Micro-controller Lab Assignment 3 & 4: I2C & SPI Supplementary Document for Addressing Few Issues About SPI

Computer Science and Engineering, University of Dhaka,
Version 1.0

June 10, 2022

Contents

1 Objectives

This document intends to elucidate problem regarding implementing SPI to read data from or communicate to the BMP280 (not BME280) to get the temperature and pressure of the proximity.

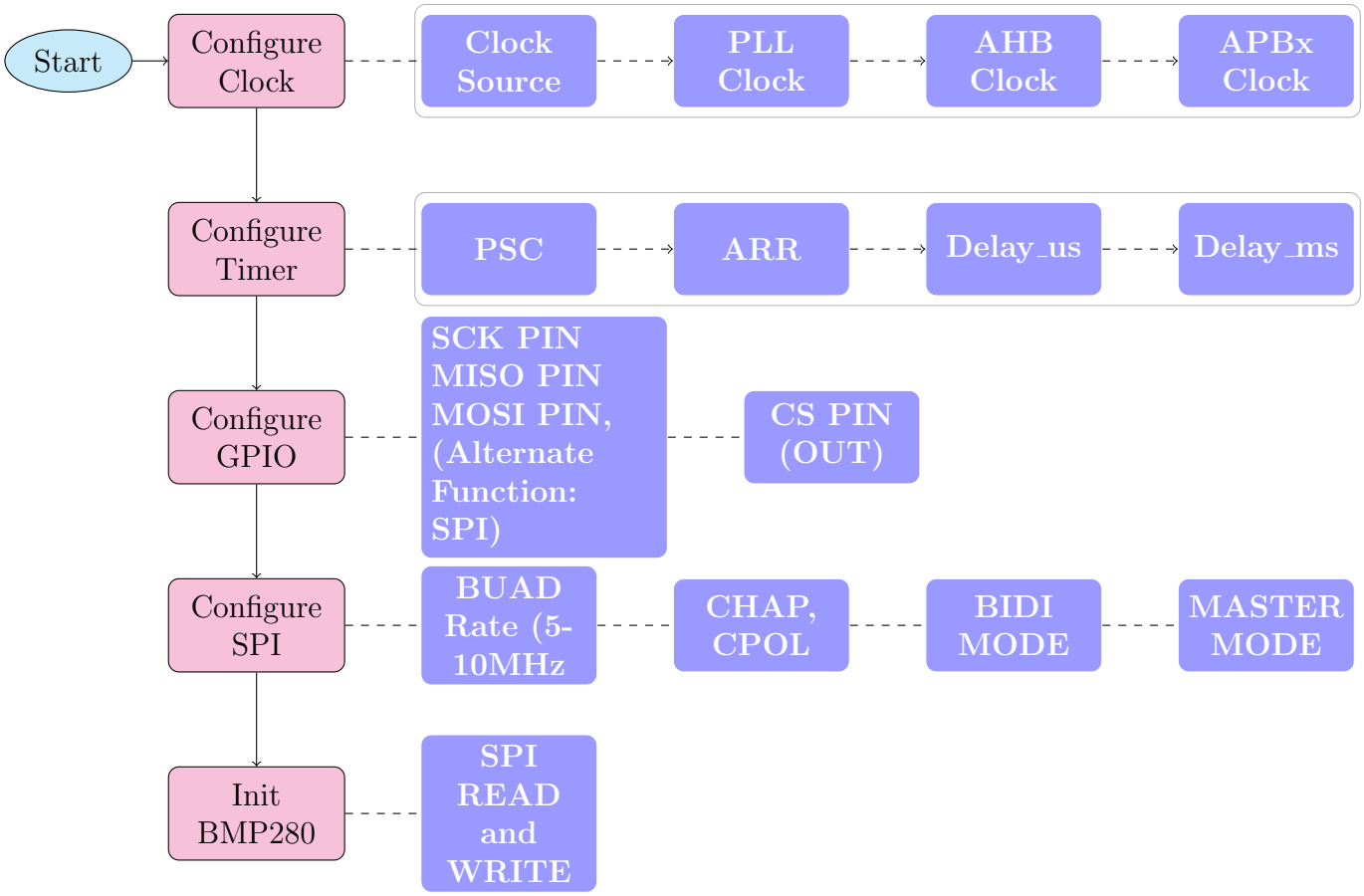
2 Issues

We must consider a few issues for finding a solution to LAB 3 and 4. The issues are associated with the chip and inadequate knowledge of the practical deployment of SPI protocol. We here documented a few problems with the LAB programming, and I encountered a significant problem: SPI reading.

2.1 BME280 or BMP280 Issue

Unfortunately, the chip is not BME280, and it is, in fact, BMP280. The chip only transmits the pressure and temperature of the current location to the microcontroller. According to the vendor, it is believed to be a BME280: temperature, humidity, and pressure sensor. However, it is not; it is a Bosch BMP280: pressure and temperature sensor. We can verify it simply by reading the ID of the chip. The chip ID is '0x58', which has a match with BMP280, not with BME280 (ID: '0x60').

2.2 Step by Step Procedure – in short



2.2.1 Initialize BMP280

Algorithm 1 BMP280_init

function PowerOnReset(void)
    $RESET\_REG \leftarrow 0xB6$ ▷ See Datasheet
    $Delay\_ms \leftarrow 500$
end function
function BMP_Mode(PwrMode,TempOvrSampling,PressOvrSampling)
    $CONFIG\_REG \leftarrow (PwrMode, TempOvrSampling, PressOvrSampling)$
end function
function SetFilterCoefficient(FilterCoEfficient)
    $x \leftarrow CONFIG\_REG$
    $CONFIG\_REG \leftarrow (x \ \& \ 0x1F)|FilterCoEfficient$
end function
$StandbyTime \leftarrow 625000\mu S$
$ReadTrimValue()$
$SetReferencePressure(sample, delay)$

2.2.2 Reading Trim Value

You can use the following algorithm for reading Trim Value; you need these values to calculate the temperature and pressure.

```
static uint16_t dig_T1, dig_P1;
static int16_t  dig_T2, dig_T3,dig_P2, dig_P3, dig_P4,
dig_P5, dig_P6, dig_P7, dig_P8, dig_P9;
void BMP280_Read_Trim_Value(void){
uint8_t trimdata[24];
SPI1_MulReadReg(TRIM_START_REG,trimdata,24);
// Arrange the data as per the datasheet (page no. 24)
dig_T1 = (uint16_t)((trimdata[1]<<8) | trimdata[0]);
dig_T2 = (int16_t)((trimdata[3]<<8) | trimdata[2]);
dig_T3 = (int16_t)((trimdata[5]<<8) | trimdata[4]);
dig_P1 = (uint16_t)((trimdata[7]<<8) | trimdata[5]);
dig_P2 = (int16_t)((trimdata[9]<<8) | trimdata[6]);
dig_P3 = (int16_t)((trimdata[11]<<8) | trimdata[10]);
dig_P4 = (int16_t)((trimdata[13]<<8) | trimdata[12]);
dig_P5 = (int16_t)((trimdata[15]<<8) | trimdata[14]);
dig_P6 = (int16_t)((trimdata[17]<<8) | trimdata[16]);
dig_P7 = (int16_t)((trimdata[19]<<8) | trimdata[18]);
dig_P8 = (int16_t)((trimdata[21]<<8) | trimdata[20]);
dig_P9 = (int16_t)((trimdata[23]<<8) | trimdata[22]);
    }
```

2.2.3 Measurement of Temperature and Pressure values

```
void BMP_Measure(float* temperature,float* pressure,float* altitude)
{
uint8_t data[6];
int32_t adc_P,adc_T;
SPI1_MulReadReg(PRESS_MSB_REG,data,6);
adc_P = data[0] << 12 | data[1] << 4 | data[2] >> 4;
adc_T = data[3] << 12 | data[4] << 4 | data[5] >> 4;
*pressure = (float)(BMP_Press_Compensation(adc_P)/256.0);
*temperature = (float)(BMP_Temp_Compensation(adc_T)/100);
if(p_reference > 0)
{
*altitude = (float)(1.0- pow(((double)*pressure)/((double)p_reference), 0.1903)) *
(float)4433076.0;
*altitude = (*altitude /10)*(-1);
}
}
```

2.2.4 Temperature and Pressure compensation

Use the trim values to compensate for calculating temperature and pressure data collected from BMP280. Otherwise, your calculation will be wrong.

```
    /**
* Defined in data sheet
* Temperature Compensation Function return 32 bit value
**/
int32_t BMP_Temp_Compensation(int32_t adc_T)
{
int32_t var1, var2,T;
var1 = ((((adc_T>>3) - ((int32_t) dig_T1<<1))) * ((int32_t) dig_T2)) >> 11;
var2 = (((((adc_T>>4) - ((int32_t)dig_T1))*((adc_T>>4) -
((int32_t)dig_T1)))>>12)*((int32_t)dig_T3))>>14;
t_fine =var1+var2;
T = (t_fine*5+128)>>8;
return T;
}

    /**
* Defined in data sheet
* Pressure Compensation -- return 64-bit value
**/
uint32_t BMP_Press_Compensation(int32_t adc_P)
{
int64_t var1,var2,p;
var1 = (int64_t) t_fine - 128000;
var2 = var1 * var1 * (int64_t) dig_P6;
var2 = var2 + ((var1*(int64_t)dig_P5)<<17);
var2 = var2 + (((int64_t)dig_P4)<<35);
var1 = ((var1 * var1 * (int64_t)dig_P3)>>8)+((var1 * (int64_t)dig_P2)<<12);
var1 = (((((int64_t)1)<<47)+var1))*((int64_t)dig_P1)>>33;
if(var1 == 0 ) return 0;
p = 1048576 - adc_P;
p = (((p<<31)-var2)*3125)/var1;
var1 = (((int64_t)dig_P9) * (p>>13) * (p>>13)) >> 25;
var2 = (((int64_t)dig_P8) * p)>>19;
p = ((p + var1 + var2) >> 8) + (((int64_t)dig_P7) << 4);
return (uint32_t)p;
}
```

2.3 SPI Read and Write

The most critical issue is reading data from BMP280 using SPI. Remember, SPI is a bidirectional data communication with MISO and MOSI for data in and out of the master. To read or write, you must keep the CS pin low connected t to the CSB of BMP280. When you send the Register address to read (for any SPI reading), the device (slave) will immediately start transmission (see Fig. 1). Here it is '0x00' or any other value. However, the received (master) data is garbage because the address is still in transit. After transmitting the address, it disables the clock; therefore, you will not get the data that you are requesting. Somehow, you have to keep the clock running from the master. The trick is to send another byte: I send '0x00'. Now the desired data is available. You must ignore the first (garbage) byte and accept subsequent bytes for multiple readings. However, it would help if you continued sending

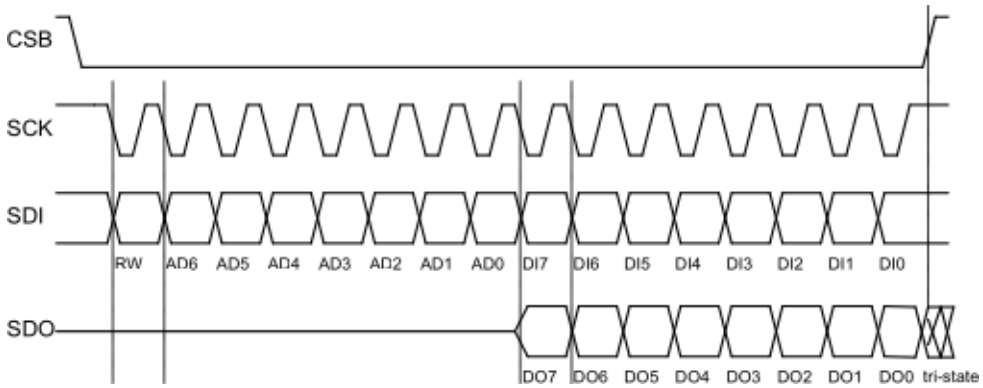


Figure 1: SPI protocol

0x00 or a valid Register address until all data are received. In most cases, you may transfer the Register addresses if you want to read a series of Register values and end with '0x00'.