Presentación trabajo final protocolo de comunicación en sistemas embebidos.

Driver de sAPI para sensor BMP 280

FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Alumno: Ing. Luciano Francisco Vittori

Características del sensor

BMP280

DIGITAL PRESSURE SENSOR

Key parameters

• Pressure range	300 … 1100 hPa (equiv. to +9000…-500 m above/below sea level)
• Package	8-pin LGA metal-lid Footprint : 2.0 × 2.5 mm², height: 0.95 mm
• Relative accuracy (700 … 900hPa @25°C)	±0.12 hPa, equiv. to ±1 m
• Absolute accuracy (950 ...1050 hPa, 0 ...+40 °C)	typ. ±1 hPa
• Temperature coefficient offset (25 ... 40°C @900hPa)	1.5 Pa/K, equiv. to 12.6 cm/K
• Digital interfaces	I²C (up to 3.4 MHz) SPI (3 and 4 wire, up to 10 MHz)
• Current consumption	2.7µA @ 1 Hz sampling rate
• Temperature range	-40 … +85 °C
• RoHS compliant, halogen-free	
• MSL 1	

Usos del sensor

Typical applications

- Enhancement of GPS navigation
  (e.g. time-to-first-fix improvement, dead-reckoning, slope detection)
- Indoor navigation (floor detection, elevator detection)
- Outdoor navigation, leisure and sports applications
- Weather forecast
- Vertical velocity indication (e.g. rise/sink speed)

Los registros para interactuar

4.2 Memory map

The memory map is given in Table 18 below. Reserved registers are not shown.

Table 18: Memory map

Register Name	Address	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	Reset state
temp_xlsb	0xFC	temp_xlsb<7:4>				0	0	0	0	0x00
temp_lsb	0xFB	temp_lsb<7:0>								0x00
temp_msb	0xFA	temp_msb<7:0>								0x80
press_xlsb	0xF9	press_xlsb<7:4>				0	0	0	0	0x00
press_lsb	0xF8	press_lsb<7:0>								0x00
press_msb	0xF7	press_msb<7:0>								0x80
config	0xF5	t_sb[2:0]			filter[2:0]				spi3w_en[0]	0x00
ctrl_meas	0xF4	osrs_t[2:0]			osrs_p[2:0]			mode[1:0]		0x00
status	0xF3					measuring[0]			im_update[0]	0x00
reset	0xE0	reset[7:0]								0x00
id	0xD0	chip_id[7:0]								0x58
calib25...calib00	0xA1…0x88	calibration data								individual

Registers:	Reserved registers	Calibration data	Control registers	Data registers	Status registers	Revision	Reset
Type:	do not write	read only	read / write	read only	read only	read only	write only

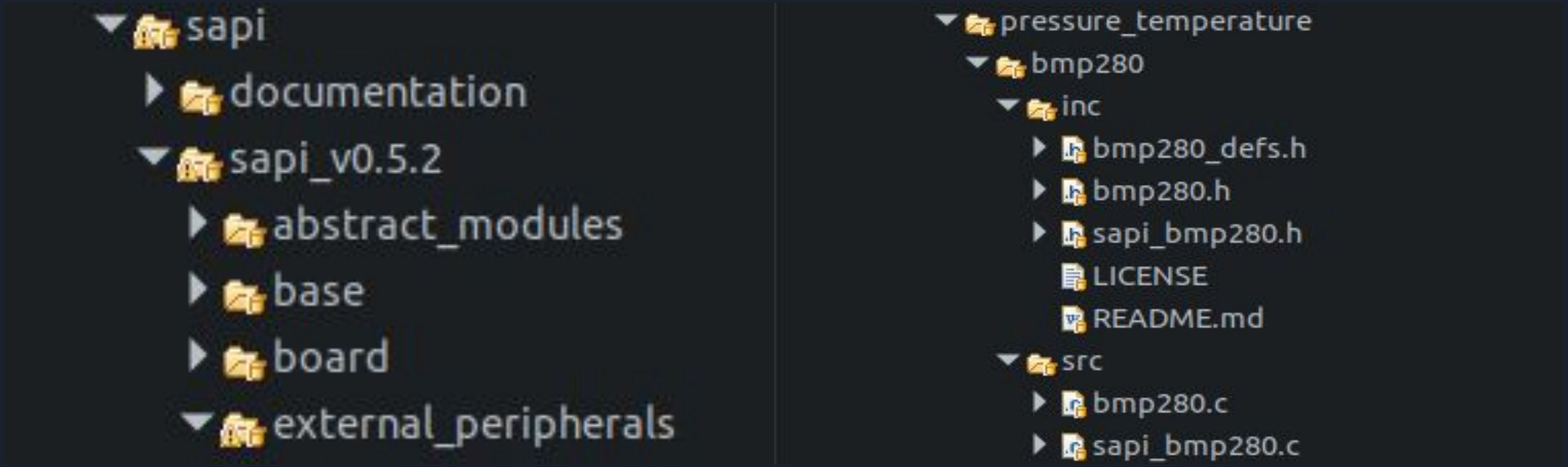
El punto de partida.

Biblioteca oficial del fabricante, es decir BOSCH

https://github.com/BoschSensortec/BMP280_driver

Lo que se agrega: ejemplo de uso y adaptación para las funciones de lectura y escritura

Driver para BMP 280 sAPI



Definición de nuevo periferico en archivo

en archivo module.mk

```
# Pressure_temperature ---------------------------------------------

INCLUDES += -I$(EXTERNAL_PERIPH_BASE)/pressure_temperature/bmp280/inc
SRC += $(wildcard $(EXTERNAL_PERIPH_BASE)/pressure_temperature/bmp280/src/*.c)

```

Include para sapi.h

```c
// External Peripheral Drivers

#include "sapi_7_segment_display.h"      // Use sapi_gpio and sapi_delay modules
#include "sapi_keypad.h"                 // Use sapi_gpio and sapi_delay modules
#include "sapi_dht11.h"                  // Use sapi_gpio peripheral
#include "sapi_lcd.h"                    // Use sapi_gpio peripherals
#include "sapi_servo.h"                  // Use sapi_gpio modules and sapi_timer
#include "sapi_rgb.h"                    // Use TIMER peripheral

#include "sapi_esp8266.h"                // Use sapi_uart module

#include "sapi_magnetometer_hmc5883l.h"  // Use sapi_i2c module
#include "sapi_magnetometer_qmc5883l.h"  // Use sapi_i2c module
#include "sapi_imu_mpu9250.h"            // Use sapi_i2c module
#include "sapi_imu_mpu60X0.h"            // Use sapi_i2c module
#include "sapi_eeprom24xx1025.h"         // Use sapi_i2c module

#include "sapi_ultrasonic_hcsr04.h"      //

#include "sapi_bmp280.h"

```

Escritura en I2C

```c
int8_t i2c_reg_write(uint8_t i2c_addr, uint8_t reg_addr, uint8_t *reg_data, uint16_t length) {
    int8_t status = -1;
    int8_t i2c_ok = 0;
    uint8_t i;

    union {
        uint8_t buff[length + 1];
        struct {
            uint8_t addr;
            uint8_t data[length];
        };
    } i2c_data;

    i2c_data.addr = reg_addr;
    for(i=0;i<length;i++){
        i2c_data.data[i] = *reg_data+i;
    }

    i2c_ok = i2cWrite(I2C0, i2c_addr, i2c_data.buff, length+1, TRUE);

    status += i2c_ok;

    return status;
}
```

Lectura en I2C

```c
/*!
 *  @brief Function for reading the sensor's registers through I2C bus.
 *
 *  @param[in] i2c_addr : Sensor I2C address.
 *  @param[in] reg_addr : Register address.
 *  @param[out] reg_data    : Pointer to the data buffer to store the read data.
 *  @param[in] length   : No of bytes to read.
 *
 *  @return Status of execution
 *  @retval 0 -> Success
 *  @retval >0 -> Failure Info
 *
 */
int8_t i2c_reg_read(uint8_t i2c_addr, uint8_t reg_addr, uint8_t *reg_data, uint16_t length) {
    int8_t status = -1;
    int8_t i2c_ok = 0;

    i2c_ok = i2cRead(I2C0, i2c_addr, &reg_addr, 1, TRUE, reg_data, length, TRUE);

    status += i2c_ok;

    return status;
}
```

Escritura SPI

```
int8_t spi_reg_write(uint8_t cs, uint8_t reg_addr, uint8_t *reg_data, uint16_t length)
{
    int8_t status = -1;
    int8_t spi_ok = 0;
    uint8_t i;

    union {
        uint8_t buff[length + 1];
        struct {
            uint8_t addr;
            uint8_t* data;
        };
    } spi_data;

    spi_data.addr = reg_addr;
    spi_data.data = reg_data;

    BMP280_CS_LOW();
    spi_ok = spiWrite( SPI0, &spi_data.addr, 1);
    spi_ok = spiWrite( SPI0, spi_data.data, length);
    delayInaccurateUs(10);
    BMP280_CS_HIGH();

    status += spi_ok;

    return status;
}
```

Lectura SPI

```
int8_t spi_reg_read(uint8_t cs, uint8_t reg_addr, uint8_t *reg_data, uint16_t length)
{
    int8_t status = -1;
    int8_t spi_ok = 0;
    uint8_t i;

    union {
        uint8_t buff[length + 1];
        struct {
            uint8_t addr;
            uint8_t* data;
        };
    } spi_data;

    spi_data.addr = reg_addr;
    spi_data.data = reg_data;

    BMP280_CS_LOW();
    spi_ok = spiWrite( SPI0, &spi_data.addr, 1);
    spi_ok = spiRead( SPI0, spi_data.data, length);
    delayInaccurateUs(10);
    BMP280_CS_HIGH();

    status += spi_ok;

    return status;
}
```

Ejemplo de uso

```c
int8_t bmp280_init(struct bmp280_dev *dev)
{
    int8_t rslt;

    /* Maximum number of tries before timeout */
    uint8_t try_count = 5;

    rslt = null_ptr_check(dev);
    if (rslt == BMP280_OK)
    {
        while (try_count)
        {
            rslt = bmp280_get_regs(BMP280_CHIP_ID_ADDR, &dev->chip_id, 1, dev);

            /* Check for chip id validity */
            if ((rslt == BMP280_OK) &&
                (dev->chip_id == BMP280_CHIP_ID1 || dev->chip_id == BMP280_CHIP_ID2 || dev->chip_id == BMP280_CHIP_ID3))
            {
                rslt = bmp280_soft_reset(dev);
                if (rslt == BMP280_OK)
                {
                    rslt = get_calib_param(dev);
                }
                break;
            }
```

Ejemplo de uso

4.3 Register description

4.3.1 Register 0xD0 *"id"*

The *"id"* register contains the chip identification number chip_id[7:0], which is 0x58. This number can be read as soon as the device finished the power-on-reset.

4.3.2 Register 0xE0 *"reset"*

The *"reset"* register contains the soft reset word reset[7:0]. If the value 0xB6 is written to the register, the device is reset using the complete power-on-reset procedure. Writing other values than 0xB6 has no effect. The readout value is always 0x00.

Escritura SPI

5.3.1 SPI write

Writing is done by lowering CSB and sending pairs control bytes and register data. The control bytes consist of the SPI register address (= full register address without bit 7) and the write command (bit7 = RW = ‘0’). Several pairs can be written without raising CSB. The transaction is ended by a raising CSB. The SPI write protocol is depicted in Figure 10.

	Control byte		Data byte	Control byte		Data byte	
Start	RW	Register address (F4h)	Data register - address F4h	RW	Register address (F5h)	Data register - adress F5h	Stop
CSB = 0	0	1 1 1 0 1 0 0	bit7 bit6 bit5 bit4 bit3 bit2 bit1 bit0	0	1 1 1 0 1 0 1	bit7 bit6 bit5 bit4 bit3 bit2 bit1 bit0	CSB = 1

Figure 10: SPI multiple byte write (not auto-incremented)

Lectura SPI

5.3.2 SPI read

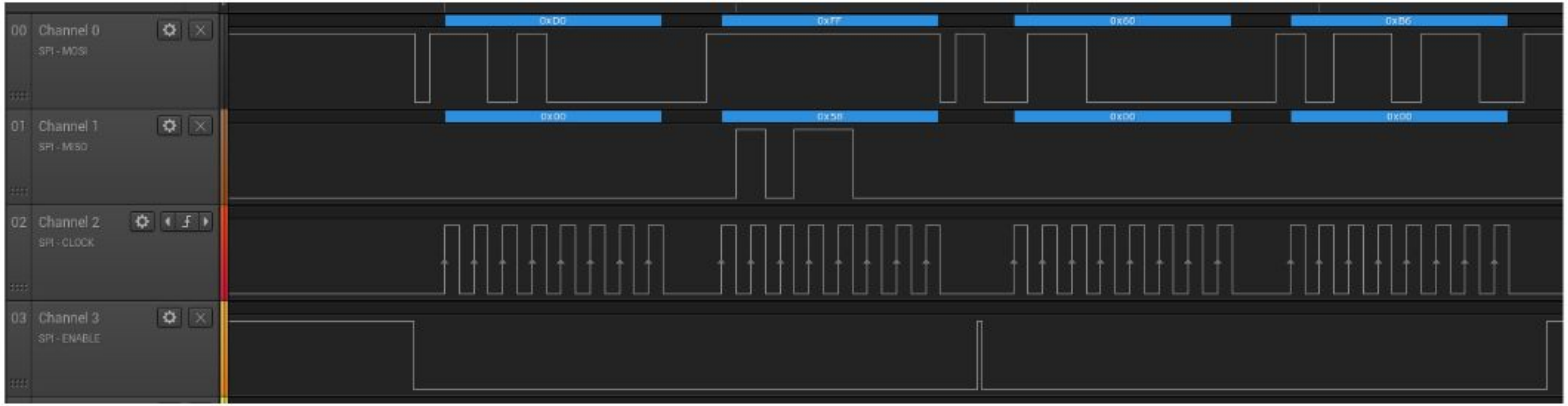
Reading is done by lowering CSB and first sending one control byte. The control bytes consist of the SPI register address (= full register address without bit 7) and the read command (bit 7 = RW = ‘1’). After writing the control byte, data is sent out of the SDO pin (SDI in 3-wire mode); the register address is automatically incremented. The SPI read protocol is shown in Figure 11.

	Control byte								Data byte								Data byte								
Start	RW	Register address (F6h)							Data register - address F6h								Data register - address F7h								Stop
CSB = 0	1	1	1	1	0	1	1	0	bit15	bit14	bit13	bit12	bit11	bit10	bit9	bit8	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	CSB = 1

Figure 11: SPI multiple byte read

Control SPI

Tanto para escribir registros o para leer, primero siempre **se escribe** el registro de control.



0xD0 = 1101 0000 -> para leer -----> 1101 0000 = 0xD0

0xE0 = 1110 0000 -> para escribir -> 0110 0000 = 0x60

Escritura I2C

5.2.1 I²C write

Writing is done by sending the slave address in write mode (RW = ‘0’), resulting in slave address 111011X0 (‘X’ is determined by state of SDO pin. Then the master sends pairs of register addresses and register data. The transaction is ended by a stop condition. This is depicted in Figure 7.

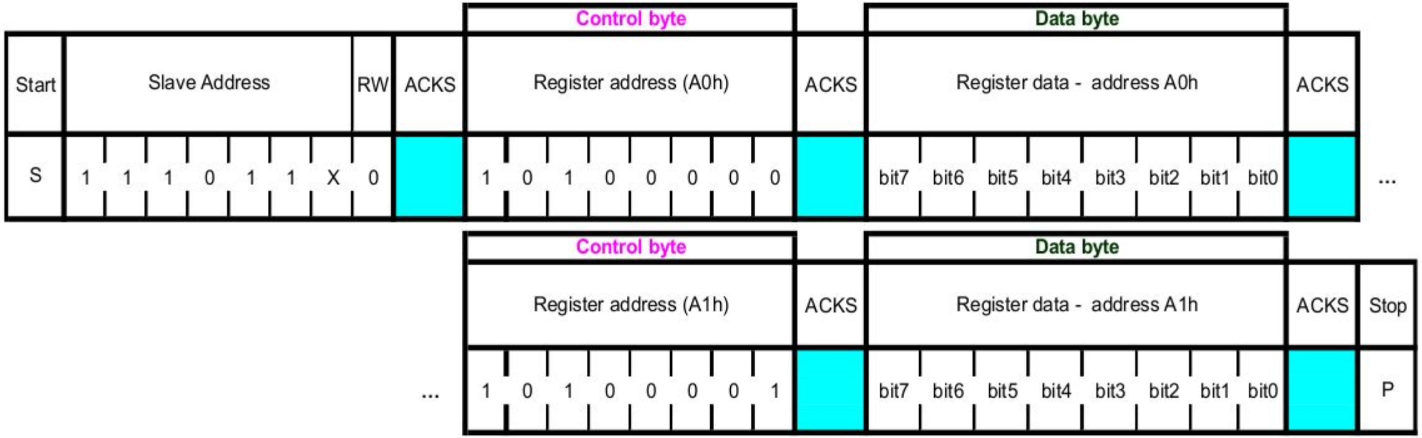


Figure 7: I²C multiple byte write (not auto-incremented)

Lectura I2C

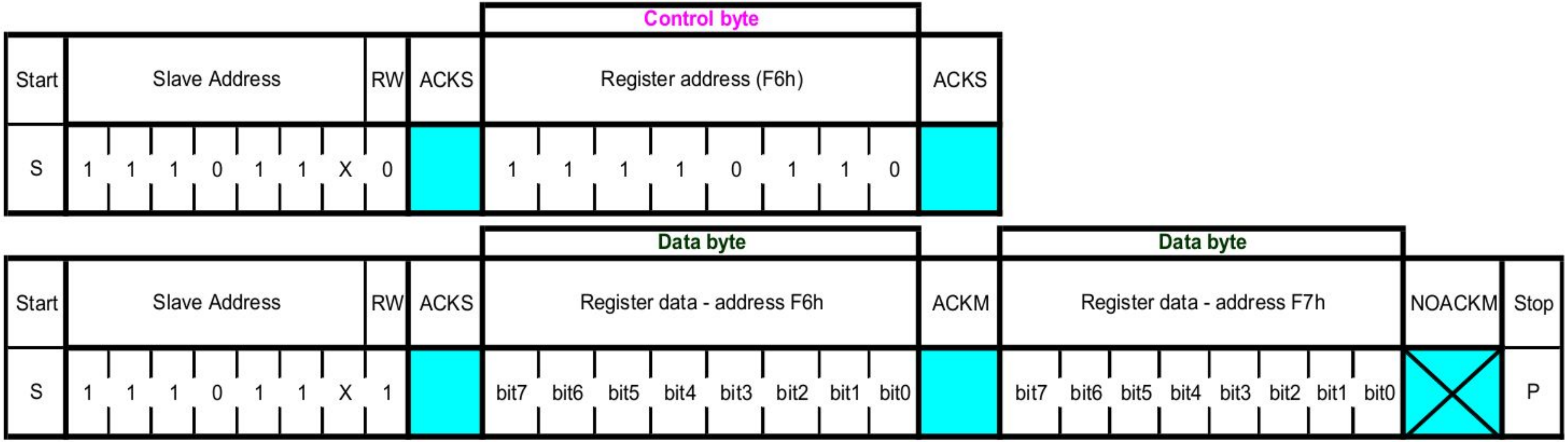


Figure 8: I²C multiple byte read

Escritura I2C

Se le agrega el bit RW al slave address

SLAVE ADDRESS = 0x76 = 0111 0110

Para escribir-> 0x76<<1 + RW = 1110 1100 = 0xEC

Para leer -----> 0x76<<1 + RW = 1110 1101 = 0xED

Muestra de capturas

Vemos ejemplo en analizador lógico

Gracias