Exercises

Basic Programming Exercise:

A bit of programming

Connect your ESP32 to the PC using the micro USB cable and start thonny. Locate the shell window and check if you see the PEPL prompt ">>>". If not, try to press <enter>. Make sure the shell window has the input focus. If this does not work, press the stop button. If this also doesn't help, press the reset button on the ESP32 CPU board.

Playing with REPL

```
print "Hello World!"
calculate 127,9 * 157.6 * 17.2 /(3.5*2**4) # 2**4 means 2 to the
power of 4
```

you may try to calculate e.g 2**4 separately first and check that you get the result expected
Continue experimenting with REPL

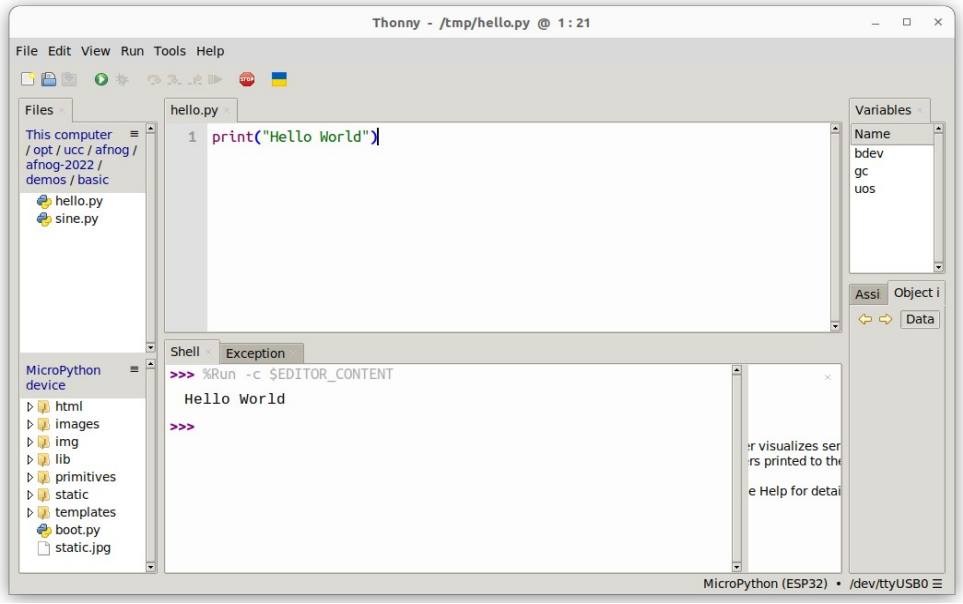
Hello World

Instead of working on the shell window, we will now write a Python script (a small program). For this to work, select File -> New. This will create an editor window named <untitled>. Save this empty program to the disk on your PC. Select File -> Save, select "This Computer" give it a file name (e.g. hello.py) and press "OK". The window title changes to hello.py.

Now enter

```
print("Hello World!")
```

Finally, run the program by pressing the green button with the white triangle. You should see "Hello World!" printed in the shell window. This is what you should get:



Some Arithmetic

Assign 2 values to 2 variables a and b. Calculate the sum, the difference, the product and the division and print the results. Start assigning integer values. Try integer and float division.

Then assign float values and try again.

Conditions

Assign again values to two variables a and b with a being bigger than b . In your program, check if a is equal, bigger or smaller than b. Modify the values you assign to a and b, with b now bigger than a and try again.

In the lectures, we used the "<" and the ">" conditions to find out if the value in the variable a is bigger, smaller or equal to the value in variable b. Can you modify the program using "<" and "!=" to find out?

What happens if you try:

```
if a:
   print("yes")
else:
   print("No")
```

Try this with a being assigned to zero and a being assigned to 7. What do you conclude?

Loops

Write a script that prints "Hello World!" forever. You can stop the program with the red Stop button.

Write a program that prints "Hello World!" 7 times. Do this using a while loop and in a second program, using a for loop, Which of the programs is more elegant?

Too easy? Try to calculate the Fibonacci numbers

The Fibonacci numbers are defined as follows:

- F(0) = 0
- F(1) = 1
- F(n) = F(n-1) + F(n-2)

Write a program that prints the Fibonacci series of numbers for n=10, n=20, n=30.

This is the end of the first exercise session

Exercises on Functions and Modules

MicroPython modules

Modify the loop program in such a way that "Hello World!" is printed only every second. Import the utime module and make the program sleep for 1s after each print statement.

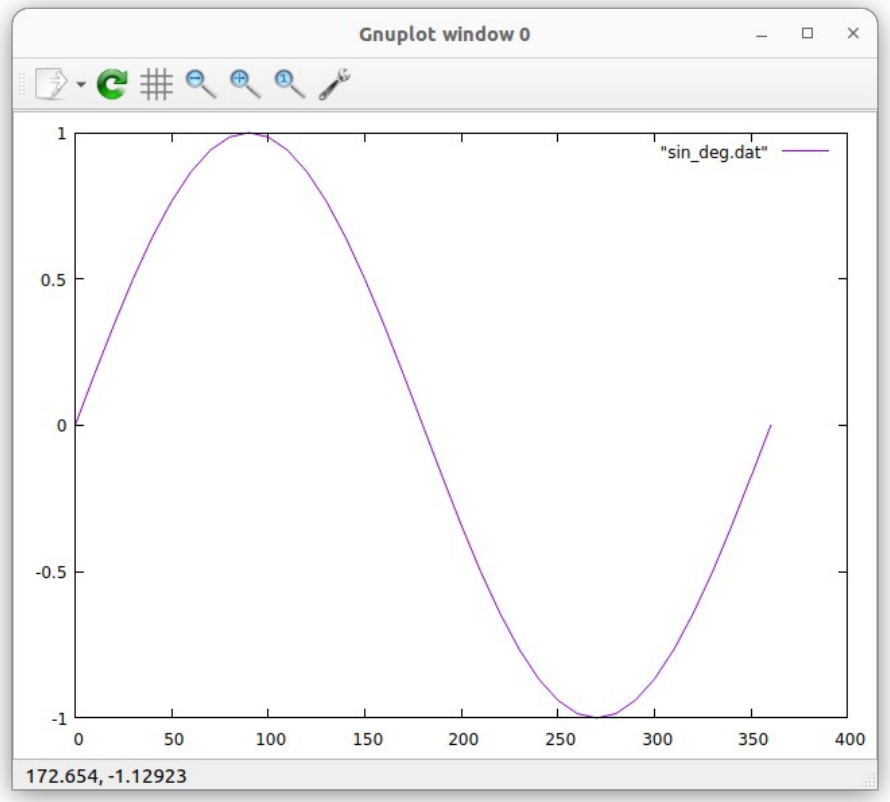
Import the sin function and the value of pi from the math module

```
from math import sin,pi
```
Remember that the sin function takes its values in units of radians. The angle values for a complete sin period will range 0 .. 2*pi. Print the angles and the sin values for a full sin period using 30 equidistant angle values. (x and sin(x)).

Creating and using your own function

Create a sin_deg function which takes angles in units of degrees instead of radians. Print the angles and the sin values for 36 equidistant angle values.

You may copy/paste the result to a new editor window in thonny and save them to a file on the PC. Then you can plot the function using gnuplot.



Too easy? Calculate the throwing parabola

If the above is too simple for you, you may try to calculate the *throwing parabola*. Let's say, a stone is thrown at a speed of 30m/s and an angle of 30° with respect to ground. Calculate the trajectory of the stone until it hits the ground. You way define a function *trajectory*, taking initial speed, the angle and the time resolution for which you want to calculate the stone positions.

v_hor = v_initial * cos(angle)

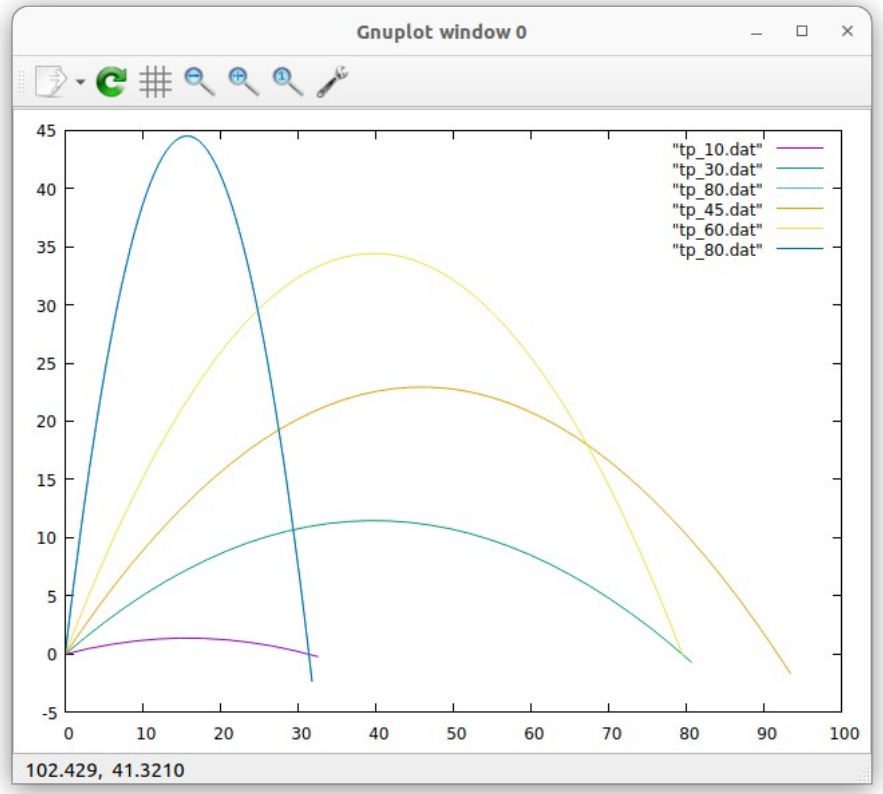
v_ver = v_initial * sin(angle)

x(t) = v_hor * t

y(t) = v_ver * t - 1/2 * g * t**2

Like this, you can easily calculate the trajectories for different angles and different initial speeds. Use a time resolution of 0.1 ms to get enough curve points.

In the plot below an initial speed on 30 m/s is assumed. Tp_10 shows the trajectory for an angle of 10 degrees.



Improving the program

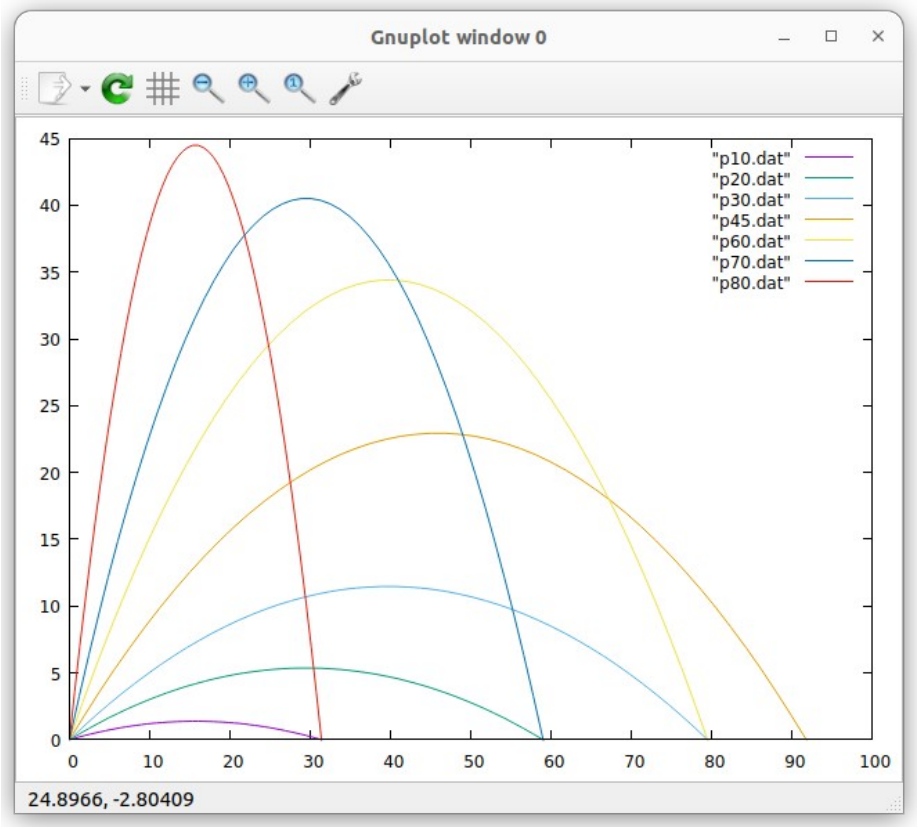
This program can further be improved by plotting the same number of points for each parabola. To achieve this, we can first calculate the total flight duration until the object hits the ground again:

```
vt = 1/2*g*t**2, v => 1/2*g*t, t= 2*v/g
```

```
v is the vertical speed in this case.
```

Then we calculate a fixed number of points (e.g. 50) for each parabola in a for loop.

Like this, we can easily see that for an angle of 30° and 60° the distance covered is the same. This is also true for 10° and 80° or 20° and 70°.



End of the second exercise session

Exercises with GPIO, the Pushbutton and LEDs

Programming the user LED on the CPU card

Write a program that blinks the user programmable LED on the CPU card at a frequency of 1 Hz (500 ms on, 500 ms off)
The user LED on the CPU card is connected to GPIO 2.

Write a function that blinks the LED once with a delay that is passed as a parameter:

```
def blink_once(delay):
```

Test the function with a main program calling is with different delays.

Use the function to implement a program that blinks an SOS: 3 short pulses, followed by 3 long pulses, followed by 3 short pulses, followed by a 2s pause. You may use a 200 ms delay for the short pulses and 700 ms delay for the long ones.

Reading the pushbutton

Plug the pushbutton shield into the triple base.
Write a program that reads the push button state ever 100 ms and prints its state (pressed or released). The push button is connected to GPIO 17.

Improve the program printing the state only when there was a state change.

Too easy? Change the light intensity on the LED using PWM

Write a program that linearly increases the light intensity of the user programmable LED using **P**ulse **W**idth **M**odulation (PWM).

End of the third exercise session

Exercises with Analogue Signals and NeoPixels

Reading an analogue signal level from the slider potentiometer

Read the documentation of the ADC driver in the MicroPython manual
Connect the potentiometer to the triple base according to this table:

Potentiometer	**Triple Base**
OTA	A0 (ADC) GPIO 36
VCC	3V3
GND	GND

Create an ADC object and make sure you set the attenuator to 11 dB
Read the slider value every 100 ms and print its raw 12 bit value.
Move the slider and observe the changes in the value

End of the forth exercise session

Controlling the rgb LED ring

The LED ring consists of 7 WS2812B addressable and cascadable LEDs. Each rgb LED consists of 3 tiny single color LEDs emitting red, green and blue light. The color you finally see is the mixture between these color components. The MicroPython driver for this type of LED is called NeoPixel. The color is defined as a tuple with three 8 bit color component intensities. 8 bits means that you can have values in the range 0..255. Since the LEDs are extremely bright, please restrict the values to 0..31.

Write a program trying to find out to which LED an LED number corresponds to. Where on the LED ring is LED0, LED1 ... Do this by lighting the LEDs one by one with only the red color component switched on. The green and blue are set to zero.
Create a mapping table such that the first LED is mapped to the top one, the following LEDs are addressed clockwise and the last LED is the middle one.

Then change the color. Try all color combinations (31,0,0), (0,31,0), (0,0,31) but also (31,31,0), (31,0,31) ... Which colors do you get?

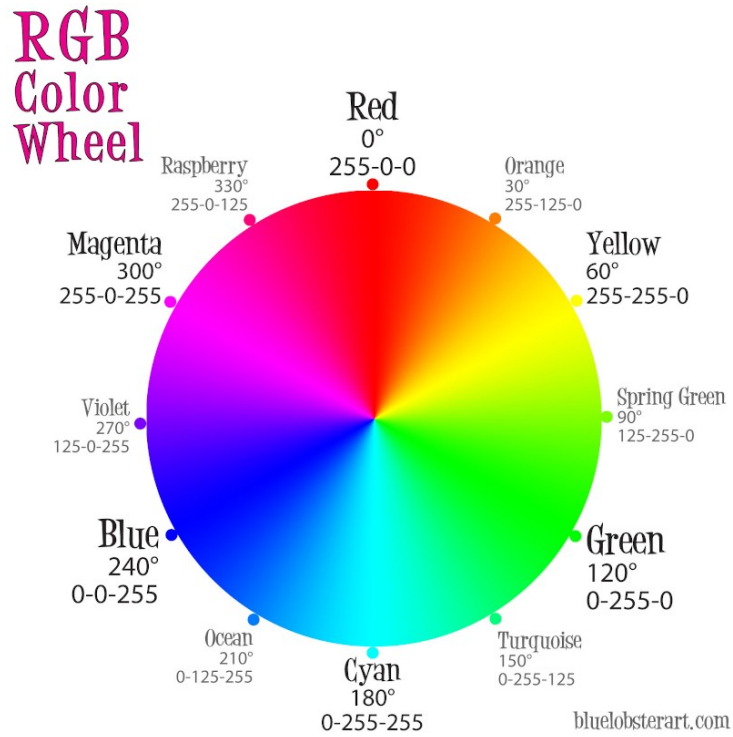
You can further modify the colors by changing the intensity values. For example: (12,27,5) Play around trying to generate as many different colors as you like.

Write a program that lights the top LED in blue and then lights the following LEDs in clockwise direction to :

- blue (top LED)
- cyan
- green
- yellow
- magenta
- red
- the middle LED becomes white

Too easy? Program the Color Wheel

Write a program that shows all the colors of the color wheel on the LEDs of the rgb LED ring. The



color on the LEDs will slowly change from red to yellow to green … to magenta to red.

A small project

As a small project, we are going to combine the reading of the ADC and the control of the rgb LED ring. As we saw, the ADC delivers values between 0 and 4095 (12 bits). We can divide this range in 7 slices, corresponding to the 7 LEDs in the ring. For the lowest slice, the top LED will be lit. Then we go clockwise around the ring: the next LED lit will be to the right of the first one. For the highest slice, the middle LED will be lit.
First, print the led number for the signal level read from the slider. Move the slider and make sure that you get the correct LED number.

Then light the LED you found with a single color, e,g, red.

Very often, the color indicates the signal level: For low signal levels, we use "cold" colors (blue, cyan) for higher levels we use "hot" colors (magenta, red, white). The color for the top LED will therefore be blue, the color for the middle LED will be white.

We can then attribute a different color to each LED, as the lighting of the LED depends on the signal level reached. We only use the basic colors:

LED number	color	
1	blue	(0,0,31)
2	cyan	(0,31,31)
3	green	… you find the others
4	yellow	
5	magenta	
6	red	

0 white

The top LED (lowest signal level) will therefore have the color blue, while the middle LED (highest signal level) will become white.

Light the LED with the color corresponding to the signal level.

End of the fifth exercise session

Exercises of the I2C bus

Finding I2C addresses

Connect the BMP180 temperature and air pressor sensor to the WeMos D1 mini bus.

Find the addresses of all I2C slaves connected to the bus and print them. Use the ESP32 hardware interface of the I2C bus connected to bus no. 1. The I2C interface uses the following pins:

	Bus number	**SCL**	**SDA**
	1		
GPIO		22	21

You can find the MicroPython functions to be used with the I2C bus on https://docs.micropython.org/en/latest/esp32/quickref.html#hardware-i2c-bus and

https://docs.micropython.org/en/latest/library/machine.I2C.html#machine.I2C

i2c.scan returns an address list of all I2C slaves found on the bus. With just the SHT30 connected you should see:

```
Scanning the I2C bus
Program written for the workshop on IoT at the
African Internet Summit 2019
Copyright: U.Raich
Released under the Gnu Public License
Running on ESP32
I2C slaves found on the bus:
0x45

>>>
```

I found this program very useful because each time an I2C interface is tested we need to make sure that the device is actually seen by our program. Try to improve the formatting of the program to print all I2C addresses in a table:

```
>>> %Run -c $EDITOR_CONTENT

MPY: soft reboot
Scanning the I2C bus
Program written for the workshop on IoT at the
African Internet Summit 2019
Copyright: U.Raich
Released under the Gnu Public License
Running on ESP32
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- 3c -- -- --
40: -- -- -- -- -- 45 -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- 77 -- -- -- -- -- -- -- --

>>>
```

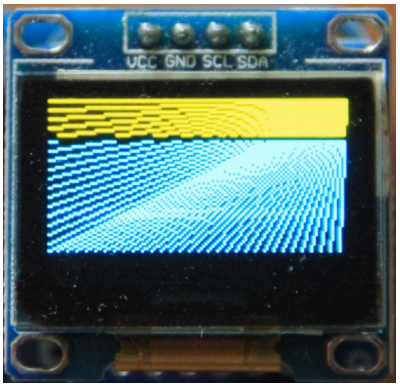
End of the sixth exercise session

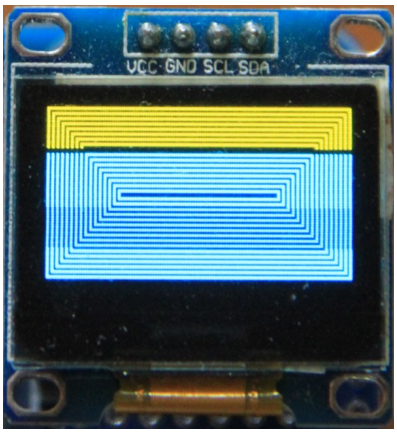
Exercises on the SSD1306 display driver and the 128x64 pixel OLED display

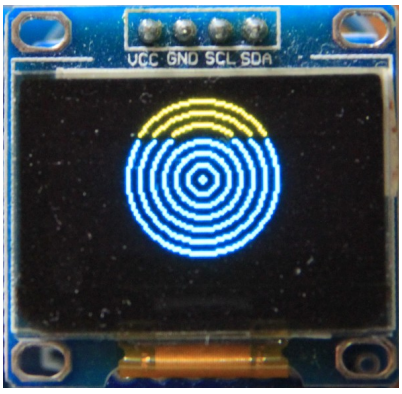
Instead of reading and writing the SSD1306 registers directly, we will rely on a software module, the SSD1306 class, which is already included in the MicroPython firmware. In fact the SSD1306 class is a sub-class of framebuf (https://docs.micropython.org/en/latest/library/framebuf.html) and inherits all its methods. The SSD1306 class does not write into the display's pixel memory directly but uses a memory buffer on the ESP32 RAM having the same size as the display. The SSD1306 documentation (https://docs.micropython.org/en/latest/esp8266/tutorial/ssd1306.html) shows how to use the driver and also provides simple example code. Just remember that on our ESP32, SCL is connected to GPIO 21 and SDA to GPIO 22.

- Modify the example code correspondingly and try the "Hello World!" example.
- Then copy/paste the example code showing the MicroPython logo and some text.
- Try the graphics primitives to draw lines, rectangles, circles, text, triangles (there is no primitive to draw triangles but you can easily draw them by drawing 3 lines). You may also experiment with filling these graphics elements.

Here is how the result may look like:







The text examples shows all ASCII characters from 0x20 to 0x7e. The other codes are command codes and are not visible.

End of the seventh exercise session

Exercises on I2C programming and the BMP180

The BMP180 is a barometric pressure and temperature sensor. On this sensor we can easily try the I2C read and write functions. All the necessary descriptions are found in the BMP180 data sheet.

As we can see from the table below, there is the ID register at address 0xD0, which, when being read, returns the constant value 0x55 identifying the device as a BMP180. The ctrl_meas register is a read/write register and which we can use to test reading and writing.

Register Name	Register Adress	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	Reset state
out_xlsb	F8h	adc_out_xlsb<7:3>					0	0	0	00h
out_lsb	F7h	adc_out_lsb<7:0>								00h
out_msb	F6h	adc_out_msb<7:0>								80h
ctrl_meas	F4h	oss<1:0>		sco	measurement control					00h
soft reset	E0h	reset								00h
id	D0h	id<7:0>								55h
calib21 downto calib0	BFh *down* to AAh	calib21<7:0> *down* to calib0<7:0>								n/a

Registers:	Control registers	Calibration registers	Data registers	Fixed
Type:	read / write	read only	read only	read only

In order to further explore I2C access we want to try reading and writing a register on the BMP180 air pressure sensor.

- Write a program that reads the BMP180 ID register and returns its value. Use the method readfrom_mem of the MicroPython I2C class (https://docs.micropython.org/en/latest/library/machine.I2C.html#machine.I2C) to accomplish this.

- Write the value 0xC0 to the BMP180 ctrl_meas register (method writeto_mem) and read the register back. Make sure you read the same value that you have written (You should only write the two most significant bits).

- You may also try to start a measurement and check if the conversion has been accomplished by looking at the sco bit in the ctrl_meas register.

The Bmp180Class

The algorithm to convert raw temperature and humidity values into physical values is pretty complex. Therefore a driver class is provided in the MicroPython firmware, hiding this complexity.

- Write a program reading and printing temperature and air pressure from the BMP180 using the Bmp180 class. You must import Bmp180 from the Bmp180Class module, create a Bmp180 object, call the measure() method and finally read the temperature and air pressure with getTemperature() and getPressure(). These methods will return physical values in °C for the temperature and hPa for the air pressure.

End of the eighth exercise session

Exercise reading temperature and humidity on the SHT30

Sensors can be quite complex devices with a big number of registers and programming them can be a challenge. You may check this out for yourself, having a look at the SHT30 data sheet.

To make things easier for you, a driver is included in the MicroPython interpreter installed on your ESP32 CPU.

Here are the methods you need to make temperature and humidity measurements with the driver:

First import the SHT3X and SHT3XError modules:

```
# import the SHT3X class
from sht3x import SHT3X,SHT3XError
```

After that, create an SHT3X object:

```
# create a SHT3X object
try:
    sht30 = SHT3X()
except SHT3XError as exception:
    if exception.error_code == SHT3XError.BUS_ERROR:
        print("SHT30 module not found on the I2C bus, please connect it")
        sys.exit(-1)
    else:
         raise exception
```

Now we are ready to read the serial number and to start measurements and read the results:

```
# read out the serial number
print("Serial number: 0x{:02x}".format(sht30.serialNumber()))

# start a measurement and read the results
tempC, humi = sht30.getTempAndHumi()
```

- Write a program to read the serial number of your SHT30 and then repeatedly read the current temperature and humidity at a frequency of 1 Hz and print the results.

A small project: The weather station

Now that we know how to read temperature , humidity and the air pressure and we also know how to write text to the OLED display, we can create a simple weather station.

- First write the text frame onto the OLED display into which the measurement results will be inserted.
- Write a program that reads temperature, humidity and air pressure and prints the results onto the serial monitor
- Calculate the position where the results must be written onto the OLED screen and integrate the values into the weather frame.

End of the nineth exercise session

The frame for the weather station

The weather station with results integrated

Stepping motor exercise

Before you start programming, the stepper motor driver card must be connected to the WeMos D1 mini bus. Please make the connections as described in the connection table. Please double check the connections before powering the micro-controller.

Driver Card	WeMos D1 mini bus	GPIO number
- (5--12V)	GND	
+ (5--12V)	5V	
IN1	D8	5
IN2	D7	23
IN3	D6	19
IN4	D5	18

A stepping motor driver implemented as a Python class and is already available in the MicroPython firmware. The module name is *stepper*, the class name is *steppingMotor*. You can therefore create an object of the steppingMotor class with the following code:

```
from stepper import steppingMotor
stepper = steppingMotor()      # this creates an object of class steppingMotor
                               # named stepper
```

Here are the features of the class:

```
move(noOfSteps)  # move noOfSteps using the current stepping motor mode
(default: single step forward)
clrAll()         # sets all four coil signals to GND
stepMode(mode)   # if mode == None: returns the current mode else: sets the
stepping mode to "mode"
                 # mode can be:
                 #   stepper.SINGLE_PHASE_FORWARD
                 #   stepper.SINGLE_PHASE_BACKWARD
                 #   stepper.DOUBLE_PHASE_FORWARD
                 #   stepper.DOUBLE_PHASE_BACKWARD
                 #   stepper.HALF_STEP_FORWARD
                 #   stepper.HALF_STEP_BACKWARD
waitAfterSteps(ms__wait) # wait for for ms_wait [ms] after each step (if ms_wait
!= None). ms_wait must be in the range 2..1000
                           # else return the current ms_wait value
```

- Write a program that makes use of the steppingMotor class. Try the different stepping modes and change the movement speed.
- Write a program to move the stepping motor in single phase forward mode, without passing though the driver. At first use a long delay between each step (500ms). This allows you to see the state of the phases on the hardware driver LEDs.

End of the tenth exercise session