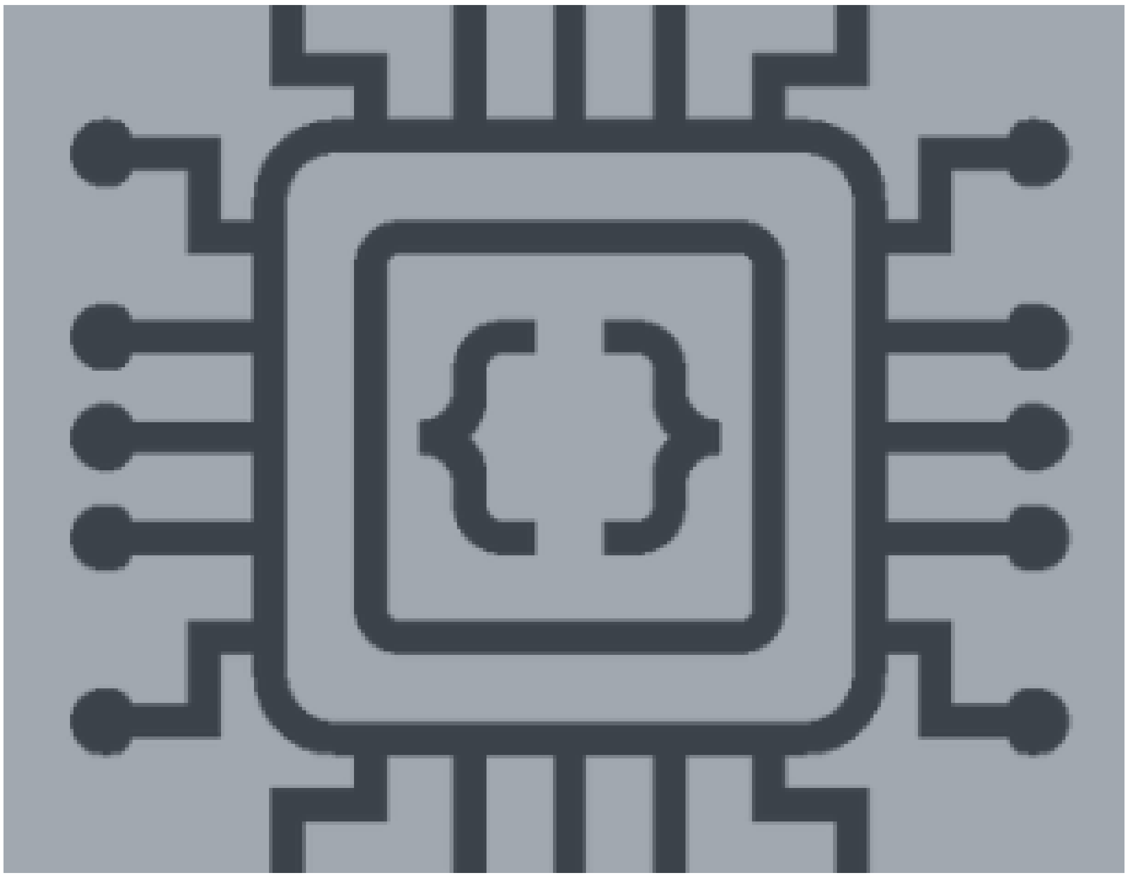
Embedded Systems



Assignment: 4
Date: 29/04/2023
Group: Johnson Domacasse, Toon Spitshuis
Version: V1

Contents

Abstract
I2C is a communication protocol that connects a master and a slave device together. It is a powerful two wire interface that is easily implemented when understood. It works in the form on requesting and receiving data.

Introduction
The I2C bus (I square C bus) is a two wire interface bus that connects a master device with slave devices. This practicum is so we can put this communication protocol for four different applications. These applications are:

1. An Arduino communicating with a BME280 sensor (I2C device), by creating a driver for the sensor.
2. Two Arduinos communicating via the I2C bus using the Arduino framework.
3. Making a parallel computer using two or more Arduinos using the master-slave concept from 1 and 2.
4. Demonstrate that the applications 1 and 3 can work simultaneously on one I2C bus.

The general conclusion that can be pulled from these experiments is that I2C is indeed a powerful way in order to connect multiple systems together. It is just a matter of requesting and receiving once you understand the communication protocol. Like most protocol, it is error prone, however the error can be easily found because it would almost tell you exactly which wire the error occurred.

Procedures
Hardware setup:
The following hardware was used in order to pull off these experiments:

- A laptop + VS Code + Platform IO
- 2 x Arduinos + USB cable
- 1 x breadboard
- 1 x BME280 sensor
- Jumper wires

Software setup:
We will be making use of the “Wire” library for I2C as provided by the Arduino framework.

We will be making use of header files, the Wire library and previously mentioned and of course the main program.

BME280 Device sensor:
This part of the assignment is about an Arduino that needs to communicate with a multifunction air sensor (BME280) by creating a device driver for it.

Plan for code
We will create the software using C. This is because the device driver is low-level and it is related to the datasheets of the sensor, e.g. the technical terms are borrowed from the datasheets.

We will create the code based on the connection to be considered in figure 1.

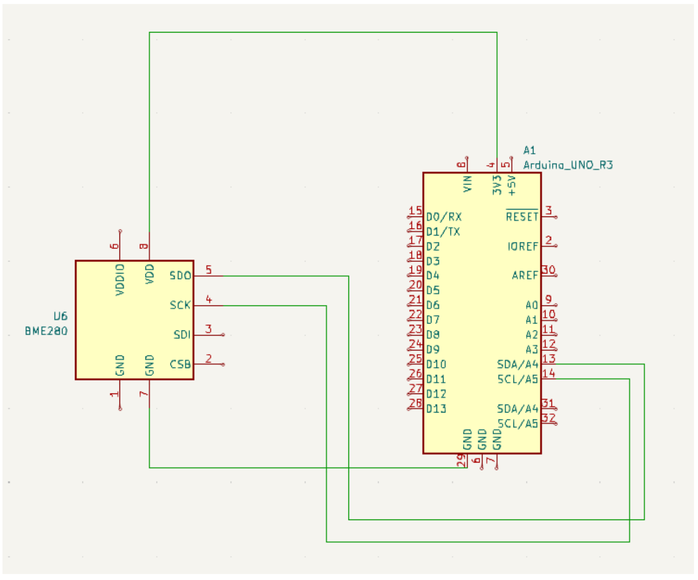


Figure 1: Circuit diagram of BME280 and Arduino Uno

Notice that the VCC pin is connected to the 3.3V pin on the Arduino. The sensor can function up until 3.6V. Even though the SDL and SCL can tolerate the 5V.

We will only be translating what we perceive from the datasheets into code. Meaning, we will not be using any external libraries for this part of the assignment.

For this assignment the Arduino will act as the master and the sensor will act as the slave. The master program directly addresses the hardware registers of the sensor via I2C. The hardware registers are described in the datasheets of the sensor. The master prints the sensor air properties humidity, temperature and pressure on the serial plotter.

We will create this device driver using functions declared and implemented in the BME280.cpp and BME280.h files. See ***table 1*** for complete list of functions. The driver will also includes the usage of some predefined addresses that cannot be changed. See ***table 2*** for complete list of all of these addresses.

The main program will be used to setup the sensor and then read the designated air properties.

Code.

We will be making use of the Wire.h library and its capabilities as mentioned in software setup section. A majority of the code that was created was code derived from the BME280 datasheets. We will be controlling the BME280 by connecting it as a slave to the Arduino and control it via its hardware registers. These are found in the datasheet.

Register addresses.

Before we began with programming the functions, we went to work on trying to find the values of the addresses we needed to pass to the functions. We did this by going through the datasheets. Upon scanning the datasheet, we found that the value of the BME280 address is the hexadecimal number: 0x76. This is found at the bottom of page 32. On page 27, In table 18: memory map from the datasheets, we can find the address of the most and least significant bits (MLB and LSB) for the temperature and humidity registers.

We also found the address for the CTRL_HUM and the CTRL_MEAS registers as well. We also found the reset and ID register addresses within this table. (**include new addresses).** All of this can be found again in ***table 2***.

Functions.

We were provided with a few functions that we need to implement. We noticed that some functions have a common property upon implementation. Some of them read from an address. Others would write to an address. In order to simplify this we made two additional functions: the WriteToAddress and ReadFromAddress. All of these functions can be found in table 1.

The WriteToAddress function is our writing function and it accepts 3 parameters: an I2Caddress, an address and a message. All of type "uint8_t". Inside the function, you will find 4 lines of code that we use in order to write a message to a specific address. We begin the transmission by connecting it with the I2Caddress. We then write to the address we will be writing too. We then write a specific message to the address previously provided (not to be confused with the I2Caddress). Finally, we end the transmission.

The ReadFromAddress function is our reading function and it accepts 2 parameters: an I2Caddress and an address. All of type "uint8_t". Inside this function, you will find 5 lines of code that we use to read from the address that was provided. We begin the transmission by connecting it with the I2Caddress. We then write to the address we will be reading from. We then restart the transmission by using the endTransmission function, but passing false as a parameter. We then request an amount of bytes (in this case 1) from the I2Caddress. Finally we return the value that we read from the address.

It was a matter of providing the right address and message to the remaining functions once these 2 were complete. The I2Caddress for every one of the functions was the defined constant: BME280_ADDRESS. See ***table 2*** for all of the address and messages used in the functions.

The BME280_GetID function uses the reading function with the following address: BME280_ID_ADDRESS.

The BME280_Reset function uses the writing function with the following address and message: BME280_RESET_ADDRESS and BME280_RESET_MESSAGE.

The BME280_CtrlHum function uses the reading function with the following address: BME280_CTRLHUM_ADDRESS.

The BME280_CtrlHum function uses the writing function with the following address and message: BME280_CTRLHUM_ADDRESS and bitpattern.

The BME280_CtrlMeas function uses the reading function with the following address: BME280_CTRLMEAS_ADDRESS.

The BME280_CtrlMeas function uses the writing function with the following address and message: BME280_CTRLMEAS_ADDRESS and bitpattern.

After these function were implemented, we got to work on some functions to return the value of the temperature and humidity that it had read. See ***table 1*** for these functions.

Reading Functions

Disclaimer, In order to implement these functions, we copied and re-applied the code that was provided to us from the datasheets. You can find these provided codes under section 4.2.3 Compensation formulas in the datasheet. The types of the different values can be determined using the text under the same section. For example, a variable with type

“BME280_S32_t” can be defined as a “long signed int”. Unsigned short can be defined as uint16 and signed short can be defined as int16.

Our functions make use of a the compensation variable “*dig”.* Each dig is a combination of 2 words (8bits each). It is then followed by an underscore and the capital first letter of what they related to. For example, “dig_T#” is related to the temperature. In table 16: Compensation parameter storage, naming and data type we can see the register addresses along with the datatype of each dig variable that we later need to use.

The ReadTemperature function is implemented as follows. We start of by reading the MLB, LSB and XLSB from their respective addresses and then assigning them to the “adc_T” variable. We then declare the variables var1, var2 and T. From there, we use table 16 in order to set the value of each “dig” variable using their respective register addresses. We shift the first word to the left by 8 and leave the second one in place. We then use an or operator to compare them and then assign them to their respective dig values: “dig_T1” , “dig_T2” and “dig_T3”. We then follow the compensation formula so that we can finally return the Temperature value.

The ReadHumidity function is implemented as follows. First declare am array of 2 values called “buf”. We then read the MLB and LSB from their respective addresses and assign each to their own place in the buf array. We then use table 16 in order to set the value of the single humidity dig variable (dig_H#). From the we get the raw humidity simply by shifting the MSB to the right and using the or operator along with the LSB**.** We then use the consolation formulas to calculate the humidity. We then make sure that the humidity can’t go above the value of 100000 and below the value of 0. Finally, we return the refined humidity if it is fine.

The ReadPressure function is implemented as follows. We first declare an array called data, with type uint8_t and size of 3. We also have the variable called ADC_P. We then use the reading function with the BME280 address and the address of each of the 3 bits. These are then assigned to a single spot within the array. We then use the ADC_P variable in order to convert all of the data to 20-bits. We do this by shifting the bits to their appropriate spots. We then assign the dig_p# values each by reading the bits at their respective address. We then use then use the compensation formulas to calculate the pressure. Finally we return the calculated pressure.

Table 1, *complete list of driver functions.*

Function name	***Return type***	***description***
WriteToAddress(uint8_t I2Caddress, uint8_t address, uint8_t message)	Void	Function that writes a message to a specific address.
ReadFromAddress(uint8_t I2Caddress, uint8_t address)	uint8_t	Function that reads from a singular byte from an address and then returns the value.
BME280_GetID()	Uint8_t	Function that reads the single byte from the “ID_ADDRESS” and returns this.
BME280_Reset	Void	Function that writes the reset message value to the reset address.
BME280_CtrlHum()	Uint8_t	Function that reads the single byte from the “CTRLHUM” address and returns this.

BME280_CtrlHum(uint8_t bitpattern)	Void	Function that writes the amount of bits to the "CTRLHUM" address.
BME280_CtrlMeas()	Uint8_t	Function that reads the single byte from the "CTRLMEAS" address and returns this.
BME280_CtrlMeas(uint8_t bitpattern)	Void	Function that writes the amount of bits to the "CTRLMEAS" address.
BME280_ReadTemperature()	Long	Function to read the temperature value.
BME280_ReadHumidity()	Int	Function to read the humidity value.

Table 2: *Complete list of address names and values.*

Address name	Address value
BME280_ADDRESS	0x76
BME280_RESET_ADDRESS	0xE0
BME280_RESET_MESSAGE	0xB6
BME280_ID_ADDRESS	0x60
BME280_CTRLHUM_ADDRESS	0xF2
BME280_CTRLMEAS_ADDRESS	0xF4
BME280_TEMPMSB_ADDRESS	0xFA
BME280_TEMPLSB_ADDRESS	0xFB
BME280_TEMPXLSB_ADDRESS	0xFC
BME280_HUMMSB_ADDRESS	0xFD
BME280_HUMLSB_ADDRESS	0xFE

I2C and Arduino framework:

Plan for code.

For this part of the assignment we needed to combine two Arduinos together and connect them with each other using I2C. See **figure 2.** Before we begin programming, we need to understand a few key factors here. First like previously, we will be using the Wire library. Within this library there are many functions that we can use in order to properly connect the Arduinos together using I2C. you can find these functions and their description in **table 3.** Now that this has been addressed we can get started.

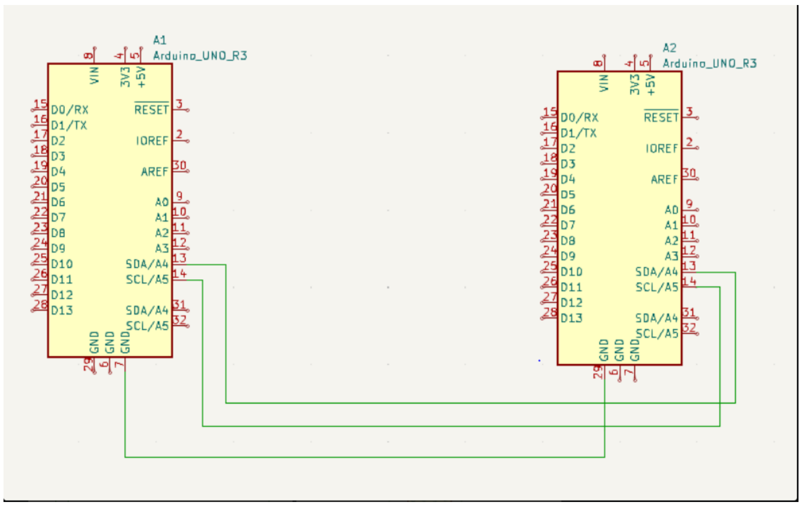


Figure 2: Circuit diagram of Arduino Uno's

Code.

The first part of creating our code was understanding two cases. One where the master would write and the slave would receive. The other would be where the master reads and the slave sends. We will start by looking at the first case. Then we will look at the second case. Finally we will create our own code based on these two cases. We assume the usage of serial objects in order to begin and print our values so we can see them.

Case 1:

The communication code between the two Arduinos looks quite simple. In the master program, you will see the begin function (without an address) to begin the wire communication in the setup function. We declare a variable called x of type byte as 0. Within the loop function, we begin by using the begin transmission function with address 8. This is going to be the address that we are writing to. We then use the write function twice. One to write 5 bytes of char characters and the other for the, previously declared, x variable. We then use the end transmission function. Finally, we increment variable x. This is all done within a loop of half a second. What we understand here is when we want the master to send bytes to the slave, it would need to include an address within the begin transmission function. The begin function can be left without an address in this case.

In the slave program, you will find that the begin function in the setup as well however this time it was provided an address. This address is 8 and is an important detail for the master program. We can also find the on receive function that accepts the receive event function as a parameter. What is important here is what is inside the receive event. Inside you will find that if the available function is true, it will read what was sent to it from the other side and print this on the serial monitor.

Case 2:

The communication code between the two Arduinos looks simple here as well. There will be some similarities like the first one. For example the begin function within the setup function. In this case we also use a serial monitor in order to see what is being sent from the other side. The only difference to a master reading values instead of the slave is that there is no on receive or on request functions. The master reads by requesting an amount of bytes from a specific address using the request from function. In the checks with the available function if a connection is available. If this is the case, it will read and print the values on the monitor. Again all within half of a second as delay.

In the slave program, you will find the begin function with the provided address. This value again will be 8. However this time it has an on request function that accept the request even function as a parameter. This means that this function only works when the master makes use of the request form function with the right amount of bytes and address. Within the request even function we just have a simple write function that writes “hello “.

Our implementation:

Now that we understood how the Arduinos communicated which each other (sending and receiving from both sides) we could finally start with our implementation. We also made some slight improvements by using the MILLIS function instead of the usual delay function.

Let us begin by implementing the slave program. We will be using a predefined address for the Arduino. This address has a value of 0x42. We define a variable called output with type byte and starting value 0. We then implement both the on request and on receive functions. This is because the program must be able to both send and receive data. They both accept two functions as parameters. A request event function and a receive event function. We were asked that if the received byte from the master was above a certain value, the slave would send the value 2 else it would send the value 4. So this is what we did in our receive event function. We would read the input from the master and we would check if that input is above the certain value. If the value was above the certain value, we would assign 2 to the output value if not we would assign 4 to the output value. In our request event function we would write the previously assigned output value to the master. In our loop function, we have a small delay of 0.1 seconds.

Now we began to work on the master program as well. We defined our Arduino address value with 0x42. We also defined the delay value with 500. Instead of using delay, we wanted to implement the MILLIS function, so we declared the variable LastTime of type long. We also have the variable x of type byte. In our setup, we used the begin function without an address since it is the master program. In our loop function, we use the begin transmission function with the address of the slave Arduino. We then write the variable x to it. Finally we end the transmission. We also use the request from function in order to receive the value of output from the other side. So as parameters we use the slave address and 1 (one byte that needs to be sent). We then increment x after every transmission. We assign the current time to the last time variable. Finally we have a small loop to continuously read messages if the connection is available.

Register addresses.

We only used an address for the slave Arduino in our implementation. This address is 0x42.

Functions.

Table 3, *complete list of WIRE library functions used.*

Function Name	***Description***
begin()	*Initializes the wire library and join the I2C bus. Optionally, the slave address can be passed as a parameter.*
onReceive()	*Function registers a function to be called when slave device receives transmission from master device.*

onRequest()	*Function registers a function to be called when the master device requests data from the slave device.*
Available()	*Returns the number of bytes available for retrieval with the read function. Should be called after the request from function on the master device. On slave device, it should be within the on receive function.*
Read()	*Reads a byte that was transmitted from slave device to master device after the request from function. Can also be used in slave device to read bytes send from master device.*
Write()	*Writes data from data from slave device in response to a request from master device. Also transmits bytes to slave device from master device (in between the begin and end transmission functions).*
beginTransmission()	*Begins transmission with the slave device with given address. Can be used with the write and end transmission functions to transmit bytes.*
endTransmission()	*Ends transmission with the slave device with given address and transmits bytes given by the write function.*
requestFrom()	*Used by the master device to request a set amount of bytes from the slave device. These bytes are retrieved using the available and the read functions.*

Arduino is a slave:

Plan for code.

We are going to be using the concepts we have used in part B of this assignment for this part. This is because the Arduino must be able to both send and receive. Meaning, the same wire library functions, previously used, will be used here as well.

Of course for this part we will add some additional functions to make the implementation more readable.

For this part we must use the "datasheet" of this device. This is given to us in the table, found on the assignment pdf. From this sheet, we get the address name and value, the meaning per address, the size of the byte and finally the IO functionality. We will use the same circuit as we did in the previous part. See **figure 2.**

Register addresses.

There are 4 register address that are important for this part. You can find these address in **table 5**. Along with the address of the actual device.

Code.

Once we understood the assignment we could get started. Before we began, we implemented a few functions that would help us later. See **Table 4.** On top of this, we will use a switch case in the slave program to work based on the register that was sent by the master.

The first part was to implement the master program by making it so that the master can ask for the min or max at any time. We implemented this by creating the request value function. Within this function, the master would used the request from function to request 4 bytes from the slave address. These 4 values are: a, b, input min and input max. On the slave side of the program, upon calling the request event function, the slave would write the values of a, b, max and min to the master. Upon using the receive event function, the slave would get the data that was transmitted to it via the write value function from the master.

The second part was to see what happens when the master writes to the MIN or MAX registers. The master would generate a random value and send this to the slave. On the slave side, the slave would check which register it is, then it would read the data that was sent, and then assign this value to the variable it needs to be. For example, variable b. In the switch case, the register would need to be the VALUEB register. Then it would assign the data that was read, using the receive event function, then it would assign this value to the INB variable.

Functions.

Table 4, *complete list of additional functions.*

Function name	**Return type**	**Description**
WriteValue(int address)	Void	This function first generates a random value. Then it sets up a transmission in order to write to value a or b. This is dependant on the address. Finally the slaves sets the value of a or b based on the address.
requestValue()	Void	This function request 4 bytes from the slave. A, b, min and max from the slave.
calcMinMax()	Void	This function sets up the transmission in order to write to the min and max addresses in order for the slave to set the value of min and max at those addresses.
Slave_init()	Void	This function has the begin function with the slave address as a parameter. It has both the on receive and on request functions. Finally, it has an initialization of the previous register variable for the switch case.
receiveEvent(int howMany)	Void	This function checks if there is a connection available before reading the which register the master is asking for. If the register address is above 1 then it would read the data from that register.
RequestEvent()	Void	This function would write the values of the 4 bytes requested by the master. This function is called as a parameter in the on request function.

Table 5, *complete list of register address names and values.*

Address name	***Address value***
Device address	*0x50*
VALUEA	*0x21*
VALUEB	*0x22*
MIN_ADD	*0x23*
MAX_ADD	*0x24*

Demo one master and multiple slave on I2C:

Plan for code.

We need to add another slave device additionally on top of the original slave for this part of the assignment. We noticed that this can easily be done by combining what we did in part A and part C together. All of the key factors that we need to understand before we get started are understood because of the previous exercises in this assignment.

We also needed to implement a new system in order for this to work. See **figure 3.**

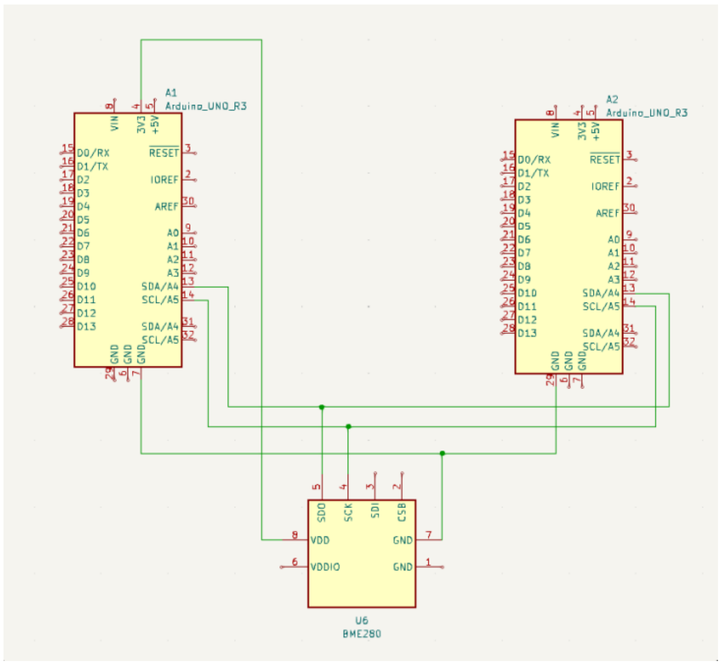


Figure 3: Circuit diagram of BME280 and two Arduino Uno's

Code.

Aside from the code being large, you will not notice much of a change. Part D is essentially part A and part C combined. It works around the same principle as the slave awaiting a command to send back to the master. Consult part A and C, in order to find the code for both the BME280 and the Arduino as a slave.

Notable anomaly:

There were some anomalies we encountered when working on this assignment. We will go over these in this section.

While we were working on the BME280 driver, we noticed that the functions all did the same thing. So in order for us to stop repeating ourselves, we implemented additional functions. One that reads from an address. The other that writes to the address. This made our code smaller. All we had to do within the driver functions was call the correct function and pass parameters that that function needed.

Another anomaly was the reading the values from a function in the BME280 driver as well. We noticed that the values print out values without a decimal point. So once we went back to the datasheet we found the cause. We noticed that if you use a 64 bit integer support you are capable of receiving values with double precision. In other words, values like 53.45. However when you use a 32 bit integer support, you are left with a whole decimal value. In other words, using the same value as before, 5345. The resolution here is 0.01. So the value of 5345 should be read as 53.45.

Conclusion

The general conclusion is that I2C is a powerful and useful communication protocol. It Is not difficult to understand. When you do, you realize that the protocol is just a matter of requesting and receiving. Like most other communication protocol, an I2C protocol is also error prone. However we think that it can solve these errors easily. It works based on request and receive. So you can easily find where the error sits. For example, if the message was requested properly but not received, that means that the problem sits near the receive wire.

References

[1] Humidity Sensor BME280 - Bosch Sensortec GmbH. (2022, January)
https://www.bosch-sensortec.com/products/environmental-sensors/humidity-sensors-bme280/

[2] *Wire - Arduino Reference*. (n.d.).
https://www.arduino.cc/reference/en/language/functions/communication/wire/