IoT Lab 0

IoT Architectures

0.1 Introduction

In this preparation, we introduce several core concepts and principles used to design **low-power IoT architectures**. We then present our **IoT platform**, including its **hardware, firmware, and software** components. This platform is used to implement IoT functionalities in line with the principles presented.
First, let's look at the overall **communication and operational space** involving IoT **terminals (end devices)**, **routers**, **gateways**, and **servers**.

In an IoT architecture, these network components work together to enable communication, data processing, and system control:

Terminals (IoT Devices / End Nodes)

Terminals collect sensor data and/or perform actions (actuation). Some devices also run lightweight local processing (for example, filtering, thresholding, or simple analytics) before transmitting data. They typically send data to an IoT router, an IoT gateway, or directly to the cloud using connectivity such as Wi-Fi or cellular networks.

Routers

Routers forward packets between networks (for example, from a local area network to a wide area network).
They generally rely on standard **IP networking** and routing mechanisms.

Gateways (IoT Gateways)

Gateways act as a bridge between IoT devices and Internet infrastructure. They often translate data from IoT-oriented protocols into standard IP-based protocols. In addition, gateways may **filter, aggregate, and compress** data before forwarding it to cloud services, reducing bandwidth usage and improving efficiency.

Servers (IoT Cloud Servers)

Servers store, process, and manage IoT data. In our case, we use lightweight **MQTT servers (brokers)** as well as full IoT cloud platforms such as **ThingSpeak** for visualization, analytics, and application integration.

Key considerations

The main design constraints in IoT networks include:

- **Low power consumption and low latency**
- **Scalability** (supporting many devices and data flows)
- **Security** (confidentiality, integrity, authentication)
-

A particularly important requirement is **terminal identification and addressing**, which ensures that each device can be uniquely recognized, securely authenticated, and correctly routed within the system.

0.2 Global IoT space , IoT Sockets and data “streams”

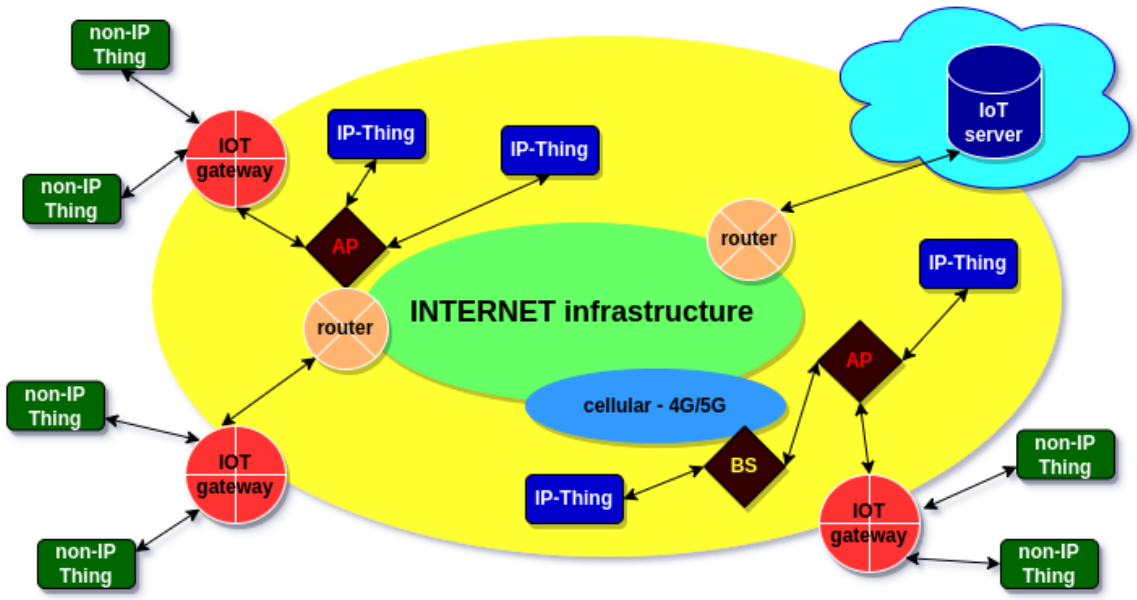

Fig.0.1 Global IoT space

The **global IoT space model** is associated to the concept of **IoT Socket**.

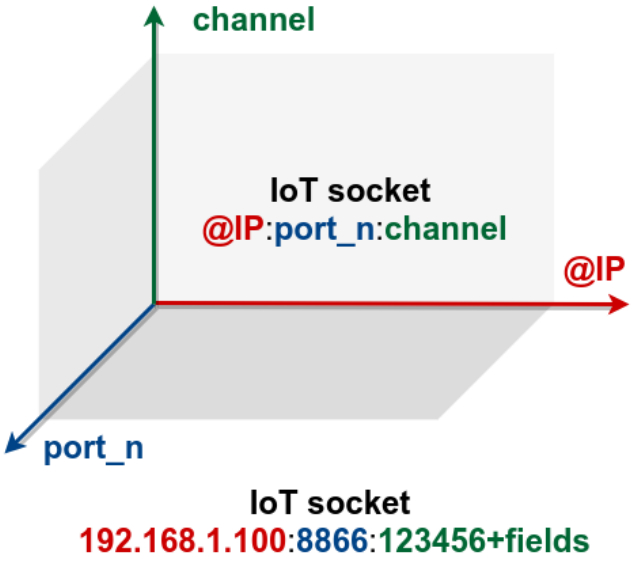

Fig.0.2 IoT Socket and Terminal/Gateway identifier

In our approach, an **IoT socket** is defined as a triple composed of an **IP address**, a **service port**, and a **channel number**. These elements correspond respectively to the addressing identifiers at:

- **Layer 3** (IP address),
- **Layer 4** (transport port),
- **Layer 5** (application channel).

The **channel** itself is structured into several fields, where each field carries a single **sensor (or actuator) data stream**. The channel number uniquely identifies the **terminal or gateway**, and in the case of gateways, it may also act as a **control channel identifier**.

IoT Architecture Overview

The proposed IoT architecture includes several types of nodes:

- **Direct terminals**
- **Remote terminals**
- **Close terminals**
- **Gateways**
- **IoT servers**

Direct terminals are connected directly to the Internet infrastructure using Wi-Fi or cellular (4G/5G) radio links and operate using the **IP protocol stack**.

Remote terminals and **close terminals** communicate through **gateways** (for example, LoRa–WiFi or WiFiMAC–WiFi gateways). These terminals do **not** use IP directly.

Identification and Communication Model

Remote and close terminals are identified **only by their channel number** on the corresponding IoT server. Gateways, on the other hand:

- Know the **IP address** and **port number** of the IoT server,
- Do **not** know the server's data channel numbers,
- May use an additional **control channel** on the server to receive configuration and control parameters.

Data Streams and Compression

Each IoT channel can carry and optionally store up to **eight independent data streams**.
Each stream corresponds to a specific **sensor or actuator** associated with a terminal node.

The proposed **IoT data compression protocol** operates independently on each data stream.
Its goal is to **minimize the number of data packets** transmitted over communication links such as Wi-Fi, WiFiMAC, or LoRa.

This operational mode can be described as an **ALAP protocol** — *As Little As Possible* — emphasizing minimal communication overhead and reduced power consumption.

The **control** of this mechanism (protocol) is done via **three types of parameters**:

- **`delta`** – the value of the delta parameter indicates the minimum distance between the last sent and the current sensor value. Note that the delta value my indicate the absolute difference (ex. 0.1 °C or relative difference , ex. 0.1%)
- **`cycle`** – indicates the time period between two consecutive sensor reads and eventual packet transmission
- **`t_high,t_low`** – indicate the high and low threshols

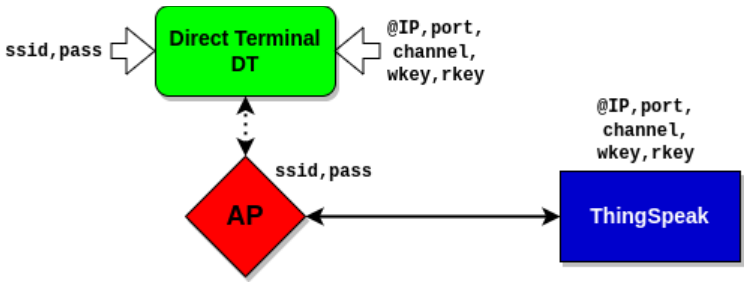

Fig.0.3 Simple IoT Architecture with **Direct Terminal** and its **parameters**

In the figure above, we illustrate a simple **IoT architecture** featuring a **direct terminal**.
To transmit a data packet, the direct terminal must know:

- The **SSID** and **password credentials** required to establish a communication link with the access point (AP),
- The complete **IoT socket**, which includes the **IP address**, **port number**, and **channel number**, along with the associated **write key**.

The next figure presents the essential parameters required to establish communication links and transfer sensor data from a **remote terminal** to the **IoT server**. In this case, the data packets (payloads) transmitted by the remote terminal are protected using **128-bit AES encryption**.
Access to the IoT channel itself is secured through **write and read keys** (16 bytes), ensuring controlled and secure data transmission and retrieval.

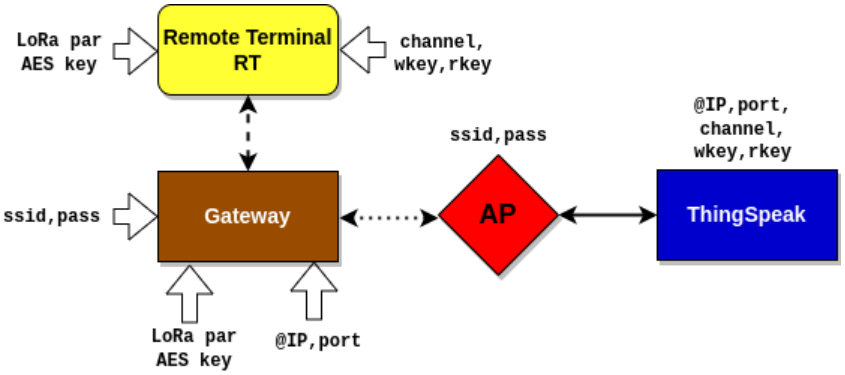

Fig.0.4 IoT Architecture: **Remote** or **Close Terminal** and **Gateway** parameters

For example, to transmit a data packet, a **remote terminal** only needs to know:

- The associated **channel number**, which also serves as the terminal's unique identifier (address),
- The **write key** for the channel,
- The **AES key** used to encrypt the packet payload.

Importantly, remote terminals do **not** know the **IP address** or **port number** of the corresponding IoT server. These elements of the IoT socket are known **only to the gateway**.
In addition, when the gateway uses a **Wi-Fi access point**, it must know the **SSID** and **password** of that access point in order to establish Internet connectivity. The gateway must also know the **AES key** in order to decrypt incoming **LoRa packets** received from remote terminals.

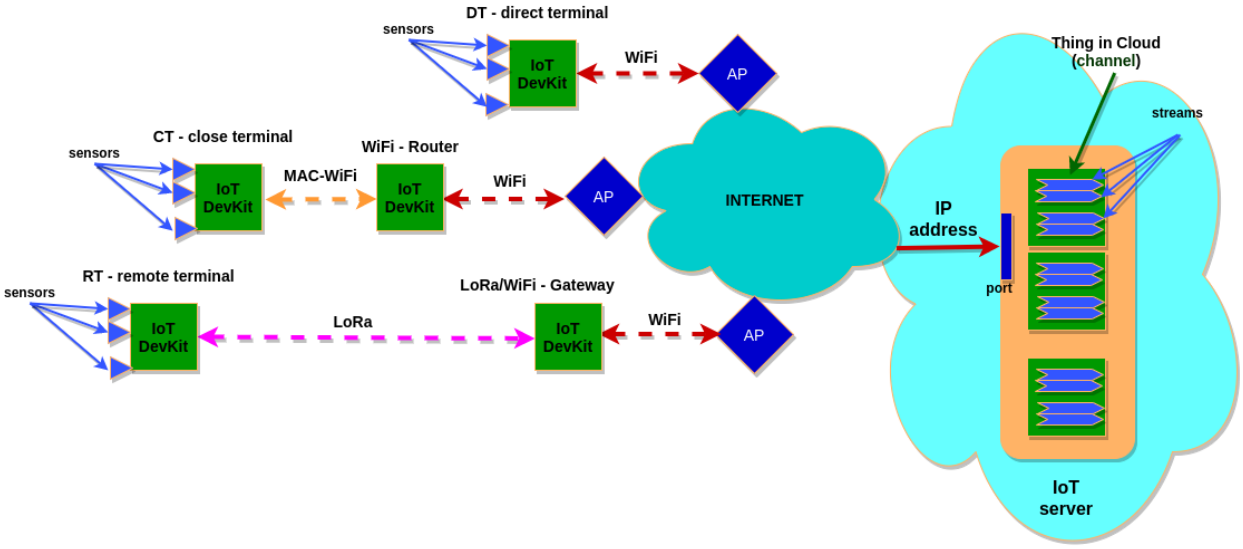

Fig.0.5 IoT Architecture: **Direct**, **Close** and **Remote Terminals** and **Gateway** to the IoT **server**

0.3 Low and Very Low Power consumption IoT components

Low power consumption is a fundamental requirement in the design of IoT architectures that rely on **autonomous terminal nodes**, including both **direct** and **remote terminals**.

At the heart of low-power IoT devices is the **IoT system-on-chip (SoC)** and its ability to operate in **ultra-low-power modes**, such as **deep sleep**. In these modes, the device can remain operational while consuming as little as **10–20 µA** of current.

Building on this capability, it is possible to design IoT devices with an **average current consumption** of less than **1 mA** (low power), or even below **100 µA** (very low power), depending on the duty cycle, communication technology, and application requirements.

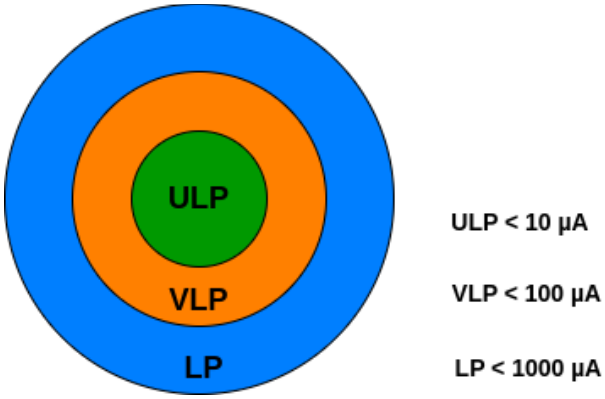

Fig.0.6 Levels of power (current) consumption: **ULP**-ultra low, **VLP** – very low, and **LP** – low power consumption

As an example, consider a device with an **ultra-low-power (ULP)** current consumption of **10 µA** while operating in **deep-sleep mode** (low-power stage) for **100 s**.
Assume that the device then enters an **active period** (high-power stage) with a current consumption of **40 mA** lasting **0.5 s**.

Together, the **low-power** and **high-power** stages form a complete **operational cycle**.

The question is: **what is the average current consumption over this cycle?**

To answer this, we begin by calculating the **total electrical charge** consumed during the cycle:

`low_power` charge + `high_power` charge= 10µA*100s + 40 000µA*0.5s = 1000µC+20000µC= 21mC

The average current is:

average_current = charge/time = 21mC/100.5s = 0.21mA = 210µA

To do:

Calculate the same for **`low_power` stage** duration of **600s**.

0.4 Memory hierarchy for data, parameters and meta-parameters

One of the essential features for designing **low-power IoT protocols**, and in particular **adaptive low-power IoT protocols**, is the availability of multiple **types and levels of memory**.

These typically include **main SRAM**, **low-power SRAM**, and **internal or external EEPROM** memory units.

The following figure illustrates the **memory hierarchy** exploited by low-power protocols to optimize energy consumption, data retention, and system responsiveness.

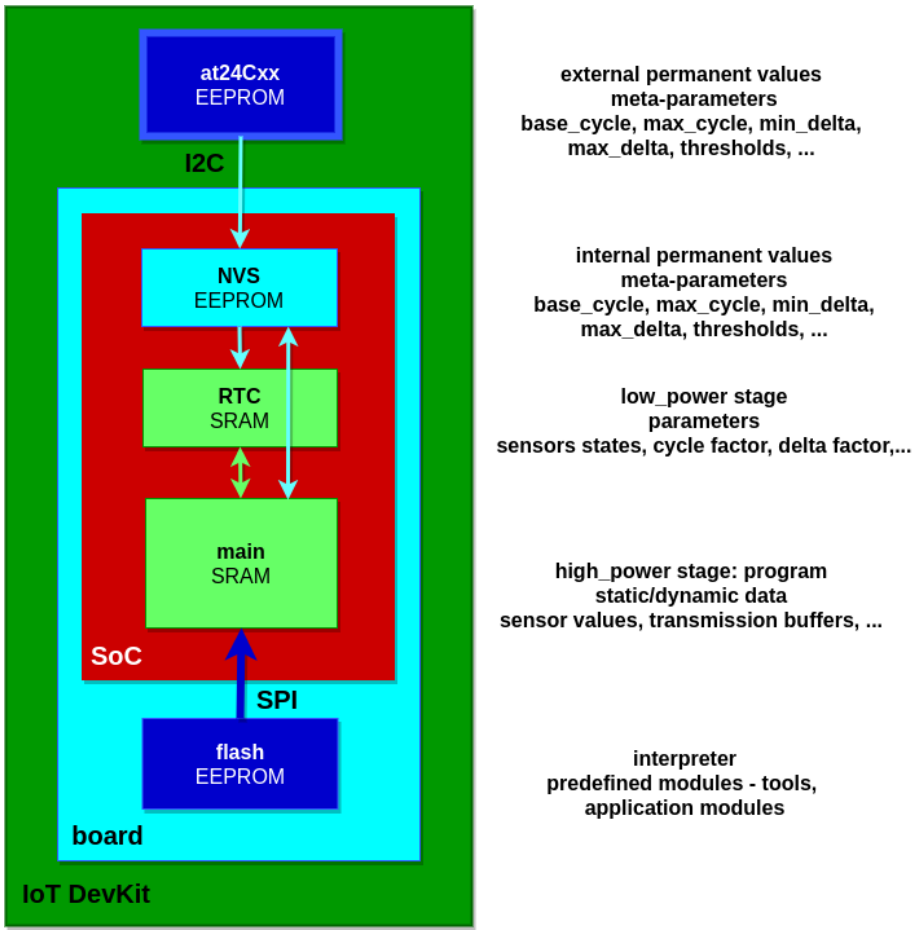

Fig.0.7 Memory types and hierarchy used in IoT SoC, board and DevKit.

The above memory hierarchy operates in high_power and low_power stages. In the

0.5 Terminals - Operational modes

The primary goal of **IoT architectures** is to transmit sensor data from **terminal nodes** to their corresponding **IoT servers (channels)**. There are many ways to capture physical phenomena (sensing) and transmit the resulting data over communication links. These approaches depend on the **type of sensor** and its specific **operational characteristics**.
Terminal nodes typically operate in **cycles**. The cycle duration (or frequency) may be fixed in the terminal's program code, or it can be dynamically modified based on new parameters provided by the **gateway**.
In some cases, the cycles are **asynchronous**. This occurs when an external event (such as an interrupt or trigger) initiates a new operational cycle.

Often, the captured sensor values undergo a **pre-processing phase** before being transmitted (or possibly discarded) by the terminal. This pre-processing typically involves:

- Comparing current sensor values with previously transmitted values,
- Evaluating differences against predefined **thresholds** associated with each sensing parameter.

The **sensing**, **pre-processing**, **data transmission**, and **data reception** activities are performed during phases that correspond to the **high-power consumption stage (H)**.
To significantly reduce overall energy consumption, the system makes use of **deep-sleep modes**, which introduce a **low-power stage (L)**. For example, the average current in the high-power stage may reach **150 mA**, whereas in the low-power stage it can be reduced to as little as **20 µA**.

It is important to note that when the high-power stage does **not** include packet transmission or reception, the average current consumption can be significantly lower, around **20 mA**, or even **2.5 mA** when using **ultra-low-power RISC-V cores**.

In this context, it becomes clear that reducing power consumption primarily relies on **extending the operational cycle duration** (i.e., lowering the activity frequency) and maximizing the time spent in the **low-power stage (L)**.

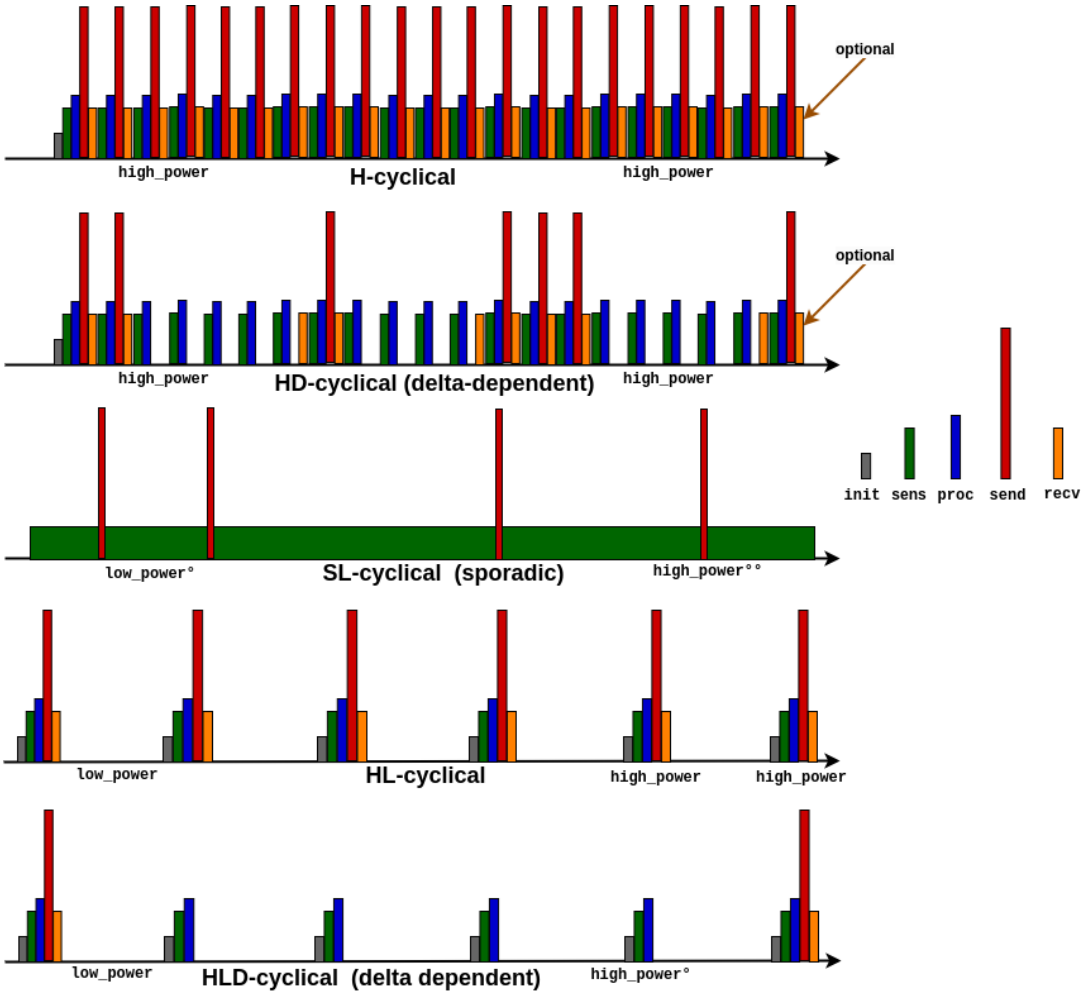

Fig.0.8 Basic operational modes (cycles) including **`high_power` stages** (**H**) with up to 5 phases and **`low_power` stages** (**L**). **Note: SL cyclical-sporadic** mode may be driven by an additional **ULP RISC-V processor** operating with its own execution cycle and calculating delta and threshold values to be used to wake up the main processor. This feature adds the capacity of **very low power pre-processing of sensor data**.

0.6 Cyclical (adaptive) mode with parameters and meta-parameters

The following figure illustrates the **use of different memory types** during the consecutive phases of the **high-power stage**.
During the **initialization phase**, the interpreter and user program modules are loaded into **main SRAM** from **external EEPROM/flash memory** located outside the SoC. Next, the system reads the **meta-parameters** either from the external EEPROM module or, if unavailable, from the **internal non-volatile storage (NVS EEPROM)**. The **runtime parameters** are retrieved from **RTC SRAM**.
In the subsequent **sensing phase**, sensor data are acquired. During the **processing phase**, these data are processed using both the meta-parameters and the current runtime parameters. Any newly computed parameters are then stored back into **RTC memory** for persistence across sleep cycles.

Depending on the application logic, the system may then proceed to the **transmission phase**:

- For a **direct terminal**, data are transmitted directly to the IoT server via a **Wi-Fi connection**.
- For a **close terminal**, data are sent to a **Wi-Fi router** using **Wi-Fi MAC-layer frames**.
- For a **remote terminal**, data packets are transmitted over a **LoRa channel** to a **LoRa–WiFi gateway**. In this case, the data packets are protected using **AES encryption**, typically accelerated by hardware cryptographic units available in the SoC.
-

After transmission, the system may enter a **reception phase**, during which it waits for acknowledgment (ACK) packets sent by the gateway. Two types of ACK packets can be distinguished:

- **Simple ACK packets**, confirming successful reception,
- **Control ACK packets**, which may carry updated **meta-parameter values**.
-

All ACK packets are also encrypted to ensure secure communication.

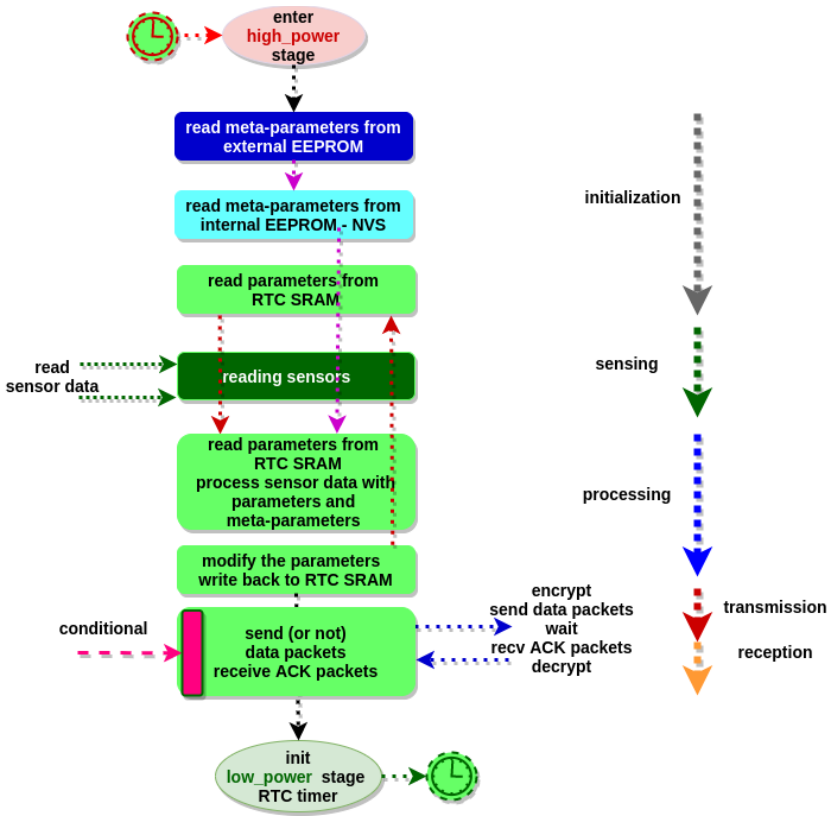

Fig.0.9 Cycle phases and the use of different memory types

0.7 From SoC to an IoT Platform

The essential components of **IoT systems** are typically built around a **SoC (System on Chip)** or a **SoM (System on Module)**. These components integrate processing units, communication modems, and I/O interfaces into a compact and energy-efficient solution.

A typical **ESP32 SoC** integrates one or two CPUs (such as **Xtensa LX6**, **LX7**, or **RISC-V RV32** cores). These SoCs also include wireless communication modems such as **Wi-Fi**, **Bluetooth**, and **Zigbee/Thread**, and expose a variety of serial interfaces for connecting external components.
As a result, **ESP32-based boards** are well suited for building **direct terminals** using Wi-Fi connectivity. Depending on the application design and duty cycle, they can achieve **low** or even **very low power consumption**.

Another category of components consists of **SoMs**, such as the **ASR560X** from **ASR Microelectronics**. These modules integrate **ultra-low-power CPUs** (e.g., **ARM Cortex-M0**) together with **LoRa modems** such as the **SX1262**. ASR560X-based solutions are particularly suitable for building **very low-power remote terminals** using **LoRa radio links**.

IoT Boards and Development Kits

IoT boards typically combine an IoT SoC or SoM with external components such as:

- EEPROM or flash memory,
- USB-to-TTY interfaces,
- Battery charging and power conversion circuits,
- Status LEDs,
- GPIO and expansion connectors.
-

We build our **IoT development kits (DevKits)** around boards such as:

- **Heltec ESP32-C3**,
- **ASR01**,
- **DFRobot ESP32-C6**.
-

These DevKits provide additional hardware elements, including batteries, solar panels, radio modems (e.g., **SX1276**/**SX1278**), GPIO headers, and serial bus connectors. They are designed to interface easily with a wide range of **sensors and actuators**.

Firmware, Software, and Platforms

An **IoT platform** completes the underlying hardware layers by providing firmware and software support. IoT applications can be developed using **C**/**C++** and/or **MicroPython** programming environments.
These environments are complemented by the required **drivers**, **protocol stacks**, and **software libraries**.
In this study, we use several categories of SoCs, boards, DevKits, and platforms:

1. **RISC-V–based platforms**
    - ESP32-C3 or ESP32-C6 SoCs
    - MicroPython programming
    - Thonny IDE
2. **LoRa-based ultra-low-power platforms**
    - ARM Cortex-M0 with SX1262 modem
    - CubeCell (CC) boards and DevKits
    - C/C++ programming using the Arduino IDE

In both cases, power consumption is analyzed using the **PPK II Power Profiler** from **Nordic Semiconductor**.

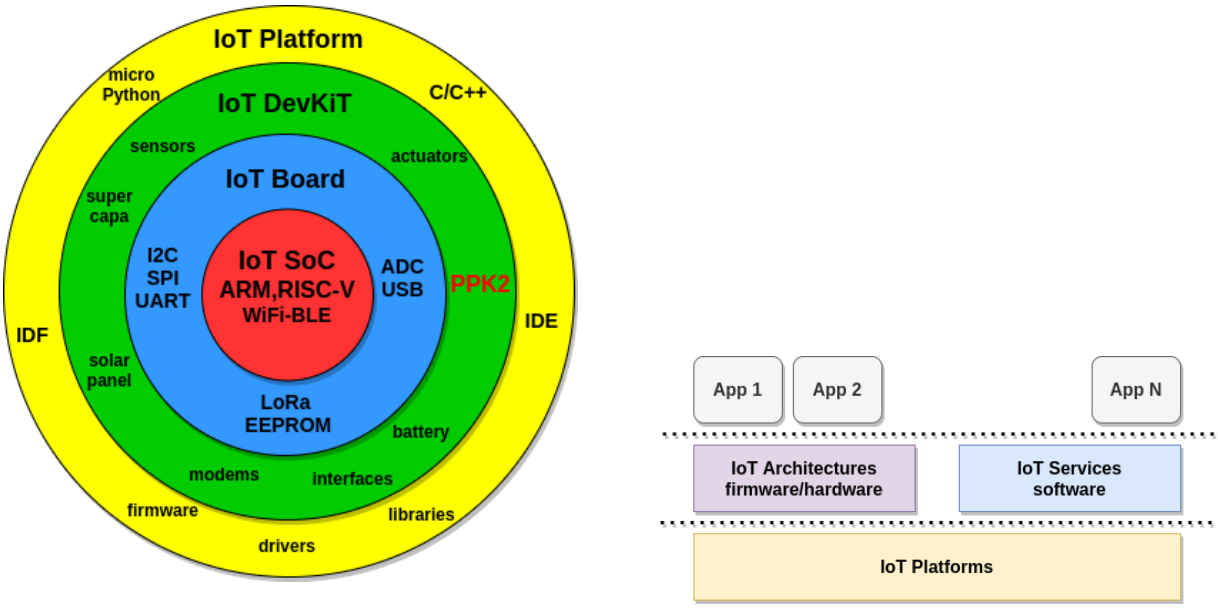

Fig 0.10 (a) IoT Platform(s) for the experimentation/development of **Low** and **Very Low Power** consumption IoT Architectures. **(b)** From platforms to Applications

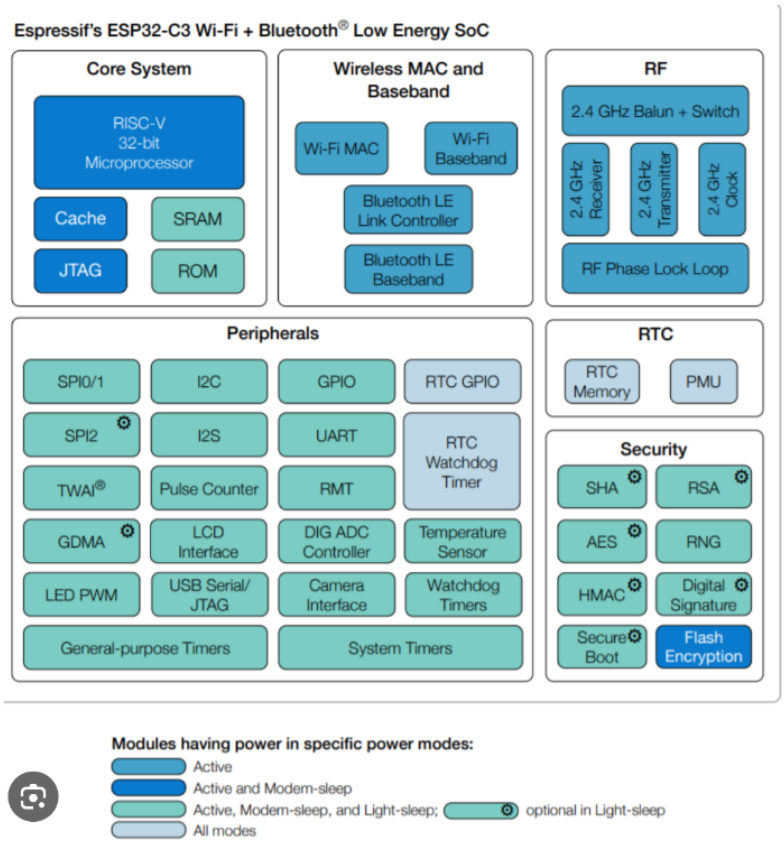

Fig 0.11 IoT ESP32C3 SoC - architecture block diagram

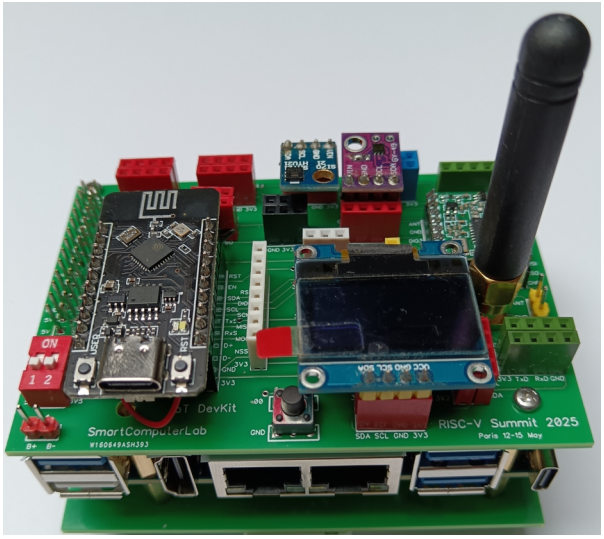

Fig 0.12 IoT **ESP32C3 SoC, HT Board,** and **DevKit** for the experimentation/development of **Low** and **Very Low Power** consumption IoT Architectures. The board is mounted on the top of **RISC-V SBC with SpacemiT – K1 SoC**.

The firmware/software "layers" are developed with **µPython** using **Thonny IDE.**
Let us ask ChatGPT about Thonny IDE and µPython. Note that **we do not need** to indicate the use of ESP32 familly of SOCs.

0.8 What is Thonny

Thonny IDE is a lightweight, beginner-friendly integrated development environment (IDE) designed for programming in **Python**. It is particularly well-suited for beginners due to its simplicity, clean interface, and easy-to-use features. In addition to regular Python programming, Thonny has excellent support for **MicroPython**, making it a popular choice for programming microcontrollers and IoT devices such as **ESP8266**, **ESP32**, and **Raspberry Pi Pico**.
Thonny makes it easy to develop, deploy, and test code on MicroPython-compatible devices. It has built-in tools to communicate with microcontrollers, upload scripts, and monitor real-time output from IoT devices.

Features of Thonny for MicroPython:

- **Simple Interface**: Easy-to-navigate interface for beginners.
- **MicroPython Support**: Built-in support for MicroPython, making it easy to flash firmware and upload code to devices like ESP32/ESP8266.
- **Interactive Python Shell**: Interactive REPL (Read-Eval-Print-Loop) for running Python commands directly on the microcontroller.
- **File System Management**: Allows you to manage files on the microcontroller's file system, such as uploading and downloading scripts.
- **Serial Monitor**: Communicates with the microcontroller via a serial connection, allowing you to debug and observe output from the device.
- **Code Debugging**: Provides simple debugging tools like stepping through code and viewing variable values.

How to Use Thonny IDE to Program IoT Devices with MicroPython
Below is a step-by-step guide on how to use Thonny IDE to program IoT devices with MicroPython, particularly focusing on ESP32/ESP8266 and Raspberry Pi Pico.

1. **Install Thonny IDE**
   - Download and install **Thonny** from the official website (https://thonny.org/).
     - Thonny is available for Windows, macOS, and Linux.
   - During installation, you can opt to install Python alongside Thonny if it's not already installed on your system.

2. **Set Up Thonny for MicroPython**

Once Thonny is installed, you need to configure it for MicroPython development.

- **Choose the MicroPython Interpreter**:
  - Open Thonny.
  - Go to the menu: **Tools** > **Options** > **Interpreter**.
  - From the "Interpreter" drop-down list, select the appropriate MicroPython interpreter for your board:
    - For ESP32 or ESP8266: Select **MicroPython (ESP32)** or **MicroPython (ESP8266)**.
    - For Raspberry Pi Pico: Select **MicroPython (Raspberry Pi Pico)**.
  - Choose the correct serial port (COM port on Windows or `/dev/ttyUSBx` on Linux/macOS) that your microcontroller is connected to.

3. **Install MicroPython Firmware on the IoT Device**

Before you can start programming, you need to flash MicroPython firmware onto your IoT device (if it's not already installed). The process differs slightly based on your microcontroller.

For ESP32:

- **Download the MicroPython firmware** for your board from the official MicroPython website.
  - ESP8266: Get the `.bin` file for ESP8266.
  - ESP32: Get the `.bin` file for ESP32.
- **Flash the Firmware**:
  - Connect your ESP32/ESP8266 board to your computer via USB.
  - In Thonny, go to **Tools** > **Options** > **Interpreter**.
  - Click the "Install or update firmware" button.
  - In the "Install MicroPython" dialog:
    - Select the **target port** (your board's COM port).
    - Select the **MicroPython variant** (ESP32 or ESP8266).
    - Click the **Install** button to flash the firmware onto the board.

4. **Writing and Uploading MicroPython Code**

Once the MicroPython firmware is installed on your IoT device, you can start writing code using Thonny.
Example: Blinking an LED on ESP32

- Open Thonny and write the following code in the editor:

```
from machine import Pin
from time import sleep
led = Pin(3, Pin.OUT)
# Blink the LED
while True:
    led.on()
    sleep(1)
    led.off()
    sleep(1)
```

- Click the **Run** button (green arrow) or press **F5**.
  - Thonny will automatically upload the code to the microcontroller and start running it.
  - You should see the LED blinking on the device.
  -

Serial Monitor and REPL:

- Thonny provides access to the **MicroPython REPL** in the terminal at the bottom of the window.

- You can enter commands directly and get instant feedback.
- Example: Type **`led.on()`** in the REPL, and the LED will turn on immediately.
-

5. **File Management**

Thonny allows you to manage the files on your microcontroller's file system.

- **Upload a Script**:
  - Write a script (e.g., **`main.py`**) and save it directly to the device by selecting **File** > **Save As** and choosing "MicroPython Device" as the location.
  - Files saved on the device can run automatically on startup if named **`main.py`** or **`boot.py`**.
- **List Files on the Device**:
  - Use the **Files** pane in Thonny (located on the right) to see the files stored on your MicroPython device.
  - You can add, remove, or download files from the microcontroller.

6. **Debugging and Monitoring Output**

Thonny has basic debugging features that are useful when developing IoT applications:

- You can use **breakpoints** and **step-through** execution to check how the code behaves on the microcontroller.
- The serial monitor (or REPL) at the bottom of the IDE can display output from `print()` statements in your code, making it useful for debugging.

Example of Signal Output:

```
from machine import Pin
from time import sleep
led = Pin(3, Pin.OUT)
while True:
    led.on()
    print("LED is ON")
    sleep(1)
    led.off()
    print("LED is OFF")
    sleep(1)
```

- The **`print()`** statements will show up in the Thonny terminal as the program runs, helping you verify the behavior.

7. **Advanced Features in Thonny**

- **Code Assistance**: Thonny provides code completion and hints to help write MicroPython code faster.
- **Graphical Output**: If your code generates data (such as sensor readings), Thonny has support for displaying graphical plots using `matplotlib` or similar libraries.
- **External Libraries**: You can upload external Python libraries (e.g., sensor drivers) to the MicroPython device using the **Files** pane.
-

Typical Use Cases for IoT with Thonny and MicroPython

1. **Sensor Data Collection**: Program an ESP32 or Raspberry Pi Pico to read data from sensors (e.g., temperature, humidity) and log or transmit it via Wi-Fi or Bluetooth.
2. **Home Automation**: Write MicroPython scripts to control IoT devices like smart lights, door locks, or relays based on sensor input or remote commands.
3. **Wireless Communication**: Use MicroPython to send sensor data to the cloud using protocols like MQTT or HTTP via Wi-Fi-enabled microcontrollers (e.g., ESP32).
4. **Prototyping IoT Projects**: Thonny's ease of use makes it ideal for quickly prototyping IoT devices and testing them in real-time.

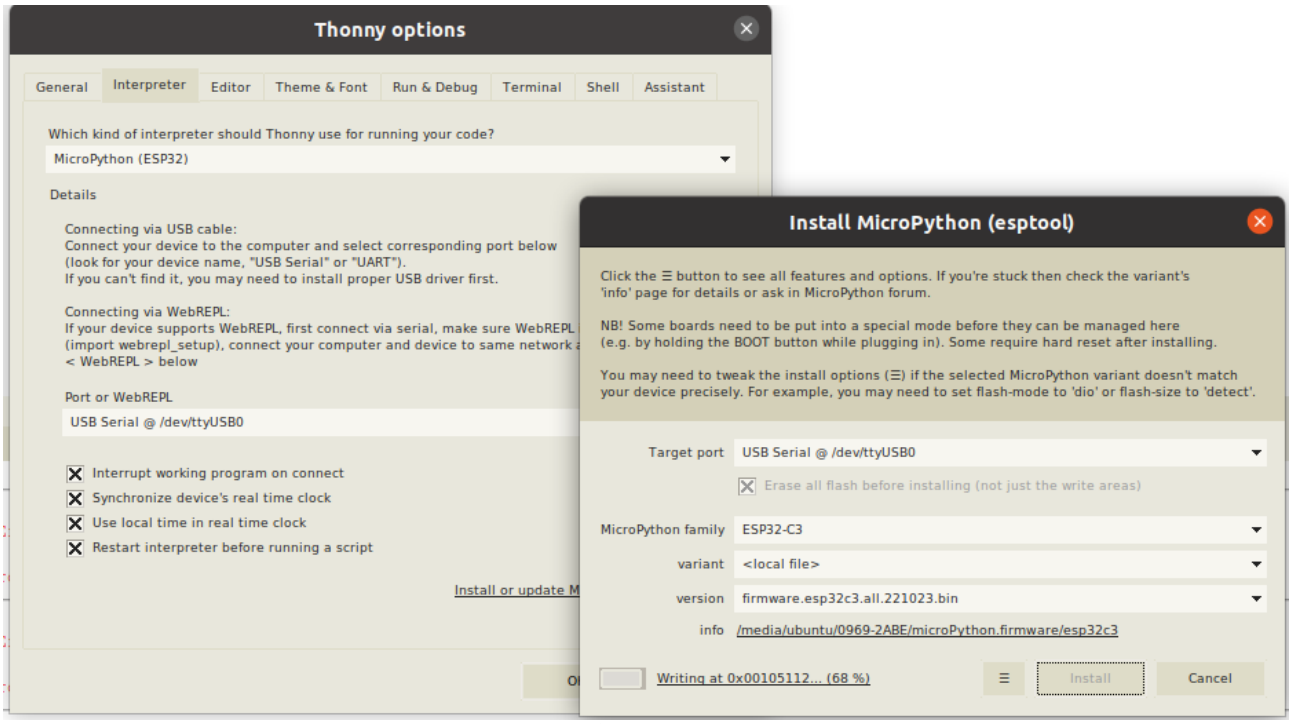

Fig.0.13a Thonny IDE – Options: MicroPython with ESP32 **Interpreter**, Serial port: **USB0**

Now if the **MicroPython** firmware is not flashed on the board go to: **Install or update**. Then choose the type of the board : **ESP32-C3** , local file, and look for the available firmware version, **generic** or prepared **for your board**. (click on **=** button) After these operations (flushing and loading) you should find the following windows with **MicroPython device** including only the **`boot.py`** module.

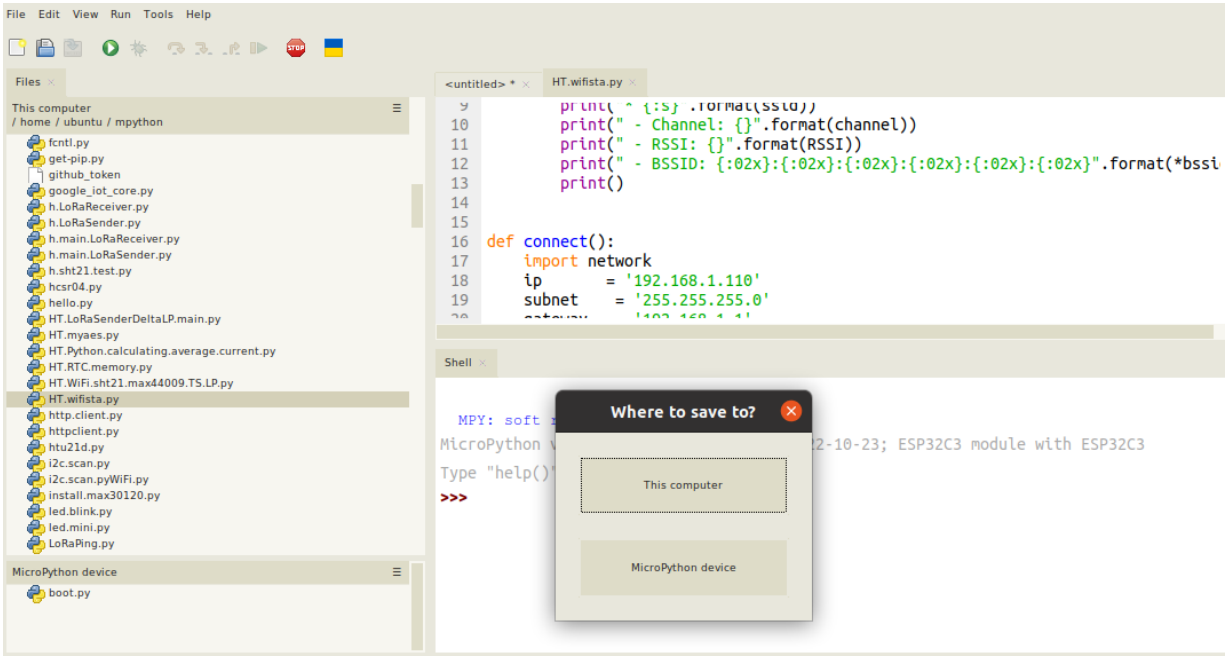

Fig.0.13b Thonny IDE with **4 windows**: **files** on your **PC**, **files** on your Micropython **Device** (HT board), **editor** window, **terminal** window with python prompt **>>>**. The edited files may be saved on your PC or on the Device (board).

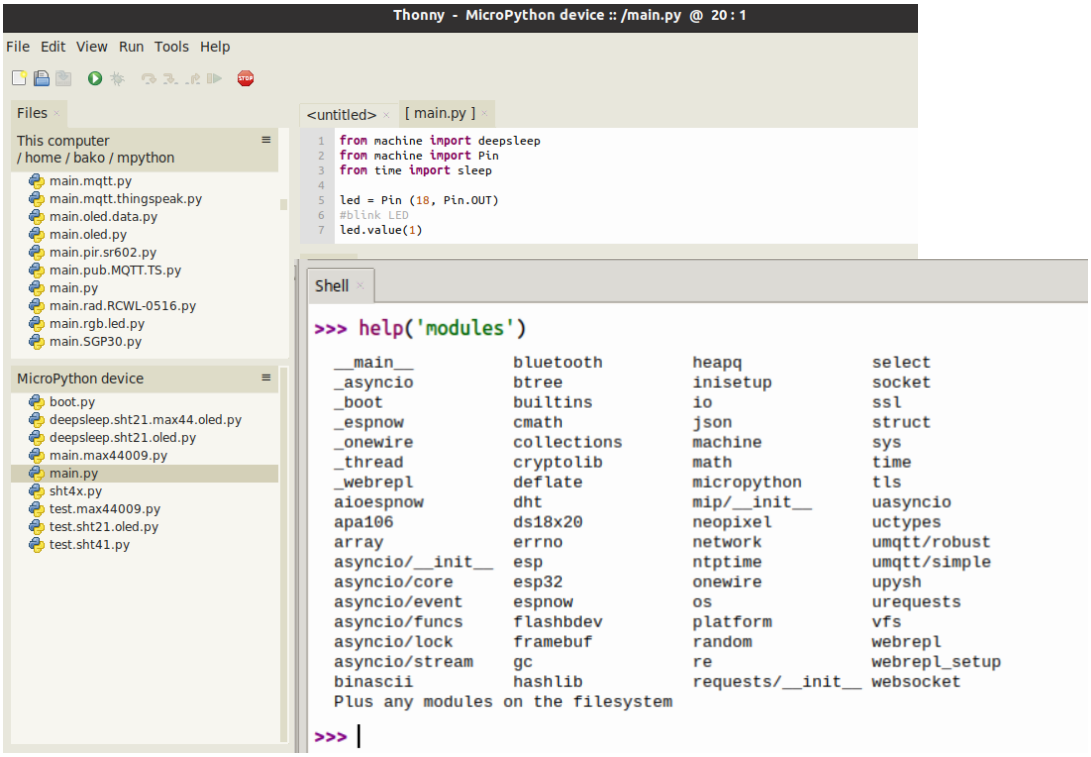

Fig.0.13c Thonny IDE with **4 windows**: **files** on your **PC**, **files** on your Micropython **Device** (HT board), **editor** window, **terminal** window with python prompt