```
#include <avr/eeprom.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/wdt.h>
#include <util/delay.h>
#include <util/twi.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <stdint.h>
#ifndef SERIAL_RINGBUFFER_SIZE
        #define SERIAL_RINGBUFFER_SIZE 256
#endif
#ifndef RS485ADR
   #define RS485ADR 0x02
#endif
#define RS485_DIRECTION_PORT          PORTD
#define RS485_DIRECTION_BIT           0x04
#define RS485_DIRECTION_DDR           DDRD
#ifndef __cplusplus
   typedef unsigned char bool;
   #define true 1
   #define false 0
#endif
#define SHT10_READ_T                0b00000011
#define SHT10_READ_H                0b00000101 //0x05 //0b00000101
#define SHT10_SOFT_RESET            0b00011110 //0x05 //0b00000101
#define SHT10_DELAY_T               3200
#define SHT10_DELAY_H               800 //0x05 //0b00000101
#define SHT10_T                     0
#define SHT10_H                     1
#define SHT10_COM                   0
#define SHT10_HI                    1
#define SHT10_LO                    2
#define SHT10_CRC                   3
#define BME280_TEMP_PRESS_CALIB_DATA_ADDR       UINT8_C(0x88)
#define BME280_HUMIDITY_CALIB_DATA_ADDR         UINT8_C(0xE1)
#define BME280_PWR_CTRL_ADDR                    UINT8_C(0xF4)
#define BME280_CTRL_HUM_ADDR                    UINT8_C(0xF2)
#define BME280_CTRL_MEAS_ADDR                   UINT8_C(0xF4)
#define BME280_CONFIG_ADDR                      UINT8_C(0xF5)
#define BME280_DATA_ADDR                        UINT8_C(0xF7)
#define BME280_TEMP_PRESS_CALIB_DATA_LEN        UINT8_C(26)
#define BME280_HUMIDITY_CALIB_DATA_LEN          UINT8_C(7)
#define BME280_P_T_H_DATA_LEN                   UINT8_C(8)
#define BME280_SLEEP_MODE               UINT8_C(0x00)
#define BME280_FORCED_MODE              UINT8_C(0x01)
#define BME280_REG_OUT_T_L          0xFB
#define BME280_REG_OUT_T_H          0xFA
#define BME280_REG_OUT_T_X          0xFC
#define BME280_REG_OUT_P_L          0xF8
#define BME280_REG_OUT_P_H          0xF7
#define BME280_REG_OUT_P_X          0xF9
#define BME280_REG_OUT_H_L          0xFE
```

```
#define BME280_REG_OUT_H_H              0xFD
#define LIS3DH_REG_STATUS1       0x07
#define LIS3DH_REG_OUTADC1_L     0x08
#define LIS3DH_REG_OUTADC1_H     0x09
#define LIS3DH_REG_OUTADC2_L     0x0A
#define LIS3DH_REG_OUTADC2_H     0x0B
#define LIS3DH_REG_OUTADC3_L     0x0C
#define LIS3DH_REG_OUTADC3_H     0x0D
#define LIS3DH_REG_INTCOUNT      0x0E
#define LIS3DH_REG_WHOAMI        0x0F
#define LIS3DH_REG_TEMPCFG       0x1F
#define LIS3DH_REG_CTRL1         0x20
#define LIS3DH_REG_CTRL2         0x21
#define LIS3DH_REG_CTRL3         0x22
#define LIS3DH_REG_CTRL4         0x23
#define LIS3DH_REG_CTRL5         0x24
#define LIS3DH_REG_CTRL6         0x25
#define LIS3DH_REG_REFERENCE     0x26
#define LIS3DH_REG_STATUS2       0x27
#define LIS3DH_REG_OUT_X_L       0x28
#define LIS3DH_REG_OUT_X_H       0x29
#define LIS3DH_REG_OUT_Y_L       0x2A
#define LIS3DH_REG_OUT_Y_H       0x2B
#define LIS3DH_REG_OUT_Z_L       0x2C
#define LIS3DH_REG_OUT_Z_H       0x2D
#define LIS3DH_REG_FIFOCTRL      0x2E
#define LIS3DH_REG_FIFOSRC       0x2F
#define LIS3DH_REG_INT1CFG       0x30
#define LIS3DH_REG_INT1SRC       0x31
#define LIS3DH_REG_INT1THS       0x32
#define LIS3DH_REG_INT1DUR       0x33
#define LIS3DH_REG_CLICKCFG      0x38
#define LIS3DH_REG_CLICKSRC      0x39
#define LIS3DH_REG_CLICKTHS      0x3A
#define LIS3DH_REG_TIMELIMIT     0x3B
#define LIS3DH_REG_TIMELATENCY   0x3C
#define LIS3DH_REG_TIMEWINDOW    0x3D
#define LIS3DH_REG_ACTTHS        0x3E
#define LIS3DH_REG_ACTDUR        0x3F
#define SENSORS_GRAVITY_EARTH           (9.80665F)              /**< Earth's gravity in m/s^2 */
#define SENSORS_GRAVITY_STANDARD        (SENSORS_GRAVITY_EARTH)

#define LIS3DH_RANGE_16_G 0b11   // +/- 16g
#define LIS3DH_RANGE_8_G 0b10   // +/- 8g
#define LIS3DH_RANGE_4_G 0b01   // +/- 4g
#define LIS3DH_RANGE_2_G 0b00    // +/- 2g (default value)
#define LIS3DH_DATARATE_400_HZ 0b0111 //  400Hz
#define LIS3DH_DATARATE_200_HZ 0b0110 //  200Hz
#define LIS3DH_DATARATE_100_HZ 0b0101 //  100Hz
#define LIS3DH_DATARATE_50_HZ 0b0100 //   50Hz
#define LIS3DH_DATARATE_25_HZ 0b0011 //   25Hz
#define LIS3DH_DATARATE_10_HZ 0b0010 // 10 Hz
#define LIS3DH_DATARATE_1_HZ 0b0001 // 1 Hz
```

```
//adafruit defines for max31865
#define MAX31865_CONFIG_REG           0x00
#define MAX31865_CONFIG_BIAS          0x80
#define MAX31865_CONFIG_MODEAUTO      0x40
#define MAX31865_CONFIG_MODEOFF       0x00
#define MAX31865_CONFIG_1SHOT         0x20
#define MAX31865_CONFIG_3WIRE         0x10
#define MAX31865_CONFIG_24WIRE        0x00
#define MAX31865_CONFIG_FAULTSTAT     0x02
#define MAX31865_CONFIG_FILT50HZ      0x01
#define MAX31865_CONFIG_FILT60HZ      0x00

#define MAX31865_RTDMSB_REG           0x01
#define MAX31865_RTDLSB_REG           0x02
#define MAX31865_HFAULTMSB_REG        0x03
#define MAX31865_HFAULTLSB_REG        0x04
#define MAX31865_LFAULTMSB_REG        0x05
#define MAX31865_LFAULTLSB_REG        0x06
#define MAX31865_FAULTSTAT_REG        0x07


#define MAX31865_FAULT_HIGHTHRESH     0x80
#define MAX31865_FAULT_LOWTHRESH      0x40
#define MAX31865_FAULT_REFINLOW       0x20
#define MAX31865_FAULT_REFINHIGH      0x10
#define MAX31865_FAULT_RTDINLOW       0x08
#define MAX31865_FAULT_OVUV           0x04

//from sg
#define errordelay 750
#define capacityswitch PD6
//#define c2reset PD2
//#define c2wdts PD3

#define TRUE 1
#define FALSE 0
#define SHTDELAY 350
//#define MAXBT 50
#define BTDB 5
#define COOLTOB 0x29 //or 41  COOLTOB * COOLTOF is about 56s
#define COOLTOF 2 / 2 //3 / 2 //1 / 2 //
//#define WDTIMEOUT 0x20 //or 32  about 8s//0x294  //or 660, about 60s
#define MAXRESTARTS 5 // 1 //
#define TRANSMITDELAY 3000 //1300
#define RTD_A 3.9083e-3
#define RTD_B -5.775e-7
#define RTDNOMINAL 100
#define RREF 430.0
#define MAX31865_3WIRE 1
#define ONDELAY 500//8000//1000 //
#define DELAY500MS 5000
#define DELAY300MS 3000
#define DELAY100MS 1000
#define MSGSIZE 22 //22 //25 //22 //18 //22 //51 //20 //30 //40 test
```

```
#define CRCSIZE 2
#define TPCRMAX 20 //1//200
#define TSHTMAX 102//20//205//230 //1//2000
#define TBMEMAX 205//230 //1//2000
#define F_CPU 11059200ul
//#define F_CPU 7372800ul
#define USART_BAUDRATE 38400 //9600
#define UBRR_VALUE (((F_CPU/(USART_BAUDRATE*16UL)))-1)
#define DEVADDR 0x18
#define BME280_I2C_ADDR_PRIM   UINT8_C(0x76)
#define BME280_I2C_ADDR_SEC              UINT8_C(0x77)
struct ringBuffer {
   volatile unsigned long int dwHead;
   volatile unsigned long int dwTail;

   volatile unsigned char buffer[SERIAL_RINGBUFFER_SIZE];
};
//struct ringBuffer lpBuf;
static volatile struct ringBuffer rbTX;
static volatile struct ringBuffer rbRX;
static unsigned char bOwnAddressRS485;
volatile uint8_t boilertemperature;
//uint16_t ElapsedQSeconds = COOLTOB * COOLTOF;
//unsigned char ElapsedQsgSeconds = 0;
uint16_t ElapsedQ0MilliSeconds = 0;
uint16_t ElapsedQ2MilliSeconds = 0;
float Z1;
float Z2;
float Z3;
float Z4;
//uint8_t cycles[] = {0,0,0};
volatile float temperaturertd = 0;
volatile uint8_t cnt = 0;
volatile uint8_t maxabtemp = 180;
volatile uint8_t maxtimer = 41;  //this is actually 56s so ratio is 1.366
volatile uint8_t mcusr = 0;  //this is actually 56s so ratio is 1.366
//uint16_t ElapsedQSeconds = 41;
volatile uint8_t ElapsedQSeconds = 41;
int tpcr = 0;
int tbme = 0;
int tsht = 0;
uint8_t sht10com[] = {SHT10_READ_T,SHT10_READ_H};
volatile uint8_t sht10l[] = {0x26,0x46};
volatile uint8_t sht10h[] = {0x36,0x56};
uint8_t bmeaddr[] = {BME280_I2C_ADDR_PRIM,BME280_I2C_ADDR_SEC};
volatile uint8_t bmetl[] = {0x25,0x45};
volatile uint8_t bmeth[] = {0x35,0x55};
volatile uint8_t bmetx[] = {0x65,0x75};
volatile uint8_t bmepl[] = {0x25,0x45};
volatile uint8_t bmeph[] = {0x35,0x55};
volatile uint8_t bmepx[] = {0x65,0x75};
volatile uint8_t bmehl[] = {0x25,0x45};
volatile uint8_t bmehh[] = {0x35,0x55};
volatile uint8_t xl = 0;
```

```
volatile uint8_t xh = 0;
volatile uint8_t yl = 0;
volatile uint8_t yh = 0;
volatile uint8_t zl = 0;
volatile uint8_t zh = 0;
volatile uint8_t pcr0 = 0;

volatile unsigned char msg[MSGSIZE + CRCSIZE] = {0};

static unsigned char auchCRCSHT1x[] = {
  0x00, 0x31, 0x62, 0x53, 0xc4, 0xf5, 0xa6, 0x97, 0xb9, 0x88, 0xdb, 0xea, 0x7d, 0x4c, 0x1f,
0x2e,
  0x43, 0x72, 0x21, 0x10, 0x87, 0xb6, 0xe5, 0xd4, 0xfa, 0xcb, 0x98, 0xa9, 0x3e, 0x0f, 0x5c,
0x6d,
  0x86, 0xb7, 0xe4, 0xd5, 0x42, 0x73, 0x20, 0x11, 0x3f, 0x0e, 0x5d, 0x6c, 0xfb, 0xca, 0x99,
0xa8,
  0xc5, 0xf4, 0xa7, 0x96, 0x01, 0x30, 0x63, 0x52, 0x7c, 0x4d, 0x1e, 0x2f, 0xb8, 0x89, 0xda,
0xeb,
  0x3d, 0x0c, 0x5f, 0x6e, 0xf9, 0xc8, 0x9b, 0xaa, 0x84, 0xb5, 0xe6, 0xd7, 0x40, 0x71, 0x22,
0x13,
  0x7e, 0x4f, 0x1c, 0x2d, 0xba, 0x8b, 0xd8, 0xe9, 0xc7, 0xf6, 0xa5, 0x94, 0x03, 0x32, 0x61,
0x50,
  0xbb, 0x8a, 0xd9, 0xe8, 0x7f, 0x4e, 0x1d, 0x2c, 0x02, 0x33, 0x60, 0x51, 0xc6, 0xf7, 0xa4,
0x95,
  0xf8, 0xc9, 0x9a, 0xab, 0x3c, 0x0d, 0x5e, 0x6f, 0x41, 0x70, 0x23, 0x12, 0x85, 0xb4, 0xe7,
0xd6,
  0x7a, 0x4b, 0x18, 0x29, 0xbe, 0x8f, 0xdc, 0xed, 0xc3, 0xf2, 0xa1, 0x90, 0x07, 0x36, 0x65,
0x54,
  0x39, 0x08, 0x5b, 0x6a, 0xfd, 0xcc, 0x9f, 0xae, 0x80, 0xb1, 0xe2, 0xd3, 0x44, 0x75, 0x26,
0x17,
  0xfc, 0xcd, 0x9e, 0xaf, 0x38, 0x09, 0x5a, 0x6b, 0x45, 0x74, 0x27, 0x16, 0x81, 0xb0, 0xe3,
0xd2,
  0xbf, 0x8e, 0xdd, 0xec, 0x7b, 0x4a, 0x19, 0x28, 0x06, 0x37, 0x64, 0x55, 0xc2, 0xf3, 0xa0,
0x91,
  0x47, 0x76, 0x25, 0x14, 0x83, 0xb2, 0xe1, 0xd0, 0xfe, 0xcf, 0x9c, 0xad, 0x3a, 0x0b, 0x58,
0x69,
  0x04, 0x35, 0x66, 0x57, 0xc0, 0xf1, 0xa2, 0x93, 0xbd, 0x8c, 0xdf, 0xee, 0x79, 0x48, 0x1b,
0x2a,
  0xc1, 0xf0, 0xa3, 0x92, 0x05, 0x34, 0x67, 0x56, 0x78, 0x49, 0x1a, 0x2b, 0xbc, 0x8d, 0xde,
0xef,
  0x82, 0xb3, 0xe0, 0xd1, 0x46, 0x77, 0x24, 0x15, 0x3b, 0x0a, 0x59, 0x68, 0xff, 0xce, 0x9d,
0xac
};



static unsigned char auchCRCHi[] = {
  0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
  0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
  0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01,
  0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41,
  0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81,
  0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0,
  0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01,
```

```
  0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
  0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
  0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
  0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01,
  0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
  0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
  0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0,
  0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01,
  0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
  0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81,
  0x40
};

static char auchCRCLo[] = {
  0x00, 0xC0, 0xC1, 0x01, 0xC3, 0x03, 0x02, 0xC2, 0xC6, 0x06, 0x07, 0xC7, 0x05, 0xC5,
0xC4,
  0x04, 0xCC, 0x0C, 0x0D, 0xCD, 0x0F, 0xCF, 0xCE, 0x0E, 0x0A, 0xCA, 0xCB, 0x0B, 0xC9,
0x09,
  0x08, 0xC8, 0xD8, 0x18, 0x19, 0xD9, 0x1B, 0xDB, 0xDA, 0x1A, 0x1E, 0xDE, 0xDF, 0x1F,
0xDD,
  0x1D, 0x1C, 0xDC, 0x14, 0xD4, 0xD5, 0x15, 0xD7, 0x17, 0x16, 0xD6, 0xD2, 0x12, 0x13,
0xD3,
  0x11, 0xD1, 0xD0, 0x10, 0xF0, 0x30, 0x31, 0xF1, 0x33, 0xF3, 0xF2, 0x32, 0x36, 0xF6, 0xF7,
  0x37, 0xF5, 0x35, 0x34, 0xF4, 0x3C, 0xFC, 0xFD, 0x3D, 0xFF, 0x3F, 0x3E, 0xFE, 0xFA, 0x3A,
  0x3B, 0xFB, 0x39, 0xF9, 0xF8, 0x38, 0x28, 0xE8, 0xE9, 0x29, 0xEB, 0x2B, 0x2A, 0xEA,
0xEE,
  0x2E, 0x2F, 0xEF, 0x2D, 0xED, 0xEC, 0x2C, 0xE4, 0x24, 0x25, 0xE5, 0x27, 0xE7, 0xE6, 0x26,
  0x22, 0xE2, 0xE3, 0x23, 0xE1, 0x21, 0x20, 0xE0, 0xA0, 0x60, 0x61, 0xA1, 0x63, 0xA3, 0xA2,
  0x62, 0x66, 0xA6, 0xA7, 0x67, 0xA5, 0x65, 0x64, 0xA4, 0x6C, 0xAC, 0xAD, 0x6D, 0xAF,
0x6F,
  0x6E, 0xAE, 0xAA, 0x6A, 0x6B, 0xAB, 0x69, 0xA9, 0xA8, 0x68, 0x78, 0xB8, 0xB9, 0x79,
0xBB,
  0x7B, 0x7A, 0xBA, 0xBE, 0x7E, 0x7F, 0xBF, 0x7D, 0xBD, 0xBC, 0x7C, 0xB4, 0x74, 0x75,
0xB5,
  0x77, 0xB7, 0xB6, 0x76, 0x72, 0xB2, 0xB3, 0x73, 0xB1, 0x71, 0x70, 0xB0, 0x50, 0x90, 0x91,
  0x51, 0x93, 0x53, 0x52, 0x92, 0x96, 0x56, 0x57, 0x97, 0x55, 0x95, 0x94, 0x54, 0x9C, 0x5C,
  0x5D, 0x9D, 0x5F, 0x9F, 0x9E, 0x5E, 0x5A, 0x9A, 0x9B, 0x5B, 0x99, 0x59, 0x58, 0x98, 0x88,
  0x48, 0x49, 0x89, 0x4B, 0x8B, 0x8A, 0x4A, 0x4E, 0x8E, 0x8F, 0x4F, 0x8D, 0x4D, 0x4C,
0x8C,
  0x44, 0x84, 0x85, 0x45, 0x87, 0x47, 0x46, 0x86, 0x82, 0x42, 0x43, 0x83, 0x41, 0x81, 0x80,
  0x40
};

unsigned short crc16 (unsigned char *puchMsg, unsigned short int usDataLen );
static void serialInit();
static void serialHandleData();
static inline void ringBuffer_Init(volatile struct ringBuffer* lpBuf);
static inline unsigned long int ringBuffer_AvailableN(volatile struct ringBuffer* lpBuf);
static unsigned char ringBuffer_ReadChar(volatile struct ringBuffer* lpBuf);
static inline void serialModeRX();
static inline void serialModeTX();
static void ringBuffer_WriteChars(volatile struct ringBuffer* lpBuf,unsigned char*
bData,unsigned long int dwLen);
```

```
static void serialHandleData(){
   unsigned char bLenByte = 8;
   unsigned char bAdrByte;
   unsigned char bCommandByte;
   unsigned char bAvailable;
   bAvailable = ringBuffer_AvailableN(&rbRX);
     wdt_reset();
   if(bAvailable < 2) {
      return;
   }
   bAdrByte = ringBuffer_ReadChar(&rbRX);
   if(bAdrByte == bOwnAddressRS485) {
      ringBuffer_ReadChar(&rbRX); /* skip Length byte since we already know that value */
      bCommandByte = ringBuffer_ReadChar(&rbRX);
      memset(msg,'\0',sizeof(msg));
      switch(bCommandByte) {
         case 0x03:
           {
              ringBuffer_ReadChar(&rbRX); /* skip Length byte since we already know that value
*/
              ringBuffer_ReadChar(&rbRX); /* skip Length byte since we already know that value
*/
              ringBuffer_ReadChar(&rbRX); /* skip Length byte since we already know that value
*/
              ringBuffer_ReadChar(&rbRX); /* skip Length byte since we already know that value
*/
              ringBuffer_ReadChar(&rbRX); /* skip Length byte since we already know that value
*/
              uint16_t bth;
              bth = temperaturertd * 100;
              if(pcr0 >= 256) pcr0 = 0;
              msg[0] = pcr0++; //0x5f; //

/*
              msg[1] = 0x0f; //bth >>8;//0x07; //
              msg[2] = 0x4f; //(uint8_t)bth; //0x8e; //
              msg[3] = 0x00; //cnt; //0x00; //
              msg[4] = 0x7f; //xh; //0x02; //xh; //0x02; //
              msg[5] = 0x80; //xl; //0x80; //xl; //0x80; //
              msg[6] = 0x00; //yh; //0x01; //yh; //0x01; //
              msg[7] = 0x00; //yl; //0x00; //yl; //0x00; //
              msg[8] = 0x3e; //zh; //0x41; //zh; //0x41; //
              msg[9] = 0x40; //zl; //0x80; //zl; //0x80; //
              msg[10] = 0x9d; //bmetl[0]; //0x5f; // this is real data
              msg[11] = 0x7e; //bmeth[0]; //0x5f; //
              msg[12] = 0x00; //bmetx[0]; //0x5f; //
              msg[13] = 0x5e; //bmepl[0]; //0x5f; //
              msg[14] = 0x4f; //bmeph[0]; //0x5f; //
              msg[15] = 0x00; //bmepx[0]; //0x5f; //
              msg[16] = 0xe7; //bmehl[0]; //0x5f; //
              msg[17] = 0x81; //bmehh[0]; //0x5f; //
              msg[18] = 0x19; //sht10l[SHT10_T]; //0x5f; //
              msg[19] = 0x10; //sht10h[SHT10_T]; //0x5f; //
```

```
            msg[20] = 0x77; //sht10l[SHT10_H]; //0x5f; //
            msg[21] = 0x09; //sht10h[SHT10_H]; //0x5f; //
*/



            msg[1] = bth >>8;//0x07; //
            msg[2] = (uint8_t)bth; //0x8e; //
            msg[3] = cnt; //0x00; //
            msg[4] = xh; //0x02; //xh; //0x02; //
            msg[4] = maxabtemp; //0x02; //xh; //0x02; //
            //msg[4] = maxtimer; //0x02; //xh; //0x02; //
            //msg[5] = xl; //0x80; //xl; //0x80; //
            msg[5] = maxtimer; //0x80; //xl; //0x80; //
            //msg[6] = yh; //0x01; //yh; //0x01; //
            msg[6] = mcusr; //0x01; //yh; //0x01; //
            msg[7] = yl; //0x00; //yl; //0x00; //
            msg[8] = zh; //0x41; //zh; //0x41; //
            msg[9] = zl; //0x80; //zl; //0x80; //
            msg[10] = bmetl[0]; //0x5f; // this is real data
            msg[11] = bmeth[0]; //0x5f; //
            msg[12] = bmetx[0]; //0x5f; //
            msg[13] = bmepl[0]; //0x5f; //
            msg[14] = bmeph[0]; //0x5f; //
            msg[15] = bmepx[0]; //0x5f; //
            msg[16] = bmehl[0]; //0x5f; //
            msg[17] = bmehh[0]; //0x5f; //
            msg[18] = sht10l[SHT10_T]; //0x5f; //
            msg[19] = sht10h[SHT10_T]; //0x5f; //
            msg[20] = sht10l[SHT10_H]; //0x5f; //
            msg[21] = sht10h[SHT10_H]; //0x5f; //
            crc16(msg,MSGSIZE); //pmctest
            ringBuffer_WriteChars(&rbTX, msg, sizeof(msg));
            serialModeTX();
           }
    wdt_reset();
          rbRX.dwTail = (rbRX.dwTail + (bLenByte-8)) % SERIAL_RINGBUFFER_SIZE; /*
Compatibility with invalid protocol: Skip any remaining bytes */
          break;
        case 0x04:
          {
            cnt = ringBuffer_ReadChar(&rbRX); /* skip Length byte since we already know that
value */
            eeprom_update_byte((uint8_t*)32,cnt);
            msg[0] = bAdrByte; //0x5f; //
            msg[1] = 0x08; //0x5f; //
            msg[2] = bCommandByte; //0x5f; //
            msg[3] = cnt; //0x5f; //
            msg[4] = ringBuffer_ReadChar(&rbRX); //0x5f; //
            msg[5] = ringBuffer_ReadChar(&rbRX); //0x5f; //
            ringBuffer_ReadChar(&rbRX); /* skip Length byte since we already know that value
*/
            ringBuffer_ReadChar(&rbRX); /* skip Length byte since we already know that value
*/
```

```
            crc16(msg,6); //pmctest
            ringBuffer_WriteChars(&rbTX, msg, 8);
            serialModeTX();
          }
          rbRX.dwTail = (rbRX.dwTail + (bLenByte-8)) % SERIAL_RINGBUFFER_SIZE; /*
Compatibility with invalid protocol: Skip any remaining bytes */
          break;
        case 0x05:
          {
            maxabtemp = ringBuffer_ReadChar(&rbRX); /* skip Length byte since we already
know that value */
            eeprom_update_byte((uint8_t*)33,maxabtemp);
            msg[0] = bAdrByte; //0x5f; //
            msg[1] = 0x08; //0x5f; //
            msg[2] = bCommandByte; //0x5f; //
            msg[3] = maxabtemp; //0x5f; //
            msg[4] = ringBuffer_ReadChar(&rbRX); //0x5f; //
            msg[5] = ringBuffer_ReadChar(&rbRX); //0x5f; //
            ringBuffer_ReadChar(&rbRX); /* skip Length byte since we already know that value
*/
            ringBuffer_ReadChar(&rbRX); /* skip Length byte since we already know that value
*/
            crc16(msg,6); //pmctest
            ringBuffer_WriteChars(&rbTX, msg, 8);
            serialModeTX();
          }
          rbRX.dwTail = (rbRX.dwTail + (bLenByte-8)) % SERIAL_RINGBUFFER_SIZE; /*
Compatibility with invalid protocol: Skip any remaining bytes */
          break;
        case 0x06:
          {
            maxtimer = ringBuffer_ReadChar(&rbRX); /* skip Length byte since we already know
that value */
            eeprom_update_byte((uint8_t*)34,maxtimer);
            msg[0] = bAdrByte; //0x5f; //
            msg[1] = 0x08; //0x5f; //
            msg[2] = bCommandByte; //0x5f; //
            msg[3] = maxtimer; //0x5f; //
            msg[4] = ringBuffer_ReadChar(&rbRX); //0x5f; //
            msg[5] = ringBuffer_ReadChar(&rbRX); //0x5f; //
            ringBuffer_ReadChar(&rbRX); /* skip Length byte since we already know that value
*/
            ringBuffer_ReadChar(&rbRX); /* skip Length byte since we already know that value
*/
            crc16(msg,6); //pmctest
            ringBuffer_WriteChars(&rbTX, msg, 8);
            serialModeTX();
          }
          rbRX.dwTail = (rbRX.dwTail + (bLenByte-8)) % SERIAL_RINGBUFFER_SIZE; /*
Compatibility with invalid protocol: Skip any remaining bytes */
          break;
        case 0x07:
          {
```

```
            cnt = ringBuffer_ReadChar(&rbRX); /* skip Length byte since we already know that
value */
            eeprom_update_byte((uint8_t*)32,cnt);
            maxabtemp = ringBuffer_ReadChar(&rbRX); /* skip Length byte since we already
know that value */
            eeprom_update_byte((uint8_t*)33,maxabtemp);
            maxtimer = ringBuffer_ReadChar(&rbRX); /* skip Length byte since we already know
that value */
            eeprom_update_byte((uint8_t*)34,maxtimer);
            msg[0] = bAdrByte; //0x5f; //
            msg[1] = 0x08; //0x5f; //
            msg[2] = bCommandByte; //0x5f; //
            msg[3] = cnt; //0x5f; //
            msg[4] = maxabtemp; //0x5f; //
            msg[5] = maxtimer; //0x5f; //
            ringBuffer_ReadChar(&rbRX); /* skip Length byte since we already know that value
*/
            ringBuffer_ReadChar(&rbRX); /* skip Length byte since we already know that value
*/
            crc16(msg,6); //pmctest
            ringBuffer_WriteChars(&rbTX, msg, 8);
            serialModeTX();
          }
          rbRX.dwTail = (rbRX.dwTail + (bLenByte-8)) % SERIAL_RINGBUFFER_SIZE; /*
Compatibility with invalid protocol: Skip any remaining bytes */
          break;
        default:
          /* Unknown command */
          rbRX.dwTail = (rbRX.dwTail + (bLenByte-8)) % SERIAL_RINGBUFFER_SIZE;
          break;
    }
  } else {
    /*
      Discard message ...
    */
    //rbRX.dwTail = (rbRX.dwTail + (bLenByte-1)) % SERIAL_RINGBUFFER_SIZE;
    //rbRX.dwTail = (rbRX.dwTail + (8-1)) % SERIAL_RINGBUFFER_SIZE; //8 is MODBUS
message length and 1 is for the address byte that has been read
    rbRX.dwTail = (rbRX.dwTail + (bAvailable-1)) % SERIAL_RINGBUFFER_SIZE; //8 is
MODBUS message length and 1 is for the address byte that has been read
  }
}

static inline void ringBuffer_Init(volatile struct ringBuffer* lpBuf) {
  lpBuf->dwHead = 0;
  lpBuf->dwTail = 0;
}

static inline bool ringBuffer_Available(volatile struct ringBuffer* lpBuf) {
  return (lpBuf->dwHead != lpBuf->dwTail) ? true : false;
}
static inline bool ringBuffer_Writable(volatile struct ringBuffer* lpBuf) {
  return (((lpBuf->dwHead + 1) % SERIAL_RINGBUFFER_SIZE) != lpBuf->dwTail) ? true : false;
}
```

```
static inline unsigned long int ringBuffer_AvailableN(
        volatile struct ringBuffer* lpBuf
) {
    if(lpBuf->dwHead >= lpBuf->dwTail) {
        return lpBuf->dwHead - lpBuf->dwTail;
    } else {
        return (SERIAL_RINGBUFFER_SIZE - lpBuf->dwTail) + lpBuf->dwHead;
    }
}
static inline unsigned long int ringBuffer_WriteableN(
        volatile struct ringBuffer* lpBuf
) {
    return SERIAL_RINGBUFFER_SIZE - ringBuffer_AvailableN(lpBuf);
}
static unsigned char ringBuffer_ReadChar(
        volatile struct ringBuffer* lpBuf
) {
    char t;

    if(lpBuf->dwHead == lpBuf->dwTail) {
        return 0x00;
    }

    t = lpBuf->buffer[lpBuf->dwTail];
    lpBuf->dwTail = (lpBuf->dwTail + 1) % SERIAL_RINGBUFFER_SIZE;

    return t;
}

static void ringBuffer_WriteChar(
    volatile struct ringBuffer* lpBuf,
    unsigned char bData
) {
    if(((lpBuf->dwHead + 1) % SERIAL_RINGBUFFER_SIZE) == lpBuf->dwTail) {
        return; /* Simply discard data */
    }

    lpBuf->buffer[lpBuf->dwHead] = bData;
    lpBuf->dwHead = (lpBuf->dwHead + 1) % SERIAL_RINGBUFFER_SIZE;
}

static void ringBuffer_WriteChars(
    volatile struct ringBuffer* lpBuf,
    unsigned char* bData,
    unsigned long int dwLen
) {
    unsigned long int i;

    for(i = 0; i < dwLen; i=i+1) {
        ringBuffer_WriteChar(lpBuf, bData[i]);
    }
}

static inline void serialModeRX() {
```

```
    /*
       Set to receive mode on RS485 driver
       Toggle receive enable bit on UART, disable transmit enable bit
    */
          //uint8_t oldSREG = SREG;
          cli();

    RS485_DIRECTION_PORT = RS485_DIRECTION_PORT & (~RS485_DIRECTION_BIT);
    UCSR0B = (UCSR0B & (~0xE8)) | 0x10 | 0x80; /* Disable all transmit interrupts, enable
receiver, enable receive complete interrupt */
    //PORTD = PORTD & (~(0x08)); /* Set RE and DE to low (RE: active, DE: inactive) */
    //UCSR0B = (UCSR0B & (~0xE8)) | 0x10 | 0x80; /* Disable all transmit interrupts, enable
receiver, enable receive complete interrupt */
    sei();
    return;

          //SREG = oldSREG;
}

static inline void serialModeTX() {
    /*
       Set to transmit mode on RS485 driver
       and toggle transmit enable bit in UART
    */
          //uint8_t oldSREG = SREG;
          cli();

    RS485_DIRECTION_PORT = RS485_DIRECTION_PORT | RS485_DIRECTION_BIT;
    UCSR0A = UCSR0A | 0x40; /* Clear TXCn */
    UCSR0B = (UCSR0B & (~0x90)) | 0x08 | 0x20 | 0x40; /* Enable UDRE interrupt handler,
enable transmitter and disable receive interrupt & receiver */
    //PORTD = PORTD | 0x08; /* Set RE and DE to high (RE: inactive, DE: active) */
    //UCSR0B = (UCSR0B & (~0x90)) | 0x08 | 0x20; /* Enable UDRE interrupt handler, enable
transmitter and disable receive interrupt & receiver */
        sei();
          //SREG = oldSREG;

    return;
}

unsigned short crc16 (unsigned char *puchMsg, unsigned short int usDataLen )
{
  unsigned char uchCRCHi = 0xFF ;
  unsigned char uchCRCLo = 0xFF ;
  unsigned uIndex ;
  while (usDataLen--)
  {
    uIndex = uchCRCLo ^ *puchMsg++ ;
    uchCRCLo = uchCRCHi ^ auchCRCHi[uIndex] ;
    uchCRCHi = auchCRCLo[uIndex] ;
  }
  if(*puchMsg++ == uchCRCLo && *puchMsg == uchCRCHi)
  {
    return TRUE;
```

```
  }
  else
  {
    if(*puchMsg--);
    *puchMsg++ = uchCRCLo;
    *puchMsg = uchCRCHi;
  }
  return FALSE ;
}

/*

void WDT_off(void)
{
   __disable_interrupt();
   __watchdog_reset();
   /* Clear WDRF in MCUSR */
   //MCUSR &= ~(1<<WDRF);
   /* Write logical one to WDCE and WDE */
   /* Keep old prescaler setting to prevent unintentional time-out */
   //WDTCSR |= (1<<WDCE) | (1<<WDE);
   /* Turn off WDT */
   //WDTCSR = 0x00;
   //__enable_interrupt();
//}

unsigned char reverse(unsigned char b) {
  b = (b & 0xF0) >> 4 | (b & 0x0F) << 4;
  b = (b & 0xCC) >> 2 | (b & 0x33) << 2;
  b = (b & 0xAA) >> 1 | (b & 0x55) << 1;
  return b;
}

uint8_t shtcrccheck(uint8_t data[])
{
uint8_t crc = 0; //this must change for non default settings
int i;
  for (i=0; i<3; i++)
  {
    crc ^= data[i];
    crc = auchCRCSHT1x[crc];
  }
  crc = reverse(crc);
  if(crc == data[3])
  {
    return 1;
  }
  return 0;
}

void resettimer(void)
{
  ElapsedQ0MilliSeconds = 0;
  //ElapsedQSeconds = COOLTOB * COOLTOF;
```

```
  ElapsedQSeconds = maxtimer;
  eeprom_update_byte((uint8_t*)35,ElapsedQSeconds);
}

uint8_t spixfer(uint8_t data)
{
  SPDR = data;
  while(!(SPSR & (1<<SPIF) ))
  {
  }
  return SPDR;
}

uint8_t shiftIn(void)
{
uint8_t value = 0;
uint8_t bit = 0;
uint8_t i;
  for (i = 0; i < 8; i++)
  {
    PORTC |= (1 << PC5);
    _delay_us(SHTDELAY);
    bit = PINC & (1 << PC4);
    if(bit == 0x10)
    {
      value |= (1 << (7 - i));
    }
    PORTC &= ~(1 << PC5);
    _delay_us(SHTDELAY);
  }
  return value;
}

void shiftOut(uint8_t val)
{
  uint8_t i;
  uint8_t mask = 0;
  uint8_t kind = 0;
  uint8_t oldSREG = SREG;
  cli();
  for (i = 0; i < 8; i++)
  {
    mask = 0;
    kind = !!(val & (1 << (7 - i)));
    mask |= (1 << PC4);
    if(kind)
    {
      PORTC |= mask;
    }
    else
    {
      PORTC &= ~mask;
    }
```

```
      _delay_us(SHTDELAY);
      PORTC |= (0x01 << PC5);
      _delay_us(SHTDELAY);
      PORTC &= ~(0x01 << PC5);
      _delay_us(SHTDELAY);
    }
    SREG = oldSREG;
}

void sht_start(void)
{
   DDRC |= (1<<PC4)|(1 << PC5);
   PORTC |= (1 << PC4);
   PORTC |= (1 << PC5);
   _delay_us(SHTDELAY);
   PORTC &= ~(1 << PC4);
    _delay_us(SHTDELAY);
   PORTC &= ~(1 << PC5);
    _delay_us(SHTDELAY);
   PORTC |= (1 << PC5);
    _delay_us(SHTDELAY);
   PORTC |= (1 << PC4);
    _delay_us(SHTDELAY);
   PORTC &= ~(1 << PC5);
   _delay_us(SHTDELAY);
   PORTC &= ~(1 << PC4);
    _delay_us(SHTDELAY);
}

void twi_write(uint8_t data)
{
    //Load SLA_W into TWDR Register. Clear TWINT bit in
    //TWCR to start transmission of address
    TWDR = data;
    TWCR = (1<<TWINT) | (1<<TWEN);
    //Wait for TWINT Flag set. This indicates that the SLA+W has
    //been transmitted, and ACK/NACK has been received.
    while (!(TWCR & (1<<TWINT)));
}

uint8_t twi_readack()
{
    //TWI Interrupt Flag, TWI Enable Bit, TWI Enable Acknowledge Bit
    TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWEA);
    //Wait for TWINT Flag set.
    while (!(TWCR & (1<<TWINT)));
    return TWDR;
}

//void twi_start(uint8_t address)
int8_t twi_start(uint8_t address)
{
    // From datasheet page 226
   TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN); // Send start condition
```

```
   while (!(TWCR & (1<<TWINT))); // Wait for TWINT Flag set. This indicates that the START
condition has been transmitted
   //if(address != SHT10_READ_T && address != SHT10_READ_H) twi_write((address << 1));
   twi_write((address << 1));
   //while (!(TWCR & (1<<TWINT))); // Wait for TWINT Flag set. This indicates that the START
condition has been transmitted
   if((TWSR & 0xF8) != 0x18)
   {
      twi_stop();
      return -1;
   }
   return 0;
}

void twi_rep_start(uint8_t address)
{
    // From datasheet page 226
   TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN); // Send start condition
   while (!(TWCR & (1<<TWINT))); // Wait for TWINT Flag set. This indicates that the START
condition has been transmitted
   twi_write((address << 1) | 1);
}

void twi_stop()
{
    TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWSTO); // Transmit STOP condition
   _delay_ms(1); // Allow time for stop to send
}

uint8_t twi_readnack()
{
    //TWI Interrupt Flag, TWI Enable Bit
    TWCR = (1<<TWINT)|(1<<TWEN);
    //Wait for TWINT Flag set.
    while (!(TWCR & (1<<TWINT)));
    return TWDR;
}

uint8_t readbyte(uint8_t devaddress, uint8_t address)
{
   uint8_t data = 0;
   twi_start(devaddress); // Starts TWI
   //if(!twi_start(devaddress))
   //{
 //if (TWIReadStatus() != 0x08)
        //return -1;
   //send address
    twi_write(address);
    //if (TWIReadStatus() != 0x28)
        //return -1;
    // Send Start
    twi_rep_start(devaddress);
   //if (TWIReadStatus() != 0x10)
        //return -1;
```

```
    //select device and send read bit
    //twi_write(address|1);
    //if (TWIReadStatus() != 0x40)
        //return -1;
    //Reads
    //uint8_t data = twi_readnack();
    data = twi_readnack();
     //if (TWIReadStatus() != 0x58)
        //return -1;
   //}
    twi_stop();
    return data;
}

void writectlregstwi(void)
{
   twi_start(DEVADDR);
   //if(!twi_start(DEVADDR)) //this has to happen so removed the condition
   //{
   twi_write(LIS3DH_REG_CTRL1);
   twi_write(0b01000111);
   twi_stop();
   //these cntrol registers must be written separetely to prevent overwriting so start again
   twi_start(DEVADDR);
     //readbyte(DEVADDR,LIS3DH_REG_CTRL1);
   twi_write(LIS3DH_REG_CTRL4);
   twi_write(0b00001000);
     //readbyte(DEVADDR,LIS3DH_REG_CTRL4);
   //}
   twi_stop();
   //readbyte(DEVADDR,LIS3DH_REG_CTRL1);
   //readbyte(DEVADDR,LIS3DH_REG_CTRL4);
}

uint8_t shtgetdata(void)
{
   DDRC &= ~(1<<PC4);
   _delay_us(SHTDELAY);
   return shiftIn();
}

void shtsendack(void)
{
   DDRC |= (1<<PC4)|(1<<PC5);
   _delay_us(SHTDELAY);
   PORTC &= ~(1 << PC4);
   _delay_us(SHTDELAY);
   PORTC |= (1 << PC5);
   _delay_us(SHTDELAY);
   PORTC &= ~(1 << PC5);
   _delay_us(SHTDELAY);
}

uint8_t writemeasurecmd2sht(int cmd)
```

```
{
  sht_start();
  shiftOut(cmd);
  PORTC |= (1 << PC5);
  DDRC &= ~(1<<PC4);
  _delay_us(SHTDELAY);

  int ack = PINC & (1 << PC4);
  _delay_us(SHTDELAY);
  if (ack != 0)
  {
    return 0;
  }
  PORTC &= ~(1 << PC5);
  return 1;
}

int8_t writestartmeasbme280twi(int addr)
{
  uint8_t data = readbyte(addr,BME280_CTRL_HUM_ADDR);
  if(!twi_start(addr))
  {
    twi_write(BME280_CTRL_HUM_ADDR);
    twi_write(data | 0b00000001); //set h oversampling to x1
    twi_write(BME280_CTRL_MEAS_ADDR);
    twi_write(0b00100100); //set t and p oversampling to x1, sleep mode
    twi_write(BME280_CTRL_MEAS_ADDR);
    twi_write(0b00100101); //set t and p oversampling to x1, forced mode
    twi_stop();
    return 0;
  }
  else
  {
    twi_stop();
    return -1;
  }
}
void writeadg715(void)
{
  if(!twi_start(0x48))
  {
    twi_write(0xff);
    twi_stop();
  }
}
void readxyztwi(void)
{
  xl = readbyte(DEVADDR,LIS3DH_REG_OUT_X_L);
  xh = readbyte(DEVADDR,LIS3DH_REG_OUT_X_H);
  yl = readbyte(DEVADDR,LIS3DH_REG_OUT_Y_L);
  yh = readbyte(DEVADDR,LIS3DH_REG_OUT_Y_H);
  zl = readbyte(DEVADDR,LIS3DH_REG_OUT_Z_L);
  zh = readbyte(DEVADDR,LIS3DH_REG_OUT_Z_H);
}
```

```
void readbshttwi(void)
{
uint8_t crcarray[4] = {0};
int i;
  TWCR &= ~(1<<TWEN);
  for(i=0;i<2;i++)
  {
    if(writemeasurecmd2sht(sht10com[i]))
    {
      if(i)
      {
        _delay_ms(SHT10_DELAY_H);
      }
      else
      {
        _delay_ms(SHT10_DELAY_T);
      }
      crcarray[SHT10_HI]  = shtgetdata();
      shtsendack();
      crcarray[SHT10_LO] = shtgetdata();
      shtsendack();
      crcarray[SHT10_CRC] = shtgetdata();
      shtsendack();
      crcarray[SHT10_COM] = sht10com[i];
      if(shtcrccheck(crcarray))
      {
        sht10h[i] = crcarray[SHT10_HI];
        sht10l[i] = crcarray[SHT10_LO];
      }
      else
      {
        sht_start();
        shiftOut(SHT10_SOFT_RESET);
        _delay_us(SHTDELAY);
      }
    }

  }
  TWCR |= (1<<TWEN);
}

void readbmetwi(void)
{
  int i;
  for(i=0;i<1;i++)
  {
    if(!writestartmeasbme280twi(bmeaddr[i]))
    {
    _delay_ms(320); //this needs correction for more data
    bmetl[i] = readbyte(bmeaddr[i],BME280_REG_OUT_T_L); //this is real data and must be
uncommented
    bmeth[i] = readbyte(bmeaddr[i],BME280_REG_OUT_T_H);
    bmetx[i] = readbyte(bmeaddr[i],BME280_REG_OUT_T_X);
```

```
      bmepl[i] = readbyte(bmeaddr[i],BME280_REG_OUT_P_L);
      bmeph[i] = readbyte(bmeaddr[i],BME280_REG_OUT_P_H);
      bmepx[i] = readbyte(bmeaddr[i],BME280_REG_OUT_P_X);
      bmehl[i] = readbyte(bmeaddr[i],BME280_REG_OUT_H_L);
      bmehh[i] = readbyte(bmeaddr[i],BME280_REG_OUT_H_H);
      }
    }
}

uint8_t readRegister8(uint8_t addr)
{
  uint8_t ret = 0;
  readRegisterN(addr, &ret, 1);
  return ret;
}

uint16_t readRegister16(uint8_t addr)
{
  uint8_t buffer[2] = {0, 0};
  readRegisterN(addr, buffer, 2);
  int16_t ret = buffer[0];
  ret <<= 8;
  ret |=  buffer[1];
  return ret;
}

void readRegisterN(uint8_t addr, uint8_t buffer[], uint8_t n)
{
  SPCR |= (1<<CPHA);
  addr &= 0x7F; // make sure top bit is not set
  PORTD &= ~(0x01 << PD5);
  spixfer(addr);
  while (n--)
  {
    buffer[0] = spixfer(0xFF);
    buffer++;
  }
  PORTD |= (0x01 << PD5);
  SPCR &= ~(1<<CPHA);
}

void writeRegister8(uint8_t addr, uint8_t data)
{
  SPCR |= (1<<CPHA);
  addr |= 0x80; // make sure top bit is set
  PORTD &= ~(0x01 << PD5);
  spixfer(addr);
  spixfer(data);
  PORTD |= (0x01 << PD5);
  SPCR &= ~(1<<CPHA);
}

void acceleration_init(void)
{
```

```
   writectlregstwi();
}

void temperature_init(void)
{
      setWires();
      enableBias(0);
      autoConvert();
      clearFault();
      DDRD |= (1<<PD7)|(1<<PD5);
}

void  twi_init(void)
{
    // Set up TWI module to 27.9KHz
    // SCLfreq = Fclock/(16+2*TWBR*(prescaler))
    // Set up TWI but rate to 2
    //TWBR = 4;
    //TWBR = 0;
    TWBR = 2;
    // Prescaler to 64
    //TWSR |= (1<<TWPS1) | (1<<TWPS0);
    //TWSR |= (1<<TWPS0);
    TWSR |= (1<<TWPS1);
    // Enable TWI
    TWCR |= (1<<TWEN);

}

void spi_init_master (void)
{
   // Set SCK, MOSI, SS' as Output
   //DDRB |= (1<<PB5)|(1<<PB3)|(1<<PB2)|(1<<PB1);
   //DDRB |= (1<<PB5)|(1<<PB3)|(1<<PB2);
   DDRB |= (1<<PB5)|(1<<PB3);
   //DDRD |= (1<<PD7)|(1<<PD6);
   //DDRD |= (1<<PD7)|(1<<PD6)|(1<<PD5)|(1<<PD2); //this should move out as it is only
outputs not spi
   // Enable SPI, Set as Master
   //Prescaler: Fosc/128 and Clock Phase: 1 (as required by max31865)
   //SPCR |= (1<<SPE)|(1<<MSTR)|(1<<CPHA)|(1<<SPR1)|(1<<SPR0);
   //Prescaler: Fosc/32 and Clock Phase: 1 and clock pol: 1 (as required by max31865)
   //SPCR |= (1<<SPE)|(1<<MSTR)|(1<<CPOL)|(1<<CPHA)|(1<<SPR1);
   //Prescaler: Fosc/32 and Clock Phase: 0 (as orignanlly set) max31865 says clock phase must
be 1 but that does not work
   SPCR |= (1<<SPE)|(1<<MSTR)|(1<<SPR1);
   SPSR |= (1<<SPI2X);
   //set unused ports pull-ups
   PORTB |= (0x01 << PB0)|(0x01 << PB1)|(0x01 << PB2);
   PORTC |= (0x01 << PC0)|(0x01 << PC1)|(0x01 << PC2)|(0x01 << PC3);
   PORTD |= (0x01 << PD3)|(0x01 << PD4)|(0x01 << PD6);
   PORTD |= (0x01 << PD5);  //CS for rtd
}
```

```
uint8_t readFault(void) {
  return readRegister8(MAX31865_FAULTSTAT_REG);
}

void clearFault(void)
{
   uint8_t t = readRegister8(MAX31865_CONFIG_REG);
   t &= ~0x2C;
   t |= MAX31865_CONFIG_FAULTSTAT;
   writeRegister8(MAX31865_CONFIG_REG, t);
}

void enableBias(int b)
{
   uint8_t t = readRegister8(MAX31865_CONFIG_REG);
   if (b)
   {
     t |= MAX31865_CONFIG_BIAS;       // enable bias
   }
   else
   {
     t &= ~MAX31865_CONFIG_BIAS;       // disable bias
   }
   writeRegister8(MAX31865_CONFIG_REG, t);
}

void autoConvert(void) {
   uint8_t t = readRegister8(MAX31865_CONFIG_REG);
   t &= ~MAX31865_CONFIG_MODEAUTO;       // disable autoconvert
   writeRegister8(MAX31865_CONFIG_REG, t);
}

void setWires(void )
{
   uint8_t t = readRegister8(MAX31865_CONFIG_REG);
   t |= MAX31865_CONFIG_3WIRE;
   writeRegister8(MAX31865_CONFIG_REG, t);
}

uint16_t readRTD (void)
{
   clearFault();
   enableBias(1);
   _delay_ms(100);
   uint8_t t = readRegister8(MAX31865_CONFIG_REG);
   t |= MAX31865_CONFIG_1SHOT;
   writeRegister8(MAX31865_CONFIG_REG, t);
   _delay_ms(650);
   uint16_t rtd = readRegister16(MAX31865_RTDMSB_REG);
   enableBias(0); //added because of overheating with bias current
   return rtd >>= 1;
}

float temperature(float refResistor)
```

```
{
  // http://www.analog.com/media/en/technical-documentation/application-notes/AN709_0.pdf
   float Rt;
   float temp;
   Rt = readRTD();
   Rt /= 32768;
   Rt *= refResistor;
   temp = Z2 + (Z3 * Rt);
   temp = (sqrtf(temp) + Z1) / Z4;
   if (temp >= 0) return temp;
   float rpoly = Rt;
   temp = -242.02;
   temp += 2.2228 * rpoly;
   rpoly *= Rt;  // square
   temp += 2.5859e-3 * rpoly;
   rpoly *= Rt;  // ^3
   temp -= 4.8260e-6 * rpoly;
   rpoly *= Rt;  // ^4
   temp -= 2.8183e-8 * rpoly;
   rpoly *= Rt;  // ^5
   temp += 1.5243e-10 * rpoly;
   return temp;
}

void timer_init(void)
{
   TCCR0A |= (1 << WGM21); //  Configure timer 0 for CTC mode (2)
   TIMSK0 |= (1 << OCIE0A); // Enable CTC interrupt
   OCR0A = 229;
   TCCR0B |= (1 << CS21); // Start timer at Fcpu/8
   sei();
}

static void serialInit() {
   cli();
   ringBuffer_Init(&rbRX);
   ringBuffer_Init(&rbTX);
   bOwnAddressRS485 = RS485ADR;
   UBRR0   = UBRR_VALUE;
   //UCSR0A  = 0x02;
   UCSR0A  = 0x00;
   UCSR0B  = 0x00;
   UCSR0C  = 0x06;
   RS485_DIRECTION_DDR = RS485_DIRECTION_DDR | RS485_DIRECTION_BIT;
   RS485_DIRECTION_PORT = RS485_DIRECTION_PORT & (~RS485_DIRECTION_BIT);
   serialModeRX();
   //serialModeTX();
   sei();
}

int main(void)
{
   cli();
   mcusr = MCUSR;
```

```
  MCUSR = 0;
  //cli();
  //WDT_off();
  //wdt_disable();
  wdt_enable(WDTO_8S);
  //wdt_disable();
  Z1 = -RTD_A;
  Z2 = RTD_A * RTD_A - (4 * RTD_B);
  Z3 = (4 * RTD_B) / RTDNOMINAL;
  Z4 = 2 * RTD_B;
//   float temperaturertd = 20;
//   uint8_t cnt = 0;
  uint8_t skiph = 0;
  uint8_t skipl = 0;
  spi_init_master ();
  temperature_init();
  serialInit();
  twi_init();
  timer_init();
  _delay_ms(5);
  acceleration_init();
  cnt = eeprom_read_byte((uint8_t*)32);
  maxabtemp = eeprom_read_byte((uint8_t*)33);
  maxtimer = eeprom_read_byte((uint8_t*)34);
  //ElapsedQSeconds = eeprom_read_byte((uint8_t*)35);
  temperaturertd = temperature(RREF);
  if(temperaturertd >= (float)maxabtemp)
  {
    PORTD |= (0x01 << PD7);
    skiph = 1;
    cnt++;
    eeprom_update_byte((uint8_t*)32,cnt);

  }
  if(temperaturertd < (float)maxabtemp) //&& temperaturertd >= MAXBT - BTDB)
  {
    PORTD &= ~(0x01 << PD7);
    skipl = 1;
  }
  for(;;)
  {
    temperaturertd = temperature(RREF);
    if(temperaturertd > (float)maxabtemp && !skiph)
    {
      PORTD |= (0x01 << PD7);  //set refrigerator cutoff relay (NC) on so not running
      skiph++;
      skipl--;
      //if(cnt <= MAXRESTARTS)
      //{
        cnt++;
      //}
      eeprom_update_byte((uint8_t*)32,cnt);
      resettimer();
    }
```

```
    if(((temperaturertd < (float)(maxabtemp - BTDB) && !skipl) || (!ElapsedQSeconds && !skipl))
&& cnt <= MAXRESTARTS)
    {
      PORTD &= ~(0x01 << PD7);  //set refrigerator cutoff relay (NC) off so running
      skipl++;
      skiph--;
      resettimer();
    }
    wdt_reset();
    if(!tpcr)
    {
      tpcr = TPCRMAX;
    }
    wdt_reset();
    if(!tsht)
    {
      readbshttwi();  //temp removal to test
      tsht = TSHTMAX;
    }
    wdt_reset();
    if(!tbme)
    {
      readbmetwi();  //temp removal to test
      tbme = TBMEMAX;
    }
    //writeadg715();
    acceleration_init();
    wdt_reset();
    readxyztwi();  //temp removal to test
    wdt_reset();
    serialHandleData(); /* In case we received data, handle that data ... */
    wdt_reset();
    if(ElapsedQ0MilliSeconds >= 8000)
    {
      if(ElapsedQSeconds > 0)
      {
        ElapsedQSeconds--;
        eeprom_update_byte((uint8_t*)35,ElapsedQSeconds);
        ElapsedQ0MilliSeconds = 0;
      }
    }
/*
    if(ElapsedQ2MilliSeconds >= 8000)
    {
      ElapsedQsgSeconds++;
    }
*/
/*
    if(!ElapsedQSeconds ) //&& ((regD >> c2wdtm) & 0x01))
    {
      //togglepin(MINCOOLINGPERIOD); //for test only must be commented
      //resettimer(); //for test only must be commented
    }
*/
```

```
/*
    if(ElapsedQsgSeconds > WDTIMEOUT && wdtm)
    {
      resetc2timer();
      togglepin(c2reset);
    }
*/
    wdt_reset();
    tpcr--;
    tbme--;
    tsht--;

/*
    cycles[0]++;
    if(cycles[0] > 254)
    {
      cycles[1]++;
      cycles[0] = 0;
      if(cycles[1] > 254)
      {
        cycles[2]++;
        cycles[1] = 0;
      }
    }
*/

  }
}

/*
ISR(TIMER2_COMPA_vect)
{
  ElapsedQ2MilliSeconds++;
}
*/
ISR(TIMER0_COMPA_vect)
{
  ElapsedQ0MilliSeconds++;
}
/*
ISR(INT1_vect)
{
  resetc2timer();
}
*/

/*
ISR(INT0_vect)
{
  senddata();
}
*/

ISR(USART_UDRE_vect) {
```

```
    /*
       Transmit as long as data is available to transmit. If there
       is no more data we simply stop to transmit and enter receive mode
       again
    */
    cli();

    if(ringBuffer_AvailableN(&rbTX) == 0) {
       /* Disable transmit mode again ... */
       UCSR0B = UCSR0B & (~0x20);
    } else {
       /* Shift next byte to the outside world ... */
       UDR0 = ringBuffer_ReadChar(&rbTX);
    }

    sei();
}

ISR(USART_TX_vect) {
    RS485_DIRECTION_PORT = RS485_DIRECTION_PORT & (~RS485_DIRECTION_BIT);
    UCSR0B = (UCSR0B & (~0xE8)) | 0x10 | 0x80; /* Disable all transmit interrupts, enable
receiver, enable receive complete interrupt */
    //PORTD = PORTD & (~(0x08)); /* Set RE and DE to low (RE: active, DE: inactive) */
    //UCSR0B = (UCSR0B & (~0xE8)) | 0x10 | 0x80; /* Disable all transmit interrupts, enable
receiver, enable receive complete interrupt */
}

ISR(USART_RX_vect) {
    cli();
    ringBuffer_WriteChar(&rbRX, UDR0);
    sei();
}
```