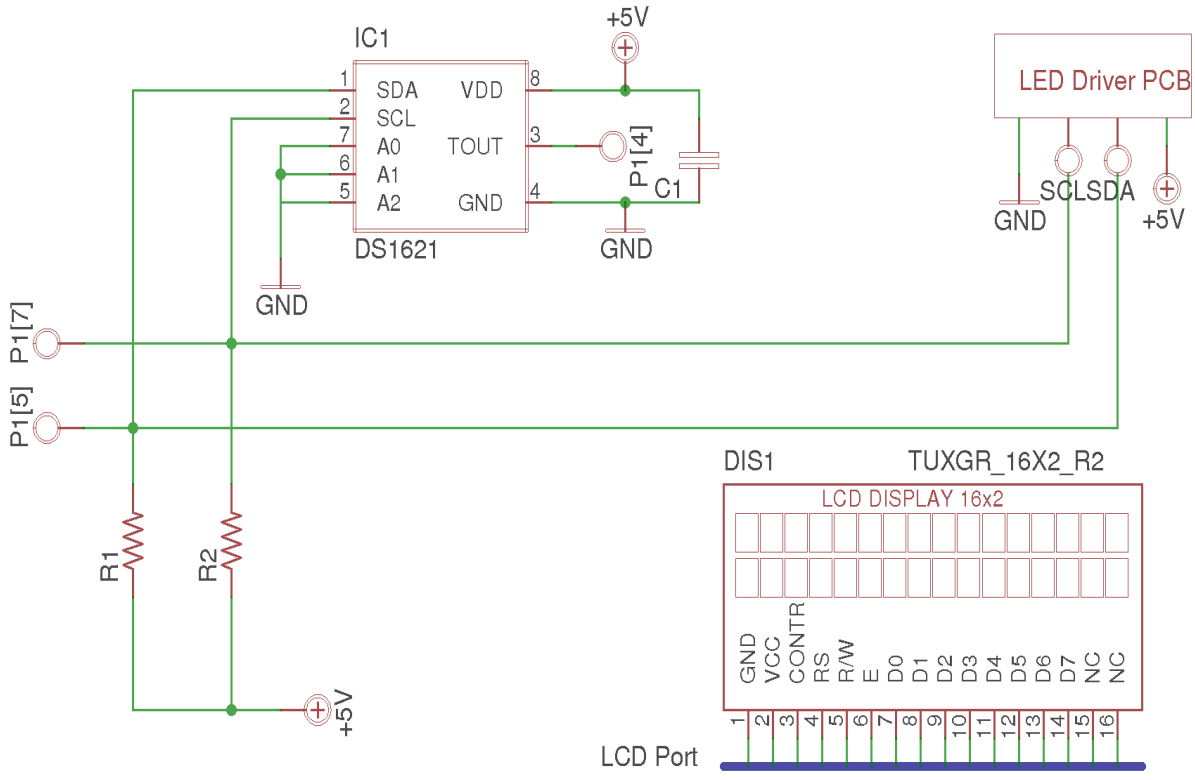
Parts Needed:

1- PSoC Microcontroller
1- DB-9 to DB-9 cable or DB-9 to USB serial cable
1- DS1621 thermostat chip
1- .1μF ceramic capacitor
2- 10KΩ Resistors
1- I2C 7-Segment display board

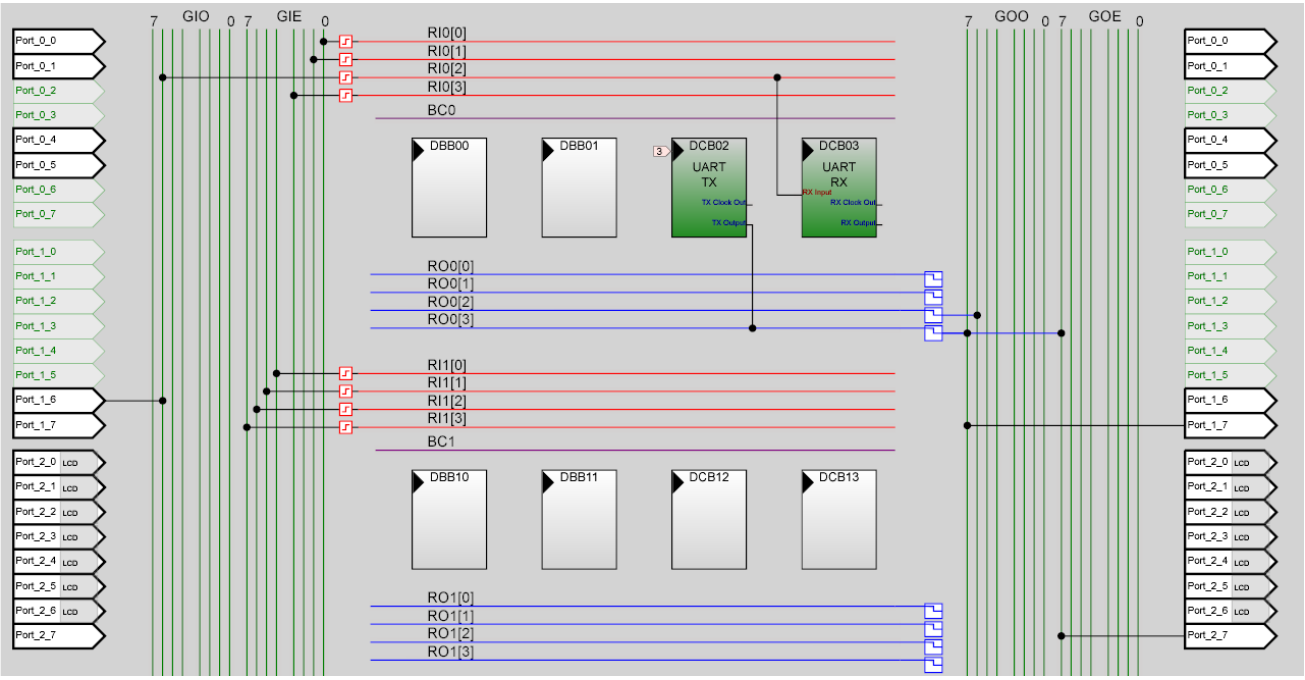
Hardware/Pin-Out Configuration:

After hooking your PSoC up with your computer you need to wire your thermostat chip to your PSoC. This involves setting your group two chip select address with the three address pins on your thermostat chip. These three pins can be set to any combination of power and ground but it is preferred to tie them all to ground. Next tie your power and ground pins to power and ground (pins 4 and 8 respectively). Next you want to connect your SCL and SDA I2C line's to your PSoC they go into port one pin five and port one pin seven respectively. After that you want to add your .1μF ceramic decupling capacitor to the chip, they should be placed in between the power and ground pins, and there leads should be cut as short as possible to reduce noise.

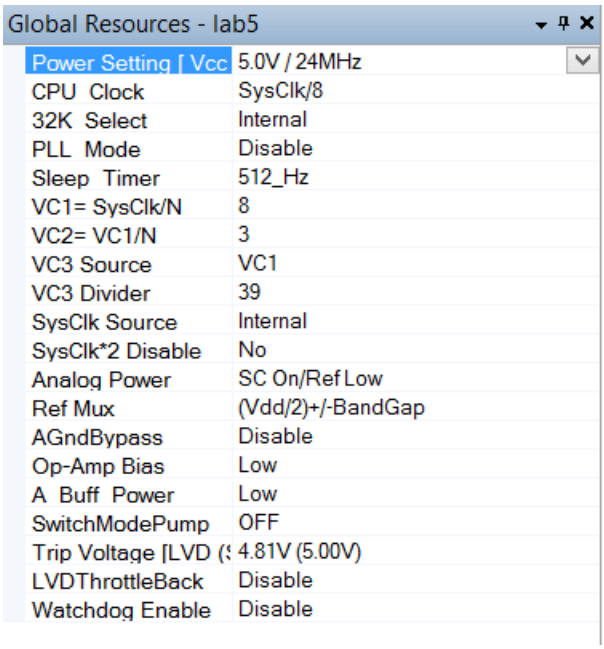

Max 6mA/Segment with PSoC Prog


IC1
SDA
SCL
A0
A1
A2
VDD
TOUT
GND
DS1621
+5V
P1[4]
C1
GND
GND
LED Driver PCB
SCL
SDA
GND
+5V
P1[7]
P1[5]
R1
R2
+5V
DIS1
TUXGR_16X2_R2
LCD DISPLAY 16x2
GND
VCC
CONTR
RS
R/W
E
D0
D1
D2
D3
D4
D5
D6
D7
NC
NC
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
LCD Port

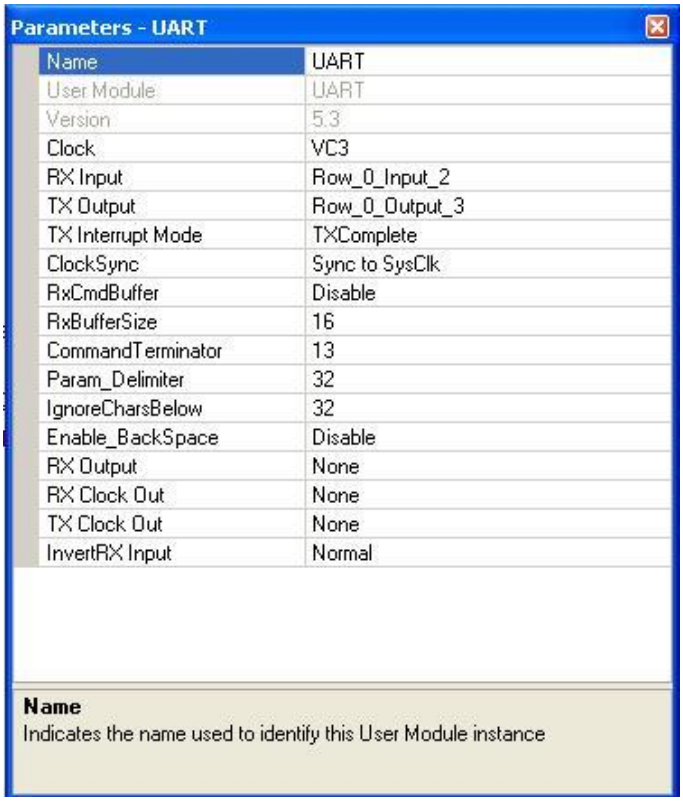
Now you're ready to create your project, clone the previous project to inherit all the UART and I2C settings and configurations. If you chose not to clone the previous project be sure to assign your PSoC's port 1 pin 6 (P1[8]) to the UART's RX line and port 1 pin 7 (P1[7]) to the UART's TX line, also change the variable names at those pins. This allows you to view the binary bit streams you're sending and receiving from your PSoC. This can be done from the chip level view of your project by shift-clicking on the UART modules outputs and assigning them to the appropriate pins. Below is how the chip level view of your project should look.



Next you need to configure a clock source to drive the UART. Since the required baud rate is 9600 we need to set our source clock to three times that (76,800Hz). In the UART data sheet we discovered the best way to do this was to divide the system clock by 8 and 39 and set it to VC3 as shown below:



Next you need to change the properties of your UART module to make sure everything's configured properly. The clock should be set to VC3 and the RxCmdBuffer should be disabled the UART properties window should look like this:



Next you should make sure your LCD is set to port two through its module properties. The Real Term software used in interface with the UART needs to configured with its Baud rate set to 9600 and its port set 1 if your UART is connected to COMA or COM1 or set to port 2 if its connect to COMB or COM2.

Next you'll set the CPU system clock to SysClk/1 then you'll have to add an I2C system module to the chip level view (however the module does not use a digital block so it's not visible) of your project in order to use the appropriate functions from the API. Refer to the tables below to confirm your settings:

User Module
User Module name.

Pinout - lab5	
⊞ P0[0]	Port_0_0, StdCPU, High Z Analog, DisableInt, 0
⊞ P0[1]	Port_0_1, StdCPU, High Z Analog, DisableInt, 0
⊞ P0[2]	Port_0_2, StdCPU, High Z Analog, DisableInt, 0
⊞ P0[3]	Port_0_3, StdCPU, High Z Analog, DisableInt, 0
⊞ P0[4]	Port_0_4, StdCPU, High Z Analog, DisableInt, 0
⊞ P0[5]	Port_0_5, StdCPU, High Z Analog, DisableInt, 0
⊞ P0[6]	Port_0_6, StdCPU, High Z Analog, DisableInt, 0
⊞ P0[7]	Port_0_7, StdCPU, High Z Analog, DisableInt, 0
⊞ P1[0]	Port_1_0, StdCPU, High Z Analog, DisableInt, 0
⊞ P1[1]	Port_1_1, StdCPU, High Z Analog, DisableInt, 0
⊞ P1[2]	Port_1_2, StdCPU, High Z Analog, DisableInt, 0
⊞ P1[3]	Port_1_3, StdCPU, High Z Analog, DisableInt, 0
⊞ P1[4]	Port_1_4, StdCPU, High Z Analog, DisableInt, 0
⊞ P1[5]	I2CHWSDA, I2C SDA, Open Drain Low, DisableInt, 1
⊞ P1[6]	Port_1_6, GlobalInOdd_6, High Z, DisableInt, 0
⊞ P1[7]	I2CHWSCL, I2C SCL, Open Drain Low, DisableInt, 1
⊞ P2[0]	LCDD4, StdCPU, Strong, DisableInt, 0
⊞ P2[1]	LCDD5, StdCPU, Strong, DisableInt, 0
⊞ P2[2]	LCDD6, StdCPU, Strong, DisableInt, 0
⊞ P2[3]	LCDD7, StdCPU, Strong, DisableInt, 0
⊞ P2[4]	LCDE, StdCPU, Strong, DisableInt, 0
⊞ P2[5]	LCDRS, StdCPU, Strong, DisableInt, 0
⊞ P2[6]	LCDRW, StdCPU, Strong, DisableInt, 0
⊞ P2[7]	Port_2_7, GlobalOutEven_7, Strong, DisableInt, 0

Parameters - I2CHW	
Name	I2CHW
User Module	I2CHW
Version	2.00
Read_Buffer_Types	RAM ONLY
I2C Clock	100K Standard
I2C Pin	P[1]5-P[1]7

Software Description:

After configuring the pin-out settings, chip level settings and real term settings you're ready to write your program. All your variables, pointers and constant strings should be declared globally above your main method. Your main method should initialize your strings to null as well as start your UART and set its parity to none as well as all other I2C startup functions. Your main function should also start your LCD and counter and should also enable all necessary interrupts. From there your program should print your initial message (explaining to the user the different commands to operate the thermostat) to the UART's console and read in the character they input, then the program should enter an infinite while loop. You're infinite while loop should have a nested while that allows the program to restart whenever an error flag is thrown. Nested within that while you want to call your read in text function and string parser function which parses the commands and calls functions depending on which command is entered. If the set command is given the program concatenates and converts the two inputted characters to prepare the TH TL registers to be written. If the tolerance command is entered the program computes the two temperature thresholds and puts them in the TH and TL registers. If the mode command is inputted the program reassigns the appropriate mode and flips the POL bit. We had trouble integrating the interrupts with project so weren't able to finish a few parts of the lab including the 7 segment display.

CODE:

```
//-------------------------------------------------------------------------
-
// C main line
//-------------------------------------------------------------------------
-

#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules
#include <string.h>
#include "string.h"
#include "PSoCGPIOINT.h"
#include "stdlib.h"
#include <stdio.h>
#include <stdlib.h>
#include "I2CHWMstr.h"

#pragma interrupt_handler TempReset
#pragma interrupt_handler T_INT


BYTE currentTempA[2];
BYTE currentTemp;
BYTE currentTemp1;
BYTE currentTemp2;
BYTE POLbitFlip[2];

char string0[17];
char string1[17];
char string2[17];
char string3[17];
char* tempP;

char* configOutP;

char TH;
char TL;

char* THP;
char* TLP;

char messageString[79];
int parseCounter=0;
int stringSize=0;
char temp = 0;
int addressInt=0;
char addressHex;
int i;
int j;
char sizeOfString;
char command;


BOOL Interupt = FALSE;
```

```
BOOL hexOrAscii = FALSE;
BOOL finishFlag = FALSE;
BOOL invaildFlag = FALSE;
BOOL readOrWrite = FALSE;
BOOL firstPass = TRUE;
BOOL fanOn;
/* Define slave address */

#define ThermoStat 0x48

BYTE THF = 0x40;
BYTE TLF = 0x20;

/* Define command set */
BYTE tempRead = 0xAA;
BYTE tempStartConvert = 0xEE;
BYTE tempStopConvert = 0x22;
BYTE accessConfigReg = 0xAC;
BYTE accessTL = 0xA2;
BYTE accessTH = 0xA1;

/* Define buffer size */
#define BUFFER_SIZE 0xFF

BYTE txBuffer[BUFFER_SIZE];
BYTE rxBuffer[2];

BYTE THvalue[3];
BYTE TLvalue[3];

BYTE THvalueTest[2];
BYTE TLvalueTest[2];

BYTE status;
BYTE configOut[2];
BYTE configOut2[2];


//int configOutReg;
BYTE* Address_Pointer;

BYTE Read_Address;
BYTE Write_Address;
char temp1;
char temp2;
char temp3;
char charIn = '\0';
char charIn1= '\0';
char charIn2= '\0';
char charIn3= '\0';
char charIn4= '\0';

char instBuffer= '\0';
int placeCounter = 0;
int count;
char tolerence;
BOOL heatOrCool;
```

```
char rawString[79];

BOOL update = FALSE;

const char welcome[] = "Welcome to our Programmable Thermostat!";
const char setTempraturePrompt[] = "To set the temprature enter it in the
following format: 'S ##' (## is the desired temprature in Celsius)";
const char setTolerancePrompt[] = "To set the tolerance enter it in the
following format: 'T #' (# is the width of the temprature tolerance in
Celsius)";
const char setThermModePrompt[] = "To set the thermostat mode enter it it in
the following format: 'M X' (X should be set to C for cool, H for heat and F
for off)";
const char setFanModePrompt[] = "To set the mode of the fan enter it in the
following format: 'F X' (X should be set to A for automatic fan and M for
manual which keeps the fan on all the time)";

//function prototypes
void startFunction (void);
void readFunction(void);
void stringParser(void);
char asciiToInt(char temp1);
void setTolerenceLevel(char THvalue[], char TLvalue[]);
void compareParameters(void);
void cleerLCD(void);
void outputConfig(void);
void changePOLbit(void);
void initial(void);

void main(void)
{
            M8C_EnableIntMask(INT_MSK0,INT_MSK0_GPIO); ///start all modules
and interupts
        I2CHW_Start();
        I2CHW_EnableMstr();
        I2CHW_EnableInt();
        LCD_Start();
        UART_Start(UART_PARITY_NONE);

            M8C_EnableGInt;
            Counter24_Start();
            Counter24_EnableInt();

        firstPass = FALSE;
            invaildFlag = FALSE;
            txBuffer[0] = 0xAA;
            startFunction();
            configOut2[2] = 0x00;

            initial();  // initial I2C finctions starts temp reads and fills
reads config register


        while(!firstPass)
        {

                  while (!invaildFlag){        // error restart loop
```

```
                        readFunction();    //reads in command
                      stringParser();   //parses command and calls
appropriate functions depending on command

                  if (update){    // counter ISR subroutine

                              I2CHW_bWriteBytes(ThermoStat, &tempStartConvert
, 1 , I2CHW_CompleteXfer);  //reads current temp
                           while(!(I2CHW_bReadI2CStatus() &
I2CHW_WR_COMPLETE));
                           I2CHW_ClrWrStatus();

                              I2CHW_bWriteBytes(ThermoStat, &tempRead , 1 ,
I2CHW_NoStop);
                           while(!(I2CHW_bReadI2CStatus() &
I2CHW_WR_COMPLETE));
                           I2CHW_ClrWrStatus();

                              I2CHW_fReadBytes(ThermoStat, currentTempA , 2,
I2CHW_RepStart);
                           while(!(I2CHW_bReadI2CStatus() &
I2CHW_RD_COMPLETE));
                           I2CHW_ClrRdStatus();

                              utoa(tempP, currentTempA[0],10);


                              cleerLCD();        // writes mode to LCD
                              UART_PutCRLF();
                              UART_CPutString("Current Mode:");
                              if (heatOrCool){
                              LCD_Position(0,0);
                              UART_CPutString("Cool");
                              UART_PutCRLF();
                              }else {
                              LCD_Position(0,0);
                              UART_CPutString("Heat");
                              UART_PutCRLF();
                              }
                              UART_PutCRLF();
                              UART_PutCRLF();
                              UART_CPutString("Current temperature:");
//displays current temperature
                              utoa(tempP, currentTempA[0],10);
                              UART_PutString(tempP);

                              if(currentTempA[1] == 0x00){
                              UART_CPutString(".5 C");


                              }else {

                              UART_CPutString(".0 C");
                              }
                              compareParameters();
                              update = FALSE;
```

```
                                if (fanOn){ LED_Fan_Indicator_Data_ADDR ^=
LED_Fan_Indicator_MASK;  // turns fan on if flag is raised
                                }else {LED_Fan_Indicator_Data_ADDR &=
~LED_Fan_Indicator_MASK;}

                           //firstPass = TRUE;
                           }
          }//end invalid input loop

  }//End while(1)

} //End Main

void startFunction (void)
{
      UART_CPutString(welcome);
      UART_PutCRLF();
      UART_PutCRLF();
      UART_CPutString(setThermModePrompt);
      UART_PutCRLF();
      UART_PutCRLF();
      UART_CPutString(setFanModePrompt);
      UART_PutCRLF();
      UART_PutCRLF();
      UART_CPutString(setTolerancePrompt);
      UART_PutCRLF();
      UART_PutCRLF();
      UART_CPutString(setTempraturePrompt);
      UART_PutCRLF();
      UART_PutCRLF();


}//end startFunction

void readFunction (void)
{
          UART_CPutString("\r\n");
            UART_CPutString(">");
        placeCounter = 0;
            finishFlag= FALSE ;

        while (!finishFlag) {
              charIn = UART_cReadChar();
              while (charIn == 0x00){charIn = UART_cReadChar();}
                     if (placeCounter<79){

                   if (charIn == 0x0d){
                              UART_CPutString("\r\n");
                              UART_CPutString("\r\n");
                              finishFlag = TRUE;
                           } //end if CR
                                    else if ((charIn == 0x08 || charIn
==0x7f) && placeCounter > 0 ){ //if backspace
                              placeCounter--;
                              UART_CPutString("\x8\x20\x8");
                                        } //end if backspace
```

```
                                    else {
                                    rawString[placeCounter] = charIn;
                                    UART_PutChar(rawString[placeCounter]);
                                    placeCounter++;
                                }// end else write into string
                    }  else {

                                        UART_PutCRLF();
                                UART_CPutString("String Full");
                                        UART_PutChar(0x07);
                                finishFlag = TRUE;
                                        UART_PutCRLF();

                      }// end placeCounter if

    }//end while (!finsihFlag)

}//end readFunction
void stringParser(void)
{
            if (rawString[2]>= 0x30 && rawString[2]<= 0x39 &&  rawString[3]>=
0x30 && rawString[3]<= 0x39 || rawString[2] == 'h' || rawString[2] == 'c'
){//checks for valid number

                    if(!(rawString[1] ==' ')){

                          invaildFlag = TRUE;
                          UART_PutCRLF();
                          UART_PutCRLF();

                    }// maybe put else

                          if(rawString[0]>= 0x41 && rawString[0]<= 0x5a)
{rawString[0] = rawString[0]+0x20;} //checks if inputted charter & makes it
lower case if it is
                          if(rawString[2]>= 0x41 && rawString[2]<= 0x5a)
{rawString[2] = rawString[2]+0x20;}

                               switch(rawString[0])//Determines if operation
is read or write
                            {
                                    case  's':

                          temp2 = asciiToInt(rawString[2]);
                          temp3 = asciiToInt(rawString[3]);

                          temp2 = temp2 * 10;
                          temp = temp2 + temp3;

                                          outputConfig();
                                  break; //end case r

                                  case 't':

                                          tolerence =
asciiToInt(rawString[2]);
```

```
TH = temp + tolerence/2;
TL = temp - tolerence/2;

THvalue[0] = accessTH;
THvalue[1] = TH;
THvalue[2] = 0x00;

TLvalue[0] = accessTL ;
TLvalue[1] = TL;
TLvalue[2] = 0x00;

setTolerenceLevel(THvalue, TLvalue);
outputConfig();

break; //end case w

case 'm':

if (rawString[2]=='c'){

heatOrCool=TRUE;
//changePOLbit();
}else if(rawString[2]=='h'){

heatOrCool = FALSE;
//changePOLbit();

}else{ fanOn = FALSE;}

break;

case 'f':

break;

default:

UART_PutCRLF();
UART_CPutString("invalid input charecter");
UART_PutCRLF();
invaildFlag = TRUE;

}//end command switch

            }else{

            UART_PutCRLF();
            UART_CPutString("Invalid input number");
            UART_PutCRLF();
            invaildFlag = TRUE;

            }//end of invalid input if-else

}//parse string

void TempReset(void){
            update = TRUE;
```

```
}

void setTolerenceLevel(char THvalue[], char TLvalue[]){

            I2CHW_bWriteBytes(ThermoStat, THvalue , 3 ,I2CHW_CompleteXfer);
          while(!(I2CHW_bReadI2CStatus() & I2CHW_WR_COMPLETE));
          I2CHW_ClrWrStatus();

            I2CHW_bWriteBytes(ThermoStat, TLvalue , 3 ,I2CHW_CompleteXfer);
          while(!(I2CHW_bReadI2CStatus() & I2CHW_WR_COMPLETE));
          I2CHW_ClrWrStatus();

            I2CHW_bWriteBytes(ThermoStat, &accessTH , 1 ,I2CHW_NoStop);
          while(!(I2CHW_bReadI2CStatus() & I2CHW_WR_COMPLETE));
          I2CHW_ClrWrStatus();

            I2CHW_fReadBytes(ThermoStat, THvalueTest , 2, I2CHW_RepStart);
          while(!(I2CHW_bReadI2CStatus() & I2CHW_RD_COMPLETE));
          I2CHW_ClrRdStatus();

            I2CHW_bWriteBytes(ThermoStat, &accessTL , 1 ,I2CHW_NoStop);
          while(!(I2CHW_bReadI2CStatus() & I2CHW_WR_COMPLETE));
          I2CHW_ClrWrStatus();

            I2CHW_fReadBytes(ThermoStat, TLvalueTest , 2, I2CHW_RepStart);
          while(!(I2CHW_bReadI2CStatus() & I2CHW_RD_COMPLETE));
          I2CHW_ClrRdStatus();

              UART_PutCRLF();
              UART_PutCRLF();
              UART_CPutString("TL value:");


              utoa(TLP, TLvalueTest[0],10);
              UART_PutString(TLP);

              UART_PutCRLF();
              UART_PutCRLF();
              UART_CPutString("TH value:");


              utoa(THP, THvalueTest[0],10);
              UART_PutString(THP);

}

void T_INT(void)
{

LED_Data_ADDR &= LED_MASK;

//Interupt = TRUE;

}
char asciiToInt(char temp1){

      if(temp1>= 0x61 && temp1<= 0x66){
```

```
            temp1= temp1-0x20;
      }else

      if (temp1  >= 0x30 && temp1 <= 0x39){

            temp1 = temp1 - 0x30;

      } else if(temp1  >= 0x41 && temp1 <= 0x46){
            temp1 = temp1 - 0x31;

      }
      else{
            return '0';
      }

      return temp1;

}

void compareParameters(void){   // compares current temp with threshold temps
to determine if fan needs to be on

      if(heatOrCool && currentTemp >= TH){

            fanOn = TRUE;

      }else if(!heatOrCool && currentTemp <= TL){

            fanOn = TRUE;
      }
}
void cleerLCD(void){

LCD_Position(0,0);
LCD_PrCString("                ");
LCD_Position(1,0);
LCD_PrCString("                ");

}
void outputConfig(void){  //Outputs config register and performs logic to see
which flags are on


      I2CHW_bWriteBytes(ThermoStat, &accessConfigReg , 1 ,I2CHW_NoStop);
        while(!(I2CHW_bReadI2CStatus() & I2CHW_WR_COMPLETE));
        I2CHW_ClrWrStatus();

            I2CHW_fReadBytes(ThermoStat, configOut , 1, I2CHW_RepStart);
        while(!(I2CHW_bReadI2CStatus() & I2CHW_RD_COMPLETE));
        I2CHW_ClrRdStatus();

            UART_PutCRLF();
            UART_PutCRLF();
            UART_CPutString("Configuration Register:");
            UART_PutCRLF();
            UART_PutCRLF();
```

```
            utoa(configOutP, configOut[0], 2);
            UART_PutString(configOutP);
            UART_PutCRLF();
            UART_PutCRLF();

            if (configOut[0] & THF){
            UART_PutCRLF();
            UART_PutCRLF();
            UART_CPutString("Temp High Flag On!");
            UART_PutCRLF();
            UART_PutCRLF();

            }else if (configOut[0] & TLF){
            UART_PutCRLF();
            UART_PutCRLF();
            UART_CPutString("Temp Low Flag On!");
            UART_PutCRLF();
            UART_PutCRLF();

            }else{UART_CPutString("High Low flag fail");}

}
void changePOLbit(void){  //flips the POL bit when mode is changed

            I2CHW_bWriteBytes(ThermoStat, &accessConfigReg , 1
,I2CHW_NoStop);
        while(!(I2CHW_bReadI2CStatus() & I2CHW_WR_COMPLETE));
        I2CHW_ClrWrStatus();

            I2CHW_fReadBytes(ThermoStat, configOut2 , 1, I2CHW_RepStart);
        while(!(I2CHW_bReadI2CStatus() & I2CHW_RD_COMPLETE));
        I2CHW_ClrRdStatus();

            UART_PutCRLF();
            UART_PutCRLF();
//          UART_PutSHexByte(configOut2[0]);
//          UART_PutCRLF();
//          UART_PutCRLF();

            POLbitFlip[0] = accessConfigReg;
            POLbitFlip[1] = configOut2[0] ^ 0x02;

          I2CHW_bWriteBytes(ThermoStat, POLbitFlip , 2 ,I2CHW_CompleteXfer);
        while(!(I2CHW_bReadI2CStatus() & I2CHW_WR_COMPLETE));
        I2CHW_ClrWrStatus();

}

void initial(void){ //initial function to perform initial temp read and
configuration register read

            I2CHW_bWriteBytes(ThermoStat, &tempStartConvert , 1 ,
I2CHW_CompleteXfer);
        while(!(I2CHW_bReadI2CStatus() & I2CHW_WR_COMPLETE));
        I2CHW_ClrWrStatus();

            I2CHW_bWriteBytes(ThermoStat, &tempRead , 1 , I2CHW_NoStop);
```

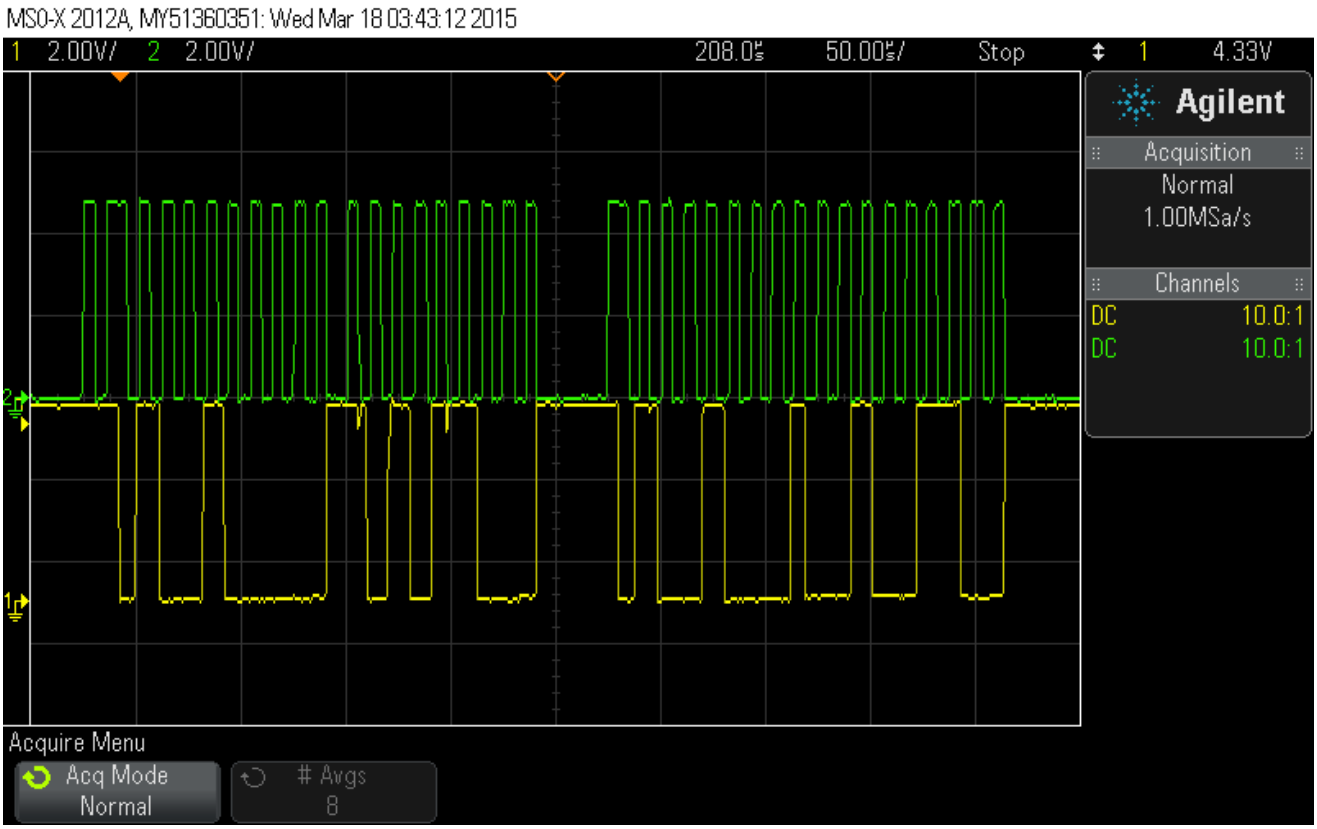
```
        while(!(I2CHW_bReadI2CStatus() & I2CHW_WR_COMPLETE));
        I2CHW_ClrWrStatus();

            j = 0;
            for(j = 0; j < 5000; j++);

            I2CHW_fReadBytes(ThermoStat, currentTempA , 2, I2CHW_RepStart);
        while(!(I2CHW_bReadI2CStatus() & I2CHW_RD_COMPLETE));
        I2CHW_ClrRdStatus();

            I2CHW_bWriteBytes(ThermoStat, &accessConfigReg , 1
,I2CHW_NoStop);
        while(!(I2CHW_bReadI2CStatus() & I2CHW_WR_COMPLETE));
        I2CHW_ClrWrStatus();

            I2CHW_fReadBytes(ThermoStat, configOut , 1, I2CHW_RepStart);
        while(!(I2CHW_bReadI2CStatus() & I2CHW_RD_COMPLETE));
        I2CHW_ClrRdStatus();

            j = 0;
            for(j = 0; j < 5000; j++);
}
```

Testing:

After the program is written is should be built and uploaded to the PSoC. Turn the PSoC on and go to the Real Term console and look for the initial message. Next start typing in commands to confirm everything is working correctly. First enter the set temperature command by typing an “s” and the desired temperature in Celsius. After that enter the temperature tolerance by typing “t” and the desired tolerance in Celsius the program should return the TH and TL register that were written to the chip. After that change the mode and by typing “m” and then either “c” for cool and “h” for hot. You may also want to set the temperature to lower then the TL temperature to test if the fan come on as expected if it does then the LED fan indicator light should turn on. You may also want to measure the current in your I2C lines to see if the right signals are being exchanged between the master and slave. The command prompt window should look similar to the one below during testing.

```
Welcome to our Programmable Thermostat!

To set the thermostat mode enter it it in the following format: 'M X' (X should
be set to C for cool, H for heat and F for off)

To set the mode of the fan enter it in the following format: 'F X' (X should be
set to A for automatic fan and M for manual which keeps the fan on all the time)

To set the tolerance enter it in the following format: 'T #' (# is the width of
the temprature tolerance in Celsius)

To set the temprature enter it in the following format: 'S ##' (## is the desire
d temprature in Celsius)


>s 23


Configuration Register:

1001100


Temp High Flag On!


Current Mode:Heat


Current temperature:25.0 C
>t 2


TL value:22

TH value:24

Configuration Register:

1001100


Temp High Flag On!


Current Mode:Heat


Current temperature:26.5 C
>
m c


Current Mode:Cool


Current temperature:26.5 C
>
```

Your I2C lines should look similar to this during the initial startup when starting the temperature conversions and reading the temperature values.



Below are functions used from the LCD, UART & I2C data sheet's:

I2CHW_fWrite

Description:

Sends a single-byte I²C bus write and ACK. This function does not generate a start or stop condition. This routine should ONLY be called when a previous start and address has been generated on the I2C bus. It should only be used when I2C_BYTE_COMPL is set in the I2C_SCR register.

C Prototype:

```
BYTE I2CHW_fWrite( BYTE bData );
```

Assembler:

```
mov   A,[bRamData]             ; Load data to send to slave
lcall  I2CHW_fWrite             ; Initiate I2C write
```

Parameters:

bData: Byte to be sent to slave.

Return Value:

The return value is nonzero, if the slave acknowledged the master. The return value is zero, if the slave did not acknowledge the master. If the Slave failed to acknowledged the Master, the value of bStatus is 0xff.

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently, only the CUR_PP page pointer register is modified. The I2CHW interrupt is disabled if previously enabled.

I2CHW_bRead

Description:

Initiates a single-byte I²C bus read and ACK phase. This function does not generate a start or stop condition. The fACK flag determines whether the slave is acknowledged upon receiving the data. This routine should ONLY be called when a previous start and address has been generated on the I2C bus. It should only be used when I2C_BYTE_COMPL is set in the I2C_SCR register. If fACK is set, it should be followed by next I2CHW_bRead call. To finish the read transaction Master should call this function with I2CHW_NAKslave parameter.

C Prototype:

```
BYTE I2CHW_bRead( BYTE fACK );
```

Assembler:

```
mov   A,I2CHW_ACKslave        ; Set flag to ACK slave
lcall  I2CHW_bRead             ; Read single byte from slave
                              ; Return data is in reg A
```

Parameters:

fACK: Set to I2CHW_ACKslave if master should ACK the slave after receiving the data; otherwise, flag should be set to I2CHW_NAKslave. In general, the ACK from master means request for the next data byte from the Slave. If set to I2CHW_ACKslave, the master after receiving current data byte and ACKing will immediately clock in the next byte from the slave. This next byte will be returned the next time I2CHW_bRead() is called. If set to I2CHW_NAKslave, the master will only NAK the current byte and will not clock in a the next byte of data.

Return Value:

Byte received from slave.

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions. Currently, only the CUR_PP page pointer register is modified. The I2CHW interrupt is disabled if previously enabled.

UART_Start

Description:

Sets the parity and enables the UART receiver and transmitter. When enabled, data can be received and transmitted.

C Prototype:

```
void  UART_Start(BYTE bParitySetting)
```

Assembly:

```
mov   A, UART_PARITY_NONE
lcall  UART_Start
```

Parameters:

bParitySetting: One byte that specifies the transmit parity. Symbolic names are given in C and assembly, and their associated values are listed in the following table:

TX Parity	Value
UART_PARITY_NONE	0x00
UART_PARITY_EVEN	0x02
UART_PARITY_ODD	0x06

Return Value:

None

Side Effects:

The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_CPutString

Description:

Sends constant (ROM), null terminated string out the UART TX port.

C Prototype:

```
void UART_CPutString(const char * szROMString)
```

Assembler:

```
mov  A,>szRomString   ; Load MSB part of pointer to ROM based null
                      ; terminated string.
mov  X,<szRomString   ; Load LSB part of pointer to ROM based null
                      ; terminated string.
lcall UART_CPutString  ; Call function to send constant string out
                      ; UART TX
```

Parameters:

const char * szROMString: Pointer to string to be sent to the UART. MSB of string pointer is passed in the Accumulator and LSB of the pointer is passed in the X register.

Return Value:

None

Side Effects:

Program flow stays in this function, until the last character is loaded into the UART transmit buffer. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_PutChar

Description:

Writes a single character to the UART TX port when the port buffer is empty.

C Prototype:

```
void UART_PutChar(CHAR cData)
```

Assembler:

```
mov  A,0x33            ; Load ASCII character "3" in A
lcall UART_PutChar      ; Call function to send single character to
                       ; UART TX port.
```

Parameters:

CHAR cData: The character to be sent to the UART TX port. Data is passed in the Accumulator.

Return Value:

None

Side Effects:

Program flow stays in this function until the data can be written to the UART TX buffer. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model (CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.

UART_cReadChar

Description:

Reads UART RX port immediately if data is not available or an error condition exists, or zero is returned. Otherwise, the character is read and returned.

C Prototype:

```
CHAR UART_cReadChar(void)
```

Assembler:

```
lcall UART_cReadChar          ; Call function to read a character
cmp  A,0x00                  ; Check for error
jz   ProcessError            ; If error, Process the error condition
mov  [CharBuffer],A          ; Store retrieved character in buffer
```

Parameters:

None

Return Value:

CHAR bData: Character read from UART RX port. ASCII characters from 1 to 255 are valid. A returned zero signifies an error condition or no data available.

Side Effects:

Function only accepts characters from 1 to 255 as valid. A 0x00 (null) character is detected as an error condition. The A and X registers may be modified by this or future implementations of this function. The same is true for all RAM page pointer registers in the Large Memory Model

(CY8C29xxx and CY8CLED16). When necessary, it is the calling function's responsibility to preserve the values across calls to fastcall16 functions.