Bachelor's Thesis Computing Science

Sniffing communications between an Arduino and its peripheral sensor

For fun and profit

Stefan van Ieperen
s1045156

August 26, 2022

First supervisor/assessor:
dr. Ileana Buhan

Second assessor:
dr. Erik Poll

Radboud University

Abstract

I2C and SPI are communication protocols that are often used in embedded systems, where one component needs to communicate to another. Listening in on these conversations can tell you a lot about the inner workings of a device. This can then lead to the discovery of security weaknesses.

Typically, spying on digital communications happens with a logic analyzer; an impressive device that has many capabilities. These devices are not cheap and are too complex to operate for beginners. A more affordable and user friendly solution is desirable.

In this thesis, we create a low-cost protoype setup using an Arduino and a BMP280 temperature and pressure sensor. Next we attach the Bus Pirate, a small device used for debugging and analyzing chips. We explore if this device can be a viable alternative to a logic analyzer, when sniffing on the I2C and SPI protocols.

Our results suggest that the Bus Pirate is able to intercept both I2C and SPI communications. For I2C it sometimes drops a few bytes, but it works reasonably well overall. With the SPI sniffing mode, the Bus Pirate could reliably intercept all communications.

Contents

Chapter 1

Introduction

In this chapter we will describe the problem and the relevant background. We will also list the hardware that we will use and explain why we chose specific components.

1.1 Problem description

I2C and SPI are protocols that are commonly used for intra-board communication (when two components on the same board need to talk to each other). Hardware security researchers often try to intercept these communications, to gain insight into what a particular device is doing. This can help identify possible security weaknesses.

For people that are interested in learning about hardware security, it is often beneficial to get some hands-on experience. Unfortunately it is not obvious where to begin. One problem is deciding what device to analyze. Most people don't have spare routers, smartphones or game consoles laying around. Moreover, such devices are generally too complicated to analyze for people with no experience. The traditional method of intercepting bus communication data is to attach tiny probes to a chip on an embedded device. The wires on these chips are often unmarked, since they are so small. Additionally, the connections these probes make are not very reliable. They can be disconnected by accident, which makes the learning process more frustrating.

Another issue is that logic analyzers, the devices most commonly used to inspect digital communications, are expensive and often difficult to operate. Many logic analyzers have specifications that are far above the level that is required for beginners. There are variants that include a built-in oscilloscope, or a hard drive to store measurements. For analyzing a less complex system, these features are redundant. A simpler and more affordable device would be more suitable.

The first issue can be addressed relatively easily. There already exists an ecosystem of microcontrollers that we can use to create our own target device. The Arduino platform is an open source project that aims to create an environment that makes it easy for anyone to create software/hardware projects. Their site contains many guides, documentation and a forum where people can ask questions [3]. There are several models available, that all have a small form factor and include clearly labeled connectors. These connectors can connect to so-called 'DuPont wires'. These small wires are very easy to attach and remove, yet the connections they make are strong enough that they will not be undone accidentally. Because of the popularity of the Arduino project, there are many peripheral sensors with dedicated libraries that work with the Arduino boards. In the first part of our research we will describe a setup with one of these sensors, called the BMP280.

The second issue will be addressed by using the Bus Pirate, shown in Figure 1.1c. This is a small board designed by a company named Dangerous Prototypes. Its purpose is to debug and prototype other boards. The Bus Pirate claims to be able to intercept I2C and SPI communications, using a dedicated 'sniffing mode'. At a price point of around €30, it is a lot less expensive than a typical logic analyzer, which can cost hundreds or even thousands of euros. We will investigate whether the Bus Pirate can be used as an alternative to a logic analyzer in the setup that we will be creating.

The questions that we will answer in this thesis are:

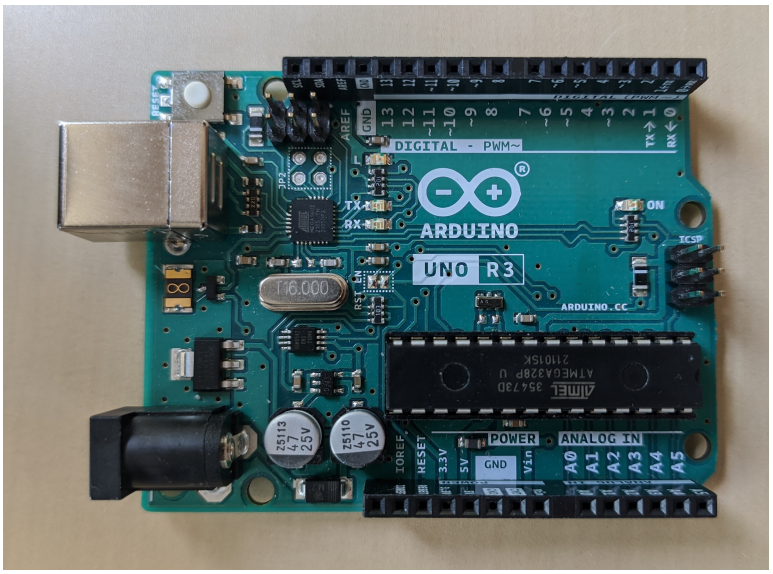
- Is it possible to connect a BMP280 sensor to an Arduino?
- Can the Bus Pirate be used to capture traffic on the I2C bus?
- Can the Bus Pirate be used to capture traffic on the SPI bus?

Our hypothesis is that we will be able to create and describe a simple setup with the Arduino and the BMP280. Such a setup would be useful for learning purposes. We also hypothesize that the Bus Pirate can successfully capture I2C and SPI traffic on this setup.

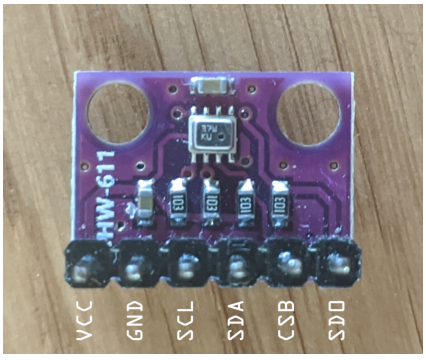
1.2 Hardware

For our setup, we will use the Arduino UNO, shown in Figure 1.1a. This particular model has a very large number of input/output pins, which means that it can connect to two different I2C devices at the same time. For SPI additional steps will be necessary, which will later be explained in Section 3.4. The device can be purchased from the Arduino store for €24.

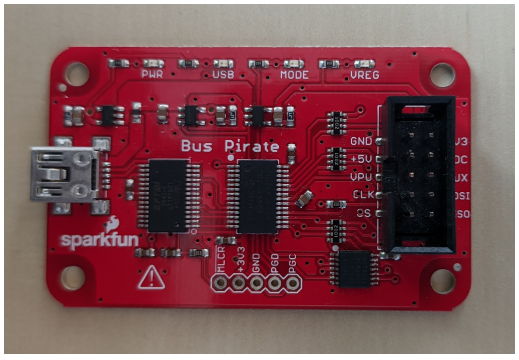
The idea behind our setup is that we can connect our peripheral device and the Bus Pirate at the same time. The Bus Pirate should then be able to pick up on any communications. Sniffing on protocols in this way is far more



(a) Arduino UNO



(b) BMP280



(c) Bus Pirate

Figure 1.1: Hardware used in this project

convenient than in an embedded device, while still being realistic enough that it does not diminish the educational value of such a project.

The peripheral device we will use is a BMP280 pressure and temperature sensor from Bosch, with model number HW-611 [6]. The sensor operates at 3.3 Volt, which the Arduino can provide. The chip has 6 pins, which are labeled in Figure 1.1b. With these pins, the sensor can communicate via the I2C and SPI protocols. The sensor is quite cheap, costing only €5.

Finally, there is the Bus Pirate, which we will use to intercept the communications between the Arduino and the BMP280. The device has 10 pins on the right side and 4 LED indicators on top, as can be seen in Figure 1.1c. There are several versions of the Bus Pirate. For our research we will use version 3.6.

In Chapter 2, we will explain the I2C and SPI protocols. Then in Chapter 3 we describe how to connect the BMP280 to the Arduino and attempt to use the Bus Pirate on such a setup. Next, in Chapter 4 we show how this relates to previous works. Finally, we will summarize our findings in the conclusion in Chapter 5.

Chapter 2

Preliminaries

In this chapter we will briefly explain the I2C and SPI protocols. We will need this to understand the communications that the Bus Pirate intercepts.

2.1 I2C

I2C is a very simple bus protocol that is commonly used by chips communicating on the same board. It only consists of two wires; `CLK` (clock) and `SDA` (data). If during a certain clock cycle `SDA` is low, then it is read as a logical `0`. Conversely, a high `SDA` signal means a logical `1`. Devices that communicate using the I2C protocol can either be a *controller node* or a *target node*. The controller node will initiate the communication, terminate the communication and provide the clock signal. The target node receives the clock and responds when it is addressed. I2C is a multi-controller bus protocol. This means that multiple devices can connect to the same data bus, and they can even alternate between being a controller or a target [15].

There are also two special signals called START (S) and STOP (P). START is defined as a high to low transition on `SDA`, while `CLK` is high. STOP is defined as a low to high transition, while `CLK` is high. These signals are used to indicate the beginning and end of a data exchange. In cases where multiple devices take on the role of controller node, these signals provide clear boundaries between communications.

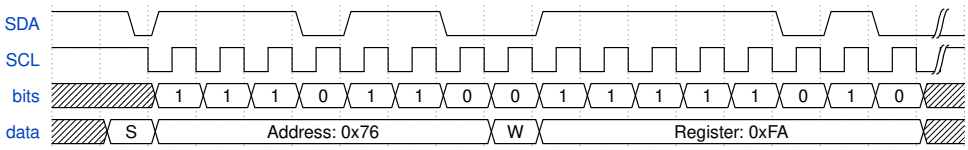
To address a specific device, a unique 7-bit address is sent by the controller node. For instance, the BMP280 that we are using has an address of `0x76` and it will only respond if this address immediately follows the START signal. After the address a single bit is sent to indicate whether the controller node wants to read data (`1`) or write data (`0`).

After each byte, the receiving party has to send an ACK/NACK bit, indicating whether they have received the data. To reply with an ACK, the `SDA` line has to be actively held low. The NACK is sent by keeping `SDA` high. There are several reasons why the sending party might get a NACK after

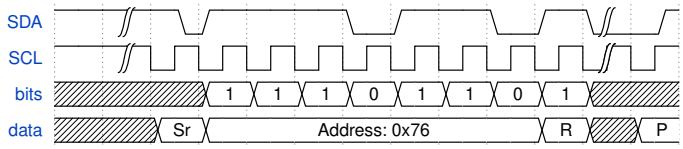
sending a byte. For example, if the device does not exist on the bus or the device was unable to read the data due to an overfull buffer.

Because the BMP280 has several registers, the controller node can't simply send a single read request. How would the device know what register the controller wants to read from? To solve this, a frequent pattern is the write-then-read transaction. First, the controller writes the register address to the target node and then it starts a new read instruction to the same device. It does this without sending a STOP signal, which is a special case, called repeated START (Sr).

With this knowledge, we can craft the instruction to read the temperature from the BMP280. We know that the BMP280 has an address of `0x76` and from the datasheet we can also find that the temperature register has an address of `0xFA` [6]. From this, we can create the transaction, as shown in Figure 2.1.



(a) We write the register address to the device address of the BMP280



(b) Immediately afterwards we request a read from the BMP280

Figure 2.1: A full write-then-read transaction

One problem that may arise is if two devices try to be a controller at the same time. To resolve this issue, the I2C specification defines an arbitration protocol which solves any conflicts. The process works as follows; two devices that "fight" for control over the bus will send their data bit by bit. After each bit, they will check if the data line (`SDA`) matches with what they have sent. If it does, they send the next bit. But if it doesn't then the device that set `SDA` to high will lose the arbitration. It has to restart their transmission, as soon as the bus is free. Because arbitration only stops if there is a conflict, transmissions sent by two devices at the same time may actually succeed, if they are identical.

2.2 SPI

Serial Peripheral Interface, or SPI, is a communication protocol that is also commonly used in embedded devices, just like I2C. SPI is slightly more complicated, since it requires a minimum of 4 wires, instead of 2. These wires are Serial Clock (`SCLK`), Master Out Slave In (`MOSI`), Master In Slave Out (`MISO`) and Chip Select (`CS`). `MOSI` and `MISO` on the Master device are connected to the `MOSI` and `MISO` wires on the Slave device. Unlike I2C, SPI does not use addressing to determine which device should receive certain communications. Instead, SPI uses the `CS` wire. A Slave device will only listen to communications if their `CS` line is a logical `0` [13].

SPI communications happen via two shift registers. One in the Master and one in the Slave. At every clock cycle, the Master will send out the least significant bit over the `MOSI` line and the Slave sends out the least significant bit via the `MISO` line. Both Master and Slave shift the contents of their registers to the right by one and then set the most significant bit to the bit that was transmitted by the other device. This process continues until the contents of the registers have been swapped and new values can be loaded into the registers.

To start communication, the Master must configure certain clock parameters. First there is the frequency of the clock, which must also be supported by any slave devices. Second there is the clock polarity, or `CPOL`. A `CPOL` of 0 means that the clock idles at `0` and each cycle consists of a pulse of `1`. A `CPOL` of 1 means the exact opposite. That is, the clock idles at `1` and a cycle consists of a pulse of `0`. Finally, there is the clock phase, or `CPHA`. This parameter determines at what point during a clock cycle the data is sent and at what point it will be read. With a `CPHA` of 0, data is sent at the trailing edge of a clock pulse. A `CPHA` of 1 means that data is sent at the leading edge of a pulse.

One benefit of SPI over I2C is that communication can happen in parallel, because there is a data wire for both the Master and the Slave device. Another benefit is the flexibility of SPI. Messages don't have to come in 8-bit words and devices are free to chose the purpose of each message. This allows for some additional variations of SPI. One such variation is Dual SPI, where after a certain instruction from the Master device, both `MISO` and `MOSI` will be used to sent data from one device to the other. This allows two bits to be sent at the same time.

Chapter 3

Research

In this chapter we show how to use the Bus Pirate to spy on communications between the Arduino and a peripheral device. We will connect the BMP280 to the Arduino using the I2C and SPI protocols. Then we demonstrate that the Bus Pirate can be used on these setups to intercept data sent between the primary and the secondary device. We will also show how to flash code, or 'sketches' as Arduino developers like to call them, to the Arduino UNO.

3.1 Connecting the BMP280 via I2C

The thermometer we have operates at 3.3 Volt, which the Arduino UNO can supply. The chip can interface with both I2C and SPI devices, but for now we will be using the I2C mode. We connect the chip to the Arduino by wiring it up as shown in Figure 3.1.

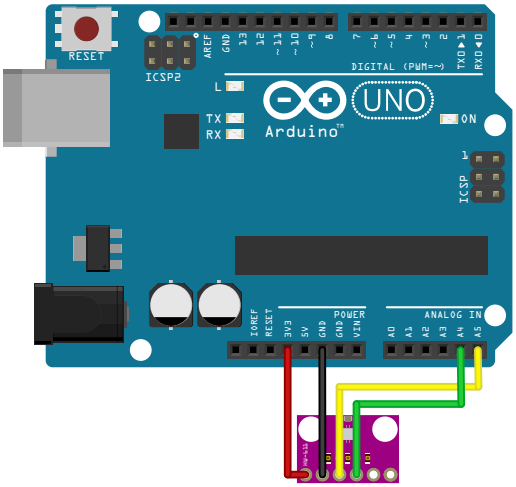


Figure 3.1: Connecting the BMP280 to the Arduino using I2C

3.1.1 Uploading the sketch to the Arduino

Now that the hardware side is taken care of, we will turn to the software. The BMP280 is a fairly popular chip, which means that there are quite some libraries online for it. The sketch we created is shown in Appendix A and it is inspired by a sketch from the Adafruit BMP280 library [1]. For our script, we removed the pressure and height readings. Intercepting these works in a similar way as the temperature, since they communicate via the same I2C bus.

To upload our sketch, we will use beta version 2.0.0-rc6 of the Arduino IDE [4]. We found that this version is stable, while also providing improvements over the regular version, such as a monospaced font and a better user interface in general. These instructions were however also tested successfully on version 1.8 of the Arduino IDE.

We will first need to install the Adafruit BMP280 library [1]. To do this, we click on the 'Tools' option in the menu bar and then on 'Manage Libraries...'. Now a search field should appear. We can find the required library by typing its name, 'Adafruit BMP280 Library'. From there we can install version 2.6.2, as can be seen in Figure 3.2. A pop-up might appear asking if you want to install all the dependencies of this library. We do want this, so select 'Install all'.

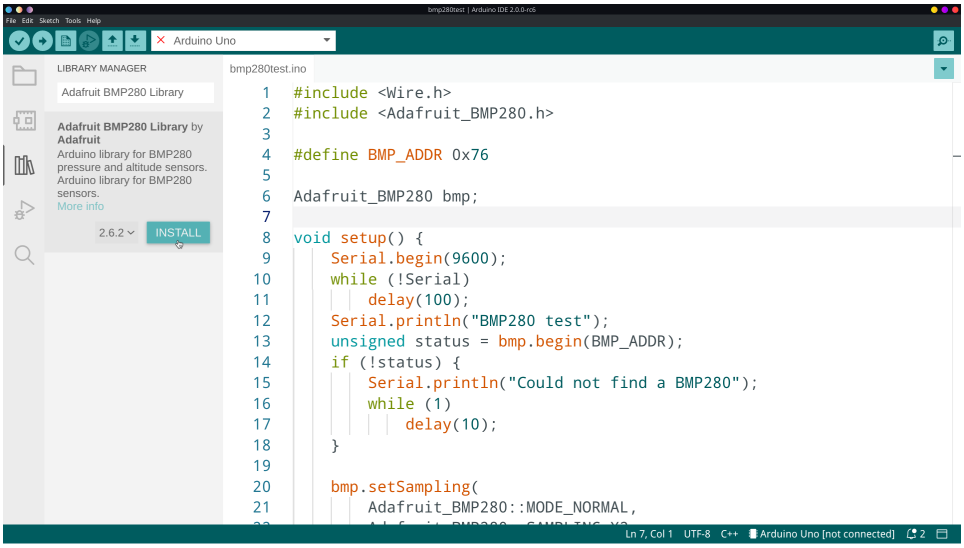


Figure 3.2: Installing the Adafruit BMP280 Library

Now we can connect the Arduino UNO to our computer. For this we need a USB type B cable. Once the device is plugged in, we should be able to select it in the Arduino IDE, like in Figure 3.3. Now we can click the circular arrow button in the top left. This will compile our sketch and upload it to the Arduino. The BMP280 should now start measuring the temperature.

To see the output, we can open the Arduino's serial monitor. This can be found by first clicking on 'Tools' in the menu bar and then selecting 'Serial Monitor'. The readings of the thermometer are displayed in Figure 3.4. The chip was held between my fingers, which explains the rising temperature.

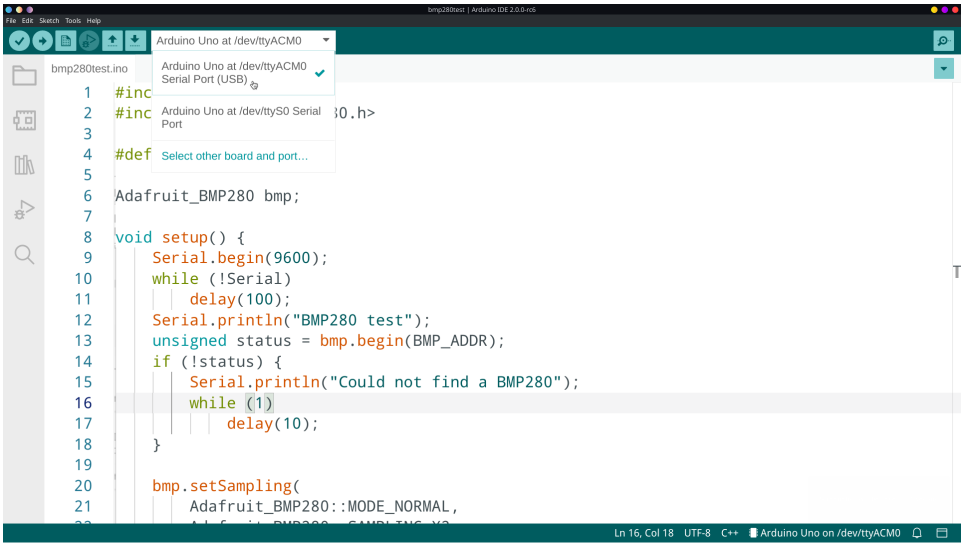


Figure 3.3: Selecting the Arduino UNO board

Figure 3.4: The output of the BMP280 when held between my fingers

3.1.2 Setting up the Bus Pirate

We need to be able to communicate with the Bus Pirate. To do so, we will be installing a program that can talk to devices over a serial connection.

Before we install this program however, there is one optional step that makes working with the Bus Pirate easier on Linux. Normally, the device is mounted at `/dev/ttyUSB[0-9]`, where the last digit can change depending

on how many other USB devices are connected to the computer. Linux has the *udev* system, which can simplify things a bit by recognizing the Bus Pirate and creating a symbolic link to `/dev/buspirate`. That way, we can communicate with our device under this new name, instead of having to figure out the last digit of the `ttyUSB` file. To achieve this, we create the following file, consisting of a single line.[1]

/etc/udev/rules.d/98-buspirate.rules

```
SUBSYSTEM=="tty", ATTRS{idVendor}=="0403",
↪ ATTRS{idProduct}=="6001", GROUP="users", MODE="0666",
↪ SYMLINK+="buspirate"
```

The idea for this file comes from the ArchWiki [8]. This rule is specifically meant for version 3.x of the Bus Pirate.

Now we will be installing *picocom*, a serial console program that can talk to Bus Pirate [16]. On Linux, the software can be installed via the package manager. For instance, on Debian based distributions, one can run `apt install picocom`. If we now connect the Bus Pirate to our computer with a type B mini USB cable, we should be able to communicate with it by running the following command:

Initiate communication with the Bus Pirate

```
$ picocom --baud 115200 /dev/buspirate
```

We use `--baud` to specify the baud rate (symbols per second) that the device expects, which in this case is `115200`. After executing this command, picocom will display some status information. Pressing the enter/return key will display `HiZ>`, which is the default prompt. According to the Bus Pirate documentation, *HiZ* stands for 'high impedance mode' [9]. Here, all outputs are disabled to prevent attached devices from reaching conditions that are not according to their specification.

[1] From now on, we will assume that this file has been created and thus we can refer to the device as `/dev/buspirate` in any future commands.

If we enter `?` in the prompt, we will get a list of all possible commands.

```
HiZ>?
General                              Protocol interaction
-------------------------------------------------------------------------
?       This help                    (0)     List current macros
=X/|X   Converts X/reverse X         (x)     Macro x
~       Selftest                     [       Start
#       Reset                        ]       Stop
$       Jump to bootloader           {       Start with read
&/%     Delay 1 us/ms                }       Stop
a/A/@   AUXPIN (low/HI/READ)         "abc"   Send string
b       Set baudrate                 123
c/C     AUX assignment (aux/CS)      0x123
d/D     Measure ADC (once/CONT.)     0b110   Send value
f       Measure frequency            r       Read
g/S     Generate PWM/Servo           /       CLK hi
h       Commandhistory               \       CLK lo
i       Versioninfo/statusinfo       ^       CLK tick
l/L     Bitorder (msb/LSB)           -       DAT hi
m       Change mode                  _       DAT lo
o       Set output type              .       DAT read
p/P     Pullup resistors (off/ON)    !       Bit read
s       Script engine                :       Repeat e.g. r:10
v       Show volts/states            .       Bits to read/write e.g. 0x55.2
w/W     PSU (off/ON)       <x>/<x= >/<0>   Usermacro x/assign x/list all
```

The following commands are particularly useful to us:

- `[` and `]` represent the I2C START and STOP signals that we discussed in Section 2.1. We will see these brackets a lot when intercepting I2C communications.
- `(0)` and `(x)` are used to list or activate macros. For I2C specifically, there is an 'I2C sniffer' macro, that we will be utilizing.
- `m` is used to change the mode of the Bus Pirate, because the device supports many different protocols. We use this to switch from the standard high impedance mode to I2C and SPI modes.
- `r` and `:` will be used to read bytes from certain registers on the BMP280.

To end our communication with the Bus Pirate, we put the device in *HiZ* mode, if it isn't already. We can achieve this by entering `m` followed by `1`. To exit from picocom, we first press the 'Ctrl' and 'A' keys and then the 'Ctrl' and 'X' keys.

We will now expand the setup shown in Figure 3.1 to also include the Bus Pirate. First ensure that the Bus Pirate and Arduino are unplugged, to prevent any damage to the components. Using three additional wires, we can create the setup in Figure 3.5.

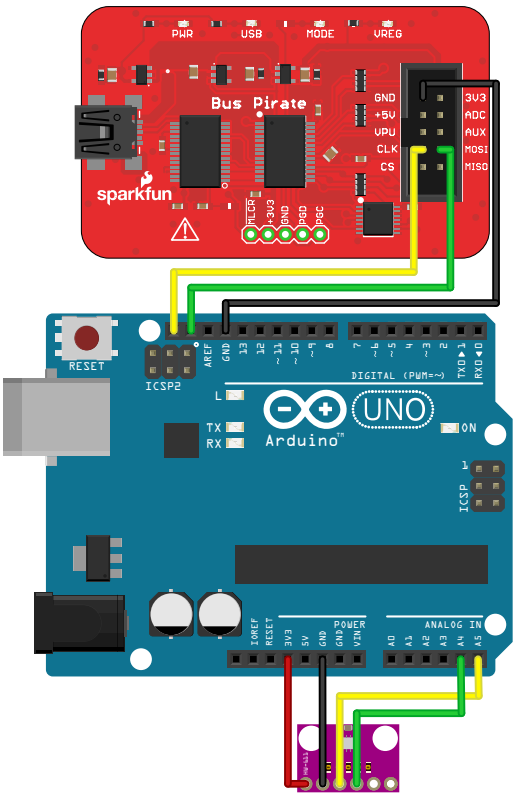


Figure 3.5: Connecting the Bus Pirate to the I2C setup

3.2 Intercepting I2C communications

We can now connect both the Arduino and the Bus Pirate to our computer via USB. Next we start the Arduino IDE and picocom. This will allow us to see the data that the Arduino reads in the serial monitor and the data that the Bus Pirate reads with the I2C sniffer macro. We will enter the following commands inside picocom. First enter `m` to select a mode. We then see a list of different modes to chose from. Select I2C mode by entering `4`. Now a speed is expected. Enter `3` to chose a speed of `100KHz`. The prompt should now be changed to `I2C>`, indicating that we have successfully switched to the I2C mode. To view a list of all available macros in this mode, we can enter `(0)`. We see that there's two macros. The I2C sniffer is listed second, so we activate it by typing `(2)`.

The sniffing macro now outputs some hex code every time the Arduino requests the temperature from the BMP280.

```
I2C>m
1. HiZ
2. 1-WIRE
3. UART
4. I2C
5. SPI
6. 2WIRE
7. 3WIRE
8. LCD
9. DIO
x. exit(without change)

(1)>4
Set speed:
 1. ~5KHz
 2. ~50KHz
 3. ~100KHz
 4. ~400KHz

(1)>3
Ready
I2C>(2)
Sniffer
Any key to exit
[0xEC+0xFA+][0xED+0xFE-0x84+][0xEC+0xFA+[0xED+0xFE-0x81+][0xEC+]
[0xED+0xFE-0x81+][0xEC+0xFA+][0xED+0x7F+0xE0+0x00-]
```

In these sniffed bytes we can see the write-then-read transaction that we discussed in Section 2.1. In hexadecimal, `0xEC` is BMP280's address (`0x76`) with a write instruction (`0`) appended to it. We then see `0xFA`, the register that contains the temperature. In the next transmission, we see `0xED`, which is the BMP280's address followed by a read instruction (`1`).

Now we run into an inconsistency. If we look at the source code for the Adafruit library, then we can see that it reads from the temperature register by using a function called `read24` [1]. This is named for the fact that it reads 24 bits, so three bytes. Yet, if we look at the bytes we intercepted, we can see that most of the time the read instruction (`0xED`) is only followed by two bytes. We're not sure why this is happening. It might be due to the fact that the I2C sniffing mode is still in alpha mode and thus it is not completely reliable. Or maybe the Arduino optimizes the number of bytes it has to send in a certain way. We could not find any evidence for this though. For now, we will let the Sniffer mode run until we see a block that has the `0xED` byte, followed by three other bytes.

Another thing we noticed is that the Arduino does not always send a repeated START signal. Perhaps the reason for this is that it sometimes tries to leave the bus free for another device to send a message.

We are now ready to decode the bytes. One problem is that the values are analog, instead of digital. According to Bosch, each sensing element can behave differently. Therefore, each BMP280 has some registers that it uses, in order to 'compensate' these raw values [6]. These registers are called `dig_T1`, `dig_T2` and `dig_T3`. In the Bus Pirate's I2C mode, it is possible to enter I2C messages directly. We will use this to create our own write-then-read instructions and send those to the BMP280. This will then give us the values of the registers.

Entering the read instructions from Table 3.1, we will get some verbose output that contains all the data that was written/read. The line we are interested in starts with "READ:", followed by two register bytes. We need to convert these bytes to the correct data type, as displayed in the table. The helper script in Appendix B can be used for this. For each register, enter the two output bytes in the corresponding prompt and the script will compute the short values, taking into account whether they should be signed or unsigned.

register address	content	data type	read instruction
`0x88`	`dig_T1`	unsigned short	`[0xEC 0x88 [0xED r:2]`
`0x8A`	`dig_T2`	signed short	`[0xEC 0x8A [0xED r:2]`
`0x8C`	`dig_T3`	signed short	`[0xEC 0x8C [0xED r:2]`

Table 3.1: Registers needed to compensate the analog temperature reading

In Figure 3.6 this process is illustrated. In the top terminal window, we are communicating with the Bus Pirate in I2C mode. The bottom terminal shows our interaction with our helper script, that can convert the raw bytes to (unsigned) short values. First we have to enter the read instructions in the top window. These instructions are highlighted in blue. Each instruction gives us a few lines of output. For us, only the lines that are highlighted in pink are relevant. These contain the bytes that were read from the BMP280's registers. We can now run the script, which will ask for the register bytes. Because our script uses a regular expression to parse out the bytes, it does not matter if we enter the entire line, or just the bytes. This can be seen in the lines that are highlighted green. Finally, in orange, we get the output values we are looking for.

In our case, `dig_T1 = 28095`, `dig_T2 = 27081` and `dig_T3 = -1000`. We can enter these values in the script in Appendix C, which makes it ready to be executed. This script uses the same calculations that are described in the BMP280's datasheet [6]. These calculations will convert the raw bytes

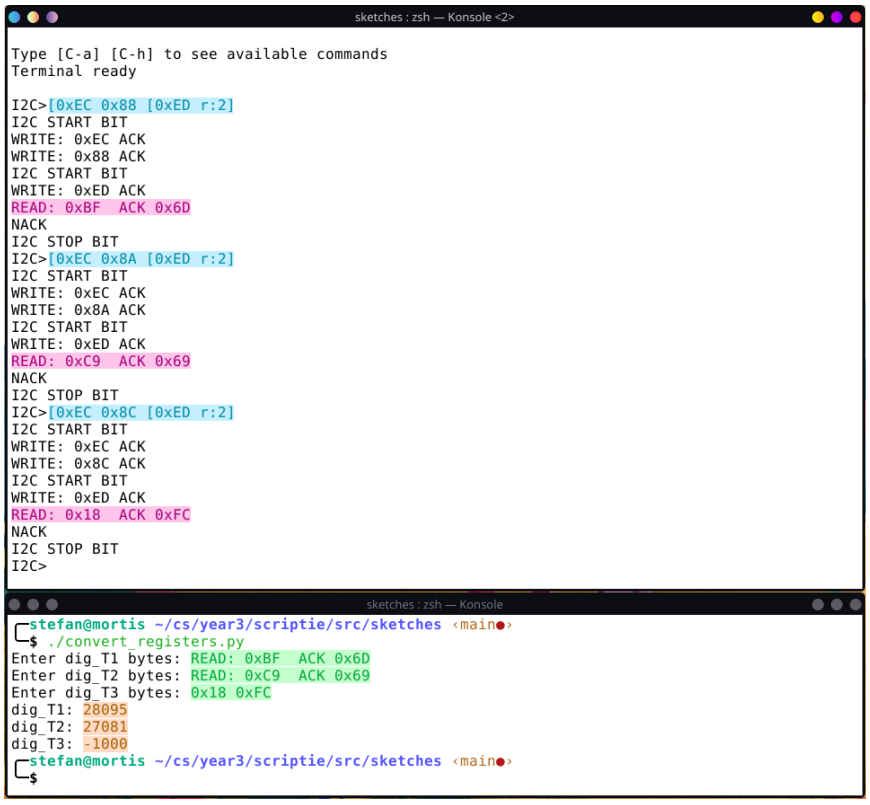


Figure 3.6: Converting the raw register bytes to (unsigned) shorts

that we intercepted to the corresponding temperature. Upon execution, the script will ask for the sniffed bytes. Here we enter `0x7F 0xE0 0x00`, which is the last output of the Bus Pirate that we saw earlier. The script now shows the decoded temperature for these bytes; in this case 23.91 °C. We have now successfully intercepted and decoded I2C communications between the BMP280 and the Arduino.

3.3 Inspecting I2C traffic with a logic analyzer

We also intercepted the traffic between the BMP280 and the Arduino with a logic analyzer. We used a Saleae Logic Pro 16, which can capture I2C, but it cannot display it because it does not have a screen. Instead we use a piece of complementary software called *Logic 2* to analyze the capture. Using some miniature alligator clips, we connected the logic analyzer to the wires on our setup. Next we started a recording in Logic 2.

The view in Figure 3.7 shows two lines, the clock line (on top) and the data line (on the bottom). Because the program has an integrated I2C mode, we see several labels above the `SDA` line, indicating what is being sent. These

labels correspond exactly to a successful reading with the Bus Pirate.

The green circles represent (repeated) START signals. These consist of a high to low transition on the data line and a high signal on the clock line. As explained in Section 2.1. The orange square represents a STOP signal, which is a low to high transition on the data line, while the clock line is high. Notably, the last byte is not acknowledged. This does not seem to be accidental, as it happens in all communications we inspected. A possible explanation could be that the Arduino (which should do the acknowledging in this case) will stop the communication anyway and therefore the library implementation does not bother to send the ACK signal.

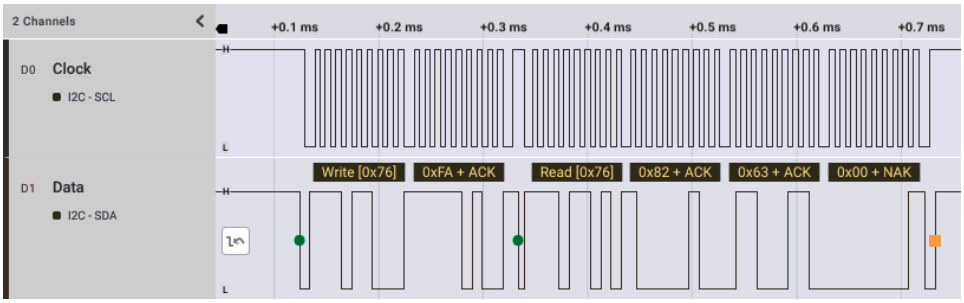


Figure 3.7: I2C measurements done with the Saleae logic analyzer

A difference with the Bus Pirate is that the inconsistency mentioned in Section 3.2 is not present with the logic analyzer. During our capture of 5 seconds, there are no communications where there are only two bytes read. All of the transactions are in the exact same format. This is an indication that the incorrect readings are indeed a flaw in the Sniffer Mode of the Bus Pirate.

If we look at the transmission in Figure 3.7 in more detail, then we see that it took 0.6 ms to send 6 bytes. This corresponds to a rate of 1 bit every 12.5 µs. With this information, we can now compute the frequency of the signal. $\frac{1}{1.25 \cdot 10^{-5}} = 80\text{kHz}$, which is lower than the speed that we set for the Bus Pirate's I2C sniffing mode. This means that the dropped bytes are not caused by insufficient speed of the Bus Pirate. Even at the fastest mode (400kHz), we observed dropped bytes.

3.4 Connecting the BMP280 via SPI

The BMP280 also supports SPI. In this section we will describe how to connect the sensor and investigate if we can also use the Bus Pirate to intercept SPI communications.

The wiring of the BMP280 to the Arduino is shown in Figure 3.8. Since SPI is slightly more complicated than I2C, the base setup requires two more wires. Now we can flash the sketch from Appendix D to the Arduino. The code does the same as the I2C version, but there are some slight differences in how we initialize the BMP280. To flash the sketch, we can again use the procedure described in Section 3.1.1.

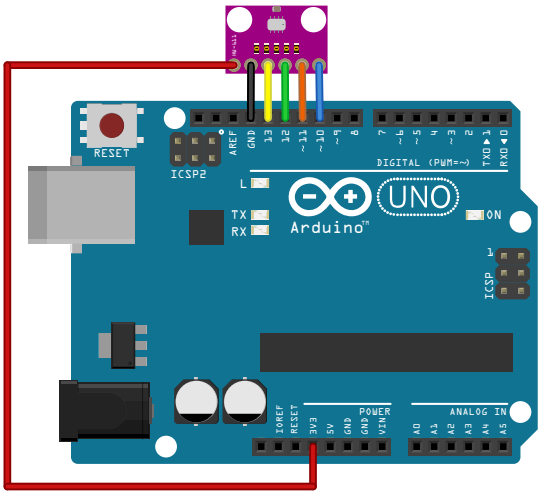


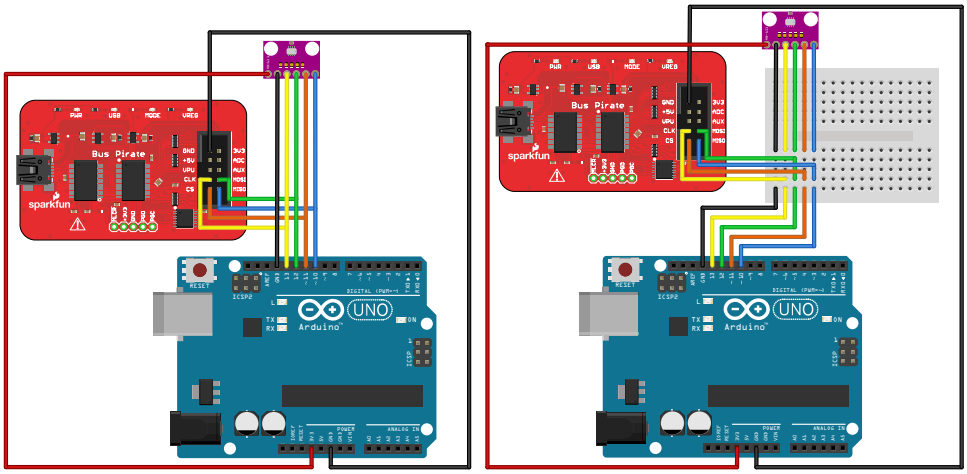
Figure 3.8: Connecting the BMP280 to the Arduino via SPI

After flashing the code, temperature readings should start showing up in the serial monitor of the Arduino IDE. This means that we are ready to add the Bus Pirate to the setup.

3.5 Intercepting SPI communications

With I2C, connecting the Bus Pirate was simple. This is because the Arduino UNO has two `SCL` pins and two `SDA` pins, as shown in its datasheet [5]. Pins `A4` and `SDA` are internally connected and so are pins `A5` and `SCL`. We used this fact to connect the Bus Pirate to the Arduino directly, while reading the same I2C bus. There is no such duplication present for the SPI protocol on the Arduino. We can use the following two options to insert the Bus Pirate between the Arduino and the BMP280:

1. using splitter cables, as shown in Figure 3.9a, or
2. using a breadboard, as shown in Figure 3.9b.



(a) Using custom splitter cables (b) Using a breadboard

Figure 3.9: Different ways to connect the Bus Pirate to the SPI setup

Option 2 is likely easier, as it requires no soldering. At the time of writing however, we did not have access to a breadboard, therefore we went for option 1 first. To create the cables, we followed a similar method as described in a blogpost by Micheal Schoeffler [12]. The resulting cable is shown in Figure 3.10. In total, we need 4 of these wires to create the setup. Each should have one male and two female connections.

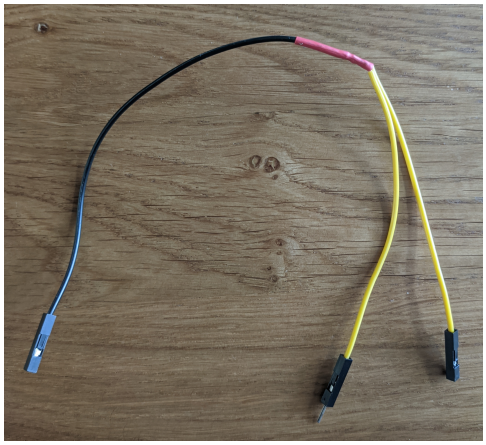


Figure 3.10: A custom splitter cable

Now that we have the Bus Pirate connected, we will connect the Arduino and the Bus Pirate to our computer via USB. Using the same picocom command as before, we can start the communication with the Bus Pirate. This time we will be setting it to the SPI mode. We will be asked to set a few parameters, such as the speed, clock polarity and output clock edge, as can be seen in the following transcript. The parameter values we chose can also be found in Table 3.2, for clarity.

```
HiZ>m
1. HiZ
2. 1-WIRE
3. UART
4. I2C
5. SPI
6. 2WIRE
7. 3WIRE
8. LCD
9. DIO
x. exit(without change)

(1)>5
Set speed:
 1. 30KHz
 2. 125KHz
 3. 250KHz
 4. 1MHz

(1)>3
Clock polarity:
1. Idle low *default
2. Idle high
```

```
(1)>1
Output clock edge:
 1. Idle to active
 2. Active to idle *default

(2)>1
Input sample phase:
 1. Middle *default
 2. End

(1)>1
CS:
 1. CS
 2. /CS *default

(2)>2
Select output type:
 1. Open drain (H=Hi-Z, L=GND)
 2. Normal (H=3.3V, L=GND)

(1)>2
Ready
```

Parameter	Value	Option #
Speed	250KHz	3
Clock polarity	Idle low	1
Output clock edge	Idle to active	1
Input sample phase	Middle	1
CS	/CS	2
Select output type	Normal	2

Table 3.2: Parameters for the SPI mode of the Bus Pirate

With the Bus Pirate in the correct mode, we can start the sniffer. If we enter `(0)` to list all available macros, we see that there are two options. We can either sniff all the traffic (macro `2`), or sniff only when `CS` is low (macro `1`). Since `CS` is only low whenever the Arduino communicates with the BMP280, it makes sense to chose macro `1`. If we do this, then we get the following output in picocom.

```
SPI>(1)
Sniffer
Any key to exit
[0xFA(0x00)0xFF(0x83)0xFF(0xD7)0xFF(0x80)]
[0xFA(0x00)0xFF(0x83)0xFF(0xD8)0xFF(0x80)]
[0xFA(0x00)0xFF(0x83)0xFF(0xD9)0xFF(0x80)]
```

The bytes between parentheses are sent from the BMP280, while the remaining bytes are from the Arduino. The Arduino first sends `0xFA`, the register address that it wants to read from. At the same time, the BMP280 sends a zero byte, which we can ignore. After that, the BMP280 sends the bytes from its temperature register, while the Arduino just keeps its data line high, since it has nothing to send. This shows up as only `0xFF` bytes. If we take out only the bytes from the last BMP280 measurement, we get `0x83 0xD9 0x80`. We can use the same decoding script that we used for I2C to decode the temperature. Running the script gives us an output of 29.13 °C, which is also the temperature that the serial monitor in the Arduino IDE shows.

For completeness, we also built the setup in Figure 3.9b. This setup performed the same as the one with our splitter cables. In both, the Bus Pirate is able to intercept the traffic between the Arduino and the BMP280. The SPI sniffing mode of the Bus Pirate proved to be very reliable. We observed no dropped bytes, like with I2C.

3.6 Inspecting SPI traffic with a logic analyzer

We also inspected the SPI communications with the Saleae Logic 16 Pro. We connected the logic analyzer using the clips that come with it. This time, we measured while the Bus Pirate was still connected, which creates the setup shown in Figure 3.11. The red box is the logic analyzer. The alligator clips connect between the connections on the Arduino and the wires themselves.

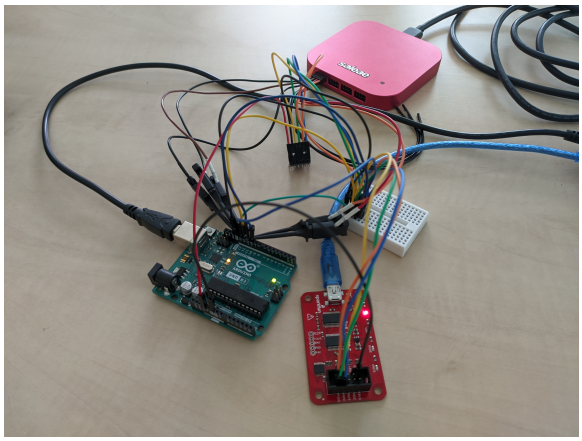


Figure 3.11: Sniffing on the SPI setup

The results of our measurements can be seen in Figure 3.12. Unlike with I2C, the clock does not run continuously. This is likely explained by the fact that with SPI, both devices have to load a new value in their registers, as we explained in Section 2.2. Loading these registers probably takes some time, which is why we see the pauses.

Looking at the second group of clock cycles, we can see in the label that one cycle takes 4.96 µs. This means that the frequency of the clock signal is $\frac{1}{4.96 \cdot 10^{-6}} \approx 200\text{kHz}$. This is quite a bit faster than I2C, but still well within the range that the Bus Pirate can measure.

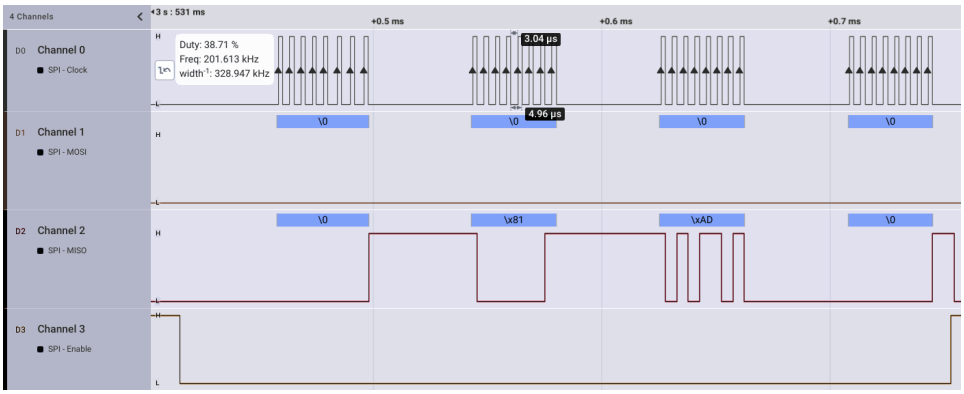


Figure 3.12: SPI measurements done with the Saleae logic analyzer

One inconsistency we see is that the `MOSI` line is constantly low. This would mean the Arduino never sends any data. But this is not what we see when we measured with the Bus Pirate. There, the Arduino always sent `0x76` as a first byte, to indicate that it wants to read from the temperature register. We believe that this data is missing. Likely the alligator clip that we used did not make a proper connection.

Chapter 4

Related Work

Some research has been published that includes the Bus Pirate. Nadir et al. describe an auditing framework for IoT devices [14]. They list three attack vectors: communications, firmware and hardware. The idea is to first analyze these facets and then audit them using both software and hardware tools. The authors mention that the Bus Pirate can be used to audit on chip memory by using the debug ports of an IoT device. While a method is not explicitly described, this could possibly be achieved by using the SPI sniffing mode, since flash memory often uses the SPI protocol.

Another usage of the Bus Pirate is presented by Allwright et al. [2]. In the paper, they describe a system of robots that can assemble structures with electronic cubes, that they call *Stigmergic blocks*. The robots and the blocks both use ATMega328P microcontrollers. In order to install the bootloader to these devices, they use a Bus Pirate and a terminal program called `avrdude`. To achieve this, they connect the Bus Pirate to the ICSP port on the ATMega328P, which is a port specifically meant to reprogram microcontrollers while they are in a complete system. This research shows how versatile the Bus Pirate can be.

The Bus Pirate is also used by Kwonyoup Kim et al. [11]. For their research they attempted to crack the AES encryption for a wireless keyboard. First they disassemble the keyboard and connect the Bus Pirate to the debug ports of the flash memory. Then they use a program to get the firmware from the device. Next they try to reverse engineer the keyboard's software, focusing specifically on the encryption functions. They modify the firmware slightly, to prepare for a side channel analysis on the signal traces of the keyboard. Here they try to obtain the AES key, based on the electromagnetic signals that the keyboard sends.

Iwen Coisel et al. utilized the Bus Pirate to demonstrate that they could break DECT encryption, which is commonly used in cordless phones [7]. They used the Bus pirate on several phones to extract the contents of EEPROM chips (non-volatile memory). These chips used both the I2C and

SPI protocols. Their cryptanalysis attack proved successful, as it had better success probabilities than the previous best solution.

When it comes to the security of I2C in general, Mohammed Khelif et al. describe two attacks that can be used against I2C buses [10]. The first one is a *heartbleed* type attack, where the attacking device will read the address from the first byte of an I2C read instruction from the controlling node. Then, it will keep a synchronized clock going and send acknowledgments back to the target node. The attacking device can continue this process, even after the actual controller node has already stopped sending a clock signal. This way, the controlling node will keep reading new information, even though this was not the intention. The second attack is a buffer overflow attack. This attack is similar to the first one, because they again take control of the clock. The difference is that now the controlling device is writing data, instead of reading data. The attacking device forces the controller device's data line to `0`, in order to prevent a STOP signal. This also makes it possible for the attacker to write their own data to up to one page of memory.

Two blogposts worth mentioning are those made by Robert Seger. Though these are not peer reviewed literature, they do relate closely to our research. In the first one Seger uses a Bus Pirate to interface with an EEPROM chip [17]. Here he uses the scanning macro available in the Bus Pirate I2C mode to discover the address of the device. Later he writes and reads some bytes to the memory using the Bus Pirate syntax. The read instruction is similar to the one we used to obtain the BMP280 `dig_T` registers. He then takes things a step further by connecting four EEPROMs to a Raspberry Pi simultaneously. Instead of using a Bus Pirate in this case, he uses a Python library called `smbus`, which is running on the Pi. This library is subsequently used to create code that can read/write to an entire EEPROM in one go.

In the second post, Seger also experiments with more EEPROMs, but this time using the SPI protocol [18]. The second EEPROM he attempts to analyze is embedded in a UPS (Uninterruptible Power Supply). First an attempt is made to solder to the chip's connections, while it is still on the board. This yields no good results. Then the author tries to remove the chip from the board, to analyze it while it is more isolated. At this point he measures some flaws in the EEPROM. Some read-only bits have different values than they should have. The author believes that this chip is likely become defective. This shows how difficult analyzing chips in a real embedded system can be. In the end of the post, he uses a photodiode (light sensor) to create an automatic night light. The sensor is attached to a Raspberry Pi via SPI.

Chapter 5

Conclusions

From our research, we can conclude that it is possible to create a setup with the Arduino UNO and the BMP280. We first demonstrated this using the I2C mode of the sensor. With this setup, we were able to get consistent temperature measurements, which we could read with the serial monitor in the Arduino IDE. Later, we also successfully created a setup where the BMP280 works in SPI mode.

We found that the Arduino UNO was very easy to work with. Since the device is so popular, there are plenty of resources available online. These included datasheets, tutorials and schematics. Often, if we were stuck on something, we were could find a forum post of someone with a similar problem, which helped tremendously. Because of these reasons and the relatively low cost of the device, we think the Arduino was the perfect choice for this project.

Working with the Bus Pirate went smoothly as well. The *udev* rule that we created made it a lot easier to start the communication with picocom. The Bus Pirate's interface was not difficult to understand. We found that the verbose help menu makes it clear what options are available, and how you can switch between different modes.

The I2C sniffing mode of the Bus Pirate worked okay, but we discovered that it is not always reliable. Sometimes it would not read all the bytes that were being sent. We believe that this is due to the fact that the Bus Pirate's I2C sniffing mode is in alpha stage. Another thing to take into account is that the raw bytes that the thermometer sends do not directly contain the temperature. We first had to convert these bytes by using certain register values that we manually extracted from the BMP280. This also holds true for SPI. For understanding what information is relevant in the intercepted bytes, some knowledge of the I2C protocol is required.

The SPI sniffing capabilities of the Bus Pirate worked very well. There were no instances of dropped bytes. The initial setup is a bit more difficult for SPI, because of the fact that it is not possible to directly connect both

the BMP280 and the Bus Pirate to the Arduino simultaneously. Instead, we demonstrated that inserting the Bus Pirate is possible using either splitter cables, or a breadboard. In picocom, there are a more parameters to configure than with I2C. Afterwards, starting the sniffing macro works the same way.

Using a different peripheral device could perhaps make the sniffing a bit easier. It is possible that not all peripheral devices need their readings to be converted in a special way. Perhaps some send data that can be interpreted directly, which makes it easier to identify in the intercepted data. There are many possibilities, such as 7-segment displays, heart rate monitors or RGB color sensors. Investigating whether other devices make I2C and SPI sniffing easier, may be an interesting topic for future research.

Circling back to our research questions, we find that we indeed were able to connect the BMP280 to the Arduino. Sniffing on I2C traffic proved to be possible, although not entirely perfect. SPI sniffing is also possible and works very well.

Bibliography

[1] Adafruit. *Adafruit BMP280 Driver (Barometric Pressure Sensor)*. May 2022. URL: https://github.com/adafruit/Adafruit_BMP280_Library (visited on 05/03/2022).

[2] Michael Allwright, Weixu Zhu, and Marco Dorigo. "An open-source multi-robot construction system". en. In: *HardwareX* 5 (Apr. 2019). ISSN: 24680672. DOI: 10.1016/j.ohx.2018.e00050. URL: https://linkinghub.elsevier.com/retrieve/pii/S2468067218300786.

[3] Arduino. *Arduino - Home*. Aug. 2022. URL: https://www.arduino.cc/ (visited on 08/04/2022).

[4] Arduino. *Arduino IDE 2.x (beta)*. Apr. 2022. URL: https://github.com/arduino/arduino-ide (visited on 05/04/2022).

[5] Arduino. *Arduino® UNO R3 Product Reference Manual*. Mar. 2022. URL: https://docs.arduino.cc/resources/datasheets/A000066-datasheet.pdf.

[6] Bosch. *BMP280 Digital Pressure Sensor*. Oct. 2021. URL: https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bmp280-ds001.pdf.

[7] Iwen Coisel and Ignacio Sanchez. "Improved cryptanalysis of the DECT standard cipher". In: *Journal of Cryptographic Engineering* 6.2 (June 2016), pp. 155–169. ISSN: 2190-8508, 2190-8516. DOI: 10.1007/s13389-016-0127-4. URL: http://link.springer.com/10.1007/s13389-016-0127-4.

[8] ArchWiki Contributors. *Bus Pirate - ArchWiki*. Wiki. Mar. 2022. URL: https://wiki.archlinux.org/title/Bus_Pirate (visited on 03/08/2022).

[9] Dangerous Prototypes. *Bus Pirate - DP*. June 2022. URL: http://dangerousprototypes.com/docs/Bus_Pirate (visited on 05/10/2022).

[10] Mohamed Amine Khelif, Jordane Lorandel, and Olivier Romain. "Non-invasive I2C Hardware Trojan Attack Vector". In: *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. ISSN: 2765-933X. Oct. 2021, pp. 1–6. DOI: 10.1109/DFT52944.2021.9568347.

[11] Kwonyoup Kim et al. *AES Wireless Keyboard : Template Attack for Eavesdropping*. Mar. 2018. URL: https://i.blackhat.com/briefings/asia/2018/asia-18-Kim-AES-Wireless-Keyboard-Template-Attack-for-Eavesdropping.pdf.

[12] Micheal Schoeffler. *DIY: Y-Adapter Jumper Wire*. Dec. 2017. URL: https://mschoeffler.com/2017/12/26/diy-y-adapter-jumper-wire/ (visited on 08/05/2022).

[13] Motorola, NXP Semiconductors, and Freescale Semiconductor. *SPI Block Guide*. July 2004. URL: https://www.nxp.com/files-static/microcontrollers/doc/ref_manual/S12SPIV4.pdf.

[14] Ibrahim Nadir et al. "An Auditing Framework for Vulnerability Analysis of IoT System". In: *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. June 2019, pp. 39–47. DOI: 10.1109/EuroSPW.2019.00011.

[15] NXP Semiconductors. *I2C-bus specification and user manual*. Oct. 2021. URL: https://www.nxp.com/docs/en/user-guide/UM10204.pdf.

[16] Nick Patavalis. *picocom*. Feb. 2018. URL: https://github.com/npat-efault/picocom (visited on 03/17/2022).

[17] R. X. Seger. *I2C interfacing on the Bus Pirate and Raspberry Pi to serial EEPROMs for a HAT*. Aug. 2016. URL: https://medium.com/@rxseger/i2c-interfacing-on-the-bus-pirate-and-raspberry-pi-to-serial-eeproms-for-a-hat-c776ea50f8f4 (visited on 08/13/2022).

[18] R. X. Seger. *SPI interfacing experiments: EEPROMs, Bus Pirate, ADC/OPT101 with Raspberry Pi*. Sept. 2016. URL: https://medium.com/@rxseger/spi-interfacing-experiments-eeproms-bus-pirate-adc-opt101-with-raspberry-pi-9c819511efea (visited on 08/13/2022).

Appendix A

Measure the temperature using I2C

The following Arduino code initializes the BMP280 in I2C mode. It then reads and prints the temperature every second.

bmp280I2C.ino

```
#include <Wire.h>
#include <Adafruit_BMP280.h>

#define BMP_ADDR 0x76

Adafruit_BMP280 bmp;

void setup() {
    Serial.begin(9600);
    while (!Serial)
        delay(100);
    Serial.println("BMP280 I2C test");
    unsigned status = bmp.begin(BMP_ADDR);
    if (!status) {
        Serial.println("Could not find a BMP280");
        while (1);
    }

    bmp.setSampling(
        Adafruit_BMP280::MODE_NORMAL,
        Adafruit_BMP280::SAMPLING_X2,
        Adafruit_BMP280::SAMPLING_X16,
        Adafruit_BMP280::FILTER_X16,
        Adafruit_BMP280::STANDBY_MS_500
    );
}

void loop() {
```

```
    Serial.print("Temperature = ");
30  Serial.print(bmp.readTemperature());
    Serial.println(" *C");
    Serial.println();
    delay(1000);
}
```

Appendix B

Convert dig_T registers to the correct data type

The following Python script converts the BMP280 register bytes to short values. As input it takes three strings, which contain the hexadecimal numbers. These numbers can be retrieved from the BMP280 with the Bus Pirate.

convert_registers.py

```
import re

HEX_PATTERN = "0x[0-9a-fA-F]{2}"

def find_bytes_in_string(string):
    """
    Takes a string and returns a list of all
    bytes contained in that string
    """
    return [int(i, 16) for i in re.findall(HEX_PATTERN, string)]

dig_T1 = find_bytes_in_string(input("Enter dig_T1 bytes: "))
dig_T2 = find_bytes_in_string(input("Enter dig_T2 bytes: "))
dig_T3 = find_bytes_in_string(input("Enter dig_T3 bytes: "))

dig_T1 = int.from_bytes(dig_T1, byteorder="little", signed=False)
dig_T2 = int.from_bytes(dig_T2, byteorder="little", signed=True)
dig_T3 = int.from_bytes(dig_T3, byteorder="little", signed=True)

print(f"dig_T1: {dig_T1}")
print(f"dig_T2: {dig_T2}")
print(f"dig_T3: {dig_T3}")
```

Appendix C

Compensate BMP280 temperature

The following Python script can decode the bytes that the Bus Pirate intercepts. The calculations are done in a similar way as described in the Bosch BMP280 datasheet [6].

decode_bmp280.py

```
import re

HEX_PATTERN = "0x[0-9a-fA-F]{2}"

def find_bytes_in_string(string):
    """
    Takes a string and returns a list of all
    bytes contained in that string
    """
    return [int(i, 16) for i in re.findall(HEX_PATTERN, string)]

def compensate_temperature(adc_T, dig_T1, dig_T2, dig_T3):
    """
    Sensor output needs to be compensated,
    as per section 3.11 of the BOSCH BMP280
    datasheet
    """
    adc_T >>= 4

    var1 = (((adc_T >> 3) - (dig_T1 << 1)) * dig_T2) >> 11
    var2 = (
        ((((adc_T >> 4) - dig_T1) * ((adc_T >> 4) - dig_T1)) >> 12)
        * dig_T3
    ) >> 14

    t_fine = var1 + var2
```

```
    T = (t_fine * 5 + 128) >> 8
    return T / 100

# Find these in registers 0x88, 0x8a, 0x8c
dig_T1 = 28095
dig_T2 = 27081
dig_T3 = -1000

pirate_output = input("Enter the intercepted bytes: ")
output_bytes = find_bytes_in_string(pirate_output)

raw_int = int.from_bytes(output_bytes, byteorder="big")

temperature = compensate_temperature(
    raw_int,
    dig_T1,
    dig_T2,
    dig_T3,
)

print(f"BMP: {temperature:.2f}°C")
```

Appendix D

Measure the temperature using SPI

The following Arduino code initializes the BMP280 in SPI mode. It then reads and prints the temperature every second.

bmp280SPI.ino

```
#include <Adafruit_BMP280.h>

#define BMP_SCL 13
#define BMP_SDA 12
#define BMP_CSB 11
#define BMP_SDO 10

Adafruit_BMP280 bmp(BMP_CSB, BMP_SDA, BMP_SDO, BMP_SCL);

void setup() {
    Serial.begin(9600);
    while(!Serial)
        delay(100);
    Serial.println("BMP280 test");
    unsigned status = bmp.begin();
    if (!status) {
        Serial.println("Could not find a BMP280");
        while (1);
    }

    bmp.setSampling(Adafruit_BMP280::MODE_NORMAL,
            Adafruit_BMP280::SAMPLING_X2,
            Adafruit_BMP280::SAMPLING_X16,
            Adafruit_BMP280::FILTER_X16,
            Adafruit_BMP280::STANDBY_MS_500);
}

void loop() {
```

```
    Serial.print("Temperature = ");
30  Serial.print(bmp.readTemperature());
    Serial.println(" *C");
    Serial.println();
    delay(1000);
}
```