Lab 40

RISC-V microPython – Installation and first examples

MicroPython is a lean and efficient implementation of the Python 3 programming language that includes a small subset of the Python standard library and is optimised to run on microcontrollers and in constrained environments. It aims to be as compatible with normal Python as possible to allow you to transfer code with ease from the desktop to a microcontroller or embedded system.

We are going to use Thonny IDE to develop and load the microPython codes to our RISC-V based IoT Devkit (pyWiFi-ESP32-C3).

Thonny is an **Integrated Development Environment** (**IDE**) for **Python** beginners. It is created with **Python** and released under **MIT** License. It is cross-platform and can run in **Linux**, **macOS**, **Windows**.

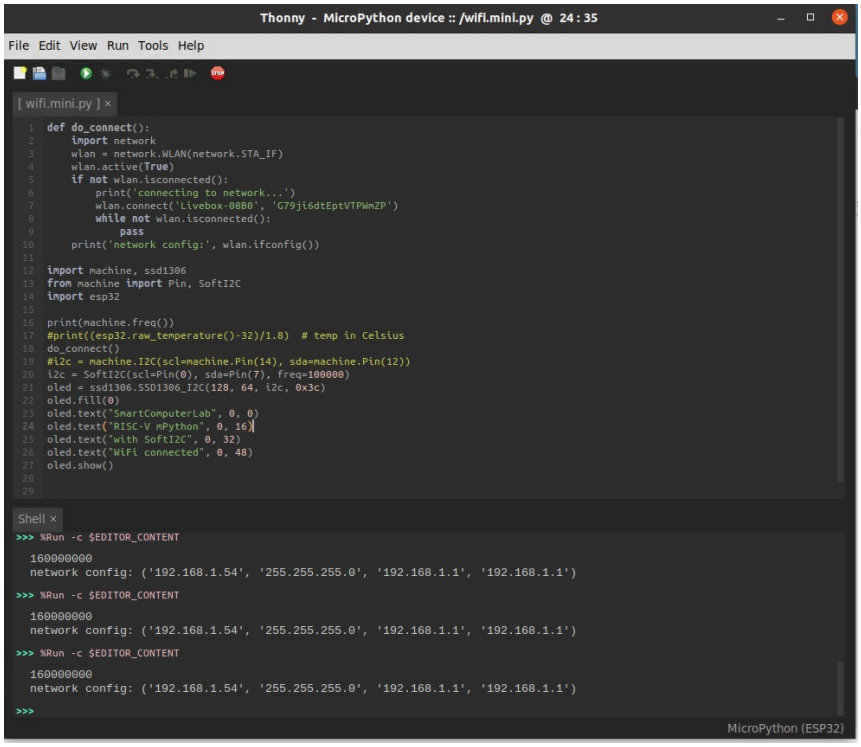
Some of the key features of **thonny** include

- Python 3.7 is installed by default with **Thonny** setup.
- Built-in Debugger and Step through evaluation.
- Variable Explorer.
- Heap, Stack, Assistant, Object Inspector.
- Built-in Python shell (Python 3.7).
- Simple PIP GUI Interface to install 3rd party packages.
- Support code completion.
- Highlights syntax errors and explain scopes.

Thonny IDE installation

```
pip3 install thonny
```

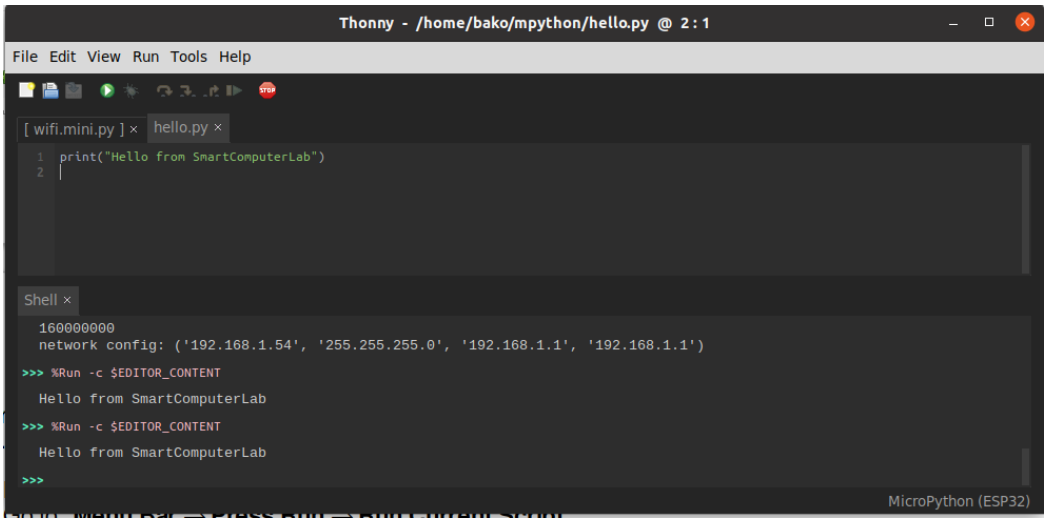
To launch **thonny**, go to the installed directory and type "`./thonny`" or absolute path to thonny. Thonny will ask you to set up Language and Initial settings.



There are 3 ways you can run the code you created. First, your code should be saved to a file for Thonny to execute.

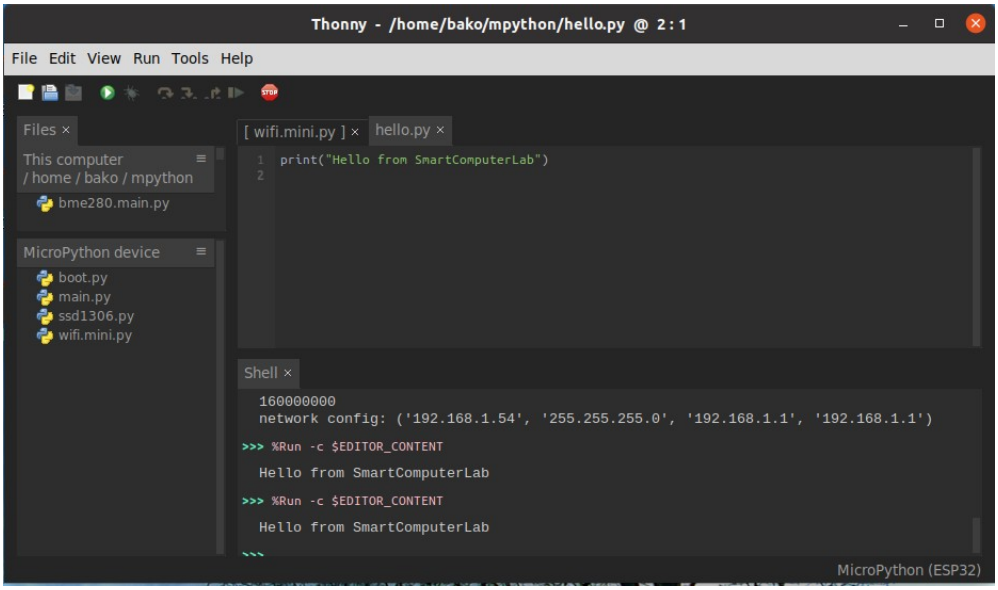
- Press **F5** or **Execute Icon** as shown in Image.
- Go to "**Menu Bar** → **Press Run** → **Run Current Script**".
- Press "**CTRL+T**" or Go to "**Run** → **Press Run current script in terminal**".

The first two methods will switch the directory to wherever your code is and invoke the program file in the Built-in terminal.



The third option allows you to run your code in an external terminal.

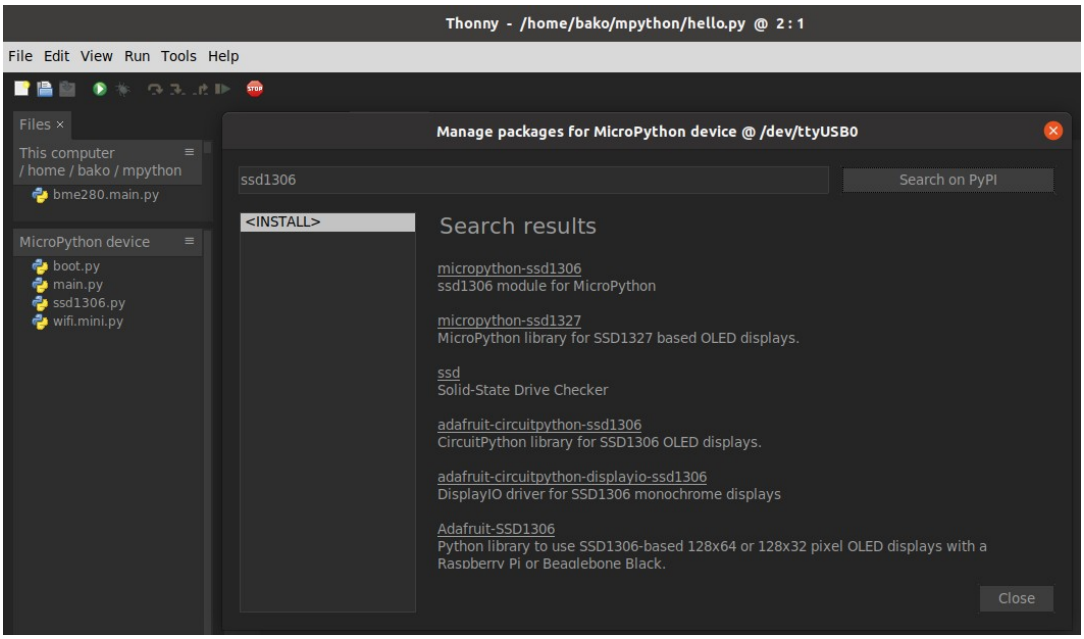
The real power of thonny comes with built-in features like File Explorer, Variable Explorer, Shell, Assistant, Notes, Heap, Outline, Stack. To Toggle on-off these features Go to "**View** → toggle Feature **ON**/**OFF**".



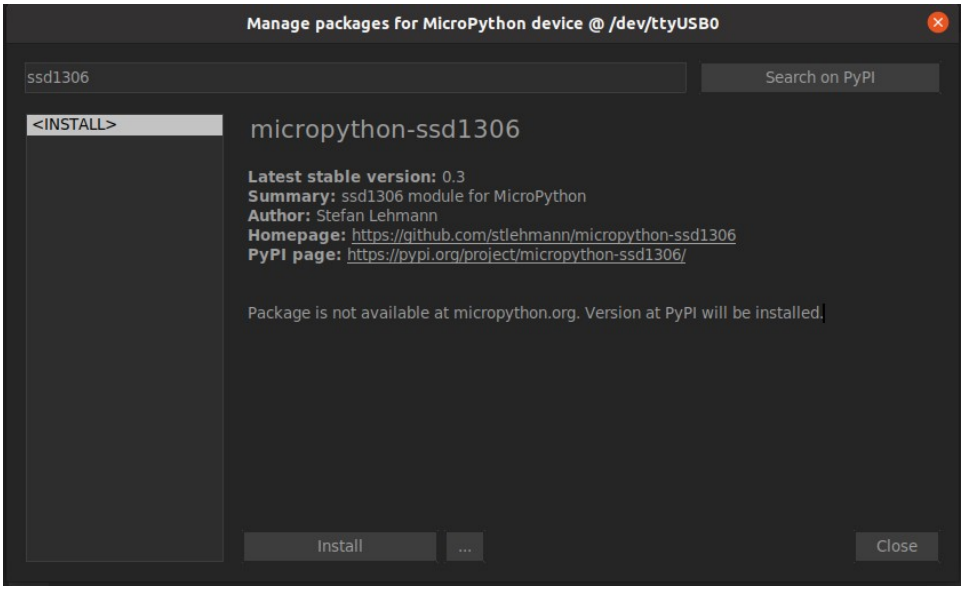
Thonny Package Manager

It is known that all the python packages are hosted at **PyPI**. We will normally use **PIP** (**Python Package Manager**) to install desired packages from **PyPI**. But with **Thonny**, a GUI interface is available to manage packages.
Go to "**Menu Bar** → **Tools** → **Packages**". In the search bar, you can type a package name and press search. It will search the **PyPI** index and displays the list of package matching the name.

In my case, I am trying to install a package called **ssd1306**



When you select the package from the list, It will take you to the installation page. You can install the latest version or choose different versions as shown in the image. Dependencies are automatically installed.



Lab 1 – OLED display and basic sensors (I2C)

In this lab we are going to build simple applications including the use of OLED display and a number of sensors. All this sensors as well as the OLED display are connected to the main board via I2C bus.

The **I2C** bus operates with two lines: **SDA** (Serial Data/Address communication) and **SCL** (Serial Clock) for the synchronization of the binary values send over the SDA line. Each device connected to the I2C bus is identified by its address.

For example the OLED screen is identified by **`0x3c`** in hexadecimal coding.

We start with the OLED display.

1.1 OLED

SSD1306 is a single-chip CMOS OLED/ driver with controller for organic / polymer light emitting diode dot-matrix graphic display system. It consists of 128 columns and 64 lines.

```
import machine, ssd1306
from machine import Pin, SoftI2C
import esp32

print(machine.freq())

i2c = SoftI2C(scl=Pin(0), sda=Pin(7), freq=100000)
oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
for i in range(100):
    d1=i+1
    d2=i*2
    ds1=str(d1)
    ds2=str(d2)
    oled.fill(0)
    oled.text("SmartComputerLab", 0, 0)
    oled.text(ds1, 0, 16)
    oled.text(ds2, 0, 32)
    oled.show()
```

The following is the **ssd1306** driver

```
# MicroPython SSD1306 OLED driver, I2C and SPI interfaces

from micropython import const
import framebuf

# register definitions
SET_CONTRAST = const(0x81)
SET_ENTIRE_ON = const(0xA4)
SET_NORM_INV = const(0xA6)
SET_DISP = const(0xAE)
SET_MEM_ADDR = const(0x20)
SET_COL_ADDR = const(0x21)
SET_PAGE_ADDR = const(0x22)
SET_DISP_START_LINE = const(0x40)
SET_SEG_REMAP = const(0xA0)
SET_MUX_RATIO = const(0xA8)
SET_COM_OUT_DIR = const(0xC0)
SET_DISP_OFFSET = const(0xD3)
SET_COM_PIN_CFG = const(0xDA)
SET_DISP_CLK_DIV = const(0xD5)
SET_PRECHARGE = const(0xD9)
SET_VCOM_DESEL = const(0xDB)
SET_CHARGE_PUMP = const(0x8D)

# Subclassing FrameBuffer provides support for graphics primitives
# http://docs.micropython.org/en/latest/pyboard/library/framebuf.html
class SSD1306(framebuf.FrameBuffer):
```

```
    def __init__(self, width, height, external_vcc):
        self.width = width
        self.height = height
        self.external_vcc = external_vcc
        self.pages = self.height // 8
        self.buffer = bytearray(self.pages * self.width)
        super().__init__(self.buffer, self.width, self.height, framebuf.MONO_VLSB)
        self.init_display()

    def init_display(self):
        for cmd in (
            SET_DISP,  # display off
            # address setting
            SET_MEM_ADDR,
            0x00,  # horizontal
            # resolution and layout
            SET_DISP_START_LINE,  # start at line 0
            SET_SEG_REMAP | 0x01,  # column addr 127 mapped to SEG0
            SET_MUX_RATIO,
            self.height - 1,
            SET_COM_OUT_DIR | 0x08,  # scan from COM[N] to COM0
            SET_DISP_OFFSET,
            0x00,
            SET_COM_PIN_CFG,
            0x02 if self.width > 2 * self.height else 0x12,
            # timing and driving scheme
            SET_DISP_CLK_DIV,
            0x80,
            SET_PRECHARGE,
            0x22 if self.external_vcc else 0xF1,
            SET_VCOM_DESEL,
            0x30,  # 0.83*Vcc
            # display
            SET_CONTRAST,
            0xFF,  # maximum
            SET_ENTIRE_ON,  # output follows RAM contents
            SET_NORM_INV,  # not inverted
            # charge pump
            SET_CHARGE_PUMP,
            0x10 if self.external_vcc else 0x14,
            SET_DISP | 0x01,  # display on
        ):  # on
            self.write_cmd(cmd)
        self.fill(0)
        self.show()

    def poweroff(self):
        self.write_cmd(SET_DISP)

    def poweron(self):
        self.write_cmd(SET_DISP | 0x01)

    def contrast(self, contrast):
        self.write_cmd(SET_CONTRAST)
        self.write_cmd(contrast)

    def invert(self, invert):
        self.write_cmd(SET_NORM_INV | (invert & 1))

    def rotate(self, rotate):
        self.write_cmd(SET_COM_OUT_DIR | ((rotate & 1) << 3))
        self.write_cmd(SET_SEG_REMAP | (rotate & 1))

    def show(self):
        x0 = 0
        x1 = self.width - 1
        if self.width == 64:
            # displays with width of 64 pixels are shifted by 32
            x0 += 32
            x1 += 32
        self.write_cmd(SET_COL_ADDR)
        self.write_cmd(x0)
        self.write_cmd(x1)
        self.write_cmd(SET_PAGE_ADDR)
        self.write_cmd(0)
        self.write_cmd(self.pages - 1)
```

```
        self.write_data(self.buffer)


class SSD1306_I2C(SSD1306):
    def __init__(self, width, height, i2c, addr=0x3C, external_vcc=False):
        self.i2c = i2c
        self.addr = addr
        self.temp = bytearray(2)
        self.write_list = [b"\x40", None]  # Co=0, D/C#=1
        super().__init__(width, height, external_vcc)

    def write_cmd(self, cmd):
        self.temp[0] = 0x80  # Co=1, D/C#=0
        self.temp[1] = cmd
        self.i2c.writeto(self.addr, self.temp)

    def write_data(self, buf):
        self.write_list[1] = buf
        self.i2c.writevto(self.addr, self.write_list)


class SSD1306_SPI(SSD1306):
    def __init__(self, width, height, spi, dc, res, cs, external_vcc=False):
        self.rate = 10 * 1024 * 1024
        dc.init(dc.OUT, value=0)
        res.init(res.OUT, value=0)
        cs.init(cs.OUT, value=1)
        self.spi = spi
        self.dc = dc
        self.res = res
        self.cs = cs
        import time

        self.res(1)
        time.sleep_ms(1)
        self.res(0)
        time.sleep_ms(10)
        self.res(1)
        super().__init__(width, height, external_vcc)

    def write_cmd(self, cmd):
        self.spi.init(baudrate=self.rate, polarity=0, phase=0)
        self.cs(1)
        self.dc(0)
        self.cs(0)
        self.spi.write(bytearray([cmd]))
        self.cs(1)

    def write_data(self, buf):
        self.spi.init(baudrate=self.rate, polarity=0, phase=0)
        self.cs(1)
        self.dc(1)
        self.cs(0)
        self.spi.write(buf)
        self.cs(1)
```

1.2 SHT21

The SHT21 is an I2C temperature and humidity sensor from Sensirion. The **SHT21** digital humidity and temperature sensor is fully calibrated and offers high precision and excellent long-term stability at low cost. The digital CMOSens Technology integrates two sensors and readout circuitry on one single chip.

In this example the driver part is included in the begining of the main program.

```
# Read SHT21/HTU21 temperature and humidity
# Print results to shell
# Device I2C address = 0x40 (hex) = 64 decimal
# Command registers
# Soft reset = 0xFE
# Write user register = 0xE6
# Read user register = 0xE7
# Trigger temperature reading and hold the bus = 0xE3
# Trigger humidity reading and hold the bus = 0xE5
# Trigger temperature reading without holding the bus = 0xF3
# Trigger humidity reading without holding the bus = 0xF5

STATUS_BITS_MASK = 0xFFFC

import machine
from machine import  Pin,I2C
import utime
sda=machine.Pin(7)
scl=machine.Pin(0)

i2c=machine.I2C(0,sda=sda, scl=scl, freq=400000) # I2C channel 0, pins, 400kHz max

print("I2C Address       : "+hex(i2c.scan()[0]).upper()) # Display device address
print("I2C Configuration : "+str(i2c))                   # Display I2C config

data = []
address = 64

# The status register must be read and then just the valid control bits changed
i2c.writeto(address, b'\xE7')    # Read user register
data = i2c.readfrom(address, 1)  # Get the 1 byte result
##print ("Status register = " '{:08b}'.format(data[0])) # print as a 8 bit binary with leading zeros

if data[0] & (1<<6):
    print ("Supply voltage is under 2.25V")
else:
    print ("Supply voltage is over 2.25V")

reg = bytearray(1)

reg[0] = (data[0] & 0xFB) # Turn off heater
#reg[0] = (data[0] | 0x04) # Turn on heater

i2c.writeto_mem(address, 0xE6, reg)

# Read the register back to check it has been changed
i2c.writeto(address, b'\xE7')    # Read user register
data = i2c.readfrom(address, 1)  # Get the 1 byte result
#print ("Status register = " '{:08b}'.format(data[0])) # print as a 8 bit binary with leading zeros

if data[0] & (1<<2):
    print ("On-chip heater is ON")
else:
    print ("On-chip heater is OFF")

print("Measurement resolution: ",sep=' ', end='') # print with no linefeed

if data[0] & (1<<7):
    if data[0] & (1<<0):
        print ("11 bit humidity, 11 bit temperature")
    else:
        print ("10 bit humidity, 13 bit temperature")
else:
    if data[0] & (1<<0):
        print ("8 bit humidity, 12 bit temperature")
```

```
    else:
        print ("12 bit humidity, 14 bit temperature")


def read_humidity():
    i2c.writeto(address, b'\xF5')   # Trigger humidity measurement
    utime.sleep_ms(29)              # Wait for it to finish (29ms max)
    data = i2c.readfrom(address, 2) # Get the 2 byte result
    adjusted = (data[0] << 8) + data[1]   # convert to 16 bit value
    adjusted &= STATUS_BITS_MASK          # zero the status bits
    adjusted *= 125                       # scale
    adjusted /= 1 << 16                   # divide by 2^16
    adjusted -= 6                         # subtract 6
    sht21_humi=adjusted
    return sht21_humi


def read_temperature():
    i2c.writeto(address, b'\xF3')   # Trigger temperature measurement
    utime.sleep_ms(85)              # Wait for it to finish (85ms max)
    data = i2c.readfrom(address, 2) # Get the 2 byte result
    #print (data[0]) # Debug only print byte 0
    #print (data[1]) # Debug only print byte 1
    ## Compute temperature
    adjusted = (data[0] << 8) + data[1]   # convert to 16 bit value
    adjusted &= STATUS_BITS_MASK          # zero the status bits
    adjusted *= 175.72                    # scale
    adjusted /= 1 << 16                   # divide by 2^16
    adjusted -= 46.85                     # subtract offset
    sht21_temp=adjusted
    return sht21_temp
```

```
for i in range(100):
    humi=read_humidity()
    print ("Humidity    = %.1f" % humi)
    temp=read_temperature()
    print ("Temperature = %.1f" % temp)
    utime.sleep(5)
```

To do

Write a code that combines the **STH21** sensor readings and display on OLED screen (SSD1306)

1.3 MAX44009 (GY-49)

The MAX44009 ambient light sensor features an I2C digital output that is ideal for a number of portable applications such as smartphones, notebooks, and industrial sensors. At less than 1µA operating current, it is the lowest power ambient light sensor in the industry and features an ultra-wide 22-bit dynamic range from 0.045 lux to 188,000 lux.

1.3.1 The MAX44009 driver code - max44009.py

```
from micropython import const

MAX44009_I2C_DEFAULT_ADDRESS = const(0x4A)

_MAX44009_REG_CONFIGURATION = const(0x02)
_MAX44009_REG_LUX_HIGH_BYTE = const(0x03)
_MAX44009_REG_LUX_LOW_BYTE  = const(0x04)

MAX44009_REG_CONFIG_CONTMODE_DEFAULT     = const(0x00)
# Default, low power, measures once every 800ms regardless of integration time
MAX44009_REG_CONFIG_CONTMODE_CONTINUOUS  = const(0x80)
# Continuous mode, readings are taken every integration time
MAX44009_REG_CONFIG_MANUAL_OFF           = const(0x00)
# Automatic mode with CDR and Integration Time  by autoranging
MAX44009_REG_CONFIG_MANUAL_ON            = const(0x40)
# Manual mode and range with CDR and Integration Time programmed by the user
MAX44009_REG_CONFIG_CDR_NODIVIDED        = const(0x00)
# Current Division Ratio not divided, all photodiode current goes to the ADC
MAX44009_REG_CONFIG_CDR_DIVIDED          = const(0x08)
# CDR  divided by 8, used in high-brightness situations
MAX44009_REG_CONFIG_INTRTIMER_800        = const(0x00)
# Integration Time = 800ms, preferred mode for boosting low-light sensitivity
MAX44009_REG_CONFIG_INTRTIMER_400        = const(0x01)
# Integration Time = 400ms
MAX44009_REG_CONFIG_INTRTIMER_200        = const(0x02)
# Integration Time = 200ms
MAX44009_REG_CONFIG_INTRTIMER_100        = const(0x03)
# Integration Time = 100ms, preferred mode for high-brightness applications
MAX44009_REG_CONFIG_INTRTIMER_50         = const(0x04)
# Integration Time = 50ms, manual mode only
MAX44009_REG_CONFIG_INTRTIMER_25         = const(0x05)
# Integration Time = 25ms, manual mode only
MAX44009_REG_CONFIG_INTRTIMER_12_5       = const(0x06)
# Integration Time = 12.5ms, manual mode only
MAX44009_REG_CONFIG_INTRTIMER_6_25       = const(0x07)
# Integration Time = 6.25ms, manual mode only

class MAX44009:

    def __init__(self, i2c, address=MAX44009_I2C_DEFAULT_ADDRESS):
        self.i2c = i2c
        self.address = address
        buffer = [0, 0]
        self.configuration = bytes(buffer[1])
 #MAX44009_REG_CONFIG_CONTMODE_DEFAULT | MAX44009_REG_CONFIG_MANUAL_OFF

    @property
    def configuration(self):
        return self._config

    @configuration.setter
    def configuration(self, value):
        self._config = value
        self.i2c.writeto_mem(self.address,0x03, self._config)

    @property
    def illuminance_lux(self):
       data = self.i2c.readfrom_mem(self.address,_MAX44009_REG_LUX_HIGH_BYTE, 2)
       exponent = (data[0] & 0xF0) >> 4
       mantissa = ((data[0] & 0x0F) << 4) | (data[1] & 0x0F)
       illuminance = (2**exponent)*mantissa*0.045
       return illuminance # float in lux
```

1.3.2 The main program

```
from machine import Pin, SoftI2C
import esp32
import max44009
import utime

address=0x4A
i2c = SoftI2C(scl=Pin(0), sda=Pin(7), freq=100000)

sensor=max44009.MAX44009(i2c,address)

for i in range(100):
    vf=sensor.illuminance_lux
    print(vf)
    utime.sleep(2)
```

1.4 BH1750

This is the BH1750 16-bit Ambient Light sensor from Rohm. Because of how important it is to humans and most other living things, sensing the amount of light in an environment is a common place to get started when learning to work with microcontrollers and sensors. Should we turn up the brightness of our display or dim it to save power? Which direction should your robot move to stay in an area with the most light? Is it day or night? All of these questions can be answered with the help of the BH1750.

The BH1750 provides 16-bit light measurements in lux, the SI unit for measuring light making it easy to compare against other values like references and measurements from other sensors. Able to measure from 0 to 65K+ lux, the BH1750. With some calibration and advanced adjustment of the measurement time, it can even be convinced to measure as much as 100,000 lux.

1.4.1 Combined driver and the main code

```
import time

OP_SINGLE_HRES1 = 0x20
OP_SINGLE_HRES2 = 0x21
OP_SINGLE_LRES = 0x23
DELAY_HMODE = 180  # 180ms in H-mode
DELAY_LMODE = 24  # 24ms in L-mode

def sample(i2c, mode=OP_SINGLE_HRES1, i2c_addr=0x23):
    #Performs a single sampling. returns the result in lux
    i2c.writeto(i2c_addr, b"\x00")  # make sure device is in a clean state
    i2c.writeto(i2c_addr, b"\x01")  # power up
    i2c.writeto(i2c_addr, bytes([mode]))  # set measurement mode
    time.sleep_ms(DELAY_LMODE if mode == OP_SINGLE_LRES else DELAY_HMODE)
    raw = i2c.readfrom(i2c_addr, 2)
    i2c.writeto(i2c_addr, b"\x00")  # power down again
    # we must divide the end result by 1.2 to get the lux
    return ((raw[0] << 24) | (raw[1] << 16)) // 78642
```

```
import machine
from machine import  Pin,I2C

sda=machine.Pin(7)
scl=machine.Pin(0)
i2c=machine.I2C(0,sda=sda,scl=scl,freq=400000) # I2C channel 0, pins, 400kHz max
lum=sample(i2c,mode=OP_SINGLE_HRES1, i2c_addr=0x23)
print('lum')
print(lum)
```

1.5 BMP280

Bosch has stepped up their game with their new BMP280 sensor, an environmental sensor with temperature, barometric pressure that is the next generation upgrade to the BMP085/BMP180/BMP183. This sensor is great for all sorts of weather sensing and can even be used in both I2C and SPI!

This precision sensor from Bosch is the best low-cost, precision sensing solution for measuring barometric pressure with ±1 hPa absolute accuracy, and temperature with ±1.0°C accuracy. Because pressure changes with altitude and the pressure measurements are so good, you can also use it as an altimeter with ±1 meter accuracy.

The BMP280 is the next-generation of sensors from Bosch and is the upgrade to the BMP085/BMP180/ - with a low altitude noise of 0.25m and the same fast conversion time. It has the same specifications but can use either I2C or SPI. For simple easy wiring, go with I2C.

1.5.1 The driver code

```
# Author: Paul Cunnane 2016
#
# This module borrows heavily from the Adafruit BME280 Python library
# and the Adafruit GPIO/I2C library. Original copyright notices are reproduced
# below.
#
# Those libraries were written for the Raspberry Pi. This modification is
# intended for the MicroPython and WiPy boards.
#
#
# Copyright (c) 2014 Adafruit Industries
# Author: Tony DiCola
#
# Based on the BMP280 driver with BME280 changes provided by
# David J Taylor, Edinburgh (www.satsignal.eu)
#
# Based on Adafruit_I2C.py created by Kevin Townsend.
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.

import time
import math


# BME280 default address.
BME280_I2CADDR = 0x76

# Operating Modes
BME280_OSAMPLE_1 = 1
BME280_OSAMPLE_2 = 2
BME280_OSAMPLE_4 = 3
BME280_OSAMPLE_8 = 4
BME280_OSAMPLE_16 = 5

# BME280 Registers

BME280_REGISTER_DIG_T1 = 0x88  # Trimming parameter registers
BME280_REGISTER_DIG_T2 = 0x8A
```

```
BME280_REGISTER_DIG_T3 = 0x8C

BME280_REGISTER_DIG_P1 = 0x8E
BME280_REGISTER_DIG_P2 = 0x90
BME280_REGISTER_DIG_P3 = 0x92
BME280_REGISTER_DIG_P4 = 0x94
BME280_REGISTER_DIG_P5 = 0x96
BME280_REGISTER_DIG_P6 = 0x98
BME280_REGISTER_DIG_P7 = 0x9A
BME280_REGISTER_DIG_P8 = 0x9C
BME280_REGISTER_DIG_P9 = 0x9E

BME280_REGISTER_DIG_H1 = 0xA1
BME280_REGISTER_DIG_H2 = 0xE1
BME280_REGISTER_DIG_H3 = 0xE3
BME280_REGISTER_DIG_H4 = 0xE4
BME280_REGISTER_DIG_H5 = 0xE5
BME280_REGISTER_DIG_H6 = 0xE6
BME280_REGISTER_DIG_H7 = 0xE7

BME280_REGISTER_CHIPID = 0xD0
BME280_REGISTER_VERSION = 0xD1
BME280_REGISTER_SOFTRESET = 0xE0

BME280_REGISTER_CONTROL_HUM = 0xF2
BME280_REGISTER_CONTROL = 0xF4
BME280_REGISTER_CONFIG = 0xF5
BME280_REGISTER_PRESSURE_DATA = 0xF7
BME280_REGISTER_TEMP_DATA = 0xFA
BME280_REGISTER_HUMIDITY_DATA = 0xFD


class Device:
    """Class for communicating with an I2C device.

    Allows reading and writing 8-bit, 16-bit, and byte array values to
    registers on the device."""

    def __init__(self, address, i2c):
        """Create an instance of the I2C device at the specified address using
        the specified I2C interface object."""
        self._address = address
        self._i2c = i2c

    def writeRaw8(self, value):
        """Write an 8-bit value on the bus (without register)."""
        value = value & 0xFF
        self._i2c.writeto(self._address, value.to_bytes(1, "little"))

    def write8(self, register, value):
        """Write an 8-bit value to the specified register."""
        value = value & 0xFF
        self._i2c.writeto_mem(self._address, register, value.to_bytes(1, "little"))

    def write16(self, register, value):
        """Write a 16-bit value to the specified register."""
        value = value & 0xFFFF
        self.i2c.writeto_mem(self._address, register, value.to_bytes(1, "little"))

    def readRaw8(self):
        """Read an 8-bit value on the bus (without register)."""
        return int.from_bytes(self._i2c.readfrom(self._address, 1), "little") & 0xFF

    def readU8(self, register):
        """Read an unsigned byte from the specified register."""
        return int.from_bytes(
            self._i2c.readfrom_mem(self._address, register, 1), "little") & 0xFF

    def readS8(self, register):
        """Read a signed byte from the specified register."""
        result = self.readU8(register)
        if result > 127:
            result -= 256
        return result

    def readU16(self, register, little_endian=True):
```

```
        """Read an unsigned 16-bit value from the specified register, with the
        specified endianness (default little endian, or least significant byte
        first)."""
        result = int.from_bytes(
            self._i2c.readfrom_mem(self._address, register, 2), "little") & 0xFFFF
        if not little_endian:
            result = ((result << 8) & 0xFF00) + (result >> 8)
        return result

    def readS16(self, register, little_endian=True):
        """Read a signed 16-bit value from the specified register, with the
        specified endianness (default little endian, or least significant byte
        first)."""
        result = self.readU16(register, little_endian)
        if result > 32767:
            result -= 65536
        return result

    def readU16LE(self, register):
        """Read an unsigned 16-bit value from the specified register, in little
        endian byte order."""
        return self.readU16(register, little_endian=True)

    def readU16BE(self, register):
        """Read an unsigned 16-bit value from the specified register, in big
        endian byte order."""
        return self.readU16(register, little_endian=False)

    def readS16LE(self, register):
        """Read a signed 16-bit value from the specified register, in little
        endian byte order."""
        return self.readS16(register, little_endian=True)

    def readS16BE(self, register):
        """Read a signed 16-bit value from the specified register, in big
        endian byte order."""
        return self.readS16(register, little_endian=False)


class BME280:
    def __init__(self, mode=BME280_OSAMPLE_16, address=BME280_I2CADDR, i2c=None,
                 **kwargs):
        # Check that mode is valid.
        if mode not in [BME280_OSAMPLE_1, BME280_OSAMPLE_2, BME280_OSAMPLE_4,
                        BME280_OSAMPLE_8, BME280_OSAMPLE_16]:
            raise ValueError(
                'Unexpected mode value {0}. Set mode to one of '
                'BME280_ULTRALOWPOWER, BME280_STANDARD, BME280_HIGHRES, or '
                'BME280_ULTRAHIGHRES'.format(mode))
        self._mode = mode
        # Create I2C device.
        if i2c is None:
            raise ValueError('An I2C object is required.')
        self._device = Device(address, i2c)
        # Load calibration values.
        self._load_calibration()
        self._device.write8(BME280_REGISTER_CONTROL, 0x3F)
        self.t_fine = 0

    def _load_calibration(self):

        self.dig_T1 = self._device.readU16LE(BME280_REGISTER_DIG_T1)
        self.dig_T2 = self._device.readS16LE(BME280_REGISTER_DIG_T2)
        self.dig_T3 = self._device.readS16LE(BME280_REGISTER_DIG_T3)

        self.dig_P1 = self._device.readU16LE(BME280_REGISTER_DIG_P1)
        self.dig_P2 = self._device.readS16LE(BME280_REGISTER_DIG_P2)
        self.dig_P3 = self._device.readS16LE(BME280_REGISTER_DIG_P3)
        self.dig_P4 = self._device.readS16LE(BME280_REGISTER_DIG_P4)
        self.dig_P5 = self._device.readS16LE(BME280_REGISTER_DIG_P5)
        self.dig_P6 = self._device.readS16LE(BME280_REGISTER_DIG_P6)
        self.dig_P7 = self._device.readS16LE(BME280_REGISTER_DIG_P7)
        self.dig_P8 = self._device.readS16LE(BME280_REGISTER_DIG_P8)
        self.dig_P9 = self._device.readS16LE(BME280_REGISTER_DIG_P9)

        self.dig_H1 = self._device.readU8(BME280_REGISTER_DIG_H1)
```

```
        self.dig_H2 = self._device.readS16LE(BME280_REGISTER_DIG_H2)
        self.dig_H3 = self._device.readU8(BME280_REGISTER_DIG_H3)
        self.dig_H6 = self._device.readS8(BME280_REGISTER_DIG_H7)

        h4 = self._device.readS8(BME280_REGISTER_DIG_H4)
        h4 = (h4 << 24) >> 20
        self.dig_H4 = h4 | (self._device.readU8(BME280_REGISTER_DIG_H5) & 0x0F)

        h5 = self._device.readS8(BME280_REGISTER_DIG_H6)
        h5 = (h5 << 24) >> 20
        self.dig_H5 = h5 | (
            self._device.readU8(BME280_REGISTER_DIG_H5) >> 4 & 0x0F)

    def read_raw_temp(self):
        """Reads the raw (uncompensated) temperature from the sensor."""
        meas = self._mode
        self._device.write8(BME280_REGISTER_CONTROL_HUM, meas)
        meas = self._mode << 5 | self._mode << 2 | 1
        self._device.write8(BME280_REGISTER_CONTROL, meas)
        sleep_time = 1250 + 2300 * (1 << self._mode)
        sleep_time = sleep_time + 2300 * (1 << self._mode) + 575
        sleep_time = sleep_time + 2300 * (1 << self._mode) + 575
        time.sleep_us(sleep_time)  # Wait the required time
        msb = self._device.readU8(BME280_REGISTER_TEMP_DATA)
        lsb = self._device.readU8(BME280_REGISTER_TEMP_DATA + 1)
        xlsb = self._device.readU8(BME280_REGISTER_TEMP_DATA + 2)
        raw = ((msb << 16) | (lsb << 8) | xlsb) >> 4
        return raw

    def read_raw_pressure(self):
        """Reads the raw (uncompensated) pressure level from the sensor."""
        """Assumes that the temperature has already been read """
        """i.e. that enough delay has been provided"""
        msb = self._device.readU8(BME280_REGISTER_PRESSURE_DATA)
        lsb = self._device.readU8(BME280_REGISTER_PRESSURE_DATA + 1)
        xlsb = self._device.readU8(BME280_REGISTER_PRESSURE_DATA + 2)
        raw = ((msb << 16) | (lsb << 8) | xlsb) >> 4
        return raw

    def read_raw_humidity(self):
        """Assumes that the temperature has already been read """
        """i.e. that enough delay has been provided"""
        msb = self._device.readU8(BME280_REGISTER_HUMIDITY_DATA)
        lsb = self._device.readU8(BME280_REGISTER_HUMIDITY_DATA + 1)
        raw = (msb << 8) | lsb
        return raw

    def read_temperature(self):
        """Get the compensated temperature in 0.01 of a degree celsius."""
        adc = self.read_raw_temp()
        var1 = ((adc >> 3) - (self.dig_T1 << 1)) * (self.dig_T2 >> 11)
        var2 = ((
            (((adc >> 4) - self.dig_T1) * ((adc >> 4) - self.dig_T1)) >> 12) *
            self.dig_T3) >> 14
        self.t_fine = var1 + var2
        return (self.t_fine * 5 + 128) >> 8

    def read_pressure(self):
        """Gets the compensated pressure in Pascals."""
        adc = self.read_raw_pressure()
        var1 = self.t_fine - 128000
        var2 = var1 * var1 * self.dig_P6
        var2 = var2 + ((var1 * self.dig_P5) << 17)
        var2 = var2 + (self.dig_P4 << 35)
        var1 = (((var1 * var1 * self.dig_P3) >> 8) +
                ((var1 * self.dig_P2) >> 12))
        var1 = (((1 << 47) + var1) * self.dig_P1) >> 33
        if var1 == 0:
            return 0
        p = 1048576 - adc
        p = (((p << 31) - var2) * 3125) // var1
        var1 = (self.dig_P9 * (p >> 13) * (p >> 13)) >> 25
        var2 = (self.dig_P8 * p) >> 19
        return ((p + var1 + var2) >> 8) + (self.dig_P7 << 4)

    def read_humidity(self):
```

```
        adc = self.read_raw_humidity()
        # print 'Raw humidity = {0:d}'.format (adc)
        h = self.t_fine - 76800
        h = (((((adc << 14) - (self.dig_H4 << 20) - (self.dig_H5 * h)) +
             16384) >> 15) * (((((((h * self.dig_H6) >> 10) * (((h *
                                 self.dig_H3) >> 11) + 32768)) >> 10) + 2097152) *
                                 self.dig_H2 + 8192) >> 14))
        h = h - (((((h >> 15) * (h >> 15)) >> 7) * self.dig_H1) >> 4)
        h = 0 if h < 0 else h
        h = 419430400 if h > 419430400 else h
        return h >> 12

    def temperature(self):
        "Return the temperature in degrees."
        return self.read_temperature() / 100


    def pressure(self):
        "Return the pressure in hPa."
        return self.read_pressure() / 25600

    def humidity(self):
        "Return the humidity in percent."
        return self.read_humidity() / 1024
```

1.5.2 The main program

```
import machine
import bme280

sda=machine.Pin(12)
scl=machine.Pin(14)
i2c=machine.I2C(0,sda=sda, scl=scl, freq=400000) # I2C , pins, 400kHz max
bme = bme280.BME280(i2c=i2c)

print(bme.temperature(), bme.pressure(), bme.humidity())
```

To do:

Write the combined codes to display:

- temperature
- humidity
- luminosity and
- pressur

Lab 2 – WiFi and ThingSpeak IoT server

In this lab we are going to study the essential use of the WiFi modem and we will try to send and receive the sensor data to the external IoT server called ThingSpeak.

2.1 Scanning the WiFi networks

```
print("Scanning for WiFi networks, please wait...")
print("")

import network
sta_if = network.WLAN(network.STA_IF)
sta_if.active(True)
```

```
authmodes = ['Open', 'WEP', 'WPA-PSK' 'WPA2-PSK4', 'WPA/WPA2-PSK']
for (ssid, bssid, channel, RSSI, authmode, hidden) in sta_if.scan():
  print("* {:s}".format(ssid))
  print("   - Channel: {}".format(channel))
  print("   - RSSI: {}".format(RSSI))
  print("   - BSSID: {:02x}:{:02x}:{:02x}:{:02x}:{:02x}:{:02x}".format(*bssid))
  print()
```

The result displayed in the terminal window

```
%Run -c $EDITOR_CONTENT
Scanning for WiFi networks, please wait...

* DIRECT-G8M2070 Series
   - Channel: 11
   - RSSI: -57
   - BSSID: 86:25:19:53:78:8f

* PIX-LINK-2.4G
   - Channel: 11
   - RSSI: -70
   - BSSID: 90:91:64:50:7e:04

* Livebox-08B0
   - Channel: 11
   - RSSI: -72
   - BSSID: 78:81:02:31:08:b0

* VAIO-MQ35AL
   - Channel: 5
   - RSSI: -76
- BSSID: d0:ae:ec:bf:3a:82

..

* FreeWifi_secure
   - Channel: 11
   - RSSI: -91
- BSSID: 14:0c:76:b2:49:71
```

2.2 Connecting to WiFi AP

The following example shows hot to connect your board to the available AP.

```
def do_connect():
    import network
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    if not wlan.isconnected():
        print('connecting to network...')
        wlan.connect('Livebox-08B0', 'G79ji6dtEptVTPWmZP')
        while not wlan.isconnected():
            pass
    print('network config:', wlan.ifconfig())

import machine, ssd1306
from machine import Pin, SoftI2C
import esp32
```

```
print(machine.freq())
#print((esp32.raw_temperature()-32)/1.8)  # temp in Celsius
do_connect()
#i2c = machine.I2C(scl=machine.Pin(14), sda=machine.Pin(12))
i2c = SoftI2C(scl=Pin(0), sda=Pin(7), freq=100000)
oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
oled.fill(0)
oled.text("SmartComputerLab", 0, 0)
oled.text("RISC-V mPython", 0, 16)
oled.text("with SoftI2C", 0, 32)
oled.text("WiFi connected", 0, 48)
oled.show()
```

2.3 Sending data to and Receiving from ThingSpeak server

2.3.1 `main` code

```
import network
import urequests
import json
import time
import esp
esp.osdebug(None)

import gc  # garbage collector

def do_connect():
    import network
    station = network.WLAN(network.STA_IF)
    station.active(True)
    if not station.isconnected():
        print('connecting to network...')
        station.connect('Livebox-08B0', 'G79ji6dtEptVTPWmZP')
        while not station.isconnected():
            pass
    print('network config:', station.ifconfig())

gc.collect()

api_write_key = 'YOX31M0EDKO0JATK'
api_read_key = '20E9AQVFW7Z6XXOM'

do_connect()
print('Connection successful')
print('writing to ThingSpeak')
urequests.get("https://api.thingspeak.com/update?api_key=YOX31M0EDKO0JATK&field1=32.8&field2=76.9")
time.sleep(15)
print('reading from ThingSpeak')
data = urequests.get("https://api.thingspeak.com/channels/1538804/feeds.json?
api_key=20E9AQVFW7Z6XXOM&results=4")

print(data.text)
buff=json.loads(data.text)
print()
print()
print(buff)
print(buff["channel"]["id"])
print(buff["channel"]["created_at"])
print(buff["channel"]["last_entry_id"])
leid=buff["channel"]["last_entry_id"]
print(buff["feeds"])
print(buff["feeds"][leid-1]["field1"])
time.sleep(5)
data.close()
```

2.3.2 `urequests` library

The following is the complete **`urequests.py`** library that contains several methods to **`GET`** and **`POST`** **URL requests**.

```
import usocket
```

```
class Response:
    def __init__(self, f):
        self.raw = f
        self.encoding = "utf-8"
        self._cached = None

    def close(self):
        if self.raw:
            self.raw.close()
            self.raw = None
        self._cached = None

    @property
    def content(self):
        if self._cached is None:
            try:
                self._cached = self.raw.read()
            finally:
                self.raw.close()
                self.raw = None
        return self._cached

    @property
    def text(self):
        return str(self.content, self.encoding)

    def json(self):
        import ujson
        return ujson.loads(self.content)

def request(method, url, data=None, json=None, headers={}, stream=None):
    try:
        proto, dummy, host, path = url.split("/", 3)
    except ValueError:
        proto, dummy, host = url.split("/", 2)
        path = ""
    if proto == "http:":
        port = 80
    elif proto == "https:":
        import ussl
        port = 443
    else:
        raise ValueError("Unsupported protocol: " + proto)
    if ":" in host:
        host, port = host.split(":", 1)
        port = int(port)
    ai = usocket.getaddrinfo(host, port, 0, usocket.SOCK_STREAM)
    ai = ai[0]
    s = usocket.socket(ai[0], ai[1], ai[2])
    try:
        s.connect(ai[-1])
        if proto == "https:":
            s = ussl.wrap_socket(s, server_hostname=host)
        s.write(b"%s /%s HTTP/1.0\r\n" % (method, path))
        if not "Host" in headers:
            s.write(b"Host: %s\r\n" % host)
        # Iterate over keys to avoid tuple alloc
        for k in headers:
            s.write(k)
            s.write(b": ")
            s.write(headers[k])
            s.write(b"\r\n")
        if json is not None:
            assert data is None
            import ujson

            data = ujson.dumps(json)
            s.write(b"Content-Type: application/json\r\n")
        if data:
            s.write(b"Content-Length: %d\r\n" % len(data))
        s.write(b"\r\n")
        if data:
            s.write(data)
        l = s.readline()
        # print(l)
```

```
        l = l.split(None, 2)
        status = int(l[1])
        reason = ""
        if len(l) > 2:
            reason = l[2].rstrip()
        while True:
            l = s.readline()
            if not l or l == b"\r\n":
                break
            # print(l)
            if l.startswith(b"Transfer-Encoding:"):
                if b"chunked" in l:
                    raise ValueError("Unsupported " + l)
            elif l.startswith(b"Location:") and not 200 <= status <= 299:
                raise NotImplementedError("Redirects not yet supported")
    except OSError:
        s.close()
        raise
    resp = Response(s)
    resp.status_code = status
    resp.reason = reason
    return resp

def head(url, **kw):
    return request("HEAD", url, **kw)

def get(url, **kw):
    return request("GET", url, **kw)

def post(url, **kw):
    return request("POST", url, **kw)

def put(url, **kw):
    return request("PUT", url, **kw)

def patch(url, **kw):
    return request("PATCH", url, **kw)

def delete(url, **kw):
    return request("DELETE", url, **kw)
```

To do:

Test the above codes
Modify the code to send the sensor (temperature/humidity) data and to receive the data from ThingSpeak server

Lab 3

LoRa – Long Range communication with SX1276/8 modems

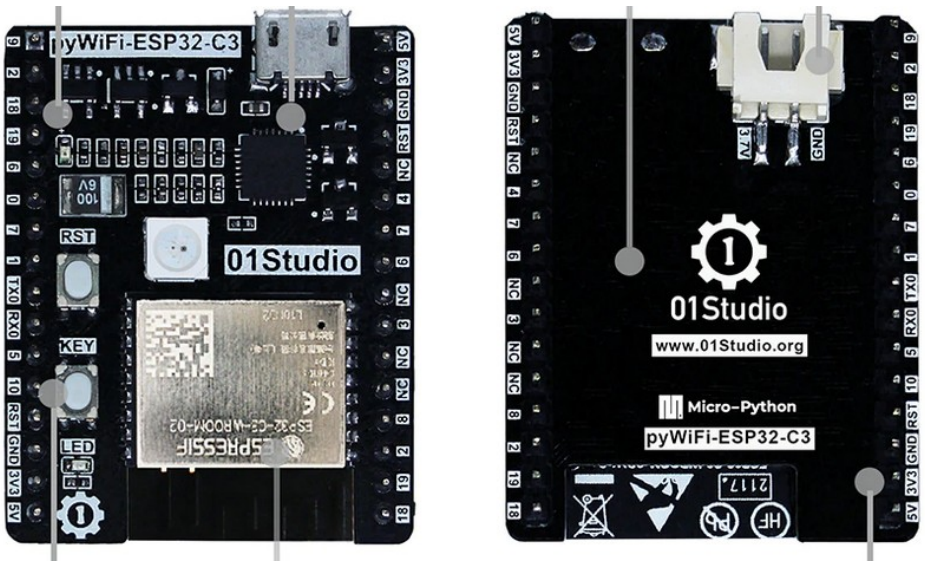
In this laboratory we are going to implement long range communication with LoRa modems (SX1276/78).

3.1 LoRa technology

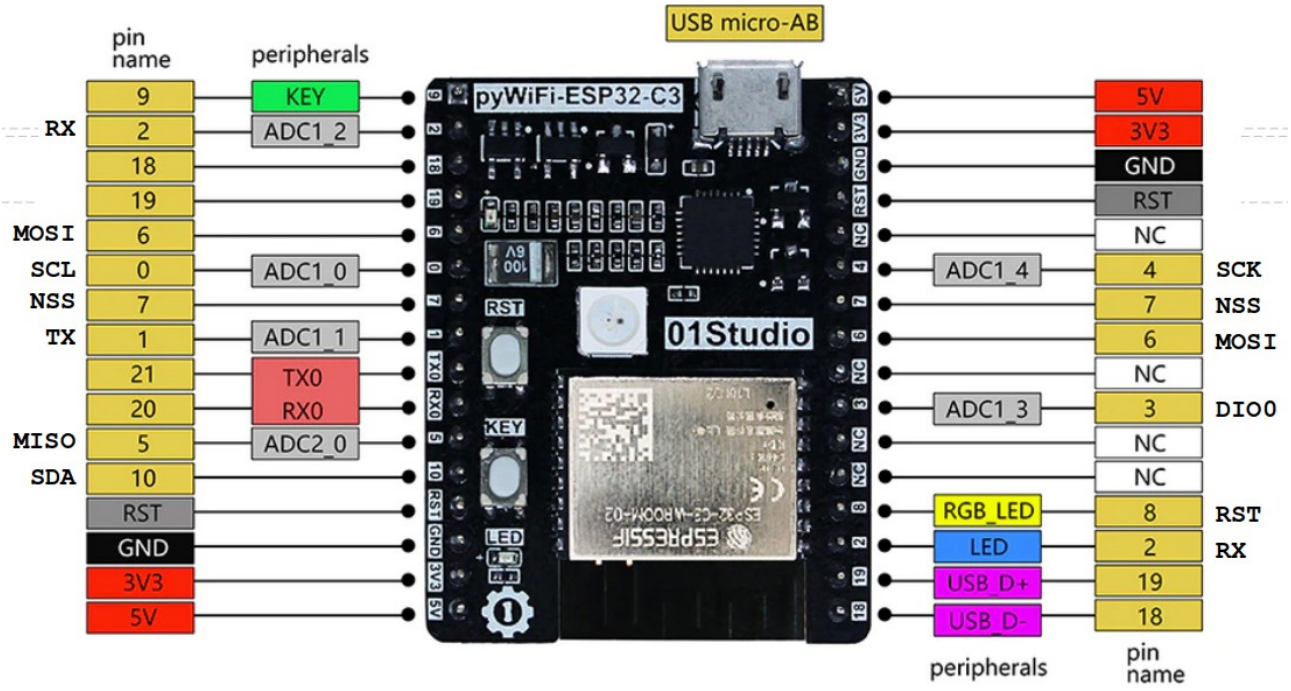
3.2 Simple LoRa sender and receiver nodes

MicroPython with ESP32-C3 RISC-V

Carte :



Code for SPI and LoRa – CLK-4, MISO-5, MOSI-6,CS-7, RST-8, INT (DIO0) – 2



Our SPI bus (SoftSPI):

SCK-4, MISO-5, MOSI-6, SS-7, RST-8, INT-3 (the last RST and INT (DIO0) are used to drive LoRa modem.

Our I2C bus (SoftI2C):

SCL-0, SDA – 10

Programming environment – SSD1306 (OLED) I2C

Main program:

main.py

```
import machine, ssd1306

i2c = machine.I2C(scl=machine.Pin(0), sda=machine.Pin(10))
oled = ssd1306.SSD1306_I2C(128, 64, i2c, 0x3c)
oled.fill(0)
oled.text("Hello World", 0, 0)
oled.show()
```

SSD1306 driver (we use only **SSD1306_I2C** class):

ssd1306.py

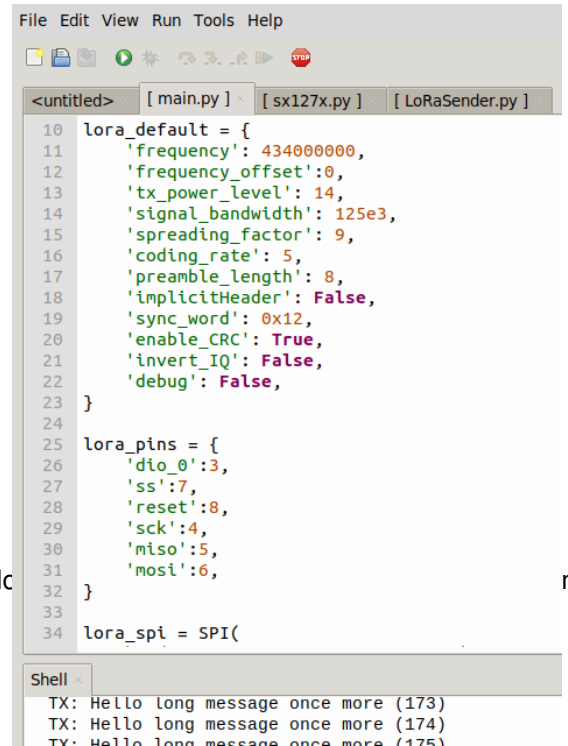
```
# MicroPython SSD1306 OLED driver, I2C and SPI interfaces

from micropython import const
import framebuf
```

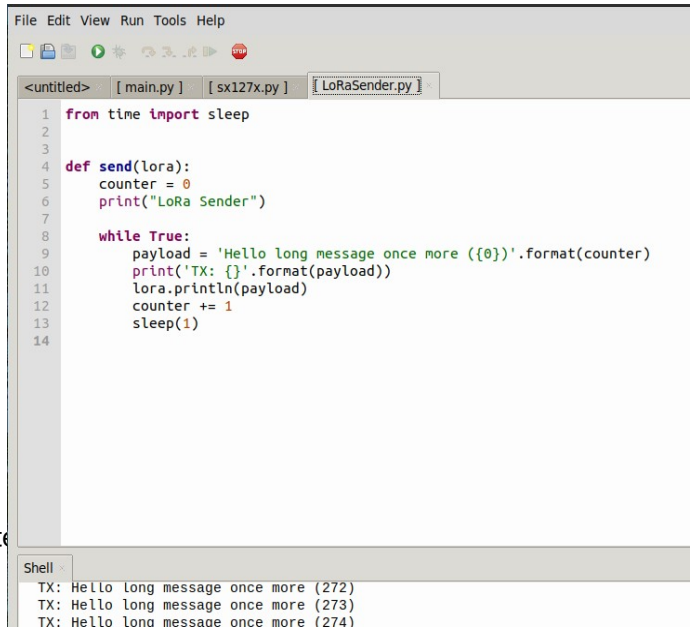
```
# register definitions
SET_CONTRAST = const(0x81)
SET_ENTIRE_ON = const(0xA4)
SET_NORM_INV = const(0xA6)
SET_DISP = const(0xAE)
SET_MEM_ADDR = const(0x20)
SET_COL_ADDR = const(0x21)
SET_PAGE_ADDR = const(0x22)
SET_DISP_START_LINE = const(0x40)
SET_SEG_REMAP = const(0xA0)
SET_MUX_RATIO = const(0xA8)
SET_IREF_SELECT = const(0xAD)
SET_COM_OUT_DIR = const(0xC0)
SET_DISP_OFFSET = const(0xD3)
SET_COM_PIN_CFG = const(0xDA)
SET_DISP_CLK_DIV = const(0xD5)
SET_PRECHARGE = const(0xD9)
SET_VCOM_DESEL = const(0xDB)
SET_CHARGE_PUMP = const(0x8D)

# Subclassing FrameBuffer provides support for graphics primitives
# http://docs.micropython.org/en/latest/pyboard/library/framebuf.html

class SSD1306(framebuf.FrameBuffer):
    def __init__(self, width, height, external_vcc):
        self.width = width
        self.height = height
        self.external_vcc = external_vcc
        self.pages = self.height // 8
        self.buffer = bytearray(self.pages * self.width)
        super().__init__(self.buffer, self.width, self.height,
framebuf.MONO_VLSB)
        self.init_display()

    def init_display(self):
        for cmd in (
            SET_DISP,  # display off
            # address setting
            SET_MEM_ADDR,
            0x00,  # horizontal
            # resolution and layout
            SET_DISP_START_LINE,  # start at line 0
            SET_SEG_REMAP | 0x01,  # column addr 127 mapped to SEG0
            SET_MUX_RATIO,
            self.height - 1,
            SET_COM_OUT_DIR | 0x08,  # scan from COM[N] to COM0
            SET_DISP_OFFSET,
            0x00,
            SET_COM_PIN_CFG,
            0x02 if self.width > 2 * self.height else 0x12,
            # timing and driving scheme
            SET_DISP_CLK_DIV,
            0x80,
            SET_PRECHARGE,
            0x22 if self.external_vcc else 0xF1,
            SET_VCOM_DESEL,
            0x30,  # 0.83*Vcc
            # display
            SET_CONTRAST,
            0xFF,  # maximum
            SET_ENTIRE_ON,  # output follows RAM contents
```

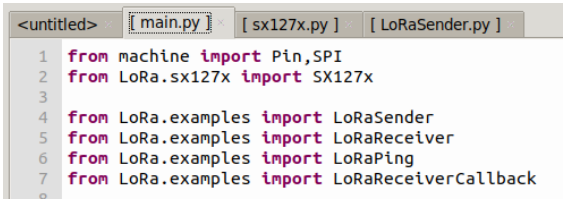
```
            SET_NORM_INV,  # not inverted
            SET_IREF_SELECT,
            0x30,  # enable internal IREF during display on
            # charge pump
            SET_CHARGE_PUMP,
            0x10 if self.external_vcc else 0x14,
            SET_DISP | 0x01,  # display on
        ):  # on
            self.write_cmd(cmd)
        self.fill(0)
        self.show()

    def poweroff(self):
        self.write_cmd(SET_DISP)

    def poweron(self):
        self.write_cmd(SET_DISP | 0x01)

    def contrast(self, contrast):
        self.write_cmd(SET_CONTRAST)
        self.write_cmd(contrast)

    def invert(self, invert):
        self.write_cmd(SET_NORM_INV | (invert & 1))

    def rotate(self, rotate):
        self.write_cmd(SET_COM_OUT_DIR | ((rotate & 1) << 3))
        self.write_cmd(SET_SEG_REMAP | (rotate & 1))

    def show(self):
        x0 = 0
        x1 = self.width - 1
        if self.width != 128:
            # narrow displays use centred columns
            col_offset = (128 - self.width) // 2
            x0 += col_offset
            x1 += col_offset
        self.write_cmd(SET_COL_ADDR)
        self.write_cmd(x0)
        self.write_cmd(x1)
        self.write_cmd(SET_PAGE_ADDR)
        self.write_cmd(0)
        self.write_cmd(self.pages - 1)
        self.write_data(self.buffer)


class SSD1306_I2C(SSD1306):
    def __init__(self, width, height, i2c, addr=0x3C, external_vcc=False):
        self.i2c = i2c
        self.addr = addr
        self.temp = bytearray(2)
        self.write_list = [b"\x40", None]  # Co=0, D/C#=1
        super().__init__(width, height, external_vcc)

    def write_cmd(self, cmd):
        self.temp[0] = 0x80  # Co=1, D/C#=0
        self.temp[1] = cmd
        self.i2c.writeto(self.addr, self.temp)

    def write_data(self, buf):
        self.write_list[1] = buf
```

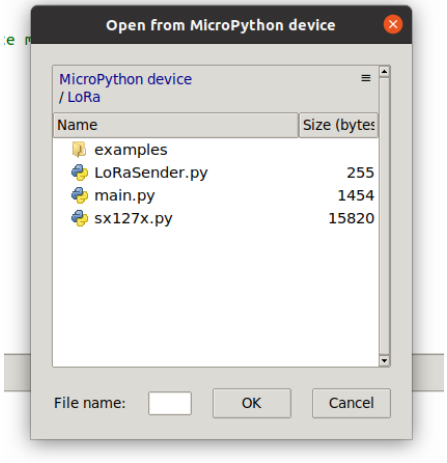
```
        self.i2c.writevto(self.addr, self.write_list)


class SSD1306_SPI(SSD1306):
    def __init__(self, width, height, spi, dc, res, cs, external_vcc=False):
        self.rate = 10 * 1024 * 1024
        dc.init(dc.OUT, value=0)
        res.init(res.OUT, value=0)
        cs.init(cs.OUT, value=1)
        self.spi = spi
        self.dc = dc
        self.res = res
        self.cs = cs
        import time

        self.res(1)
        time.sleep_ms(1)
        self.res(0)
        time.sleep_ms(10)
        self.res(1)
        super().__init__(width, height, external_vcc)

    def write_cmd(self, cmd):
        self.spi.init(baudrate=self.rate, polarity=0, phase=0)
        self.cs(1)
        self.dc(0)
        self.cs(0)
        self.spi.write(bytearray([cmd]))
        self.cs(1)

    def write_data(self, buf):
        self.spi.init(baudrate=self.rate, polarity=0, phase=0)
        self.cs(1)
        self.dc(1)
        self.cs(0)
        self.spi.write(buf)
        self.cs(1)
```

Programming environment - LoRa










```
from machine import Pin,SPI
from LoRa.sx127x import SX127x

from LoRa.examples import LoRaSender
from LoRa.examples import LoRaReceiver
from LoRa.examples import LoRaPing
from LoRa.examples import LoRaReceiverCallback


lora_default = {
    'frequency': 434000000,
    'frequency_offset':0,
    'tx_power_level': 14,
    'signal_bandwidth': 125e3,
    'spreading_factor': 9,
    'coding_rate': 5,
    'preamble_length': 8,
    'implicitHeader': False,
    'sync_word': 0x12,
    'enable_CRC': True,
    'invert_IQ': False,
    'debug': False,
}

lora_pins = {
    'dio_0':3,
    'ss':7,
    'reset':8,
    'sck':4,
    'miso':5,
    'mosi':6,
}

lora_spi = SPI(
    baudrate=10000000, polarity=0, phase=0,
    bits=8, firstbit=SPI.MSB,
    sck=Pin(lora_pins['sck'], Pin.OUT, Pin.PULL_DOWN),
    mosi=Pin(lora_pins['mosi'], Pin.OUT, Pin.PULL_UP),
    miso=Pin(lora_pins['miso'], Pin.IN, Pin.PULL_UP),
)

lora = SX127x(lora_spi, pins=lora_pins, parameters=lora_default)


type = 'sender'
# type = 'receiver'
# type = 'ping_master'
# type = 'ping_slave'
# type = 'receiver_callback'

if __name__ == '__main__':
    if type == 'sender':
        LoRaSender.send(lora)
```

```
    if type == 'receiver':
        LoRaReceiver.receive(lora)
    if type == 'ping_master':
        LoRaPing.ping(lora, master=True)
    if type == 'ping_slave':
        LoRaPing.ping(lora, master=False)
    if type == 'receiver_callback':
        LoRaReceiverCallback.receiveCallback(lora)
```

LoRa sender/receiver – `LoRaSender.py`, `LoRaReceiver.py`, `LoRaReceiverCallback.py`, and **`LoRaPing.py`:**

```
from time import sleep

def send(lora):
    counter = 0
    print("LoRa Sender")

    while True:
        payload = 'Hello ({0})'.format(counter)
        print('TX: {}'.format(payload))
        lora.println(payload)
        counter += 1
        sleep(5)
#####
def receive(lora):
    print("LoRa Receiver")

    while True:
        if lora.receivedPacket():
            try:
                payload = lora.readPayload().decode()
                rssi = lora.packetRssi()
                print("RX: {} | RSSI: {}".format(payload, rssi))
            except Exception as e:
                print(e)
#####
def receiveCallback(lora):
    print("LoRa Receiver Callback")
    lora.onReceive(onReceive)
    lora.receive()

def onReceive(lora, payload):
    try:
        payload = payload.decode()
        rssi = lora.packetRssi()
        print("RX: {} | RSSI: {}".format(payload, rssi))
    except Exception as e:
        print(e)

#####
from time import ticks_ms

def ping(lora, master):

    last_send_time = 0
    interval = 3000
    counter = 0

    lora.receive()
    while True:
        if master:
            if ticks_ms() - last_send_time > interval:
                last_send_time = ticks_ms()
                message = 'ping ({})'.format(counter)
                print('{} TX: {}'.format(ticks_ms(), message))
                lora.println(message)

                try:
                    payload = lora.listen(1000).decode()
                except AttributeError:
                    print('{} RX: fail'.format(ticks_ms()))
```

```
            else:
                print('{} RX: {}'.format(ticks_ms(), payload))

            print()
            counter += 1
    else:
        if lora.receivedPacket():
            payload = lora.readPayload().decode()
            print('{} RX: {}'.format(ticks_ms(), payload))
            lora.println(payload)
```

sx127x.py – SPI bus and modem configuration

```
from time import sleep, ticks_ms
from machine import SPI, Pin
from micropython import const
import gc


PA_OUTPUT_RFO_PIN = const(0)
PA_OUTPUT_PA_BOOST_PIN = const(1)

# registers
REG_FIFO = const(0x00)
REG_OP_MODE = const(0x01)
REG_FRF_MSB = const(0x06)
REG_FRF_MID = const(0x07)
REG_FRF_LSB = const(0x08)
REG_PA_CONFIG = const(0x09)
REG_LNA = const(0x0C)
REG_FIFO_ADDR_PTR = const(0x0D)

REG_FIFO_TX_BASE_ADDR = const(0x0E)
FifoTxBaseAddr = const(0x00)

REG_FIFO_RX_BASE_ADDR = const(0x0F)
FifoRxBaseAddr = const(0x00)
REG_FIFO_RX_CURRENT_ADDR = const(0x10)
REG_IRQ_FLAGS_MASK = const(0x11)
REG_IRQ_FLAGS = const(0x12)
REG_RX_NB_BYTES = const(0x13)
REG_PKT_RSSI_VALUE = const(0x1A)
REG_PKT_SNR_VALUE = const(0x19)
REG_MODEM_CONFIG_1 = const(0x1D)
REG_MODEM_CONFIG_2 = const(0x1E)
REG_PREAMBLE_MSB = const(0x20)
REG_PREAMBLE_LSB = const(0x21)
REG_PAYLOAD_LENGTH = const(0x22)
REG_FIFO_RX_BYTE_ADDR = const(0x25)
REG_MODEM_CONFIG_3 = const(0x26)
REG_RSSI_WIDEBAND = const(0x2C)
REG_DETECTION_OPTIMIZE = const(0x31)
REG_DETECTION_THRESHOLD = const(0x37)
REG_SYNC_WORD = const(0x39)
REG_DIO_MAPPING_1 = const(0x40)
REG_VERSION = const(0x42)

# invert IQ
REG_INVERTIQ = const(0x33)
RFLR_INVERTIQ_RX_MASK = const(0xBF)
RFLR_INVERTIQ_RX_OFF = const(0x00)
RFLR_INVERTIQ_RX_ON = const(0x40)
RFLR_INVERTIQ_TX_MASK = const(0xFE)
RFLR_INVERTIQ_TX_OFF = const(0x01)
RFLR_INVERTIQ_TX_ON = const(0x00)

REG_INVERTIQ2 = const(0x3B)
RFLR_INVERTIQ2_ON = const(0x19)
RFLR_INVERTIQ2_OFF = const(0x1D)

# modes
# bit 7: 1 => LoRa mode
```

```
MODE_LONG_RANGE_MODE = const(0x80)
MODE_SLEEP = const(0x00)
MODE_STDBY = const(0x01)
MODE_TX = const(0x03)
MODE_RX_CONTINUOUS = const(0x05)
MODE_RX_SINGLE = const(0x06)

# PA config
PA_BOOST = const(0x80)

# IRQ masks
IRQ_TX_DONE_MASK = const(0x08)
IRQ_PAYLOAD_CRC_ERROR_MASK = const(0x20)
IRQ_RX_DONE_MASK = const(0x40)
IRQ_RX_TIME_OUT_MASK = const(0x80)

# Buffer size
MAX_PKT_LENGTH = const(255)


class SX127x:

    default_parameters = {
        "frequency": 869525000,
        "frequency_offset": 0,
        "tx_power_level": 14,
        "signal_bandwidth": 125e3,
        "spreading_factor": 9,
        "coding_rate": 5,
        "preamble_length": 8,
        "implicitHeader": False,
        "sync_word": 0x12,
        "enable_CRC": True,
        "invert_IQ": False,
    }

    def __init__(self, spi, pins, parameters={}):
        self.spi = spi
        self.pins = pins
        self.parameters = parameters

        self.pin_ss = Pin(self.pins["ss"], Pin.OUT)

        self.lock = False
        self.implicit_header_mode = None

        self.parameters = SX127x.default_parameters
        if parameters:
            self.parameters.update(parameters)

        # check version
        version = None
        for i in range(5):
            version = self.readRegister(REG_VERSION)
            if version:
                break
        # debug output
        print("SX version: {}".format(version))

        # put in LoRa and sleep mode
        self.sleep()
        # config
        self.setFrequency(self.parameters["frequency"])
        self.setSignalBandwidth(self.parameters["signal_bandwidth"])
        # set LNA boost
        self.writeRegister(REG_LNA, self.readRegister(REG_LNA) | 0x03)
        # set auto AGC
        self.writeRegister(REG_MODEM_CONFIG_3, 0x04)
        self.setTxPower(self.parameters["tx_power_level"])
        self.implicitHeaderMode(self.parameters["implicitHeader"])
        self.setSpreadingFactor(self.parameters["spreading_factor"])
        self.setCodingRate(self.parameters["coding_rate"])
        self.setPreambleLength(self.parameters["preamble_length"])
        self.setSyncWord(self.parameters["sync_word"])
        self.enableCRC(self.parameters["enable_CRC"])
        self.invertIQ(self.parameters["invert_IQ"])
```

```
        # set LowDataRateOptimize flag if symbol time > 16ms (default disable on reset)
        # self.writeRegister(REG_MODEM_CONFIG_3, self.readRegister(REG_MODEM_CONFIG_3) & 0xF7)
        # default disable on reset
        bw = self.parameters["signal_bandwidth"]
        sf = self.parameters["spreading_factor"]
        if 1000 / bw / 2 ** sf > 16:
            self.writeRegister(
                REG_MODEM_CONFIG_3, self.readRegister(REG_MODEM_CONFIG_3) | 0x08
            )

        # set base addresses
        self.writeRegister(REG_FIFO_TX_BASE_ADDR, FifoTxBaseAddr)
        self.writeRegister(REG_FIFO_RX_BASE_ADDR, FifoRxBaseAddr)

        self.standby()

    def beginPacket(self, implicitHeaderMode=False):
        self.standby()
        self.implicitHeaderMode(implicitHeaderMode)

        # reset FIFO address and payload length
        self.writeRegister(REG_FIFO_ADDR_PTR, FifoTxBaseAddr)
        self.writeRegister(REG_PAYLOAD_LENGTH, 0)

    def endPacket(self):
        # put in TX mode
        self.writeRegister(REG_OP_MODE, MODE_LONG_RANGE_MODE | MODE_TX)
        # wait for TX done, standby automatically on TX_DONE
        while (self.readRegister(REG_IRQ_FLAGS) & IRQ_TX_DONE_MASK) == 0:
            pass
        # clear IRQ's
        self.writeRegister(REG_IRQ_FLAGS, IRQ_TX_DONE_MASK)

    def write(self, buffer):
        currentLength = self.readRegister(REG_PAYLOAD_LENGTH)
        size = len(buffer)

        # check size
        size = min(size, (MAX_PKT_LENGTH - FifoTxBaseAddr - currentLength))

        # write data
        for i in range(size):
            self.writeRegister(REG_FIFO, buffer[i])

        # update length
        self.writeRegister(REG_PAYLOAD_LENGTH, currentLength + size)
        return size

    def aquirelock(self, lock=False):
        self.lock = False

    def println(self, message, implicitHeader=False, repeat=1):
        # wait until RX_Done, lock and begin writing
        self.aquirelock(True)

        if isinstance(message, str):
            message = message.encode()

        self.beginPacket(implicitHeader)
        self.write(message)

        for i in range(repeat):
            self.endPacket()

        # unlock when done writing
        self.aquirelock(False)
        self.collectGarbage()

    def getIrqFlags(self):
        irqFlags = self.readRegister(REG_IRQ_FLAGS)
        self.writeRegister(REG_IRQ_FLAGS, irqFlags)
        return irqFlags

    def packetRssi(self, rfi="hf"):
        packet_rssi = self.readRegister(REG_PKT_RSSI_VALUE)
        return packet_rssi - (157 if rfi == "hf" else 164)
```

```
    def packetSnr(self):
        return (self.readRegister(REG_PKT_SNR_VALUE)) * 0.25

    def standby(self):
        self.writeRegister(REG_OP_MODE, MODE_LONG_RANGE_MODE | MODE_STDBY)

    def sleep(self):
        self.writeRegister(REG_OP_MODE, MODE_LONG_RANGE_MODE | MODE_SLEEP)

    def setTxPower(self, level, outputPin=PA_OUTPUT_PA_BOOST_PIN):
        self.parameters["tx_power_level"] = level
        if outputPin == PA_OUTPUT_RFO_PIN:
            # RFO
            level = min(max(level, 0), 14)
            self.writeRegister(REG_PA_CONFIG, 0x70 | level)
        else:
            # PA BOOST
            level = min(max(level, 2), 17)
            self.writeRegister(REG_PA_CONFIG, PA_BOOST | (level - 2))

    def setFrequency(self, frequency):
        # TODO min max limit
        frequency = int(frequency)
        self.parameters["frequency"] = frequency
        frequency += self.parameters["frequency_offset"]

        frf = (frequency << 19) // 32000000
        self.writeRegister(REG_FRF_MSB, (frf >> 16) & 0xFF)
        self.writeRegister(REG_FRF_MID, (frf >> 8) & 0xFF)
        self.writeRegister(REG_FRF_LSB, (frf >> 0) & 0xFF)

    def setSpreadingFactor(self, sf):
        sf = min(max(sf, 6), 12)
        self.writeRegister(REG_DETECTION_OPTIMIZE, 0xC5 if sf == 6 else 0xC3)
        self.writeRegister(REG_DETECTION_THRESHOLD, 0x0C if sf == 6 else 0x0A)
        self.writeRegister(
            REG_MODEM_CONFIG_2,
            (self.readRegister(REG_MODEM_CONFIG_2) & 0x0F) | ((sf << 4) & 0xF0),
        )

    def setSignalBandwidth(self, sbw):
        bins = (
            7.8e3,
            10.4e3,
            15.6e3,
            20.8e3,
            31.25e3,
            41.7e3,
            62.5e3,
            125e3,
            250e3,
        )
        bw = 9

        if sbw < 10:
            bw = sbw
        else:
            for i in range(len(bins)):
                if sbw <= bins[i]:
                    bw = i
                    break

        self.writeRegister(
            REG_MODEM_CONFIG_1,
            (self.readRegister(REG_MODEM_CONFIG_1) & 0x0F) | (bw << 4),
        )

    def setCodingRate(self, denominator):
        denominator = min(max(denominator, 5), 8)
        cr = denominator - 4
        self.writeRegister(
            REG_MODEM_CONFIG_1,
            (self.readRegister(REG_MODEM_CONFIG_1) & 0xF1) | (cr << 1),
        )
```

```
    def setPreambleLength(self, length):
        self.writeRegister(REG_PREAMBLE_MSB, (length >> 8) & 0xFF)
        self.writeRegister(REG_PREAMBLE_LSB, (length >> 0) & 0xFF)

    def enableCRC(self, enable_CRC=False):
        modem_config_2 = self.readRegister(REG_MODEM_CONFIG_2)
        config = modem_config_2 | 0x04 if enable_CRC else modem_config_2 & 0xFB
        self.writeRegister(REG_MODEM_CONFIG_2, config)

    def invertIQ(self, invertIQ):
        self.parameters["invertIQ"] = invertIQ
        if invertIQ:
            self.writeRegister(
                REG_INVERTIQ,
                (
                    (
                        self.readRegister(REG_INVERTIQ)
                        & RFLR_INVERTIQ_TX_MASK
                        & RFLR_INVERTIQ_RX_MASK
                    )
                    | RFLR_INVERTIQ_RX_ON
                    | RFLR_INVERTIQ_TX_ON
                ),
            )
            self.writeRegister(REG_INVERTIQ2, RFLR_INVERTIQ2_ON)
        else:
            self.writeRegister(
                REG_INVERTIQ,
                (
                    (
                        self.readRegister(REG_INVERTIQ)
                        & RFLR_INVERTIQ_TX_MASK
                        & RFLR_INVERTIQ_RX_MASK
                    )
                    | RFLR_INVERTIQ_RX_OFF
                    | RFLR_INVERTIQ_TX_OFF
                ),
            )
            self.writeRegister(REG_INVERTIQ2, RFLR_INVERTIQ2_OFF)

    def setSyncWord(self, sw):
        self.writeRegister(REG_SYNC_WORD, sw)

    def setChannel(self, parameters):
        self.standby()
        for key in parameters:
            if key == "frequency":
                self.setFrequency(parameters[key])
                continue
            if key == "invert_IQ":
                self.invertIQ(parameters[key])
                continue
            if key == "tx_power_level":
                self.setTxPower(parameters[key])
                continue

    def dumpRegisters(self):
        # TODO end=''
        for i in range(128):
            print("0x{:02X}: {:02X}".format(i, self.readRegister(i)), end="")
            if (i + 1) % 4 == 0:
                print()
            else:
                print(" | ", end="")

    def implicitHeaderMode(self, implicitHeaderMode=False):
        if (
            self.implicit_header_mode != implicitHeaderMode
        ):  # set value only if different.
            self.implicit_header_mode = implicitHeaderMode
            modem_config_1 = self.readRegister(REG_MODEM_CONFIG_1)
            config = (
                modem_config_1 | 0x01
                if implicitHeaderMode
                else modem_config_1 & 0xFE
            )
```

```
        self.writeRegister(REG_MODEM_CONFIG_1, config)

    def receive(self, size=0):
        self.implicitHeaderMode(size > 0)
        if size > 0:
            self.writeRegister(REG_PAYLOAD_LENGTH, size & 0xFF)

        # The last packet always starts at FIFO_RX_CURRENT_ADDR
        # no need to reset FIFO_ADDR_PTR
        self.writeRegister(
            REG_OP_MODE, MODE_LONG_RANGE_MODE | MODE_RX_CONTINUOUS
        )

    def listen(self, time=1000):
        time = min(max(time, 0), 10000)
        self.receive()

        start = ticks_ms()
        while True:
            if self.receivedPacket():
                return self.readPayload()
            if ticks_ms() - start > time:
                return None

    def onReceive(self, callback):
        self.onReceive = callback

        if "dio_0" in self.pins:
            self.pin_rx_done = Pin(self.pins["dio_0"], Pin.IN)

        if self.pin_rx_done:
            if callback:
                self.writeRegister(REG_DIO_MAPPING_1, 0x00)
                self.pin_rx_done.irq(
                    trigger=Pin.IRQ_RISING, handler=self.handleOnReceive
                )
            else:
                pass
                # TODO detach irq

    def handleOnReceive(self, event_source):
        # lock until TX_Done
        self.aquirelock(True)
        irqFlags = self.getIrqFlags()
        # RX_DONE only, irqFlags should be 0x40
        if irqFlags & IRQ_RX_DONE_MASK == IRQ_RX_DONE_MASK:
            # automatically standby when RX_DONE
            if self.onReceive:
                payload = self.readPayload()
                self.onReceive(self, payload)

        elif self.readRegister(REG_OP_MODE) != (
            MODE_LONG_RANGE_MODE | MODE_RX_SINGLE
        ):
            # no packet received.
            # reset FIFO address / # enter single RX mode
            self.writeRegister(REG_FIFO_ADDR_PTR, FifoRxBaseAddr)
            self.writeRegister(
                REG_OP_MODE, MODE_LONG_RANGE_MODE | MODE_RX_SINGLE
            )

        self.aquirelock(False)  # unlock in any case.
        self.collectGarbage()
        return True

    def receivedPacket(self, size=0):
        irqFlags = self.getIrqFlags()
        self.implicitHeaderMode(size > 0)
        if size > 0:
            self.writeRegister(REG_PAYLOAD_LENGTH, size & 0xFF)

        # if (irqFlags & IRQ_RX_DONE_MASK) and \
        # (irqFlags & IRQ_RX_TIME_OUT_MASK == 0) and \
        # (irqFlags & IRQ_PAYLOAD_CRC_ERROR_MASK == 0):

        if (
```

```
            irqFlags == IRQ_RX_DONE_MASK
        ):  # RX_DONE only, irqFlags should be 0x40
            # automatically standby when RX_DONE
            return True

        elif self.readRegister(REG_OP_MODE) != (
            MODE_LONG_RANGE_MODE | MODE_RX_SINGLE
        ):
            # no packet received.
            # reset FIFO address / # enter single RX mode
            self.writeRegister(REG_FIFO_ADDR_PTR, FifoRxBaseAddr)
            self.writeRegister(
                REG_OP_MODE, MODE_LONG_RANGE_MODE | MODE_RX_SINGLE
            )

    def readPayload(self):
        # set FIFO address to current RX address
        # fifo_rx_current_addr = self.readRegister(REG_FIFO_RX_CURRENT_ADDR)
        self.writeRegister(
            REG_FIFO_ADDR_PTR, self.readRegister(REG_FIFO_RX_CURRENT_ADDR)
        )

        # read packet length
        packet_length = 0
        if self.implicit_header_mode:
            packet_length = self.readRegister(REG_PAYLOAD_LENGTH)
        else:
            packet_length = self.readRegister(REG_RX_NB_BYTES)

        payload = bytearray()
        for i in range(packet_length):
            payload.append(self.readRegister(REG_FIFO))

        self.collectGarbage()
        return bytes(payload)

    def readRegister(self, address, byteorder="big", signed=False):
        response = self.transfer(address & 0x7F)
        return int.from_bytes(response, byteorder)

    def writeRegister(self, address, value):
        self.transfer(address | 0x80, value)

    def transfer(self, address, value=0x00):
        response = bytearray(1)

        self.pin_ss.value(0)

        self.spi.write(bytes([address]))
        self.spi.write_readinto(bytes([value]), response)

        self.pin_ss.value(1)

        return response

    def collectGarbage(self):
        gc.collect()
        # print('[Mem aft - free: {}   allocated: {}]'.format(gc.mem_free(), gc.mem_alloc()))
```

UART bus (1):

```
import machine

# Setup the connection to your GPS here
# This example uses UART 31 with RX on pin 2 and TX on pin 1
# Baudrate is 9600bps, with the standard 8 bits, 1 stop bit, no parity
uart = machine.UART(1, rx=2, tx=1, baudrate=9600, bits=8, parity=None, stop=1) #, timeout=5000,
rxbuf=1024)

# Basic UART --> terminal printer, use to test your GPS module
while True:
    if uart.any():
        print(uart.read())
```