LINUX DRIVER FOR BMP085 PRESSURE SENSOR

Gangula Dilip Reddy, Kunreddy Chandrashekar Reddy, Anurag Gupta

Department of Electrical and Electronics

Birla Institute of Science and Technology Goa Campus

Device drivers are software interface between software applications and hardware devices. Device drivers take on a special role in the Linux kernel. They are distinct "black boxes" that make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works. User activities are performed by means of a set of standardized calls that are independent of the specific driver; mapping those calls to device-specific operations that act on real hardware is then the role of the device driver.

Keywords—Raspberry Pi, Code generation, Linux, Device Driver, BMP085

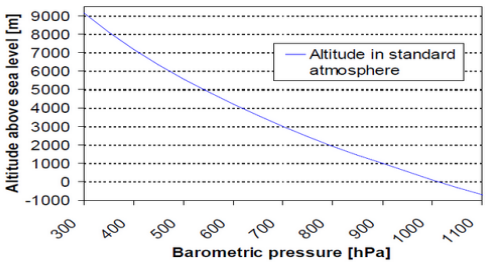
INTRODUCTION

The BMP085 has a digital interface, I^2C to be specific. I^2C is a synchronous two-wire interface, the first wire, SDA, transmits data, while a second wire, SCL, transmits a clock, which is used to keep track of the data.

WHY DO WE MORE CARE ABOUT PRESSURE?

You've probably heard weather reporters going on about low pressure systems and high pressure systems, that's because atmospheric pressure can be directly related to changes in weather. Low pressure typically means cloudier, rainier, snowier, and just generally uglier weather. While high pressure generally means clearer, sunnier weather. This means the BMP085 would be a perfect component for your Weather Prophet robot.

A second, widely applied use for pressure sensors is in altimetry. Barometric pressure has a measurable relationship with altitude, meaning you can use the BMP085 to deduce how high you (or maybe one of your robots) have climbed. At sea level air pressure is on average 1013 hPa, while here in Boulder, CO, at 5184 ft above sea level, average air pressure is about 831.4 hPa. The measuring limits of the BMP085 should allow you to measure pressure at elevations anywhere between -1640 to about 29,000 ft above sea level.

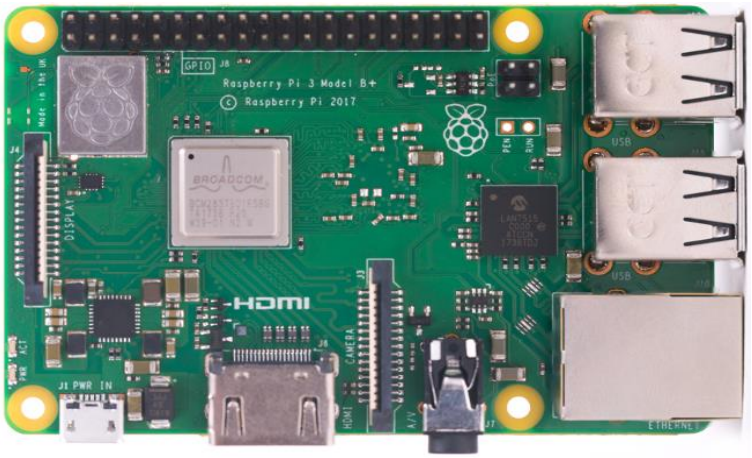


HARDWARE EXPLANATION

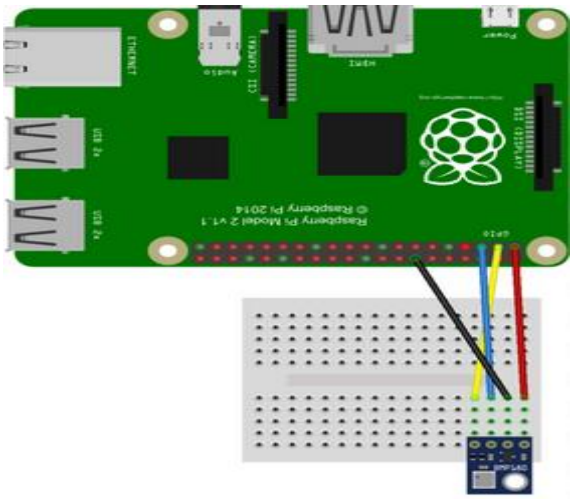
BMP085 Pin	Pin Function
VCC	Power (1.8V-3.6V)
GND	Ground
EOC	End of conversion output
XCLR	Master Clear (low-active)
SCL	Serial Clock I/O
SDA	Serial Data I/O

RASPBERRY PI

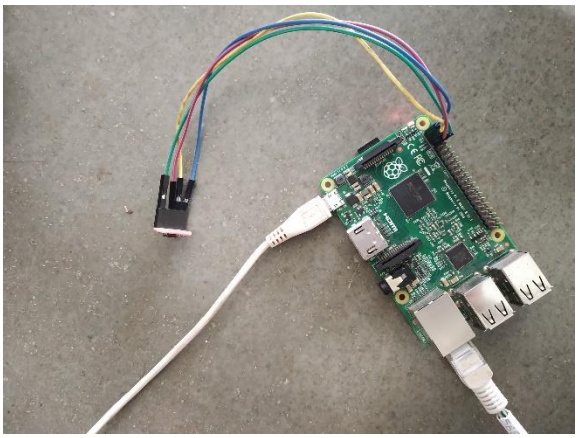
The Raspberry Pi is a very cheap computer that runs Linux, but it also provides a set of GPIO (general purpose input/output) pins that allow you to control electronic components for physical computing



SCHEMATIC DIAGRAM



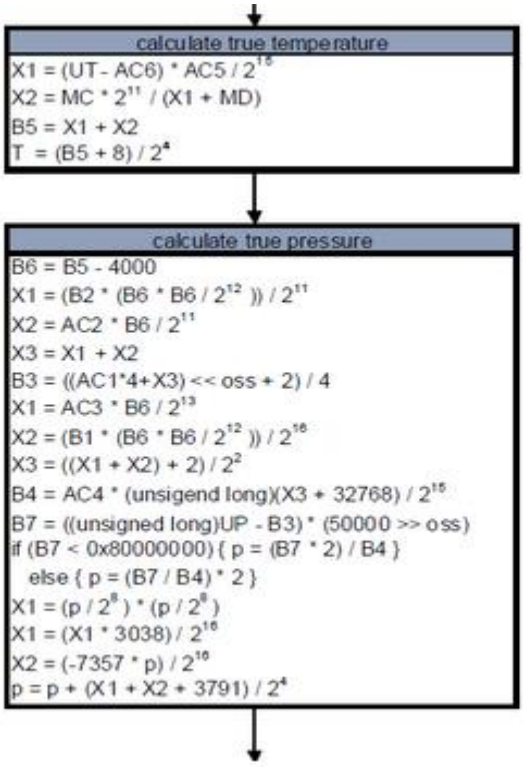
ACTUAL SCHEMATIC



CALCULATING TEMPERATURE AND PRESSURE

Now, if you've glanced through the BMP085 datasheet, you may have noticed some very cryptic variables, and a lot of ugly math involving them. The BMP085 is calibrated in the factory, and left with a series of eleven 16-bit calibration coefficients stored in the device's EEPROM. These variables all play a small role in calculating the absolute pressure. They need to be read just once, at the start of the program, and stored for later use. We can use the I^2C functions above to read and store each of the calibration coefficients.

Once we've read the calibration values, we just need two more variables in order to calculate temperature and pressure. 'ut' and 'up' are the uncompensated temperature and pressure values, they're our starting point for finding the actual temperature and pressure. Every time we want to measure temperature or pressure, we need to first find out what these values are. The uncompensated temperature value is an unsigned 16-bit (int) number while 'up' is an unsigned 32-bit number (long).



FUNCTION TO GET CALIBRATION DATA

```c
static s32 bmp085_read_calibration_data(struct bmp085_data *data)
{
    u16 tmp[BMP085_CALIBRATION_DATA_LENGTH];
    struct bmp085_calibration_data *cali = &(data->calibration);
    s32 status = regmap_bulk_read(data->regmap,
                BMP085_CALIBRATION_DATA_START, (u8 *)tmp,
                (BMP085_CALIBRATION_DATA_LENGTH << 1));
    if (status < 0)
        return status;

    cali->AC1 =  be16_to_cpu(tmp[0]);
    cali->AC2 =  be16_to_cpu(tmp[1]);
    cali->AC3 =  be16_to_cpu(tmp[2]);
    cali->AC4 =  be16_to_cpu(tmp[3]);
    cali->AC5 =  be16_to_cpu(tmp[4]);
    cali->AC6 = be16_to_cpu(tmp[5]);
    cali->B1 = be16_to_cpu(tmp[6]);
    cali->B2 = be16_to_cpu(tmp[7]);
    cali->MB = be16_to_cpu(tmp[8]);
    cali->MC = be16_to_cpu(tmp[9]);
    cali->MD = be16_to_cpu(tmp[10]);
    return 0;
}
```

FUNCTION TO GET RAW TEMPERATURE

```
static s32 bmp085_update_raw_temperature(struct bmp085_data *data)
{
    u16 tmp;
    s32 status;
    mutex_lock(&data->lock);
    init_completion(&data->done);
    status = regmap_write(data->regmap, BMP085_CTRL_REG,
                BMP085_TEMP_MEASUREMENT);
    if (status < 0) {
        dev_err(data->dev,
            "Error while requesting temperature measurement.\n");
        goto exit;
    }
    wait_for_completion_timeout(&data->done, 1 + msecs_to_jiffies(
                        BMP085_TEMP_CONVERSION_TIME));

    status = regmap_bulk_read(data->regmap, BMP085_CONVERSION_REGISTER_MSB,
                &tmp, sizeof(tmp));
    if (status < 0) {
        dev_err(data->dev,
            "Error while reading temperature measurement result\n");
        goto exit;
    }
    data->raw_temperature = be16_to_cpu(tmp);
    data->last_temp_measurement = jiffies;
    status = 0; /* everything ok, return 0 */
exit:
    mutex_unlock(&data->lock);
    return status;
}
```

FUNCTION T0 GET RAW PRESSURE

```
static s32 bmp085_update_raw_pressure(struct bmp085_data *data)
{
    u32 tmp = 0;
    s32 status;

    mutex_lock(&data->lock);

    init_completion(&data->done);

    status = regmap_write(data->regmap, BMP085_CTRL_REG,
            BMP085_PRESSURE_MEASUREMENT +
            (data->oversampling_setting << 6));
    if (status < 0) {
        dev_err(data->dev,
            "Error while requesting pressure measurement.\n");
        goto exit;
    }

    /* wait for the end of conversion */
    wait_for_completion_timeout(&data->done, 1 + msecs_to_jiffies(
                    2+(3 << data->oversampling_setting)));
    /* copy data into a u32 (4 bytes), but skip the first byte. */
    status = regmap_bulk_read(data->regmap, BMP085_CONVERSION_REGISTER_MSB,
                ((u8 *)&tmp)+1, 3);
    if (status < 0) {
        dev_err(data->dev,
            "Error while reading pressure measurement results\n");
        goto exit;
    }
    data->raw_pressure = be32_to_cpu((tmp));
    data->raw_pressure >>= (8-data->oversampling_setting);
    status = 0; /* everything ok, return 0 */

exit:
    mutex_unlock(&data->lock);
    return status;
}
```

REGISTER MAPPING IN LINUX

Linux is divided into many sub-systems in order to factor out common code in different parts and to simplify driver development, which helps in code maintenance.

Linux has sub-systems such as I2C and SPI, which are used to connect to devices that reside on these buses. Both these buses have the common function of reading and writing registers from the devices connected to them. So the code to read and write these registers, cache the value of the registers, etc, will have to be present in both these sub-systems. This causes redundant code to be present in all sub-systems that have this register read and write functionality. To avoid this and to factor out common code, as well as for easy driver maintenance and development, Linux developers introduced a new kernel API from version 3.1, which is called regmap. This infrastructure was previously present in the Linux AsoC (ALSA) sub-system, but has now been made available to entire Linux through the regmap API.

Earlier, if a driver was written for a device residing on the SPI bus, then the driver directly used the SPI bus read and write calls from the SPI sub-system to talk to the device. Now, it uses the regmap API to do so. The regmap sub-system takes care of calling the relevant calls of the SPI sub-system. So two devices—one residing on the I2C bus and one on the SPI bus—will have the same regmap read and write calls to talk to their respective devices on the bus.

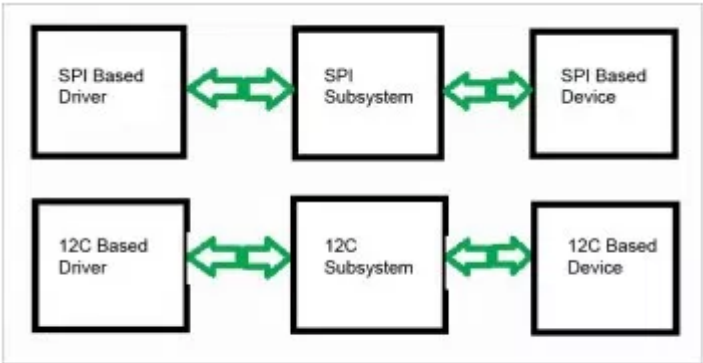


Figure 1: I2C and SPI driver before regmap

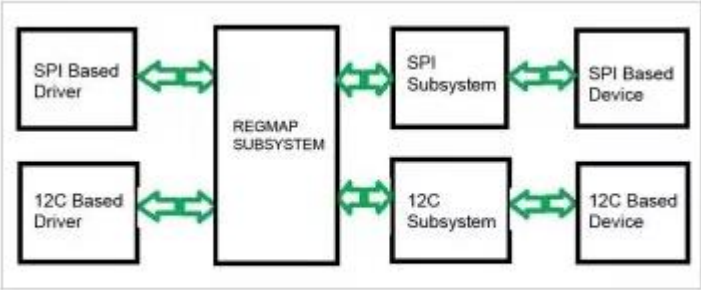


Figure 2: I2C and SPI driver after regmap

Initialisation routines:

The following routine initialises regmap data structures based on the SPI configuration:

```
struct regmap * devm_regmap_init_spi(struct spi_device *spi, const
struct regmap_config);
```

The following routine initialises regmap data structures based on the I2C configuration:

```
struct regmap * devm_regmap_init_i2c(struct i2c_client *i2c, const
struct regmap_config);
```

PROCEDURE:

1) Open the project folder and type → make all
2) Once got compiled successfully then type → sudo make install
3) To insert module → sudo insmod bmp085.ko , sudo insmod bmp085-i2c.ko , sudo insmod bmp085-probe.ko
4) To confirm whether modules are inserted are not type → lsmod
5) sudo depmod -a
6) sudo modprobe bmp085 bmp085-i2c bmp085-probe
7) After the files are created in /sys/bus/i2c/drivers/bmp085/1-0077
8) By using cat we can fetch the details from the file
9) cat /sys/bus/i2c/drivers/bmp085/1-0077/pressure0_input →to read pressure
10) cat /sys/bus/i2c/drivers/bmp085/1-0077/temp0_input → to read temperature

CONCLUSION

In this project, the implementation of Linux device driver for the BMP085 sensor has been explained and tested on Raspberry Pi 2 running the Linux kernel version 4.14

REFERENCES

1. www.linuxkernel.org
2. https://opensourceforu.com/2017/01/regmap-reducing-redundancy-linux-code/