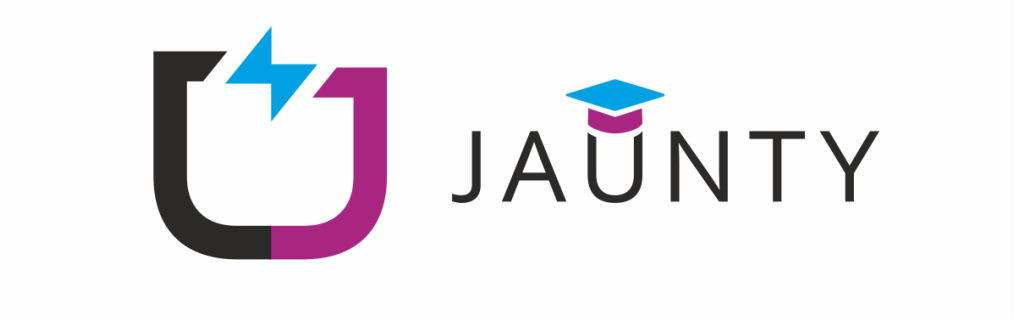
I2C communication with Raspberry Pi 4B



Author: Public Power Corporation S.A

Co-funded by the
Erasmus+ Programme
of the European Union

Copyright

Consisting of

Coordinator:	Technical University of Sofia	Bulgaria
Partners:	University of Western Macedonia	Greece
	International Hellenic University	Greece
	Public Power Corporation S.A.	Greece
	University of Cyprus	Cyprus
	K3Y Ltd	Bulgaria
	Software Company EOOD	Bulgaria

Co-funded by the Erasmus+ Programme of the European Union

Table of Contents

1. Abbreviations

SBC	Single Board Computer
DC	Direct Current
IoT	Internet of Things
RPI4B	Raspberry Pi 4B
USB	Universal Serial Bus
HDMI	High-Definition Multimedia Interface
GPIO	General Purpose Input/Output
SSH	Secure Shell
SDA	Serial Data Line
SCL	Serial Clock Line
CRC	Cyclic Redundancy Check

2. Scope

The scope of this laboratory module is to provide a comprehensive introduction to the Raspberry Pi 4B Single Board Computer (SBC), including the communication with external sensors using the I2C protocol. Moreover, the students will be engaged in the implementation of an electronic circuit, including the INA260 sensor for Direct Current (DC) power measurement and the SHT40 temperature and humidity sensor.

Finally, the students will be introduced to some basic concepts of the Python programming language, necessary to retrieve the measurements from the IoT devices, using the I2C protocol.

2.1. Specific outcomes

Upon completion of this lab, individuals will be able to:

- Understand and implement simple breadboard diagrams.
- Understand and operate the Raspberry Pi 4B.
- Understand the I2C communication protocol and the relevant information provided by user manuals of I2C sensors.
- Use the Python programming language to communicate with I2C sensors, retrieve and transform data in human-readable format.

2.2. General description

In this laboratory, a DC circuit will be implemented, that supplies a DC load (a resistor or a DC motor). Purpose of one of the laboratory exercises is to utilise the INA260 sensor to measure the DC consumption of the circuit load.

The INA260 sensor is illustrated in Figure 1. INA260 is a powerful current sensor, being able to measure up to 36V DC and up to 15A, on either high or low side[1]. High-side measurement is preferred, due to its improved diagnostic capabilities, as well as it maintains the integrity of the ground path by avoiding the addition of undesirable extraneous resistance. Moreover, INA260 integrates a shunt resistor of 2 mΩ, allowing resolution of 1.5 mA. The INA260 datasheet will be used as the main reference document, while conducting the steps of this lab [1].

[1] High-side current sensing is a method of measuring the current, according to which the current sensor is placed between the hot wire (positive supply) and the load that needs to be measured. On the other hand, low-side current sensing requires the current sensor to be placed between the load and the ground (negative supply).

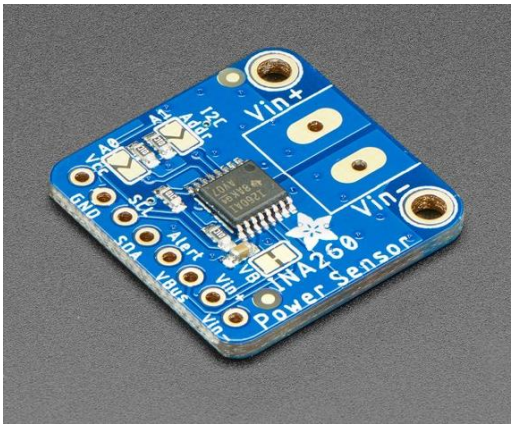


Figure 1: The INA260 sensor

The SHT40 sensor is depicted in Figure 2. SHT40 can measure both the relative humidity[2] and the temperature of the environment, with ± 1.8% typical relative humidity accuracy from 25 to 75% and ± 0.2°C from 0 to 75°C. Similarly, the SHT40 datasheet will be the main reference document for interacting with the sensor [2].

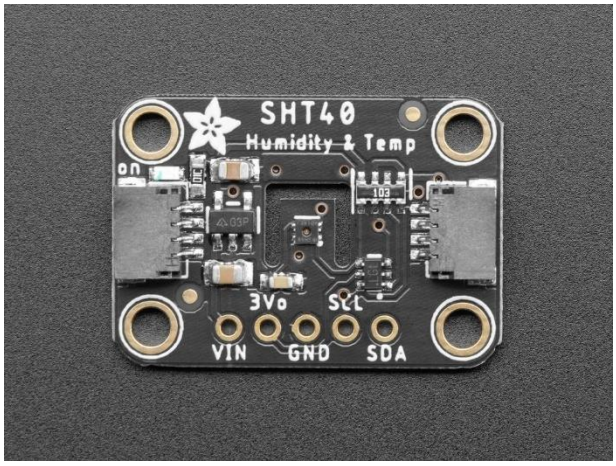


Figure 2: The SHT40 sensor

Finally, the Raspberry Pi 4B (RPI4B) will be used throughout this laboratory, to provide power supply to the sensors and retrieve data from those. RPI4B is a complete computer built on a single board, that integrates popular I/O interfaces (e.g., Universal Serial Bus (USB), High-Definition Multimedia Interface (HDMI), etc) as well as multiple network interfaces, including Bluetooth, Wi-Fi, and Ethernet [3]. Moreover, RPI4B has multiple General-Purpose Input/Output (GPIO) pins, that are used for various purposes, whereas they can also be programmed by the user. Some of those pins on the GPIO of RPI4B are used for communication with external sensors via the I2C serial communication protocol [4]. I2C is ideal for peripheral devices with low-rate data transmission requirements. The official I2C specification

[2] Relative humidity is the ratio of how much water vapour is in the air and how much water vapour the air could potentially contain at a given temperature (e.g., colder air can hold less vapour). Therefore, depending on the air temperature, the relative humidity can change, even when the absolute humidity remains constant.

will occasionally be mentioned as a reference document, to understand how the protocol works [5]. The GPIO pinout diagram of RPI4B, showing the usage of each pin, is illustrated in Figure 3.

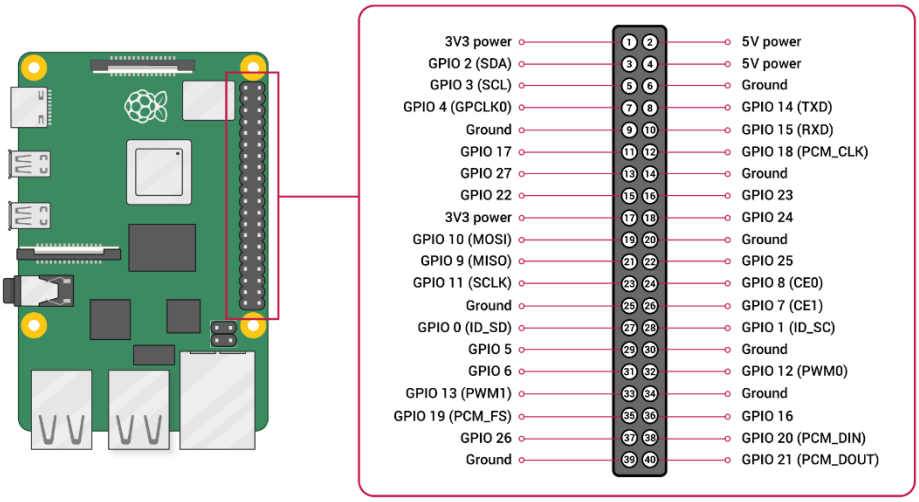


Figure 3: The RPI4B pinout diagram[3]

2.3. Lab configuration

This section should be filled by the lab instructor, depending on whether the student will utilise a RPI4B remotely available at the physical premises of the laboratory.

Placeholder / Property	Value
SSH Username	
SSH Password	
Raspberry Pi 4B IP Address	

[3] **https://www.the-diy-life.com/gpio-pinout-diagram-2/**

Exercise 1: Implement the breadboard diagram

Purpose of this exercise is to implement the breadboard diagram by interconnecting the correct GPIO pins of RPI4B with the INA260 sensor and the SHT40 sensor.

Before implementing any change to the topology, you must first deactivate any DC power supply. Provide power supply to RPI4B and activate the bus supply, ONLY when the implementation has been completed.

Figure 4 depicts the wiring diagram of the prototype topology for this lab. To implement the topology, the following equipment is needed:

1. x1 Raspberry Pi 4B (RPI4B). The platform should have at least:
    - An external SD card of at least 32GB space.
    - At least 4GB RAM.
    - A keyboard and a mouse.
    - A mini-HDMI cable to connect RPI4B to a screen.
    - An Ethernet cable or available Wi-Fi connection.
2. x1 breadboard.
3. x1 Adafruit INA260 Current and Voltage meter.
4. x1 Adafruit SHT40 temperature and humidity sensor.
5. x15 wires (depending on the implementation).
6. x1 DC power supply (“Best BST-305D” is used in the lab).
7. x1 resistor (around 100Ω) or a 5V DC motor.
8. x1 mini-HDMI to VGA or HDMI, to see the display output.
9. Mouse, keyboard and screen for initial configuration of RPI4B.

Purpose of the prototype topology is to utilise the INA260 sensor to measure the power consumption of a load. By retrieving the current and voltage measurements, the power and resistance values can be calculated. Moreover, at the same topology, the SHT40 sensor will be interconnected to the same I2C bus to retrieve the temperature and humidity of the environment.

This prototype topology has already been implemented and is remotely available in the physical laboratory premises. It is recommended to follow the steps of this exercise to replicate the prototype with your own equipment. In case that this is not possible, then you could study the implementation steps and proceed to Exercise 2 by accessing the RPI4B already provided.

The topology that will be implemented in this exercise is depicted in Figure 4. The building of this topology will break down in specific steps, during this exercise.

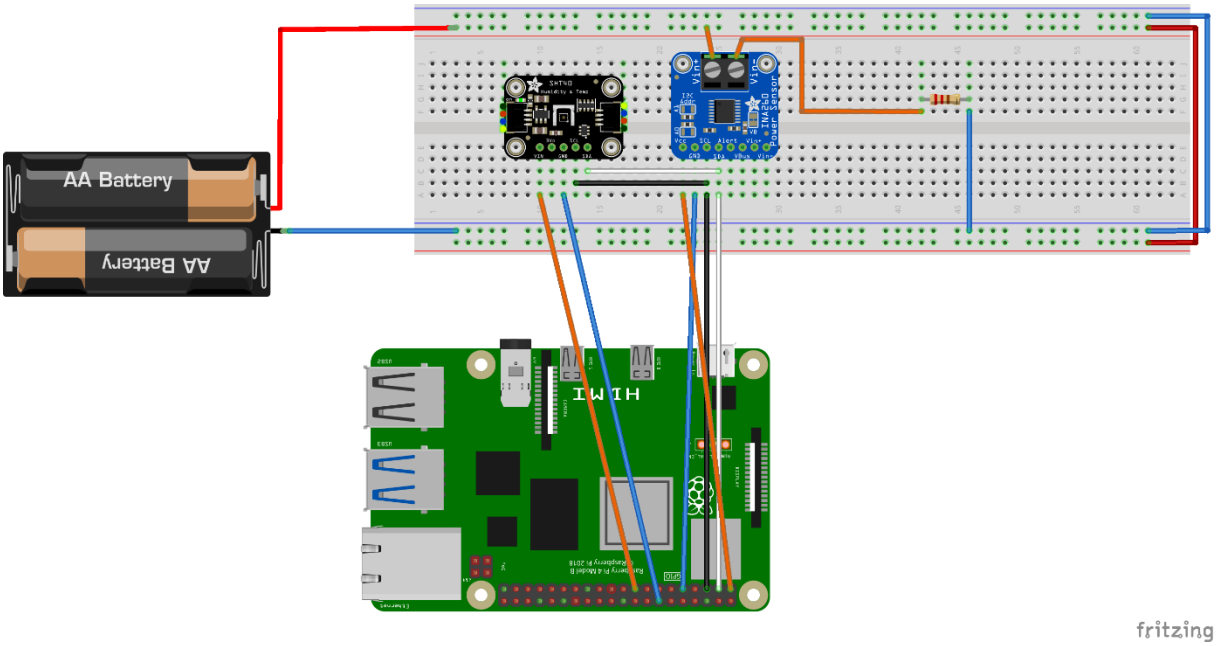


Figure 4: The prototype Raspberry Pi 4B topology

Step 1: Soldering the sensor contacts and place the sensors in the breadboard

To ensure reliable electrical contact, the header strip should be soldered to the corresponding sensor contacts, using a soldering iron for electronics. For INA60, the terminal block should also be soldered to the Vin+ and Vin- contacts of the INA260 sensor. Figure 5 depicts the header strip and the terminal block of the INA260 sensor.

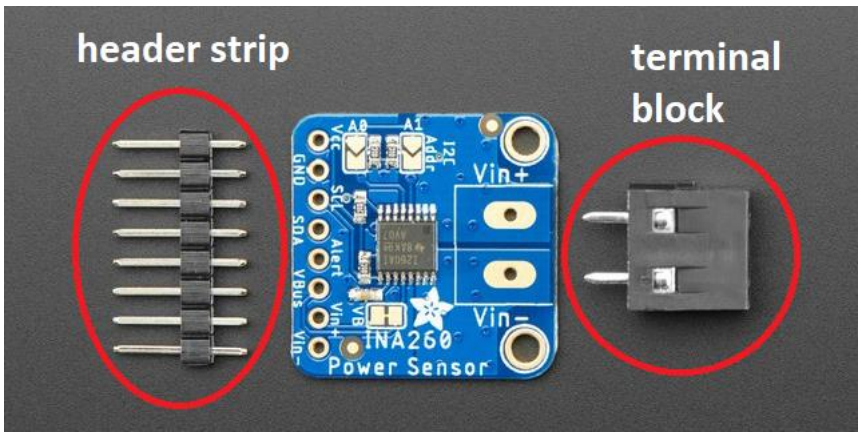


Figure 5: The header strip and the terminal block of INA260

By finishing the soldering process, the soldered sensor should look as depicted in Figure 6.

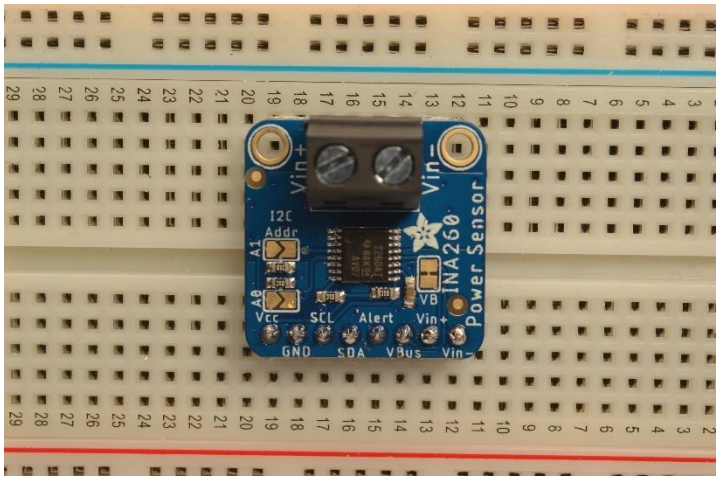


Figure 6: Soldered INA260 sensor placed on the breadboard

For more detailed instructions on soldering and assembling, you are advised to visit the official instructions provided by Adafruit [6].

Similarly, a header strip should be soldered to the contacts of the SHT40 sensor.

Finally, place the sensors in the breadboard, as depicted in Figure 6 and Figure 4. Ensure to leave enough space on the front of the sensor contacts, to place additional wires.

At this step, the topology should look like the diagram in Figure 7.

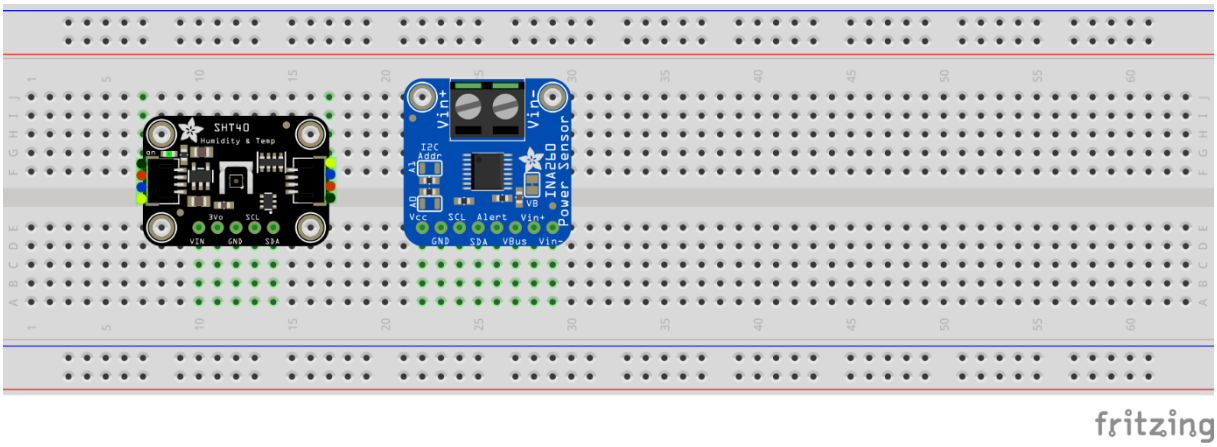


Figure 7: The topology after finishing Step 1

Step 2: Connecting the SDA and SCL contacts with Raspberry Pi 4B

Ensure that RPI4B is turned off before proceeding!

In this step, the Serial Data Line (SDA) and Serial Clock Line (SCL) pins of RPI4B will be connected to the INA260 and SHT40 sensors, aiming to form a shared I2C bus utilised for data exchange. According to the I2C terminology, the RPI4B is the master node, which generates the clock and initiates the communication with slaves. INA260 and SHT40 are slave nodes, since they receive the clock generated by the master and respond only when addressed.

According to the RPI4B pinout scheme, Pin 3 corresponds to data transfer (SDA) and Pin 5 corresponds to clock (SCL). Therefore, an interconnection should be implemented, like the one depicted in Figure 8. Note that the SDA channel is distinguished by white wires, while the SCL channel is indicated by black wires.

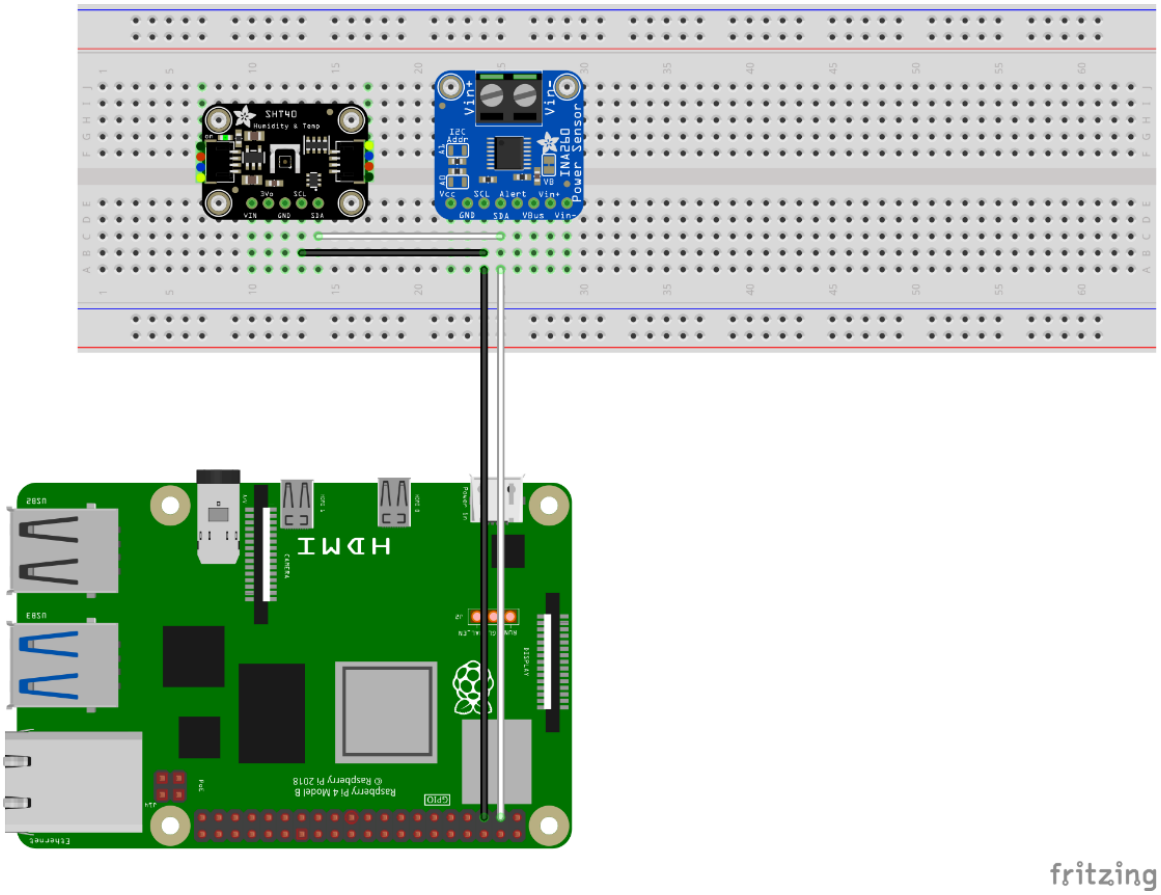


Figure 8: SDA and SCL connections with RPI4B

Step 3: Supplying the sensors via the Raspberry Pi 4B platform

The RPI4B platform integrates multiple 5V and 3.3V DC supplies, which can be used to power on various peripheral devices. According to the RPI4B pinout scheme, the following connections can be applied:

- Pin 1 (3V3) of RPI4 to VCC contact of INA260 sensor.
- Pin 9 (GND) of RPI4B to GND contact of INA260 sensor.

- Pin 14 (GND) of RPI4B to GND contact of SHT40 sensor.
- Pin 17 (3V3) of RPI4B to Vin contact of SHT40 sensor.

The connections described above are summarised in Figure 9.

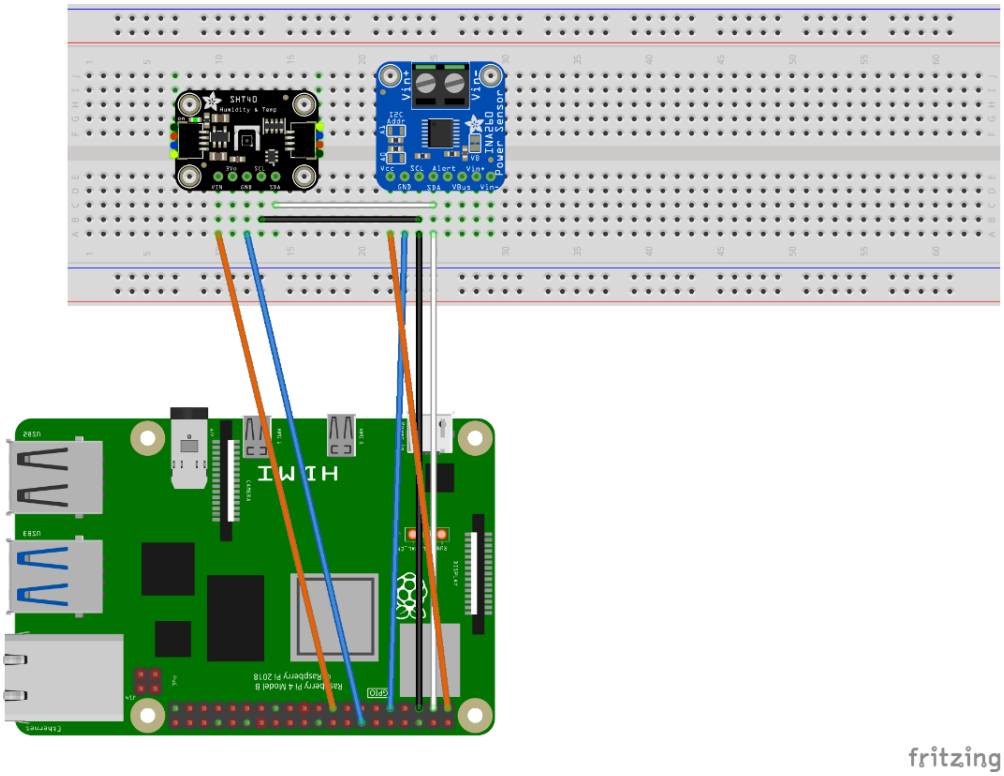


Figure 9: Providing power supply to the I2C sensors

Step 4: Applying high-side current sensing to the load

As a final implementation step, the DC circuit that includes the load needs to be interconnected with the INA260 sensor, to measure the power consumption. As illustrated in Figure 4, a DC source is connected to the supply buses of the breadboard. The buses are connected at the right end of the breadboard, to utilise both sides of the supply buses.

To measure the power consumption of the DC load, the INA260 sensor must be connected in series with the DC load. In particular, the INA260 sensor must be between the positive supply and the load. In this way, the high-side current sensing is applied.

The topology prototype that is already implemented on the physical laboratory premises utilises a 92Ω resistor as the circuit load. The supplied voltage is 4.3V. If this setting is not available for your custom circuit, you can use any other resistor (or DC motor) available. In that case, though, you need to adjust the bus voltage, for the current not to exceed 15A. In particular:

- If you are using a DC battery supply, then the supply voltage cannot be altered. Thus, a load of an acceptable resistance must be chosen, so that the current remains lower than 15A.
- If instead of a battery supply, you are using a laboratory DC supply, then instead of finding the right load, you can adjust the voltage supply, so that the current remains under 15A.

After finalising the interconnections and the implemented topology is like Figure 4, turn on the power supply to the RPI4B platform and the external DC supply.

Exercise 2: Initialize the Python environment in Raspberry Pi 4B

Purpose of this exercise is to get familiarized with the Raspberry Pi 4B environment and prepare the Python programming environment needed to enable interaction with the I2C sensors.

Step 1: Prepare the Raspberry Pi 4B platform

This step is needed only if you are using your own RPI4B. If instead you are using the RPI4B already deployed in the physical laboratory premises, then skip Step 1 and go to Step 2.

First, follow the official guidelines[4] to prepare the platform. By following the guidelines, you should have:

1. Installed Raspbian OS to an SD card.
2. Completed the initial setup for setting up password, region, Wi-Fi (if needed) and installing basic updates.

After the initial installation, SSH must be enabled to allow remote connections to the system shell. To this purpose, supposing you are using mouse/keyboard and the HDMI display for the initial configuration, open the sidebar menu of the RPI4B desktop environment (top left), and select the "Raspberry Pi Configuration" entry, as indicated in Figure 10.

[4] **https://projects.raspberrypi.org/en/projects/raspberry-pi-getting-started/4**

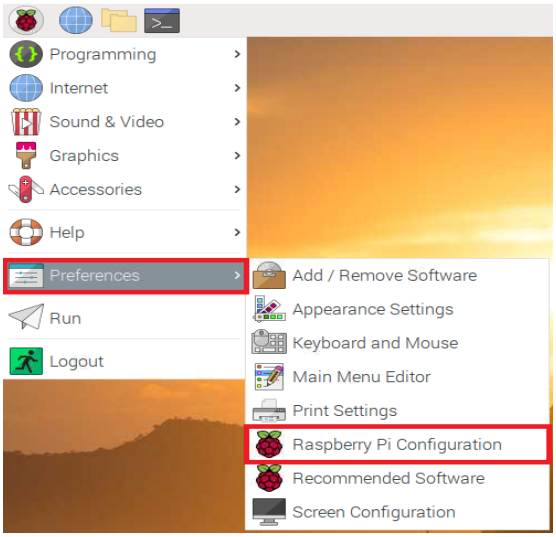


Figure 10: Open the Raspberry Pi Configuration menu

Then, according to Figure 11,navigate to the "Interfaces" tab and enable SSH. Click "OK" to save changes.

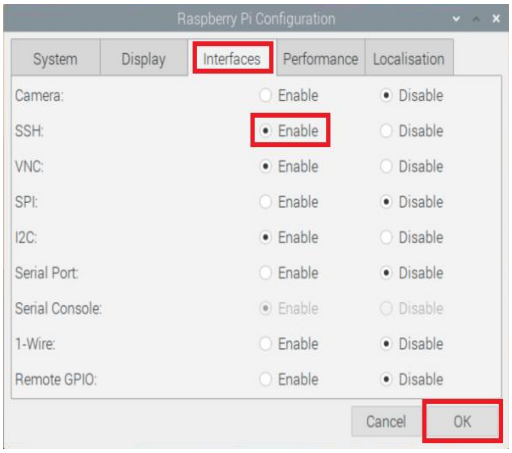


Figure 11: Enable SSH in the Raspberry Pi Configuration menu

Step 2: Connect via SSH to Raspberry Pi 4B and install OS packages

In case you are using the shared lab infrastructure, retrieve the IP address and the SSH username/password from section 2.3.

The most common and secure way to establish connection with the Raspberry Pi 4B platform is by using the SSH (Secure Shell) protocol. SSH allows to establish encrypted remote connections with operating systems and use their shell remotely.

In case an SSH client is not available to your system, you can use PuTTY, an open source SSH client for Windows. You can apply the configuration depicted in Figure 12 to connect via PuTTY.

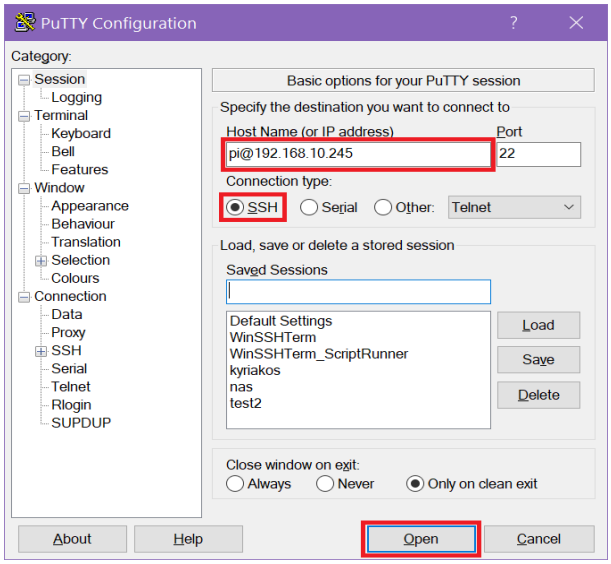


Figure 12: PuTTY configuration

After clicking on "Open", you will be prompted to provide the SSH password. Use the password provided or the one configured during step 1, and press Enter. Note that during typing, the password is not visible.

Alternatively, if SSH client is available to your system and the Path environmental variable, then you can access the remote shell from the command line. As depicted in Figure 4, you can type **`ssh pi@192.168.10.245`** (replace depending on the RPI4B IP address), to connect via SSH. When prompted, provide the SSH password.

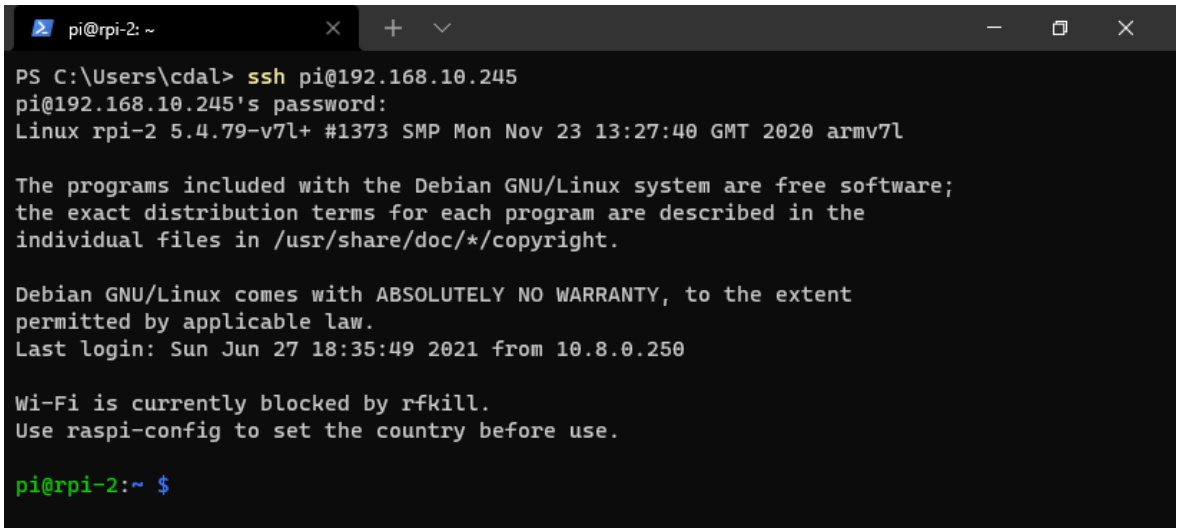


Figure 13: Connecting to Raspberry Pi 4B from terminal

Finally, the RPI4B platform needs to have some basic OS tools that allows interaction with the I2C bus and testing. For this purpose, the `i2cdetect` tool needs to be installed, by executing the commands provided in Table 1.

Table 1: Instructions for installing I2C tools

Step	Description	Action
#1	Update the local package repository.	`sudo apt update`
#2	Install the i2c-tools package	`sudo apt install i2c-tools`
#3	Add the current user to the I2C access group	`sudo adduser pi i2c`
#4	Exit the shell session and reconnect via SSH.	-
#5	Test the interaction with the I2C bus	`i2cdetect -y 1`

Figure 14 depicts the results of the `i2cdetect` command. The -y argument disables interactive mode and 1 specifies the I2C bus. The number of the I2C bus depends on the manufacturer. For RPI4B, I2C bus number is always 1. The command probes the entire address range according to the I2C standard, to discover I2C devices. The output indicates that one device responds at the 0x40 I2C address. This command is useful to verify that the I2C connections are implemented successfully and to verify the I2C addresses to be used for data exchange.

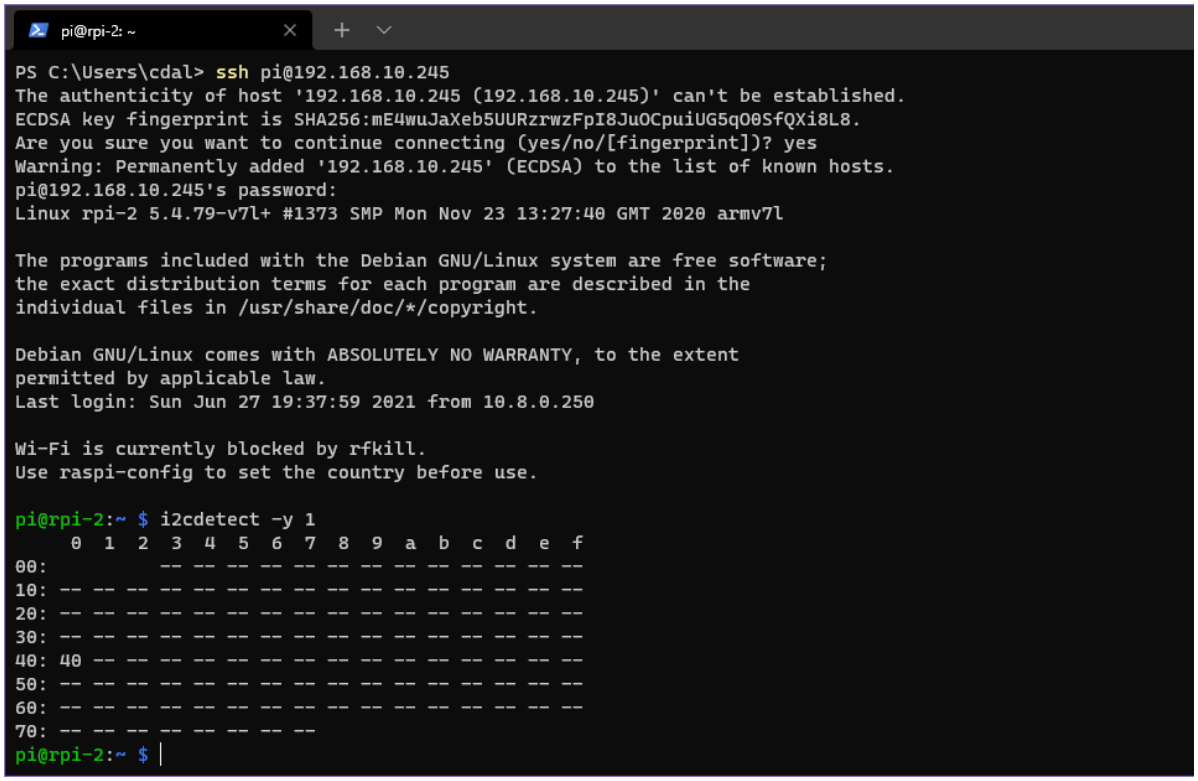


Figure 14: Output of the i2cdetect command

Step 3: Introduction to Python - Create the Python virtual environment

Python is a high-level general-purpose programming language. Python is a dynamic programming language, meaning that the source code is interpreted and executed at runtime, while the source code of other languages, like C++ and Java, is interpreted during compilation. In other words, the Python

executable reads the source code and directly executes the instructions defined in the source code, without the need of compilation and other intermediary steps.

Another characteristic of Python is the availability of a wide range of libraries, that allows to perform advanced tasks in the fields of industrial automation, data analytics, machine learning, web development, system/network administration, and scientific computing, among others. In this lab we intent to use the appropriate Python libraries to exchange I2C messages.

Python provides a flexible way of working with libraries, inside virtual environments. Virtual environment (venv) is a self-contained directory tree that includes a complete Python installation with its own packages, which are independent from the operating system and any other Python installation. For example, the developer could create a venv that has version 1.0 of package A, while in a second venv the developer has the version 2.0 of the same package A. The developer can test their Python script in both environments and draw conclusions about compatibility. The two environments are not interacting with each other, and any modification/addition in one venv does not propagate or affect the other venv.

In this step, you are going to prepare a Python venv that contains all the necessary Python packages to retrieve information from the I2C bus. Please follow the instructions provided in Table 2.

When no Python environment is activated, you should use the `python3` command to refer to the system Python 3.X installation. When a Python environment is activated (action #6 of Table 2), the PATH environmental variable is modified accordingly for the current shell session. After action #6, you should use the `python` command (instead of `python3`) to refer to the venv Python installation.

Table 2: Instructions for preparing the Python venv

Action	Description	Action
#1	Update the local package repository.	`sudo apt update`
#2	Install the python-venv package, which allows to create virtual environments.	`sudo apt install python3-venv`
#3	Create a personal folder, where your code will be stored. Please rename the folder accordingly.	`mkdir ~/john_smith`
#4	Move to your directory	`cd john_smith`
#5	Create a virtual environment, named venv-i2c.	`python3 -m venv venv-i2c`
#6	Activate the environment.	`source ~/john_smith/venv-i2c/bin/activate`
#7	Install the smbus2 package	`pip install smbus2`

For this lab, a single Python file can be created for writing the necessary source code that interacts with the I2C bus. The following source code structure should be kept:

```
1  import package1
2  import package2
3
4  def functionA(var1):
5      print("This is function A")
6      return var1*5
7
8  def function():
9      Print("This is function B")
10     return 5
11
12 if __name__ == "__main__":
13     a = functionA(5)
14     functionB()
15     print("success")
```

Any external package should be imported at the beginning of the source code (lines 1-2). Then, the definition of functions should be followed (lines 4-10). Optionally, global variables could be defined after importing the packages (line 3), so that they will be available both to the main function and the custom functions. Finally, the main function should be placed after function definition (line 12).

Exercise 3: INA260 – Communication with I2C

Purpose of this exercise is to retrieve the electricity-related measurements of the INA260 sensor using I2C and the general-purpose smbus2 library. Final output of this exercise will be a Python script that retrieves and displays the measurements of the INA260 sensor in a configurable interval.

While the manufacturer (Adafruit) provides a ready-to-use library to communicate via I2C with the sensor, without the need of knowing details about the sensor characteristics, purpose of this lab is to develop the necessary skills in order to communicate with any I2C device and adapt the usage of the smbus2 library to the sensor characteristics.

The INA260 datasheet is used as the main reference document in this exercise [1].

Step 1: Identify the I2C addresses

Before writing the source code, you need to record all the necessary I2C addresses and register pointers that can be used to specify the information that needs to be retrieved.

First, the I2C address of the INA260 sensor is needed. This address is used to uniquely identify the sensor and is essential, since I2C is a shared medium, where each message is received by every I2C sensor wired in the bus. For INA260, the default I2C address is **0x40** (1000000_2), as indicated in section 8.5.3.1 – Table 2 of the datasheet. As described, it is possible to change the default address if the A1 and A0 pins are connected with other pins of the sensor. However, for this lab, we will leave the I2C address to its default value.

Next, you need to identify the measurements that must be retrieved. Table 4 of section 8.6 Register Maps summarises the register addresses, where settings, measurement results, minimum/maximum limits, and status information is stored. For this lab, the following registers are needed:

- **Current register**: Contains the value of the current flowing through the shunt resistor. Register address is 0x01.
- **Voltage register**: Contains the voltage measurement. Register address is 0x02.
- **Power register:** Contains the power value, calculated by the sensor. Register address is 0x03.
- **Manufacturer register:** Contains the identifier of the sensor manufacturer. Register address is 0xFE.

Step 2: Ensure correct encoding of static values

The INA260 sensor provides some registers that hold static values (e.g., Manufacturer ID), the values of those are also provided in the datasheet. For example, the Manufacturer ID is always 0x5549 and the Die ID is always 0x2270. Therefore, in this step you will try to retrieve and successfully decode this information to ensure that the source code is valid.

To begin with, create a new file named `ina260.py` in your personal directory, in the RPI4B. Then, insert the following source code:

```
1  import smbus2 import SMBus
2
3  if __name__ == "__main__":
4      i2caddress = 0x40
5      MAN_ID_POINTER = 0xFE
6
7      i2cbus = SMBus(1)
8      manid = i2cbus.read_i2c_block_data(i2caddress, MAN_ID_POINTER, 2)
9      print("Raw returned value: ")
10     print(manid)
11
12     manid = bytes(manid)
13     manid_hex = manid.hex()
14     manid_dec = int.from_bytes(manid, "big")
15     print("After conversion (hex): " + str(manid_hex))
16     print("After conversion (dec): " + str(manid_dec))
```

First, the SMBus module of the smbus2 library is imported (line 1), which is necessary to interact with the I2C bus.

Then, the I2C address and the pointer of the information we need to retrieve are defined in lines 4-5.

In line 7, the I2C bus is opened and stored in a variable that is later used to interact with the bus. Please note that the bus ID is provided as a parameter to the SMBus constructor. The bus ID depends on the RPI4B manufacturer, and for the specific model, the value is always 1.

The value stored in the Manufacturer ID register is retrieved by using the `read_i2c_block_data()` function. The function needs three parameters:

- First parameter is the i2caddress, 0x40 in our case.
- Second parameter is the register address, 0xFE in our case.
- Third parameter is the number of bytes that should be retrieved from the register. The datasheet provides 01010100 01001001 as an expected content of that register, therefore, we conclude that we need to retrieve 2 bytes.

The returned value of `read_i2c_block_data()` is a Python list of two values, corresponding to 2 bytes forming the manufacturer ID. In particular, the returned value should be: [84, 73]. However, these values are in decimal form and need further processing.

To calculate the human-readable value, we should: a) convert the two bytes in binary, b) concatenate the two bytes, c) convert the concatenated value in hexadecimal. To this end, line 12 converts and concatenates the Python list to a single byte series. Then, line 13 converts the bytes to hexadecimal. Line 14 converts the same byte series to its decimal form. However, in the case of decimal conversion, we need to specify the endianness of the input bytes. According to the INA260 datasheet, "register bytes are sent most-significant byte first, followed by the least significant byte" (pg. 19). Therefore, we need to specify the `big` parameter.

Finally, the printed value should be 0x5449 and 21577. The intended output is depicted in Figure 15.

```
>>> %Run i2c-generic.py

  Raw returned value:
  [84, 73]
  After conversion (hex): 5449
  After conversion (dec): 21577

>>>
```

Figure 15: Output of I2C script for INA260

Step 3: Retrieve electricity-related measurements

As with step 2, the current and voltage measurements can also be retrieved by the INA260 sensor, by sending I2C read commands to the corresponding register addresses, according to the datasheet.

The source code provided bellow can be used to retrieve the values of the voltage and current registers from INA260.

```
1  import smbus2 import SMBus
2
3  if __name__ == "__main__":
4      i2caddress = 0x40
5      CURRENT_REG = 0x01
6      VOLTAGE_REG = 0x02
```

```
7       i2cbus = SMBus(1)
8
9       current = i2cbus.read_i2c_block_data(i2caddress, CURRENT_REG, 2)
10      current = bytes(current)
11      current = int.from_bytes(current, "big")
12      current = (1.25*current)/1000
13
14      voltage = i2cbus.read_i2c_block_data(i2caddress, VOLTAGE_REG, 2)
15      voltage = bytes(voltage)
16      voltage = int.from_bytes(voltage, "big")
17      voltage = (1.25*voltage)/1000
```

In more detail, the relevant address values are initialized in lines 4-6. The I2C bus object is created in line 7, while the current and voltage measurements are retrieved in lines 9-12 and 14-17 respectively. To retrieve an electricity measurement, the `read_i2c_block_data()` function is used, where the I2C address, the register of the measurement, and the length of retrieved data (2 bytes) are specified. Then, the Python list is concatenated to a byte series, which is converted to decimal.

Last step of the conversion process is to multiply the current or voltage with the scale value, introduced by the integrated shunt resistor. According to the datasheet (section 8.5.1), this value is 1.25 mV/bit and 1.25 mA/bit respectively. Therefore, the contents of the register should be multiplied by 1.25 and divided by 1000 to retrieve the Voltage/Ampere value in its SI (International System of Units) form.

Step 4: Calculate power and resistance

Since voltage and current have been retrieved, useful electrical attributes can be calculated, like power consumed by the load as well as its resistance. According to Ohm's law, the resistance is calculated as follows:

$$R = \frac{V}{I}$$

To validate the source code of step 3, try to calculate the resistance, based on the retrieved voltage and current measurements. The calculated value should be close to the nominal resistance used in the circuit.

Moreover, the measured power can be calculated as follows:

$$P = V * I$$

However, the INA260 sensor is also able to calculate power by multiplying the contents of voltage and current registers.

Try to calculate the consumed power, according to the retrieved voltage and current measurements, and compare it with the power value that is automatically calculated by the INA260 sensor. Read the INA260 datasheet for the information needed to retrieve the

value of the power register. Not forget to apply the appropriate conversions, according to the datasheet instructions.

Step 5: Converting measurements in JSON format

Final step of this exercise is to insert the measurements into the appropriate structure, to be readable by humans or other applications. JSON (JavaScript Object Notation) is a lightweight data-interchange format, that is used by software components to exchange information, in a common, intuitive format that can be easily parsed and generated.

There are two basic JSON structures, JSON arrays and JSON objects. A JSON object is a collection of key-value pairs that are denoted in the following form:

```
{
    "key1": 5,
    "key2": "value2"
}
```

In Python, JSON manipulation is natively supported by the Python dictionaries. A Python dictionary definition follows bellow:

```
json_object = {
    "key1": 5,
    "key2": "value2"
}
```

The value of a key can easily be accessed as follows: `json_object["key1"]`.

The source code that follows bellow puts together all measurements in a JSON object, that is renewed each 5 seconds:

```
1  import smbus2 import SMBus
2
3  if __name__ == "__main__":
4      i2caddress = 0x40
5      CURRENT_REG = 0x01
6      VOLTAGE_REG = 0x02
7      POWER_REG = 0x03
8
9      i2cbus = SMBus(1)
10     while True:
11         current = i2cbus.read_i2c_block_data(i2caddress, VOLTAGE_REG, 2)
12         current = bytes(current)
13         current = int.from_bytes(current, "big")
14         current = (1.25*current)/1000
15
16         # Similar operations for voltage and power
```

```
17
18          result = {
19              "timestamp": str(datetime.now().isoformat())
20              "current": current,
21              "voltage": voltage,
22              "power": power
23          }
24          print(result)
25          time.sleep(5)
26
```

After the registers and I2C bus initialization (lines 4-9), a while structure is defined that runs indefinitely. Each time the while structure runs, the current voltage, current and power measurement values are retrieved by the I2C bus, while the appropriate conversions are applied as well. Then, the JSON object containing all measurements is defined.

In addition to the measurements, the current timestamp is also included in the JSON object (line 19). This is necessary, to build timeseries and being able to study how the measurements are changing in time. There are multiple ways to represent timestamps (combination of both date and time), however, the ISO 8601 format is recommended to ensure maximum interoperability and support with the IoT ecosystem. To this purpose, the `now()` method of datetime is used to retrieve the current time (`utcnow()` can be used instead to retrieve the timestamp in UTC time zone). Then, the result is converted to ISO format by using the `isoformat()` method. Timestamp example in ISO format:

```
2021-07-05T15:22:43.45321
```

Note the T character that separates the date and time parts.

Exercise 4: SHT40 – Communication with I2C

Purpose of this exercise is to retrieve the temperature and relative humidity measurements of the SHT40 sensor using I2C and the general-purpose smbus2 library. Final output of this exercise will be a Python script that retrieves and displays the measurements of the SHT40 sensor in a configurable interval.

The SHT40 datasheet is used as the main reference document in this exercise (**https://www.sensirion.com/fileadmin/user_upload/customers/sensirion/Dokumente/2_Humidity_Sensors/Datasheets/Sensirion_Humidity_Sensors_SHT4x_Datasheet.pdf**).

Step 1: Identify the I2C addresses

First, the I2C address of the SHT40 sensor needs to be identified, in order to communicate with the device on the shared I2C bus. According to the SHT40 datasheet, the I2C address is **0x44** (pg. 1).

Regarding the access to measurements, it should be noted that the SHT40 holds only a single register, where both temperature and humidity are stored in a single byte series. According to the I2C standard [7], even if it is common to have registers in I2C slaves, some devices contain only one register, which can be read or written directly to by sending the register data immediately after the slave address, instead of addressing a register.

Moreover, to read the desired value from the single register, a I2C write command should be sent to the device first, to indicate the measurement method that the sensor should apply. Table 7 of the SHT40 datasheet provides all supported commands. In our case, we will request from the sensor to measure Temperature and Relative humidity with high precision. The byte sequence that should be written is **0xFD**.

Step 2: Retrieve the temperature and humidity measurements

To retrieve the temperature and humidity measurements, it is needed to understand how the measurements are transmitted from the SHT40 sensor. The pseudocode for retrieving the measurements is provided bellow, according to the datasheet:

```
1  i2c_write(i2c_addr=0x44, tx_bytes=[0xFD])
2  wait_seconds(0.01)
3  rx_bytes = i2c_read(i2c_addr=0x44, number_of_bytes=6)
```

According to the pseudocode, a write command should be sent first directly to the I2C device (without specifying any I2C register), which indicates the measurement method. Then, after 0.01 seconds, an I2C command should be sent to read the first 6 bytes of the single SHT40 register. Again, the I2C register is not defined in the read command since it is not necessary.

After retrieving the byte sequence, this it should be appropriately interpreted to extract both temperature and relative humidity values. According to section 4.2 of the SHT40 datasheet [2], the first transmitted value is the temperature signal, consisting of two 8-bit sequences, followed by an 8-bit Cyclic Redundancy Check (CRC) sequence. Then, a two 8-bit sequence follows for the humidity signal, again followed by its 8-bit CRC sequence. Figure 16 illustrates the described rationale, if the 0x89 command is sent to retrieve the SHT40 serial number. Please note that in the case of the 0xFD command, a second 16-bit read data sequence will follow the last CRC, to also transmit the relative humidity measurement.

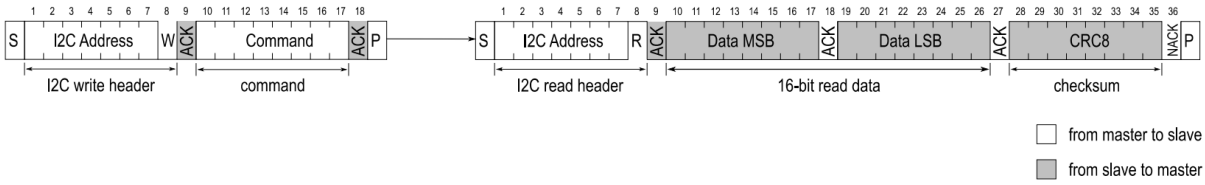


Figure 16: I2C communication sequence for SHT40 [2]

Finally, the signal output should be converted to a human-readable format. The formulae bellow can be applied to retrieve the final form of each measurement: (RH denotes relative humidity, S_{RH} and S_T are the raw measurements retrieved by the sensor.

$$RH = \left(-6 + 125 * \frac{S_{RH}}{2^{16} - 1}\right)\%$$

$$T = \left(-45 + 175 * \frac{S_T}{2^{16} - 1}\right)°C$$

Based on the above information, the source code provided bellow can be used to retrieve the measurements from the SHT40 sensor:

```
1  from smbus2 import SMBus, i2c_msg
2  import time
3
4
5  if __name__ == "__main__":
6      i2caddress = 0x44
7      i2cbus = SMBus(1)
8      dividend = (2**16-1)
9
10     i2cbus.write_byte(i2caddress, 0xFD)
11     time.sleep(0.01)
12     command = i2c_msg.read(i2caddress, 6)
13     i2cbus.i2c_rdwr(command)
14     rx_bytes = bytes(command)
15
16     temperature = rx_bytes[0:2]
17     temperature_crc = rx_bytes[3]
18     humidity = rx_bytes[3:5]
19     humidity_crc = rx_bytes[5]
20
21     temperature = int.from_bytes(temperature, "big")
22     temperature = -45 + 175*(temperature/dividend)
23
24     humidity = int.from_bytes(humidity, "big")
25     humidity = -6 + 125*(humidity/dividend)
```

In line 1, the necessary smbus2 libraries are imported. In addition to the `SMBus` structure, the `i2c_msg` is imported. This is required to be able to read I2C data without specifying register address.

In lines 6-8, the I2C address and the I2C bus are initialised, along with the dividend used during the final conversion of the retrieved measurements.

Line 10 sends the 0xFD sequence to the SHT40 sensor, addressed at 0x44. This instructs the I2C slave device to calculate the relative humidity and temperate with high accuracy. Then, the script hangs for 0.01 seconds, in order to leave enough time for the SHT40 sensor to make the calculations and store the results to the single register. After the timeout is reached, the read() method of the i2c_msg struct is called in line 12, in order to prepare a Python object that describes the desired read command (retrieve the first 6 byte of the SHT40 device). Then, the read instruction object is provided to the

`i2c_rdwr()` method of the I2C bus. This method can send one or multiple read and write commands sequentially, without a stop bit. In our case, we need to send only the read command previously prepared. The returned value will be stored in the input argument of `i2c_rdwr()`. Finally, the returned value is converted to a byte sequence, to proceed to byte manipulation.

Lines 16-19 parse the byte sequence by extracting the measurements, according to the datasheet instructions. The CRC sequences are also retrieved, but not further used.

Finally, lines 21-22 and 24-25 convert the byte sequences to decimal form and convert them to the final format, according to the formulae.

Step 3: Converting measurements in JSON format

Try to expand the script of step 2, in order to retrieve the temperature and humidity measurements repeatedly, each 5 seconds. At the end of the loop, put the measurements in JSON format, along with the current timestamp, and print them.

3. Exam Questions

1. Which pins of Raspberry Pi 4B would you use for connecting an I2C sensor? What is the purpose of each pin?
2. What are all the possible ways to access a Raspberry Pi 4B device?
3. Supposing you have access to a Raspberry Pi 4B device and an I2C sensor is connected, how would you check that the Raspberry Pi 4B device can interact with the sensor using the I2C bus?
4. What is the information you need to interact with an I2C sensor?
5. How would you create and use a Python environment?
6. By looking into web resources and/or the respective datasheets, specify how would you access the temperature and humidity measurements of an Si7021 sensor via the I2C bus.

4. References

[1] Texas Instruments, “INA260 - Precision Digital Current and Power Monitor With Low-Drift, Precision Integrated Shunt,” 2016. [Online]. Available: https://www.ti.com/lit/ds/symlink/ina260.pdf. [Accessed 28 April 2022].

[2] Sensirion, “SHT4x - 4th Generation, High-Accuracy, Ultra-Low-Power, 16-bit Relative Humidity and Temperature Sensor,” July 2021. [Online]. Available: https://eu.mouser.com/datasheet/2/682/Sensirion_Humidity_Sensors_SHT4x_Datasheet-2001050.pdf. [Accessed 28 April 2022].

[3] Raspberry Pi Foundation, “Raspberry Pi 4,” [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/. [Accessed 28 April 2022].

[4] S. Monk, “Adafruit's Raspberry Pi Lesson 4. GPIO Setup - Configuring I2C,” Adafruit, [Online]. Available: https://learn.adafruit.com/adafruits-raspberry-pi-lesson-4-gpio-setup/configuring-i2c. [Accessed 28 April 2022].

[5] NXP Semiconductors, “UM10204 - I2C-bus specification and user manual,” 1 October 2021. [Online]. Available: https://www.nxp.com/docs/en/user-guide/UM10204.pdf. [Accessed 28 April 2022].

[6] B. Siepert, “Adafruit INA260 Current + Voltage + Power Sensor Breakout - Assembly,” Adafruit, [Online]. Available: https://learn.adafruit.com/adafruit-ina260-current-voltage-power-sensor-breakout/assembly. [Accessed 28 April 2022].

[7] J. Valdez and J. Becker, “Understanding the I2C Bus,” Texas Instruments, June 2015. [Online]. Available: https://www.ti.com/lit/an/slva704/slva704.pdf. [Accessed 28 April 2022].

5. Contacts

Project Coordinator:

- Name: Technical University of Sofia
- Address:
    - Technical University of Sofia,
      Kliment Ohridsky Bd 8
      1000, Sofia, Bulgaria
- Phone: +3592623073

Output 2 Leader:

- Name: FOSS Research Centre for Sustainable Energy, University of Cyprus
- Address:
    - University of Cyprus,
      Panepistimiou 1 Avenue
      P.O. Box 20537
      1678, Nicosia, Cyprus
- Email: foss@ucy.ac.cy
- Phone: +357 22 894288