Brangaitis & Varghese
110864246 & 111604890
Lab #10
L-01
Bench #15

Anthony Brangaitis's video
https://drive.google.com/file/d/1myNCaHVZr83AiA5-XouLT39QFSG-M-ey/view?usp=sharing

Aaron Varghese's video
https://drive.google.com/open?id=1wb1HJiRYqarG2U-3ZWCOIzV9stO3Lr0C

```c
/*
 * temp_raw.c
 *
 * Created: 5/1/2020 11:04:22 AM
 * Author : Aaron
 This program basically does the same
 task as Lab9 Task2, but we will be using
 the way that the functions are defined in
 the Bosch API. This will later on allow us
 to be able to calculate the temperature,
 pressure,humidity and gas measurements from
 the BME680 sensor.
 data[0] = status register
 data[1] = id
 data[2] = measure_flag
 data[3] = temp_msb
 data[4] = temp_lsb
 data[5] = temp_least
*/


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


unsigned char * ARRAY_PINCFG0 = (unsigned char*) & REG_PORT_PINCFG0;
unsigned char * ARRAY_PMUX0 = (unsigned char*) & REG_PORT_PMUX0;
static void init_spi_bme680(void);
static void user_delay_ms(uint32_t period);

static uint8_t spi_transfer(uint8_t data);
static int8_t user_spi_read(uint8_t dev_id, uint8_t reg_addr, uint8_t* reg_data, uint16_t len);
static int8_t user_spi_write(uint8_t dev_id, uint8_t reg_addr, uint8_t* reg_data, uint16_t len);

uint8_t status,id;
```

```
36  uint32_t temperature_raw;
37
38
39  int main(void)
40  {
41      init_spi_bme680();
42
43      uint8_t commands[] = {0x00,0xB6,0x10};
44
45      uint8_t data[9] = {0,0,0,0,0,0,0,0,0};
46
47      //software reset BME680
48      user_spi_write(0,0x60,&commands[1],1);
49
50      //reading the BME680 status register
51      user_spi_read(0,0x73,&data[0],1);
52      status = data[0];
53
54      //read the id register
55      user_spi_read(0,0x50,&data[1],1);
56      id = data[1];
57
58      //switch to page 1 of the memory map
59      user_spi_write(0,0x73,&commands[2],1);
60      user_spi_read(0,0x73,&data[0],1);
61      status =  data[0];
62      uint8_t* measure_flag = &data[2];
63      uint8_t* temp_msb = &data[3];
64      uint8_t* temp_lsb = &data[4];
65      uint8_t* temp_least = &data[5];
66      commands[0] = 0;
67      commands[1] = 0x20;
68      commands[2] = 0x21;
69
70      while (1)
```

```
71      {
72          //enable the BME680 to read only temperature
73          //humidity skipped
74          user_spi_write(0,0x72,&commands[0],1);
75          //temp set to 1x oversampling, pressure skipped
76          user_spi_write(0,0x74,&commands[1],1);
77
78          //setting up gas to be disabled
79          user_spi_write(0,0x71,&commands[0],1);
80
81          //start the conversion using forced mode
82          user_spi_write(0,0x74,&commands[2],1);
83
84
85
86          //waiting for the measurement to finish
87          while(!(*measure_flag&0x80))
88          {
89          //checking the measure flag
90          user_spi_read(0,0x1D,&data[2],1);
91          }
92
93          user_spi_read(0,0x22,temp_msb,3);
94
95
96          temperature_raw = (*temp_msb<<12)|(*temp_lsb<<4)|(*temp_least>>4);
97      }
98  }
99
100 /************************
101 NAME:       init_spi_bme680
102 ASSUMES:    The BME680 sensor is interfaced by SPI through SERCOM1
103 SERCOM1 is being used for the SPI data transfer
104 PA16 (PAD[0])---> SDI
105 PA17 (PAD[1])---> SCK
```

```
106  PA19 (PAD[3])--->SD0
107  PB07 ---->/CSB
108
109  RETURNS:    N/A
110  MODIFIES:   N/A
111  CALLED BY:
112  DESCRITION: init SPI port for communication with the BME680
113  ****************************************************************/
114  static void init_spi_bme680(void)
115  {
116      //this initializes the SERCOM SPI unit
117      //REG_MCLK_AHBMASK |= 0x04; //APBC bus enabled by default
118      //REG_MCLK_APBCMASK|= 0x02; //SERCOM1 APBC bus clock enabled
119      //by default
120      //using generic clock generator 0 (4 MHz) for peripheral clock
121      REG_GCLK_PCHCTRL19 = 0x40;// enabling SERCOM1 core clock
122
123      ARRAY_PINCFG0[16] |= 1; //setting PMUX config
124      ARRAY_PINCFG0[17] |= 1; //setting PMUX config
125      ARRAY_PINCFG0[19] |= 1; //setting PMUX config
126      ARRAY_PMUX0[8] = 0x22; //PA16 = SDO, PA17 = SCK
127      ARRAY_PMUX0[9] = 0x20; //PA19 = SDI
128
129      REG_SERCOM1_SPI_CTRLA = 1; //software reset SERCOM1
130      while(REG_SERCOM1_SPI_CTRLA & 1){};//waiting for reset to finish
131      //SDI = PAD[0], SDO = PAD[3], SCK = PAD[1], using PB07 for /CSB
132      REG_SERCOM1_SPI_CTRLA = 0x3030000C;//CPOL and CPHA are 11
133      REG_SERCOM1_SPI_CTRLB = 0x020000; //8-bit data
134      //SPI clock is going to be written to 2MHz
135      REG_SERCOM1_SPI_BAUD = 0;
136      REG_SERCOM1_SPI_CTRLA|=2; //SERCOM1 enabled
137
138      //setting up /CSB, and turning it off
139      REG_PORT_DIR1 |=128;
140      REG_PORT_OUT1 |=128;
```

```
141 }
142
143 /************************
144 NAME:       user_delay_ms()
145 ASSUMES:    An integer value is given that specifies amount of ms
146 delay that is needed
147
148 RETURNS:    N/A
149 MODIFIES:   N/A
150 CALLED BY:  N/A
151 DESCRITION: delay that lasts (uint32_t) period ms long
152 ************************************************************/
153 static void user_delay_ms(uint32_t period)
154 {
155     for(;period>0;period--)
156     {
157         for(int i=0;i<199;i++)
158         {
159             __asm("nop");
160         }
161     }
162 }
163
164
165 /************************
166 NAME:       spi_transfer()
167 ASSUMES:    1) The BME680 sensor is interfaced by SPI through SERCOM1
168 SERCOM1 is being used for the SPI data transfer.
169 2) Also assumes that data is given as a parameter to transfer
170
171
172 RETURNS:    N/A
173 MODIFIES:   N/A
174 CALLED BY: spi_read_BME680, spi_write_BME680
175 DESCRITION: transfers data between the sensor and SERCOM1
```

```
176 ****************************************************************/
177 static uint8_t spi_transfer(uint8_t data)
178 {
179     while(!(REG_SERCOM1_SPI_INTFLAG&0x01)){}
180     REG_SERCOM1_SPI_DATA = data;
181     while(!(REG_SERCOM1_SPI_INTFLAG&0x04)){}
182     return REG_SERCOM1_SPI_DATA;
183 }
184
185
186 /************************
187 NAME:       user_spi_read()
188 ASSUMES:
189 1) Address of register to read from is given
190 2) Chip ID is given (no need to do anything for it in our case)
191 3) Pointer to data is given
192 4) Amount of bytes needed to be written sequentially is given
193
194 RETURNS:    data in the address
195 MODIFIES:   N/A
196 CALLED BY:  N/A
197 DESCRITION: Returns a result that says if the read function was
198 done successfully
199 ****************************************************************/
200 static int8_t user_spi_read(uint8_t dev_id, uint8_t reg_addr, uint8_t* reg_data, uint16_t len)
201 {
202     //operation is done successfully
203     int8_t rslt = 0, j= 0;
204
205     //enabling the BME680
206     REG_PORT_OUTCLR1 = 128;
207
208     //control byte for the transfer
209     spi_transfer(reg_addr|(1<<7));
210
```

```
211     //this will read through each address and increment automatically
212     for(uint8_t i = 0;i<len;i++)
213     {
214         *reg_data = spi_transfer(0x00);
215         reg_data++;
216     }
217     //disabling the BME680
218     REG_PORT_OUTSET1 = 128;
219     return rslt;
220 }
221 
222 /***********************
223 NAME:       user_spi_write()
224 ASSUMES:
225 1) Address of register to write to is given
226 2) Chip ID is given (no need to do anything for it in our case)
227 3) Pointer to data is given
228 4) Amount of bytes needed to be written sequentially is given
229 
230 RETURNS:    N/A
231 MODIFIES:   N/A
232 CALLED BY:  N/A
233 DESCRITION: writes specified data to a specific register
234 ************************************************************/
235 static int8_t user_spi_write(uint8_t dev_id, uint8_t reg_addr, uint8_t* reg_data, uint16_t len)
236 {
237     //operation is done successfully
238     int8_t rslt =0;
239 
240     //enabling the BME680
241     REG_PORT_OUTCLR1 = 128;
242             for(uint8_t i =reg_addr;i<reg_addr+len;i++)
243             {
244                 spi_transfer(i);
245                 spi_transfer(*reg_data);
```

```
246                     reg_data++;
247                 }
248         //disabling the BME680
249         REG_PORT_OUTSET1 = 128;
250         return rslt;
251     }
```

```
1  #ifndef LCD_DOG_DRIVER_H_
2  #define LCD_DOG_DRIVER_H_
3
4  char dsp_buff_1[17], dsp_buff_2[17],dsp_buff_3[17];
5
6  void delay_30us(void);
7
8  void v_delay(int a,int b);
9
10 void delay_40mS(void);
11
12 void init_spi_lcd(void);
13
14 void lcd_spi_transmit_CMD(char command);
15
16 void lcd_spi_transmit_DATA(char data);
17
18 void init_lcd_dog(void);
19
20 void update_lcd_dog(void);
21
22
23 #endif
24
25
26
27
28
```

```
/*********************************************************************

   * File Name: lcd_dog_driver.c
   *
   * Date: 3/26/2020
   * Author : Aaron Varghese
   * Version 1.0
   * Target: ATSAML21J18B
   * Target Hardware: SAML21 XPlained PRO

   This driver contains procedures to initialize and update
   DOG text based LCD display modules, including the EA DOG163W-A LCD
   modules configured with three (3) 16 characters display lines.

   The display module hardware interface uses a 1-direction, write only
   SPI interface. (See below for more information.)

   The display module software interface uses three (3) 16-byte
   data (RAM) based display buffers - One for each line of the display.
   (See below for more information.)
*/

/*look at page 43 of the EA DOG163W-A LCD data sheet for details on the
schematic for SPI data transfer and the initialization process of the
LCD screen

SERCOM1 is being used for the SPI data transfer
PA16 (PAD[0])---> MOSI
PA17 (PAD[1])---> SCK
PA18 (PAD[2])---> /SS
PA19 (PAD[3])--->MISO //NOT USED, since the LCD is write only

PB06 --->/RS
*/

```

```
36 #include "saml21j18b.h"
37 //for each of the lines in the LCD screen
38 unsigned char* ARRAY_PINCFG0 =(unsigned char*) &REG_PORT_PINCFG0;
39 unsigned char* ARRAY_PMUX0 = (unsigned char*) &REG_PORT_PMUX0;
40 char dsp_buff_1[17];
41 char dsp_buff_2[17];
42 char dsp_buff_3[17];
43
44 /*************************
45 NAME:        delay_30uS
46 ASSUMES:     nothing
47 CALLED BY:   init_dsp
48 DESCRIPTION: This procedure will generate a fixed delay of just over
49              30 uS (assuming a 4 MHz clock).
50 ************************************************************************/
51 void delay_30us(void)
52 {
53     for(int i= 40;i>0;i--){ __asm("nop");   }
54 };
55
56 /*********************
57 NAME:        v_delay
58 ASSUMES:     Integers a and b= initial count values defining how many
59              30uS delays will be called. This procedure can generate
60              short delays (a = small #) or much longer delays (where
61              b value is large).
62 RETURNS:     nothing
63 CALLED BY:   init_dsp, plus...
64 DESCRIPTION: This procedure will generate a variable delay for a fixed
65              period of time based the values pasted in a and b.
66
67 a is the inner loop value, and b is the outer loop value
68
69 ************************************************************************/
70 void v_delay(int a,int b)
```

```
71  {
72      for(;a>0;a--)
73      {
74          for(;b>0;b--){}
75      }
76  };
77
78  /************************
79  NAME:        delay_40mS
80  ASSUMES:     nothing
81  RETURNS:     nothing
82  MODIFIES:    N/A
83  CALLED BY:   init_dsp
84  DESCRIPTION: This procedure will generate a fixed delay of
85               40 mS.
86  ************************************************************************/
87  void delay_40mS(void)
88  {
89      v_delay(700,15000);
90  };
91
92  /************************
93  NAME:       init_spi_lcd
94  ASSUMES:    The LCD module is interfaced by SPI through SERCOM1
95  SERCOM1 is being used for the SPI data transfer
96  PA16 (PAD[0])---> MOSI
97  PA17 (PAD[1])---> SCK
98  PA18 (PAD[2])---> /SS
99  PA19 (PAD[3])--->MISO
100
101 PB06 ----> /RS
102
103 RETURNS:    N/A
104 MODIFIES:   N/A
105 CALLED BY:  init_dsp, update
```

```
106 DESCRITION: init SPI port for command and data writes to LCD via SPI
107 ***********************************************************************/
108 void init_spi_lcd(void)
109 {
110 //this initializes the SERCOM SPI unit
111 //REG_MCLK_AHBMASK |= 0x04; //APBC bus enabled by default
112 //REG_MCLK_APBCMASK|= 0x02; //SERCOM1 APBC bus clock enabled
113 //by default
114 //using generic clock generator 0 (4 MHz) for peripheral clock
115 REG_GCLK_PCHCTRL19 = 0x40;// enabling SERCOM1 core clock
116
117 ARRAY_PINCFG0[16] |= 1; //setting PMUX config
118 ARRAY_PINCFG0[17] |= 1; //setting PMUX config
119 ARRAY_PINCFG0[18] |= 1; //setting PMUX config
120 ARRAY_PINCFG0[19] |= 1; //setting PMUX config
121 ARRAY_PMUX0[8] = 0x22; //PA16 = MOSI, PA17 = SCK
122 ARRAY_PMUX0[9] = 0x22; //PA18 = SS, PA19 = MISO
123
124 REG_SERCOM1_SPI_CTRLA = 1; //software reset SERCOM1
125 while(REG_SERCOM1_SPI_CTRLA & 1){};//waiting for reset to finish
126 //MISO = PAD[3], MOSI = PAD[0], SCK = PAD[1], SS = PAD[2], SPI master
127 REG_SERCOM1_SPI_CTRLA = 0x3030000C;//CPOL and CPHA are 11
128 REG_SERCOM1_SPI_CTRLB = 0x2000; //master SS, 8-bit data
129 //SPI clock should be a maximum of 3.125 MHz
130 //SPI clock is going to be written to 2MHz
131 REG_SERCOM1_SPI_BAUD = 0;
132 REG_SERCOM1_SPI_CTRLA|=2; //SERCOM1 enabled
133
134 //setting up /RS, and turning it off
135 REG_PORT_DIR1 |=64;
136 REG_PORT_OUT1 |=64;
137
138 };
139
140 /*******************************
```

```
141 NAME:       lcd_spi_transmit_CMD
142 ASSUMES:    command = byte for LCD.
143             SPI port are already configured.
144 RETURNS:    N/A
145 MODIFIES:   N/A
146 CALLED BY:  init_dsp, update
147 DESCRIPTION: outputs a byte passed in r16 via SPI port. Waits for data
148             to be written by SPI port before continuing.
149 ********************************************************************/
150 void lcd_spi_transmit_CMD(char command)
151 {
152     REG_PORT_OUTCLR1 |=64;//clearing /RS --> command
153     while(!(REG_SERCOM1_SPI_INTFLAG&1)) {}//wait until transmission is done
154     REG_SERCOM1_SPI_DATA = command;
155     while(!(REG_SERCOM1_SPI_INTFLAG&1)) {}//wait until transmission is done
156
157 };
158
159 /*******************************
160 NAME:       lcd_spi_transmit_DATA
161 ASSUMES:    data = byte to transmit to LCD.
162             SPI port is configured.
163 RETURNS:    N/A
164 MODIFIES:   N/A
165 CALLED BY:  init_dsp, update
166 DESCRIPTION: outputs a byte passed in r16 via SPI port. Waits for
167             data to be written by SPI port before continuing.
168 ********************************************************************/
169 void lcd_spi_transmit_DATA(char data)
170 {
171     REG_PORT_OUTSET1 |=64; //setting /RS --> data
172     while(!(REG_SERCOM1_SPI_INTFLAG&1)) {}//wait until transmission is done
173     REG_SERCOM1_SPI_DATA = data;
174     while(!(REG_SERCOM1_SPI_INTFLAG&1)) {}//wait until transmission is done
175 };
```

```
176
177 /************************
178 NAME:       init_lcd_dog
179 ASSUMES:    nothing
180 RETURNS:    nothing
181 MODIFIES:   R16, R17
182 CALLED BY:  main application
183 DESCRITION: inits DOG module LCD display for SPI (serial) operation.
184 NOTE:  Can be used as is with MCU clock speeds of 4MHz or less.
185 ************************************************************/
186 void init_lcd_dog(void)
187 {
188     //initialize the LCD DOG SPI protocol for the SAML21J18B
189     init_spi_lcd();
190
191     //delay of 40ms so VDD is stable
192     delay_40mS();
193
194     //function set 1
195     lcd_spi_transmit_CMD(0x39);
196     delay_30us();
197
198     //function set 2
199     lcd_spi_transmit_CMD(0x39);
200     delay_30us();
201
202     //setting the bias value/internal osc frequency
203     lcd_spi_transmit_CMD(0x1E);
204     delay_30us();
205
206     //contrast set
207     //~77 for 5V
208     //~7F for 3.3V
209     lcd_spi_transmit_CMD(0x7F);
210     delay_30us();
```

```
211
212     //POWER/ICON/CONTRAST control
213     //0x50 nominal for 5V
214     //0x55 for 3.3V
215     lcd_spi_transmit_CMD(0x55);
216     delay_30us();
217
218     //follower control
219     lcd_spi_transmit_CMD(0x6C);
220     //delay 200ms
221     delay_40mS();
222     delay_40mS();
223     delay_40mS();
224     delay_40mS();
225     delay_40mS();
226
227     //display ON/OFF control
228
229     lcd_spi_transmit_CMD(0x0C);//display on, cursor off, blink off
230     delay_30us();
231
232
233     lcd_spi_transmit_CMD(0x01);//clears display, cursor home
234     delay_30us();
235
236     //entry mode
237     lcd_spi_transmit_CMD(0x06);//clear display
238     delay_30us();
239 };
240
241 /*************************
242 NAME:       update_lcd_dog
243 ASSUMES:    display buffers loaded with display data
244 RETURNS:    N/A
245 MODIFIES:   N/A
```

```
246
247 DESCRIPTION: Updates the LCD display lines 1, 2, and 3, using the
248   contents of dsp_buff_1, dsp_buff_2, and dsp_buff_3, respectively.
249 ************************************************************/
250 void update_lcd_dog(void)
251 {
252     //initializing the SPI port for the LCD
253     init_spi_lcd();
254
255 //sending line 1 to the LCD module
256
257     //initialize the DDRAM address counter to the first line(00H-0FH)
258     lcd_spi_transmit_CMD(0x80);
259     delay_30us();
260     //putting in the data into dsp_buff_1
261     for(int i=0;i<16;i++)
262     {
263         lcd_spi_transmit_DATA(dsp_buff_1[i]);
264         delay_30us();
265     }
266
267 //sending line 2 to the LCD module
268
269 //initialize the DDRAM address counter to the second line(10H-1FH)
270 lcd_spi_transmit_CMD(0x90);
271 delay_30us();
272 //putting in the data from dsp_buff_2
273 for(int i=0;i<16;i++)
274 {
275     lcd_spi_transmit_DATA(dsp_buff_2[i]);
276     delay_30us();
277 }
278
279 //sending line 3 to the LCD module
280
```

```
281 //initialize the DDRAM address counter to the third line(20H-2FH)
282 lcd_spi_transmit_CMD(0xA0);
283 delay_30us();
284 //putting in the data into dsp_buff_1
285 for(int i=0;i<16;i++)
286 {
287     lcd_spi_transmit_DATA(dsp_buff_3[i]);
288     delay_30us();
289 }
290 };
291
292
```

```
1  #ifndef SERCOM4_RS232_H_
2  #define SERCOM4_RS232_H_
3  void UART_init(void);
4  void UART_write(char data);
5  void delayMs(int n);
6  char UART_read(void);
7  #endif
8
9
```

```c
/*

 * File Name: SERCOM4_echo_rs232.c
 *
 * Date: 3/26/2020
 * Author : Aaron Varghese
 * Version 1.0
 * Target: ATSAML21J18B
 * Target Hardware: SAML21 XPlained PRO

 Description:
 This program is to test out the RS232
 capabilities of the ATSAML21J18B.
 PB08------->Tx
 PB09------->Rx
 */


#include "saml21j18b.h"
unsigned char* ARRAY_PINCFG1 = (unsigned char*) &REG_PORT_PINCFG1;
unsigned char* ARRAY_PMUX1 = (unsigned char*) &REG_PORT_PMUX1;


/************************
NAME:       UART_init()
ASSUMES:    PB08 and PB09 are not being used
SERCOM4 is being used for the RS232 communication
 PB08------->Tx
 PB09------->Rx

RETURNS:    N/A
MODIFIES:   N/A
DESCRITION: initializes SERCOM4 for RS232 communication.
************************************************************/
void UART_init(void)
```

```
36  {
37      //Clock setup for the SAML21J18B
38      //REG_MCLK_AHBMASK |= 0x04;  //APBC bus enabled by default
39      //REG_MCLK_APBCMASK |= 0x10; //enabling the APBC bus clock
40      //for the SERCOM4
41
42      //Generic Clock Generator 0 will be used as the source of
43      //the SERCOM4 core clock
44      REG_GCLK_PCHCTRL22 |= 0x40;
45
46      //configure port pins Tx & Rx
47      //enable the PMUX
48      ARRAY_PINCFG1[8] |= 1;
49      ARRAY_PINCFG1[9] |= 1;
50      //PB08--->PAD[0] of SERCOM4
51      //PB09--->PAD[1] of SERCOM4
52      ARRAY_PMUX1[4] = 0x33;
53
54      //Configure the SERCOM4
55      REG_SERCOM4_USART_CTRLA |= 1;
56      //wait for reset to complete
57      while((REG_SERCOM4_USART_SYNCBUSY&1)){}
58      //LSB first, async, no parity, PAD[1]->Rx, PAD[0]->Tx, BAUD
59      //uses fraction, 8x oversampling, internal clock
60      REG_SERCOM4_USART_CTRLA |= 0x40106004;
61
62      //enable Tx, Rx, one stop bit, 8 bit
63      REG_SERCOM4_USART_CTRLB = 0x30000;
64
65      // (4MHz)/(8*9600) = 52.08
66      REG_SERCOM4_USART_BAUD = 52;
67      //enable the SERCOM4 peripheral
68      REG_SERCOM4_USART_CTRLA |= 2;
69
70      //waiting for enable to complete
```

```
71      while(REG_SERCOM4_USART_SYNCBUSY&2){}
72  }
73
74  /************************
75  NAME:       UART_write()
76  ASSUMES:    N/A
77  RETURNS:    N/A
78  MODIFIES:   SERCOM4's data register
79  DESCRITION: transfers a character out to the RS232 communication line
80  ************************************************************************/
81  void UART_write(char data)
82  {
83      //waiting for the the data register to be empty
84      while(!(REG_SERCOM4_USART_INTFLAG&1)){};
85      REG_SERCOM4_USART_DATA = data;
86  }
87
88  /************************
89  NAME:       UART_read()
90  ASSUMES:    N/A
91  RETURNS:    N/A
92  MODIFIES:   N/A
93  DESCRITION: reads in a character from the RS232 communication line
94  ************************************************************************/
95  char UART_read(void)
96  {
97      //waiting for transfer of incoming data to be fully sent
98      while(!(REG_SERCOM4_USART_INTFLAG&4)){};
99      //read the value of the data sent
100     return REG_SERCOM4_USART_DATA;
101 }
102
103 /************************
104 NAME:       delayMs()
105 ASSUMES:    an integer n is given in the parameter
```

```
106 RETURNS:    N/A
107 MODIFIES:   N/A
108 DESCRITION: sets an n Ms delay
109 ***************************************************************/
110 void delayMs(int n)
111 {
112     for(;n>0;n--)
113     {
114         for(int i=0;i<199;i++)
115         {
116             __asm("nop");
117         }
118     }
119 }
120
```

```h
#ifndef CONSOLE_IO_SUPPORT_H_
#define CONSOLE_IO_SUPPORT_H_

#include <stdio.h>
#include "SERCOM4_RS232.h"

int _write(FILE *f,char *buf, int n);

int _read(FILE *f, char *buf, int n);

int _close(FILE *f);

int _fstat(FILE *f, void *p);

int _isatty(FILE *f);

int _lseek(FILE *f, int o, int w);

void* _sbrk(int i);

#endif


```

```c
#include <stdio.h>
#include "SERCOM4_RS232.h"

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

int _read(FILE *f, char *buf, int n)
{
    *buf = UART_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;
}
```

```
36
37 int _isatty(FILE *f)
38 {
39     return 1;
40 }
41
42 int _lseek(FILE *f, int o, int w)
43 {
44     return 0;
45 }
46
47 void* _sbrk(int i)
48 {
49     return (void*) 0x20006000;
50 }
51
```

```
/**
 * Copyright (c) 2020 Bosch Sensortec GmbH. All rights reserved.
 *
 * BSD-3-Clause
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. 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.
 *
 * 3. Neither the name of the copyright holder nor the names of its
 *    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 THE
 * 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.
 *
 * @file    bme680.h
 * @date    23 Jan 2020
 * @version 3.5.10
```

```
 * @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
 */
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);

```

```
106 /*!
107  * @brief This API is used to set the power mode of the sensor.
108  *
109  * @param[in] dev : Structure instance of bme680_dev
110  * @note : Pass the value to bme680_dev.power_mode structure variable.
111  *
112  *  value    |   mode
113  * ----------|------------------
114  *  0x00     |   BME680_SLEEP_MODE
115  *  0x01     |   BME680_FORCED_MODE
116  *
117  * * @return Result of API execution status
118  * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
119  */
120 int8_t bme680_set_sensor_mode(struct bme680_dev *dev);
121
122 /*!
123  * @brief This API is used to get the power mode of the sensor.
124  *
125  * @param[in] dev : Structure instance of bme680_dev
126  * @note : bme680_dev.power_mode structure variable hold the power mode.
127  *
128  *  value    |   mode
129  * ---------|------------------
130  *  0x00     |   BME680_SLEEP_MODE
131  *  0x01     |   BME680_FORCED_MODE
132  *
133  * @return Result of API execution status
134  * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
135  */
136 int8_t bme680_get_sensor_mode(struct bme680_dev *dev);
137
138 /*!
139  * @brief This API is used to set the profile duration of the sensor.
140  *
```

```
141  * @param[in] dev      : Structure instance of bme680_dev.
142  * @param[in] duration : Duration of the measurement in ms.
143  *
144  * @return Nothing
145  */
146 void bme680_set_profile_dur(uint16_t duration, struct bme680_dev *dev);
147
148 /*!
149  * @brief This API is used to get the profile duration of the sensor.
150  *
151  * @param[in] dev      : Structure instance of bme680_dev.
152  * @param[in] duration : Duration of the measurement in ms.
153  *
154  * @return Nothing
155  */
156 void bme680_get_profile_dur(uint16_t *duration, const struct bme680_dev *dev);
157
158 /*!
159  * @brief This API reads the pressure, temperature and humidity and gas data
160  * from the sensor, compensates the data and store it in the bme680_data
161  * structure instance passed by the user.
162  *
163  * @param[out] data: Structure instance to hold the data.
164  * @param[in] dev : Structure instance of bme680_dev.
165  *
166  * @return Result of API execution status
167  * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
168  */
169 int8_t bme680_get_sensor_data(struct bme680_field_data *data, struct bme680_dev *dev);
170
171 /*!
172  * @brief This API is used to set the oversampling, filter and T,P,H, gas selection
173  * settings in the sensor.
174  *
175  * @param[in] dev : Structure instance of bme680_dev.
```

```
176  * @param[in] desired_settings : Variable used to select the settings which
177  * are to be set in the sensor.
178  *
179  *   Macros                    |  Functionality
180  *-------------------------------|-------------------------------------------
181  *  BME680_OST_SEL             |    To set temperature oversampling.
182  *  BME680_OSP_SEL             |    To set pressure oversampling.
183  *  BME680_OSH_SEL             |    To set humidity oversampling.
184  *  BME680_GAS_MEAS_SEL        |    To set gas measurement setting.
185  *  BME680_FILTER_SEL          |    To set filter setting.
186  *  BME680_HCNTRL_SEL          |    To set humidity control setting.
187  *  BME680_RUN_GAS_SEL         |    To set run gas setting.
188  *  BME680_NBCONV_SEL          |    To set NB conversion setting.
189  *  BME680_GAS_SENSOR_SEL      |    To set all gas sensor related settings
190  *
191  * @note : Below are the macros to be used by the user for selecting the
192  * desired settings. User can do OR operation of these macros for configuring
193  * multiple settings.
194  *
195  * @return Result of API execution status
196  * @retval zero -> Success / +ve value -> Warning / -ve value -> Error.
197  */
198 int8_t bme680_set_sensor_settings(uint16_t desired_settings, struct bme680_dev *dev);
199
200 /*!
201  * @brief This API is used to get the oversampling, filter and T,P,H, gas selection
202  * settings in the sensor.
203  *
204  * @param[in] dev : Structure instance of bme680_dev.
205  * @param[in] desired_settings : Variable used to select the settings which
206  * are to be get from the sensor.
207  *
208  * @return Result of API execution status
209  * @retval zero -> Success / +ve value -> Warning / -ve value -> Error.
210  */
```

```
211 int8_t bme680_get_sensor_settings(uint16_t desired_settings, struct bme680_dev *dev);
212 #ifdef __cplusplus
213 }
214 #endif /* End of CPP guard */
215 #endif /* BME680_H_ */
216 /** @}*/
217
```

```
/**
 * Copyright (c) 2020 Bosch Sensortec GmbH. All rights reserved.
 *
 * BSD-3-Clause
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. 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.
 *
 * 3. Neither the name of the copyright holder nor the names of its
 *    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 THE
 * 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.
 *
 * @file    bme680_defs.h
 * @date    23 Jan 2020
 * @version 3.5.10
```

```
 * @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)
```

```
71  #endif
72
73  #if !defined(INT32_C) && !defined(UINT32_C)
74  #define INT32_C(x)      S32_C(x)
75  #define UINT32_C(x)     U32_C(x)
76  #endif
77
78  #if !defined(INT64_C) && !defined(UINT64_C)
79  #define INT64_C(x)      S64_C(x)
80  #define UINT64_C(x)     U64_C(x)
81  #endif
82
83  /**@}*/
84
85  /**\name C standard macros */
86  #ifndef NULL
87  #ifdef __cplusplus
88  #define NULL   0
89  #else
90  #define NULL   ((void *) 0)
91  #endif
92  #endif
93
94  /** BME680 configuration macros */
95  /** Enable or un-comment the macro to provide floating point data output */
96  #ifndef BME680_FLOAT_POINT_COMPENSATION
97  /* #define BME680_FLOAT_POINT_COMPENSATION */
98  #endif
99
100 /** BME680 General config */
101 #define BME680_POLL_PERIOD_MS       UINT8_C(10)
102
103 /** BME680 I2C addresses */
104 #define BME680_I2C_ADDR_PRIMARY     UINT8_C(0x76)
105 #define BME680_I2C_ADDR_SECONDARY   UINT8_C(0x77)
```

```
106
107 /** BME680 unique chip identifier */
108 #define BME680_CHIP_ID  UINT8_C(0x61)
109
110 /** BME680 coefficients related defines */
111 #define BME680_COEFF_SIZE       UINT8_C(41)
112 #define BME680_COEFF_ADDR1_LEN      UINT8_C(25)
113 #define BME680_COEFF_ADDR2_LEN      UINT8_C(16)
114
115 /** BME680 field_x related defines */
116 #define BME680_FIELD_LENGTH     UINT8_C(15)
117 #define BME680_FIELD_ADDR_OFFSET    UINT8_C(17)
118
119 /** Soft reset command */
120 #define BME680_SOFT_RESET_CMD   UINT8_C(0xb6)
121
122 /** Error code definitions */
123 #define BME680_OK       INT8_C(0)
124 /* Errors */
125 #define BME680_E_NULL_PTR           INT8_C(-1)
126 #define BME680_E_COM_FAIL           INT8_C(-2)
127 #define BME680_E_DEV_NOT_FOUND      INT8_C(-3)
128 #define BME680_E_INVALID_LENGTH     INT8_C(-4)
129
130 /* Warnings */
131 #define BME680_W_DEFINE_PWR_MODE    INT8_C(1)
132 #define BME680_W_NO_NEW_DATA        INT8_C(2)
133
134 /* Info's */
135 #define BME680_I_MIN_CORRECTION     UINT8_C(1)
136 #define BME680_I_MAX_CORRECTION     UINT8_C(2)
137
138 /** Register map */
139 /** Other coefficient's address */
140 #define BME680_ADDR_RES_HEAT_VAL_ADDR   UINT8_C(0x00)
```

```
141 #define BME680_ADDR_RES_HEAT_RANGE_ADDR UINT8_C(0x02)
142 #define BME680_ADDR_RANGE_SW_ERR_ADDR   UINT8_C(0x04)
143 #define BME680_ADDR_SENS_CONF_START UINT8_C(0x5A)
144 #define BME680_ADDR_GAS_CONF_START  UINT8_C(0x64)
145
146 /** Field settings */
147 #define BME680_FIELD0_ADDR      UINT8_C(0x1d)
148
149 /** Heater settings */
150 #define BME680_RES_HEAT0_ADDR       UINT8_C(0x5a)
151 #define BME680_GAS_WAIT0_ADDR       UINT8_C(0x64)
152
153 /** Sensor configuration registers */
154 #define BME680_CONF_HEAT_CTRL_ADDR      UINT8_C(0x70)
155 #define BME680_CONF_ODR_RUN_GAS_NBC_ADDR    UINT8_C(0x71)
156 #define BME680_CONF_OS_H_ADDR           UINT8_C(0x72)
157 #define BME680_MEM_PAGE_ADDR            UINT8_C(0xf3)
158 #define BME680_CONF_T_P_MODE_ADDR       UINT8_C(0x74)
159 #define BME680_CONF_ODR_FILT_ADDR       UINT8_C(0x75)
160
161 /** Coefficient's address */
162 #define BME680_COEFF_ADDR1  UINT8_C(0x89)
163 #define BME680_COEFF_ADDR2  UINT8_C(0xe1)
164
165 /** Chip identifier */
166 #define BME680_CHIP_ID_ADDR UINT8_C(0xd0)
167
168 /** Soft reset register */
169 #define BME680_SOFT_RESET_ADDR      UINT8_C(0xe0)
170
171 /** Heater control settings */
172 #define BME680_ENABLE_HEATER        UINT8_C(0x00)
173 #define BME680_DISABLE_HEATER       UINT8_C(0x08)
174
175 /** Gas measurement settings */
```

```
176 #define BME680_DISABLE_GAS_MEAS     UINT8_C(0x00)
177 #define BME680_ENABLE_GAS_MEAS      UINT8_C(0x01)
178
179 /** Over-sampling settings */
180 #define BME680_OS_NONE      UINT8_C(0)
181 #define BME680_OS_1X        UINT8_C(1)
182 #define BME680_OS_2X        UINT8_C(2)
183 #define BME680_OS_4X        UINT8_C(3)
184 #define BME680_OS_8X        UINT8_C(4)
185 #define BME680_OS_16X       UINT8_C(5)
186
187 /** IIR filter settings */
188 #define BME680_FILTER_SIZE_0    UINT8_C(0)
189 #define BME680_FILTER_SIZE_1    UINT8_C(1)
190 #define BME680_FILTER_SIZE_3    UINT8_C(2)
191 #define BME680_FILTER_SIZE_7    UINT8_C(3)
192 #define BME680_FILTER_SIZE_15   UINT8_C(4)
193 #define BME680_FILTER_SIZE_31   UINT8_C(5)
194 #define BME680_FILTER_SIZE_63   UINT8_C(6)
195 #define BME680_FILTER_SIZE_127  UINT8_C(7)
196
197 /** Power mode settings */
198 #define BME680_SLEEP_MODE   UINT8_C(0)
199 #define BME680_FORCED_MODE  UINT8_C(1)
200
201 /** Delay related macro declaration */
202 #define BME680_RESET_PERIOD UINT32_C(10)
203
204 /** SPI memory page settings */
205 #define BME680_MEM_PAGE0    UINT8_C(0x10)
206 #define BME680_MEM_PAGE1    UINT8_C(0x00)
207
208 /** Ambient humidity shift value for compensation */
209 #define BME680_HUM_REG_SHIFT_VAL    UINT8_C(4)
210
```

```
211 /** Run gas enable and disable settings */
212 #define BME680_RUN_GAS_DISABLE  UINT8_C(0)
213 #define BME680_RUN_GAS_ENABLE   UINT8_C(1)
214
215 /** Buffer length macro declaration */
216 #define BME680_TMP_BUFFER_LENGTH    UINT8_C(40)
217 #define BME680_REG_BUFFER_LENGTH    UINT8_C(6)
218 #define BME680_FIELD_DATA_LENGTH    UINT8_C(3)
219 #define BME680_GAS_REG_BUF_LENGTH   UINT8_C(20)
220
221 /** Settings selector */
222 #define BME680_OST_SEL          UINT16_C(1)
223 #define BME680_OSP_SEL          UINT16_C(2)
224 #define BME680_OSH_SEL          UINT16_C(4)
225 #define BME680_GAS_MEAS_SEL     UINT16_C(8)
226 #define BME680_FILTER_SEL       UINT16_C(16)
227 #define BME680_HCNTRL_SEL       UINT16_C(32)
228 #define BME680_RUN_GAS_SEL      UINT16_C(64)
229 #define BME680_NBCONV_SEL       UINT16_C(128)
230 #define BME680_GAS_SENSOR_SEL       (BME680_GAS_MEAS_SEL | BME680_RUN_GAS_SEL | BME680_NBCONV_SEL)
231
232 /** Number of conversion settings*/
233 #define BME680_NBCONV_MIN       UINT8_C(0)
234 #define BME680_NBCONV_MAX       UINT8_C(10)
235
236 /** Mask definitions */
237 #define BME680_GAS_MEAS_MSK UINT8_C(0x30)
238 #define BME680_NBCONV_MSK   UINT8_C(0X0F)
239 #define BME680_FILTER_MSK   UINT8_C(0X1C)
240 #define BME680_OST_MSK      UINT8_C(0XE0)
241 #define BME680_OSP_MSK      UINT8_C(0X1C)
242 #define BME680_OSH_MSK      UINT8_C(0X07)
243 #define BME680_HCTRL_MSK    UINT8_C(0x08)
244 #define BME680_RUN_GAS_MSK  UINT8_C(0x10)
245 #define BME680_MODE_MSK     UINT8_C(0x03)
```

```
246 #define BME680_RHRANGE_MSK  UINT8_C(0x30)
247 #define BME680_RSERROR_MSK  UINT8_C(0xf0)
248 #define BME680_NEW_DATA_MSK UINT8_C(0x80)
249 #define BME680_GAS_INDEX_MSK    UINT8_C(0x0f)
250 #define BME680_GAS_RANGE_MSK    UINT8_C(0x0f)
251 #define BME680_GASM_VALID_MSK   UINT8_C(0x20)
252 #define BME680_HEAT_STAB_MSK    UINT8_C(0x10)
253 #define BME680_MEM_PAGE_MSK UINT8_C(0x10)
254 #define BME680_SPI_RD_MSK   UINT8_C(0x80)
255 #define BME680_SPI_WR_MSK   UINT8_C(0x7f)
256 #define BME680_BIT_H1_DATA_MSK  UINT8_C(0x0F)
257
258 /** Bit position definitions for sensor settings */
259 #define BME680_GAS_MEAS_POS UINT8_C(4)
260 #define BME680_FILTER_POS   UINT8_C(2)
261 #define BME680_OST_POS      UINT8_C(5)
262 #define BME680_OSP_POS      UINT8_C(2)
263 #define BME680_RUN_GAS_POS  UINT8_C(4)
264
265 /** Array Index to Field data mapping for Calibration Data*/
266 #define BME680_T2_LSB_REG   (1)
267 #define BME680_T2_MSB_REG   (2)
268 #define BME680_T3_REG       (3)
269 #define BME680_P1_LSB_REG   (5)
270 #define BME680_P1_MSB_REG   (6)
271 #define BME680_P2_LSB_REG   (7)
272 #define BME680_P2_MSB_REG   (8)
273 #define BME680_P3_REG       (9)
274 #define BME680_P4_LSB_REG   (11)
275 #define BME680_P4_MSB_REG   (12)
276 #define BME680_P5_LSB_REG   (13)
277 #define BME680_P5_MSB_REG   (14)
278 #define BME680_P7_REG       (15)
279 #define BME680_P6_REG       (16)
280 #define BME680_P8_LSB_REG   (19)
```

```
281 #define BME680_P8_MSB_REG   (20)
282 #define BME680_P9_LSB_REG   (21)
283 #define BME680_P9_MSB_REG   (22)
284 #define BME680_P10_REG      (23)
285 #define BME680_H2_MSB_REG   (25)
286 #define BME680_H2_LSB_REG   (26)
287 #define BME680_H1_LSB_REG   (26)
288 #define BME680_H1_MSB_REG   (27)
289 #define BME680_H3_REG       (28)
290 #define BME680_H4_REG       (29)
291 #define BME680_H5_REG       (30)
292 #define BME680_H6_REG       (31)
293 #define BME680_H7_REG       (32)
294 #define BME680_T1_LSB_REG   (33)
295 #define BME680_T1_MSB_REG   (34)
296 #define BME680_GH2_LSB_REG  (35)
297 #define BME680_GH2_MSB_REG  (36)
298 #define BME680_GH1_REG      (37)
299 #define BME680_GH3_REG      (38)
300
301 /** BME680 register buffer index settings*/
302 #define BME680_REG_FILTER_INDEX     UINT8_C(5)
303 #define BME680_REG_TEMP_INDEX       UINT8_C(4)
304 #define BME680_REG_PRES_INDEX       UINT8_C(4)
305 #define BME680_REG_HUM_INDEX        UINT8_C(2)
306 #define BME680_REG_NBCONV_INDEX     UINT8_C(1)
307 #define BME680_REG_RUN_GAS_INDEX    UINT8_C(1)
308 #define BME680_REG_HCTRL_INDEX      UINT8_C(0)
309
310 /** BME680 pressure calculation macros */
311 /*! This max value is used to provide precedence to multiplication or division
312  * in pressure compensation equation to achieve least loss of precision and
313  * avoiding overflows.
314  * i.e Comparing value, BME680_MAX_OVERFLOW_VAL = INT32_C(1 << 30)
315  */
```

```
316 #define BME680_MAX_OVERFLOW_VAL      INT32_C(0x40000000)
317
318 /** Macro to combine two 8 bit data's to form a 16 bit data */
319 #define BME680_CONCAT_BYTES(msb, lsb)   (((uint16_t)msb << 8) | (uint16_t)lsb)
320
321 /** Macro to SET and GET BITS of a register */
322 #define BME680_SET_BITS(reg_data, bitname, data) \
323         ((reg_data & ~(bitname##_MSK)) | \
324         ((data << bitname##_POS) & bitname##_MSK))
325 #define BME680_GET_BITS(reg_data, bitname)  ((reg_data & (bitname##_MSK)) >> \
326     (bitname##_POS))
327
328 /** Macro variant to handle the bitname position if it is zero */
329 #define BME680_SET_BITS_POS_0(reg_data, bitname, data) \
330                 ((reg_data & ~(bitname##_MSK)) | \
331                 (data & bitname##_MSK))
332 #define BME680_GET_BITS_POS_0(reg_data, bitname)  (reg_data & (bitname##_MSK))
333
334 /** Type definitions */
335 /*!
336  * Generic communication function pointer
337  * @param[in] dev_id: Place holder to store the id of the device structure
338  *                    Can be used to store the index of the Chip select or
339  *                    I2C address of the device.
340  * @param[in] reg_addr: Used to select the register the where data needs to
341  *                      be read from or written to.
342  * @param[in/out] reg_data: Data array to read/write
343  * @param[in] len: Length of the data array
344  */
345 typedef int8_t (*bme680_com_fptr_t)(uint8_t dev_id, uint8_t reg_addr, uint8_t *data, uint16_t len);
346
347 /*!
348  * Delay function pointer
349  * @param[in] period: Time period in milliseconds
350  */
```

```
351 typedef void (*bme680_delay_fptr_t)(uint32_t period);
352
353 /*!
354  * @brief Interface selection Enumerations
355  */
356 enum bme680_intf {
357     /*! SPI interface */
358     BME680_SPI_INTF,
359     /*! I2C interface */
360     BME680_I2C_INTF
361 };
362
363 /* structure definitions */
364 /*!
365  * @brief Sensor field data structure
366  */
367 struct  bme680_field_data {
368     /*! Contains new_data, gasm_valid & heat_stab */
369     uint8_t status;
370     /*! The index of the heater profile used */
371     uint8_t gas_index;
372     /*! Measurement index to track order */
373     uint8_t meas_index;
374
375 #ifndef BME680_FLOAT_POINT_COMPENSATION
376     /*! Temperature in degree celsius x100 */
377     int16_t temperature;
378     /*! Pressure in Pascal */
379     uint32_t pressure;
380     /*! Humidity in % relative humidity x1000 */
381     uint32_t humidity;
382     /*! Gas resistance in Ohms */
383     uint32_t gas_resistance;
384 #else
385     /*! Temperature in degree celsius */
```

```
386     float temperature;
387     /*! Pressure in Pascal */
388     float pressure;
389     /*! Humidity in % relative humidity x1000 */
390     float humidity;
391     /*! Gas resistance in Ohms */
392     float gas_resistance;
393 
394 #endif
395 
396 };
397 
398 /*!
399  * @brief Structure to hold the Calibration data
400  */
401 struct  bme680_calib_data {
402     /*! Variable to store calibrated humidity data */
403     uint16_t par_h1;
404     /*! Variable to store calibrated humidity data */
405     uint16_t par_h2;
406     /*! Variable to store calibrated humidity data */
407     int8_t par_h3;
408     /*! Variable to store calibrated humidity data */
409     int8_t par_h4;
410     /*! Variable to store calibrated humidity data */
411     int8_t par_h5;
412     /*! Variable to store calibrated humidity data */
413     uint8_t par_h6;
414     /*! Variable to store calibrated humidity data */
415     int8_t par_h7;
416     /*! Variable to store calibrated gas data */
417     int8_t par_gh1;
418     /*! Variable to store calibrated gas data */
419     int16_t par_gh2;
420     /*! Variable to store calibrated gas data */
```

```
421     int8_t par_gh3;
422     /*! Variable to store calibrated temperature data */
423     uint16_t par_t1;
424     /*! Variable to store calibrated temperature data */
425     int16_t par_t2;
426     /*! Variable to store calibrated temperature data */
427     int8_t par_t3;
428     /*! Variable to store calibrated pressure data */
429     uint16_t par_p1;
430     /*! Variable to store calibrated pressure data */
431     int16_t par_p2;
432     /*! Variable to store calibrated pressure data */
433     int8_t par_p3;
434     /*! Variable to store calibrated pressure data */
435     int16_t par_p4;
436     /*! Variable to store calibrated pressure data */
437     int16_t par_p5;
438     /*! Variable to store calibrated pressure data */
439     int8_t par_p6;
440     /*! Variable to store calibrated pressure data */
441     int8_t par_p7;
442     /*! Variable to store calibrated pressure data */
443     int16_t par_p8;
444     /*! Variable to store calibrated pressure data */
445     int16_t par_p9;
446     /*! Variable to store calibrated pressure data */
447     uint8_t par_p10;
448
449 #ifndef BME680_FLOAT_POINT_COMPENSATION
450     /*! Variable to store t_fine size */
451     int32_t t_fine;
452 #else
453     /*! Variable to store t_fine size */
454     float t_fine;
455 #endif
```

```
456     /*! Variable to store heater resistance range */
457     uint8_t res_heat_range;
458     /*! Variable to store heater resistance value */
459     int8_t res_heat_val;
460     /*! Variable to store error range */
461     int8_t range_sw_err;
462 };
463 
464 /*!
465  * @brief BME680 sensor settings structure which comprises of ODR,
466  * over-sampling and filter settings.
467  */
468 struct  bme680_tph_sett {
469     /*! Humidity oversampling */
470     uint8_t os_hum;
471     /*! Temperature oversampling */
472     uint8_t os_temp;
473     /*! Pressure oversampling */
474     uint8_t os_pres;
475     /*! Filter coefficient */
476     uint8_t filter;
477 };
478 
479 /*!
480  * @brief BME680 gas sensor which comprises of gas settings
481  *  and status parameters
482  */
483 struct  bme680_gas_sett {
484     /*! Variable to store nb conversion */
485     uint8_t nb_conv;
486     /*! Variable to store heater control */
487     uint8_t heatr_ctrl;
488     /*! Run gas enable value */
489     uint8_t run_gas;
490     /*! Heater temperature value */
```

```
491     uint16_t heatr_temp;
492     /*! Duration profile value */
493     uint16_t heatr_dur;
494 };
495 
496 /*!
497  * @brief BME680 device structure
498  */
499 struct  bme680_dev {
500     /*! Chip Id */
501     uint8_t chip_id;
502     /*! Device Id */
503     uint8_t dev_id;
504     /*! SPI/I2C interface */
505     enum bme680_intf intf;
506     /*! Memory page used */
507     uint8_t mem_page;
508     /*! Ambient temperature in Degree C */
509     int8_t amb_temp;
510     /*! Sensor calibration data */
511     struct bme680_calib_data calib;
512     /*! Sensor settings */
513     struct bme680_tph_sett tph_sett;
514     /*! Gas Sensor settings */
515     struct bme680_gas_sett gas_sett;
516     /*! Sensor power modes */
517     uint8_t power_mode;
518     /*! New sensor fields */
519     uint8_t new_fields;
520     /*! Store the info messages */
521     uint8_t info_msg;
522     /*! Bus read function pointer */
523     bme680_com_fptr_t read;
524     /*! Bus write function pointer */
525     bme680_com_fptr_t write;
```

```
526     /*! delay function pointer */
527     bme680_delay_fptr_t delay_ms;
528     /*! Communication function result */
529     int8_t com_rslt;
530 };
531
532
533
534 #endif /* BME680_DEFS_H_ */
535 /** @}*/
536 /** @}*/
537
```

```
/**\mainpage
 * Copyright (c) 2020 Bosch Sensortec GmbH. All rights reserved.
 *
 * BSD-3-Clause
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. 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.
 *
 * 3. Neither the name of the copyright holder nor the names of its
 *    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 THE
 * 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.
 *
 * File     bme680.c
 * @date    23 Jan 2020
 * @version 3.5.10
```

```
 36  *
 37  */
 38
 39 /*! @file bme680.c
 40  @brief Sensor driver for BME680 sensor */
 41 #include "bme680.h"
 42
 43 /*!
 44  * @brief This internal API is used to read the calibrated data from the sensor.
 45  *
 46  * This function is used to retrieve the calibration
 47  * data from the image registers of the sensor.
 48  *
 49  * @note Registers 89h  to A1h for calibration data 1 to 24
 50  *       from bit 0 to 7
 51  * @note Registers E1h to F0h for calibration data 25 to 40
 52  *       from bit 0 to 7
 53  * @param[in] dev   :Structure instance of bme680_dev.
 54  *
 55  * @return Result of API execution status.
 56  * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
 57  */
 58 static int8_t get_calib_data(struct bme680_dev *dev);
 59
 60 /*!
 61  * @brief This internal API is used to set the gas configuration of the sensor.
 62  *
 63  * @param[in] dev   :Structure instance of bme680_dev.
 64  *
 65  * @return Result of API execution status.
 66  * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
 67  */
 68 static int8_t set_gas_config(struct bme680_dev *dev);
 69
 70 /*!
```

```
 71   * @brief This internal API is used to get the gas configuration of the sensor.
 72   * @note heatr_temp and heatr_dur values are currently register data
 73   * and not the actual values set
 74   *
 75   * @param[in] dev   :Structure instance of bme680_dev.
 76   *
 77   * @return Result of API execution status.
 78   * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
 79   */
 80  static int8_t get_gas_config(struct bme680_dev *dev);
 81
 82  /*!
 83   * @brief This internal API is used to calculate the Heat duration value.
 84   *
 85   * @param[in] dur   :Value of the duration to be shared.
 86   *
 87   * @return uint8_t threshold duration after calculation.
 88   */
 89  static uint8_t calc_heater_dur(uint16_t dur);
 90
 91  #ifndef BME680_FLOAT_POINT_COMPENSATION
 92
 93  /*!
 94   * @brief This internal API is used to calculate the temperature value.
 95   *
 96   * @param[in] dev   :Structure instance of bme680_dev.
 97   * @param[in] temp_adc  :Contains the temperature ADC value .
 98   *
 99   * @return uint32_t calculated temperature.
100   */
101  static int16_t calc_temperature(uint32_t temp_adc, struct bme680_dev *dev);
102
103  /*!
104   * @brief This internal API is used to calculate the pressure value.
105   *
```

```
106  * @param[in] dev   :Structure instance of bme680_dev.
107  * @param[in] pres_adc  :Contains the pressure ADC value .
108  *
109  * @return uint32_t calculated pressure.
110  */
111 static uint32_t calc_pressure(uint32_t pres_adc, const struct bme680_dev *dev);
112
113 /*!
114  * @brief This internal API is used to calculate the humidity value.
115  *
116  * @param[in] dev   :Structure instance of bme680_dev.
117  * @param[in] hum_adc   :Contains the humidity ADC value.
118  *
119  * @return uint32_t calculated humidity.
120  */
121 static uint32_t calc_humidity(uint16_t hum_adc, const struct bme680_dev *dev);
122
123 /*!
124  * @brief This internal API is used to calculate the Gas Resistance value.
125  *
126  * @param[in] dev       :Structure instance of bme680_dev.
127  * @param[in] gas_res_adc   :Contains the Gas Resistance ADC value.
128  * @param[in] gas_range     :Contains the range of gas values.
129  *
130  * @return uint32_t calculated gas resistance.
131  */
132 static uint32_t calc_gas_resistance(uint16_t gas_res_adc, uint8_t gas_range, const struct bme680_dev *dev);
133
134 /*!
135  * @brief This internal API is used to calculate the Heat Resistance value.
136  *
137  * @param[in] dev   : Structure instance of bme680_dev
138  * @param[in] temp  : Contains the target temperature value.
139  *
140  * @return uint8_t calculated heater resistance.
```

```
141  */
142 static uint8_t calc_heater_res(uint16_t temp, const struct bme680_dev *dev);
143
144 #else
145 /*!
146  * @brief This internal API is used to calculate the
147  * temperature value value in float format
148  *
149  * @param[in] dev   :Structure instance of bme680_dev.
150  * @param[in] temp_adc  :Contains the temperature ADC value .
151  *
152  * @return Calculated temperature in float
153  */
154 static float calc_temperature(uint32_t temp_adc, struct bme680_dev *dev);
155
156 /*!
157  * @brief This internal API is used to calculate the
158  * pressure value value in float format
159  *
160  * @param[in] dev   :Structure instance of bme680_dev.
161  * @param[in] pres_adc  :Contains the pressure ADC value .
162  *
163  * @return Calculated pressure in float.
164  */
165 static float calc_pressure(uint32_t pres_adc, const struct bme680_dev *dev);
166
167 /*!
168  * @brief This internal API is used to calculate the
169  * humidity value value in float format
170  *
171  * @param[in] dev   :Structure instance of bme680_dev.
172  * @param[in] hum_adc   :Contains the humidity ADC value.
173  *
174  * @return Calculated humidity in float.
175  */
```

```
176 static float calc_humidity(uint16_t hum_adc, const struct bme680_dev *dev);
177
178 /*!
179  * @brief This internal API is used to calculate the
180  * gas resistance value value in float format
181  *
182  * @param[in] dev       :Structure instance of bme680_dev.
183  * @param[in] gas_res_adc   :Contains the Gas Resistance ADC value.
184  * @param[in] gas_range     :Contains the range of gas values.
185  *
186  * @return Calculated gas resistance in float.
187  */
188 static float calc_gas_resistance(uint16_t gas_res_adc, uint8_t gas_range, const struct bme680_dev *dev);
189
190 /*!
191  * @brief This internal API is used to calculate the
192  * heater resistance value in float format
193  *
194  * @param[in] temp  : Contains the target temperature value.
195  * @param[in] dev   : Structure instance of bme680_dev.
196  *
197  * @return Calculated heater resistance in float.
198  */
199 static float calc_heater_res(uint16_t temp, const struct bme680_dev *dev);
200
201 #endif
202
203 /*!
204  * @brief This internal API is used to calculate the field data of sensor.
205  *
206  * @param[out] data :Structure instance to hold the data
207  * @param[in] dev   :Structure instance of bme680_dev.
208  *
209  *  @return int8_t result of the field data from sensor.
210  */
```

```c
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
 */
```

```
246 static int8_t get_mem_page(struct bme680_dev *dev);
247
248 /*!
249  * @brief This internal API is used to validate the device pointer for
250  * null conditions.
251  *
252  * @param[in] dev   :Structure instance of bme680_dev.
253  *
254  * @return Result of API execution status
255  * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
256  */
257 static int8_t null_ptr_check(const struct bme680_dev *dev);
258
259 /*!
260  * @brief This internal API is used to check the boundary
261  * conditions.
262  *
263  * @param[in] value :pointer to the value.
264  * @param[in] min   :minimum value.
265  * @param[in] max   :maximum value.
266  * @param[in] dev   :Structure instance of bme680_dev.
267  *
268  * @return Result of API execution status
269  * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
270  */
271 static int8_t boundary_check(uint8_t *value, uint8_t min, uint8_t max, struct bme680_dev *dev);
272
273 /****************** Global Function Definitions *******************************/
274 /*!
275  *@brief This API is the entry point.
276  *It reads the chip-id and calibration data from the sensor.
277  */
278 int8_t bme680_init(struct bme680_dev *dev)
279 {
280     int8_t rslt;
```

```c

    /* 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) {
            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);
```

```
316             if (rslt == BME680_OK)
317                 reg_addr = reg_addr | BME680_SPI_RD_MSK;
318         }
319         dev->com_rslt = dev->read(dev->dev_id, reg_addr, reg_data, len);
320         if (dev->com_rslt != 0)
321             rslt = BME680_E_COM_FAIL;
322     }
323 
324     return rslt;
325 }
326 
327 /*!
328  * @brief This API writes the given data to the register address
329  * of the sensor.
330  */
331 int8_t bme680_set_regs(const uint8_t *reg_addr, const uint8_t *reg_data, uint8_t len, struct bme680_dev *dev)
332 {
333     int8_t rslt;
334     /* Length of the temporary buffer is 2*(length of register)*/
335     uint8_t tmp_buff[BME680_TMP_BUFFER_LENGTH] = { 0 };
336     uint16_t index;
337 
338     /* Check for null pointer in the device structure*/
339     rslt = null_ptr_check(dev);
340     if (rslt == BME680_OK) {
341         if ((len > 0) && (len < BME680_TMP_BUFFER_LENGTH / 2)) {
342             /* Interleave the 2 arrays */
343             for (index = 0; index < len; index++) {
344                 if (dev->intf == BME680_SPI_INTF) {
345                     /* Set the memory page */
346                     rslt = set_mem_page(reg_addr[index], dev);
347                     tmp_buff[(2 * index)] = reg_addr[index] & BME680_SPI_WR_MSK;
348                 } else {
349                     tmp_buff[(2 * index)] = reg_addr[index];
350                 }
```

```
351                 tmp_buff[(2 * index) + 1] = reg_data[index];
352             }
353             /* Write the interleaved array */
354             if (rslt == BME680_OK) {
355                 dev->com_rslt = dev->write(dev->dev_id, tmp_buff[0], &tmp_buff[1], (2 * len) - 1);
356                 if (dev->com_rslt != 0)
357                     rslt = BME680_E_COM_FAIL;
358             }
359         } else {
360             rslt = BME680_E_INVALID_LENGTH;
361         }
362     }
363
364     return rslt;
365 }
366
367 /*!
368  * @brief This API performs the soft reset of the sensor.
369  */
370 int8_t bme680_soft_reset(struct bme680_dev *dev)
371 {
372     int8_t rslt;
373     uint8_t reg_addr = BME680_SOFT_RESET_ADDR;
374     /* 0xb6 is the soft reset command */
375     uint8_t soft_rst_cmd = BME680_SOFT_RESET_CMD;
376
377     /* Check for null pointer in the device structure*/
378     rslt = null_ptr_check(dev);
379     if (rslt == BME680_OK) {
380         if (dev->intf == BME680_SPI_INTF)
381             rslt = get_mem_page(dev);
382
383         /* Reset the device */
384         if (rslt == BME680_OK) {
385             rslt = bme680_set_regs(&reg_addr, &soft_rst_cmd, 1, dev);
```

```
386             /* Wait for 5ms */
387             dev->delay_ms(BME680_RESET_PERIOD);
388 
389             if (rslt == BME680_OK) {
390                 /* After reset get the memory page */
391                 if (dev->intf == BME680_SPI_INTF)
392                     rslt = get_mem_page(dev);
393             }
394         }
395     }
396 
397     return rslt;
398 }
399 
400 /*!
401  * @brief This API is used to set the oversampling, filter and T,P,H, gas selection
402  * settings in the sensor.
403  */
404 int8_t bme680_set_sensor_settings(uint16_t desired_settings, struct bme680_dev *dev)
405 {
406     int8_t rslt;
407     uint8_t reg_addr;
408     uint8_t data = 0;
409     uint8_t count = 0;
410     uint8_t reg_array[BME680_REG_BUFFER_LENGTH] = { 0 };
411     uint8_t data_array[BME680_REG_BUFFER_LENGTH] = { 0 };
412     uint8_t intended_power_mode = dev->power_mode; /* Save intended power mode */
413 
414     /* Check for null pointer in the device structure*/
415     rslt = null_ptr_check(dev);
416     if (rslt == BME680_OK) {
417         if (desired_settings & BME680_GAS_MEAS_SEL)
418             rslt = set_gas_config(dev);
419 
420         dev->power_mode = BME680_SLEEP_MODE;
```

```
421         if (rslt == BME680_OK)
422             rslt = bme680_set_sensor_mode(dev);
423
424         /* Selecting the filter */
425         if (desired_settings & BME680_FILTER_SEL) {
426             rslt = boundary_check(&dev->tph_sett.filter, BME680_FILTER_SIZE_0, BME680_FILTER_SIZE_127, dev);
427             reg_addr = BME680_CONF_ODR_FILT_ADDR;
428
429             if (rslt == BME680_OK)
430                 rslt = bme680_get_regs(reg_addr, &data, 1, dev);
431
432             if (desired_settings & BME680_FILTER_SEL)
433                 data = BME680_SET_BITS(data, BME680_FILTER, dev->tph_sett.filter);
434
435             reg_array[count] = reg_addr; /* Append configuration */
436             data_array[count] = data;
437             count++;
438         }
439
440         /* Selecting heater control for the sensor */
441         if (desired_settings & BME680_HCNTRL_SEL) {
442             rslt = boundary_check(&dev->gas_sett.heatr_ctrl, BME680_ENABLE_HEATER,
443                 BME680_DISABLE_HEATER, dev);
444             reg_addr = BME680_CONF_HEAT_CTRL_ADDR;
445
446             if (rslt == BME680_OK)
447                 rslt = bme680_get_regs(reg_addr, &data, 1, dev);
448             data = BME680_SET_BITS_POS_0(data, BME680_HCTRL, dev->gas_sett.heatr_ctrl);
449
450             reg_array[count] = reg_addr; /* Append configuration */
451             data_array[count] = data;
452             count++;
453         }
454
455         /* Selecting heater T,P oversampling for the sensor */
```

```
456         if (desired_settings & (BME680_OST_SEL | BME680_OSP_SEL)) {
457             rslt = boundary_check(&dev->tph_sett.os_temp, BME680_OS_NONE, BME680_OS_16X, dev);
458             reg_addr = BME680_CONF_T_P_MODE_ADDR;
459 
460             if (rslt == BME680_OK)
461                 rslt = bme680_get_regs(reg_addr, &data, 1, dev);
462 
463             if (desired_settings & BME680_OST_SEL)
464                 data = BME680_SET_BITS(data, BME680_OST, dev->tph_sett.os_temp);
465 
466             if (desired_settings & BME680_OSP_SEL)
467                 data = BME680_SET_BITS(data, BME680_OSP, dev->tph_sett.os_pres);
468 
469             reg_array[count] = reg_addr;
470             data_array[count] = data;
471             count++;
472         }
473 
474         /* Selecting humidity oversampling for the sensor */
475         if (desired_settings & BME680_OSH_SEL) {
476             rslt = boundary_check(&dev->tph_sett.os_hum, BME680_OS_NONE, BME680_OS_16X, dev);
477             reg_addr = BME680_CONF_OS_H_ADDR;
478 
479             if (rslt == BME680_OK)
480                 rslt = bme680_get_regs(reg_addr, &data, 1, dev);
481             data = BME680_SET_BITS_POS_0(data, BME680_OSH, dev->tph_sett.os_hum);
482 
483             reg_array[count] = reg_addr; /* Append configuration */
484             data_array[count] = data;
485             count++;
486         }
487 
488         /* Selecting the runGas and NB conversion settings for the sensor */
489         if (desired_settings & (BME680_RUN_GAS_SEL | BME680_NBCONV_SEL)) {
490             rslt = boundary_check(&dev->gas_sett.run_gas, BME680_RUN_GAS_DISABLE,
```

```
491                 BME680_RUN_GAS_ENABLE, dev);
492             if (rslt == BME680_OK) {
493                 /* Validate boundary conditions */
494                 rslt = boundary_check(&dev->gas_sett.nb_conv, BME680_NBCONV_MIN,
495                     BME680_NBCONV_MAX, dev);
496             }
497 
498             reg_addr = BME680_CONF_ODR_RUN_GAS_NBC_ADDR;
499 
500             if (rslt == BME680_OK)
501                 rslt = bme680_get_regs(reg_addr, &data, 1, dev);
502 
503             if (desired_settings & BME680_RUN_GAS_SEL)
504                 data = BME680_SET_BITS(data, BME680_RUN_GAS, dev->gas_sett.run_gas);
505 
506             if (desired_settings & BME680_NBCONV_SEL)
507                 data = BME680_SET_BITS_POS_0(data, BME680_NBCONV, dev->gas_sett.nb_conv);
508 
509             reg_array[count] = reg_addr; /* Append configuration */
510             data_array[count] = data;
511             count++;
512         }
513 
514         if (rslt == BME680_OK)
515             rslt = bme680_set_regs(reg_array, data_array, count, dev);
516 
517         /* Restore previous intended power mode */
518         dev->power_mode = intended_power_mode;
519     }
520 
521     return rslt;
522 }
523 
524 /*!
525  * @brief This API is used to get the oversampling, filter and T,P,H, gas selection
```

```
526  * settings in the sensor.
527  */
528 int8_t bme680_get_sensor_settings(uint16_t desired_settings, struct bme680_dev *dev)
529 {
530     int8_t rslt;
531     /* starting address of the register array for burst read*/
532     uint8_t reg_addr = BME680_CONF_HEAT_CTRL_ADDR;
533     uint8_t data_array[BME680_REG_BUFFER_LENGTH] = { 0 };
534
535     /* Check for null pointer in the device structure*/
536     rslt = null_ptr_check(dev);
537     if (rslt == BME680_OK) {
538         rslt = bme680_get_regs(reg_addr, data_array, BME680_REG_BUFFER_LENGTH, dev);
539
540         if (rslt == BME680_OK) {
541             if (desired_settings & BME680_GAS_MEAS_SEL)
542                 rslt = get_gas_config(dev);
543
544             /* get the T,P,H ,Filter,ODR settings here */
545             if (desired_settings & BME680_FILTER_SEL)
546                 dev->tph_sett.filter = BME680_GET_BITS(data_array[BME680_REG_FILTER_INDEX],
547                     BME680_FILTER);
548
549             if (desired_settings & (BME680_OST_SEL | BME680_OSP_SEL)) {
550                 dev->tph_sett.os_temp = BME680_GET_BITS(data_array[BME680_REG_TEMP_INDEX], BME680_OST);
551                 dev->tph_sett.os_pres = BME680_GET_BITS(data_array[BME680_REG_PRES_INDEX], BME680_OSP);
552             }
553
554             if (desired_settings & BME680_OSH_SEL)
555                 dev->tph_sett.os_hum = BME680_GET_BITS_POS_0(data_array[BME680_REG_HUM_INDEX],
556                     BME680_OSH);
557
558             /* get the gas related settings */
559             if (desired_settings & BME680_HCNTRL_SEL)
560                 dev->gas_sett.heatr_ctrl = BME680_GET_BITS_POS_0(data_array[BME680_REG_HCTRL_INDEX],
```

```
561                     BME680_HCTRL);
562
563             if (desired_settings & (BME680_RUN_GAS_SEL | BME680_NBCONV_SEL)) {
564                 dev->gas_sett.nb_conv = BME680_GET_BITS_POS_0(data_array[BME680_REG_NBCONV_INDEX],
565                     BME680_NBCONV);
566                 dev->gas_sett.run_gas = BME680_GET_BITS(data_array[BME680_REG_RUN_GAS_INDEX],
567                     BME680_RUN_GAS);
568             }
569         }
570     } else {
571         rslt = BME680_E_NULL_PTR;
572     }
573
574     return rslt;
575 }
576
577 /*!
578  * @brief This API is used to set the power mode of the sensor.
579  */
580 int8_t bme680_set_sensor_mode(struct bme680_dev *dev)
581 {
582     int8_t rslt;
583     uint8_t tmp_pow_mode;
584     uint8_t pow_mode = 0;
585     uint8_t reg_addr = BME680_CONF_T_P_MODE_ADDR;
586
587     /* Check for null pointer in the device structure*/
588     rslt = null_ptr_check(dev);
589     if (rslt == BME680_OK) {
590         /* Call repeatedly until in sleep */
591         do {
592             rslt = bme680_get_regs(BME680_CONF_T_P_MODE_ADDR, &tmp_pow_mode, 1, dev);
593             if (rslt == BME680_OK) {
594                 /* Put to sleep before changing mode */
595                 pow_mode = (tmp_pow_mode & BME680_MODE_MSK);
```

```
596
597                 if (pow_mode != BME680_SLEEP_MODE) {
598                     tmp_pow_mode = tmp_pow_mode & (~BME680_MODE_MSK); /* Set to sleep */
599                     rslt = bme680_set_regs(&reg_addr, &tmp_pow_mode, 1, dev);
600                     dev->delay_ms(BME680_POLL_PERIOD_MS);
601                 }
602             }
603         } while (pow_mode != BME680_SLEEP_MODE);
604
605         /* Already in sleep */
606         if (dev->power_mode != BME680_SLEEP_MODE) {
607             tmp_pow_mode = (tmp_pow_mode & ~BME680_MODE_MSK) | (dev->power_mode & BME680_MODE_MSK);
608             if (rslt == BME680_OK)
609                 rslt = bme680_set_regs(&reg_addr, &tmp_pow_mode, 1, dev);
610         }
611     }
612
613     return rslt;
614 }
615
616 /*!
617  * @brief This API is used to get the power mode of the sensor.
618  */
619 int8_t bme680_get_sensor_mode(struct bme680_dev *dev)
620 {
621     int8_t rslt;
622     uint8_t mode;
623
624     /* Check for null pointer in the device structure*/
625     rslt = null_ptr_check(dev);
626     if (rslt == BME680_OK) {
627         rslt = bme680_get_regs(BME680_CONF_T_P_MODE_ADDR, &mode, 1, dev);
628         /* Masking the other register bit info*/
629         dev->power_mode = mode & BME680_MODE_MSK;
630     }
```

```
631
632     return rslt;
633 }
634
635 /*!
636  * @brief This API is used to set the profile duration of the sensor.
637  */
638 void bme680_set_profile_dur(uint16_t duration, struct bme680_dev *dev)
639 {
640     uint32_t tph_dur; /* Calculate in us */
641     uint32_t meas_cycles;
642     uint8_t os_to_meas_cycles[6] = {0, 1, 2, 4, 8, 16};
643
644     meas_cycles = os_to_meas_cycles[dev->tph_sett.os_temp];
645     meas_cycles += os_to_meas_cycles[dev->tph_sett.os_pres];
646     meas_cycles += os_to_meas_cycles[dev->tph_sett.os_hum];
647
648     /* TPH measurement duration */
649     tph_dur = meas_cycles * UINT32_C(1963);
650     tph_dur += UINT32_C(477 * 4); /* TPH switching duration */
651     tph_dur += UINT32_C(477 * 5); /* Gas measurement duration */
652     tph_dur += UINT32_C(500); /* Get it to the closest whole number.*/
653     tph_dur /= UINT32_C(1000); /* Convert to ms */
654
655     tph_dur += UINT32_C(1); /* Wake up duration of 1ms */
656     /* The remaining time should be used for heating */
657     dev->gas_sett.heatr_dur = duration - (uint16_t) tph_dur;
658 }
659
660 /*!
661  * @brief This API is used to get the profile duration of the sensor.
662  */
663 void bme680_get_profile_dur(uint16_t *duration, const struct bme680_dev *dev)
664 {
665     uint32_t tph_dur; /* Calculate in us */
```

```
666     uint32_t meas_cycles;
667     uint8_t os_to_meas_cycles[6] = {0, 1, 2, 4, 8, 16};
668
669     meas_cycles = os_to_meas_cycles[dev->tph_sett.os_temp];
670     meas_cycles += os_to_meas_cycles[dev->tph_sett.os_pres];
671     meas_cycles += os_to_meas_cycles[dev->tph_sett.os_hum];
672
673     /* TPH measurement duration */
674     tph_dur = meas_cycles * UINT32_C(1963);
675     tph_dur += UINT32_C(477 * 4); /* TPH switching duration */
676     tph_dur += UINT32_C(477 * 5); /* Gas measurement duration */
677     tph_dur += UINT32_C(500); /* Get it to the closest whole number.*/
678     tph_dur /= UINT32_C(1000); /* Convert to ms */
679
680     tph_dur += UINT32_C(1); /* Wake up duration of 1ms */
681
682     *duration = (uint16_t) tph_dur;
683
684     /* Get the gas duration only when the run gas is enabled */
685     if (dev->gas_sett.run_gas) {
686         /* The remaining time should be used for heating */
687         *duration += dev->gas_sett.heatr_dur;
688     }
689 }
690
691 /*!
692  * @brief This API reads the pressure, temperature and humidity and gas data
693  * from the sensor, compensates the data and store it in the bme680_data
694  * structure instance passed by the user.
695  */
696 int8_t bme680_get_sensor_data(struct bme680_field_data *data, struct bme680_dev *dev)
697 {
698     int8_t rslt;
699
700     /* Check for null pointer in the device structure*/
```

```
701     rslt = null_ptr_check(dev);
702     if (rslt == BME680_OK) {
703         /* Reading the sensor data in forced mode only */
704         rslt = read_field_data(data, dev);
705         if (rslt == BME680_OK) {
706             if (data->status & BME680_NEW_DATA_MSK)
707                 dev->new_fields = 1;
708             else
709                 dev->new_fields = 0;
710         }
711     }
712
713     return rslt;
714 }
715
716 /*!
717  * @brief This internal API is used to read the calibrated data from the sensor.
718  */
719 static int8_t get_calib_data(struct bme680_dev *dev)
720 {
721     int8_t rslt;
722     uint8_t coeff_array[BME680_COEFF_SIZE] = { 0 };
723     uint8_t temp_var = 0; /* Temporary variable */
724
725     /* Check for null pointer in the device structure*/
726     rslt = null_ptr_check(dev);
727     if (rslt == BME680_OK) {
728         rslt = bme680_get_regs(BME680_COEFF_ADDR1, coeff_array, BME680_COEFF_ADDR1_LEN, dev);
729         /* Append the second half in the same array */
730         if (rslt == BME680_OK)
731             rslt = bme680_get_regs(BME680_COEFF_ADDR2, &coeff_array[BME680_COEFF_ADDR1_LEN]
732             , BME680_COEFF_ADDR2_LEN, dev);
733
734         /* Temperature related coefficients */
735         dev->calib.par_t1 = (uint16_t) (BME680_CONCAT_BYTES(coeff_array[BME680_T1_MSB_REG],
```

```
736             coeff_array[BME680_T1_LSB_REG]));
737         dev->calib.par_t2 = (int16_t) (BME680_CONCAT_BYTES(coeff_array[BME680_T2_MSB_REG],
738             coeff_array[BME680_T2_LSB_REG]));
739         dev->calib.par_t3 = (int8_t) (coeff_array[BME680_T3_REG]);
740 
741         /* Pressure related coefficients */
742         dev->calib.par_p1 = (uint16_t) (BME680_CONCAT_BYTES(coeff_array[BME680_P1_MSB_REG],
743             coeff_array[BME680_P1_LSB_REG]));
744         dev->calib.par_p2 = (int16_t) (BME680_CONCAT_BYTES(coeff_array[BME680_P2_MSB_REG],
745             coeff_array[BME680_P2_LSB_REG]));
746         dev->calib.par_p3 = (int8_t) coeff_array[BME680_P3_REG];
747         dev->calib.par_p4 = (int16_t) (BME680_CONCAT_BYTES(coeff_array[BME680_P4_MSB_REG],
748             coeff_array[BME680_P4_LSB_REG]));
749         dev->calib.par_p5 = (int16_t) (BME680_CONCAT_BYTES(coeff_array[BME680_P5_MSB_REG],
750             coeff_array[BME680_P5_LSB_REG]));
751         dev->calib.par_p6 = (int8_t) (coeff_array[BME680_P6_REG]);
752         dev->calib.par_p7 = (int8_t) (coeff_array[BME680_P7_REG]);
753         dev->calib.par_p8 = (int16_t) (BME680_CONCAT_BYTES(coeff_array[BME680_P8_MSB_REG],
754             coeff_array[BME680_P8_LSB_REG]));
755         dev->calib.par_p9 = (int16_t) (BME680_CONCAT_BYTES(coeff_array[BME680_P9_MSB_REG],
756             coeff_array[BME680_P9_LSB_REG]));
757         dev->calib.par_p10 = (uint8_t) (coeff_array[BME680_P10_REG]);
758 
759         /* Humidity related coefficients */
760         dev->calib.par_h1 = (uint16_t) (((uint16_t) coeff_array[BME680_H1_MSB_REG] << BME680_HUM_REG_SHIFT_VAL)
761             | (coeff_array[BME680_H1_LSB_REG] & BME680_BIT_H1_DATA_MSK));
762         dev->calib.par_h2 = (uint16_t) (((uint16_t) coeff_array[BME680_H2_MSB_REG] << BME680_HUM_REG_SHIFT_VAL)
763             | ((coeff_array[BME680_H2_LSB_REG]) >> BME680_HUM_REG_SHIFT_VAL));
764         dev->calib.par_h3 = (int8_t) coeff_array[BME680_H3_REG];
765         dev->calib.par_h4 = (int8_t) coeff_array[BME680_H4_REG];
766         dev->calib.par_h5 = (int8_t) coeff_array[BME680_H5_REG];
767         dev->calib.par_h6 = (uint8_t) coeff_array[BME680_H6_REG];
768         dev->calib.par_h7 = (int8_t) coeff_array[BME680_H7_REG];
769 
770         /* Gas heater related coefficients */
```

```
771         dev->calib.par_gh1 = (int8_t) coeff_array[BME680_GH1_REG];
772         dev->calib.par_gh2 = (int16_t) (BME680_CONCAT_BYTES(coeff_array[BME680_GH2_MSB_REG],
773             coeff_array[BME680_GH2_LSB_REG]));
774         dev->calib.par_gh3 = (int8_t) coeff_array[BME680_GH3_REG];
775
776         /* Other coefficients */
777         if (rslt == BME680_OK) {
778             rslt = bme680_get_regs(BME680_ADDR_RES_HEAT_RANGE_ADDR, &temp_var, 1, dev);
779
780             dev->calib.res_heat_range = ((temp_var & BME680_RHRANGE_MSK) / 16);
781             if (rslt == BME680_OK) {
782                 rslt = bme680_get_regs(BME680_ADDR_RES_HEAT_VAL_ADDR, &temp_var, 1, dev);
783
784                 dev->calib.res_heat_val = (int8_t) temp_var;
785                 if (rslt == BME680_OK)
786                     rslt = bme680_get_regs(BME680_ADDR_RANGE_SW_ERR_ADDR, &temp_var, 1, dev);
787             }
788         }
789         dev->calib.range_sw_err = ((int8_t) temp_var & (int8_t) BME680_RSERROR_MSK) / 16;
790     }
791
792     return rslt;
793 }
794
795 /*!
796  * @brief This internal API is used to set the gas configuration of the sensor.
797  */
798 static int8_t set_gas_config(struct bme680_dev *dev)
799 {
800     int8_t rslt;
801
802     /* Check for null pointer in the device structure*/
803     rslt = null_ptr_check(dev);
804     if (rslt == BME680_OK) {
805
```

```
806         uint8_t reg_addr[2] = {0};
807         uint8_t reg_data[2] = {0};
808 
809         if (dev->power_mode == BME680_FORCED_MODE) {
810             reg_addr[0] = BME680_RES_HEAT0_ADDR;
811             reg_data[0] = calc_heater_res(dev->gas_sett.heatr_temp, dev);
812             reg_addr[1] = BME680_GAS_WAIT0_ADDR;
813             reg_data[1] = calc_heater_dur(dev->gas_sett.heatr_dur);
814             dev->gas_sett.nb_conv = 0;
815         } else {
816             rslt = BME680_W_DEFINE_PWR_MODE;
817         }
818         if (rslt == BME680_OK)
819             rslt = bme680_set_regs(reg_addr, reg_data, 2, dev);
820     }
821 
822     return rslt;
823 }
824 
825 /*!
826  * @brief This internal API is used to get the gas configuration of the sensor.
827  * @note heatr_temp and heatr_dur values are currently register data
828  * and not the actual values set
829  */
830 static int8_t get_gas_config(struct bme680_dev *dev)
831 {
832     int8_t rslt;
833     /* starting address of the register array for burst read*/
834     uint8_t reg_addr1 = BME680_ADDR_SENS_CONF_START;
835     uint8_t reg_addr2 = BME680_ADDR_GAS_CONF_START;
836     uint8_t reg_data = 0;
837 
838     /* Check for null pointer in the device structure*/
839     rslt = null_ptr_check(dev);
840     if (rslt == BME680_OK) {
```

```
841         if (BME680_SPI_INTF == dev->intf) {
842             /* Memory page switch the SPI address*/
843             rslt = set_mem_page(reg_addr1, dev);
844         }
845 
846         if (rslt == BME680_OK) {
847             rslt = bme680_get_regs(reg_addr1, &reg_data, 1, dev);
848             if (rslt == BME680_OK) {
849                 dev->gas_sett.heatr_temp = reg_data;
850                 rslt = bme680_get_regs(reg_addr2, &reg_data, 1, dev);
851                 if (rslt == BME680_OK) {
852                     /* Heating duration register value */
853                     dev->gas_sett.heatr_dur = reg_data;
854                 }
855             }
856         }
857     }
858 
859     return rslt;
860 }
861 
862 #ifndef BME680_FLOAT_POINT_COMPENSATION
863 
864 /*!
865  * @brief This internal API is used to calculate the temperature value.
866  */
867 static int16_t calc_temperature(uint32_t temp_adc, struct bme680_dev *dev)
868 {
869     int64_t var1;
870     int64_t var2;
871     int64_t var3;
872     int16_t calc_temp;
873 
874     var1 = ((int32_t) temp_adc >> 3) - ((int32_t) dev->calib.par_t1 << 1);
875     var2 = (var1 * (int32_t) dev->calib.par_t2) >> 11;
```

```
876     var3 = ((var1 >> 1) * (var1 >> 1)) >> 12;
877     var3 = ((var3) * ((int32_t) dev->calib.par_t3 << 4)) >> 14;
878     dev->calib.t_fine = (int32_t) (var2 + var3);
879     calc_temp = (int16_t) (((dev->calib.t_fine * 5) + 128) >> 8);
880
881     return calc_temp;
882 }
883
884 /*!
885  * @brief This internal API is used to calculate the pressure value.
886  */
887 static uint32_t calc_pressure(uint32_t pres_adc, const struct bme680_dev *dev)
888 {
889     int32_t var1;
890     int32_t var2;
891     int32_t var3;
892     int32_t pressure_comp;
893
894     var1 = (((int32_t)dev->calib.t_fine) >> 1) - 64000;
895     var2 = ((((var1 >> 2) * (var1 >> 2)) >> 11) *
896         (int32_t)dev->calib.par_p6) >> 2;
897     var2 = var2 + ((var1 * (int32_t)dev->calib.par_p5) << 1);
898     var2 = (var2 >> 2) + ((int32_t)dev->calib.par_p4 << 16);
899     var1 = (((((var1 >> 2) * (var1 >> 2)) >> 13) *
900         ((int32_t)dev->calib.par_p3 << 5)) >> 3) +
901         (((int32_t)dev->calib.par_p2 * var1) >> 1);
902     var1 = var1 >> 18;
903     var1 = ((32768 + var1) * (int32_t)dev->calib.par_p1) >> 15;
904     pressure_comp = 1048576 - pres_adc;
905     pressure_comp = (int32_t)((pressure_comp - (var2 >> 12)) * ((uint32_t)3125));
906     if (pressure_comp >= BME680_MAX_OVERFLOW_VAL)
907         pressure_comp = ((pressure_comp / var1) << 1);
908     else
909         pressure_comp = ((pressure_comp << 1) / var1);
910     var1 = ((int32_t)dev->calib.par_p9 * (int32_t)(((pressure_comp >> 3) *
```

```
911         (pressure_comp >> 3)) >> 13)) >> 12;
912     var2 = ((int32_t)(pressure_comp >> 2) *
913         (int32_t)dev->calib.par_p8) >> 13;
914     var3 = ((int32_t)(pressure_comp >> 8) * (int32_t)(pressure_comp >> 8) *
915         (int32_t)(pressure_comp >> 8) *
916         (int32_t)dev->calib.par_p10) >> 17;
917
918     pressure_comp = (int32_t)(pressure_comp) + ((var1 + var2 + var3 +
919         ((int32_t)dev->calib.par_p7 << 7)) >> 4);
920
921     return (uint32_t)pressure_comp;
922
923 }
924
925 /*!
926  * @brief This internal API is used to calculate the humidity value.
927  */
928 static uint32_t calc_humidity(uint16_t hum_adc, const struct bme680_dev *dev)
929 {
930     int32_t var1;
931     int32_t var2;
932     int32_t var3;
933     int32_t var4;
934     int32_t var5;
935     int32_t var6;
936     int32_t temp_scaled;
937     int32_t calc_hum;
938
939     temp_scaled = (((int32_t) dev->calib.t_fine * 5) + 128) >> 8;
940     var1 = (int32_t) (hum_adc - ((int32_t) ((int32_t) dev->calib.par_h1 * 16)))
941         - (((temp_scaled * (int32_t) dev->calib.par_h3) / ((int32_t) 100)) >> 1);
942     var2 = ((int32_t) dev->calib.par_h2
943         * (((temp_scaled * (int32_t) dev->calib.par_h4) / ((int32_t) 100))
944             + (((temp_scaled * ((temp_scaled * (int32_t) dev->calib.par_h5) / ((int32_t) 100))) >> 6)
945                 / ((int32_t) 100)) + (int32_t) (1 << 14))) >> 10;
```

```
946     var3 = var1 * var2;
947     var4 = (int32_t) dev->calib.par_h6 << 7;
948     var4 = ((var4) + ((temp_scaled * (int32_t) dev->calib.par_h7) / ((int32_t) 100))) >> 4;
949     var5 = ((var3 >> 14) * (var3 >> 14)) >> 10;
950     var6 = (var4 * var5) >> 1;
951     calc_hum = (((var3 + var6) >> 10) * ((int32_t) 1000)) >> 12;
952
953     if (calc_hum > 100000) /* Cap at 100%rH */
954         calc_hum = 100000;
955     else if (calc_hum < 0)
956         calc_hum = 0;
957
958     return (uint32_t) calc_hum;
959 }
960
961 /*!
962  * @brief This internal API is used to calculate the Gas Resistance value.
963  */
964 static uint32_t calc_gas_resistance(uint16_t gas_res_adc, uint8_t gas_range, const struct bme680_dev *dev)
965 {
966     int64_t var1;
967     uint64_t var2;
968     int64_t var3;
969     uint32_t calc_gas_res;
970     /**Look up table 1 for the possible gas range values */
971     uint32_t lookupTable1[16] = { UINT32_C(2147483647), UINT32_C(2147483647), UINT32_C(2147483647), UINT32_C
        (2147483647),
972         UINT32_C(2147483647), UINT32_C(2126008810), UINT32_C(2147483647), UINT32_C(2130303777),
973         UINT32_C(2147483647), UINT32_C(2147483647), UINT32_C(2143188679), UINT32_C(2136746228),
974         UINT32_C(2147483647), UINT32_C(2126008810), UINT32_C(2147483647), UINT32_C(2147483647) };
975     /**Look up table 2 for the possible gas range values */
976     uint32_t lookupTable2[16] = { UINT32_C(4096000000), UINT32_C(2048000000), UINT32_C(1024000000), UINT32_C
        (512000000),
977         UINT32_C(255744255), UINT32_C(127110228), UINT32_C(64000000), UINT32_C(32258064), UINT32_C(16016016),
978         UINT32_C(8000000), UINT32_C(4000000), UINT32_C(2000000), UINT32_C(1000000), UINT32_C(500000),
```

```
979         UINT32_C(250000), UINT32_C(125000) };
980
981     var1 = (int64_t) ((1340 + (5 * (int64_t) dev->calib.range_sw_err)) *
982         ((int64_t) lookupTable1[gas_range])) >> 16;
983     var2 = (((int64_t) ((int64_t) gas_res_adc << 15) - (int64_t) (16777216)) + var1);
984     var3 = (((int64_t) lookupTable2[gas_range] * (int64_t) var1) >> 9);
985     calc_gas_res = (uint32_t) ((var3 + ((int64_t) var2 >> 1)) / (int64_t) var2);
986
987     return calc_gas_res;
988 }
989
990 /*!
991  * @brief This internal API is used to calculate the Heat Resistance value.
992  */
993 static uint8_t calc_heater_res(uint16_t temp, const struct bme680_dev *dev)
994 {
995     uint8_t heatr_res;
996     int32_t var1;
997     int32_t var2;
998     int32_t var3;
999     int32_t var4;
1000    int32_t var5;
1001    int32_t heatr_res_x100;
1002
1003    if (temp > 400) /* Cap temperature */
1004        temp = 400;
1005
1006    var1 = (((int32_t) dev->amb_temp * dev->calib.par_gh3) / 1000) * 256;
1007    var2 = (dev->calib.par_gh1 + 784) * (((((dev->calib.par_gh2 + 154009) * temp * 5) / 100) + 3276800) / 10);
1008    var3 = var1 + (var2 / 2);
1009    var4 = (var3 / (dev->calib.res_heat_range + 4));
1010    var5 = (131 * dev->calib.res_heat_val) + 65536;
1011    heatr_res_x100 = (int32_t) (((var4 / var5) - 250) * 34);
1012    heatr_res = (uint8_t) ((heatr_res_x100 + 50) / 100);
1013
```

```
1014      return heatr_res;
1015  }
1016
1017  #else
1018
1019
1020  /*!
1021   * @brief This internal API is used to calculate the
1022   * temperature value in float format
1023   */
1024  static float calc_temperature(uint32_t temp_adc, struct bme680_dev *dev)
1025  {
1026      float var1 = 0;
1027      float var2 = 0;
1028      float calc_temp = 0;
1029
1030      /* calculate var1 data */
1031      var1  = ((((float)temp_adc / 16384.0f) - ((float)dev->calib.par_t1 / 1024.0f))
1032              * ((float)dev->calib.par_t2));
1033
1034      /* calculate var2 data */
1035      var2  = (((((float)temp_adc / 131072.0f) - ((float)dev->calib.par_t1 / 8192.0f)) *
1036          (((float)temp_adc / 131072.0f) - ((float)dev->calib.par_t1 / 8192.0f))) *
1037          ((float)dev->calib.par_t3 * 16.0f));
1038
1039      /* t_fine value*/
1040      dev->calib.t_fine = (var1 + var2);
1041
1042      /* compensated temperature data*/
1043      calc_temp  = ((dev->calib.t_fine) / 5120.0f);
1044
1045      return calc_temp;
1046  }
1047
1048  /*!
```

```
1049  * @brief This internal API is used to calculate the
1050  * pressure value in float format
1051  */
1052 static float calc_pressure(uint32_t pres_adc, const struct bme680_dev *dev)
1053 {
1054     float var1 = 0;
1055     float var2 = 0;
1056     float var3 = 0;
1057     float calc_pres = 0;
1058
1059     var1 = (((float)dev->calib.t_fine / 2.0f) - 64000.0f);
1060     var2 = var1 * var1 * (((float)dev->calib.par_p6) / (131072.0f));
1061     var2 = var2 + (var1 * ((float)dev->calib.par_p5) * 2.0f);
1062     var2 = (var2 / 4.0f) + (((float)dev->calib.par_p4) * 65536.0f);
1063     var1 = (((((float)dev->calib.par_p3 * var1 * var1) / 16384.0f)
1064         + ((float)dev->calib.par_p2 * var1)) / 524288.0f);
1065     var1 = ((1.0f + (var1 / 32768.0f)) * ((float)dev->calib.par_p1));
1066     calc_pres = (1048576.0f - ((float)pres_adc));
1067
1068     /* Avoid exception caused by division by zero */
1069     if ((int)var1 != 0) {
1070         calc_pres = (((calc_pres - (var2 / 4096.0f)) * 6250.0f) / var1);
1071         var1 = (((float)dev->calib.par_p9) * calc_pres * calc_pres) / 2147483648.0f;
1072         var2 = calc_pres * (((float)dev->calib.par_p8) / 32768.0f);
1073         var3 = ((calc_pres / 256.0f) * (calc_pres / 256.0f) * (calc_pres / 256.0f)
1074             * (dev->calib.par_p10 / 131072.0f));
1075         calc_pres = (calc_pres + (var1 + var2 + var3 + ((float)dev->calib.par_p7 * 128.0f)) / 16.0f);
1076     } else {
1077         calc_pres = 0;
1078     }
1079
1080     return calc_pres;
1081 }
1082
1083 /*!
```

```
1084  * @brief This internal API is used to calculate the
1085  * humidity value in float format
1086  */
1087 static float calc_humidity(uint16_t hum_adc, const struct bme680_dev *dev)
1088 {
1089     float calc_hum = 0;
1090     float var1 = 0;
1091     float var2 = 0;
1092     float var3 = 0;
1093     float var4 = 0;
1094     float temp_comp;
1095
1096     /* compensated temperature data*/
1097     temp_comp  = ((dev->calib.t_fine) / 5120.0f);
1098
1099     var1 = (float)((float)hum_adc) - (((float)dev->calib.par_h1 * 16.0f) + (((float)dev->calib.par_h3 / 2.0f)
1100         * temp_comp));
1101
1102     var2 = var1 * ((float)(((float) dev->calib.par_h2 / 262144.0f) * (1.0f + (((float)dev->calib.par_h4 /
        16384.0f)
1103         * temp_comp) + (((float)dev->calib.par_h5 / 1048576.0f) * temp_comp * temp_comp))));
1104
1105     var3 = (float) dev->calib.par_h6 / 16384.0f;
1106
1107     var4 = (float) dev->calib.par_h7 / 2097152.0f;
1108
1109     calc_hum = var2 + ((var3 + (var4 * temp_comp)) * var2 * var2);
1110
1111     if (calc_hum > 100.0f)
1112         calc_hum = 100.0f;
1113     else if (calc_hum < 0.0f)
1114         calc_hum = 0.0f;
1115
1116     return calc_hum;
1117 }
```

```
1118
1119 /*!
1120  * @brief This internal API is used to calculate the
1121  * gas resistance value in float format
1122  */
1123 static float calc_gas_resistance(uint16_t gas_res_adc, uint8_t gas_range, const struct bme680_dev *dev)
1124 {
1125     float calc_gas_res;
1126     float var1 = 0;
1127     float var2 = 0;
1128     float var3 = 0;
1129
1130     const float lookup_k1_range[16] = {
1131     0.0, 0.0, 0.0, 0.0, 0.0, -1.0, 0.0, -0.8,
1132     0.0, 0.0, -0.2, -0.5, 0.0, -1.0, 0.0, 0.0};
1133     const float lookup_k2_range[16] = {
1134     0.0, 0.0, 0.0, 0.0, 0.1, 0.7, 0.0, -0.8,
1135     -0.1, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
1136
1137     var1 = (1340.0f + (5.0f * dev->calib.range_sw_err));
1138     var2 = (var1) * (1.0f + lookup_k1_range[gas_range]/100.0f);
1139     var3 = 1.0f + (lookup_k2_range[gas_range]/100.0f);
1140
1141     calc_gas_res = 1.0f / (float)(var3 * (0.000000125f) * (float)(1 << gas_range) * (((((float)gas_res_adc)
1142         - 512.0f)/var2) + 1.0f));
1143
1144     return calc_gas_res;
1145 }
1146
1147 /*!
1148  * @brief This internal API is used to calculate the
1149  * heater resistance value in float format
1150  */
1151 static float calc_heater_res(uint16_t temp, const struct bme680_dev *dev)
1152 {
```

```
1153      float var1 = 0;
1154      float var2 = 0;
1155      float var3 = 0;
1156      float var4 = 0;
1157      float var5 = 0;
1158      float res_heat = 0;
1159
1160      if (temp > 400) /* Cap temperature */
1161          temp = 400;
1162
1163      var1 = (((float)dev->calib.par_gh1 / (16.0f)) + 49.0f);
1164      var2 = ((((float)dev->calib.par_gh2 / (32768.0f)) * (0.0005f)) + 0.00235f);
1165      var3 = ((float)dev->calib.par_gh3 / (1024.0f));
1166      var4 = (var1 * (1.0f + (var2 * (float)temp)));
1167      var5 = (var4 + (var3 * (float)dev->amb_temp));
1168      res_heat = (uint8_t)(3.4f * ((var5 * (4 / (4 + (float)dev->calib.res_heat_range)) *
1169          (1/(1 + ((float) dev->calib.res_heat_val * 0.002f)))) - 25));
1170
1171      return res_heat;
1172  }
1173
1174  #endif
1175
1176  /*!
1177   * @brief This internal API is used to calculate the Heat duration value.
1178   */
1179  static uint8_t calc_heater_dur(uint16_t dur)
1180  {
1181      uint8_t factor = 0;
1182      uint8_t durval;
1183
1184      if (dur >= 0xfc0) {
1185          durval = 0xff; /* Max duration*/
1186      } else {
1187          while (dur > 0x3F) {
```

```
1188                dur = dur / 4;
1189                factor += 1;
1190            }
1191            durval = (uint8_t) (dur + (factor * 64));
1192        }
1193
1194        return durval;
1195    }
1196
1197    /*!
1198     * @brief This internal API is used to calculate the field data of sensor.
1199     */
1200    static int8_t read_field_data(struct bme680_field_data *data, struct bme680_dev *dev)
1201    {
1202        int8_t rslt;
1203        uint8_t buff[BME680_FIELD_LENGTH] = { 0 };
1204        uint8_t gas_range;
1205        uint32_t adc_temp;
1206        uint32_t adc_pres;
1207        uint16_t adc_hum;
1208        uint16_t adc_gas_res;
1209        uint8_t tries = 10;
1210
1211        /* Check for null pointer in the device structure*/
1212        rslt = null_ptr_check(dev);
1213        do {
1214            if (rslt == BME680_OK) {
1215                rslt = bme680_get_regs(((uint8_t) (BME680_FIELD0_ADDR)), buff, (uint16_t) BME680_FIELD_LENGTH,
1216                    dev);
1217
1218                data->status = buff[0] & BME680_NEW_DATA_MSK;
1219                data->gas_index = buff[0] & BME680_GAS_INDEX_MSK;
1220                data->meas_index = buff[1];
1221
1222                /* read the raw data from the sensor */
```

```
1223                adc_pres = (uint32_t) (((uint32_t) buff[2] * 4096) | ((uint32_t) buff[3] * 16)
1224                    | ((uint32_t) buff[4] / 16));
1225                adc_temp = (uint32_t) (((uint32_t) buff[5] * 4096) | ((uint32_t) buff[6] * 16)
1226                    | ((uint32_t) buff[7] / 16));
1227                adc_hum = (uint16_t) (((uint32_t) buff[8] * 256) | (uint32_t) buff[9]);
1228                adc_gas_res = (uint16_t) ((uint32_t) buff[13] * 4 | (((uint32_t) buff[14]) / 64));
1229                gas_range = buff[14] & BME680_GAS_RANGE_MSK;
1230
1231                data->status |= buff[14] & BME680_GASM_VALID_MSK;
1232                data->status |= buff[14] & BME680_HEAT_STAB_MSK;
1233
1234                if (data->status & BME680_NEW_DATA_MSK) {
1235                    data->temperature = calc_temperature(adc_temp, dev);
1236                    data->pressure = calc_pressure(adc_pres, dev);
1237                    data->humidity = calc_humidity(adc_hum, dev);
1238                    data->gas_resistance = calc_gas_resistance(adc_gas_res, gas_range, dev);
1239                    break;
1240                }
1241                /* Delay to poll the data */
1242                dev->delay_ms(BME680_POLL_PERIOD_MS);
1243            }
1244            tries--;
1245        } while (tries);
1246
1247        if (!tries)
1248            rslt = BME680_W_NO_NEW_DATA;
1249
1250        return rslt;
1251    }
1252
1253    /*!
1254     * @brief This internal API is used to set the memory page based on register address.
1255     */
1256    static int8_t set_mem_page(uint8_t reg_addr, struct bme680_dev *dev)
1257    {
```

```
1258      int8_t rslt;
1259      uint8_t reg;
1260      uint8_t mem_page;
1261
1262      /* Check for null pointers in the device structure*/
1263      rslt = null_ptr_check(dev);
1264      if (rslt == BME680_OK) {
1265          if (reg_addr > 0x7f)
1266              mem_page = BME680_MEM_PAGE1;
1267          else
1268              mem_page = BME680_MEM_PAGE0;
1269
1270          if (mem_page != dev->mem_page) {
1271              dev->mem_page = mem_page;
1272
1273              dev->com_rslt = dev->read(dev->dev_id, BME680_MEM_PAGE_ADDR | BME680_SPI_RD_MSK, &reg, 1);
1274              if (dev->com_rslt != 0)
1275                  rslt = BME680_E_COM_FAIL;
1276
1277              if (rslt == BME680_OK) {
1278                  reg = reg & (~BME680_MEM_PAGE_MSK);
1279                  reg = reg | (dev->mem_page & BME680_MEM_PAGE_MSK);
1280
1281                  dev->com_rslt = dev->write(dev->dev_id, BME680_MEM_PAGE_ADDR & BME680_SPI_WR_MSK,
1282                      &reg, 1);
1283                  if (dev->com_rslt != 0)
1284                      rslt = BME680_E_COM_FAIL;
1285              }
1286          }
1287      }
1288
1289      return rslt;
1290  }
1291
1292  /*!
```

```
1293  * @brief This internal API is used to get the memory page based on register address.
1294  */
1295 static int8_t get_mem_page(struct bme680_dev *dev)
1296 {
1297     int8_t rslt;
1298     uint8_t reg;
1299
1300     /* Check for null pointer in the device structure*/
1301     rslt = null_ptr_check(dev);
1302     if (rslt == BME680_OK) {
1303         dev->com_rslt = dev->read(dev->dev_id, BME680_MEM_PAGE_ADDR | BME680_SPI_RD_MSK, &reg, 1);
1304         if (dev->com_rslt != 0)
1305             rslt = BME680_E_COM_FAIL;
1306         else
1307             dev->mem_page = reg & BME680_MEM_PAGE_MSK;
1308     }
1309
1310     return rslt;
1311 }
1312
1313 /*!
1314  * @brief This internal API is used to validate the boundary
1315  * conditions.
1316  */
1317 static int8_t boundary_check(uint8_t *value, uint8_t min, uint8_t max, struct bme680_dev *dev)
1318 {
1319     int8_t rslt = BME680_OK;
1320
1321     if (value != NULL) {
1322         /* Check if value is below minimum value */
1323         if (*value < min) {
1324             /* Auto correct the invalid value to minimum value */
1325             *value = min;
1326             dev->info_msg |= BME680_I_MIN_CORRECTION;
1327         }
```

```
1328         /* Check if value is above maximum value */
1329         if (*value > max) {
1330             /* Auto correct the invalid value to maximum value */
1331             *value = max;
1332             dev->info_msg |= BME680_I_MAX_CORRECTION;
1333         }
1334     } else {
1335         rslt = BME680_E_NULL_PTR;
1336     }
1337
1338     return rslt;
1339 }
1340
1341 /*!
1342  * @brief This internal API is used to validate the device structure pointer for
1343  * null conditions.
1344  */
1345 static int8_t null_ptr_check(const struct bme680_dev *dev)
1346 {
1347     int8_t rslt;
1348
1349     if ((dev == NULL) || (dev->read == NULL) || (dev->write == NULL) || (dev->delay_ms == NULL)) {
1350         /* Device structure pointer is not valid */
1351         rslt = BME680_E_NULL_PTR;
1352     } else {
1353         /* Device structure is fine */
1354         rslt = BME680_OK;
1355     }
1356
1357     return rslt;
1358 }
1359
```

```c
/*
 * final_program.c
 *
 * Created: 5/2/2020 3:40:08 PM
 * Author : Aaron
 This program uses the BME680 sensor
 to output the temperature, pressure,
 humidity and gas resistance measurements.
 For this program, we will be using the
 BME680 sensor API that is given by Bosch to
 help make the calculations and everything
 easier to use and follow.This code is also
 supposed to output T,H, and P before the
 pushbutton press, and then H,P,G after the
 pushbutton press.
 */


#include "saml21j18b.h"
#include "lcd_dog_driver.h"
#include "SERCOM4_RS232.h"
#include "bme680.h"
#include "stdio.h"
#include "console_io_support.h"
#include "stdint.h"

unsigned char * ARRAY_PINCFG0 = (unsigned char*) & REG_PORT_PINCFG0;
unsigned char * ARRAY_PMUX0 = (unsigned char*) & REG_PORT_PMUX0;

//bme680 needed functions
void init_spi_bme680(void);
void user_delay_ms(uint32_t period);
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);
```

```
36
37  //switching screens
38  void pushbuttonpress(_Bool *point, struct bme680_field_data data);
39  void tph(struct bme680_field_data data);
40  void phg(struct bme680_field_data data);
41
42
43
44
45  int main(void)
46  {
47      UART_init();
48      init_lcd_dog();
49      init_spi_bme680();
50
51      //initializing the bme680 device structure
52      struct bme680_dev sensor;
53      //sensor.dev_id = 0;
54      sensor.intf = BME680_SPI_INTF;
55      sensor.read = user_spi_read;
56      sensor.write = user_spi_write;
57      sensor.delay_ms = user_delay_ms;
58      sensor.amb_temp = 25;
59      int8_t rslt = BME680_OK;
60      rslt = bme680_init(&sensor);
61
62      uint8_t set_required_settings;
63
64      //setting temp, pressure and humidity settings
65      sensor.tph_sett.os_hum = BME680_OS_2X;
66      sensor.tph_sett.os_pres = BME680_OS_4X;
67      sensor.tph_sett.os_temp = BME680_OS_8X;
68      sensor.tph_sett.filter = BME680_FILTER_SIZE_3;
69
70      //setting the remaining gas sensor settings and link
```

```
71      //the heating profile
72      sensor.gas_sett.run_gas = BME680_ENABLE_GAS_MEAS;
73      //creating a ramp heat waveform in 3 steps
74      sensor.gas_sett.heatr_temp = 200;//degrees Celsius
75      sensor.gas_sett.heatr_dur = 85; //milliseconds
76
77      //selecting power mode
78      sensor.power_mode = BME680_FORCED_MODE;
79
80      //setting the required sensor settings needed
81      set_required_settings = (BME680_OST_SEL)|(BME680_OSP_SEL)|
82      (BME680_OSH_SEL)|(BME680_FILTER_SEL)|(BME680_GAS_SENSOR_SEL);
83
84      //setting the desired sensor settings
85      rslt = bme680_set_sensor_settings(set_required_settings,&sensor);
86      //set the power mode
87      rslt = bme680_set_sensor_mode(&sensor);
88
89      //getting the measurement duration needed so it the sensor
90      //could sleep or wait until the measurement is complete
91       uint16_t meas_period;
92       bme680_get_profile_dur(&meas_period,&sensor);
93
94       //state = 0 will be tph code, state = 1 will be phg code
95       _Bool state = 0;
96       //says  if pb is pressed
97       _Bool pbpress = 0;
98       _Bool* point = &state;
99
100      REG_PORT_DIR0 &= ~(0x04);
101      ARRAY_PINCFG0[2] |=6;
102      REG_PORT_OUT0 = 0x04;
103
104      struct bme680_field_data data;
105
```

```
106
107     while (1)
108     {
109         //wait until the measurement is ready
110         user_delay_ms(meas_period);
111         rslt = bme680_get_sensor_data(&data, &sensor);
112         printf("T: %.2f degC, P: %.2f hPa, H %.2f %%rH ", data.temperature / 100.0f,
113         data.pressure / 100.0f, data.humidity / 1000.0f );
114
115         //avoid using measurements from an unstable heating setup
116         if((data.status & BME680_GASM_VALID_MSK))
117         {
118             printf(", G: %ld ohms", data.gas_resistance);
119         }
120
121         printf("\r\n");
122
123
124
125         //this is the code for outputting to the LCD
126         //PA02 is used for the pushbutton press
127         if(!(REG_PORT_IN0&0x04))
128         //if it is pressed, then wait for it to be released
129         {
130             //while it is still pressed, then just output current state
131             //the current state will be taken, and then the next state will
132             //be calculated assuming that the pushbutton is still being held
133             while(!(REG_PORT_IN0&0x04))
134             {
135                 //in case of state 0, output tph
136                 if(state==0)
137                 {
138                     tph(data);
139                     init_spi_bme680();
140                     // Trigger the next measurement if you would like to read data out
```

```
141                     // continuously
142                     if(sensor.power_mode == BME680_FORCED_MODE)
143                     {
144                         rslt = bme680_set_sensor_mode(&sensor);
145                     }
146                     user_delay_ms(meas_period);
147                     rslt = bme680_get_sensor_data(&data, &sensor);
148                     printf("T: %.2f degC, P: %.2f hPa, H %.2f %%rH ", data.temperature / 100.0f,
149                     data.pressure / 100.0f, data.humidity / 1000.0f );
150 
151                     //avoid using measurements from an unstable heating setup
152                     if((data.status & BME680_GASM_VALID_MSK))
153                     {
154                     printf(", G: %ld ohms", data.gas_resistance);
155                     }
156                     printf("\r\n");
157                 }
158 
159                 //in case of state 1, output phg
160                 else if (state == 1)
161                 {
162                     phg(data);
163                     init_spi_bme680();
164                     // Trigger the next measurement if you would like to read data out
165                     // continuously
166                     if(sensor.power_mode == BME680_FORCED_MODE)
167                     {
168                     rslt = bme680_set_sensor_mode(&sensor);
169                     }
170                     user_delay_ms(meas_period);
171                     rslt = bme680_get_sensor_data(&data, &sensor);
172                     printf("T: %.2f degC, P: %.2f hPa, H %.2f %%rH ", data.temperature / 100.0f,
173                     data.pressure / 100.0f, data.humidity / 1000.0f );
174                     //avoid using measurements from an unstable heating setup
175                     if((data.status & BME680_GASM_VALID_MSK))
```

```
176                     {
177                     printf(", G: %ld ohms", data.gas_resistance);
178                     }
179                     printf("\r\n");
180                 }
181 
182             }
183             //if it is released, update pbpress and change state
184             if((REG_PORT_IN0&0x04))
185             {
186                 pbpress=1;
187                 pushbuttonpress(point,data);
188                 pbpress = 0;
189             }
190         }
191 
192         //check which state is currently on and then move on to
193         //calculate the next result
194         if(state==0)
195         {
196             //state 0 is task 1 code
197             tph(data);
198         }
199         else
200         {
201             //state 1 is task 2 code
202             phg(data);
203         }
204 
205         init_spi_bme680();
206         // Trigger the next measurement if you would like to read data out
207         // continuously
208         if(sensor.power_mode == BME680_FORCED_MODE)
209         {
210             rslt = bme680_set_sensor_mode(&sensor);
```

```
211         }
212     }
213 }
214
215
216
217
218
219 //Functions for the BME680 sensor
220 /************************
221 NAME:       init_spi_bme680
222 ASSUMES:    The BME680 sensor is interfaced by SPI through SERCOM1
223 SERCOM1 is being used for the SPI data transfer
224 PA16 (PAD[0])---> SDI
225 PA17 (PAD[1])---> SCK
226 PA19 (PAD[3])--->SD0
227 PB07 ---->/CSB
228
229 RETURNS:    N/A
230 MODIFIES:   N/A
231 CALLED BY:
232 DESCRITION: init SPI port for communication with the BME680
233 ************************************************************************/
234 void init_spi_bme680(void)
235 {
236     //this initializes the SERCOM SPI unit
237     //REG_MCLK_AHBMASK |= 0x04; //APBC bus enabled by default
238     //REG_MCLK_APBCMASK|= 0x02; //SERCOM1 APBC bus clock enabled
239     //by default
240     //using generic clock generator 0 (4 MHz) for peripheral clock
241     REG_GCLK_PCHCTRL19 = 0x40;// enabling SERCOM1 core clock
242
243     ARRAY_PINCFG0[16] |= 1; //setting PMUX config
244     ARRAY_PINCFG0[17] |= 1; //setting PMUX config
245     ARRAY_PINCFG0[19] |= 1; //setting PMUX config
```

```
246     ARRAY_PMUX0[8] = 0x22; //PA16 = SDO, PA17 = SCK
247     ARRAY_PMUX0[9] = 0x20; //PA19 = SDI
248
249     REG_SERCOM1_SPI_CTRLA = 1; //software reset SERCOM1
250     while(REG_SERCOM1_SPI_CTRLA & 1){};//waiting for reset to finish
251     //SDI = PAD[0], SDO = PAD[3], SCK = PAD[1], using PB07 for /CSB
252     REG_SERCOM1_SPI_CTRLA = 0x3030000C;//CPOL and CPHA are 11
253     REG_SERCOM1_SPI_CTRLB = 0x020000; //8-bit data
254     //SPI clock is going to be written to 2MHz
255     REG_SERCOM1_SPI_BAUD = 0;
256     REG_SERCOM1_SPI_CTRLA|=2; //SERCOM1 enabled
257
258     //setting up /CSB, and turning it off
259     REG_PORT_DIR1 |=128;
260     REG_PORT_OUTSET1 = 128;
261 }
262
263
264
265 /************************
266 NAME:       user_delay_ms()
267 ASSUMES:    An integer value is given that specifies amount of ms
268 delay that is needed
269
270 RETURNS:    N/A
271 MODIFIES:   N/A
272 CALLED BY:  N/A
273 DESCRITION: delay that lasts (uint32_t) period ms long
274 ************************************************************/
275 void user_delay_ms(uint32_t period)
276 {
277     for(;period>0;period--)
278     {
279         for(int i=0;i<199;i++)
280         {
```

```
281             __asm("nop");
282         }
283     }
284 }
285 
286 
287 
288 /************************
289 NAME:       spi_transfer()
290 ASSUMES:    1) The BME680 sensor is interfaced by SPI through SERCOM1
291 SERCOM1 is being used for the SPI data transfer.
292 2) Also assumes that data is given as a parameter to transfer
293 
294 
295 RETURNS:    N/A
296 MODIFIES:   N/A
297 CALLED BY: spi_read_BME680, spi_write_BME680
298 DESCRITION: transfers data between the sensor and SERCOM1
299 ************************************************************************/
300 uint8_t spi_transfer(uint8_t data)
301 {
302     while(!(REG_SERCOM1_SPI_INTFLAG&0x01)){}
303     REG_SERCOM1_SPI_DATA = data;
304     while(!(REG_SERCOM1_SPI_INTFLAG&0x04)){}
305     return REG_SERCOM1_SPI_DATA;
306 }
307 
308 
309 
310 /************************
311 NAME:       user_spi_read()
312 ASSUMES:
313 1) Address of register to read from is given
314 2) Chip ID is given (no need to do anything for it in our case)
315 3) Pointer to data is given
```

```
316 4) Amount of bytes needed to be written sequentially is given
317
318 RETURNS:    data in the address and rslt
319 MODIFIES:   N/A
320 CALLED BY:  N/A
321 DESCRITION: Returns a result that says if the read function was
322 done successfully
323 ****************************************************************/
324 int8_t user_spi_read(uint8_t dev_id, uint8_t reg_addr, uint8_t* reg_data, uint16_t len)
325 {
326     //operation is done successfully
327     int8_t rslt = 0;
328
329     //enabling the BME680
330     REG_PORT_OUTCLR1 = 128;
331
332     //control byte for the transfer
333     spi_transfer(reg_addr|(1<<7));
334     //this will read through each address and increment automatically
335     for(uint16_t i = 0;i<len;i++)
336     {
337         *reg_data = spi_transfer(0x00);
338         reg_data++;
339     }
340     //disabling the BME680
341     REG_PORT_OUTSET1 = 128;
342     return rslt;
343 }
344 //
345 /************************
346 NAME:       user_spi_write()
347 ASSUMES:
348 1) Address of register to write to is given
349 2) Chip ID is given (no need to do anything for it in our case)
350 3) Pointer to data is given
```

```
351 4) Amount of bytes needed to be written sequentially is given
352
353 RETURNS:    rslt, which says that the operation is successful
354 MODIFIES:   N/A
355 CALLED BY:  N/A
356 DESCRITION: writes specified data to a specific register
357 ****************************************************************/
358 int8_t user_spi_write(uint8_t dev_id, uint8_t reg_addr, uint8_t* reg_data, uint16_t len)
359 {
360     //operation is done successfully
361     int8_t rslt =0;
362
363     //enabling the BME680
364     REG_PORT_OUTCLR1 = 128;
365
366     spi_transfer(reg_addr);
367     for(uint8_t i = 0;i<len;i++)
368         {
369             spi_transfer(*reg_data);
370             reg_data++;
371         }
372     //disabling the BME680
373     REG_PORT_OUTSET1 = 128;
374     return rslt;
375 }
376
377
378 //Functions to output the TPH and PHG to separate pages
379 //when pushbutton is pressed
380 /************************
381 NAME:       tph()
382 ASSUMES:
383
384 RETURNS:    N/A
385 MODIFIES:   N/A
```

```
386 CALLED BY:  N/A
387 DESCRITION: When a pushbutton is pressed, the screen will change
388 the output to temperature, pressure and humidity
389 ****************************************************************/
390 void tph(struct bme680_field_data data)
391 {
392     sprintf(dsp_buff_1,"T:%.2f %cC    ",data.temperature / 100.0f,0xDF);
393     sprintf(dsp_buff_2, "P:%0.2f hPa   ",data.pressure / 100.0f);
394     sprintf(dsp_buff_3, "H:%.2f %%rH   ",data.humidity / 1000.0f);
395     update_lcd_dog();
396 }
397
398
399
400 /************************
401 NAME:       phg()
402 ASSUMES:
403
404 RETURNS:    N/A
405 MODIFIES:   N/A
406 CALLED BY:  N/A
407 DESCRITION: When a pushbutton is pressed, the screen will change
408 the output to pressure, humidity and gas
409 ****************************************************************/
410 void phg(struct bme680_field_data data)
411 {
412     sprintf(dsp_buff_1, "P:%0.2f hPa   ",data.pressure / 100.0f);
413     sprintf(dsp_buff_2,"H:%.2f %%rH    ",data.humidity / 1000.0f);
414     sprintf(dsp_buff_3, "G:%ld ohms    ", data.gas_resistance);
415     update_lcd_dog();
416 }
417
418
419 /************************
420 NAME:        pushbuttonpress()
```

```
421 ASSUMES:
422 RETURNS:     nothing
423 MODIFIES:    N/A
424 CALLED BY:   main()
425 DESCRIPTION: Change the current state and update the screen with
426 current state values.
427 ****************************************************************/
428 void pushbuttonpress(_Bool *point,struct bme680_field_data data)
429 {
430     //if state 0 is selected, switch to state 1
431     if(*point == 0)
432     {
433         *point = 1;
434         phg(data);
435     }
436     //else, if state 1 is selected, switch to state 0
437     else
438     {
439         *point = 0;
440         tph(data);
441     }
442 }
443
```

Final Project

Version 1.0
5/11/2020 10:11:00 AM

Table of Contents

Main Page....2
Module Index....3
Data Structure Index....4
File Index....5
Module Documentation....6
SENSOR API....6
Data Structure Documentation....29
bme680_calib_data....29
bme680_dev....33
bme680_field_data....36
bme680_gas_sett....38
bme680_tph_sett....39
File Documentation....40
C:/Users/Aaron/Desktop/Doxy/bme680.c....40
C:/Users/Aaron/Desktop/Doxy/bme680.h....42
C:/Users/Aaron/Desktop/Doxy/bme680_defs.h....44
C:/Users/Aaron/Desktop/Doxy/console_io_support.c....49
C:/Users/Aaron/Desktop/Doxy/console_io_support.h....51
C:/Users/Aaron/Desktop/Doxy/lcd_dog_driver.c....53
C:/Users/Aaron/Desktop/Doxy/lcd_dog_driver.h....56
C:/Users/Aaron/Desktop/Doxy/main.c....59
C:/Users/Aaron/Desktop/Doxy/SERCOM4_RS232.c....63
C:/Users/Aaron/Desktop/Doxy/SERCOM4_RS232.h....65
Index....66

Main Page

File **bme680.c**

Date

23 Jan 2020

Version

3.5.10

Module Index

Modules

Here is a list of all modules:

SENSOR API 6

Data Structure Index

Data Structures

Here are the data structures with brief descriptions:

bme680_calib_data (Structure to hold the Calibration data) 29

bme680_dev (BME680 device structure) 33

bme680_field_data (Sensor field data structure) 36

bme680_gas_sett (BME680 gas sensor which comprises of gas settings and status parameters) 38

bme680_tph_sett (BME680 sensor settings structure which comprises of ODR, over-sampling and filter settings) 39

File Index

File List

Here is a list of all files with brief descriptions:

C:/Users/Aaron/Desktop/Doxy/bme680.c (Sensor driver for BME680 sensor) 40
C:/Users/Aaron/Desktop/Doxy/bme680.h (Sensor driver for BME680 sensor) 42
C:/Users/Aaron/Desktop/Doxy/bme680_defs.h (Sensor driver for BME680 sensor) 44
C:/Users/Aaron/Desktop/Doxy/console_io_support.c 49
C:/Users/Aaron/Desktop/Doxy/console_io_support.h 51
C:/Users/Aaron/Desktop/Doxy/lcd_dog_driver.c 53
C:/Users/Aaron/Desktop/Doxy/lcd_dog_driver.h 56
C:/Users/Aaron/Desktop/Doxy/main.c 59
C:/Users/Aaron/Desktop/Doxy/SERCOM4_RS232.c 63
C:/Users/Aaron/Desktop/Doxy/SERCOM4_RS232.h 65

Module Documentation

SENSOR API

Data Structures

- struct **bme680_field_data**
  Sensor field data structure.
- struct **bme680_calib_data**
  Structure to hold the Calibration data.
- struct **bme680_tph_sett**
  BME680 sensor settings structure which comprises of ODR, over-sampling and filter settings.
- struct **bme680_gas_sett**
  BME680 gas sensor which comprises of gas settings and status parameters.
- struct **bme680_dev**
  BME680 device structure.

Functions

- int8_t **bme680_init** (struct **bme680_dev** *dev)
  This API is the entry point. It reads the chip-id and calibration data from the sensor.

- int8_t **bme680_set_regs** (const uint8_t *reg_addr, const uint8_t *reg_data, uint8_t len, struct **bme680_dev** *dev)
  This API writes the given data to the 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)
  This API reads the data from the given register address of the sensor.

- int8_t **bme680_soft_reset** (struct **bme680_dev** *dev)
  This API performs the soft reset of the sensor.

- int8_t **bme680_set_sensor_mode** (struct **bme680_dev** *dev)
  This API is used to set the power mode of the sensor.

- int8_t **bme680_get_sensor_mode** (struct **bme680_dev** *dev)
  This API is used to get the power mode of the sensor.

- void **bme680_set_profile_dur** (uint16_t duration, struct **bme680_dev** *dev)
  This API is used to set the profile duration of the sensor.

- void **bme680_get_profile_dur** (uint16_t *duration, const struct **bme680_dev** *dev)
  This API is used to get the profile duration of the sensor.

- int8_t **bme680_get_sensor_data** (struct **bme680_field_data** *data, struct **bme680_dev** *dev)
  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_set_sensor_settings** (uint16_t desired_settings, struct **bme680_dev** *dev)
  This API is used to set 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)
  This API is used to get the oversampling, filter and T,P,H, gas selection settings in the sensor.

Common macros

- #define **INT8_C**(x) S8_C(x)
- #define **UINT8_C**(x) U8_C(x)
- #define **INT16_C**(x) S16_C(x)
- #define **UINT16_C**(x) U16_C(x)
- #define **INT32_C**(x) S32_C(x)
- #define **UINT32_C**(x) U32_C(x)
- #define **INT64_C**(x) S64_C(x)
- #define **UINT64_C**(x) U64_C(x)

C standard macros

- enum **bme680_intf** { **BME680_SPI_INTF**, **BME680_I2C_INTF** }
  Interface selection Enumerations.
- typedef int8_t(* **bme680_com_fptr_t**) (uint8_t dev_id, uint8_t reg_addr, uint8_t *data, uint16_t len)
- typedef void(* **bme680_delay_fptr_t**) (uint32_t period)
- #define **NULL** ((void *) 0)
- #define **BME680_POLL_PERIOD_MS** **UINT8_C**(10)
- #define **BME680_I2C_ADDR_PRIMARY** **UINT8_C**(0x76)
- #define **BME680_I2C_ADDR_SECONDARY** **UINT8_C**(0x77)
- #define **BME680_CHIP_ID** **UINT8_C**(0x61)
- #define **BME680_COEFF_SIZE** **UINT8_C**(41)
- #define **BME680_COEFF_ADDR1_LEN** **UINT8_C**(25)
- #define **BME680_COEFF_ADDR2_LEN** **UINT8_C**(16)
- #define **BME680_FIELD_LENGTH** **UINT8_C**(15)
- #define **BME680_FIELD_ADDR_OFFSET** **UINT8_C**(17)
- #define **BME680_SOFT_RESET_CMD** **UINT8_C**(0xb6)
- #define **BME680_OK** **INT8_C**(0)
- #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)
- #define **BME680_W_DEFINE_PWR_MODE** **INT8_C**(1)
- #define **BME680_W_NO_NEW_DATA** **INT8_C**(2)
- #define **BME680_I_MIN_CORRECTION** **UINT8_C**(1)
- #define **BME680_I_MAX_CORRECTION** **UINT8_C**(2)
- #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)
- #define **BME680_FIELD0_ADDR** **UINT8_C**(0x1d)
- #define **BME680_RES_HEAT0_ADDR** **UINT8_C**(0x5a)
- #define **BME680_GAS_WAIT0_ADDR** **UINT8_C**(0x64)
- #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)
- #define **BME680_COEFF_ADDR1 UINT8_C**(0x89)
- #define **BME680_COEFF_ADDR2 UINT8_C**(0xe1)
- #define **BME680_CHIP_ID_ADDR UINT8_C**(0xd0)
- #define **BME680_SOFT_RESET_ADDR UINT8_C**(0xe0)
- #define **BME680_ENABLE_HEATER UINT8_C**(0x00)
- #define **BME680_DISABLE_HEATER UINT8_C**(0x08)
- #define **BME680_DISABLE_GAS_MEAS UINT8_C**(0x00)
- #define **BME680_ENABLE_GAS_MEAS UINT8_C**(0x01)
- #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)
- #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)
- #define **BME680_SLEEP_MODE UINT8_C**(0)
- #define **BME680_FORCED_MODE UINT8_C**(1)
- #define **BME680_RESET_PERIOD UINT32_C**(10)
- #define **BME680_MEM_PAGE0 UINT8_C**(0x10)
- #define **BME680_MEM_PAGE1 UINT8_C**(0x00)
- #define **BME680_HUM_REG_SHIFT_VAL UINT8_C**(4)
- #define **BME680_RUN_GAS_DISABLE UINT8_C**(0)
- #define **BME680_RUN_GAS_ENABLE UINT8_C**(1)
- #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)
- #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**)
- #define **BME680_NBCONV_MIN UINT8_C**(0)
- #define **BME680_NBCONV_MAX UINT8_C**(10)
- #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)
- #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)
- #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)
- #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)
- #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)
- #define **BME680_MAX_OVERFLOW_VAL INT32_C**(0x40000000)
- #define **BME680_CONCAT_BYTES**(msb, lsb) (((uint16_t)msb << 8) | (uint16_t)lsb)
- #define **BME680_SET_BITS**(reg_data, bitname, data)
- #define **BME680_GET_BITS**(reg_data, bitname)
- #define **BME680_SET_BITS_POS_0**(reg_data, bitname, data)
- #define **BME680_GET_BITS_POS_0**(reg_data, bitname) (reg_data & (bitname##_MSK))

Detailed Description

Macro Definition Documentation

#define BME680_ADDR_GAS_CONF_START UINT8_C(0x64)

Definition at line 144 of file bme680_defs.h.

#define BME680_ADDR_RANGE_SW_ERR_ADDR UINT8_C(0x04)

Definition at line 142 of file bme680_defs.h.

#define BME680_ADDR_RES_HEAT_RANGE_ADDR UINT8_C(0x02)

Definition at line 141 of file bme680_defs.h.

#define BME680_ADDR_RES_HEAT_VAL_ADDR UINT8_C(0x00)

Register map Other coefficient's address

Definition at line 140 of file bme680_defs.h.

#define BME680_ADDR_SENS_CONF_START UINT8_C(0x5A)

Definition at line 143 of file bme680_defs.h.

#define BME680_BIT_H1_DATA_MSK UINT8_C(0x0F)

Definition at line 256 of file bme680_defs.h.

#define BME680_CHIP_ID UINT8_C(0x61)

BME680 unique chip identifier

Definition at line 108 of file bme680_defs.h.

#define BME680_CHIP_ID_ADDR UINT8_C(0xd0)

Chip identifier

Definition at line 166 of file bme680_defs.h.

#define BME680_COEFF_ADDR1 UINT8_C(0x89)

Coefficient's address

Definition at line 162 of file bme680_defs.h.

#define BME680_COEFF_ADDR1_LEN UINT8_C(25)

Definition at line 112 of file bme680_defs.h.

#define BME680_COEFF_ADDR2 UINT8_C(0xe1)

Definition at line 163 of file bme680_defs.h.

#define BME680_COEFF_ADDR2_LEN UINT8_C(16)

Definition at line 113 of file bme680_defs.h.

#define BME680_COEFF_SIZE UINT8_C(41)

BME680 coefficients related defines

Definition at line 111 of file bme680_defs.h.

#define BME680_CONCAT_BYTES(msb, lsb) (((uint16_t)msb << 8) | (uint16_t)lsb)

Macro to combine two 8 bit data's to form a 16 bit data

Definition at line 319 of file bme680_defs.h.

#define BME680_CONF_HEAT_CTRL_ADDR UINT8_C(0x70)

Sensor configuration registers

Definition at line 154 of file bme680_defs.h.

#define BME680_CONF_ODR_FILT_ADDR UINT8_C(0x75)

Definition at line 159 of file bme680_defs.h.

#define BME680_CONF_ODR_RUN_GAS_NBC_ADDR UINT8_C(0x71)

Definition at line 155 of file bme680_defs.h.

#define BME680_CONF_OS_H_ADDR UINT8_C(0x72)

Definition at line 156 of file bme680_defs.h.

#define BME680_CONF_T_P_MODE_ADDR UINT8_C(0x74)

Definition at line 158 of file bme680_defs.h.

#define BME680_DISABLE_GAS_MEAS UINT8_C(0x00)

Gas measurement settings

Definition at line 176 of file bme680_defs.h.

#define BME680_DISABLE_HEATER UINT8_C(0x08)

Definition at line 173 of file bme680_defs.h.

#define BME680_E_COM_FAIL INT8_C(-2)

Definition at line 126 of file bme680_defs.h.

#define BME680_E_DEV_NOT_FOUND INT8_C(-3)

Definition at line 127 of file bme680_defs.h.

#define BME680_E_INVALID_LENGTH INT8_C(-4)

Definition at line 128 of file bme680_defs.h.

#define BME680_E_NULL_PTR INT8_C(-1)

Definition at line 125 of file bme680_defs.h.

#define BME680_ENABLE_GAS_MEAS UINT8_C(0x01)

Definition at line 177 of file bme680_defs.h.

#define BME680_ENABLE_HEATER UINT8_C(0x00)

Heater control settings

Definition at line 172 of file bme680_defs.h.

#define BME680_FIELD0_ADDR UINT8_C(0x1d)

Field settings

Definition at line 147 of file bme680_defs.h.

#define BME680_FIELD_ADDR_OFFSET UINT8_C(17)

Definition at line 117 of file bme680_defs.h.

#define BME680_FIELD_DATA_LENGTH UINT8_C(3)

Definition at line 218 of file bme680_defs.h.

#define BME680_FIELD_LENGTH UINT8_C(15)

BME680 field_x related defines

Definition at line 116 of file bme680_defs.h.

#define BME680_FILTER_MSK UINT8_C(0X1C)

Definition at line 239 of file bme680_defs.h.

#define BME680_FILTER_POS UINT8_C(2)

Definition at line 260 of file bme680_defs.h.

#define BME680_FILTER_SEL UINT16_C(16)

Definition at line 226 of file bme680_defs.h.

#define BME680_FILTER_SIZE_0 UINT8_C(0)

IIR filter settings

Definition at line 188 of file bme680_defs.h.

#define BME680_FILTER_SIZE_1 UINT8_C(1)

Definition at line 189 of file bme680_defs.h.

#define BME680_FILTER_SIZE_127 UINT8_C(7)

Definition at line 195 of file bme680_defs.h.

#define BME680_FILTER_SIZE_15 UINT8_C(4)

Definition at line 192 of file bme680_defs.h.

#define BME680_FILTER_SIZE_3 UINT8_C(2)

Definition at line 190 of file bme680_defs.h.

#define BME680_FILTER_SIZE_31 UINT8_C(5)

Definition at line 193 of file bme680_defs.h.

#define BME680_FILTER_SIZE_63 UINT8_C(6)

Definition at line 194 of file bme680_defs.h.

#define BME680_FILTER_SIZE_7 UINT8_C(3)

Definition at line 191 of file bme680_defs.h.

#define BME680_FORCED_MODE UINT8_C(1)

Definition at line 199 of file bme680_defs.h.

#define BME680_GAS_INDEX_MSK UINT8_C(0x0f)

Definition at line 249 of file bme680_defs.h.

#define BME680_GAS_MEAS_MSK UINT8_C(0x30)

Mask definitions

Definition at line 237 of file bme680_defs.h.

#define BME680_GAS_MEAS_POS UINT8_C(4)

Bit position definitions for sensor settings

Definition at line 259 of file bme680_defs.h.

#define BME680_GAS_MEAS_SEL UINT16_C(8)

Definition at line 225 of file bme680_defs.h.

#define BME680_GAS_RANGE_MSK UINT8_C(0x0f)

Definition at line 250 of file bme680_defs.h.

#define BME680_GAS_REG_BUF_LENGTH UINT8_C(20)

Definition at line 219 of file bme680_defs.h.

#define BME680_GAS_SENSOR_SEL (BME680_GAS_MEAS_SEL | BME680_RUN_GAS_SEL | BME680_NBCONV_SEL)

Definition at line 230 of file bme680_defs.h.

#define BME680_GAS_WAIT0_ADDR UINT8_C(0x64)

Definition at line 151 of file bme680_defs.h.

#define BME680_GASM_VALID_MSK UINT8_C(0x20)

Definition at line 251 of file bme680_defs.h.

#define BME680_GET_BITS(reg_data, bitname)

```
Value:    ((reg_data & (bitname##_MSK)) >> \
    (bitname##_POS))
```

Definition at line 325 of file bme680_defs.h.

#define BME680_GET_BITS_POS_0(reg_data, bitname) (reg_data & (bitname##_MSK))

Definition at line 332 of file bme680_defs.h.

#define BME680_GH1_REG (37)

Definition at line 298 of file bme680_defs.h.

#define BME680_GH2_LSB_REG (35)

Definition at line 296 of file bme680_defs.h.

#define BME680_GH2_MSB_REG (36)

Definition at line 297 of file bme680_defs.h.

#define BME680_GH3_REG (38)

Definition at line 299 of file bme680_defs.h.

#define BME680_H1_LSB_REG (26)

Definition at line 287 of file bme680_defs.h.

#define BME680_H1_MSB_REG (27)

Definition at line 288 of file bme680_defs.h.

#define BME680_H2_LSB_REG (26)

Definition at line 286 of file bme680_defs.h.

#define BME680_H2_MSB_REG (25)

Definition at line 285 of file bme680_defs.h.

#define BME680_H3_REG (28)

Definition at line 289 of file bme680_defs.h.

#define BME680_H4_REG (29)

Definition at line 290 of file bme680_defs.h.

#define BME680_H5_REG (30)

Definition at line 291 of file bme680_defs.h.

#define BME680_H6_REG (31)

Definition at line 292 of file bme680_defs.h.

#define BME680_H7_REG (32)

Definition at line 293 of file bme680_defs.h.

#define BME680_HCNTRL_SEL UINT16_C(32)

Definition at line 227 of file bme680_defs.h.

#define BME680_HCTRL_MSK UINT8_C(0x08)

Definition at line 243 of file bme680_defs.h.

#define BME680_HEAT_STAB_MSK UINT8_C(0x10)

Definition at line 252 of file bme680_defs.h.

#define BME680_HUM_REG_SHIFT_VAL UINT8_C(4)

Ambient humidity shift value for compensation

Definition at line 209 of file bme680_defs.h.

#define BME680_I2C_ADDR_PRIMARY UINT8_C(0x76)

BME680 I2C addresses

Definition at line 104 of file bme680_defs.h.

#define BME680_I2C_ADDR_SECONDARY UINT8_C(0x77)

Definition at line 105 of file bme680_defs.h.

#define BME680_I_MAX_CORRECTION UINT8_C(2)

Definition at line 136 of file bme680_defs.h.

#define BME680_I_MIN_CORRECTION UINT8_C(1)

Definition at line 135 of file bme680_defs.h.

#define BME680_MAX_OVERFLOW_VAL INT32_C(0x40000000)

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)**

Definition at line 316 of file bme680_defs.h.

#define BME680_MEM_PAGE0 UINT8_C(0x10)

SPI memory page settings

Definition at line 205 of file bme680_defs.h.

#define BME680_MEM_PAGE1 UINT8_C(0x00)

Definition at line 206 of file bme680_defs.h.

#define BME680_MEM_PAGE_ADDR UINT8_C(0xf3)

Definition at line 157 of file bme680_defs.h.

#define BME680_MEM_PAGE_MSK UINT8_C(0x10)

Definition at line 253 of file bme680_defs.h.

#define BME680_MODE_MSK UINT8_C(0x03)

Definition at line 245 of file bme680_defs.h.

#define BME680_NBCONV_MAX UINT8_C(10)

Definition at line 234 of file bme680_defs.h.

#define BME680_NBCONV_MIN UINT8_C(0)

Number of conversion settings

Definition at line 233 of file bme680_defs.h.

#define BME680_NBCONV_MSK UINT8_C(0X0F)

Definition at line 238 of file bme680_defs.h.

#define BME680_NBCONV_SEL UINT16_C(128)

Definition at line 229 of file bme680_defs.h.

#define BME680_NEW_DATA_MSK UINT8_C(0x80)

Definition at line 248 of file bme680_defs.h.

#define BME680_OK INT8_C(0)

Error code definitions

Definition at line 123 of file bme680_defs.h.

#define BME680_OS_16X UINT8_C(5)

Definition at line 185 of file bme680_defs.h.

#define BME680_OS_1X UINT8_C(1)

Definition at line 181 of file bme680_defs.h.

#define BME680_OS_2X UINT8_C(2)

Definition at line 182 of file bme680_defs.h.

#define BME680_OS_4X UINT8_C(3)

Definition at line 183 of file bme680_defs.h.

#define BME680_OS_8X UINT8_C(4)

Definition at line 184 of file bme680_defs.h.

#define BME680_OS_NONE UINT8_C(0)

Over-sampling settings

Definition at line 180 of file bme680_defs.h.

#define BME680_OSH_MSK UINT8_C(0X07)

Definition at line 242 of file bme680_defs.h.

#define BME680_OSH_SEL UINT16_C(4)

Definition at line 224 of file bme680_defs.h.

#define BME680_OSP_MSK UINT8_C(0X1C)

Definition at line 241 of file bme680_defs.h.

#define BME680_OSP_POS UINT8_C(2)

Definition at line 262 of file bme680_defs.h.

#define BME680_OSP_SEL UINT16_C(2)

Definition at line 223 of file bme680_defs.h.

#define BME680_OST_MSK UINT8_C(0XE0)

Definition at line 240 of file bme680_defs.h.

#define BME680_OST_POS UINT8_C(5)

Definition at line 261 of file bme680_defs.h.

#define BME680_OST_SEL UINT16_C(1)

Settings selector

Definition at line 222 of file bme680_defs.h.

#define BME680_P10_REG (23)

Definition at line 284 of file bme680_defs.h.

#define BME680_P1_LSB_REG (5)

Definition at line 269 of file bme680_defs.h.

#define BME680_P1_MSB_REG (6)

Definition at line 270 of file bme680_defs.h.

#define BME680_P2_LSB_REG (7)

Definition at line 271 of file bme680_defs.h.

#define BME680_P2_MSB_REG (8)

Definition at line 272 of file bme680_defs.h.

#define BME680_P3_REG (9)

Definition at line 273 of file bme680_defs.h.

#define BME680_P4_LSB_REG (11)

Definition at line 274 of file bme680_defs.h.

#define BME680_P4_MSB_REG (12)

Definition at line 275 of file bme680_defs.h.

#define BME680_P5_LSB_REG (13)

Definition at line 276 of file bme680_defs.h.

#define BME680_P5_MSB_REG (14)

Definition at line 277 of file bme680_defs.h.

#define BME680_P6_REG (16)

Definition at line 279 of file bme680_defs.h.

#define BME680_P7_REG (15)

Definition at line 278 of file bme680_defs.h.

#define BME680_P8_LSB_REG (19)

Definition at line 280 of file bme680_defs.h.

#define BME680_P8_MSB_REG (20)

Definition at line 281 of file bme680_defs.h.

#define BME680_P9_LSB_REG (21)

Definition at line 282 of file bme680_defs.h.

#define BME680_P9_MSB_REG (22)

Definition at line 283 of file bme680_defs.h.

#define BME680_POLL_PERIOD_MS UINT8_C(10)

BME680 configuration macros Enable or un-comment the macro to provide floating point data output BME680 General config

Definition at line 101 of file bme680_defs.h.

#define BME680_REG_BUFFER_LENGTH UINT8_C(6)

Definition at line 217 of file bme680_defs.h.

#define BME680_REG_FILTER_INDEX UINT8_C(5)

BME680 register buffer index settings

Definition at line 302 of file bme680_defs.h.

#define BME680_REG_HCTRL_INDEX UINT8_C(0)

Definition at line 308 of file bme680_defs.h.

#define BME680_REG_HUM_INDEX UINT8_C(2)

Definition at line 305 of file bme680_defs.h.

#define BME680_REG_NBCONV_INDEX UINT8_C(1)

Definition at line 306 of file bme680_defs.h.

#define BME680_REG_PRES_INDEX UINT8_C(4)

Definition at line 304 of file bme680_defs.h.

#define BME680_REG_RUN_GAS_INDEX UINT8_C(1)

Definition at line 307 of file bme680_defs.h.

#define BME680_REG_TEMP_INDEX UINT8_C(4)

Definition at line 303 of file bme680_defs.h.

#define BME680_RES_HEAT0_ADDR UINT8_C(0x5a)

Heater settings

Definition at line 150 of file bme680_defs.h.

#define BME680_RESET_PERIOD UINT32_C(10)

Delay related macro declaration

Definition at line 202 of file bme680_defs.h.

#define BME680_RHRANGE_MSK UINT8_C(0x30)

Definition at line 246 of file bme680_defs.h.

#define BME680_RSERROR_MSK UINT8_C(0xf0)

Definition at line 247 of file bme680_defs.h.

#define BME680_RUN_GAS_DISABLE UINT8_C(0)

Run gas enable and disable settings

Definition at line 212 of file bme680_defs.h.

#define BME680_RUN_GAS_ENABLE UINT8_C(1)

Definition at line 213 of file bme680_defs.h.

#define BME680_RUN_GAS_MSK UINT8_C(0x10)

Definition at line 244 of file bme680_defs.h.

#define BME680_RUN_GAS_POS UINT8_C(4)

Definition at line 263 of file bme680_defs.h.

#define BME680_RUN_GAS_SEL UINT16_C(64)

Definition at line 228 of file bme680_defs.h.

#define BME680_SET_BITS(reg_data, bitname, data)

```
Value:          ((reg_data & ~(bitname##_MSK)) | \
        ((data << bitname##_POS) & bitname##_MSK))
```

Macro to SET and GET BITS of a register

Definition at line 322 of file bme680_defs.h.

#define BME680_SET_BITS_POS_0(reg_data, bitname, data)

```
Value:                  ((reg_data & ~(bitname##_MSK)) | \
                (data & bitname##_MSK))
```

Macro variant to handle the bitname position if it is zero

Definition at line 329 of file bme680_defs.h.

#define BME680_SLEEP_MODE UINT8_C(0)

Power mode settings

Definition at line 198 of file bme680_defs.h.

#define BME680_SOFT_RESET_ADDR UINT8_C(0xe0)

Soft reset register

Definition at line 169 of file bme680_defs.h.

#define BME680_SOFT_RESET_CMD UINT8_C(0xb6)

Soft reset command

Definition at line 120 of file bme680_defs.h.

#define BME680_SPI_RD_MSK UINT8_C(0x80)

Definition at line 254 of file bme680_defs.h.

#define BME680_SPI_WR_MSK UINT8_C(0x7f)

Definition at line 255 of file bme680_defs.h.

#define BME680_T1_LSB_REG (33)

Definition at line 294 of file bme680_defs.h.

#define BME680_T1_MSB_REG (34)

Definition at line 295 of file bme680_defs.h.

#define BME680_T2_LSB_REG (1)

Array Index to Field data mapping for Calibration Data

Definition at line 266 of file bme680_defs.h.

#define BME680_T2_MSB_REG (2)

Definition at line 267 of file bme680_defs.h.

#define BME680_T3_REG (3)

Definition at line 268 of file bme680_defs.h.

#define BME680_TMP_BUFFER_LENGTH UINT8_C(40)

Buffer length macro declaration

Definition at line 216 of file bme680_defs.h.

#define BME680_W_DEFINE_PWR_MODE INT8_C(1)

Definition at line 131 of file bme680_defs.h.

#define BME680_W_NO_NEW_DATA INT8_C(2)

Definition at line 132 of file bme680_defs.h.

#define INT16_C(x) S16_C(x)

Definition at line 69 of file bme680_defs.h.

#define INT32_C(x) S32_C(x)

Definition at line 74 of file bme680_defs.h.

#define INT64_C(x) S64_C(x)

Definition at line 79 of file bme680_defs.h.

#define INT8_C(x) S8_C(x)

Definition at line 64 of file bme680_defs.h.

#define NULL ((void *) 0)

Definition at line 90 of file bme680_defs.h.

#define UINT16_C(x) U16_C(x)

Definition at line 70 of file bme680_defs.h.

#define UINT32_C(x) U32_C(x)

Definition at line 75 of file bme680_defs.h.

#define UINT64_C(x) U64_C(x)

Definition at line 80 of file bme680_defs.h.

#define UINT8_C(x) U8_C(x)

Definition at line 65 of file bme680_defs.h.

Typedef Documentation

typedef int8_t(* bme680_com_fptr_t) (uint8_t dev_id, uint8_t reg_addr, uint8_t *data, uint16_t len)

Type definitions

Generic communication function pointer

Parameters

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.
in	*reg_addr*	Used to select the register the where data needs to be read from or written to.
	[in/out]	reg_data: Data array to read/write
in	*len*	Length of the data array

Definition at line 345 of file bme680_defs.h.

typedef void(* bme680_delay_fptr_t) (uint32_t period)

Delay function pointer

Parameters

in	*period*	Time period in milliseconds

Definition at line 351 of file bme680_defs.h.

Enumeration Type Documentation

enum bme680_intf

Interface selection Enumerations.

Enumerator:

BME680_SPI_INTF	SPI interface
BME680_I2C_INTF	I2C interface

Definition at line 356 of file bme680_defs.h.

Function Documentation

void bme680_get_profile_dur (uint16_t * *duration*, const struct bme680_dev * *dev*)

This API is used to get the profile duration of the sensor.

Parameters

in	*dev*	: Structure instance of **bme680_dev**.
in	*duration*	: Duration of the measurement in ms.

Returns

Nothing

Definition at line 663 of file bme680.c.

Here is the caller graph for this function:

int8_t bme680_get_regs (uint8_t *reg_addr*, uint8_t * *reg_data*, uint16_t *len*, struct bme680_dev * *dev*)

This API reads the data from the given register address of the sensor.

Parameters

in	*reg_addr*	: Register address from where the data to be read
out	*reg_data*	: Pointer to data buffer to store the read data.
in	*len*	: No of bytes of data to be read.
in	*dev*	: Structure instance of **bme680_dev**.

Returns

Result of API execution status

Return values

zero	-> Success / +ve value -> Warning / -ve value -> Error

Definition at line 306 of file bme680.c.

int8_t bme680_get_sensor_data (struct bme680_field_data * *data*, struct bme680_dev * *dev*)

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.

Parameters

out	*data*	Structure instance to hold the data.
in	*dev*	: Structure instance of **bme680_dev**.

Returns

Result of API execution status

Return values

zero	-> Success / +ve value -> Warning / -ve value -> Error

Definition at line 696 of file bme680.c.

Here is the caller graph for this function:

int8_t bme680_get_sensor_mode (struct bme680_dev * *dev*)

This API is used to get the power mode of the sensor.

Parameters

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

Returns

Result of API execution status

Return values

zero	-> Success / +ve value -> Warning / -ve value -> Error

Definition at line 619 of file bme680.c.

int8_t bme680_get_sensor_settings (uint16_t *desired_settings*, struct bme680_dev * *dev*)

This API is used to get the oversampling, filter and T,P,H, gas selection settings in the sensor.

Parameters

in	*dev*	: Structure instance of **bme680_dev**.
in	*desired_settings*	: Variable used to select the settings which are to be get from the sensor.

Returns

Result of API execution status

Return values

zero	-> Success / +ve value -> Warning / -ve value -> Error.

Definition at line 528 of file bme680.c.

int8_t bme680_init (struct bme680_dev * *dev*)

This API is the entry point. It reads the chip-id and calibration data from the sensor.

CPP guard

Parameters

in,out	*dev*	: Structure instance of **bme680_dev**

Returns

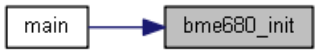
Result of API execution status

Return values

zero	-> Success / +ve value -> Warning / -ve value -> Error

Definition at line 278 of file bme680.c.

Here is the caller graph for this function:



void bme680_set_profile_dur (uint16_t *duration*, struct bme680_dev * *dev*)

This API is used to set the profile duration of the sensor.

Parameters

in	*dev*	: Structure instance of **bme680_dev**.
in	*duration*	: Duration of the measurement in ms.

Returns

Nothing

Definition at line 638 of file bme680.c.

int8_t bme680_set_regs (const uint8_t * *reg_addr*, const uint8_t * *reg_data*, uint8_t *len*, struct bme680_dev * *dev*)

This API writes the given data to the register address of the sensor.

Parameters

in	*reg_addr*	: Register address from where the data to be written.
in	*reg_data*	: Pointer to data buffer which is to be written in the sensor.
in	*len*	: No of bytes of data to write..
in	*dev*	: Structure instance of **bme680_dev**.

Returns

Result of API execution status

Return values

zero	-> Success / +ve value -> Warning / -ve value -> Error

Definition at line 331 of file bme680.c.

int8_t bme680_set_sensor_mode (struct bme680_dev * *dev*)

This API is used to set the power mode of the sensor.

Parameters

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

-
- **Returns**

  Result of API execution status

- **Return values**

zero	-> Success / +ve value -> Warning / -ve value -> Error

Definition at line 580 of file bme680.c.

Here is the caller graph for this function:

int8_t bme680_set_sensor_settings (uint16_t *desired_settings*, struct bme680_dev * *dev*)

This API is used to set the oversampling, filter and T,P,H, gas selection settings in the sensor.

Parameters

in	*dev*	: Structure instance of **bme680_dev**.
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.

Returns

Result of API execution status

Return values

zero	-> Success / +ve value -> Warning / -ve value -> Error.

Definition at line 404 of file bme680.c.

Here is the caller graph for this function:

int8_t bme680_soft_reset (struct bme680_dev * *dev*)

This API performs the soft reset of the sensor.

Parameters

in	*dev*	: Structure instance of **bme680_dev**.

Returns

Result of API execution status

Return values

zero	-> Success / +ve value -> Warning / -ve value -> Error.

Definition at line 370 of file bme680.c.

Data Structure Documentation

bme680_calib_data Struct Reference

Structure to hold the Calibration data.
`#include <bme680_defs.h>`

Data Fields

- uint16_t **par_h1**
- uint16_t **par_h2**
- int8_t **par_h3**
- int8_t **par_h4**
- int8_t **par_h5**
- uint8_t **par_h6**
- int8_t **par_h7**
- int8_t **par_gh1**
- int16_t **par_gh2**
- int8_t **par_gh3**
- uint16_t **par_t1**
- int16_t **par_t2**
- int8_t **par_t3**
- uint16_t **par_p1**
- int16_t **par_p2**
- int8_t **par_p3**
- int16_t **par_p4**
- int16_t **par_p5**
- int8_t **par_p6**
- int8_t **par_p7**
- int16_t **par_p8**
- int16_t **par_p9**
- uint8_t **par_p10**
- int32_t **t_fine**
- uint8_t **res_heat_range**
- int8_t **res_heat_val**
- int8_t **range_sw_err**

Detailed Description

Structure to hold the Calibration data.

Definition at line 401 of file bme680_defs.h.

Field Documentation

int8_t par_gh1

Variable to store calibrated gas data

Definition at line 417 of file bme680_defs.h.

int16_t par_gh2

Variable to store calibrated gas data

Definition at line 419 of file bme680_defs.h.

int8_t par_gh3

Variable to store calibrated gas data

Definition at line 421 of file bme680_defs.h.

uint16_t par_h1

Variable to store calibrated humidity data

Definition at line 403 of file bme680_defs.h.

uint16_t par_h2

Variable to store calibrated humidity data

Definition at line 405 of file bme680_defs.h.

int8_t par_h3

Variable to store calibrated humidity data

Definition at line 407 of file bme680_defs.h.

int8_t par_h4

Variable to store calibrated humidity data

Definition at line 409 of file bme680_defs.h.

int8_t par_h5

Variable to store calibrated humidity data

Definition at line 411 of file bme680_defs.h.

uint8_t par_h6

Variable to store calibrated humidity data

Definition at line 413 of file bme680_defs.h.

int8_t par_h7

Variable to store calibrated humidity data

Definition at line 415 of file bme680_defs.h.

uint16_t par_p1

Variable to store calibrated pressure data

Definition at line 429 of file bme680_defs.h.

uint8_t par_p10

Variable to store calibrated pressure data

Definition at line 447 of file bme680_defs.h.

int16_t par_p2

Variable to store calibrated pressure data

Definition at line 431 of file bme680_defs.h.

int8_t par_p3

Variable to store calibrated pressure data

Definition at line 433 of file bme680_defs.h.

int16_t par_p4

Variable to store calibrated pressure data

Definition at line 435 of file bme680_defs.h.

int16_t par_p5

Variable to store calibrated pressure data

Definition at line 437 of file bme680_defs.h.

int8_t par_p6

Variable to store calibrated pressure data

Definition at line 439 of file bme680_defs.h.

int8_t par_p7

Variable to store calibrated pressure data

Definition at line 441 of file bme680_defs.h.

int16_t par_p8

Variable to store calibrated pressure data

Definition at line 443 of file bme680_defs.h.

int16_t par_p9

Variable to store calibrated pressure data

Definition at line 445 of file bme680_defs.h.

uint16_t par_t1

Variable to store calibrated temperature data

Definition at line 423 of file bme680_defs.h.

int16_t par_t2

Variable to store calibrated temperature data

Definition at line 425 of file bme680_defs.h.

int8_t par_t3

Variable to store calibrated temperature data

Definition at line 427 of file bme680_defs.h.

int8_t range_sw_err

Variable to store error range

Definition at line 461 of file bme680_defs.h.

uint8_t res_heat_range

Variable to store heater resistance range

Definition at line 457 of file bme680_defs.h.

int8_t res_heat_val

Variable to store heater resistance value

Definition at line 459 of file bme680_defs.h.

int32_t t_fine

Variable to store t_fine size

Definition at line 451 of file bme680_defs.h.

The documentation for this struct was generated from the following file:

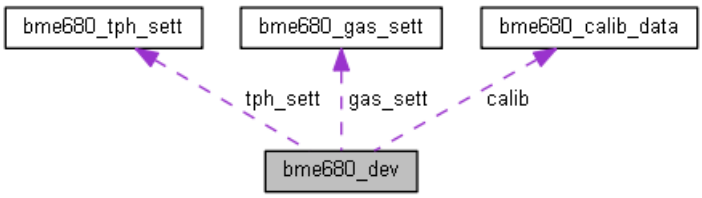
- C:/Users/Aaron/Desktop/Doxy/**bme680_defs.h**

bme680_dev Struct Reference

BME680 device structure.

`#include <bme680_defs.h>`

Collaboration diagram for bme680_dev:



Data Fields

- uint8_t **chip_id**
- uint8_t **dev_id**
- enum **bme680_intf intf**
- uint8_t **mem_page**
- int8_t **amb_temp**
- struct **bme680_calib_data calib**
- struct **bme680_tph_sett tph_sett**
- struct **bme680_gas_sett gas_sett**
- uint8_t **power_mode**
- uint8_t **new_fields**
- uint8_t **info_msg**
- **bme680_com_fptr_t read**
- **bme680_com_fptr_t write**
- **bme680_delay_fptr_t delay_ms**
- int8_t **com_rslt**

Detailed Description

BME680 device structure.

Definition at line 499 of file bme680_defs.h.

Field Documentation

int8_t amb_temp

Ambient temperature in Degree C

Definition at line 509 of file bme680_defs.h.

struct bme680_calib_data calib

Sensor calibration data

Definition at line 511 of file bme680_defs.h.

uint8_t chip_id

Chip Id

Definition at line 501 of file bme680_defs.h.

int8_t com_rslt

Communication function result

Definition at line 529 of file bme680_defs.h.

bme680_delay_fptr_t delay_ms

delay function pointer

Definition at line 527 of file bme680_defs.h.

uint8_t dev_id

Device Id

Definition at line 503 of file bme680_defs.h.

struct bme680_gas_sett gas_sett

Gas Sensor settings

Definition at line 515 of file bme680_defs.h.

uint8_t info_msg

Store the info messages

Definition at line 521 of file bme680_defs.h.

enum bme680_intf intf

SPI/I2C interface

Definition at line 505 of file bme680_defs.h.

uint8_t mem_page

Memory page used

Definition at line 507 of file bme680_defs.h.

uint8_t new_fields

New sensor fields

Definition at line 519 of file bme680_defs.h.

uint8_t power_mode

Sensor power modes

Definition at line 517 of file bme680_defs.h.

bme680_com_fptr_t read

Bus read function pointer

Definition at line 523 of file bme680_defs.h.

struct bme680_tph_sett tph_sett

Sensor settings

Definition at line 513 of file bme680_defs.h.

bme680_com_fptr_t write

Bus write function pointer

Definition at line 525 of file bme680_defs.h.

The documentation for this struct was generated from the following file:

- C:/Users/Aaron/Desktop/Doxy/**bme680_defs.h**

bme680_field_data Struct Reference

Sensor field data structure.

`#include <bme680_defs.h>`

Data Fields

- uint8_t **status**
- uint8_t **gas_index**
- uint8_t **meas_index**
- int16_t **temperature**
- uint32_t **pressure**
- uint32_t **humidity**
- uint32_t **gas_resistance**

Detailed Description

Sensor field data structure.

Definition at line 367 of file bme680_defs.h.

Field Documentation

uint8_t gas_index

The index of the heater profile used

Definition at line 371 of file bme680_defs.h.

uint32_t gas_resistance

Gas resistance in Ohms

Definition at line 383 of file bme680_defs.h.

uint32_t humidity

Humidity in % relative humidity x1000

Definition at line 381 of file bme680_defs.h.

uint8_t meas_index

Measurement index to track order

Definition at line 373 of file bme680_defs.h.

uint32_t pressure

Pressure in Pascal

Definition at line 379 of file bme680_defs.h.

uint8_t status

Contains new_data, gasm_valid & heat_stab

Definition at line 369 of file bme680_defs.h.

int16_t temperature

Temperature in degree celsius x100

Definition at line 377 of file bme680_defs.h.

The documentation for this struct was generated from the following file:

- C:/Users/Aaron/Desktop/Doxy/**bme680_defs.h**

bme680_gas_sett Struct Reference

BME680 gas sensor which comprises of gas settings and status parameters.
`#include <bme680_defs.h>`

Data Fields

- uint8_t **nb_conv**
- uint8_t **heatr_ctrl**
- uint8_t **run_gas**
- uint16_t **heatr_temp**
- uint16_t **heatr_dur**

Detailed Description

BME680 gas sensor which comprises of gas settings and status parameters.

Definition at line 483 of file bme680_defs.h.

Field Documentation

uint8_t heatr_ctrl

Variable to store heater control

Definition at line 487 of file bme680_defs.h.

uint16_t heatr_dur

Duration profile value

Definition at line 493 of file bme680_defs.h.

uint16_t heatr_temp

Heater temperature value

Definition at line 491 of file bme680_defs.h.

uint8_t nb_conv

Variable to store nb conversion

Definition at line 485 of file bme680_defs.h.

uint8_t run_gas

Run gas enable value

Definition at line 489 of file bme680_defs.h.

The documentation for this struct was generated from the following file:

- C:/Users/Aaron/Desktop/Doxy/**bme680_defs.h**

bme680_tph_sett Struct Reference

BME680 sensor settings structure which comprises of ODR, over-sampling and filter settings.
`#include <bme680_defs.h>`

Data Fields

- uint8_t **os_hum**
- uint8_t **os_temp**
- uint8_t **os_pres**
- uint8_t **filter**

Detailed Description

BME680 sensor settings structure which comprises of ODR, over-sampling and filter settings.

Definition at line 468 of file bme680_defs.h.

Field Documentation

uint8_t filter

Filter coefficient

Definition at line 476 of file bme680_defs.h.

uint8_t os_hum

Humidity oversampling

Definition at line 470 of file bme680_defs.h.

uint8_t os_pres

Pressure oversampling

Definition at line 474 of file bme680_defs.h.

uint8_t os_temp

Temperature oversampling

Definition at line 472 of file bme680_defs.h.

The documentation for this struct was generated from the following file:

- C:/Users/Aaron/Desktop/Doxy/**bme680_defs.h**

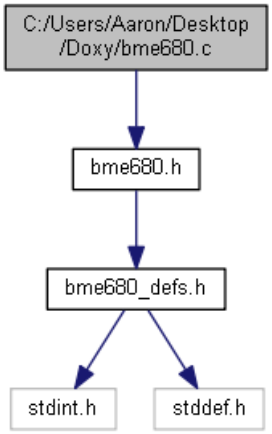
File Documentation

C:/Users/Aaron/Desktop/Doxy/bme680.c File Reference

Sensor driver for BME680 sensor.

```
#include "bme680.h"
```

Include dependency graph for bme680.c:



Functions

- int8_t **bme680_init** (struct **bme680_dev** *dev)
  This API is the entry point. It reads the chip-id and calibration data from the sensor.

- int8_t **bme680_get_regs** (uint8_t reg_addr, uint8_t *reg_data, uint16_t len, struct **bme680_dev** *dev)
  This API reads the data from the given 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)
  This API writes the given data to the register address of the sensor.

- int8_t **bme680_soft_reset** (struct **bme680_dev** *dev)
  This API performs the soft reset of the sensor.

- int8_t **bme680_set_sensor_settings** (uint16_t desired_settings, struct **bme680_dev** *dev)
  This API is used to set 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)
  This API is used to get the oversampling, filter and T,P,H, gas selection settings in the sensor.

- int8_t **bme680_set_sensor_mode** (struct **bme680_dev** *dev)
  This API is used to set the power mode of the sensor.

- int8_t **bme680_get_sensor_mode** (struct **bme680_dev** *dev)
  This API is used to get the power mode of the sensor.

- void **bme680_set_profile_dur** (uint16_t duration, struct **bme680_dev** *dev)
  This API is used to set the profile duration of the sensor.

- void **bme680_get_profile_dur** (uint16_t *duration, const struct **bme680_dev** *dev)
  This API is used to get the profile duration of the sensor.

- int8_t **bme680_get_sensor_data** (struct **bme680_field_data** *data, struct **bme680_dev** *dev)
  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.

Detailed Description

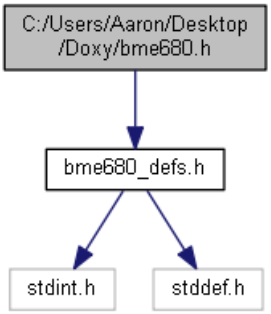
Sensor driver for BME680 sensor.

C:/Users/Aaron/Desktop/Doxy/bme680.h File Reference

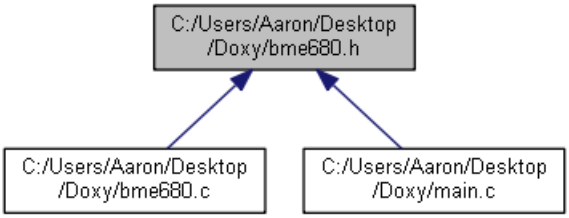
Sensor driver for BME680 sensor.

```
#include "bme680_defs.h"
```

Include dependency graph for bme680.h:



This graph shows which files directly or indirectly include this file:



Functions

- int8_t **bme680_init** (struct **bme680_dev** *dev)
  This API is the entry point. It reads the chip-id and calibration data from the sensor.

- int8_t **bme680_set_regs** (const uint8_t *reg_addr, const uint8_t *reg_data, uint8_t len, struct **bme680_dev** *dev)
  This API writes the given data to the 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)
  This API reads the data from the given register address of the sensor.

- int8_t **bme680_soft_reset** (struct **bme680_dev** *dev)
  This API performs the soft reset of the sensor.

- int8_t **bme680_set_sensor_mode** (struct **bme680_dev** *dev)
  This API is used to set the power mode of the sensor.

- int8_t **bme680_get_sensor_mode** (struct **bme680_dev** *dev)
  This API is used to get the power mode of the sensor.

- void **bme680_set_profile_dur** (uint16_t duration, struct **bme680_dev** *dev)
  This API is used to set the profile duration of the sensor.

- void **bme680_get_profile_dur** (uint16_t *duration, const struct **bme680_dev** *dev)
  This API is used to get the profile duration of the sensor.

- int8_t **bme680_get_sensor_data** (struct **bme680_field_data** *data, struct **bme680_dev** *dev)
  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_set_sensor_settings** (uint16_t desired_settings, struct **bme680_dev** *dev)
  This API is used to set 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)
  This API is used to get the oversampling, filter and T,P,H, gas selection settings in the sensor.

Detailed Description

Sensor driver for BME680 sensor.

Date

23 Jan 2020

Version

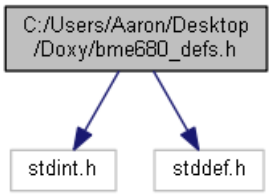
3.5.10

C:/Users/Aaron/Desktop/Doxy/bme680_defs.h File Reference

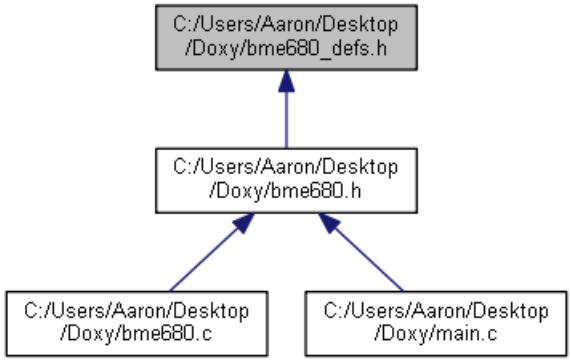
Sensor driver for BME680 sensor.

```
#include <stdint.h>
#include <stddef.h>
```

Include dependency graph for bme680_defs.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct **bme680_field_data**
  Sensor field data structure.
- struct **bme680_calib_data**
  Structure to hold the Calibration data.
- struct **bme680_tph_sett**
  BME680 sensor settings structure which comprises of ODR, over-sampling and filter settings.
- struct **bme680_gas_sett**
  BME680 gas sensor which comprises of gas settings and status parameters.
- struct **bme680_dev**
  BME680 device structure.

Macros

Common macros

- #define **INT8_C**(x) S8_C(x)
- #define **UINT8_C**(x) U8_C(x)
- #define **INT16_C**(x) S16_C(x)
- #define **UINT16_C**(x) U16_C(x)
- #define **INT32_C**(x) S32_C(x)
- #define **UINT32_C**(x) U32_C(x)
- #define **INT64_C**(x) S64_C(x)
- #define **UINT64_C**(x) U64_C(x)

C standard macros

- #define **NULL** ((void *) 0)
- #define **BME680_POLL_PERIOD_MS** **UINT8_C**(10)
- #define **BME680_I2C_ADDR_PRIMARY** **UINT8_C**(0x76)
- #define **BME680_I2C_ADDR_SECONDARY** **UINT8_C**(0x77)
- #define **BME680_CHIP_ID** **UINT8_C**(0x61)
- #define **BME680_COEFF_SIZE** **UINT8_C**(41)
- #define **BME680_COEFF_ADDR1_LEN** **UINT8_C**(25)
- #define **BME680_COEFF_ADDR2_LEN** **UINT8_C**(16)
- #define **BME680_FIELD_LENGTH** **UINT8_C**(15)
- #define **BME680_FIELD_ADDR_OFFSET** **UINT8_C**(17)
- #define **BME680_SOFT_RESET_CMD** **UINT8_C**(0xb6)
- #define **BME680_OK** **INT8_C**(0)
- #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)
- #define **BME680_W_DEFINE_PWR_MODE** **INT8_C**(1)
- #define **BME680_W_NO_NEW_DATA** **INT8_C**(2)
- #define **BME680_I_MIN_CORRECTION** **UINT8_C**(1)
- #define **BME680_I_MAX_CORRECTION** **UINT8_C**(2)
- #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)
- #define **BME680_FIELD0_ADDR** **UINT8_C**(0x1d)
- #define **BME680_RES_HEAT0_ADDR** **UINT8_C**(0x5a)
- #define **BME680_GAS_WAIT0_ADDR** **UINT8_C**(0x64)
- #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)
- #define **BME680_COEFF_ADDR1** **UINT8_C**(0x89)
- #define **BME680_COEFF_ADDR2** **UINT8_C**(0xe1)
- #define **BME680_CHIP_ID_ADDR** **UINT8_C**(0xd0)
- #define **BME680_SOFT_RESET_ADDR** **UINT8_C**(0xe0)
- #define **BME680_ENABLE_HEATER** **UINT8_C**(0x00)
- #define **BME680_DISABLE_HEATER** **UINT8_C**(0x08)
- #define **BME680_DISABLE_GAS_MEAS** **UINT8_C**(0x00)
- #define **BME680_ENABLE_GAS_MEAS** **UINT8_C**(0x01)
- #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)
- #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)
- #define **BME680_SLEEP_MODE UINT8_C**(0)
- #define **BME680_FORCED_MODE UINT8_C**(1)
- #define **BME680_RESET_PERIOD UINT32_C**(10)
- #define **BME680_MEM_PAGE0 UINT8_C**(0x10)
- #define **BME680_MEM_PAGE1 UINT8_C**(0x00)
- #define **BME680_HUM_REG_SHIFT_VAL UINT8_C**(4)
- #define **BME680_RUN_GAS_DISABLE UINT8_C**(0)
- #define **BME680_RUN_GAS_ENABLE UINT8_C**(1)
- #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)
- #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**)
- #define **BME680_NBCONV_MIN UINT8_C**(0)
- #define **BME680_NBCONV_MAX UINT8_C**(10)
- #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)
- #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)
- #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)
- #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)
- #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)
- #define **BME680_MAX_OVERFLOW_VAL** **INT32_C**(0x40000000)
- #define **BME680_CONCAT_BYTES**(msb, lsb) (((uint16_t)msb << 8) | (uint16_t)lsb)
- #define **BME680_SET_BITS**(reg_data, bitname, data)
- #define **BME680_GET_BITS**(reg_data, bitname)
- #define **BME680_SET_BITS_POS_0**(reg_data, bitname, data)
- #define **BME680_GET_BITS_POS_0**(reg_data, bitname) (reg_data & (bitname##_MSK))
- enum **bme680_intf** { **BME680_SPI_INTF**, **BME680_I2C_INTF** }
  Interface selection Enumerations.
- typedef int8_t(* **bme680_com_fptr_t**) (uint8_t dev_id, uint8_t reg_addr, uint8_t *data, uint16_t len)
- typedef void(* **bme680_delay_fptr_t**) (uint32_t period)

Detailed Description

Sensor driver for BME680 sensor.

BSD-3-Clause

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. 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.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

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 THE 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.

Date

23 Jan 2020

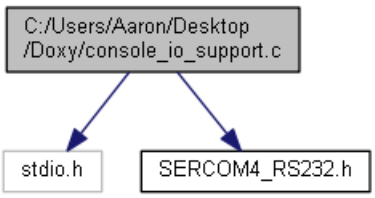
Version

3.5.10

C:/Users/Aaron/Desktop/Doxy/console_io_support.c File Reference

```
#include <stdio.h>
#include "SERCOM4_RS232.h"
```

Include dependency graph for console_io_support.c:



Functions

- int **_write** (FILE *f, char *buf, int n)
- int **_read** (FILE *f, char *buf, int n)
- int **_close** (FILE *f)
- int **_fstat** (FILE *f, void *p)
- int **_isatty** (FILE *f)
- int **_lseek** (FILE *f, int o, int w)
- void * **_sbrk** (int i)

Function Documentation

int _close (FILE * *f*)

Definition at line 26 of file console_io_support.c.

int _fstat (FILE * *f*, void * *p*)

Definition at line 31 of file console_io_support.c.

int _isatty (FILE * *f*)

Definition at line 37 of file console_io_support.c.

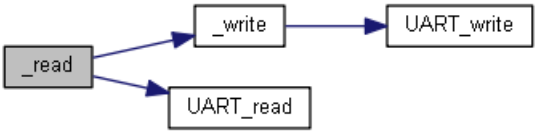
int _lseek (FILE * *f*, int *o*, int *w*)

Definition at line 42 of file console_io_support.c.

int _read (FILE * *f*, char * *buf*, int *n*)

Definition at line 14 of file console_io_support.c.

Here is the call graph for this function:



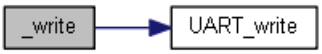
void* _sbrk (int *i*)

Definition at line 47 of file console_io_support.c.

int _write (FILE * *f*, char * *buf*, int *n*)

Definition at line 4 of file console_io_support.c.

Here is the call graph for this function:

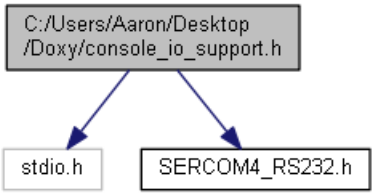


Here is the caller graph for this function:

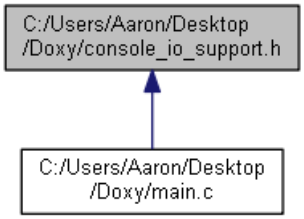
C:/Users/Aaron/Desktop/Doxy/console_io_support.h File Reference

```
#include <stdio.h>
#include "SERCOM4_RS232.h"
```

Include dependency graph for console_io_support.h:



This graph shows which files directly or indirectly include this file:



Functions

- int **_write** (FILE *f, char *buf, int n)
- int **_read** (FILE *f, char *buf, int n)
- int **_close** (FILE *f)
- int **_fstat** (FILE *f, void *p)
- int **_isatty** (FILE *f)
- int **_lseek** (FILE *f, int o, int w)
- void * **_sbrk** (int i)

Function Documentation

int _close (FILE * *f*)

Definition at line 26 of file console_io_support.c.

int _fstat (FILE * *f*, void * *p*)

Definition at line 31 of file console_io_support.c.

int _isatty (FILE * *f*)

Definition at line 37 of file console_io_support.c.

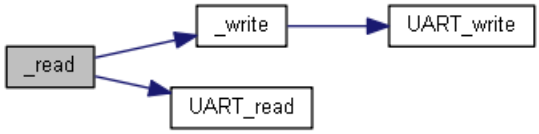
int _lseek (FILE * *f*, int *o*, int *w*)

Definition at line 42 of file console_io_support.c.

int _read (FILE * *f*, char * *buf*, int *n*)

Definition at line 14 of file console_io_support.c.

Here is the call graph for this function:



void* _sbrk (int *i*)

Definition at line 47 of file console_io_support.c.

int _write (FILE * *f*, char * *buf*, int *n*)

Definition at line 4 of file console_io_support.c.

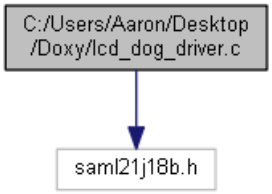
Here is the call graph for this function:

Here is the caller graph for this function:

C:/Users/Aaron/Desktop/Doxy/lcd_dog_driver.c File Reference

```
#include "saml21j18b.h"
```

Include dependency graph for lcd_dog_driver.c:



Functions

- void **delay_30us** (void)
- void **v_delay** (int a, int b)
- void **delay_40mS** (void)
- void **init_spi_lcd** (void)
- void **lcd_spi_transmit_CMD** (char command)
- void **lcd_spi_transmit_DATA** (char data)
- void **init_lcd_dog** (void)
- void **update_lcd_dog** (void)

Variables

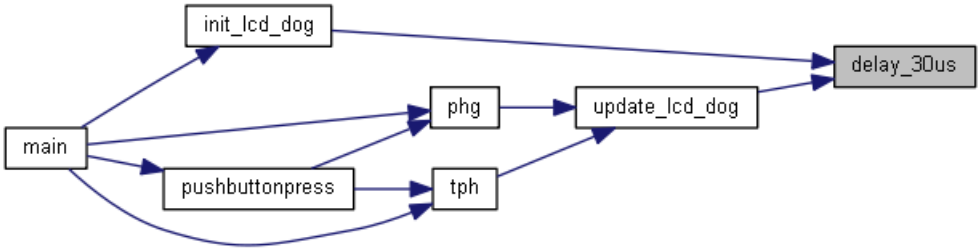
- unsigned char * **ARRAY_PINCFG0** =(unsigned char*) &REG_PORT_PINCFG0
- unsigned char * **ARRAY_PMUX0** = (unsigned char*) &REG_PORT_PMUX0
- char **dsp_buff_1** [17]
- char **dsp_buff_2** [17]
- char **dsp_buff_3** [17]

Function Documentation

void delay_30us (void)

Definition at line 51 of file lcd_dog_driver.c.

Here is the caller graph for this function:

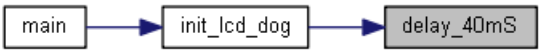


void delay_40mS (void)

Definition at line 87 of file lcd_dog_driver.c.

Here is the call graph for this function:

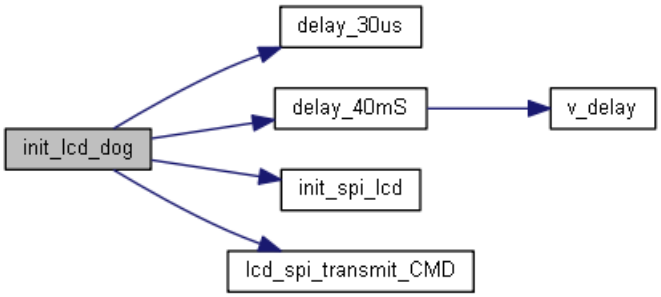
Here is the caller graph for this function:



void init_lcd_dog (void)

Definition at line 186 of file lcd_dog_driver.c.

Here is the call graph for this function:

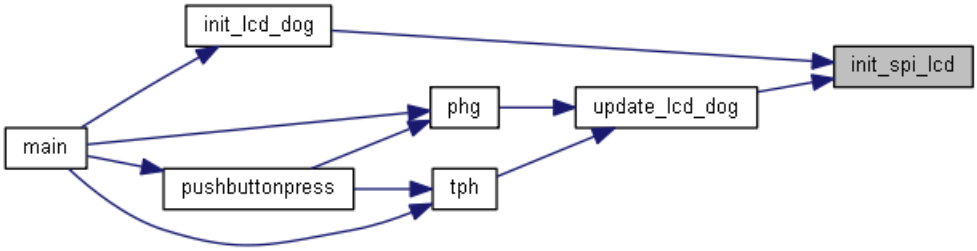


Here is the caller graph for this function:

void init_spi_lcd (void)

Definition at line 108 of file lcd_dog_driver.c.

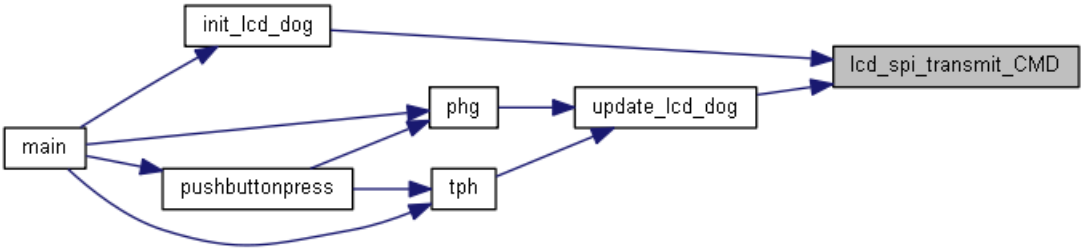
Here is the caller graph for this function:



void lcd_spi_transmit_CMD (char *command*)

Definition at line 150 of file lcd_dog_driver.c.

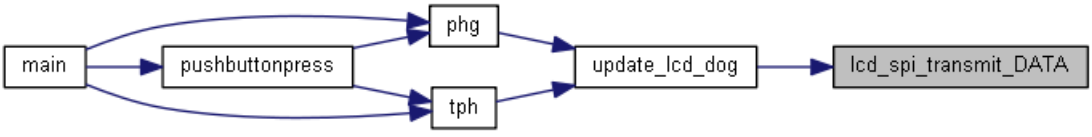
Here is the caller graph for this function:



void lcd_spi_transmit_DATA (char *data*)

Definition at line 169 of file lcd_dog_driver.c.

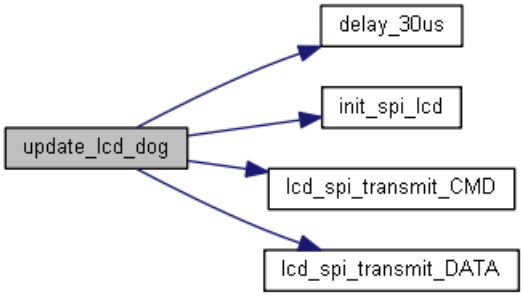
Here is the caller graph for this function:



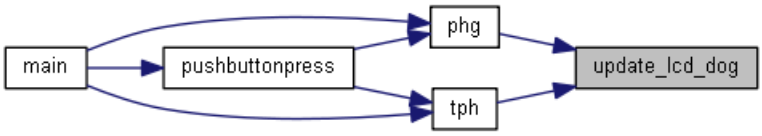
void update_lcd_dog (void)

Definition at line 250 of file lcd_dog_driver.c.

Here is the call graph for this function:



Here is the caller graph for this function:



void v_delay (int *a*, int *b*)

Definition at line 70 of file lcd_dog_driver.c.

Here is the caller graph for this function:

Variable Documentation

unsigned char* ARRAY_PINCFG0 =(unsigned char*) &REG_PORT_PINCFG0

Definition at line 38 of file lcd_dog_driver.c.

unsigned char* ARRAY_PMUX0 = (unsigned char*) &REG_PORT_PMUX0

Definition at line 39 of file lcd_dog_driver.c.

char dsp_buff_1[17]

Definition at line 40 of file lcd_dog_driver.c.

char dsp_buff_2[17]

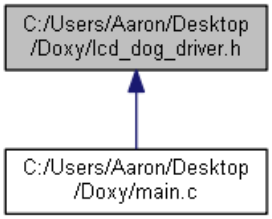
Definition at line 41 of file lcd_dog_driver.c.

char dsp_buff_3[17]

Definition at line 42 of file lcd_dog_driver.c.

C:/Users/Aaron/Desktop/Doxy/lcd_dog_driver.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- void **delay_30us** (void)
- void **v_delay** (int a, int b)
- void **delay_40mS** (void)
- void **init_spi_lcd** (void)
- void **lcd_spi_transmit_CMD** (char command)
- void **lcd_spi_transmit_DATA** (char data)
- void **init_lcd_dog** (void)
- void **update_lcd_dog** (void)

Variables

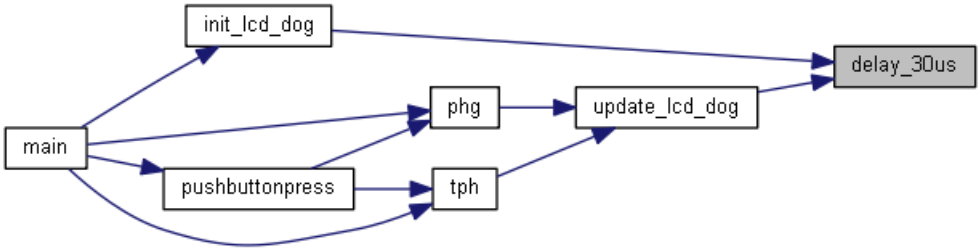
- char **dsp_buff_1** [17]
- char **dsp_buff_2** [17]
- char **dsp_buff_3** [17]

Function Documentation

void delay_30us (void)

Definition at line 51 of file lcd_dog_driver.c.

Here is the caller graph for this function:

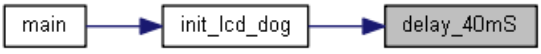


void delay_40mS (void)

Definition at line 87 of file lcd_dog_driver.c.

Here is the call graph for this function:

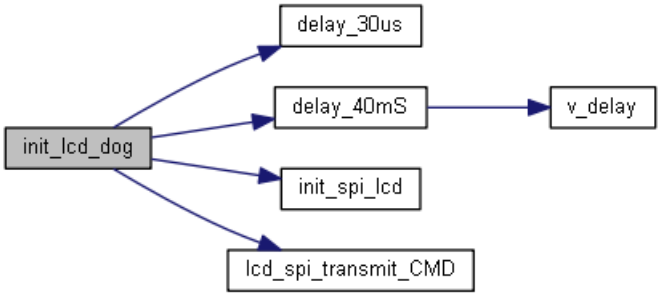
Here is the caller graph for this function:



void init_lcd_dog (void)

Definition at line 186 of file lcd_dog_driver.c.

Here is the call graph for this function:

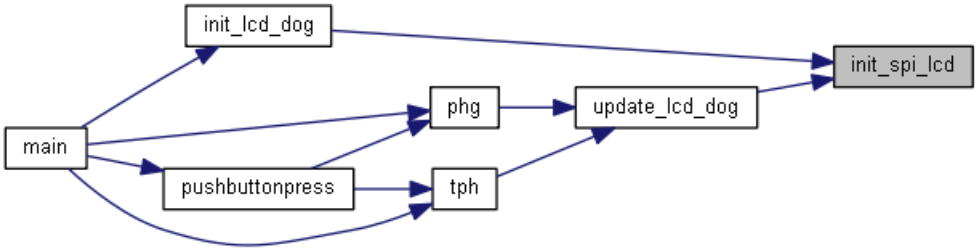


Here is the caller graph for this function:

void init_spi_lcd (void)

Definition at line 108 of file lcd_dog_driver.c.

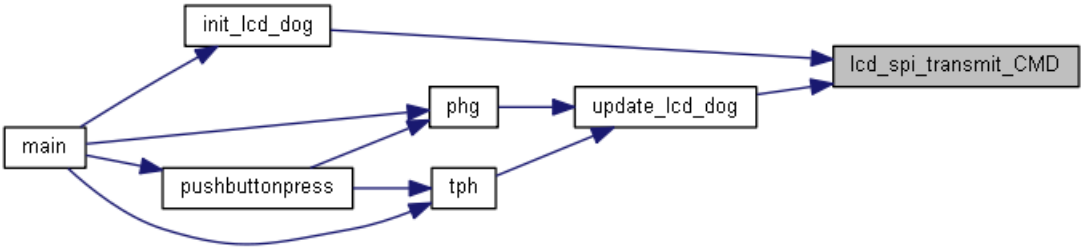
Here is the caller graph for this function:



void lcd_spi_transmit_CMD (char *command*)

Definition at line 150 of file lcd_dog_driver.c.

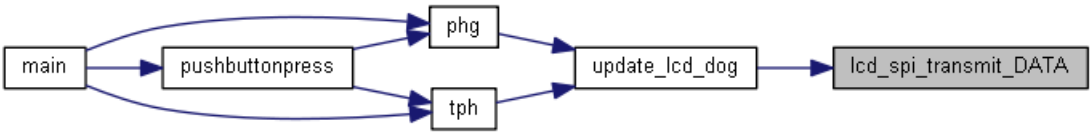
Here is the caller graph for this function:



void lcd_spi_transmit_DATA (char *data*)

Definition at line 169 of file lcd_dog_driver.c.

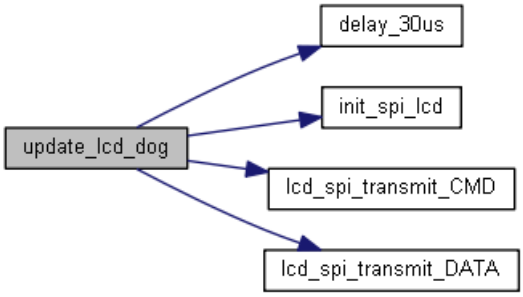
Here is the caller graph for this function:



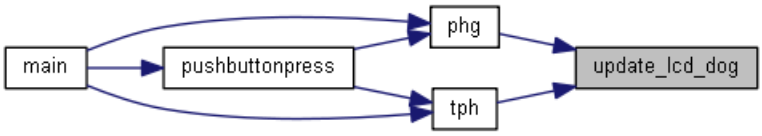
void update_lcd_dog (void)

Definition at line 250 of file lcd_dog_driver.c.

Here is the call graph for this function:



Here is the caller graph for this function:



void v_delay (int *a*, int *b*)

Definition at line 70 of file lcd_dog_driver.c.

Here is the caller graph for this function:

Variable Documentation

char dsp_buff_1[17]

Definition at line 4 of file lcd_dog_driver.h.

char dsp_buff_2[17]

Definition at line 4 of file lcd_dog_driver.h.

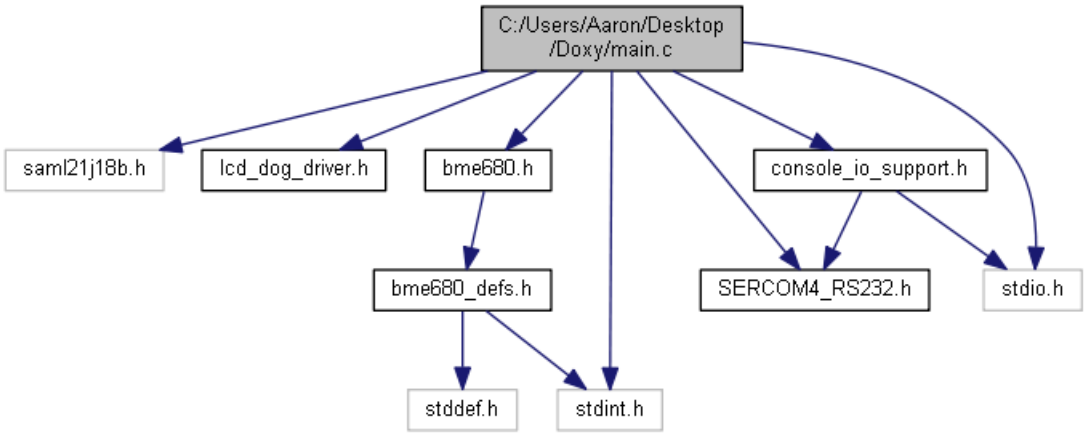
char dsp_buff_3[17]

Definition at line 4 of file lcd_dog_driver.h.

C:/Users/Aaron/Desktop/Doxy/main.c File Reference

```
#include "saml21j18b.h"
#include "lcd_dog_driver.h"
#include "SERCOM4_RS232.h"
#include "bme680.h"
#include "stdio.h"
#include "console_io_support.h"
#include "stdint.h"
```

Include dependency graph for main.c:



Functions

- void **init_spi_bme680** (void)
- void **user_delay_ms** (uint32_t period)
- 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)
- void **pushbuttonpress** (_Bool *point, struct **bme680_field_data** data)
- void **tph** (struct **bme680_field_data** data)
- void **phg** (struct **bme680_field_data** data)
- int **main** (void)

Variables

- unsigned char * **ARRAY_PINCFG0** = (unsigned char*) & REG_PORT_PINCFG0
- unsigned char * **ARRAY_PMUX0** = (unsigned char*) & REG_PORT_PMUX0

Function Documentation

void init_spi_bme680 (void)

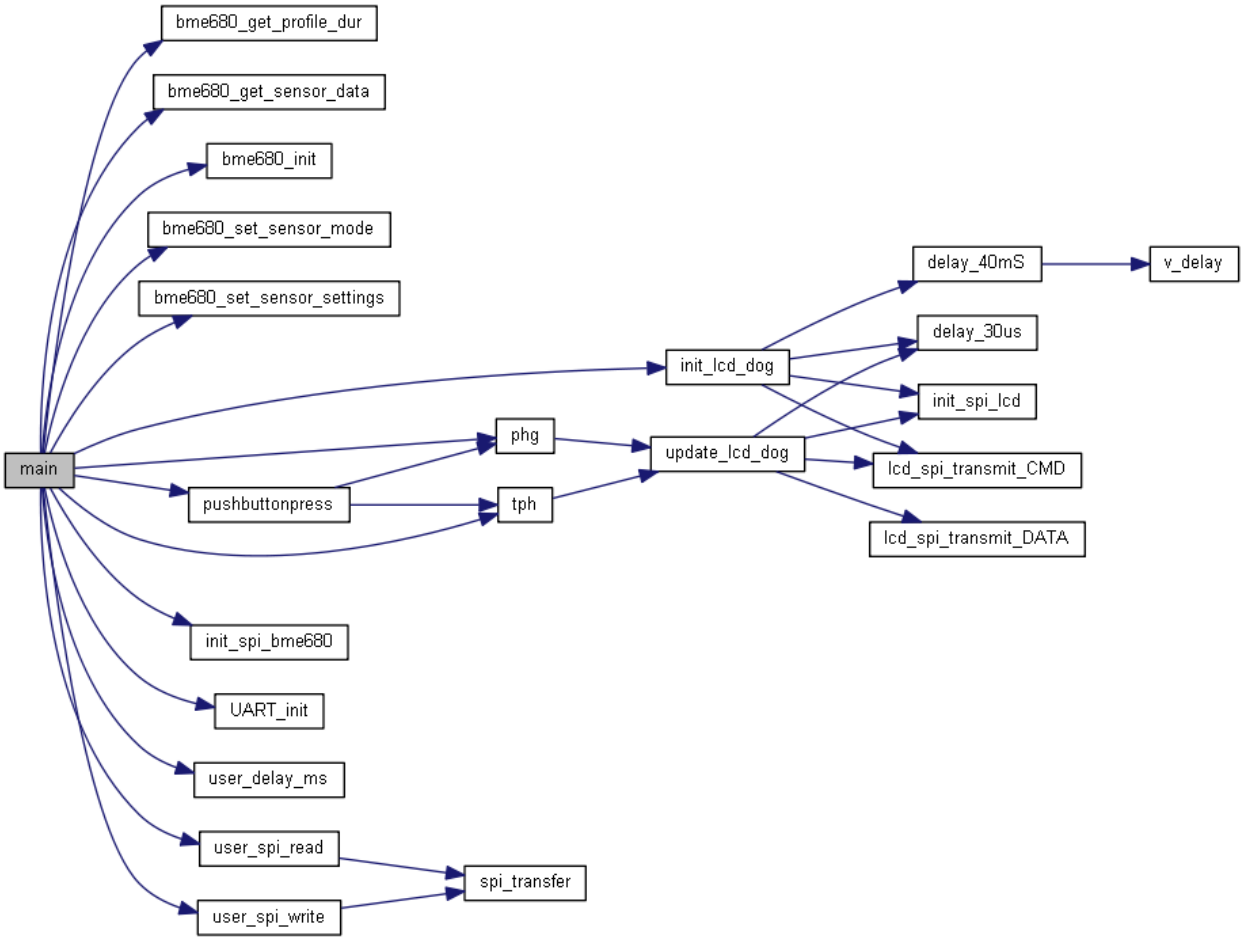
Definition at line 234 of file main.c.

Here is the caller graph for this function:

int main (void)

Definition at line 45 of file main.c.

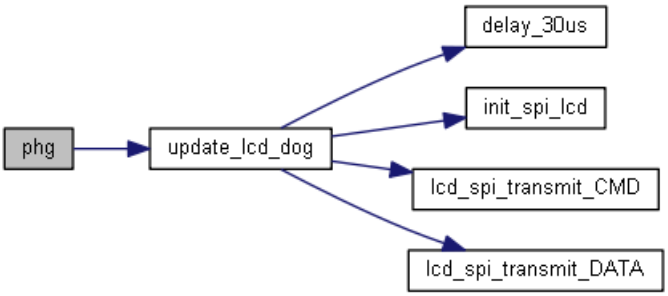
Here is the call graph for this function:



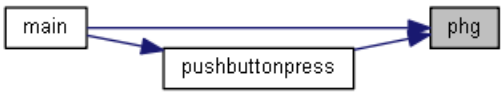
void phg (struct bme680_field_data *data*)

Definition at line 410 of file main.c.

Here is the call graph for this function:



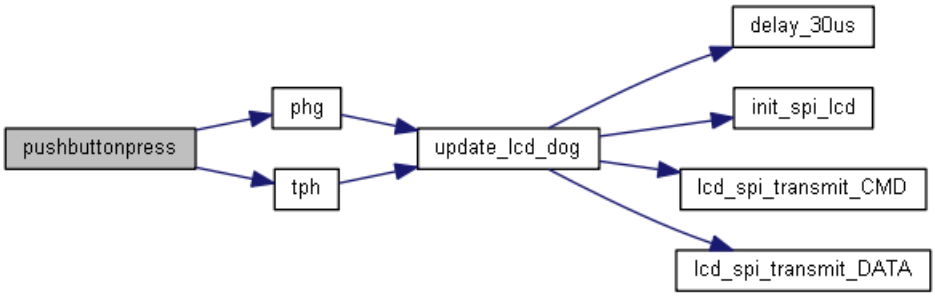
Here is the caller graph for this function:



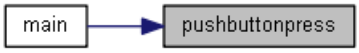
void pushbuttonpress (_Bool * *point*, struct bme680_field_data *data*)

Definition at line 428 of file main.c.

Here is the call graph for this function:



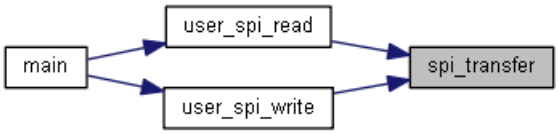
Here is the caller graph for this function:



uint8_t spi_transfer (uint8_t *data*)

Definition at line 300 of file main.c.

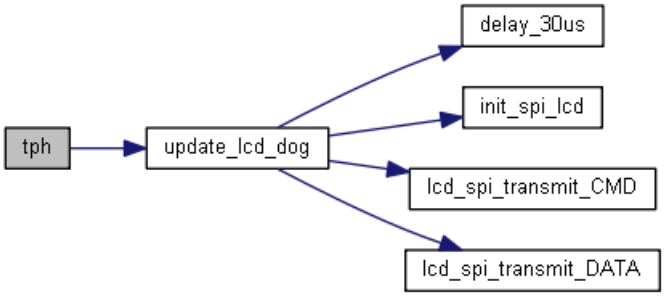
Here is the caller graph for this function:



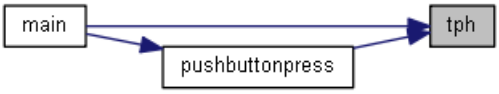
void tph (struct bme680_field_data *data*)

Definition at line 390 of file main.c.

Here is the call graph for this function:



Here is the caller graph for this function:



void user_delay_ms (uint32_t *period*)

Definition at line 275 of file main.c.

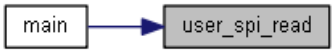
Here is the caller graph for this function:

int8_t user_spi_read (uint8_t *dev_id*, uint8_t *reg_addr*, uint8_t * *reg_data*, uint16_t *len*)

Definition at line 324 of file main.c.

Here is the call graph for this function:

Here is the caller graph for this function:



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

Definition at line 358 of file main.c.

Here is the call graph for this function:

Here is the caller graph for this function:

Variable Documentation

unsigned char* ARRAY_PINCFG0 = (unsigned char*) & REG_PORT_PINCFG0

Definition at line 27 of file main.c.

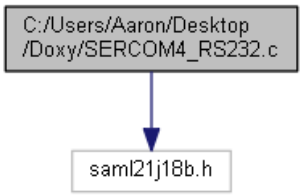
unsigned char* ARRAY_PMUX0 = (unsigned char*) & REG_PORT_PMUX0

Definition at line 28 of file main.c.

C:/Users/Aaron/Desktop/Doxy/SERCOM4_RS232.c File Reference

```
#include "saml21j18b.h"
```

Include dependency graph for SERCOM4_RS232.c:



Functions

- void **UART_init** (void)
- void **UART_write** (char data)
- char **UART_read** (void)
- void **delayMs** (int n)

Variables

- unsigned char * **ARRAY_PINCFG1** = (unsigned char*) &REG_PORT_PINCFG1
- unsigned char * **ARRAY_PMUX1** = (unsigned char*) &REG_PORT_PMUX1

Function Documentation

void delayMs (int *n*)

Definition at line 110 of file SERCOM4_RS232.c.

void UART_init (void)

Definition at line 35 of file SERCOM4_RS232.c.

Here is the caller graph for this function:

char UART_read (void)

Definition at line 95 of file SERCOM4_RS232.c.

Here is the caller graph for this function:

void UART_write (char *data*)

Definition at line 81 of file SERCOM4_RS232.c.

Here is the caller graph for this function:

Variable Documentation

unsigned char* ARRAY_PINCFG1 = (unsigned char*) &REG_PORT_PINCFG1

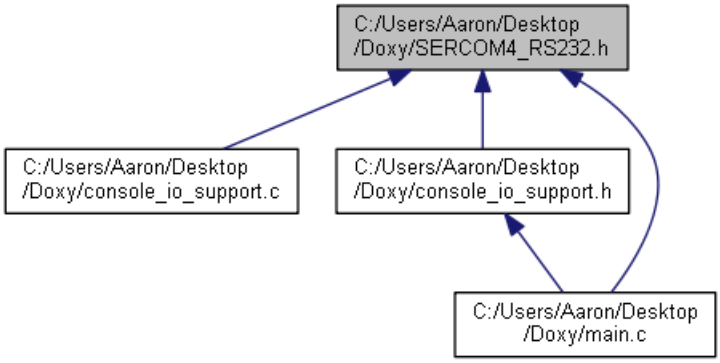
Definition at line 20 of file SERCOM4_RS232.c.

unsigned char* ARRAY_PMUX1 = (unsigned char*) &REG_PORT_PMUX1

Definition at line 21 of file SERCOM4_RS232.c.

C:/Users/Aaron/Desktop/Doxy/SERCOM4_RS232.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- void **UART_init** (void)
- void **UART_write** (char data)
- void **delayMs** (int n)
- char **UART_read** (void)

Function Documentation

void delayMs (int *n*)

Definition at line 110 of file SERCOM4_RS232.c.

void UART_init (void)

Definition at line 35 of file SERCOM4_RS232.c.

Here is the caller graph for this function:

char UART_read (void)

Definition at line 95 of file SERCOM4_RS232.c.

Here is the caller graph for this function:

void UART_write (char *data*)

Definition at line 81 of file SERCOM4_RS232.c.

Here is the caller graph for this function:

Index

_close
  console_io_support.c, 49
  console_io_support.h, 51
_fstat
  console_io_support.c, 49
  console_io_support.h, 51
_isatty
  console_io_support.c, 49
  console_io_support.h, 51
_lseek
  console_io_support.c, 49
  console_io_support.h, 51
_read
  console_io_support.c, 49
  console_io_support.h, 52
_sbrk
  console_io_support.c, 50
  console_io_support.h, 52
_write
  console_io_support.c, 50
  console_io_support.h, 52
amb_temp
  bme680_dev, 33
ARRAY_PINCFG0
  lcd_dog_driver.c, 55
  main.c, 62
ARRAY_PINCFG1
  SERCOM4_RS232.c, 64
ARRAY_PMUX0
  lcd_dog_driver.c, 55
  main.c, 62
ARRAY_PMUX1
  SERCOM4_RS232.c, 64
BME680_ADDR_GAS_CONF_START
  SENSOR API, 10
BME680_ADDR_RANGE_SW_ERR_ADDR
  SENSOR API, 10
BME680_ADDR_RES_HEAT_RANGE_AD
  DR
  SENSOR API, 10
BME680_ADDR_RES_HEAT_VAL_ADDR
  SENSOR API, 10
BME680_ADDR_SENS_CONF_START
  SENSOR API, 10
BME680_BIT_H1_DATA_MSK
  SENSOR API, 10
bme680_calib_data, 29
  par_gh1, 29
  par_gh2, 29
  par_gh3, 30
  par_h1, 30
  par_h2, 30
  par_h3, 30
  par_h4, 30
  par_h5, 30
  par_h6, 30
  par_h7, 30
  par_p1, 30
  par_p10, 30
  par_p2, 30
  par_p3, 31
  par_p4, 31
  par_p5, 31
  par_p6, 31
  par_p7, 31
  par_p8, 31
  par_p9, 31
  par_t1, 31
  par_t2, 31
  par_t3, 31
  range_sw_err, 31
  res_heat_range, 31
  res_heat_val, 32
  t_fine, 32
BME680_CHIP_ID
  SENSOR API, 10
BME680_CHIP_ID_ADDR
  SENSOR API, 10
BME680_COEFF_ADDR1
  SENSOR API, 11
BME680_COEFF_ADDR1_LEN
  SENSOR API, 11
BME680_COEFF_ADDR2
  SENSOR API, 11
BME680_COEFF_ADDR2_LEN
  SENSOR API, 11
BME680_COEFF_SIZE
  SENSOR API, 11
bme680_com_fptr_t
  SENSOR API, 23
BME680_CONCAT_BYTES
  SENSOR API, 11
BME680_CONF_HEAT_CTRL_ADDR
  SENSOR API, 11
BME680_CONF_ODR_FILT_ADDR
  SENSOR API, 11
BME680_CONF_ODR_RUN_GAS_NBC_A
  DDR
  SENSOR API, 11
BME680_CONF_OS_H_ADDR
  SENSOR API, 11
BME680_CONF_T_P_MODE_ADDR
  SENSOR API, 11
bme680_delay_fptr_t
  SENSOR API, 24
bme680_dev, 33
  amb_temp, 33
  calib, 33
  chip_id, 33
  com_rslt, 34
  delay_ms, 34
  dev_id, 34

gas_sett, 34
info_msg, 34
intf, 34
mem_page, 34
new_fields, 34
power_mode, 34
read, 34
tph_sett, 34
write, 34
BME680_DISABLE_GAS_MEAS
SENSOR API, 11
BME680_DISABLE_HEATER
SENSOR API, 12
BME680_E_COM_FAIL
SENSOR API, 12
BME680_E_DEV_NOT_FOUND
SENSOR API, 12
BME680_E_INVALID_LENGTH
SENSOR API, 12
BME680_E_NULL_PTR
SENSOR API, 12
BME680_ENABLE_GAS_MEAS
SENSOR API, 12
BME680_ENABLE_HEATER
SENSOR API, 12
BME680_FIELD_ADDR_OFFSET
SENSOR API, 12
bme680_field_data, 36
gas_index, 36
gas_resistance, 36
humidity, 36
meas_index, 36
pressure, 36
status, 36
temperature, 37
BME680_FIELD_DATA_LENGTH
SENSOR API, 12
BME680_FIELD_LENGTH
SENSOR API, 12
BME680_FIELD0_ADDR
SENSOR API, 12
BME680_FILTER_MSK
SENSOR API, 12
BME680_FILTER_POS
SENSOR API, 13
BME680_FILTER_SEL
SENSOR API, 13
BME680_FILTER_SIZE_0
SENSOR API, 13
BME680_FILTER_SIZE_1
SENSOR API, 13
BME680_FILTER_SIZE_127
SENSOR API, 13
BME680_FILTER_SIZE_15
SENSOR API, 13
BME680_FILTER_SIZE_3
SENSOR API, 13
BME680_FILTER_SIZE_31
SENSOR API, 13
BME680_FILTER_SIZE_63
SENSOR API, 13
BME680_FILTER_SIZE_7
SENSOR API, 13
BME680_FORCED_MODE
SENSOR API, 13
BME680_GAS_INDEX_MSK
SENSOR API, 13
BME680_GAS_MEAS_MSK
SENSOR API, 14
BME680_GAS_MEAS_POS
SENSOR API, 14
BME680_GAS_MEAS_SEL
SENSOR API, 14
BME680_GAS_RANGE_MSK
SENSOR API, 14
BME680_GAS_REG_BUF_LENGTH
SENSOR API, 14
BME680_GAS_SENSOR_SEL
SENSOR API, 14
bme680_gas_sett, 38
heatr_ctrl, 38
heatr_dur, 38
heatr_temp, 38
nb_conv, 38
run_gas, 38
BME680_GAS_WAIT0_ADDR
SENSOR API, 14
BME680_GASM_VALID_MSK
SENSOR API, 14
BME680_GET_BITS
SENSOR API, 14
BME680_GET_BITS_POS_0
SENSOR API, 14
bme680_get_profile_dur
SENSOR API, 24
bme680_get_regs
SENSOR API, 24
bme680_get_sensor_data
SENSOR API, 25
bme680_get_sensor_mode
SENSOR API, 25
bme680_get_sensor_settings
SENSOR API, 26
BME680_GH1_REG
SENSOR API, 14
BME680_GH2_LSB_REG
SENSOR API, 15
BME680_GH2_MSB_REG
SENSOR API, 15
BME680_GH3_REG
SENSOR API, 15
BME680_H1_LSB_REG
SENSOR API, 15
BME680_H1_MSB_REG
SENSOR API, 15
BME680_H2_LSB_REG
SENSOR API, 15
BME680_H2_MSB_REG
SENSOR API, 15
BME680_H3_REG

SENSOR API, 15
BME680_H4_REG
SENSOR API, 15
BME680_H5_REG
SENSOR API, 15
BME680_H6_REG
SENSOR API, 15
BME680_H7_REG
SENSOR API, 15
BME680_HCNTRL_SEL
SENSOR API, 16
BME680_HCTRL_MSK
SENSOR API, 16
BME680_HEAT_STAB_MSK
SENSOR API, 16
BME680_HUM_REG_SHIFT_VAL
SENSOR API, 16
BME680_I_MAX_CORRECTION
SENSOR API, 16
BME680_I_MIN_CORRECTION
SENSOR API, 16
BME680_I2C_ADDR_PRIMARY
SENSOR API, 16
BME680_I2C_ADDR_SECONDARY
SENSOR API, 16
BME680_I2C_INTF
SENSOR API, 24
bme680_init
SENSOR API, 26
bme680_intf
SENSOR API, 24
BME680_MAX_OVERFLOW_VAL
SENSOR API, 16
BME680_MEM_PAGE_ADDR
SENSOR API, 17
BME680_MEM_PAGE_MSK
SENSOR API, 17
BME680_MEM_PAGE0
SENSOR API, 16
BME680_MEM_PAGE1
SENSOR API, 16
BME680_MODE_MSK
SENSOR API, 17
BME680_NBCONV_MAX
SENSOR API, 17
BME680_NBCONV_MIN
SENSOR API, 17
BME680_NBCONV_MSK
SENSOR API, 17
BME680_NBCONV_SEL
SENSOR API, 17
BME680_NEW_DATA_MSK
SENSOR API, 17
BME680_OK
SENSOR API, 17
BME680_OS_16X
SENSOR API, 17
BME680_OS_1X
SENSOR API, 17
BME680_OS_2X
SENSOR API, 17
BME680_OS_4X
SENSOR API, 18
BME680_OS_8X
SENSOR API, 18
BME680_OS_NONE
SENSOR API, 18
BME680_OSH_MSK
SENSOR API, 18
BME680_OSH_SEL
SENSOR API, 18
BME680_OSP_MSK
SENSOR API, 18
BME680_OSP_POS
SENSOR API, 18
BME680_OSP_SEL
SENSOR API, 18
BME680_OST_MSK
SENSOR API, 18
BME680_OST_POS
SENSOR API, 18
BME680_OST_SEL
SENSOR API, 18
BME680_P1_LSB_REG
SENSOR API, 19
BME680_P1_MSB_REG
SENSOR API, 19
BME680_P10_REG
SENSOR API, 18
BME680_P2_LSB_REG
SENSOR API, 19
BME680_P2_MSB_REG
SENSOR API, 19
BME680_P3_REG
SENSOR API, 19
BME680_P4_LSB_REG
SENSOR API, 19
BME680_P4_MSB_REG
SENSOR API, 19
BME680_P5_LSB_REG
SENSOR API, 19
BME680_P5_MSB_REG
SENSOR API, 19
BME680_P6_REG
SENSOR API, 19
BME680_P7_REG
SENSOR API, 19
BME680_P8_LSB_REG
SENSOR API, 19
BME680_P8_MSB_REG
SENSOR API, 20
BME680_P9_LSB_REG
SENSOR API, 20
BME680_P9_MSB_REG
SENSOR API, 20
BME680_POLL_PERIOD_MS
SENSOR API, 20
BME680_REG_BUFFER_LENGTH
SENSOR API, 20
BME680_REG_FILTER_INDEX

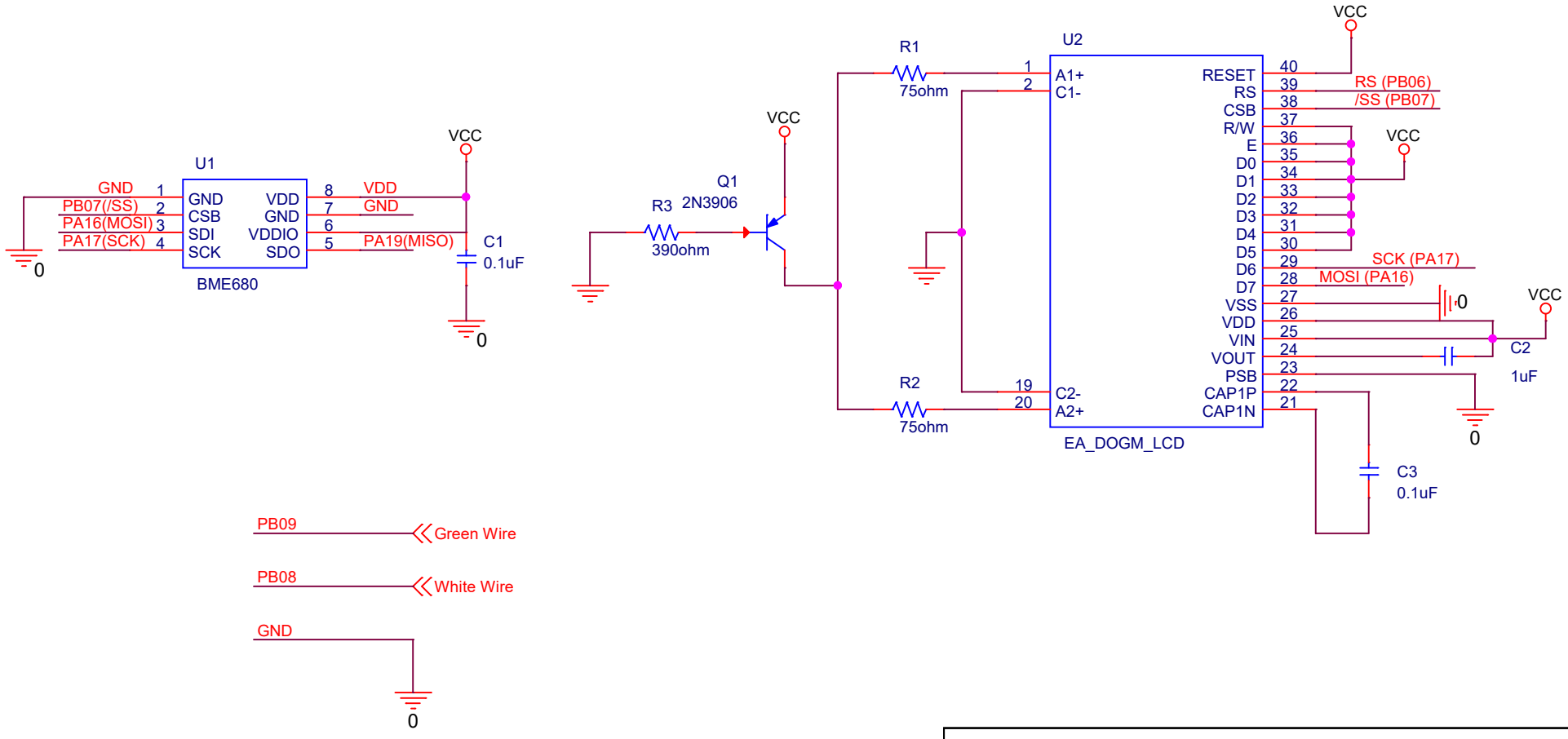
SENSOR API, 20
BME680_REG_HCTRL_INDEX
SENSOR API, 20
BME680_REG_HUM_INDEX
SENSOR API, 20
BME680_REG_NBCONV_INDEX
SENSOR API, 20
BME680_REG_PRES_INDEX
SENSOR API, 20
BME680_REG_RUN_GAS_INDEX
SENSOR API, 20
BME680_REG_TEMP_INDEX
SENSOR API, 20
BME680_RES_HEAT0_ADDR
SENSOR API, 21
BME680_RESET_PERIOD
SENSOR API, 21
BME680_RHRANGE_MSK
SENSOR API, 21
BME680_RSERROR_MSK
SENSOR API, 21
BME680_RUN_GAS_DISABLE
SENSOR API, 21
BME680_RUN_GAS_ENABLE
SENSOR API, 21
BME680_RUN_GAS_MSK
SENSOR API, 21
BME680_RUN_GAS_POS
SENSOR API, 21
BME680_RUN_GAS_SEL
SENSOR API, 21
BME680_SET_BITS
SENSOR API, 21
BME680_SET_BITS_POS_0
SENSOR API, 21
bme680_set_profile_dur
SENSOR API, 26
bme680_set_regs
SENSOR API, 26
bme680_set_sensor_mode
SENSOR API, 27
bme680_set_sensor_settings
SENSOR API, 27
BME680_SLEEP_MODE
SENSOR API, 22
bme680_soft_reset
SENSOR API, 28
BME680_SOFT_RESET_ADDR
SENSOR API, 22
BME680_SOFT_RESET_CMD
SENSOR API, 22
BME680_SPI_INTF
SENSOR API, 24
BME680_SPI_RD_MSK
SENSOR API, 22
BME680_SPI_WR_MSK
SENSOR API, 22
BME680_T1_LSB_REG
SENSOR API, 22
BME680_T1_MSB_REG
SENSOR API, 22
BME680_T2_LSB_REG
SENSOR API, 22
BME680_T2_MSB_REG
SENSOR API, 22
BME680_T3_REG
SENSOR API, 22
BME680_TMP_BUFFER_LENGTH
SENSOR API, 22
bme680_tph_sett, 39
filter, 39
os_hum, 39
os_pres, 39
os_temp, 39
BME680_W_DEFINE_PWR_MODE
SENSOR API, 22
BME680_W_NO_NEW_DATA
SENSOR API, 23
C:/Users/Aaron/Desktop/Doxy/bme680.c, 40
C:/Users/Aaron/Desktop/Doxy/bme680.h, 42
C:/Users/Aaron/Desktop/Doxy/bme680_defs.h, 44
C:/Users/Aaron/Desktop/Doxy/console_io_support.c, 49
C:/Users/Aaron/Desktop/Doxy/console_io_support.h, 51
C:/Users/Aaron/Desktop/Doxy/lcd_dog_driver.c, 53
C:/Users/Aaron/Desktop/Doxy/lcd_dog_driver.h, 56
C:/Users/Aaron/Desktop/Doxy/main.c, 59
C:/Users/Aaron/Desktop/Doxy/SERCOM4_RS232.c, 63
C:/Users/Aaron/Desktop/Doxy/SERCOM4_RS232.h, 65
calib
bme680_dev, 33
chip_id
bme680_dev, 33
com_rslt
bme680_dev, 34
console_io_support.c
_close, 49
_fstat, 49
_isatty, 49
_lseek, 49
_read, 49
_sbrk, 50
_write, 50
console_io_support.h
_close, 51
_fstat, 51
_isatty, 51
_lseek, 51
_read, 52
_sbrk, 52
_write, 52
delay_30us
lcd_dog_driver.c, 53
lcd_dog_driver.h, 56

delay_40mS
lcd_dog_driver.c, 53
lcd_dog_driver.h, 56
delay_ms
bme680_dev, 34
delayMs
SERCOM4_RS232.c, 63
SERCOM4_RS232.h, 65
dev_id
bme680_dev, 34
dsp_buff_1
lcd_dog_driver.c, 55
lcd_dog_driver.h, 58
dsp_buff_2
lcd_dog_driver.c, 55
lcd_dog_driver.h, 58
dsp_buff_3
lcd_dog_driver.c, 55
lcd_dog_driver.h, 58
filter
bme680_tph_sett, 39
gas_index
bme680_field_data, 36
gas_resistance
bme680_field_data, 36
gas_sett
bme680_dev, 34
heatr_ctrl
bme680_gas_sett, 38
heatr_dur
bme680_gas_sett, 38
heatr_temp
bme680_gas_sett, 38
humidity
bme680_field_data, 36
info_msg
bme680_dev, 34
init_lcd_dog
lcd_dog_driver.c, 54
lcd_dog_driver.h, 57
init_spi_bme680
main.c, 59
init_spi_lcd
lcd_dog_driver.c, 54
lcd_dog_driver.h, 57
INT16_C
SENSOR API, 23
INT32_C
SENSOR API, 23
INT64_C
SENSOR API, 23
INT8_C
SENSOR API, 23
intf
bme680_dev, 34
lcd_dog_driver.c
ARRAY_PINCFG0, 55
ARRAY_PMUX0, 55
delay_30us, 53
delay_40mS, 53
dsp_buff_1, 55
dsp_buff_2, 55
dsp_buff_3, 55
init_lcd_dog, 54
init_spi_lcd, 54
lcd_spi_transmit_CMD, 54
lcd_spi_transmit_DATA, 54
update_lcd_dog, 55
v_delay, 55
lcd_dog_driver.h
delay_30us, 56
delay_40mS, 56
dsp_buff_1, 58
dsp_buff_2, 58
dsp_buff_3, 58
init_lcd_dog, 57
init_spi_lcd, 57
lcd_spi_transmit_CMD, 57
lcd_spi_transmit_DATA, 57
update_lcd_dog, 58
v_delay, 58
lcd_spi_transmit_CMD
lcd_dog_driver.c, 54
lcd_dog_driver.h, 57
lcd_spi_transmit_DATA
lcd_dog_driver.c, 54
lcd_dog_driver.h, 57
main
main.c, 59
main.c
ARRAY_PINCFG0, 62
ARRAY_PMUX0, 62
init_spi_bme680, 59
main, 59
phg, 60
pushbuttonpress, 60
spi_transfer, 61
tph, 61
user_delay_ms, 61
user_spi_read, 61
user_spi_write, 62
meas_index
bme680_field_data, 36
mem_page
bme680_dev, 34
nb_conv
bme680_gas_sett, 38
new_fields
bme680_dev, 34
NULL
SENSOR API, 23
os_hum
bme680_tph_sett, 39
os_pres
bme680_tph_sett, 39
os_temp
bme680_tph_sett, 39
par_gh1
bme680_calib_data, 29
par_gh2

bme680_calib_data, 29
par_gh3
bme680_calib_data, 30
par_h1
bme680_calib_data, 30
par_h2
bme680_calib_data, 30
par_h3
bme680_calib_data, 30
par_h4
bme680_calib_data, 30
par_h5
bme680_calib_data, 30
par_h6
bme680_calib_data, 30
par_h7
bme680_calib_data, 30
par_p1
bme680_calib_data, 30
par_p10
bme680_calib_data, 30
par_p2
bme680_calib_data, 30
par_p3
bme680_calib_data, 31
par_p4
bme680_calib_data, 31
par_p5
bme680_calib_data, 31
par_p6
bme680_calib_data, 31
par_p7
bme680_calib_data, 31
par_p8
bme680_calib_data, 31
par_p9
bme680_calib_data, 31
par_t1
bme680_calib_data, 31
par_t2
bme680_calib_data, 31
par_t3
bme680_calib_data, 31
phg
main.c, 60
power_mode
bme680_dev, 34
pressure
bme680_field_data, 36
pushbuttonpress
main.c, 60
range_sw_err
bme680_calib_data, 31
read
bme680_dev, 34
res_heat_range
bme680_calib_data, 31
res_heat_val
bme680_calib_data, 32
run_gas
bme680_gas_sett, 38
SENSOR API, 6
BME680_ADDR_GAS_CONF_START, 10
BME680_ADDR_RANGE_SW_ERR_ADDR, 10
BME680_ADDR_RES_HEAT_RANGE_ADDR, 10
BME680_ADDR_RES_HEAT_VAL_ADDR, 10
BME680_ADDR_SENS_CONF_START, 10
BME680_BIT_H1_DATA_MSK, 10
BME680_CHIP_ID, 10
BME680_CHIP_ID_ADDR, 10
BME680_COEFF_ADDR1, 11
BME680_COEFF_ADDR1_LEN, 11
BME680_COEFF_ADDR2, 11
BME680_COEFF_ADDR2_LEN, 11
BME680_COEFF_SIZE, 11
bme680_com_fptr_t, 23
BME680_CONCAT_BYTES, 11
BME680_CONF_HEAT_CTRL_ADDR, 11
BME680_CONF_ODR_FILT_ADDR, 11
BME680_CONF_ODR_RUN_GAS_NBC_ADDR, 11
BME680_CONF_OS_H_ADDR, 11
BME680_CONF_T_P_MODE_ADDR, 11
bme680_delay_fptr_t, 24
BME680_DISABLE_GAS_MEAS, 11
BME680_DISABLE_HEATER, 12
BME680_E_COM_FAIL, 12
BME680_E_DEV_NOT_FOUND, 12
BME680_E_INVALID_LENGTH, 12
BME680_E_NULL_PTR, 12
BME680_ENABLE_GAS_MEAS, 12
BME680_ENABLE_HEATER, 12
BME680_FIELD_ADDR_OFFSET, 12
BME680_FIELD_DATA_LENGTH, 12
BME680_FIELD_LENGTH, 12
BME680_FIELD0_ADDR, 12
BME680_FILTER_MSK, 12
BME680_FILTER_POS, 13
BME680_FILTER_SEL, 13
BME680_FILTER_SIZE_0, 13
BME680_FILTER_SIZE_1, 13
BME680_FILTER_SIZE_127, 13
BME680_FILTER_SIZE_15, 13
BME680_FILTER_SIZE_3, 13
BME680_FILTER_SIZE_31, 13
BME680_FILTER_SIZE_63, 13
BME680_FILTER_SIZE_7, 13
BME680_FORCED_MODE, 13
BME680_GAS_INDEX_MSK, 13
BME680_GAS_MEAS_MSK, 14
BME680_GAS_MEAS_POS, 14
BME680_GAS_MEAS_SEL, 14
BME680_GAS_RANGE_MSK, 14
BME680_GAS_REG_BUF_LENGTH, 14
BME680_GAS_SENSOR_SEL, 14
BME680_GAS_WAIT0_ADDR, 14

BME680_GASM_VALID_MSK, 14
BME680_GET_BITS, 14
BME680_GET_BITS_POS_0, 14
bme680_get_profile_dur, 24
bme680_get_regs, 24
bme680_get_sensor_data, 25
bme680_get_sensor_mode, 25
bme680_get_sensor_settings, 26
BME680_GH1_REG, 14
BME680_GH2_LSB_REG, 15
BME680_GH2_MSB_REG, 15
BME680_GH3_REG, 15
BME680_H1_LSB_REG, 15
BME680_H1_MSB_REG, 15
BME680_H2_LSB_REG, 15
BME680_H2_MSB_REG, 15
BME680_H3_REG, 15
BME680_H4_REG, 15
BME680_H5_REG, 15
BME680_H6_REG, 15
BME680_H7_REG, 15
BME680_HCNTRL_SEL, 16
BME680_HCTRL_MSK, 16
BME680_HEAT_STAB_MSK, 16
BME680_HUM_REG_SHIFT_VAL, 16
BME680_I_MAX_CORRECTION, 16
BME680_I_MIN_CORRECTION, 16
BME680_I2C_ADDR_PRIMARY, 16
BME680_I2C_ADDR_SECONDARY, 16
BME680_I2C_INTF, 24
bme680_init, 26
bme680_intf, 24
BME680_MAX_OVERFLOW_VAL, 16
BME680_MEM_PAGE_ADDR, 17
BME680_MEM_PAGE_MSK, 17
BME680_MEM_PAGE0, 16
BME680_MEM_PAGE1, 16
BME680_MODE_MSK, 17
BME680_NBCONV_MAX, 17
BME680_NBCONV_MIN, 17
BME680_NBCONV_MSK, 17
BME680_NBCONV_SEL, 17
BME680_NEW_DATA_MSK, 17
BME680_OK, 17
BME680_OS_16X, 17
BME680_OS_1X, 17
BME680_OS_2X, 17
BME680_OS_4X, 18
BME680_OS_8X, 18
BME680_OS_NONE, 18
BME680_OSH_MSK, 18
BME680_OSH_SEL, 18
BME680_OSP_MSK, 18
BME680_OSP_POS, 18
BME680_OSP_SEL, 18
BME680_OST_MSK, 18
BME680_OST_POS, 18
BME680_OST_SEL, 18
BME680_P1_LSB_REG, 19
BME680_P1_MSB_REG, 19
BME680_P10_REG, 18
BME680_P2_LSB_REG, 19
BME680_P2_MSB_REG, 19
BME680_P3_REG, 19
BME680_P4_LSB_REG, 19
BME680_P4_MSB_REG, 19
BME680_P5_LSB_REG, 19
BME680_P5_MSB_REG, 19
BME680_P6_REG, 19
BME680_P7_REG, 19
BME680_P8_LSB_REG, 19
BME680_P8_MSB_REG, 20
BME680_P9_LSB_REG, 20
BME680_P9_MSB_REG, 20
BME680_POLL_PERIOD_MS, 20
BME680_REG_BUFFER_LENGTH, 20
BME680_REG_FILTER_INDEX, 20
BME680_REG_HCTRL_INDEX, 20
BME680_REG_HUM_INDEX, 20
BME680_REG_NBCONV_INDEX, 20
BME680_REG_PRES_INDEX, 20
BME680_REG_RUN_GAS_INDEX, 20
BME680_REG_TEMP_INDEX, 20
BME680_RES_HEAT0_ADDR, 21
BME680_RESET_PERIOD, 21
BME680_RHRANGE_MSK, 21
BME680_RSERROR_MSK, 21
BME680_RUN_GAS_DISABLE, 21
BME680_RUN_GAS_ENABLE, 21
BME680_RUN_GAS_MSK, 21
BME680_RUN_GAS_POS, 21
BME680_RUN_GAS_SEL, 21
BME680_SET_BITS, 21
BME680_SET_BITS_POS_0, 21
bme680_set_profile_dur, 26
bme680_set_regs, 26
bme680_set_sensor_mode, 27
bme680_set_sensor_settings, 27
BME680_SLEEP_MODE, 22
bme680_soft_reset, 28
BME680_SOFT_RESET_ADDR, 22
BME680_SOFT_RESET_CMD, 22
BME680_SPI_INTF, 24
BME680_SPI_RD_MSK, 22
BME680_SPI_WR_MSK, 22
BME680_T1_LSB_REG, 22
BME680_T1_MSB_REG, 22
BME680_T2_LSB_REG, 22
BME680_T2_MSB_REG, 22
BME680_T3_REG, 22
BME680_TMP_BUFFER_LENGTH, 22
BME680_W_DEFINE_PWR_MODE, 22
BME680_W_NO_NEW_DATA, 23
INT16_C, 23
INT32_C, 23
INT64_C, 23
INT8_C, 23
NULL, 23
UINT16_C, 23
UINT32_C, 23

UINT64_C, 23
UINT8_C, 23
SERCOM4_RS232.c
ARRAY_PINCFG1, 64
ARRAY_PMUX1, 64
delayMs, 63
UART_init, 63
UART_read, 63
UART_write, 63
SERCOM4_RS232.h
delayMs, 65
UART_init, 65
UART_read, 65
UART_write, 65
spi_transfer
main.c, 61
status
bme680_field_data, 36
t_fine
bme680_calib_data, 32
temperature
bme680_field_data, 37
tph
main.c, 61
tph_sett
bme680_dev, 34
UART_init
SERCOM4_RS232.c, 63
SERCOM4_RS232.h, 65
UART_read
SERCOM4_RS232.c, 63
SERCOM4_RS232.h, 65
UART_write
SERCOM4_RS232.c, 63
SERCOM4_RS232.h, 65
UINT16_C
SENSOR API, 23
UINT32_C
SENSOR API, 23
UINT64_C
SENSOR API, 23
UINT8_C
SENSOR API, 23
update_lcd_dog
lcd_dog_driver.c, 55
lcd_dog_driver.h, 58
user_delay_ms
main.c, 61
user_spi_read
main.c, 61
user_spi_write
main.c, 62
v_delay
lcd_dog_driver.c, 55
lcd_dog_driver.h, 58
write
bme680_dev, 34



Title	Aaron Varghese & Anthony Brangaitis	
Size A	Document Number <Doc>	Rev <RevCo
Date: Monday, May 11, 2020	Sheet 1 of 1	