```
/*
 * SeniorDesignI2C.c
 *
 * Created: 11/24/2014 6:54:26 PM
 * Author: Robert Bower / Alonzo Ubilla
 * Chip 1 control
 */

#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include "TWI_slave.h"

//#define F_CPU 16000000UL
#define SEI()   sei()
#define SLEEP() sleep_cpu()
#define SAMPLES 10

unsigned char start_cycle = 0x1A;
unsigned char sys_mode;
unsigned char rx_data = 0, tx_data = 0, hex_command = 0, mash_temp = 0, hop_time =
0;
unsigned char boil_temp = 0, variable_data = 0, boil_time = 0, mash_time = 0,
cool_temp = 0;
unsigned char TPUMP_STATUS = 0x00, CPUMP_STATUS = 0x00, HELEMENT_STATUS = 0x00;
uint8_t RAIL_VOLTAGE = 0;
uint8_t MASH_TEMP = 0;
uint8_t FTEMP_MASH = 0;
uint8_t CTEMP_MASH = 0;
uint8_t KETTLE_TEMP2 = 0;
uint8_t KETTLE_TEMP1 = 0;
uint8_t AVG_KET_TEMP = 0;
uint8_t FTEMP_KETTLE = 0;
uint8_t CTEMP_KETTLE = 0;
int step_nmbr = 0;
int step = 0;
int i = 0;
int k = 0;

int _ADCSample(uint8_t chnl);
int read_data(int);
int send_data(int);
void blinky(void);
void temp_control(int);
int evaluate_data(void);
void test_function(void);
void _initADC(void);
void control_loop(int);
```

```
int change_mode(int);

int main(void)
{
  unsigned char TWI_slaveAddress;

  PORTB ^= 0x04;

  //Slave address of chip being programmed
  TWI_slaveAddress = 0x10;

  DDRB = 0xFF;
  DDRD = 0xFF;

  //Initializes the TWI module
  TWI_Slave_Initialise((unsigned
char)((TWI_slaveAddress<<TWI_ADR_BITS)|(TRUE<<TWI_GEN_BIT)));

  SEI();

  //Starts the transceiver
  TWI_Start_Transceiver();

  _initADC();

  for (;;)
  {

        //Main Control Loop.  Code will be inserted here.
        i = read_data(i);

        if(i == 2)
        {
                step = evaluate_data();
                i = 0;
        }

        blinky();

        control_loop(step);
  }
}

int read_data(int i)
{
        if(!TWI_Transceiver_Busy())
        {
                if(TWI_statusReg.RxDataInBuf)
```

```
		{
			switch(i)
			{
				case 0:
				TWI_Get_Data_From_Transceiver(&hex_command, 1);
				i++;
				break;
				case 1:
				TWI_Get_Data_From_Transceiver(&variable_data, 1);
				i++;
				break;
			}
		}
	}

	return i;
}

int evaluate_data(void)
{
	int h = hex_command;

	switch (h)
	{
		//Send start Acknowledge to raspberry pi
		case 0xA0:
			k = 0;
			send_data(k);
			PORTB = 0x02;
			break;

		//Store mash temperature variable
		case 0xA1:
			mash_temp = variable_data;
			break;

		//Store mash time variable
		case 0xA2:
			mash_time = variable_data;
			break;

		//Store boil temperature variable
		case 0xA3:
			boil_temp = variable_data;
			break;

		//Store boil time variable
		case 0xA4:
```

```
        boil_time = variable_data;
        break;

//Store hop time variable
case 0xA5:
        hop_time = variable_data;
        break;

//store cool temperature variable
case 0xA6:
        cool_temp = variable_data;
        break;

//Send back user mash time to pi
case 0xA7:
        k = 1;
        send_data(k);
        break;

//Send back user boil time to pi
case 0xA8:
        k = 2;
        send_data(k);
        break;

//Send back user hop addition time to pi
case 0xA9:
        k = 3;
        send_data(k);
        break;

//Send back kettle temperature
case 0xB0:
        k = 4;
        send_data(k);
        break;

//Send back mash temperature
case 0xB1:
        k = 5;
        send_data(k);
        break;

//Turn off heating element
case 0xB2:
        PORTD = 0x00;
        break;
```

```
		//Turn on heating element
		case 0xB3:
			PORTD = 0x04;
			break;

		//Send user defined mash temperature to pi
		case 0xB4:
			k = 9;
			send_data(k);
			break;

		//Send user defined boil temperature to pi
		case 0xB5:
			k = 10;
			send_data(k);
			break;

		//Send user defined cooled temperature to pi
		case 0xB6:
			k = 11;
			send_data(k);
			break;

		//Read all analog sensors
		case 0xB7:
			RAIL_VOLTAGE = _ADCSample(3);
			MASH_TEMP = _ADCSample(2);
			KETTLE_TEMP2 = _ADCSample(1);
			KETTLE_TEMP1 = _ADCSample(0);
			AVG_KET_TEMP = (((double)KETTLE_TEMP1 +
(double)KETTLE_TEMP2)/2);
			FTEMP_KETTLE =
((double)0.000031*((double)AVG_KET_TEMP*(double)AVG_KET_TEMP*(double)AVG_KET_TEMP)-(
(double)0.012773*(double)AVG_KET_TEMP*(double)AVG_KET_TEMP)+((double)2.193257*(doubl
e)AVG_KET_TEMP)+((double)30.354098));
			CTEMP_KETTLE = ((double)FTEMP_KETTLE -
(double)32)*(double).555556;
			FTEMP_MASH =
((double)0.000031*((double)MASH_TEMP*(double)MASH_TEMP*(double)MASH_TEMP)-((double)0
.012773*(double)MASH_TEMP*(double)MASH_TEMP)+((double)2.193257*(double)MASH_TEMP)+((
double)30.354098));
			CTEMP_MASH = ((double)FTEMP_MASH -
(double)32)*(double).555556;
			break;

		//Initiate step 1 (heat mash water)
		case 0xC0:
			step_nmbr = 1;
```

```
        PORTB = 0x02;
        break;

//Initiate step 2 (transfer mash water)
case 0xC1:
        step_nmbr = 2;
        break;

//Initiate step 3 (transfer sparge water)
case 0xC2:
        step_nmbr = 3;
        break;

//Initiate step 4 (Mash Soak)
case 0xC3:
        step_nmbr = 4;
        break;

//Transfer wort back to kettle
case 0xC4:
        step_nmbr = 5;
        break;

//Begin the boil (turn on heating element)
case 0xC5:
        step_nmbr = 6;
        break;

//Turn on transfer pump(cooling step)
case 0xC6:
        step_nmbr = 7;
        break;

//End of cycle notification
case 0xC7:
        step_nmbr = 8;
        PORTB = 0x04;
        break;

//Trigger actuator for hops addition
case 0xD0:
        //Turn on actuator
        break;

//Send back transfer pump status
case 0xE0:
        k = 6;
        send_data(k);
```

```
        break;

//Send back cooling pump status
case 0xE1:
        k = 7;
        send_data(k);
        break;

//Send back heating element status
case 0xE2:
        k = 8;
        send_data(k);
        break;

//Open hop actuator
case 0xE3:
        PORTD ^= 0x08;
        break;

//Close hop actuator
case 0xE4:
        PORTD = 0x00;
        break;

//Spare
case 0xE5:
        PORTD ^= 0x00;
        break;

//Spare
case 0xE6:

        break;

//Send back wort pump status
case 0xE7:
        k = 18;
        send_data(k);
        break;

//Send back cool pump status
case 0xE8:
        k = 19;
        send_data(k);
        break;

//Receive system mode: Manual or Auto
case 0xF0:
```

```
                        sys_mode = 0x01; //Automatic mode
                        break;

                case 0xF1:
                        sys_mode = 0x02; //Manual mode
                        break;
        }
        return step_nmbr;
}

int send_data(int k)
{
        if(!TWI_Transceiver_Busy())
        {
                switch (k)
                {
                        case 0:
                        TWI_Start_Transceiver_With_Data(&start_cycle, 1);
                        break;

                        case 1:
                        TWI_Start_Transceiver_With_Data(&mash_time, 1);
                        break;

                        case 2:
                        TWI_Start_Transceiver_With_Data(&boil_time, 1);
                        break;

                        case 3:
                        TWI_Start_Transceiver_With_Data(&hop_time, 1);
                        break;

                        case 4:
                        TWI_Start_Transceiver_With_Data(&CTEMP_KETTLE, 1);
                        break;

                        case 5:
                        TWI_Start_Transceiver_With_Data(&CTEMP_MASH, 1);
                        break;

                        case 6:
                        TWI_Start_Transceiver_With_Data(&TPUMP_STATUS, 1);
                        break;

                        case 7:
                        TWI_Start_Transceiver_With_Data(&CPUMP_STATUS, 1);
                        break;
```

```
case 8:
TWI_Start_Transceiver_With_Data(&HELEMENT_STATUS, 1);
break;

case 9:
TWI_Start_Transceiver_With_Data(&mash_temp, 1);

break;

case 10:
TWI_Start_Transceiver_With_Data(&boil_temp, 1);
break;


case 11:
TWI_Start_Transceiver_With_Data(&cool_temp, 1);
break;


case 12:
//TWI_Start_Transceiver_With_Data(&cool_temp, 1);
break;


case 13:
//TWI_Start_Transceiver_With_Data(&cool_temp, 1);
break;


case 14:
//TWI_Start_Transceiver_With_Data(&cool_temp, 1);
break;


case 15:
//TWI_Start_Transceiver_With_Data(&cool_temp, 1);
break;


case 16:
//TWI_Start_Transceiver_With_Data(&cool_temp, 1);
break;


case 17:
//TWI_Start_Transceiver_With_Data(&cool_temp, 1);
break;
```

```
                        case 18:
                        //TWI_Start_Transceiver_With_Data(&cool_temp, 1);
                        break;


                        case 19:
                        //TWI_Start_Transceiver_With_Data(&cool_temp, 1);
                        break;
                }
        }
}

void control_loop(int step)
{
        switch(step)
        {
                //Preheat for mash
                case 1:
                        PORTD = 0x04;
                        HELEMENT_STATUS = 0x01;
                        break;

                //Transfer water to mash tun
                case 2:
                        PORTD = 0x01; //Turn on Transfer Pump
                        HELEMENT_STATUS = 0x00;
                        TPUMP_STATUS = 0X01;
                        break;

                //Transfer water to sparge tank
                case 3:
                        PORTD = 0x01; //Turn on Transfer Pump
                        TPUMP_STATUS = 0x01;
                        break;

                //Start mash (soak grain/timed)
                case 4:
                        PORTD = 0x00;
                        TPUMP_STATUS = 0x00;
                        break;

                //Transfer wort back to kettle
                case 5:
                        //idle for this step
                        break;

                //Begin Boil (timed step)
```

```
                case 6:
                        HELEMENT_STATUS = 0x01;
                        break;

                //Begin cooling step
                case 7:
                        PORTD = 0x03; //Turn on transfer pump and cooler pump
                        HELEMENT_STATUS = 0x00;
                        TPUMP_STATUS = 0x01;
                        CPUMP_STATUS = 0x01;
                        break;

                //Transfer to Carboy
                case 8:
                        PORTD = 0x00; //Turn off the pumps
                        TPUMP_STATUS = 0x00;
                        CPUMP_STATUS = 0x00;
                        break;

                case 9:
                        PORTD = 0x00;
                        break;
        }
}

void blinky(void)
{
        if(TWI_statusReg.RxDataInBuf)
        {
                PORTB ^= 0x01;
        }
}

void temp_control(int setpoint)
{

}

void _initADC(){
        /*
        ADLAR is in the ADMUX Register, bit 5.
        Setting ADLAR = 0 stores 8 MSB's in ADCH (ADC High Byte)
        and the 2 LSB's in bit 7 and 6 of ADCL.
           15    14    13    12    11    10    9     8
        |  -  |  -  |  -  |  -  |  -  |  -  | ADC9| ADC8|              <---ADCH
        | ADC7| ADC6| ADC5| ADC4| ADC3| ADC2| ADC1| ADC0|              <---ADCL
           7     6     5     4     3     2     1     0
```

```
        Setting ADLAR = 1 stores 8 LSB's in ADCL (ADC Low Byte)
        and the 2 MSB's in bit 1 and 0 of ADCH.
           15    14    13    12    11    10    9     8
        | ADC9| ADC8| ADC7| ADC6| ADC5| ADC4| ADC3| ADC2|               <---ADCH
        | ADC1| ADC0|  -
 |  -  |  -  |  -  |  -  |  -  |  -  |               <---ADCL
           7     6     5     4     3     2     1     0
        More info on Pg.265
        */

        ADMUX |= (1<<ADLAR);    //Turn on left adjust so that ADCH contains the 8
High bits of the output.
        //ADMUX &= ~(1<<ADLAR); //Turn on right adjust so that ADCH contains 2 high
bits, ADCL contains 8 low bits.
        ADMUX &= ~(1<<MUX0 | 1<<MUX1 | 1<<MUX2); //Set bits 0, 1, and 2, to 0
enabling ADC0.
        ADMUX &= ~(1<<REFS1);
        ADMUX |= 1<<REFS0;              //Turn off bit 7 and on bit 6 of ADMUX to
enable the AVCC REF.
        ADCSRA |= (1<<ADPS0 | 1<<ADPS1 | 1<<ADPS2);     //Turn on bit 2/1/0 of ADC
Control Register to set prescaler to 128.
        sei();                  //Pg.264 Enable Global Interrupts.
        //ADCSRA |= (1<<ADATE); //Turn on bit 5 of ADC Control Register, ADC Auto
Trigger.
        //ADCSRA |= (1<<ADIF);  //Turn on bit 3 of ADC Control Register, enabling
ADC interrupts.
        ADCSRA |= (1<<ADEN);    //Turn on bit 7 of ADC Control register.  Enable
ADC.
                                                        //Pg.263
        ADCSRA |= (1<<ADSC);    //Turn on bit 6 of ADC Control Register (START
CONVERTING).
        _delay_ms(20);
        return;
}

int _ADCSample(uint8_t chnl){
        uint8_t i = 0;
        uint16_t val = 0;

        chnl &= 0x07;           //Make sure only bits 0, 1, and 2 hold data in
"chnl" variable.
        ADMUX = (ADMUX & 0xF8)|chnl;    //ADMUX & 0xF8 clears bits 0, 1, and 2 on
ADMUX (these are the MUX0,
                                                                //MUX1, and
MUX2, bits that choose which ADC channel.
                                                                //ADMUX is
then OR'd with 'chnl' to set the proper bits for the channel selected.
```

```
        //This for loop will loop through and collect samples (#define SAMPLES at
top), pausing during sata collection.
        //All of the samples are aggregated to variable 'val'.
        for (i = 0; i < SAMPLES; i++) {
                ADCSRA |= (1<<ADSC);                    //Initiate sampling
                while (bit_is_set(ADCSRA, 6));  //Pause during sampling.
                val += ADCH;                                    //Aggregate sample.
        }
        //The samples are then averaged and returned..
        val = val/SAMPLES;
        return val;
}
```