```
//***************************************************************************
//
// File Name            : "BME680 Sensor API"
// Title                : BME680 Sensor API
// Date                 : 5/8/20
// Version              : 1.0
// Target MCU           : SAML21J18B
// Target Hardware      ; DOG LCD, BME680
// Author               : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// Uses the user_spi_write, user_spi_read and spi_transfer functions with
// the parameters defined in the BME680 Sensor API to configure the BME680
// and read the raw temperature measurement from the sensor's ADC.
//
// Warnings             :
// Restrictions         : none
// Algorithms           : none
// References           :
//
// Revision History     : Initial version
//
//
//**************************************************************************


#include "saml21j18b.h"

uint8_t status, id;
uint32_t raw_temperature;
unsigned char* ARRAY_PORT_PINCFG0 = (unsigned char*)&REG_PORT_PINCFG0;
unsigned char* ARRAY_PORT_PMUX0 = (unsigned char*)&REG_PORT_PMUX0;

void init_spi_MCU (void);
static void init_spi_BME680 (void);
void user_delay_ms (uint32_t period);
static uint8_t spi_transfer (uint8_t data);
int8_t user_spi_read (uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len);
```

```c
int8_t user_spi_write (uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len);

int main(void) {
    uint8_t meas_stat = 0x20;
    uint32_t temp;
    init_spi_MCU();                             //initialize MCU and BME680 SPI communication
    init_spi_BME680();
    user_spi_write(0, 0x72, (void *)0, 1);      //turn off humidity reading
    user_spi_write(0, 0x74, (void *)0x20, 1);   //turn off pressure reading, temperature oversampling x1, sleep mode
    user_spi_write(0, 0x75, (void *)0x04, 1);   //IIR filter coefficient = 1, SPI 4 wire mode
    user_spi_write(0, 0x70, (void *)0x08, 1);   //turn off heater

    while (1) {
        user_delay_ms(1000);                        //delay 1s
        user_spi_write(0, 0x74, (void *)0x21, 1);           //enable forced mode
        while (meas_stat & 0x20) {                  //poll measuring status flag
            user_spi_read(0, 0x1D, &meas_stat, 1);
        }
        user_spi_read(0, 0x22, &temp, 1);           //read 3 temperature data registers
        temp <<= 8;
        user_spi_read(0, 0x23, &temp, 1);
        temp <<= 8;
        user_spi_read(0, 0x24, &temp, 1);
        temp >>= 4;
        raw_temperature = temp;                     //store in raw_temperature
    }
}

void init_spi_MCU (void) {
    REG_GCLK_PCHCTRL19 = 0x00000040;    /* SERCOM1 core clock not enabled by default */

    ARRAY_PORT_PINCFG0[16] |= 1;    /* allow pmux to set PA16 pin configuration */
    ARRAY_PORT_PINCFG0[17] |= 1;    /* allow pmux to set PA17 pin configuration */
    ARRAY_PORT_PINCFG0[18] |= 1;    /* allow pmux to set PA18 pin configuration */
    ARRAY_PORT_PINCFG0[19] |= 1;    /* allow pmux to set PA19 pin configuration */
    ARRAY_PORT_PMUX0[8] = 0x22;     /* PA16 = MOSI, PA17 = SCK */
    ARRAY_PORT_PMUX0[9] = 0x22;     /* PA18 = SS,   PA19 = MISO */
    REG_PORT_DIRSET1 = 0x80;        /* PB07 = CS for BME680 */

    REG_SERCOM1_SPI_CTRLA = 1;               /* reset SERCOM1 */
    while (REG_SERCOM1_SPI_CTRLA & 1) {}    /* wait for reset to complete */

    REG_SERCOM1_SPI_CTRLA = 0x3030000C;     /* MISO-3, MOSI-0, SCK-1, SS-2, CPOL=1, CPHA=1 */
    REG_SERCOM1_SPI_CTRLB = 0x00022000;     /* Master SS, 8-bit, receiver enabled */
    REG_SERCOM1_SPI_BAUD = 0;               /* SPI clock is 4MHz/2 = 2MzHz */
    REG_SERCOM1_SPI_CTRLA |= 2;             /* enable SERCOM1 */
}

static void init_spi_BME680 (void) {
    user_spi_write (0, 0x60, (void *)0xB6, 1);      //software reset BME680
    user_spi_write(0, 0x73, (void *)0, 1);          //switch to page 0 of memory map
    user_spi_read(0, 0x73, &status, 1);     //read status register
    user_spi_read(0, 0x50, &id, 1);         //read id register
    user_spi_write(0, 0x73, (void *)0x10, 1);               //switch to page 1 of memory map
    user_spi_read(0, 0x73, &status, 1);     //read status register
}

void user_delay_ms (uint32_t period) {
    for (int i = 0; i < 170*period; i++) {      //based off of 30us delay in DOGM163W_A_SERCOM1.c
        __asm("nop");                           //delay by period ms
    }
}

static uint8_t spi_transfer (uint8_t data) {
    uint8_t Rx_data;
    while(!(REG_SERCOM1_SPI_INTFLAG & 1)) {}        //wait until Tx ready
    REG_SERCOM1_SPI_DATA = data;                    //send data byte
    while(!(REG_SERCOM1_SPI_INTFLAG & 2)) {}        //wait until transmit is complete
    while(!(REG_SERCOM1_SPI_INTFLAG & 4)) {}        //wait until receive is complete
    Rx_data = REG_SERCOM1_SPI_DATA;                 //read data register
    return Rx_data;
}

int8_t user_spi_read (uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len) {
    int8_t rslt = 0;                //return 0 for success, non-zero for failure
    reg_addr |= 0x80;               //mask bit 7 to a '1'
    REG_PORT_OUTCLR1 |= 0x80;       //CS = 0 -> BME680 selected
    spi_transfer(reg_addr);         //send control byte with register address
    while (len--) {
        *reg_data = spi_transfer(0);
        reg_data++;
    }
    REG_PORT_OUTSET1 |= 0x80;       //CS = 1 -> BME680 deselected
    return rslt;
}

int8_t user_spi_write (uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len) {
    int8_t rslt = 0;
    REG_PORT_OUTCLR1 |= 0x80;       //CS = 0 -> BME680 selected
    spi_transfer(reg_addr); //send control byte with register address
    while (len--) {
        spi_transfer(reg_data);
        reg_data++;
    }
    REG_PORT_OUTSET1 |= 0x80;       //CS = 1 -> BME680 deselected
    return rslt;
}
```

```
//***************************************************************************
//
// File Name            : "BME680 Sensor API dt2"
// Title                : BME680 Sensor API dt2
// Date                 : 5/8/20
// Version              : 1.0
// Target MCU           : SAML21J18B
// Target Hardware      ; DOG LCD, BME680
// Author               : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// Utilizes the BME680's API to convert it's ADC's readings into actual
// values to be displayed on the DOG LCD and TeraTerm.
//
// Warnings             :
// Restrictions         : none
// Algorithms           : none
// References           :
//
// Revision History     : Initial version
//
//
//**************************************************************************


#include "saml21j18b.h"
#include "bme680.h"
#include "bme680_defs.h"
#include "DOGM163W_A_SERCOM1.h"
#include "RS232_SERCOM4.h"
#include "sys_support.h"


uint8_t status, id;         //declare chip id and page number global variables

//function prototypes
void init_spi_MCU (void);
static void init_spi_BME680 (void);
```

```c
static void init_BME680(void);
void user_delay_ms (uint32_t period);
static uint8_t spi_transfer (uint8_t data);
int8_t user_spi_read (uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len);
int8_t user_spi_write (uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len);

int main(void) {
    int KeyPress=0x04, Last_KeyPress, count=0;
    uint16_t set_required_settings, meas_period;
    //initialize SERCOM4 for RS232 communication and SERCOM1 for the LCD and BME680
    UART4_init();
    init_lcd_dog();
    init_spi_BME680();
    init_BME680();

    struct bme680_dev gas_sensor;           //create instance of bme680_dev named gas_sensor
    gas_sensor.dev_id = 0;                  //fill in various parameters for gas_sensor
    gas_sensor.intf = BME680_SPI_INTF;
    gas_sensor.read = user_spi_read;
    gas_sensor.write = user_spi_write;
    gas_sensor.delay_ms = user_delay_ms;
    gas_sensor.amb_temp = 25;
    int8_t rslt = BME680_OK;
    rslt = bme680_init(&gas_sensor);

    //set the temperature, pressure and humidity oversampling. set IIR filter size
    gas_sensor.tph_sett.os_hum = BME680_OS_2X;
    gas_sensor.tph_sett.os_pres = BME680_OS_4X;
    gas_sensor.tph_sett.os_temp = BME680_OS_8X;
    gas_sensor.tph_sett.filter = BME680_FILTER_SIZE_3;

    //enable gas measurements and configure gas heat plate temperature and heating duration
    gas_sensor.gas_sett.run_gas = BME680_ENABLE_GAS_MEAS;
    gas_sensor.gas_sett.heatr_temp = 320;
    gas_sensor.gas_sett.heatr_dur = 150;

    //put BME680 into forced mode
    gas_sensor.power_mode = BME680_FORCED_MODE;

    //configure all the temperature, pressure, humidity and gas settings
    set_required_settings = BME680_OST_SEL | BME680_OSP_SEL | BME680_OSH_SEL | BME680_FILTER_SEL |
      BME680_GAS_SENSOR_SEL;
    rslt = bme680_set_sensor_settings(set_required_settings, &gas_sensor);
    rslt = bme680_set_sensor_mode(&gas_sensor);

    bme680_get_profile_dur(&meas_period, &gas_sensor);
    struct bme680_field_data data;      //create instance of bme680_field_data name 'data'

    while (1) {
        user_delay_ms(meas_period);          //delay for meas_period ms
        rslt = bme680_get_sensor_data(&data, &gas_sensor);      //read sensor measurements and store in data
        init_spi_lcd();                      //initialize spi for lcd before transactions

        Last_KeyPress = KeyPress;            //set last key press equal to current key press
        KeyPress = REG_PORT_IN0 & 0x04;      //mask pushbutton value onto current key press
        if (Last_KeyPress != KeyPress) {     //if last and current key value no equal
            if (KeyPress == 0x04) {          //and if key is not held down, increment counter
                count++;
            }
        }
        if (count % 2 == 0) {        //if count is even display temperature and pressure
            sprintf(dsp_buff_1, "T: %.2f degC", data.temperature / 100.0f);
            sprintf(dsp_buff_2, "P: %.2f hPa  ", data.pressure / 100.0f);
            update_lcd_dog();
        }
        else {                       //if count is odd display humidity and gas resistance
            sprintf(dsp_buff_1, "H: %.2f %%rH", data.humidity / 1000.0f);
            if (data.status & BME680_GASM_VALID_MSK) {
                sprintf(dsp_buff_2, "G: %ld ohms     ", data.gas_resistance);
            }
            update_lcd_dog();
        }
        //print values to TeraTerm
        printf("T: %.2f degC,  P: %.2f hPa,  H: %.2f %%rH", data.temperature / 100.0f, data.pressure / 100.0f,
            data.humidity / 1000.0f);
        if (data.status & BME680_GASM_VALID_MSK) {
            printf(",  G: %ld ohms", data.gas_resistance);
        }
        printf("\r\n");
        init_spi_BME680();      //initialize spi for BME680 before transactions
        if (gas_sensor.power_mode == BME680_FORCED_MODE) {
            rslt = bme680_set_sensor_mode(&gas_sensor);
        }
    }
}

//***************************************************************************
//
// Function Name        : "init_spi_BME680"
// Date                 : 5/9/20
// Version              : 1.0
// Target MCU           : SAML21J18B
// Target Hardware      ; BME680
// Author               : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// Configures the SAML21J18B's SERCOM1 for SPI communication with the
// BME680. PA16 = MOSI, PA17 = SCK, PA19 = MISO, PB07 = CS.
//
// Warnings             : none
// Restrictions         : none
// Algorithms           : none
// References           : none
//
// Revision History     : Initial version
//
//**************************************************************************
static void init_spi_BME680 (void) {
    REG_GCLK_PCHCTRL19 = 0x00000040;    /* SERCOM1 core clock not enabled by default */

    ARRAY_PORT_PINCFG0[16] |= 1;    /* allow pmux to set PA16 pin configuration */
    ARRAY_PORT_PINCFG0[17] |= 1;    /* allow pmux to set PA17 pin configuration */
    ARRAY_PORT_PINCFG0[18] |= 1;    /* allow pmux to set PA18 pin configuration */
    ARRAY_PORT_PINCFG0[19] |= 1;    /* allow pmux to set PA19 pin configuration */
    ARRAY_PORT_PMUX0[8] = 0x22;     /* PA16 = MOSI, PA17 = SCK */
    ARRAY_PORT_PMUX0[9] = 0x22;     /* PA18 = SS,   PA19 = MISO */
    REG_PORT_DIRSET1 = 0x80;        /* PB07 = CS for BME680 */

    REG_PORT_DIRCLR0 = 0x04;        //PA02 input for SW0
    ARRAY_PORT_PINCFG0[2] |= 6;     //enable PA02 with pull
    REG_PORT_OUTSET0 = 0x04;        //make PA02 pull-up

    REG_SERCOM1_SPI_CTRLA = 1;              /* reset SERCOM1 */
    while (REG_SERCOM1_SPI_CTRLA & 1) {}    /* wait for reset to complete */

    REG_SERCOM1_SPI_CTRLA = 0x3030000C;     /* MISO-3, MOSI-0, SCK-1, SS-2, CPOL=1, CPHA=1 */
    REG_SERCOM1_SPI_CTRLB = 0x00020000;     /* Master SS, 8-bit, receiver enabled */
    REG_SERCOM1_SPI_BAUD = 0;               /* SPI clock is 4MHz/2 = 2MzHz */
    REG_SERCOM1_SPI_CTRLA |= 2;             /* enable SERCOM1 */
}

//***************************************************************************
//
// Function Name        : "init_BME680"
// Date                 : 5/9/20
// Version              : 1.0
// Target MCU           : SAML21J18B
// Target Hardware      ; BME680
// Author               : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// Software resets the BME680 and reads its memory map page and status
// register. Sets the memory map to page 1 so the BME680 is ready for
// configuration after this function.
//
// Warnings             : none
// Restrictions         : none
// Algorithms           : none
// References           : none
//
```

```
// Revision History     : Initial version
//
//***************************************************************************
static void init_BME680(void) {
    uint8_t write_data = 0xB6;
    REG_PORT_DIRSET1 |= 0x80;
    user_spi_write (0, 0x60, &write_data, 1);       //software reset BME680
    write_data = 0x00;
    user_spi_write(0, 0x73, &write_data, 1);        //switch to page 0 of memory map
    user_spi_read(0, 0x73, &status, 1);             //read status register
    status >>= 4;
    user_spi_read(0, 0x50, &id, 1);                 //read id register
    write_data = 0x10;
    user_spi_write(0, 0x73, &write_data, 1);        //switch to page 1 of memory map
    user_spi_read(0, 0x73, &status, 1);             //read status register
    status >>= 4;
}

//****************************************************************************
//
// Function Name        : "user_delay_ms"
// Date                 : 5/9/20
// Version              : 1.0
// Target MCU           : SAML21J18B
// Target Hardware      ; N/A
// Author               : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// Delays the system by a 'period' number of ms.
//
// Warnings             : none
// Restrictions         : none
// Algorithms           : none
// References           : none
//
// Revision History     : Initial version
//
//***************************************************************************
```

```
void user_delay_ms (uint32_t period) {
    for (int i = 0; i < 170*period; i++) {      //based off of 30us delay in DOGM163W_A_SERCOM1.c
        __asm("nop");                           //delay by period ms
    }
}

//***************************************************************************
//
// Function Name        : "spi_transfer"
// Date                 : 5/9/20
// Version              : 1.0
// Target MCU           : SAML21J18B
// Target Hardware      ; BME680
// Author               : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// Transmits an unsigned integer byte 'data' through SERCOM1's SPI data
// register, then receives a transmission through SERCOM1's SPI data register
// and returns the received unsigned integer byte.
//
// Warnings             : none
// Restrictions         : none
// Algorithms           : none
// References           : none
//
// Revision History     : Initial version
//
//**************************************************************************
static uint8_t spi_transfer (uint8_t data) {
    uint8_t Rx_data;
    while(!(REG_SERCOM1_SPI_INTFLAG & 1)) {}        //wait until Tx ready
    REG_SERCOM1_SPI_DATA = data;                    //send data byte
    while(!(REG_SERCOM1_SPI_INTFLAG & 2)) {}        //wait until transmit is complete
    while(!(REG_SERCOM1_SPI_INTFLAG & 4)) {}        //wait until receive is complete
    Rx_data = REG_SERCOM1_SPI_DATA;                 //read data register
    return Rx_data;
}
```

```c
//***************************************************************************
//
// Function Name        : "user_spi_read"
// Date                 : 5/9/20
// Version              : 1.0
// Target MCU           : SAML21J18B
// Target Hardware      ; BME680
// Author               : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// This function is passed 4 parameters, only reg_addr, *reg_data and len are
// used in this definition. The MS bit of reg_addr is masked to a '1' in order
// to indicate a read transaction to the BME680. The spi_transfer function
// is first called to pass the register address to be read. It is then called
// a second time to receive the register's data and update *reg_data. If
// len > 1, the register address to be read is auto-incremented and the address
// of reg_data is incremented as well to allow for multiple read transactions.
//
// Warnings             : none
// Restrictions         : none
// Algorithms           : none
// References           : none
//
// Revision History     : Initial version
//
//**************************************************************************
int8_t user_spi_read (uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len) {
    int8_t rslt = 0;                //return 0 for success, non-zero for failure
    reg_addr |= 0x80;               //mask bit 7 to a '1' for a read transaction
    REG_PORT_OUTCLR1 &= 0x80;       //CS = 0 -> BME680 selected
    spi_transfer(reg_addr);         //send control byte with register address
    while (len--) {
        *reg_data = spi_transfer(0);
        reg_data++;
    }
    REG_PORT_OUTSET1 |= 0x80;       //CS = 1 -> BME680 unselected
    return rslt;
}
```

```
//***************************************************************************
//
// Function Name        : "user_spi_write"
// Date                 : 5/9/20
// Version              : 1.0
// Target MCU           : SAML21J18B
// Target Hardware      ; BME680
// Author               : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// This function is passed 4 parameters, only reg_addr, *reg_data and len are
// used in this definition. The spi_transfer function is first called to pass
// the register address to be written to. spi_transfer is then called a second
// time to write *reg_data to the register. If len > 1, the register address
// to be written to is auto-incremented and reg_data is incremented as well
// to allow for multiple write transactions.
//
// Warnings             : none
// Restrictions         : none
// Algorithms           : none
// References           : none
//re
// Revision History     : Initial version
//
//**************************************************************************
int8_t user_spi_write (uint8_t dev_id, uint8_t reg_addr, uint8_t *reg_data, uint16_t len) {
    int8_t rslt = 0;
    REG_PORT_OUTCLR1 &= 0x80;       //CS = 0 -> BME680 selected
    spi_transfer(reg_addr);         //send control byte with register address
    while (len--) {
        spi_transfer(*reg_data);
        reg_data++;
    }
    REG_PORT_OUTSET1 |= 0x80;       //CS = 1 -> BME680 unselected
    return rslt;
}
```

```
/**\mainpage
 * Copyright (C) 2017 - 2018 Bosch Sensortec GmbH
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 *
 * Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in the
 * documentation and/or other materials provided with the distribution.
 *
 * Neither the name of the copyright holder nor the names of the
 * contributors may be used to endorse or promote products derived from
 * this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
 * CONTRIBUTORS "AS IS" AND ANY EXPRESS OR
 * IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL COPYRIGHT HOLDER
 * OR CONTRIBUTORS BE LIABLE FOR ANY
 * DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
 * OR CONSEQUENTIAL DAMAGES(INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 * ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE
 *
 * The information provided is believed to be accurate and reliable.
 * The copyright holder assumes no responsibility
 * for the consequences of use
 * of such information nor for any infringement of patents or
 * other rights of third parties which may result from its use.
 * No license is granted by implication or otherwise under any patent or
 * patent rights of the copyright holder.
 *
 * File     bme680.c
 * @date    19 Jun 2018
 * @version 3.5.9
 *
 */

/*! @file bme680.c
 @brief Sensor driver for BME680 sensor */
#include "bme680.h"

/*!
 * @brief This internal API is used to read the calibrated data from the sensor.
 *
 * This function is used to retrieve the calibration
 * data from the image registers of the sensor.
 *
 * @note Registers 89h  to A1h for calibration data 1 to 24
 *        from bit 0 to 7
 * @note Registers E1h to F0h for calibration data 25 to 40
 *        from bit 0 to 7
 * @param[in] dev   :Structure instance of bme680_dev.
 *
 * @return Result of API execution status.
 * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
 */
static int8_t get_calib_data(struct bme680_dev *dev);

/*!
 * @brief This internal API is used to set the gas configuration of the sensor.
 *
 * @param[in] dev   :Structure instance of bme680_dev.
 *
 * @return Result of API execution status.
```

```c
 * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
 */
static int8_t set_gas_config(struct bme680_dev *dev);

/*!
 * @brief This internal API is used to get the gas configuration of the sensor.
 * @note heatr_temp and heatr_dur values are currently register data
 * and not the actual values set
 *
 * @param[in] dev   :Structure instance of bme680_dev.
 *
 * @return Result of API execution status.
 * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
 */
static int8_t get_gas_config(struct bme680_dev *dev);

/*!
 * @brief This internal API is used to calculate the Heat duration value.
 *
 * @param[in] dur   :Value of the duration to be shared.
 *
 * @return uint8_t threshold duration after calculation.
 */
static uint8_t calc_heater_dur(uint16_t dur);

#ifndef BME680_FLOAT_POINT_COMPENSATION

/*!
 * @brief This internal API is used to calculate the temperature value.
 *
 * @param[in] dev   :Structure instance of bme680_dev.
 * @param[in] temp_adc  :Contains the temperature ADC value .
 *
 * @return uint32_t calculated temperature.
 */
static int16_t calc_temperature(uint32_t temp_adc, struct bme680_dev *dev);
```

```c
/*!
 * @brief This internal API is used to calculate the pressure value.
 *
 * @param[in] dev   :Structure instance of bme680_dev.
 * @param[in] pres_adc  :Contains the pressure ADC value .
 *
 * @return uint32_t calculated pressure.
 */
static uint32_t calc_pressure(uint32_t pres_adc, const struct bme680_dev *dev);

/*!
 * @brief This internal API is used to calculate the humidity value.
 *
 * @param[in] dev   :Structure instance of bme680_dev.
 * @param[in] hum_adc   :Contains the humidity ADC value.
 *
 * @return uint32_t calculated humidity.
 */
static uint32_t calc_humidity(uint16_t hum_adc, const struct bme680_dev *dev);

/*!
 * @brief This internal API is used to calculate the Gas Resistance value.
 *
 * @param[in] dev       :Structure instance of bme680_dev.
 * @param[in] gas_res_adc   :Contains the Gas Resistance ADC value.
 * @param[in] gas_range     :Contains the range of gas values.
 *
 * @return uint32_t calculated gas resistance.
 */
static uint32_t calc_gas_resistance(uint16_t gas_res_adc, uint8_t gas_range, const struct bme680_dev *dev);

/*!
 * @brief This internal API is used to calculate the Heat Resistance value.
 *
 * @param[in] dev   : Structure instance of bme680_dev
 * @param[in] temp  : Contains the target temperature value.
 *
 * @return uint8_t calculated heater resistance.
 */
static uint8_t calc_heater_res(uint16_t temp, const struct bme680_dev *dev);

#else
/*!
 * @brief This internal API is used to calculate the
 * temperature value value in float format
 *
 * @param[in] dev   :Structure instance of bme680_dev.
 * @param[in] temp_adc  :Contains the temperature ADC value .
 *
 * @return Calculated temperature in float
 */
static float calc_temperature(uint32_t temp_adc, struct bme680_dev *dev);

/*!
 * @brief This internal API is used to calculate the
 * pressure value value in float format
 *
 * @param[in] dev   :Structure instance of bme680_dev.
 * @param[in] pres_adc  :Contains the pressure ADC value .
 *
 * @return Calculated pressure in float.
 */
static float calc_pressure(uint32_t pres_adc, const struct bme680_dev *dev);

/*!
 * @brief This internal API is used to calculate the
 * humidity value value in float format
 *
 * @param[in] dev   :Structure instance of bme680_dev.
 * @param[in] hum_adc   :Contains the humidity ADC value.
 *
 * @return Calculated humidity in float.
 */
static float calc_humidity(uint16_t hum_adc, const struct bme680_dev *dev);
```

```
/*!
 * @brief This internal API is used to calculate the
 * gas resistance value value in float format
 *
 * @param[in] dev       :Structure instance of bme680_dev.
 * @param[in] gas_res_adc   :Contains the Gas Resistance ADC value.
 * @param[in] gas_range     :Contains the range of gas values.
 *
 * @return Calculated gas resistance in float.
 */
static float calc_gas_resistance(uint16_t gas_res_adc, uint8_t gas_range, const struct bme680_dev *dev);

/*!
 * @brief This internal API is used to calculate the
 * heater resistance value in float format
 *
 * @param[in] temp  : Contains the target temperature value.
 * @param[in] dev   : Structure instance of bme680_dev.
 *
 * @return Calculated heater resistance in float.
 */
static float calc_heater_res(uint16_t temp, const struct bme680_dev *dev);

#endif

/*!
 * @brief This internal API is used to calculate the field data of sensor.
 *
 * @param[out] data :Structure instance to hold the data
 * @param[in] dev   :Structure instance of bme680_dev.
 *
 *  @return int8_t result of the field data from sensor.
 */
static int8_t read_field_data(struct bme680_field_data *data, struct bme680_dev *dev);

/*!
```

```
 * @brief This internal API is used to set the memory page
 * based on register address.
 *
 * The value of memory page
 *  value  | Description
 * --------|--------------
 *   0     | BME680_PAGE0_SPI
 *   1     | BME680_PAGE1_SPI
 *
 * @param[in] dev   :Structure instance of bme680_dev.
 * @param[in] reg_addr  :Contains the register address array.
 *
 * @return Result of API execution status
 * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
 */
static int8_t set_mem_page(uint8_t reg_addr, struct bme680_dev *dev);

/*!
 * @brief This internal API is used to get the memory page based
 * on register address.
 *
 * The value of memory page
 *  value  | Description
 * --------|--------------
 *   0     | BME680_PAGE0_SPI
 *   1     | BME680_PAGE1_SPI
 *
 * @param[in] dev   :Structure instance of bme680_dev.
 *
 * @return Result of API execution status
 * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
 */
static int8_t get_mem_page(struct bme680_dev *dev);

/*!
 * @brief This internal API is used to validate the device pointer for
 * null conditions.
 *
 * @param[in] dev   :Structure instance of bme680_dev.
 *
 * @return Result of API execution status
 * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
 */
static int8_t null_ptr_check(const struct bme680_dev *dev);

/*!
 * @brief This internal API is used to check the boundary
 * conditions.
 *
 * @param[in] value :pointer to the value.
 * @param[in] min   :minimum value.
 * @param[in] max   :maximum value.
 * @param[in] dev   :Structure instance of bme680_dev.
 *
 * @return Result of API execution status
 * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
 */
static int8_t boundary_check(uint8_t *value, uint8_t min, uint8_t max, struct bme680_dev *dev);

/****************** Global Function Definitions *******************************/
/*!
 *@brief This API is the entry point.
 *It reads the chip-id and calibration data from the sensor.
 */
int8_t bme680_init(struct bme680_dev *dev)
{
    int8_t rslt;

    /* Check for null pointer in the device structure*/
    rslt = null_ptr_check(dev);
    if (rslt == BME680_OK) {
        /* Soft reset to restore it to default values*/
        rslt = bme680_soft_reset(dev);
        if (rslt == BME680_OK) {
```

```c
            rslt = bme680_get_regs(BME680_CHIP_ID_ADDR, &dev->chip_id, 1, dev);
            if (rslt == BME680_OK) {
                if (dev->chip_id == BME680_CHIP_ID) {
                    /* Get the Calibration data */
                    rslt = get_calib_data(dev);
                } else {
                    rslt = BME680_E_DEV_NOT_FOUND;
                }
            }
        }
    }

    return rslt;
}

/*!
 * @brief This API reads the data from the given register address of the sensor.
 */
int8_t bme680_get_regs(uint8_t reg_addr, uint8_t *reg_data, uint16_t len, struct bme680_dev *dev)
{
    int8_t rslt;

    /* Check for null pointer in the device structure*/
    rslt = null_ptr_check(dev);
    if (rslt == BME680_OK) {
        if (dev->intf == BME680_SPI_INTF) {
            /* Set the memory page */
            rslt = set_mem_page(reg_addr, dev);
            if (rslt == BME680_OK)
                reg_addr = reg_addr | BME680_SPI_RD_MSK;
        }
        dev->com_rslt = dev->read(dev->dev_id, reg_addr, reg_data, len);
        if (dev->com_rslt != 0)
            rslt = BME680_E_COM_FAIL;
    }

    return rslt;
```

```c
}

/*!
 * @brief This API writes the given data to the register address
 * of the sensor.
 */
int8_t bme680_set_regs(const uint8_t *reg_addr, const uint8_t *reg_data, uint8_t len, struct bme680_dev *dev)
{
    int8_t rslt;
    /* Length of the temporary buffer is 2*(length of register)*/
    uint8_t tmp_buff[BME680_TMP_BUFFER_LENGTH] = { 0 };
    uint16_t index;

    /* Check for null pointer in the device structure*/
    rslt = null_ptr_check(dev);
    if (rslt == BME680_OK) {
        if ((len > 0) && (len < BME680_TMP_BUFFER_LENGTH / 2)) {
            /* Interleave the 2 arrays */
            for (index = 0; index < len; index++) {
                if (dev->intf == BME680_SPI_INTF) {
                    /* Set the memory page */
                    rslt = set_mem_page(reg_addr[index], dev);
                    tmp_buff[(2 * index)] = reg_addr[index] & BME680_SPI_WR_MSK;
                } else {
                    tmp_buff[(2 * index)] = reg_addr[index];
                }
                tmp_buff[(2 * index) + 1] = reg_data[index];
            }
            /* Write the interleaved array */
            if (rslt == BME680_OK) {
                dev->com_rslt = dev->write(dev->dev_id, tmp_buff[0], &tmp_buff[1], (2 * len) - 1);
                if (dev->com_rslt != 0)
                    rslt = BME680_E_COM_FAIL;
            }
        } else {
            rslt = BME680_E_INVALID_LENGTH;
        }
    }

    return rslt;
}

/*!
 * @brief This API performs the soft reset of the sensor.
 */
int8_t bme680_soft_reset(struct bme680_dev *dev)
{
    int8_t rslt;
    uint8_t reg_addr = BME680_SOFT_RESET_ADDR;
    /* 0xb6 is the soft reset command */
    uint8_t soft_rst_cmd = BME680_SOFT_RESET_CMD;

    /* Check for null pointer in the device structure*/
    rslt = null_ptr_check(dev);
    if (rslt == BME680_OK) {
        if (dev->intf == BME680_SPI_INTF)
            rslt = get_mem_page(dev);

        /* Reset the device */
        if (rslt == BME680_OK) {
            rslt = bme680_set_regs(&reg_addr, &soft_rst_cmd, 1, dev);
            /* Wait for 5ms */
            dev->delay_ms(BME680_RESET_PERIOD);

            if (rslt == BME680_OK) {
                /* After reset get the memory page */
                if (dev->intf == BME680_SPI_INTF)
                    rslt = get_mem_page(dev);
            }
        }
    }

    return rslt;
}
```

```c
/*!
 * @brief This API is used to set the oversampling, filter and T,P,H, gas selection
 * settings in the sensor.
 */
int8_t bme680_set_sensor_settings(uint16_t desired_settings, struct bme680_dev *dev)
{
    int8_t rslt;
    uint8_t reg_addr;
    uint8_t data = 0;
    uint8_t count = 0;
    uint8_t reg_array[BME680_REG_BUFFER_LENGTH] = { 0 };
    uint8_t data_array[BME680_REG_BUFFER_LENGTH] = { 0 };
    uint8_t intended_power_mode = dev->power_mode; /* Save intended power mode */

    /* Check for null pointer in the device structure*/
    rslt = null_ptr_check(dev);
    if (rslt == BME680_OK) {
        if (desired_settings & BME680_GAS_MEAS_SEL)
            rslt = set_gas_config(dev);

        dev->power_mode = BME680_SLEEP_MODE;
        if (rslt == BME680_OK)
            rslt = bme680_set_sensor_mode(dev);

        /* Selecting the filter */
        if (desired_settings & BME680_FILTER_SEL) {
            rslt = boundary_check(&dev->tph_sett.filter, BME680_FILTER_SIZE_0, BME680_FILTER_SIZE_127, dev);
            reg_addr = BME680_CONF_ODR_FILT_ADDR;

            if (rslt == BME680_OK)
                rslt = bme680_get_regs(reg_addr, &data, 1, dev);

            if (desired_settings & BME680_FILTER_SEL)
                data = BME680_SET_BITS(data, BME680_FILTER, dev->tph_sett.filter);

            reg_array[count] = reg_addr; /* Append configuration */
            data_array[count] = data;
            count++;
        }

        /* Selecting heater control for the sensor */
        if (desired_settings & BME680_HCNTRL_SEL) {
            rslt = boundary_check(&dev->gas_sett.heatr_ctrl, BME680_ENABLE_HEATER,
                BME680_DISABLE_HEATER, dev);
            reg_addr = BME680_CONF_HEAT_CTRL_ADDR;

            if (rslt == BME680_OK)
                rslt = bme680_get_regs(reg_addr, &data, 1, dev);
            data = BME680_SET_BITS_POS_0(data, BME680_HCTRL, dev->gas_sett.heatr_ctrl);

            reg_array[count] = reg_addr; /* Append configuration */
            data_array[count] = data;
            count++;
        }

        /* Selecting heater T,P oversampling for the sensor */
        if (desired_settings & (BME680_OST_SEL | BME680_OSP_SEL)) {
            rslt = boundary_check(&dev->tph_sett.os_temp, BME680_OS_NONE, BME680_OS_16X, dev);
            reg_addr = BME680_CONF_T_P_MODE_ADDR;

            if (rslt == BME680_OK)
                rslt = bme680_get_regs(reg_addr, &data, 1, dev);

            if (desired_settings & BME680_OST_SEL)
                data = BME680_SET_BITS(data, BME680_OST, dev->tph_sett.os_temp);

            if (desired_settings & BME680_OSP_SEL)
                data = BME680_SET_BITS(data, BME680_OSP, dev->tph_sett.os_pres);

            reg_array[count] = reg_addr;
            data_array[count] = data;
            count++;
        }
        /* Selecting humidity oversampling for the sensor */
        if (desired_settings & BME680_OSH_SEL) {
            rslt = boundary_check(&dev->tph_sett.os_hum, BME680_OS_NONE, BME680_OS_16X, dev);
            reg_addr = BME680_CONF_OS_H_ADDR;

            if (rslt == BME680_OK)
                rslt = bme680_get_regs(reg_addr, &data, 1, dev);
            data = BME680_SET_BITS_POS_0(data, BME680_OSH, dev->tph_sett.os_hum);

            reg_array[count] = reg_addr; /* Append configuration */
            data_array[count] = data;
            count++;
        }

        /* Selecting the runGas and NB conversion settings for the sensor */
        if (desired_settings & (BME680_RUN_GAS_SEL | BME680_NBCONV_SEL)) {
            rslt = boundary_check(&dev->gas_sett.run_gas, BME680_RUN_GAS_DISABLE,
                BME680_RUN_GAS_ENABLE, dev);
            if (rslt == BME680_OK) {
                /* Validate boundary conditions */
                rslt = boundary_check(&dev->gas_sett.nb_conv, BME680_NBCONV_MIN,
                    BME680_NBCONV_MAX, dev);
            }

            reg_addr = BME680_CONF_ODR_RUN_GAS_NBC_ADDR;

            if (rslt == BME680_OK)
                rslt = bme680_get_regs(reg_addr, &data, 1, dev);

            if (desired_settings & BME680_RUN_GAS_SEL)
                data = BME680_SET_BITS(data, BME680_RUN_GAS, dev->gas_sett.run_gas);

            if (desired_settings & BME680_NBCONV_SEL)
                data = BME680_SET_BITS_POS_0(data, BME680_NBCONV, dev->gas_sett.nb_conv);

            reg_array[count] = reg_addr; /* Append configuration */
            data_array[count] = data;
            count++;
        }

        if (rslt == BME680_OK)
            rslt = bme680_set_regs(reg_array, data_array, count, dev);

        /* Restore previous intended power mode */
        dev->power_mode = intended_power_mode;
    }

    return rslt;
}

/*!
 * @brief This API is used to get the oversampling, filter and T,P,H, gas selection
 * settings in the sensor.
 */
int8_t bme680_get_sensor_settings(uint16_t desired_settings, struct bme680_dev *dev)
{
    int8_t rslt;
    /* starting address of the register array for burst read*/
    uint8_t reg_addr = BME680_CONF_HEAT_CTRL_ADDR;
    uint8_t data_array[BME680_REG_BUFFER_LENGTH] = { 0 };

    /* Check for null pointer in the device structure*/
    rslt = null_ptr_check(dev);
    if (rslt == BME680_OK) {
        rslt = bme680_get_regs(reg_addr, data_array, BME680_REG_BUFFER_LENGTH, dev);

        if (rslt == BME680_OK) {
            if (desired_settings & BME680_GAS_MEAS_SEL)
                rslt = get_gas_config(dev);

            /* get the T,P,H ,Filter,ODR settings here */
            if (desired_settings & BME680_FILTER_SEL)
                dev->tph_sett.filter = BME680_GET_BITS(data_array[BME680_REG_FILTER_INDEX],
                    BME680_FILTER);

            if (desired_settings & (BME680_OST_SEL | BME680_OSP_SEL)) {
                dev->tph_sett.os_temp = BME680_GET_BITS(data_array[BME680_REG_TEMP_INDEX], BME680_OST);
                dev->tph_sett.os_pres = BME680_GET_BITS(data_array[BME680_REG_PRES_INDEX], BME680_OSP);
            }

            if (desired_settings & BME680_OSH_SEL)
                dev->tph_sett.os_hum = BME680_GET_BITS_POS_0(data_array[BME680_REG_HUM_INDEX],
                    BME680_OSH);

            /* get the gas related settings */
            if (desired_settings & BME680_HCNTRL_SEL)
                dev->gas_sett.heatr_ctrl = BME680_GET_BITS_POS_0(data_array[BME680_REG_HCTRL_INDEX],
                    BME680_HCTRL);

            if (desired_settings & (BME680_RUN_GAS_SEL | BME680_NBCONV_SEL)) {
                dev->gas_sett.nb_conv = BME680_GET_BITS_POS_0(data_array[BME680_REG_NBCONV_INDEX],
                    BME680_NBCONV);
                dev->gas_sett.run_gas = BME680_GET_BITS(data_array[BME680_REG_RUN_GAS_INDEX],
                    BME680_RUN_GAS);
            }
        }
    } else {
        rslt = BME680_E_NULL_PTR;
    }

    return rslt;
}

/*!
 * @brief This API is used to set the power mode of the sensor.
 */
int8_t bme680_set_sensor_mode(struct bme680_dev *dev)
{
    int8_t rslt;
    uint8_t tmp_pow_mode;
```

```c
    uint8_t pow_mode = 0;
    uint8_t reg_addr = BME680_CONF_T_P_MODE_ADDR;

    /* Check for null pointer in the device structure*/
    rslt = null_ptr_check(dev);
    if (rslt == BME680_OK) {
        /* Call repeatedly until in sleep */
        do {
            rslt = bme680_get_regs(BME680_CONF_T_P_MODE_ADDR, &tmp_pow_mode, 1, dev);
            if (rslt == BME680_OK) {
                /* Put to sleep before changing mode */
                pow_mode = (tmp_pow_mode & BME680_MODE_MSK);

                if (pow_mode != BME680_SLEEP_MODE) {
                    tmp_pow_mode = tmp_pow_mode & (~BME680_MODE_MSK); /* Set to sleep */
                    rslt = bme680_set_regs(&reg_addr, &tmp_pow_mode, 1, dev);
                    dev->delay_ms(BME680_POLL_PERIOD_MS);
                }
            }
        } while (pow_mode != BME680_SLEEP_MODE);

        /* Already in sleep */
        if (dev->power_mode != BME680_SLEEP_MODE) {
            tmp_pow_mode = (tmp_pow_mode & ~BME680_MODE_MSK) | (dev->power_mode & BME680_MODE_MSK);
            if (rslt == BME680_OK)
                rslt = bme680_set_regs(&reg_addr, &tmp_pow_mode, 1, dev);
        }
    }

    return rslt;
}

/*!
 * @brief This API is used to get the power mode of the sensor.
 */
int8_t bme680_get_sensor_mode(struct bme680_dev *dev)
{
    int8_t rslt;
    uint8_t mode;

    /* Check for null pointer in the device structure*/
    rslt = null_ptr_check(dev);
    if (rslt == BME680_OK) {
        rslt = bme680_get_regs(BME680_CONF_T_P_MODE_ADDR, &mode, 1, dev);
        /* Masking the other register bit info*/
        dev->power_mode = mode & BME680_MODE_MSK;
    }

    return rslt;
}

/*!
 * @brief This API is used to set the profile duration of the sensor.
 */
void bme680_set_profile_dur(uint16_t duration, struct bme680_dev *dev)
{
    uint32_t tph_dur; /* Calculate in us */
    uint32_t meas_cycles;
    uint8_t os_to_meas_cycles[6] = {0, 1, 2, 4, 8, 16};

    meas_cycles = os_to_meas_cycles[dev->tph_sett.os_temp];
    meas_cycles += os_to_meas_cycles[dev->tph_sett.os_pres];
    meas_cycles += os_to_meas_cycles[dev->tph_sett.os_hum];

    /* TPH measurement duration */
    tph_dur = meas_cycles * UINT32_C(1963);
    tph_dur += UINT32_C(477 * 4); /* TPH switching duration */
    tph_dur += UINT32_C(477 * 5); /* Gas measurement duration */
    tph_dur += UINT32_C(500); /* Get it to the closest whole number.*/
    tph_dur /= UINT32_C(1000); /* Convert to ms */

    tph_dur += UINT32_C(1); /* Wake up duration of 1ms */
    /* The remaining time should be used for heating */
    dev->gas_sett.heatr_dur = duration - (uint16_t) tph_dur;
```

```c
}

/*!
 * @brief This API is used to get the profile duration of the sensor.
 */
void bme680_get_profile_dur(uint16_t *duration, const struct bme680_dev *dev)
{
    uint32_t tph_dur; /* Calculate in us */
    uint32_t meas_cycles;
    uint8_t os_to_meas_cycles[6] = {0, 1, 2, 4, 8, 16};

    meas_cycles = os_to_meas_cycles[dev->tph_sett.os_temp];
    meas_cycles += os_to_meas_cycles[dev->tph_sett.os_pres];
    meas_cycles += os_to_meas_cycles[dev->tph_sett.os_hum];

    /* TPH measurement duration */
    tph_dur = meas_cycles * UINT32_C(1963);
    tph_dur += UINT32_C(477 * 4); /* TPH switching duration */
    tph_dur += UINT32_C(477 * 5); /* Gas measurement duration */
    tph_dur += UINT32_C(500); /* Get it to the closest whole number.*/
    tph_dur /= UINT32_C(1000); /* Convert to ms */

    tph_dur += UINT32_C(1); /* Wake up duration of 1ms */

    *duration = (uint16_t) tph_dur;

    /* Get the gas duration only when the run gas is enabled */
    if (dev->gas_sett.run_gas) {
        /* The remaining time should be used for heating */
        *duration += dev->gas_sett.heatr_dur;
    }
}

/*!
 * @brief This API reads the pressure, temperature and humidity and gas data
 * from the sensor, compensates the data and store it in the bme680_data
 * structure instance passed by the user.
 */
int8_t bme680_get_sensor_data(struct bme680_field_data *data, struct bme680_dev *dev)
{
    int8_t rslt;

    /* Check for null pointer in the device structure*/
    rslt = null_ptr_check(dev);
    if (rslt == BME680_OK) {
        /* Reading the sensor data in forced mode only */
        rslt = read_field_data(data, dev);
        if (rslt == BME680_OK) {
            if (data->status & BME680_NEW_DATA_MSK)
                dev->new_fields = 1;
            else
                dev->new_fields = 0;
        }
    }

    return rslt;
}

/*!
 * @brief This internal API is used to read the calibrated data from the sensor.
 */
static int8_t get_calib_data(struct bme680_dev *dev)
{
    int8_t rslt;
    uint8_t coeff_array[BME680_COEFF_SIZE] = { 0 };
    uint8_t temp_var = 0; /* Temporary variable */

    /* Check for null pointer in the device structure*/
    rslt = null_ptr_check(dev);
    if (rslt == BME680_OK) {
        rslt = bme680_get_regs(BME680_COEFF_ADDR1, coeff_array, BME680_COEFF_ADDR1_LEN, dev);
        /* Append the second half in the same array */
        if (rslt == BME680_OK)
            rslt = bme680_get_regs(BME680_COEFF_ADDR2, &coeff_array[BME680_COEFF_ADDR1_LEN]
            , BME680_COEFF_ADDR2_LEN, dev);

        /* Temperature related coefficients */
        dev->calib.par_t1 = (uint16_t) (BME680_CONCAT_BYTES(coeff_array[BME680_T1_MSB_REG],
            coeff_array[BME680_T1_LSB_REG]));
        dev->calib.par_t2 = (int16_t) (BME680_CONCAT_BYTES(coeff_array[BME680_T2_MSB_REG],
            coeff_array[BME680_T2_LSB_REG]));
        dev->calib.par_t3 = (int8_t) (coeff_array[BME680_T3_REG]);

        /* Pressure related coefficients */
        dev->calib.par_p1 = (uint16_t) (BME680_CONCAT_BYTES(coeff_array[BME680_P1_MSB_REG],
            coeff_array[BME680_P1_LSB_REG]));
        dev->calib.par_p2 = (int16_t) (BME680_CONCAT_BYTES(coeff_array[BME680_P2_MSB_REG],
            coeff_array[BME680_P2_LSB_REG]));
        dev->calib.par_p3 = (int8_t) coeff_array[BME680_P3_REG];
        dev->calib.par_p4 = (int16_t) (BME680_CONCAT_BYTES(coeff_array[BME680_P4_MSB_REG],
            coeff_array[BME680_P4_LSB_REG]));
        dev->calib.par_p5 = (int16_t) (BME680_CONCAT_BYTES(coeff_array[BME680_P5_MSB_REG],
            coeff_array[BME680_P5_LSB_REG]));
        dev->calib.par_p6 = (int8_t) (coeff_array[BME680_P6_REG]);
        dev->calib.par_p7 = (int8_t) (coeff_array[BME680_P7_REG]);
        dev->calib.par_p8 = (int16_t) (BME680_CONCAT_BYTES(coeff_array[BME680_P8_MSB_REG],
            coeff_array[BME680_P8_LSB_REG]));
        dev->calib.par_p9 = (int16_t) (BME680_CONCAT_BYTES(coeff_array[BME680_P9_MSB_REG],
            coeff_array[BME680_P9_LSB_REG]));
        dev->calib.par_p10 = (uint8_t) (coeff_array[BME680_P10_REG]);

        /* Humidity related coefficients */
        dev->calib.par_h1 = (uint16_t) (((uint16_t) coeff_array[BME680_H1_MSB_REG] << BME680_HUM_REG_SHIFT_VAL)
            | (coeff_array[BME680_H1_LSB_REG] & BME680_BIT_H1_DATA_MSK));
        dev->calib.par_h2 = (uint16_t) (((uint16_t) coeff_array[BME680_H2_MSB_REG] << BME680_HUM_REG_SHIFT_VAL)
            | ((coeff_array[BME680_H2_LSB_REG]) >> BME680_HUM_REG_SHIFT_VAL));
        dev->calib.par_h3 = (int8_t) coeff_array[BME680_H3_REG];
        dev->calib.par_h4 = (int8_t) coeff_array[BME680_H4_REG];
        dev->calib.par_h5 = (int8_t) coeff_array[BME680_H5_REG];
        dev->calib.par_h6 = (uint8_t) coeff_array[BME680_H6_REG];
        dev->calib.par_h7 = (int8_t) coeff_array[BME680_H7_REG];
        /* Gas heater related coefficients */
        dev->calib.par_gh1 = (int8_t) coeff_array[BME680_GH1_REG];
        dev->calib.par_gh2 = (int16_t) (BME680_CONCAT_BYTES(coeff_array[BME680_GH2_MSB_REG],
            coeff_array[BME680_GH2_LSB_REG]));
        dev->calib.par_gh3 = (int8_t) coeff_array[BME680_GH3_REG];

        /* Other coefficients */
        if (rslt == BME680_OK) {
            rslt = bme680_get_regs(BME680_ADDR_RES_HEAT_RANGE_ADDR, &temp_var, 1, dev);

            dev->calib.res_heat_range = ((temp_var & BME680_RHRANGE_MSK) / 16);
            if (rslt == BME680_OK) {
                rslt = bme680_get_regs(BME680_ADDR_RES_HEAT_VAL_ADDR, &temp_var, 1, dev);

                dev->calib.res_heat_val = (int8_t) temp_var;
                if (rslt == BME680_OK)
                    rslt = bme680_get_regs(BME680_ADDR_RANGE_SW_ERR_ADDR, &temp_var, 1, dev);
            }
        }
        dev->calib.range_sw_err = ((int8_t) temp_var & (int8_t) BME680_RSERROR_MSK) / 16;
    }

    return rslt;
}

/*!
 * @brief This internal API is used to set the gas configuration of the sensor.
 */
static int8_t set_gas_config(struct bme680_dev *dev)
{
    int8_t rslt;

    /* Check for null pointer in the device structure*/
    rslt = null_ptr_check(dev);
    if (rslt == BME680_OK) {
```

```c
        uint8_t reg_addr[2] = {0};
        uint8_t reg_data[2] = {0};

        if (dev->power_mode == BME680_FORCED_MODE) {
            reg_addr[0] = BME680_RES_HEAT0_ADDR;
            reg_data[0] = calc_heater_res(dev->gas_sett.heatr_temp, dev);
            reg_addr[1] = BME680_GAS_WAIT0_ADDR;
            reg_data[1] = calc_heater_dur(dev->gas_sett.heatr_dur);
            dev->gas_sett.nb_conv = 0;
        } else {
            rslt = BME680_W_DEFINE_PWR_MODE;
        }
        if (rslt == BME680_OK)
            rslt = bme680_set_regs(reg_addr, reg_data, 2, dev);
    }

    return rslt;
}

/*!
 * @brief This internal API is used to get the gas configuration of the sensor.
 * @note heatr_temp and heatr_dur values are currently register data
 * and not the actual values set
 */
static int8_t get_gas_config(struct bme680_dev *dev)
{
    int8_t rslt;
    /* starting address of the register array for burst read*/
    uint8_t reg_addr1 = BME680_ADDR_SENS_CONF_START;
    uint8_t reg_addr2 = BME680_ADDR_GAS_CONF_START;
    uint8_t reg_data = 0;

    /* Check for null pointer in the device structure*/
    rslt = null_ptr_check(dev);
    if (rslt == BME680_OK) {
        if (BME680_SPI_INTF == dev->intf) {
            /* Memory page switch the SPI address*/
            rslt = set_mem_page(reg_addr1, dev);
        }

        if (rslt == BME680_OK) {
            rslt = bme680_get_regs(reg_addr1, &reg_data, 1, dev);
            if (rslt == BME680_OK) {
                dev->gas_sett.heatr_temp = reg_data;
                rslt = bme680_get_regs(reg_addr2, &reg_data, 1, dev);
                if (rslt == BME680_OK) {
                    /* Heating duration register value */
                    dev->gas_sett.heatr_dur = reg_data;
                }
            }
        }
    }

    return rslt;
}

#ifndef BME680_FLOAT_POINT_COMPENSATION

/*!
 * @brief This internal API is used to calculate the temperature value.
 */
static int16_t calc_temperature(uint32_t temp_adc, struct bme680_dev *dev)
{
    int64_t var1;
    int64_t var2;
    int64_t var3;
    int16_t calc_temp;

    var1 = ((int32_t) temp_adc >> 3) - ((int32_t) dev->calib.par_t1 << 1);
    var2 = (var1 * (int32_t) dev->calib.par_t2) >> 11;
    var3 = ((var1 >> 1) * (var1 >> 1)) >> 12;
    var3 = ((var3) * ((int32_t) dev->calib.par_t3 << 4)) >> 14;
    dev->calib.t_fine = (int32_t) (var2 + var3);
    calc_temp = (int16_t) (((dev->calib.t_fine * 5) + 128) >> 8);
    return calc_temp;
}

/*!
 * @brief This internal API is used to calculate the pressure value.
 */
static uint32_t calc_pressure(uint32_t pres_adc, const struct bme680_dev *dev)
{
    int32_t var1;
    int32_t var2;
    int32_t var3;
    int32_t pressure_comp;

    var1 = (((int32_t)dev->calib.t_fine) >> 1) - 64000;
    var2 = ((((var1 >> 2) * (var1 >> 2)) >> 11) *
        (int32_t)dev->calib.par_p6) >> 2;
    var2 = var2 + ((var1 * (int32_t)dev->calib.par_p5) << 1);
    var2 = (var2 >> 2) + ((int32_t)dev->calib.par_p4 << 16);
    var1 = (((((var1 >> 2) * (var1 >> 2)) >> 13) *
        ((int32_t)dev->calib.par_p3 << 5)) >> 3) +
        (((int32_t)dev->calib.par_p2 * var1) >> 1);
    var1 = var1 >> 18;
    var1 = ((32768 + var1) * (int32_t)dev->calib.par_p1) >> 15;
    pressure_comp = 1048576 - pres_adc;
    pressure_comp = (int32_t)((pressure_comp - (var2 >> 12)) * ((uint32_t)3125));
    if (pressure_comp >= BME680_MAX_OVERFLOW_VAL)
        pressure_comp = ((pressure_comp / var1) << 1);
    else
        pressure_comp = ((pressure_comp << 1) / var1);
    var1 = ((int32_t)dev->calib.par_p9 * (int32_t)(((pressure_comp >> 3) *
        (pressure_comp >> 3)) >> 13)) >> 12;
    var2 = ((int32_t)(pressure_comp >> 2) *
        (int32_t)dev->calib.par_p8) >> 13;
    var3 = ((int32_t)(pressure_comp >> 8) * (int32_t)(pressure_comp >> 8) *
        (int32_t)(pressure_comp >> 8) *
        (int32_t)dev->calib.par_p10) >> 17;
    pressure_comp = (int32_t)(pressure_comp) + ((var1 + var2 + var3 +
        ((int32_t)dev->calib.par_p7 << 7)) >> 4);

    return (uint32_t)pressure_comp;

}

/*!
 * @brief This internal API is used to calculate the humidity value.
 */
static uint32_t calc_humidity(uint16_t hum_adc, const struct bme680_dev *dev)
{
    int32_t var1;
    int32_t var2;
    int32_t var3;
    int32_t var4;
    int32_t var5;
    int32_t var6;
    int32_t temp_scaled;
    int32_t calc_hum;

    temp_scaled = (((int32_t) dev->calib.t_fine * 5) + 128) >> 8;
    var1 = (int32_t) (hum_adc - ((int32_t) ((int32_t) dev->calib.par_h1 * 16)))
        - (((temp_scaled * (int32_t) dev->calib.par_h3) / ((int32_t) 100)) >> 1);
    var2 = ((int32_t) dev->calib.par_h2
        * (((temp_scaled * (int32_t) dev->calib.par_h4) / ((int32_t) 100))
            + (((temp_scaled * ((temp_scaled * (int32_t) dev->calib.par_h5) / ((int32_t) 100))) >> 6)
                / ((int32_t) 100)) + (int32_t) (1 << 14))) >> 10;
    var3 = var1 * var2;
    var4 = (int32_t) dev->calib.par_h6 << 7;
    var4 = ((var4) + ((temp_scaled * (int32_t) dev->calib.par_h7) / ((int32_t) 100))) >> 4;
    var5 = ((var3 >> 14) * (var3 >> 14)) >> 10;
    var6 = (var4 * var5) >> 1;
    calc_hum = (((var3 + var6) >> 10) * ((int32_t) 1000)) >> 12;

    if (calc_hum > 100000) /* Cap at 100%rH */
```

```c
        calc_hum = 100000;
    else if (calc_hum < 0)
        calc_hum = 0;

    return (uint32_t) calc_hum;
}

/*!
 * @brief This internal API is used to calculate the Gas Resistance value.
 */
static uint32_t calc_gas_resistance(uint16_t gas_res_adc, uint8_t gas_range, const struct bme680_dev *dev)
{
    int64_t var1;
    uint64_t var2;
    int64_t var3;
    uint32_t calc_gas_res;
    /**Look up table 1 for the possible gas range values */
    uint32_t lookupTable1[16] = { UINT32_C(2147483647), UINT32_C(2147483647), UINT32_C(2147483647), UINT32_C
      (2147483647),
        UINT32_C(2147483647), UINT32_C(2126008810), UINT32_C(2147483647), UINT32_C(2130303777),
        UINT32_C(2147483647), UINT32_C(2147483647), UINT32_C(2143188679), UINT32_C(2136746228),
        UINT32_C(2147483647), UINT32_C(2126008810), UINT32_C(2147483647), UINT32_C(2147483647) };
    /**Look up table 2 for the possible gas range values */
    uint32_t lookupTable2[16] = { UINT32_C(4096000000), UINT32_C(2048000000), UINT32_C(1024000000), UINT32_C(512000000),
        UINT32_C(255744255), UINT32_C(127110228), UINT32_C(64000000), UINT32_C(32258064), UINT32_C(16016016),
        UINT32_C(8000000), UINT32_C(4000000), UINT32_C(2000000), UINT32_C(1000000), UINT32_C(500000),
        UINT32_C(250000), UINT32_C(125000) };

    var1 = (int64_t) ((1340 + (5 * (int64_t) dev->calib.range_sw_err)) *
        ((int64_t) lookupTable1[gas_range])) >> 16;
    var2 = (((int64_t) ((int64_t) gas_res_adc << 15) - (int64_t) (16777216)) + var1);
    var3 = (((int64_t) lookupTable2[gas_range] * (int64_t) var1) >> 9);
    calc_gas_res = (uint32_t) ((var3 + ((int64_t) var2 >> 1)) / (int64_t) var2);

    return calc_gas_res;
}
```

```c
/*!
 * @brief This internal API is used to calculate the Heat Resistance value.
 */
static uint8_t calc_heater_res(uint16_t temp, const struct bme680_dev *dev)
{
    uint8_t heatr_res;
    int32_t var1;
    int32_t var2;
    int32_t var3;
    int32_t var4;
    int32_t var5;
    int32_t heatr_res_x100;

    if (temp > 400) /* Cap temperature */
        temp = 400;

    var1 = (((int32_t) dev->amb_temp * dev->calib.par_gh3) / 1000) * 256;
    var2 = (dev->calib.par_gh1 + 784) * (((((dev->calib.par_gh2 + 154009) * temp * 5) / 100) + 3276800) / 10);
    var3 = var1 + (var2 / 2);
    var4 = (var3 / (dev->calib.res_heat_range + 4));
    var5 = (131 * dev->calib.res_heat_val) + 65536;
    heatr_res_x100 = (int32_t) (((var4 / var5) - 250) * 34);
    heatr_res = (uint8_t) ((heatr_res_x100 + 50) / 100);

    return heatr_res;
}

#else


/*!
 * @brief This internal API is used to calculate the
 * temperature value in float format
 */
static float calc_temperature(uint32_t temp_adc, struct bme680_dev *dev)
{
    float var1 = 0;
    float var2 = 0;
    float calc_temp = 0;

    /* calculate var1 data */
    var1  = ((((float)temp_adc / 16384.0f) - ((float)dev->calib.par_t1 / 1024.0f))
            * ((float)dev->calib.par_t2));

    /* calculate var2 data */
    var2  = (((((float)temp_adc / 131072.0f) - ((float)dev->calib.par_t1 / 8192.0f)) *
        (((float)temp_adc / 131072.0f) - ((float)dev->calib.par_t1 / 8192.0f))) *
        ((float)dev->calib.par_t3 * 16.0f));

    /* t_fine value*/
    dev->calib.t_fine = (var1 + var2);

    /* compensated temperature data*/
    calc_temp  = ((dev->calib.t_fine) / 5120.0f);

    return calc_temp;
}

/*!
 * @brief This internal API is used to calculate the
 * pressure value in float format
 */
static float calc_pressure(uint32_t pres_adc, const struct bme680_dev *dev)
{
    float var1 = 0;
    float var2 = 0;
    float var3 = 0;
    float calc_pres = 0;

    var1 = (((float)dev->calib.t_fine / 2.0f) - 64000.0f);
    var2 = var1 * var1 * (((float)dev->calib.par_p6) / (131072.0f));
    var2 = var2 + (var1 * ((float)dev->calib.par_p5) * 2.0f);
    var2 = (var2 / 4.0f) + (((float)dev->calib.par_p4) * 65536.0f);
    var1 = (((((float)dev->calib.par_p3 * var1 * var1) / 16384.0f)
        + ((float)dev->calib.par_p2 * var1)) / 524288.0f);
    var1 = ((1.0f + (var1 / 32768.0f)) * ((float)dev->calib.par_p1));
    calc_pres = (1048576.0f - ((float)pres_adc));

    /* Avoid exception caused by division by zero */
    if ((int)var1 != 0) {
        calc_pres = (((calc_pres - (var2 / 4096.0f)) * 6250.0f) / var1);
        var1 = (((float)dev->calib.par_p9) * calc_pres * calc_pres) / 2147483648.0f;
        var2 = calc_pres * (((float)dev->calib.par_p8) / 32768.0f);
        var3 = ((calc_pres / 256.0f) * (calc_pres / 256.0f) * (calc_pres / 256.0f)
            * (dev->calib.par_p10 / 131072.0f));
        calc_pres = (calc_pres + (var1 + var2 + var3 + ((float)dev->calib.par_p7 * 128.0f)) / 16.0f);
    } else {
        calc_pres = 0;
    }

    return calc_pres;
}

/*!
 * @brief This internal API is used to calculate the
 * humidity value in float format
 */
static float calc_humidity(uint16_t hum_adc, const struct bme680_dev *dev)
{
    float calc_hum = 0;
    float var1 = 0;
    float var2 = 0;
    float var3 = 0;
    float var4 = 0;
    float temp_comp;

    /* compensated temperature data*/
    temp_comp  = ((dev->calib.t_fine) / 5120.0f);

    var1 = (float)((float)hum_adc) - (((float)dev->calib.par_h1 * 16.0f) + (((float)dev->calib.par_h3 / 2.0f)
        * temp_comp));
    var2 = var1 * ((float)(((float) dev->calib.par_h2 / 262144.0f) * (1.0f + (((float)dev->calib.par_h4 / 16384.0f)
        * temp_comp) + (((float)dev->calib.par_h5 / 1048576.0f) * temp_comp * temp_comp))));

    var3 = (float) dev->calib.par_h6 / 16384.0f;

    var4 = (float) dev->calib.par_h7 / 2097152.0f;

    calc_hum = var2 + ((var3 + (var4 * temp_comp)) * var2 * var2);

    if (calc_hum > 100.0f)
        calc_hum = 100.0f;
    else if (calc_hum < 0.0f)
        calc_hum = 0.0f;

    return calc_hum;
}

/*!
 * @brief This internal API is used to calculate the
 * gas resistance value in float format
 */
static float calc_gas_resistance(uint16_t gas_res_adc, uint8_t gas_range, const struct bme680_dev *dev)
{
    float calc_gas_res;
    float var1 = 0;
    float var2 = 0;
    float var3 = 0;

    const float lookup_k1_range[16] = {
    0.0, 0.0, 0.0, 0.0, 0.0, -1.0, 0.0, -0.8,
    0.0, 0.0, -0.2, -0.5, 0.0, -1.0, 0.0, 0.0};
    const float lookup_k2_range[16] = {
    0.0, 0.0, 0.0, 0.0, 0.1, 0.7, 0.0, -0.8,
    -0.1, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};

    var1 = (1340.0f + (5.0f * dev->calib.range_sw_err));
    var2 = (var1) * (1.0f + lookup_k1_range[gas_range]/100.0f);
    var3 = 1.0f + (lookup_k2_range[gas_range]/100.0f);

    calc_gas_res = 1.0f / (float)(var3 * (0.000000125f) * (float)(1 << gas_range) * (((((float)gas_res_adc)
        - 512.0f)/var2) + 1.0f));

    return calc_gas_res;
}

/*!
 * @brief This internal API is used to calculate the
 * heater resistance value in float format
 */
static float calc_heater_res(uint16_t temp, const struct bme680_dev *dev)
{
    float var1 = 0;
    float var2 = 0;
    float var3 = 0;
    float var4 = 0;
    float var5 = 0;
    float res_heat = 0;

    if (temp > 400) /* Cap temperature */
        temp = 400;

    var1 = (((float)dev->calib.par_gh1 / (16.0f)) + 49.0f);
    var2 = ((((float)dev->calib.par_gh2 / (32768.0f)) * (0.0005f)) + 0.00235f);
    var3 = ((float)dev->calib.par_gh3 / (1024.0f));
    var4 = (var1 * (1.0f + (var2 * (float)temp)));
    var5 = (var4 + (var3 * (float)dev->amb_temp));
    res_heat = (uint8_t)(3.4f * ((var5 * (4 / (4 + (float)dev->calib.res_heat_range)) *
        (1/(1 + ((float) dev->calib.res_heat_val * 0.002f)))) - 25));

    return res_heat;
}

#endif
```

```
/*!
 * @brief This internal API is used to calculate the Heat duration value.
 */
static uint8_t calc_heater_dur(uint16_t dur)
{
    uint8_t factor = 0;
    uint8_t durval;

    if (dur >= 0xfc0) {
        durval = 0xff; /* Max duration*/
    } else {
        while (dur > 0x3F) {
            dur = dur / 4;
            factor += 1;
        }
        durval = (uint8_t) (dur + (factor * 64));
    }

    return durval;
}

/*!
 * @brief This internal API is used to calculate the field data of sensor.
 */
static int8_t read_field_data(struct bme680_field_data *data, struct bme680_dev *dev)
{
    int8_t rslt;
    uint8_t buff[BME680_FIELD_LENGTH] = { 0 };
    uint8_t gas_range;
    uint32_t adc_temp;
    uint32_t adc_pres;
    uint16_t adc_hum;
    uint16_t adc_gas_res;
    uint8_t tries = 10;

    /* Check for null pointer in the device structure*/
    rslt = null_ptr_check(dev);
    do {
        if (rslt == BME680_OK) {
            rslt = bme680_get_regs(((uint8_t) (BME680_FIELD0_ADDR)), buff, (uint16_t) BME680_FIELD_LENGTH,
                dev);

            data->status = buff[0] & BME680_NEW_DATA_MSK;
            data->gas_index = buff[0] & BME680_GAS_INDEX_MSK;
            data->meas_index = buff[1];

            /* read the raw data from the sensor */
            adc_pres = (uint32_t) (((uint32_t) buff[2] * 4096) | ((uint32_t) buff[3] * 16)
                | ((uint32_t) buff[4] / 16));
            adc_temp = (uint32_t) (((uint32_t) buff[5] * 4096) | ((uint32_t) buff[6] * 16)
                | ((uint32_t) buff[7] / 16));
            adc_hum = (uint16_t) (((uint32_t) buff[8] * 256) | (uint32_t) buff[9]);
            adc_gas_res = (uint16_t) ((uint32_t) buff[13] * 4 | (((uint32_t) buff[14]) / 64));
            gas_range = buff[14] & BME680_GAS_RANGE_MSK;

            data->status |= buff[14] & BME680_GASM_VALID_MSK;
            data->status |= buff[14] & BME680_HEAT_STAB_MSK;

            if (data->status & BME680_NEW_DATA_MSK) {
                data->temperature = calc_temperature(adc_temp, dev);
                data->pressure = calc_pressure(adc_pres, dev);
                data->humidity = calc_humidity(adc_hum, dev);
                data->gas_resistance = calc_gas_resistance(adc_gas_res, gas_range, dev);
                break;
            }
            /* Delay to poll the data */
            dev->delay_ms(BME680_POLL_PERIOD_MS);
        }
        tries--;
    } while (tries);

    if (!tries)
        rslt = BME680_W_NO_NEW_DATA;
```

```c
    return rslt;
}

/*!
 * @brief This internal API is used to set the memory page based on register address.
 */
static int8_t set_mem_page(uint8_t reg_addr, struct bme680_dev *dev)
{
    int8_t rslt;
    uint8_t reg;
    uint8_t mem_page;

    /* Check for null pointers in the device structure*/
    rslt = null_ptr_check(dev);
    if (rslt == BME680_OK) {
        if (reg_addr > 0x7f)
            mem_page = BME680_MEM_PAGE1;
        else
            mem_page = BME680_MEM_PAGE0;

        if (mem_page != dev->mem_page) {
            dev->mem_page = mem_page;

            dev->com_rslt = dev->read(dev->dev_id, BME680_MEM_PAGE_ADDR | BME680_SPI_RD_MSK, &reg, 1);
            if (dev->com_rslt != 0)
                rslt = BME680_E_COM_FAIL;

            if (rslt == BME680_OK) {
                reg = reg & (~BME680_MEM_PAGE_MSK);
                reg = reg | (dev->mem_page & BME680_MEM_PAGE_MSK);

                dev->com_rslt = dev->write(dev->dev_id, BME680_MEM_PAGE_ADDR & BME680_SPI_WR_MSK,
                    &reg, 1);
                if (dev->com_rslt != 0)
                    rslt = BME680_E_COM_FAIL;
            }
        }
    }

    return rslt;
}

/*!
 * @brief This internal API is used to get the memory page based on register address.
 */
static int8_t get_mem_page(struct bme680_dev *dev)
{
    int8_t rslt;
    uint8_t reg;

    /* Check for null pointer in the device structure*/
    rslt = null_ptr_check(dev);
    if (rslt == BME680_OK) {
        dev->com_rslt = dev->read(dev->dev_id, BME680_MEM_PAGE_ADDR | BME680_SPI_RD_MSK, &reg, 1);
        if (dev->com_rslt != 0)
            rslt = BME680_E_COM_FAIL;
        else
            dev->mem_page = reg & BME680_MEM_PAGE_MSK;
    }

    return rslt;
}

/*!
 * @brief This internal API is used to validate the boundary
 * conditions.
 */
static int8_t boundary_check(uint8_t *value, uint8_t min, uint8_t max, struct bme680_dev *dev)
{
    int8_t rslt = BME680_OK;

    if (value != NULL) {
        /* Check if value is below minimum value */
```

```c
        if (*value < min) {
            /* Auto correct the invalid value to minimum value */
            *value = min;
            dev->info_msg |= BME680_I_MIN_CORRECTION;
        }
        /* Check if value is above maximum value */
        if (*value > max) {
            /* Auto correct the invalid value to maximum value */
            *value = max;
            dev->info_msg |= BME680_I_MAX_CORRECTION;
        }
    } else {
        rslt = BME680_E_NULL_PTR;
    }

    return rslt;
}

/*!
 * @brief This internal API is used to validate the device structure pointer for
 * null conditions.
 */
static int8_t null_ptr_check(const struct bme680_dev *dev)
{
    int8_t rslt;

    if ((dev == NULL) || (dev->read == NULL) || (dev->write == NULL) || (dev->delay_ms == NULL)) {
        /* Device structure pointer is not valid */
        rslt = BME680_E_NULL_PTR;
    } else {
        /* Device structure is fine */
        rslt = BME680_OK;
    }

    return rslt;
}
```

```
 * other rights of third parties which may result from its use.
 * No license is granted by implication or otherwise under any patent or
 * patent rights of the copyright holder.
 *
 * @file    bme680_defs.h
 * @date    19 Jun 2018
 * @version 3.5.9
 * @brief
 *
 */

/*! @file bme680_defs.h
 @brief Sensor driver for BME680 sensor */
/*!
 * @defgroup BME680 SENSOR API
 * @brief
 * @{*/
#ifndef BME680_DEFS_H_
#define BME680_DEFS_H_

/********************************************************/
/* header includes */
#ifdef __KERNEL__
#include <linux/types.h>
#include <linux/kernel.h>
#else
#include <stdint.h>
#include <stddef.h>
#endif

/******************************************************************************/
/*! @name       Common macros                        */
/******************************************************************************/

#if !defined(UINT8_C) && !defined(INT8_C)
#define INT8_C(x)       S8_C(x)
#define UINT8_C(x)      U8_C(x)
#endif

#if !defined(UINT16_C) && !defined(INT16_C)
#define INT16_C(x)      S16_C(x)
#define UINT16_C(x)     U16_C(x)
#endif

#if !defined(INT32_C) && !defined(UINT32_C)
#define INT32_C(x)      S32_C(x)
#define UINT32_C(x)     U32_C(x)
#endif

#if !defined(INT64_C) && !defined(UINT64_C)
#define INT64_C(x)      S64_C(x)
#define UINT64_C(x)     U64_C(x)
#endif

/**@}*/

/**\name C standard macros */
#ifndef NULL
#ifdef __cplusplus
#define NULL   0
#else
#define NULL   ((void *) 0)
#endif
#endif

/** BME680 configuration macros */
/** Enable or un-comment the macro to provide floating point data output */
#ifndef BME680_FLOAT_POINT_COMPENSATION
//#define BME680_FLOAT_POINT_COMPENSATION */
#endif

/** BME680 General config */
#define BME680_POLL_PERIOD_MS       UINT8_C(10)
```

```c
/** BME680 I2C addresses */
#define BME680_I2C_ADDR_PRIMARY     UINT8_C(0x76)
#define BME680_I2C_ADDR_SECONDARY   UINT8_C(0x77)

/** BME680 unique chip identifier */
#define BME680_CHIP_ID  UINT8_C(0x61)

/** BME680 coefficients related defines */
#define BME680_COEFF_SIZE       UINT8_C(41)
#define BME680_COEFF_ADDR1_LEN      UINT8_C(25)
#define BME680_COEFF_ADDR2_LEN      UINT8_C(16)

/** BME680 field_x related defines */
#define BME680_FIELD_LENGTH     UINT8_C(15)
#define BME680_FIELD_ADDR_OFFSET    UINT8_C(17)

/** Soft reset command */
#define BME680_SOFT_RESET_CMD   UINT8_C(0xb6)

/** Error code definitions */
#define BME680_OK       INT8_C(0)
/* Errors */
#define BME680_E_NULL_PTR           INT8_C(-1)
#define BME680_E_COM_FAIL           INT8_C(-2)
#define BME680_E_DEV_NOT_FOUND      INT8_C(-3)
#define BME680_E_INVALID_LENGTH     INT8_C(-4)

/* Warnings */
#define BME680_W_DEFINE_PWR_MODE    INT8_C(1)
#define BME680_W_NO_NEW_DATA        INT8_C(2)

/* Info's */
#define BME680_I_MIN_CORRECTION     UINT8_C(1)
#define BME680_I_MAX_CORRECTION     UINT8_C(2)

/** Register map */
/** Other coefficient's address */
```

```
#define BME680_ADDR_RES_HEAT_VAL_ADDR   UINT8_C(0x00)
#define BME680_ADDR_RES_HEAT_RANGE_ADDR UINT8_C(0x02)
#define BME680_ADDR_RANGE_SW_ERR_ADDR   UINT8_C(0x04)
#define BME680_ADDR_SENS_CONF_START UINT8_C(0x5A)
#define BME680_ADDR_GAS_CONF_START  UINT8_C(0x64)

/** Field settings */
#define BME680_FIELD0_ADDR      UINT8_C(0x1d)

/** Heater settings */
#define BME680_RES_HEAT0_ADDR       UINT8_C(0x5a)
#define BME680_GAS_WAIT0_ADDR       UINT8_C(0x64)

/** Sensor configuration registers */
#define BME680_CONF_HEAT_CTRL_ADDR      UINT8_C(0x70)
#define BME680_CONF_ODR_RUN_GAS_NBC_ADDR    UINT8_C(0x71)
#define BME680_CONF_OS_H_ADDR           UINT8_C(0x72)
#define BME680_MEM_PAGE_ADDR            UINT8_C(0xf3)
#define BME680_CONF_T_P_MODE_ADDR       UINT8_C(0x74)
#define BME680_CONF_ODR_FILT_ADDR       UINT8_C(0x75)

/** Coefficient's address */
#define BME680_COEFF_ADDR1  UINT8_C(0x89)
#define BME680_COEFF_ADDR2  UINT8_C(0xe1)

/** Chip identifier */
#define BME680_CHIP_ID_ADDR UINT8_C(0xd0)

/** Soft reset register */
#define BME680_SOFT_RESET_ADDR      UINT8_C(0xe0)

/** Heater control settings */
#define BME680_ENABLE_HEATER        UINT8_C(0x00)
#define BME680_DISABLE_HEATER       UINT8_C(0x08)

/** Gas measurement settings */
#define BME680_DISABLE_GAS_MEAS     UINT8_C(0x00)
```

```
#define BME680_ENABLE_GAS_MEAS      UINT8_C(0x01)

/** Over-sampling settings */
#define BME680_OS_NONE      UINT8_C(0)
#define BME680_OS_1X        UINT8_C(1)
#define BME680_OS_2X        UINT8_C(2)
#define BME680_OS_4X        UINT8_C(3)
#define BME680_OS_8X        UINT8_C(4)
#define BME680_OS_16X       UINT8_C(5)

/** IIR filter settings */
#define BME680_FILTER_SIZE_0    UINT8_C(0)
#define BME680_FILTER_SIZE_1    UINT8_C(1)
#define BME680_FILTER_SIZE_3    UINT8_C(2)
#define BME680_FILTER_SIZE_7    UINT8_C(3)
#define BME680_FILTER_SIZE_15   UINT8_C(4)
#define BME680_FILTER_SIZE_31   UINT8_C(5)
#define BME680_FILTER_SIZE_63   UINT8_C(6)
#define BME680_FILTER_SIZE_127  UINT8_C(7)

/** Power mode settings */
#define BME680_SLEEP_MODE   UINT8_C(0)
#define BME680_FORCED_MODE  UINT8_C(1)

/** Delay related macro declaration */
#define BME680_RESET_PERIOD UINT32_C(10)

/** SPI memory page settings */
#define BME680_MEM_PAGE0    UINT8_C(0x10)
#define BME680_MEM_PAGE1    UINT8_C(0x00)

/** Ambient humidity shift value for compensation */
#define BME680_HUM_REG_SHIFT_VAL    UINT8_C(4)

/** Run gas enable and disable settings */
#define BME680_RUN_GAS_DISABLE  UINT8_C(0)
#define BME680_RUN_GAS_ENABLE   UINT8_C(1)
```

```
/** Buffer length macro declaration */
#define BME680_TMP_BUFFER_LENGTH    UINT8_C(40)
#define BME680_REG_BUFFER_LENGTH    UINT8_C(6)
#define BME680_FIELD_DATA_LENGTH    UINT8_C(3)
#define BME680_GAS_REG_BUF_LENGTH   UINT8_C(20)

/** Settings selector */
#define BME680_OST_SEL          UINT16_C(1)
#define BME680_OSP_SEL          UINT16_C(2)
#define BME680_OSH_SEL          UINT16_C(4)
#define BME680_GAS_MEAS_SEL     UINT16_C(8)
#define BME680_FILTER_SEL       UINT16_C(16)
#define BME680_HCNTRL_SEL       UINT16_C(32)
#define BME680_RUN_GAS_SEL      UINT16_C(64)
#define BME680_NBCONV_SEL       UINT16_C(128)
#define BME680_GAS_SENSOR_SEL       (BME680_GAS_MEAS_SEL | BME680_RUN_GAS_SEL | BME680_NBCONV_SEL)

/** Number of conversion settings*/
#define BME680_NBCONV_MIN       UINT8_C(0)
#define BME680_NBCONV_MAX       UINT8_C(10)

/** Mask definitions */
#define BME680_GAS_MEAS_MSK UINT8_C(0x30)
#define BME680_NBCONV_MSK   UINT8_C(0X0F)
#define BME680_FILTER_MSK   UINT8_C(0X1C)
#define BME680_OST_MSK      UINT8_C(0XE0)
#define BME680_OSP_MSK      UINT8_C(0X1C)
#define BME680_OSH_MSK      UINT8_C(0X07)
#define BME680_HCTRL_MSK    UINT8_C(0x08)
#define BME680_RUN_GAS_MSK  UINT8_C(0x10)
#define BME680_MODE_MSK     UINT8_C(0x03)
#define BME680_RHRANGE_MSK  UINT8_C(0x30)
#define BME680_RSERROR_MSK  UINT8_C(0xf0)
#define BME680_NEW_DATA_MSK UINT8_C(0x80)
#define BME680_GAS_INDEX_MSK    UINT8_C(0x0f)
#define BME680_GAS_RANGE_MSK    UINT8_C(0x0f)
```

```
#define BME680_GASM_VALID_MSK   UINT8_C(0x20)
#define BME680_HEAT_STAB_MSK    UINT8_C(0x10)
#define BME680_MEM_PAGE_MSK UINT8_C(0x10)
#define BME680_SPI_RD_MSK   UINT8_C(0x80)
#define BME680_SPI_WR_MSK   UINT8_C(0x7f)
#define BME680_BIT_H1_DATA_MSK  UINT8_C(0x0F)

/** Bit position definitions for sensor settings */
#define BME680_GAS_MEAS_POS UINT8_C(4)
#define BME680_FILTER_POS   UINT8_C(2)
#define BME680_OST_POS      UINT8_C(5)
#define BME680_OSP_POS      UINT8_C(2)
#define BME680_RUN_GAS_POS  UINT8_C(4)

/** Array Index to Field data mapping for Calibration Data*/
#define BME680_T2_LSB_REG   (1)
#define BME680_T2_MSB_REG   (2)
#define BME680_T3_REG       (3)
#define BME680_P1_LSB_REG   (5)
#define BME680_P1_MSB_REG   (6)
#define BME680_P2_LSB_REG   (7)
#define BME680_P2_MSB_REG   (8)
#define BME680_P3_REG       (9)
#define BME680_P4_LSB_REG   (11)
#define BME680_P4_MSB_REG   (12)
#define BME680_P5_LSB_REG   (13)
#define BME680_P5_MSB_REG   (14)
#define BME680_P7_REG       (15)
#define BME680_P6_REG       (16)
#define BME680_P8_LSB_REG   (19)
#define BME680_P8_MSB_REG   (20)
#define BME680_P9_LSB_REG   (21)
#define BME680_P9_MSB_REG   (22)
#define BME680_P10_REG      (23)
#define BME680_H2_MSB_REG   (25)
#define BME680_H2_LSB_REG   (26)
#define BME680_H1_LSB_REG   (26)
```

```c
#define BME680_H1_MSB_REG   (27)
#define BME680_H3_REG       (28)
#define BME680_H4_REG       (29)
#define BME680_H5_REG       (30)
#define BME680_H6_REG       (31)
#define BME680_H7_REG       (32)
#define BME680_T1_LSB_REG   (33)
#define BME680_T1_MSB_REG   (34)
#define BME680_GH2_LSB_REG  (35)
#define BME680_GH2_MSB_REG  (36)
#define BME680_GH1_REG      (37)
#define BME680_GH3_REG      (38)

/** BME680 register buffer index settings*/
#define BME680_REG_FILTER_INDEX     UINT8_C(5)
#define BME680_REG_TEMP_INDEX       UINT8_C(4)
#define BME680_REG_PRES_INDEX       UINT8_C(4)
#define BME680_REG_HUM_INDEX        UINT8_C(2)
#define BME680_REG_NBCONV_INDEX     UINT8_C(1)
#define BME680_REG_RUN_GAS_INDEX    UINT8_C(1)
#define BME680_REG_HCTRL_INDEX      UINT8_C(0)

/** BME680 pressure calculation macros */
/*! This max value is used to provide precedence to multiplication or division
 * in pressure compensation equation to achieve least loss of precision and
 * avoiding overflows.
 * i.e Comparing value, BME680_MAX_OVERFLOW_VAL = INT32_C(1 << 30)
 */
#define BME680_MAX_OVERFLOW_VAL      INT32_C(0x40000000)

/** Macro to combine two 8 bit data's to form a 16 bit data */
#define BME680_CONCAT_BYTES(msb, lsb)   (((uint16_t)msb << 8) | (uint16_t)lsb)

/** Macro to SET and GET BITS of a register */
#define BME680_SET_BITS(reg_data, bitname, data) \
        ((reg_data & ~(bitname##_MSK)) | \
        ((data << bitname##_POS) & bitname##_MSK))
```

```
#define BME680_GET_BITS(reg_data, bitname)  ((reg_data & (bitname##_MSK)) >> \
    (bitname##_POS))

/** Macro variant to handle the bitname position if it is zero */
#define BME680_SET_BITS_POS_0(reg_data, bitname, data) \
                ((reg_data & ~(bitname##_MSK)) | \
                (data & bitname##_MSK))
#define BME680_GET_BITS_POS_0(reg_data, bitname)  (reg_data & (bitname##_MSK))

/** Type definitions */
/*!
 * Generic communication function pointer
 * @param[in] dev_id: Place holder to store the id of the device structure
 *                    Can be used to store the index of the Chip select or
 *                    I2C address of the device.
 * @param[in] reg_addr: Used to select the register the where data needs to
 *                      be read from or written to.
 * @param[in/out] reg_data: Data array to read/write
 * @param[in] len: Length of the data array
 */
typedef int8_t (*bme680_com_fptr_t)(uint8_t dev_id, uint8_t reg_addr, uint8_t *data, uint16_t len);

/*!
 * Delay function pointer
 * @param[in] period: Time period in milliseconds
 */
typedef void (*bme680_delay_fptr_t)(uint32_t period);

/*!
 * @brief Interface selection Enumerations
 */
enum bme680_intf {
    /*! SPI interface */
    BME680_SPI_INTF,
    /*! I2C interface */
    BME680_I2C_INTF
};
```

```
/* structure definitions */
/*!
 * @brief Sensor field data structure
 */
struct  bme680_field_data {
    /*! Contains new_data, gasm_valid & heat_stab */
    uint8_t status;
    /*! The index of the heater profile used */
    uint8_t gas_index;
    /*! Measurement index to track order */
    uint8_t meas_index;

#ifndef BME680_FLOAT_POINT_COMPENSATION
    /*! Temperature in degree celsius x100 */
    int16_t temperature;
    /*! Pressure in Pascal */
    uint32_t pressure;
    /*! Humidity in % relative humidity x1000 */
    uint32_t humidity;
    /*! Gas resistance in Ohms */
    uint32_t gas_resistance;
#else
    /*! Temperature in degree celsius */
    float temperature;
    /*! Pressure in Pascal */
    float pressure;
    /*! Humidity in % relative humidity x1000 */
    float humidity;
    /*! Gas resistance in Ohms */
    float gas_resistance;

#endif

};

/*!
```

```
 * @brief Structure to hold the Calibration data
 */
struct  bme680_calib_data {
    /*! Variable to store calibrated humidity data */
    uint16_t par_h1;
    /*! Variable to store calibrated humidity data */
    uint16_t par_h2;
    /*! Variable to store calibrated humidity data */
    int8_t par_h3;
    /*! Variable to store calibrated humidity data */
    int8_t par_h4;
    /*! Variable to store calibrated humidity data */
    int8_t par_h5;
    /*! Variable to store calibrated humidity data */
    uint8_t par_h6;
    /*! Variable to store calibrated humidity data */
    int8_t par_h7;
    /*! Variable to store calibrated gas data */
    int8_t par_gh1;
    /*! Variable to store calibrated gas data */
    int16_t par_gh2;
    /*! Variable to store calibrated gas data */
    int8_t par_gh3;
    /*! Variable to store calibrated temperature data */
    uint16_t par_t1;
    /*! Variable to store calibrated temperature data */
    int16_t par_t2;
    /*! Variable to store calibrated temperature data */
    int8_t par_t3;
    /*! Variable to store calibrated pressure data */
    uint16_t par_p1;
    /*! Variable to store calibrated pressure data */
    int16_t par_p2;
    /*! Variable to store calibrated pressure data */
    int8_t par_p3;
    /*! Variable to store calibrated pressure data */
    int16_t par_p4;
    /*! Variable to store calibrated pressure data */
    int16_t par_p5;
    /*! Variable to store calibrated pressure data */
    int8_t par_p6;
    /*! Variable to store calibrated pressure data */
    int8_t par_p7;
    /*! Variable to store calibrated pressure data */
    int16_t par_p8;
    /*! Variable to store calibrated pressure data */
    int16_t par_p9;
    /*! Variable to store calibrated pressure data */
    uint8_t par_p10;

#ifndef BME680_FLOAT_POINT_COMPENSATION
    /*! Variable to store t_fine size */
    int32_t t_fine;
#else
    /*! Variable to store t_fine size */
    float t_fine;
#endif
    /*! Variable to store heater resistance range */
    uint8_t res_heat_range;
    /*! Variable to store heater resistance value */
    int8_t res_heat_val;
    /*! Variable to store error range */
    int8_t range_sw_err;
};

/*!
 * @brief BME680 sensor settings structure which comprises of ODR,
 * over-sampling and filter settings.
 */
struct  bme680_tph_sett {
    /*! Humidity oversampling */
    uint8_t os_hum;
    /*! Temperature oversampling */
    uint8_t os_temp;
    /*! Pressure oversampling */
    uint8_t os_pres;
    /*! Filter coefficient */
    uint8_t filter;
};

/*!
 * @brief BME680 gas sensor which comprises of gas settings
 *  and status parameters
 */
struct  bme680_gas_sett {
    /*! Variable to store nb conversion */
    uint8_t nb_conv;
    /*! Variable to store heater control */
    uint8_t heatr_ctrl;
    /*! Run gas enable value */
    uint8_t run_gas;
    /*! Heater temperature value */
    uint16_t heatr_temp;
    /*! Duration profile value */
    uint16_t heatr_dur;
};

/*!
 * @brief BME680 device structure
 */
struct  bme680_dev {
    /*! Chip Id */
    uint8_t chip_id;
    /*! Device Id */
    uint8_t dev_id;
    /*! SPI/I2C interface */
    enum bme680_intf intf;
    /*! Memory page used */
    uint8_t mem_page;
    /*! Ambient temperature in Degree C */
    int8_t amb_temp;
    /*! Sensor calibration data */
    struct bme680_calib_data calib;
    /*! Sensor settings */
    struct bme680_tph_sett tph_sett;
    /*! Gas Sensor settings */
    struct bme680_gas_sett gas_sett;
    /*! Sensor power modes */
    uint8_t power_mode;
    /*! New sensor fields */
    uint8_t new_fields;
    /*! Store the info messages */
    uint8_t info_msg;
    /*! Bus read function pointer */
    bme680_com_fptr_t read;
    /*! Bus write function pointer */
    bme680_com_fptr_t write;
    /*! delay function pointer */
    bme680_delay_fptr_t delay_ms;
    /*! Communication function result */
    int8_t com_rslt;
};


#endif /* BME680_DEFS_H_ */
/** @}*/
/** @}*/
```

```
/**
 * Copyright (C) 2017 - 2018 Bosch Sensortec GmbH
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 *
 * Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in the
 * documentation and/or other materials provided with the distribution.
 *
 * Neither the name of the copyright holder nor the names of the
 * contributors may be used to endorse or promote products derived from
 * this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
 * CONTRIBUTORS "AS IS" AND ANY EXPRESS OR
 * IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL COPYRIGHT HOLDER
 * OR CONTRIBUTORS BE LIABLE FOR ANY
 * DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
 * OR CONSEQUENTIAL DAMAGES(INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 * ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE
 *
 * The information provided is believed to be accurate and reliable.
 * The copyright holder assumes no responsibility
 * for the consequences of use
 * of such information nor for any infringement of patents or
 * other rights of third parties which may result from its use.
 * No license is granted by implication or otherwise under any patent or
 * patent rights of the copyright holder.
 *
 * @file    bme680.h
 * @date    19 Jun 2018
 * @version 3.5.9
 * @brief
 *
 */
/*! @file bme680.h
 @brief Sensor driver for BME680 sensor */
/*!
 * @defgroup BME680 SENSOR API
 * @{*/
#ifndef BME680_H_
#define BME680_H_

/*! CPP guard */
#ifdef __cplusplus
extern "C"
{
#endif

/* Header includes */
#include "bme680_defs.h"

/* function prototype declarations */
/*!
 *  @brief This API is the entry point.
 *  It reads the chip-id and calibration data from the sensor.
 *
 *  @param[in,out] dev : Structure instance of bme680_dev
 *
 *  @return Result of API execution status
 *  @retval zero -> Success / +ve value -> Warning / -ve value -> Error
 */
```

```c
int8_t bme680_init(struct bme680_dev *dev);

/*!
 * @brief This API writes the given data to the register address
 * of the sensor.
 *
 * @param[in] reg_addr : Register address from where the data to be written.
 * @param[in] reg_data : Pointer to data buffer which is to be written
 * in the sensor.
 * @param[in] len : No of bytes of data to write..
 * @param[in] dev : Structure instance of bme680_dev.
 *
 * @return Result of API execution status
 * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
 */
int8_t bme680_set_regs(const uint8_t *reg_addr, const uint8_t *reg_data, uint8_t len, struct bme680_dev *dev);

/*!
 * @brief This API reads the data from the given register address of the sensor.
 *
 * @param[in] reg_addr : Register address from where the data to be read
 * @param[out] reg_data : Pointer to data buffer to store the read data.
 * @param[in] len : No of bytes of data to be read.
 * @param[in] dev : Structure instance of bme680_dev.
 *
 * @return Result of API execution status
 * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
 */
int8_t bme680_get_regs(uint8_t reg_addr, uint8_t *reg_data, uint16_t len, struct bme680_dev *dev);

/*!
 * @brief This API performs the soft reset of the sensor.
 *
 * @param[in] dev : Structure instance of bme680_dev.
 *
 * @return Result of API execution status
 * @retval zero -> Success / +ve value -> Warning / -ve value -> Error.
 */
int8_t bme680_soft_reset(struct bme680_dev *dev);

/*!
 * @brief This API is used to set the power mode of the sensor.
 *
 * @param[in] dev : Structure instance of bme680_dev
 * @note : Pass the value to bme680_dev.power_mode structure variable.
 *
 *  value   |   mode
 * -------------|------------------
 *  0x00    |   BME680_SLEEP_MODE
 *  0x01    |   BME680_FORCED_MODE
 *
 * * @return Result of API execution status
 * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
 */
int8_t bme680_set_sensor_mode(struct bme680_dev *dev);

/*!
 * @brief This API is used to get the power mode of the sensor.
 *
 * @param[in] dev : Structure instance of bme680_dev
 * @note : bme680_dev.power_mode structure variable hold the power mode.
 *
 *  value   |   mode
 * ---------|------------------
 *  0x00    |   BME680_SLEEP_MODE
 *  0x01    |   BME680_FORCED_MODE
 *
 * @return Result of API execution status
 * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
 */
int8_t bme680_get_sensor_mode(struct bme680_dev *dev);

/*!
 * @brief This API is used to set the profile duration of the sensor.
 *
 * @param[in] dev      : Structure instance of bme680_dev.
 * @param[in] duration : Duration of the measurement in ms.
 *
 * @return Nothing
 */
void bme680_set_profile_dur(uint16_t duration, struct bme680_dev *dev);

/*!
 * @brief This API is used to get the profile duration of the sensor.
 *
 * @param[in] dev      : Structure instance of bme680_dev.
 * @param[in] duration : Duration of the measurement in ms.
 *
 * @return Nothing
 */
void bme680_get_profile_dur(uint16_t *duration, const struct bme680_dev *dev);

/*!
 * @brief This API reads the pressure, temperature and humidity and gas data
 * from the sensor, compensates the data and store it in the bme680_data
 * structure instance passed by the user.
 *
 * @param[out] data: Structure instance to hold the data.
 * @param[in] dev : Structure instance of bme680_dev.
 *
 * @return Result of API execution status
 * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
 */
int8_t bme680_get_sensor_data(struct bme680_field_data *data, struct bme680_dev *dev);

/*!
 * @brief This API is used to set the oversampling, filter and T,P,H, gas selection
 * settings in the sensor.
 *
 * @param[in] dev : Structure instance of bme680_dev.
 * @param[in] desired_settings : Variable used to select the settings which
 * are to be set in the sensor.
 *
 *   Macros                      |  Functionality
 *---------------------------------|----------------------------------------------
 *  BME680_OST_SEL               |    To set temperature oversampling.
 *  BME680_OSP_SEL               |    To set pressure oversampling.
 *  BME680_OSH_SEL               |    To set humidity oversampling.
 *  BME680_GAS_MEAS_SEL          |    To set gas measurement setting.
 *  BME680_FILTER_SEL            |    To set filter setting.
 *  BME680_HCNTRL_SEL            |    To set humidity control setting.
 *  BME680_RUN_GAS_SEL           |    To set run gas setting.
 *  BME680_NBCONV_SEL            |    To set NB conversion setting.
 *  BME680_GAS_SENSOR_SEL        |    To set all gas sensor related settings
 *
 * @note : Below are the macros to be used by the user for selecting the
 * desired settings. User can do OR operation of these macros for configuring
 * multiple settings.
 *
 * @return Result of API execution status
 * @retval zero -> Success / +ve value -> Warning / -ve value -> Error.
 */
int8_t bme680_set_sensor_settings(uint16_t desired_settings, struct bme680_dev *dev);

/*!
 * @brief This API is used to get the oversampling, filter and T,P,H, gas selection
 * settings in the sensor.
 *
 * @param[in] dev : Structure instance of bme680_dev.
 * @param[in] desired_settings : Variable used to select the settings which
 * are to be get from the sensor.
 *
 * @return Result of API execution status
 * @retval zero -> Success / +ve value -> Warning / -ve value -> Error.
 */
int8_t bme680_get_sensor_settings(uint16_t desired_settings, struct bme680_dev *dev);
#ifdef __cplusplus
}
```

```
#endif /* End of CPP guard */
#endif /* BME680_H_ */
/** @}*/
```

```
//***************************************************************************
//
// File Name            : "DOGM16W_A_SERCOM1.c"
// Title                : DOGM16W_A_SERCOM1.c
// Date                 : 4/8/2020
// Version              : 1.0
// Target MCU           : SAML21J18B
// Target Hardware      ; DOG LCD
// Author               : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// Contains function and variable definitions used to initialize and perform
// SPI communication between the MCU and DOG LCD.
//
// Warnings             :
// Restrictions         : none
// Algorithms           : none
// References           :
//
// Revision History     : Initial version
//
//
//**************************************************************************

#include <stdio.h>
#include "saml21j18b.h"

#define freq 2000000

extern unsigned char* ARRAY_PORT_PINCFG0 = (unsigned char*)&REG_PORT_PINCFG0;
extern unsigned char* ARRAY_PORT_PMUX0 = (unsigned char*)&REG_PORT_PMUX0;

extern char dsp_buff_1[17], dsp_buff_2[17], dsp_buff_3[17];

extern void delay_30us (void) {
    int clock_factor = freq/1000000;                //delay loop from i=0 to i<6 is 30us long at 1 MHz.
    for (int i = 0; i < clock_factor+3; i++) {      //loop end value is adjusted based on frequency macro constant.
        __asm("nop");
    }
}

extern void v_delay (signed char inner, signed char outer) {
    delay_30us();
    inner--;
    if (inner != 0) {
        v_delay(inner, outer);
    }
    outer--;
    if (outer != 0 ) {
        v_delay(0,outer);
    }
}

extern void delay_40ms (void) {
    /*
    v_delay(0,4);       //call v_delay function with inner loop variable=0 and outer loop variable=4
    */
    for (int j=0; j<= 1240 ; j++) {
        delay_30us();
    }
}

//****************************************************************************
//
// Function Name        : "init_spi_lcd"
// Date                 :
// Version              : 1.0
// Target MCU           : SAML21J18B
// Target Hardware      ; DOG LCD
// Author               : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// Initializes the MCU's SPI communication to the DOG LCD at SERCOM1.
// PA16 = MOSI, PA17 = SCK, PA18 = /SS, PA19 = MISO (not used), PB06 = RS.
//
// Warnings             : none
// Restrictions          : none
// Algorithms            : none
// References            : none
//
// Revision History      : Initial version
//
//***************************************************************************
extern void init_spi_lcd (void) {
    REG_GCLK_PCHCTRL19 = 0x00000040;    /* SERCOM1 core clock not enabled by default */

    ARRAY_PORT_PINCFG0[16] |= 1;    /* allow pmux to set PA16 pin configuration */
    ARRAY_PORT_PINCFG0[17] |= 1;    /* allow pmux to set PA17 pin configuration */
    ARRAY_PORT_PINCFG0[18] |= 1;    /* allow pmux to set PA18 pin configuration */
    ARRAY_PORT_PINCFG0[19] |= 1;    /* allow pmux to set PA19 pin configuration */
    ARRAY_PORT_PMUX0[8] = 0x22;     /* PA16 = MOSI, PA17 = SCK */
    ARRAY_PORT_PMUX0[9] = 0x22;     /* PA18 = SS,   PA19 = MISO */
    REG_PORT_DIRSET1 = 0x40;        /* RS output */

    REG_SERCOM1_SPI_CTRLA = 1;              /* reset SERCOM1 */
    while (REG_SERCOM1_SPI_CTRLA & 1) {}    /* wait for reset to complete */

    REG_SERCOM1_SPI_CTRLA = 0x3030000C;     /* MISO-3, MOSI-0, SCK-1, SS-2, CPOL=1, CPHA=1 */
    REG_SERCOM1_SPI_CTRLB = 0x00002000;     /* Master SS, 8-bit */
    REG_SERCOM1_SPI_BAUD = 0;               /* SPI clock is 4MHz/2 = 2MzHz */
    REG_SERCOM1_SPI_CTRLA |= 2;             /* enable SERCOM1 */
}

//****************************************************************************
//
// Function Name         : "lcd_spi_transmit_CMD"
// Date                  :
// Version               : 1.0
// Target MCU            : SAML21J18B
// Target Hardware       ; DOG LCD
// Author                : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// Sends a single 8-bit value to the DOG LCD. Value seen as a command
// by the LCD.
//
// Warnings              : none
// Restrictions          : none
// Algorithms            : none
// References            : none
//
// Revision History      : Initial version
//
//**************************************************************************
extern void lcd_spi_transmit_CMD (char CMD) {
    REG_PORT_OUTCLR1 = 0x00040040;                 // RS = 0 -> command, /SS = 0 -> selected
    while(!(REG_SERCOM1_SPI_INTFLAG & 1)) {}       /* wait until Tx ready */
    REG_SERCOM1_SPI_DATA = CMD;                    /* send data byte */
    REG_PORT_OUTSET1 = 0x00040000;                 // /SS = 1 -> deselected
}

//***************************************************************************
//
// Function Name         : "lcd_spi_transmit_DATA"
// Date                  :
// Version               : 1.0
// Target MCU            : SAML21J18B
// Target Hardware       ; DOG LCD
// Author                : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// Sends a single 8-bit value to the DOG LCD. Value seen as data to be
// displayed by the LCD.
//
// Warnings              : none
// Restrictions          : none
// Algorithms            : none
// References            : none
//
// Revision History      : Initial version
//
//**************************************************************************
```

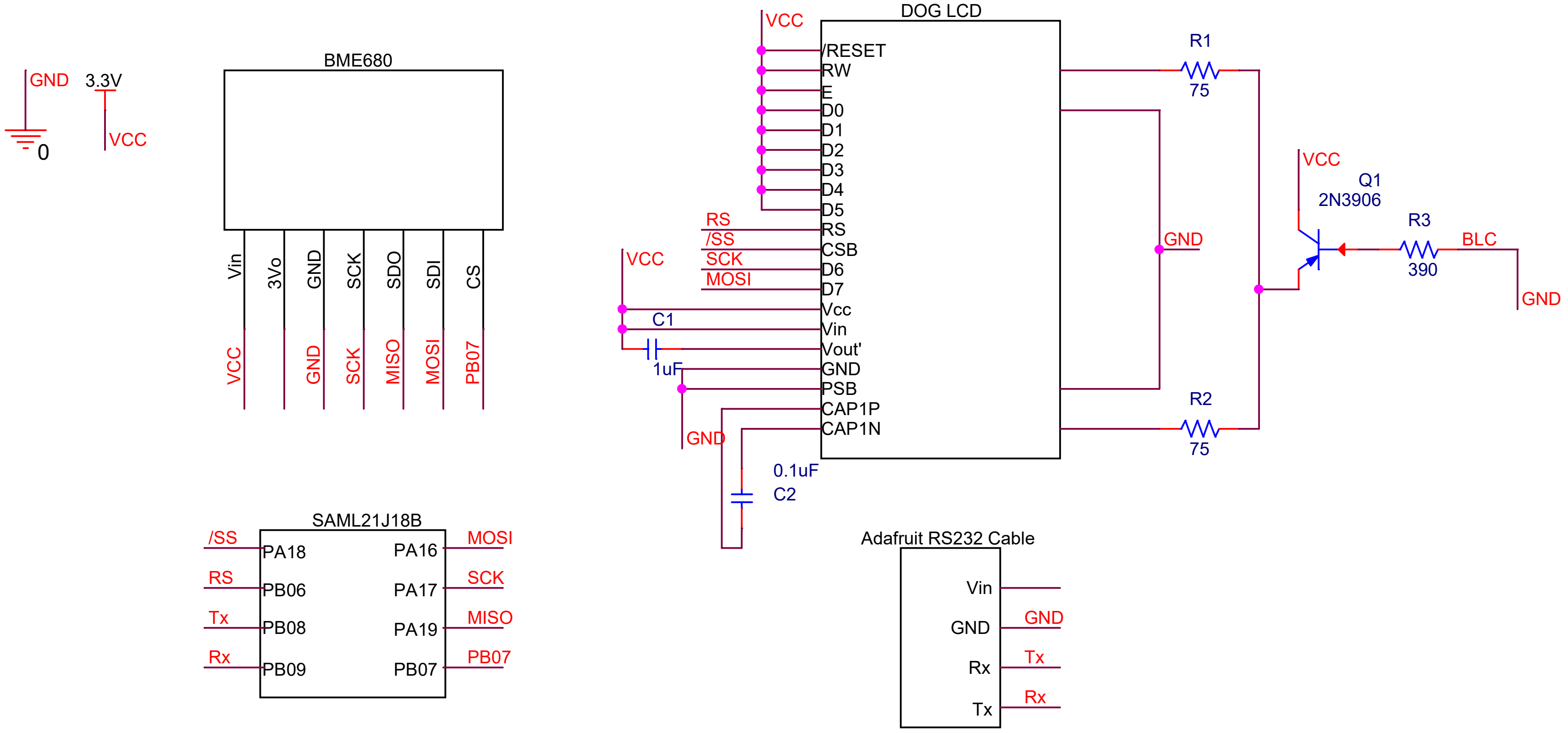
```c
extern void lcd_spi_transmit_DATA (char data) {
    REG_PORT_OUTSET1 = 0x00000040;                  // RS = 1 -> data
    REG_PORT_OUTCLR1 = 0x00040000;                  // /SS = 0 -> selected
    while(!(REG_SERCOM1_SPI_INTFLAG & 1)) {}        /* wait until Tx ready */
    REG_SERCOM1_SPI_DATA = data;                    /* send data byte */
    REG_PORT_OUTSET1 = 0x00040000;                  // /SS = 1 -> deselected
}

//****************************************************************************
//
// Function Name        : "init_lcd_dog"
// Date                 :
// Version              : 1.0
// Target MCU           : SAML21J18B
// Target Hardware      ; DOG LCD
// Author               : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// Configures and initializes DOG LCD and MCU for SPI communication.
//
// Warnings             : none
// Restrictions         : none
// Algorithms           : none
// References           : none
//
// Revision History     : Initial version
//
//***************************************************************************
extern void init_lcd_dog (void) {
    init_spi_lcd();         // initialize SPI ports for LCD DOG
    delay_40ms();

    //function set 1
    lcd_spi_transmit_CMD(0x39);     //send function set 1
    delay_30us();

    //function set 2
    lcd_spi_transmit_CMD(0x39);     //send function set 2
    delay_30us();

    //set bias value
    lcd_spi_transmit_CMD(0x1E);     //set bias value
    delay_30us();

    //power control
    lcd_spi_transmit_CMD(0x55);     //configure for 3.3 V
    delay_30us();

    //follower control
    lcd_spi_transmit_CMD(0x6C);     //follower mode on
    delay_40ms();

    //contrast set
    lcd_spi_transmit_CMD(0x7F);     //configure for 3.3 V
    delay_30us();

    //display on
    lcd_spi_transmit_CMD(0x0C);     //display on, cursor off, blink off
    delay_30us();

    //clear display
    lcd_spi_transmit_CMD(0x01);     //clear display, cursor home
    delay_30us();

    //entry mode
    lcd_spi_transmit_CMD(0x06);     //clear display, cursor home
    delay_30us();
}

//***************************************************************************
//
// Function Name        : "update_lcd_dog"
// Date                 :
// Version              : 1.0
// Target MCU           : SAML21J18B
// Target Hardware      ; DOG LCD
// Author               : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// Updates the LCD display with the current values in dsp_buff_1, dsp_buff_2
// and dsp_buff_3.
//
// Warnings             : none
// Restrictions         : none
// Algorithms           : none
// References           : none
//
// Revision History     : Initial version
//
//**************************************************************************
extern void update_lcd_dog (void) {
    char i;                         //initialize loop variable
    init_spi_lcd();                 //initialize SPI ports

    //send line 1 to LCD
    lcd_spi_transmit_CMD(0x80);     //initialize DDRAM addr-ctr
    delay_30us();

    for (i=0; i<=15; i++) {
        lcd_spi_transmit_DATA(dsp_buff_1[i]);       //send buff 1
        delay_30us();
    }

    //send line 2 to LCD
    lcd_spi_transmit_CMD(0x90);     //initialize DDRAM addr-ctr
    delay_30us();

    for (i=0; i<=15; i++) {
        lcd_spi_transmit_DATA(dsp_buff_2[i]);       //send buff 2
        delay_30us();
    }

    //send line 3 to LCD
    lcd_spi_transmit_CMD(0xA0);     //initialize DDRAM addr-ctr
    delay_30us();

    for (i=0; i<=15; i++) {
        lcd_spi_transmit_DATA(dsp_buff_3[i]);       //send buff 3
        delay_30us();
    }
}

//***************************************************************************
//
// Function Name        : "clr_dsp_buff"
// Date                 :
// Version              : 1.0
// Target MCU           : SAML21J18B
// Target Hardware      ; DOG LCD
// Author               : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// Clears LCD by filling display buffers with all spaces and calling the
// update_lcd_dog function
//
// Warnings             : none
// Restrictions         : none
// Algorithms           : none
// References           : none
//
// Revision History     : Initial version
//
//**************************************************************************
extern void clr_dsp_buff(void) {
    sprintf(dsp_buff_1, "                ");    //loads display buffers with all spaces
    sprintf(dsp_buff_2, "                ");
    sprintf(dsp_buff_3, "                ");
    update_lcd_dog();
}
```

```
//***************************************************************************
//
// File Name            : "DOGM163W_A_SERCOM1.h"
// Title                : DOGM163W_A_SERCOM1.h
// Date                 : 4/8/2020
// Version              : 1.0
// Target MCU           : SAML21J18B
// Target Hardware      ; DOG LCD
// Author               : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// Include file containing function prototypes and variable declarations
// which are defined in DOGM163W_A_SERCOM1.c.
//
// Warnings             :
// Restrictions         : none
// Algorithms           : none
// References           :
//
// Revision History     : Initial version
//
//
//**************************************************************************


#ifndef DOGM163W_A_SERCOM1_H_
#define DOGM163W_A_SERCOM1_H_

#include <stdio.h>
#include "saml21j18b.h"

//variable declarations and function prototypes for C file
unsigned char* ARRAY_PORT_PINCFG0;
unsigned char* ARRAY_PORT_PMUX0;

char dsp_buff_1[17], dsp_buff_2[17], dsp_buff_3[17];

void delay_30us (void);
void v_delay (signed char inner, signed char outer);
void delay_40us (void);
void init_spi_lcd (void);
void lcd_spi_transmit_CMD (char CMD);
void lcd_spi_transmit_DATA (char data);
void init_lcd_dog (void);
void update_lcd_dog (void);
void clr_dsp_buff (void);

#endif /* DOGM163W_A_SERCOM1_H_ */
```

```c
//***************************************************************************
//
// File Name            : "RS232_SERCOM4.c"
// Title                : RS232_SERCOM4.c
// Date                 : 4/8/2020
// Version              : 1.0
// Target MCU           : SAML21J18B
// Target Hardware      ; none
// Author               : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// Contains function and variable definitions to setup RS232 communication
// between the MCU and a terminal.
//
// Warnings             :
// Restrictions         : none
// Algorithms           : none
// References           :
//
// Revision History     : Initial version
//
//
//**************************************************************************

#include "saml21j18b.h"

extern unsigned char* ARRAY_PORT_PINCFG1 = (unsigned char*)&REG_PORT_PINCFG1;
extern unsigned char* ARRAY_PORT_PMUX1 = (unsigned char*)&REG_PORT_PMUX1;

//***************************************************************************
//
// Function Name        : "UART4_init"
// Date                 : 4/8/2020
// Version              : 1.0
// Target MCU           : SAML21J18B
// Target Hardware      : none
// Author               : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// Initializes the MCU's RS232 communication at SERCOM4.
// 9600 baud, LSB first, 8 bits, no parity bit, 1 stop bit
// PB09 = Rx, PB08 = Tx
//
// Warnings               : none
// Restrictions           : none
// Algorithms             : none
// References             : none
//
// Revision History       : Initial version
//
//**************************************************************************
/* initialize UART4 to transmit at 9600 Baud */
extern void UART4_init(void) {
    REG_GCLK_PCHCTRL22 = 0x00000040;
    REG_SERCOM4_USART_CTRLA |= 1; /* reset SERCOM4 */
    while (REG_SERCOM4_USART_SYNCBUSY & 1) {} /* wait for reset to complete */
    REG_SERCOM4_USART_CTRLA = 0x40106004; /* LSB first, async, no parity,
    PAD[1]-Rx, PAD[0]-Tx, BAUD uses fraction, 8x oversampling, internal clock */
    REG_SERCOM4_USART_CTRLB = 0x00030000; /* enable Tx, Rx, one stop bit, 8 bit */
    REG_SERCOM4_USART_BAUD = 52; /* 1000000 / 8 / 9600 = 13.02 */
    REG_SERCOM4_USART_CTRLA |= 2; /* enable SERCOM4 */
    while (REG_SERCOM4_USART_SYNCBUSY & 2) {} /* wait for enable to complete */
    ARRAY_PORT_PINCFG1[8] |= 1; /* allow pmux to set PB08 pin configuration */
    ARRAY_PORT_PINCFG1[9] |= 1; /* allow pmux to set PB09 pin configuration */
    ARRAY_PORT_PMUX1[4] = 0x33; /* PB08 = TxD, PB09 = RxD */
}

//***************************************************************************
//
// Function Name          : "UART4_write"
// Date                   : 4/8/2020
// Version                : 1.0
// Target MCU             : SAML21J18B
// Target Hardware        : none
// Author                 : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// Writes a single 8-bit value to data register of SERCOM4.
//
// Warnings             : none
// Restrictions         : none
// Algorithms           : none
// References           : none
//
// Revision History     : Initial version
//
//**************************************************************************
extern void UART4_write(char data) {
    while(!(REG_SERCOM4_USART_INTFLAG & 1)) {} /* wait for data register empty */
    REG_SERCOM4_USART_DATA = data; /* send a char */
}

//***************************************************************************
//
// Function Name        : "UART4_read"
// Date                 : 4/8/2020
// Version              : 1.0
// Target MCU           : SAML21J18B
// Target Hardware      : none
// Author               : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// Returns a single 8-bit value read from SERCOM4's data register.
//
// Warnings             : none
// Restrictions         : none
// Algorithms           : none
// References           : none
//
// Revision History     : Initial version
//
//**************************************************************************
extern char UART4_read(void) {
    while(!(REG_SERCOM4_USART_INTFLAG & 4)) {} /* wait until receive complete */
    return REG_SERCOM4_USART_DATA; /* read the receive char and return it */
}
```

```
//***************************************************************************
//
// File Name            : "RS232_SERCOM4.h"
// Title                : RS232_SERCOM4.h
// Date                 : 4/8/2020
// Version              : 1.0
// Target MCU           : SAML21J18B
// Target Hardware      ; none
// Author               : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// Include file containing function prototypes and variable declarations
// used to initialize the MCU's RS232 communication.
// Functions and variables defined in RS232_SERCOM4.c.
//
// Warnings             :
// Restrictions         : none
// Algorithms           : none
// References           :
//
// Revision History     : Initial version
//
//
//**************************************************************************


#ifndef RS232_SERCOM4_H_
#define RS232_SERCOM4_H_

//function and variable declarations
void UART4_init(void);
void UART4_write(char data);
char UART4_read(void);
unsigned char* ARRAY_PORT_PINCFG1;
unsigned char* ARRAY_PORT_PMUX1;

#endif /* RS232_SERCOM4_H_ */
```

```
 //***************************************************************************
//
// File Name            : "system.h"
// Title                : system.h
// Date                 : 4/8/2020
// Version              : 1.0
// Target MCU           : SAML21J18B
// Target Hardware      : DOG LCD
// Author               : Brandon Cheung, Ishabul Haque
// DESCRIPTION
// Mimics several operating system services to allow for formatted printing
// on the DOG LCD.
//
// Warnings             :
// Restrictions         : none
// Algorithms           : none
// References           :
//
// Revision History     : Initial version
//
//
//***************************************************************************


#ifndef SYSTEM_H_
#define SYSTEM_H_

#include "saml21j18b.h"

int n = 0;
char str[80];

int _write(FILE *f, char *buf, int n) {
    int m=n;
    for (; n>0; n--) {
        UART4_write(*buf++);
    }
    return m;
}

int _read(FILE *f, char *buf, int n) {
    *buf = UART4_read();
    if (*buf == '\r') {
        *buf = '\n';
        _write(f, "\r", 1);
    }
    _write(f, buf, 1);
    return 1;
}

int _close(FILE *f) {
    return 0;
}

int _fstat(FILE *f, void *p) {
    *((int*)p + 4) = 0x81B6;    //enable read/write
    return 0;
}

int _isatty(FILE *f) {
    return 1;
}

int _lseek(FILE *f, int o, int w) {
    return 0;
}

void* _sbrk(int i) {
    return (void*)0x20006000;
}

#endif /* SYSTEM_H_ */
```



Title	ESE 381 Lab 10 Schematic	Brandon Cheung, Ishabul Haque	
Size A	Document Number Lab 10 Schematic Page 1		Rev 1A
Date:	Monday, May 11, 2020	Sheet 1 of 1	