**A REPORT ON**

**DRIVER FOR BME280**

BY

**JEEVARAAM K 2020H1400216H**

**RISHI S PHAYE 2020H1400230H**

**M.E. EMBEDDED SYSTEMS**

Prepared in partial fulfilment of the

**(G547)**

**DEVICE DRIVERS**



**BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI**

**(DECEMBER 2021)**

**LIST OF FIGURES:**

Figure 1: BME280 sensor…………………………………………………………..4

Figure 2: Memory map for BME280……………………………………………………..5

Figure 3: Compensation parameter storage,naming and data type……………………….6

Figure 4: Connection with BME280 with raspberry pi 4…………………………………7

Figure 5: Pin diagram of raspberry pi …………………………………………………….7

Figure 6: Actual connection with BME280……………………………………………….8

**CONTENTS**

1. Summary……………………………………………….………………....4

2. Hardware design……………………………………………………...…..7

3. Kernel Space Driver and its build process………………………………..9

4. User space application build process and usage…………………………17

5. IOCTL Header file……………………………………………………….20

6. Makefile……………………………………………………………….…21

7. Makefile commands……………………………………………………...23

**SUMMARY:**

* This project collects the data(temperature, humidity and pressure) from the sensor BME280 and stores it in a log file in given intervals.
* The user space application is capable of receiving data from the kernel using IOCTL call
* The received data is logged into a log file in the same directory.
* In kernel, once the IOCTL call is received the kernel initiates the I2C read from the sensor applies compensation algorithm to the raw values and transfers them to the user space request; the process is same for all the three parameters.

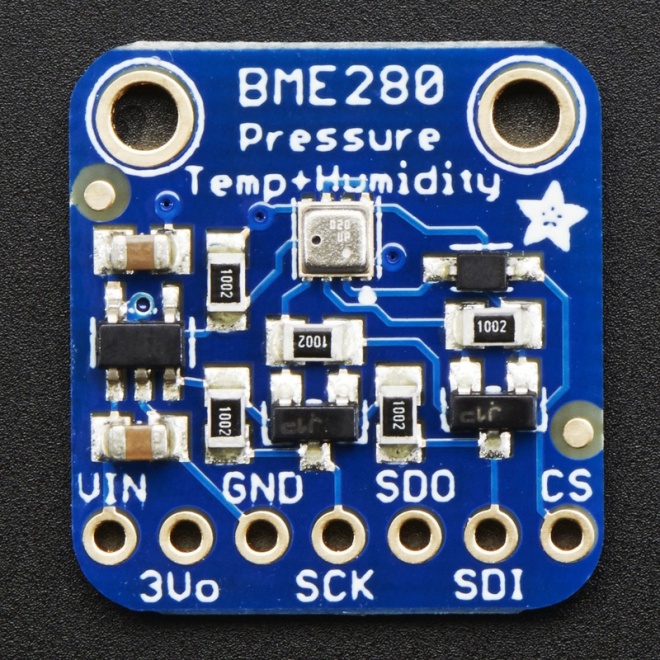


Figure 1: BME280 sensor

**Register description:**

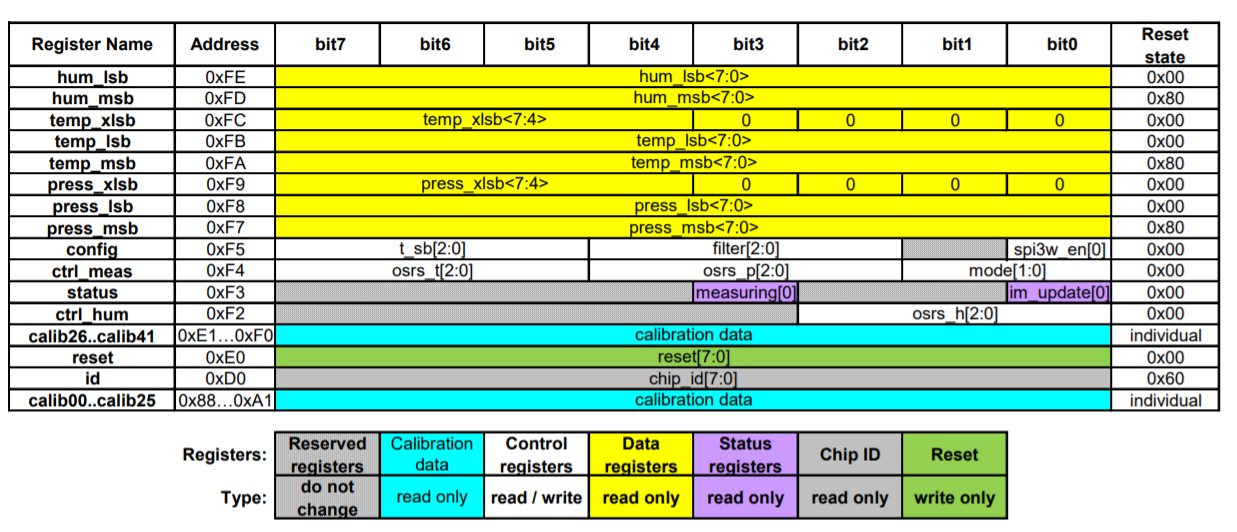


Figure 2 :Memory map for BME280

* The entire communication with the device is performed by reading from and writing to registers, the above memory map is used for all those purposes.
* The Following points must be noted:
* For humidity the register address 0xFE and 0xFD are being used to get the raw data value.
* For temperature the register address 0xFC, 0xFB and 0xFA are being used to get the raw data.
* For pressure the register address starting 0xF9, 0xF8 and 0xF7 are being used to get the raw data.
* Config(0xF5) and ctrl\_meas(0xF4) are configured with respective parameters for proper sensing and operation of sensor.

**READING PARAMETERS:**

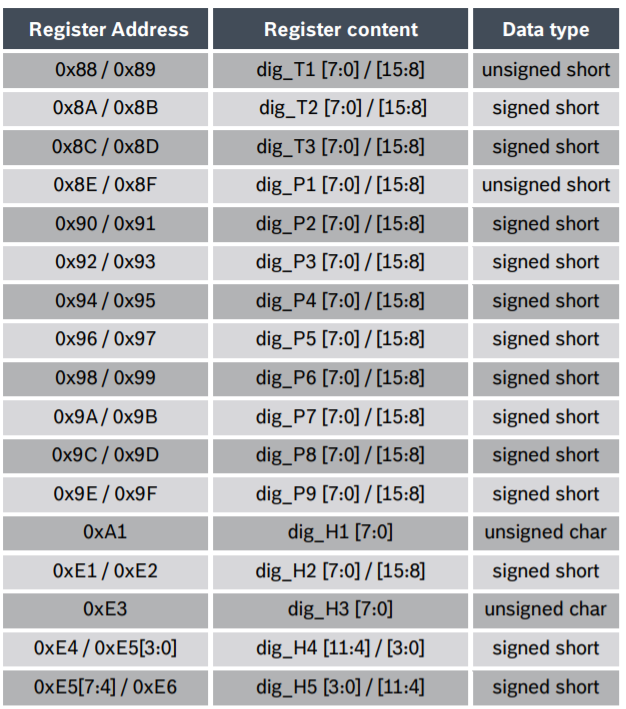




Figure 3: Compensation parameter storage,naming and data type

* The parameter are programmed inside the non-volatile memory during production and we can’t alter it.
* Each compensation word is a 16-bit signed or unsigned integer value stored in 2’s complement form.
* 2 words must always be combined in order to represent the compensation word, as the memory is organized into 8-bit words.
* These compensation parameters are read by the kernel during initialization and kept in global space of kernel module for further calculations during runtime.

**HARDWARE SCHEMATIC:**

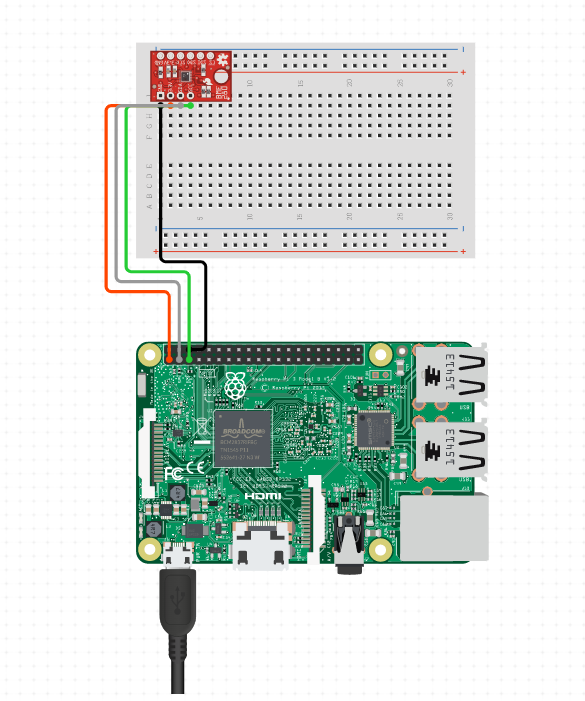


Figure 4: Connection with BME280 with raspberry pi 4

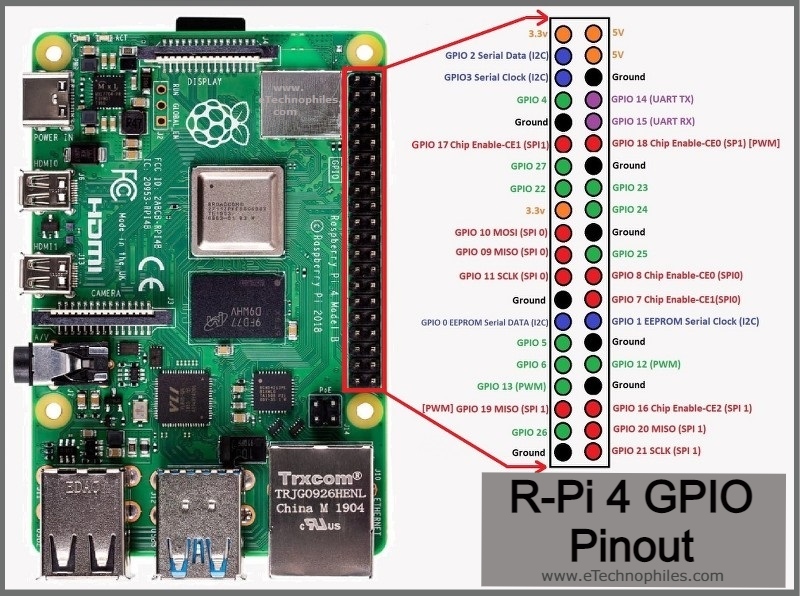


Figure 5: Pin diagram of raspberry pi

The connections for our code will be as explained below:

* 3.3V is given to pin 1 of Raspberry pi and the Vdd pin of BME280
* Pin number 3 of Raspberry Pi is connected to the SDA pin of BME280.
* Pin number 5 of Raspberry Pi is connected to the SCL pin of BME280.
* Pin number 6 of Raspberry Pi and GND pin of BME280 are connected to ground.



Figure 6: Actual connection with BME280.

**Kernel Space Driver and it’s build process:**

/\* General Includes for Kernel Compilation \*/

#include <linux/module.h>

#include <linux/init.h>

#include <linux/fs.h>

#include <linux/version.h>

#include <linux/cdev.h>

#include <linux/uaccess.h>

#include <linux/slab.h>

#include <linux/i2c.h>

#include <linux/kernel.h>

/\* IOCTL Call Definition Header \*/

#include "config.h"

#define DRIVER\_NAME "bme280" //Name of the driver to be created

#define DRIVER\_CLASS "bme280Class" //Name of the class to be used for driver

/\* User-defined data types for Pressure compensation algorithm \*/

typedef uint32\_t BME280\_U32\_t;

typedef int32\_t BME280\_S32\_t;

static struct i2c\_adapter \* bme\_i2c\_adapter = NULL; //I2C Adapter

static struct i2c\_client \* bme280\_i2c\_client = NULL; //I2C Client

/\* Variables for holding Trim parameters from the sensor \*/

int32\_t calib\_T1,calib\_T2,calib\_T3,calib\_Tfine;

int32\_t calib\_P1,calib\_P2,calib\_P3,calib\_P4,calib\_P5,calib\_P6,calib\_P7,calib\_P8,calib\_P9;;

int32\_t calib\_H1,calib\_H2,calib\_H3,calib\_H4,calib\_H5,calib\_H6;

/\* Driver Metadata \*/

MODULE\_AUTHOR("Jeevaraam K, Rishi S Phaye");

MODULE\_LICENSE("GPL");

MODULE\_DESCRIPTION("BME280 Sensor Kernel Driver");

#define I2C\_BUS\_USED       1 //I2C Bus used for Communication

#define BME280\_SENSOR\_NAME       "BME280" //Sensor name for Metadata

#define BME280\_SLAVE\_ADDRESS    0x77 //I2C Address of Sensor used

/\* Device ID Structure Initialization \*/

static const struct i2c\_device\_id bme280\_dev\_id[]={

{ BME280\_SENSOR\_NAME, 0},

{ }

};

/\* I2C Driver Structure Initialization \*/

static struct i2c\_driver bme280\_driver = {

 .driver = {

   .name = BME280\_SENSOR\_NAME,

   .owner = THIS\_MODULE

  }

};

/\* Board and Device Mapping Structure \*/

static struct i2c\_board\_info bme280\_i2c\_board\_info = {

 I2C\_BOARD\_INFO(BME280\_SENSOR\_NAME, BME280\_SLAVE\_ADDRESS)

};

static dev\_t bme280\_device\_number; //Major and Minor Number holding variable

static struct class \*bme280\_class; //Sensor class variable

static struct cdev bme280\_device; //Char device for Sensor

/\*BME280 Compensation Routines \*/

/\* Compensation Routine for reading and Converting Raw Sensor value to Humidity Values \*/

static uint32\_t read\_humidity(void)

{

    int32\_t var1;

    int32\_t var2;

    int32\_t var3;

    int32\_t var4;

    int32\_t var5;

    uint32\_t humidity;

    uint32\_t humidity\_max = 102400;

    uint8\_t lsb,msb;

    uint32\_t current\_humidity;

    /\*Read Raw values from sensor \*/

    lsb = i2c\_smbus\_read\_byte\_data(bme280\_i2c\_client, 0xFE);

    msb = i2c\_smbus\_read\_byte\_data(bme280\_i2c\_client, 0xFD);

    current\_humidity = ((uint32\_t)msb<<8) | ((uint32\_t)lsb);

    /\* Compensation Algorithm \*/

    var1 = calib\_Tfine - ((int32\_t)76800);

    var2 = (int32\_t)(current\_humidity \* 16384);

    var3 = (int32\_t)(((int32\_t)calib\_H4) \* 1048576);

    var4 = ((int32\_t)calib\_H5) \* var1;

    var5 = (((var2 - var3) - var4) + (int32\_t)16384) / 32768;

    var2 = (var1 \* ((int32\_t)calib\_H6)) / 1024;

    var3 = (var1 \* ((int32\_t)calib\_H3)) / 2048;

    var4 = ((var2 \* (var3 + (int32\_t)32768)) / 1024) + (int32\_t)2097152;

    var2 = ((var4 \* ((int32\_t)calib\_H2)) + 8192) / 16384;

    var3 = var5 \* var2;

    var4 = ((var3 / 32768) \* (var3 / 32768)) / 128;

    var5 = var3 - ((var4 \* ((int32\_t)calib\_H1)) / 16);

    var5 = (var5 < 0 ? 0 : var5);

    var5 = (var5 > 419430400 ? 419430400 : var5);

    humidity = (uint32\_t)(var5 / 4096);

    if (humidity > humidity\_max)

    {

        humidity = humidity\_max;

    }

    return humidity;

}

/\* Compensation Routine for reading and Converting Raw Sensor value to Pressure Values \*/

static uint32\_t read\_pressure(void)

{

 /\*int32\_t var1;

 int32\_t var2;

 int32\_t var3;

 int32\_t var4;

 uint32\_t var5;\*/

 uint32\_t current\_pressure;

 uint8\_t lsb,msb,xlsb;

 BME280\_S32\_t var1, var2;

 BME280\_U32\_t p;

 /\* Read raw values from Sensor \*/

 lsb = i2c\_smbus\_read\_byte\_data(bme280\_i2c\_client,0xF8);

 msb = i2c\_smbus\_read\_byte\_data(bme280\_i2c\_client,0xF7);

 xlsb = i2c\_smbus\_read\_byte\_data(bme280\_i2c\_client, 0xF9);

 current\_pressure = (((uint32\_t)msb)<<12) | ((uint32\_t)lsb<<4) | ((uint32\_t)xlsb>>4);

// current\_pressure = ( (current\_pressure<<4) | ( (uint32\_t) (xlsb&0xF0) >>4 ) );

 printk("Pressure:%x\n",current\_pressure);

 /\* Compensation Algorithm \*/

 var1 = (((BME280\_S32\_t)calib\_Tfine)>>1) - (BME280\_S32\_t)64000;

 var2 = (((var1>>2) \* (var1>>2)) >> 11 ) \* ((BME280\_S32\_t)calib\_P6);

 var2 = var2 + ((var1\*((BME280\_S32\_t)calib\_P5))<<1);

 var2 = (var2>>2)+(((BME280\_S32\_t)calib\_P4)<<16);

 var1 = (((calib\_P3 \* (((var1>>2) \* (var1>>2)) >> 13 )) >> 3) + ((((BME280\_S32\_t)calib\_P2) \* var1)>>1))>>18;

 var1 =((((32768+var1))\*((BME280\_S32\_t)calib\_P1))>>15);

 if (var1 == 0)

 {

  return 0; // avoid exception caused by division by zero

 }

 p = (((BME280\_U32\_t)(((BME280\_S32\_t)1048576)-current\_pressure)-(var2>>12)))\*3125;

 if (p < 0x80000000)

 {

  p = (p << 1) / ((BME280\_U32\_t)var1);

 }

 else

 {

  p = (p / (BME280\_U32\_t)var1) \* 2;

 }

 var1 = (((BME280\_S32\_t)calib\_P9) \* ((BME280\_S32\_t)(((p>>3) \* (p>>3))>>13)))>>12;

 var2 = (((BME280\_S32\_t)(p>>2)) \* ((BME280\_S32\_t)calib\_P8))>>13;

 p = (BME280\_U32\_t)((BME280\_S32\_t)p + ((var1 + var2 + calib\_P7) >> 4));

 return p;

static int32\_t read\_temperature(void)

{

 int32\_t var1,var2;

 uint8\_t aTemp,bTemp,cTemp;

 int32\_t temperature;

 int32\_t finalTemp;

 /\* Read raw values from Sensor \*/

 aTemp = i2c\_smbus\_read\_byte\_data(bme280\_i2c\_client, 0xFA);

 bTemp = i2c\_smbus\_read\_byte\_data(bme280\_i2c\_client, 0xFB);

 cTemp = i2c\_smbus\_read\_byte\_data(bme280\_i2c\_client, 0xFC);

 /\* Compensation Algorithm \*/

 temperature = (((int32\_t)aTemp) << 12) | (((int32\_t)bTemp) << 4) | (((int32\_t)cTemp) >> 4);

 var1 = (int32\_t)((temperature / 8) - ((int32\_t)calib\_T1 \* 2));

 var1 = (var1 \* ((int32\_t)calib\_T2)) / 2048;

 var2 = (int32\_t)((temperature / 16) - ((int32\_t)calib\_T1));

 var2 = (((var2 \* var2) / 4096) \* ((int32\_t)calib\_T3)) / 16384;

 calib\_Tfine = var1 + var2;

 finalTemp = ((var1 + var2) \* 5 + 128) / 256;

 printk("Temp:%d\n",finalTemp);

 printk("Tfine:%d\n",calib\_Tfine);

 return finalTemp;

}

/\* Direct Call Read routine for I2C device \*/

static ssize\_t bme280\_driver\_read(struct file \*File, char \_\_user \*user\_buffer, size\_t count, loff\_t \*offs)

{

 int to\_copy, not\_copied, delta;

 char out\_string[30];

 int32\_t temperature;

 int32\_t pressure;

 int32\_t humidity;

 to\_copy = min(sizeof(out\_string), count);

 /\* Read all 3 values \*/

 temperature = read\_temperature();

 pressure = read\_pressure();

 humidity = read\_humidity();

 /\* Transfer the read values \*/

 snprintf(out\_string, sizeof(out\_string), "T:%d,P:%d,H:%d\n",temperature,pressure,humidity);

 not\_copied = copy\_to\_user(user\_buffer, out\_string, to\_copy);

 delta = to\_copy - not\_copied;

 return delta;

}

/\* IOCTL Call routine \*/

long ioctl\_dev(struct file \*file, unsigned int ioctl\_num, unsigned long ioctl\_param)

{

 switch(ioctl\_num)

 {

  /\* IOCTL Call for Temperature \*/

  case IOCTL\_TEMPERATURE:

  put\_user(read\_temperature(),(int32\_t\*)ioctl\_param);

  break;

  /\* IOCTL Call for Pressure \*/

  case IOCTL\_PRESSURE:

  put\_user(read\_pressure(),(int32\_t\*)ioctl\_param);

  break;

  /\* IOCTL Call for Humidity \*/

  case IOCTL\_HUMIDITY:

  put\_user(read\_humidity(),(int32\_t\*)ioctl\_param);

  break;

 }

return 0;

}

/\* Driver open routine \*/

static int bme280\_driver\_open(struct inode \*deviceFile, struct file \*instance)

{

 printk("Driver Open\n");

 return 0;

}

/\* Driver close routine \*/

static int bme280\_driver\_close(struct inode \*deviceFile, struct file \*instance)

{

 printk("Driver Close\n");

 return 0;

}

/\* File Operations structure for Open, Close, Read and IOCTL routines linkage \*/

static struct file\_operations fops  = {

 .owner = THIS\_MODULE,

 .open = bme280\_driver\_open,

 .release = bme280\_driver\_close,

 .unlocked\_ioctl = ioctl\_dev,

 .read = bme280\_driver\_read,

};

/\* Driver Initialization Routine \*/

static int \_\_init bme280Init(void)

{

 int ret =-1;

 int32\_t tempH4a,tempH4b,tempH5a;

 u8 id;

 printk("Driver Init\n");

 /\* Allocating Random Major number from Linux with one device \*/

 if ( alloc\_chrdev\_region(&bme280\_device\_number, 0, 1, DRIVER\_NAME) < 0)

 {

  printk("Device Number could not be allocated!\n");

 }

 printk("Device Number %d was registered\n", bme280\_device\_number);

 /\* Create Device Class \*/

 if ((bme280\_class = class\_create(THIS\_MODULE, DRIVER\_CLASS)) == NULL)

 {

  printk("Device Class can not be created!\n");

  goto ClassError;

 }

 /\* Create Device file \*/

 if (device\_create(bme280\_class, NULL, bme280\_device\_number, NULL, DRIVER\_NAME) == NULL)

 {

  printk("Can not create device file!\n");

  goto FileError;

 }

 /\* Initialize Device file \*/

 cdev\_init(&bme280\_device, &fops);

 /\* register device to kernel \*/

 if (cdev\_add(&bme280\_device, bme280\_device\_number, 1) == -1)

 {

  printk("Registering of device to kernel failed!\n");

  goto KernelError;

 }

 /\* Locking I2C Bus Adapter \*/

 bme\_i2c\_adapter = i2c\_get\_adapter(I2C\_BUS\_USED);

 if(bme\_i2c\_adapter != NULL)

 {

  /\* I2C Client Static Link \*/

  bme280\_i2c\_client = i2c\_new\_client\_device(bme\_i2c\_adapter, &bme280\_i2c\_board\_info);

  if(bme280\_i2c\_client != NULL)

  {

   /\* Add I2C Driver \*/

   if(i2c\_add\_driver(&bme280\_driver) != -1)

   {

    ret = 0;

   }

   else

    printk("Can't add driver...\n");

  }

  /\* Releasing I2C Bus Adapter \*/

  i2c\_put\_adapter(bme\_i2c\_adapter);

 }

 printk("BME280 Driver Init\n");

 /\* Read Device ID for Communication Verification \*/

 id = i2c\_smbus\_read\_byte\_data(bme280\_i2c\_client, 0xD0);

 printk("ID: 0x%x\n",id);

 /\* Read Trim Values from the Sensor and store in Global Kernel Variables \*/

 calib\_T1 = i2c\_smbus\_read\_word\_data(bme280\_i2c\_client,0x88);

 calib\_T2 = i2c\_smbus\_read\_word\_data(bme280\_i2c\_client,0x8A);

 calib\_T3 = i2c\_smbus\_read\_word\_data(bme280\_i2c\_client,0x8C);

 calib\_P1 = i2c\_smbus\_read\_word\_data(bme280\_i2c\_client,0x8E);

 calib\_P2 = i2c\_smbus\_read\_word\_data(bme280\_i2c\_client,0x90);

 calib\_P3 = i2c\_smbus\_read\_word\_data(bme280\_i2c\_client,0x92);

 calib\_P4 = i2c\_smbus\_read\_word\_data(bme280\_i2c\_client,0x94);

 calib\_P5 = i2c\_smbus\_read\_word\_data(bme280\_i2c\_client,0x96);

 calib\_P6 = i2c\_smbus\_read\_word\_data(bme280\_i2c\_client,0x98);

 calib\_P7 = i2c\_smbus\_read\_word\_data(bme280\_i2c\_client,0x9A);

 calib\_P8 = i2c\_smbus\_read\_word\_data(bme280\_i2c\_client,0x9C);

 calib\_P9 = i2c\_smbus\_read\_word\_data(bme280\_i2c\_client,0x9E);

 calib\_H1 = i2c\_smbus\_read\_byte\_data(bme280\_i2c\_client,0xA1);

 calib\_H2 = i2c\_smbus\_read\_word\_data(bme280\_i2c\_client,0xE1);

 calib\_H3 = i2c\_smbus\_read\_byte\_data(bme280\_i2c\_client,0xE3);

 tempH4a = i2c\_smbus\_read\_byte\_data(bme280\_i2c\_client,0xE4);

 tempH4b = i2c\_smbus\_read\_byte\_data(bme280\_i2c\_client,0xE5);

 tempH5a = i2c\_smbus\_read\_byte\_data(bme280\_i2c\_client,0xE6);

 calib\_H6 = i2c\_smbus\_read\_byte\_data(bme280\_i2c\_client,0xE7);

 calib\_H4 = (tempH4a<<4) | (tempH4b&0x0F);

 calib\_H5 = (tempH5a<<4) | ((tempH4b&0xF0)>>4);

 if(calib\_T2 > 32767)

 {

  calib\_T2 -= 65536;

 }

 if(calib\_T3 > 32767)

 {

  calib\_T3 -= 65536;

 }

 printk("BME280 Calib Values\n");

 printk("T1:%d\nT2:%d\nT3:%d\nP1:%d\nP2:%d\nP3:%d\nP4:%d\nP5:%d\nP6:%d\nP7:%d\nP8:%d\nP9:%d\n",calib\_T1,calib\_T2,calib\_T3,calib\_P1,calib\_P2,calib\_P3,calib\_P4,calib\_P5,calib\_P6,calib\_P7,calib\_P8,calib\_P9);

 /\* Sensor Initilatization Sequence \*/

 i2c\_smbus\_write\_byte\_data(bme280\_i2c\_client, 0xf5, 5<<5);

 i2c\_smbus\_write\_byte\_data(bme280\_i2c\_client, 0xf4, ((5<<5) | (5<<2) | (3<<0)));

 return ret;

 /\* Error Handling on Failure \*/

KernelError:

        device\_destroy(bme280\_class, bme280\_device\_number);

FileError:

        class\_destroy(bme280\_class);

ClassError:

        unregister\_chrdev(bme280\_device\_number, DRIVER\_NAME);

        return (-1);

}

/\* Driver Uninitialization routine \*/

static void \_\_exit bme280Exit(void)

{

 printk("bme280\_deviceDriver - Goodbye, Kernel!\n");

 i2c\_unregister\_device(bme280\_i2c\_client);

 i2c\_del\_driver(&bme280\_driver);

 cdev\_del(&bme280\_device);

 device\_destroy(bme280\_class, bme280\_device\_number);

 class\_destroy(bme280\_class);

 unregister\_chrdev\_region(bme280\_device\_number, 1);

}

/\*Linking Init and Exit Functions to Driver Structure \*/

module\_init(bme280Init);

module\_exit(bme280Exit);

**User space application build process and usage**

/\* General Includes \*/

#include<stdio.h>

#include<stdlib.h>

#include<fcntl.h>

#include<unistd.h>

#include<sys/ioctl.h>

#include<signal.h>

#include<unistd.h>

#include<time.h>

/\* IOCTL Interface Definition \*/

#include "config.h"

/\* Char Device Variable \*/

int file\_desc;

/\* Log File \*/

FILE \*fptr;

/\* SIGINT Signal Handler \*/

void exitHandler(int sig\_num)

{

 signal(SIGINT, exitHandler);

 printf("\nExit Initiated");

 close(file\_desc);

 fprintf(fptr,"\n");

 fclose(fptr);

 printf("\nShutdown Procedures Complete.Bye\n");

 exit(0);

}

/\* IOCTL Userspace Call for Pressure \*/

int ioctl\_pressure(int file\_desc, int32\_t \*msg)

{

 int ret\_val;

 ret\_val = ioctl(file\_desc, IOCTL\_PRESSURE,msg);

 return ret\_val;

}

/\* IOCTL Userspace Call for Temperature \*/

int ioctl\_temperature(int file\_desc, int32\_t \*msg)

{

 int ret\_val;

 ret\_val = ioctl(file\_desc, IOCTL\_TEMPERATURE,msg);

 return 0;

}

/\* IOCTL Userspace Call for Humidity \*/

int ioctl\_humidity(int file\_desc, int32\_t \*msg)

{

 int ret\_val;

 ret\_val = ioctl(file\_desc, IOCTL\_HUMIDITY,msg);

 return 0;

}

/\* Main Routine \*/

int main(void)

{

 int ret\_val,i;

 int32\_t recv\_msg;

 time\_t ltime;

 struct tm result;

 char time\_string[32];

 float temperature,pressure,humidity;

 /\* Register SIGINT Handler \*/

 signal(SIGINT, exitHandler);

 /\* Open Char Device \*/

 file\_desc = open(DEVICE\_FILE\_NAME,0);

 /\* Log File Open \*/

 fptr = fopen("data.log","a");

 if(file\_desc<0)

 {

  printf("Device Open Failed for %s\n",DEVICE\_FILE\_NAME);

  exit(-1);

 }

 printf("Logging started...\n");

 /\* Infinite Execution Block \*/

 while(1)

 {

  /\* Current Timestame Retrieval \*/

  ltime = time(NULL);

  localtime\_r(&ltime, &result);

  asctime\_r(&result, time\_string);

  i=0;

  /\* Format Timestamp for Log Print \*/

  while(time\_string[i]!='\0')

  {

   if(time\_string[i]=='\n')

    time\_string[i]=0;

   i++;

  }

  /\* Print Timestamp to Log File \*/

  fprintf(fptr, "%s : ",time\_string);

  /\* Retrieve Temperature and Print to Log File \*/

  ioctl\_temperature(file\_desc,&recv\_msg);

  temperature = (float)recv\_msg/100.0;

  fprintf(fptr, "Temperature:%f ",temperature);

  /\* Retrieve Pressure and Print to Log File \*/

  ioctl\_pressure(file\_desc,&recv\_msg);

  pressure = (float)recv\_msg/100.0;

  fprintf(fptr, "Pressure:%f ",pressure);

  /\* Retrieve Humidity and Print to Log File \*/

  ioctl\_humidity(file\_desc,&recv\_msg);

  humidity = (float)recv\_msg/1024.0;

  fprintf(fptr, "Humidity:%f\n", humidity);

  /\* Delay for 5 Seconds \*/

  sleep(5);

 }

}

**IOCTL Header file:**

#ifndef CHAR\_CONFIG\_H

#define CHAR\_CONFIG\_H

#include <linux/ioctl.h>

#define MAGIC\_NUM 225

//IOCTL interface prototypes

#define IOCTL\_TEMPERATURE \_IOWR(MAGIC\_NUM, 0, int32\_t \*)

#define IOCTL\_HUMIDITY \_IOWR(MAGIC\_NUM, 1, int32\_t \*)

#define IOCTL\_PRESSURE \_IOWR(MAGIC\_NUM, 2, int32\_t \*)

//Device file interface

#define DEVICE\_FILE\_NAME "/dev/bme280"

#endif

**Makefile:**

# All the Parameters can be over-ridden while Make call from the Terminal. Default values are assigned to match with the currnet project files.

# Note: The make recepies are designed in such a way assuming the make invoke is done from superuser(root)

# Name of the Userapp file.

USERFILE ?= userapp

# Name of the Kernel File

KERN\_NAME ?= bme280

# Name of the Data Log file used for storing sensor values

DATAFILE ?= data.log

# Default Text Editor for displaying the log file

TEXT\_EDITOR ?= nano

# Kernel Module Name Definition

obj-m := $(KERN\_NAME).o

# Compiles the Kernel and Userapp files

all:

    make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules

    gcc -o $(USERFILE) $(USERFILE).c

# Compiles the Kernel code alone

kern:

    make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules

# Compiles the Userapp alone

user:

    gcc -o $(USERFILE) $(USERFILE).c

# Cleans the directory (Won't remove the log file)

clean:

    make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean

    rm userapp

# Installs the Compiled Kernel Module into Kernel Space

install:

    sudo insmod $(KERN\_NAME).ko

# Removes the loaded module from the Kernel Space

remove:

    sudo rmmod $(KERN\_NAME).ko

# Runs the Userapp Application

run:

    sudo ./$(USERFILE)

# Deletes the Sensor Log generated by Userapp application

log\_clean:

    rm $(DATAFILE)

# Displays the Log generated by the Userapp application

log\_display:

    nano $(DATAFILE)

**MAKE COMMANDS:**

All the Operations are defined as Makefile recepies to make it easier for the execution

Following are the various recepies defined for compilation and execution

1. all - Compiles Kernel and User Space Code
2. kern - Compiles Kernel Space Code
3. user - Compiles User Space Code
4. install - Installs Kernel Module into Kernel Space
5. remove - Removes Kernel Module from Kernel Space
6. log\_display - Displays the Log generated by User Space application
7. log\_delete - Deletes the Log generated by the User Space application
8. log\_clean – will delete the data log file
9. clean – erases the all the executables and compiled files
10. run - Runs the User Space Application