Practical IoT Labs with Pomme-Pi Zero IoT DevKit

SmartComputerLab

In this introduction we are going to present the overall architecture of **IoT infrastructure** and **devices**.
The following section will cover the essential features of IoT technologies. The IoT platform with all supplementary components is provided by **SmartComputerLab** .
The pedagogical content including codes is available on `github.com/smartcomputerlab` server.

Table of Contents

Introduction

IoT Architecture may be seen as an addition to or an extension of the **Internet Infrastructure**. Basically the **Internet Infrastructure** is built with **communication links** and **routers**.
The **Internet Infrastructure** provides the **communication channels** between the Internet **terminals** such as users-clients and Internet servers. The traditional terminals at the client side are personal computers, laptops, smartphones,.. . The terminals on the server side are processing and data centers (“clouds”).

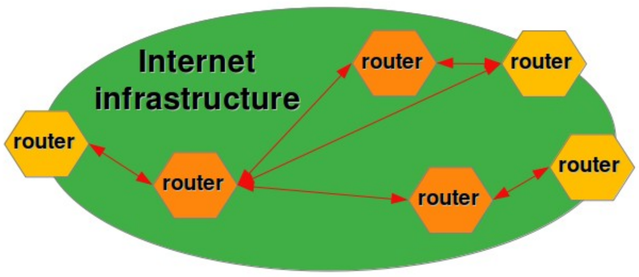


Fig 0.1 Internet Infrastructure

The Internet Infrastructure provides the routes to send and to receive the **Internet Packets**. Each Internet Packet contains the destination and source address plus the payload (content). The transmission of these packets is controlled by the **Internet Protocol** – **IP**.

The IoT devices are connected (associated) directly or indirectly to the Internet Infrastructure. The IoT devices connected directly to the Internet Infrastructure use IP protocol to carry the data.

Seen from the outside, there are two kinds of entry points to the Internet Infrastructure, **WiFi Access Points** – **AP** and cellular **base stations – BS**. The **IoT servers** in the **cloud** are connected to the Internet Infrastructure via wired/fiber links.

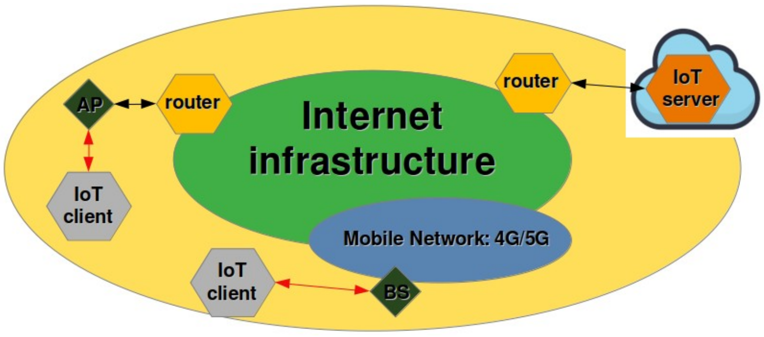


Fig 0.2 Entry points to Internet Infrastructure (**AP**, **BS**)

The **Things** may be authorized to communicate directly with the access points (**AP**, **BS**). In this case the data from/to sensors/actuators is sent in **IP protocol** packets. We can call these Things **IP-Things**. Another kind of **remote Things** , than we characterize as **NON-IP Things** may communicate with the Internet Infrastructure via the **IoT gateways**. These gateways are devices that combine the IP-based links with **Long Range** radio links such as **LoRa**. The data sent over the LoRa links is simply relayed and sent in IP packets over the links implemented with WiFi or cellular radio. **LoRa is the radio technology** specifically designed for the communication with **IoT terminals**.

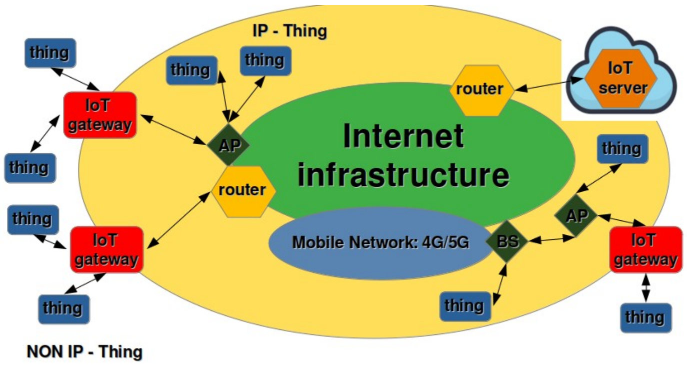


Fig 0.3 Communication links with IoT devices: **IP Things** and **NON-IP Things**

0.2 IoT Devices (IoT cores)

The core part of the IoT devices or Things, combine several kinds of electronic circuits. The **front-end** part of the core has to provide a number of i**nterconnection buses** to accommodate the sensors and the actuators. The central part (**micro-controller**) provides the processing capacity to calculate and coordinate different processing and communication tasks. Finally the **back-end** part of the core must integrate at least one type of **communication modem** such as **BT**/**BLE**, **WiFi**, cellular **4G**/**5G** and/or **LoRa**. Only **BLE** and **LoRa** have the capacity to operate in very **low power** consumption mode.

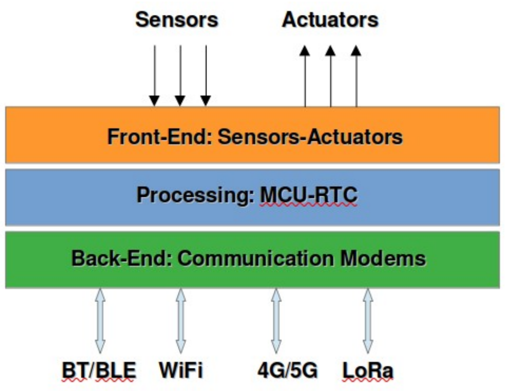


Fig 0.4 The **SoC** core of an IoT device

Modern IoT devices integrate all these circuits in one chip – **System on Chip or SoC**. One of the most popular IoT SoCs is **ESP32**.
ESP32 is a series of **low-cost**, **low-power SoC** with **Wi-Fi** and dual-mode **Bluetooth**. The ESP32 series employs either a Tensilica **Xtensa LX6/LX7** dual-core microprocessor(s) or a single-core **RISC-V** microprocessor and includes built-in low power processing unit.
ESP32 is created and developed by **Espressif Systems**, a Shanghai-based Chinese company, and is manufactured by **TSMC** using their **40 nm** process

In this series of IoT Labs we are going to learn how to build smart IoT architectures with ESP32-C3 (**RISC-V**) based boards and Pomme-Pi Zero IoT DevKit from SmartComputerLab.

0.3 The processor : RISC-V

RISC-V is an open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles. RISC-V is descendant of RISC-I to RISC-IV family. Unlike most other ISA designs, the RISC-V ISA is provided under open source licenses that do not require fees to use.

A number of companies including Espressif are offering or have announced **RISC-V hardware**, open source operating systems with RISC-V support are available and the instruction set is supported in several popular software toolchains.

Notable features of the RISC-V ISA include a load–store architecture, bit patterns to simplify the multiplexers in a CPU, IEEE 754 floating-point, a design that is architecturally neutral, and placing most-significant bits at a fixed location to speed sign extension.

The instruction set is designed for a wide range of uses. The base instruction set has a fixed length of 32-bit naturally aligned instructions, and the ISA supports variable length extensions where each instruction could be an any number of 16-bit parcels in length. Subsets support small embedded systems, personal computers, supercomputers with vector processors, and warehouse-scale 19 inch rack-mounted parallel computers.

The instruction set specification defines 32-bit and 64-bit address space variants. The specification includes a description of 128-bit flat address space variant, as an extrapolation of 32 and 64 bit variants, but the 128-bit ISA remains "not frozen" intentionally, because there is yet so little practical experience with such large memory systems.

As of June 2019, version 2.2 of the user-space ISA and version 1.11 of the privileged ISA are frozen, permitting software and hardware development to proceed.

The user-space ISA, now renamed the **Unprivileged ISA**, was updated, ratified and frozen as version 20191213. A debug specification is available as a draft, version 0.13.2.

0.4 ESP32-C3 SoC

ESP32-C3 SoC is build around 32-bit **RISC-V** processor.

ESP32-C3 integrates a 32-bit core RISC-V **RV32IMC** micro-controller with a maximum clock speed of 160 MHz. RV32IMC means base integer (I), multiplication/division (M) and compressed (C) standard extensions.

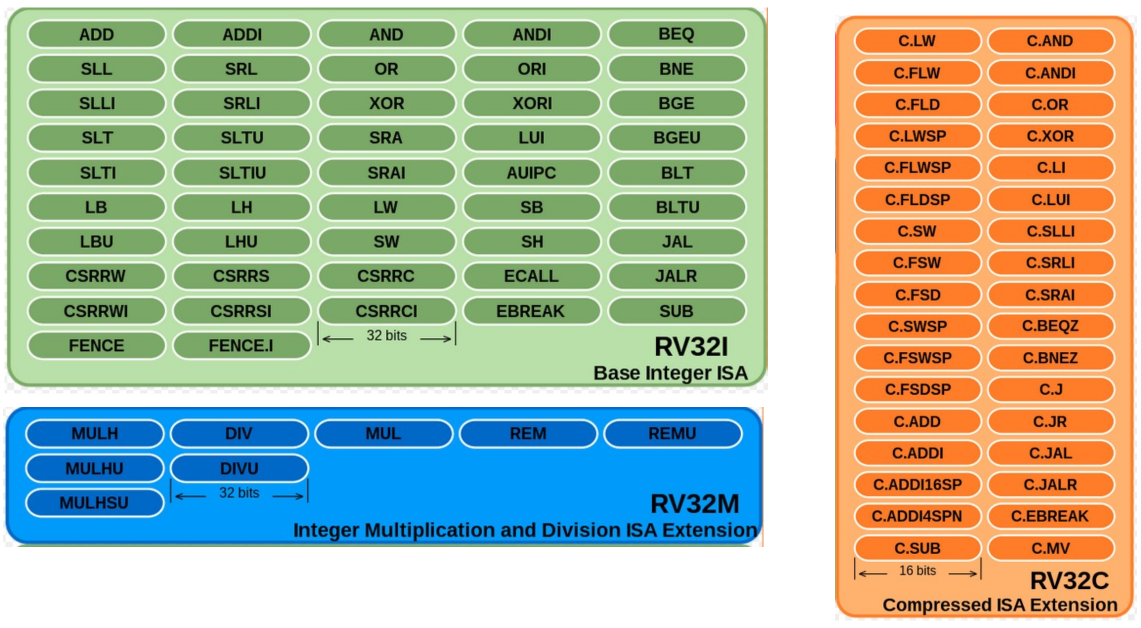


Fig.05 ESP32-C3 **RV32IMC** Instruction Set Architecture

With 22 configurable GPIOs, 400 KB of internal RAM and low-power-mode support, it can facilitate many different use-cases involving connected devices. The MCU comes in multiple variants with integrated and external flash availability. The high-temperature support makes it ideal for industrial and lighting use-cases.

Wi-Fi and Bluetooth 5 (LE) with long-range (LR) support help building devices with great coverage and improved usability. ESP32-C3 continues to support Bluetooth LE SIG Mesh and Espressif Wi-Fi Mesh.
A complete Wi-Fi subsystem that complies with IEEE 802.11b/g/n protocol and supports Station mode, SoftAP mode, SoftAP + Station mode, and promiscuous mode.

A Bluetooth LE subsystem that supports features of Bluetooth 5 and Bluetooth mesh State-of-the-art power and RF performance

32-bit RISC-V single-core processor with a four-stage pipeline that operates at up to 160 MHz 400 KB of SRAM (16 KB for cache) and 384 KB of ROM on the chip, and SPI, Dual SPI, Quad SPI, and QPI interfaces that allow connection to external flash

0.4.1 Reliable security features

- Cryptographic hardware accelerators that support AES-128/256, Hash, RSA, HMAC, digital signature and secure boot
- Random number generator
- Permission control on accessing internal memory, external memory, and peripherals
- External memory encryption and decryption

Secure Boot: ESP32-C3 implements the standard RSA-3072-based authentication scheme to ensure that only trusted applications can be used on the platform. This feature protects from executing a malicious application programmed in the flash. We understand that secure boot needs to be efficient, so that instant-on devices (such as light bulbs) can take advantage of this feature. ESP32-C3's secure boot implementation adds less than 100ms overhead in the boot process.

Flash Encryption: ESP32-C3 uses the AES-128-XTS-based flash encryption scheme, whereby the application as well as the configuration data can remain encrypted in the flash. The flash controller supports the execution of encrypted application firmware. Not only does this provide the necessary protection for sensitive data stored in the flash, but it also protects from runtime firmware changes that constitute time-of-check-time-of-use attacks.

Digital Signature and HMAC Peripheral: ESP32-C3 has a digital signature peripheral that can generate digital signatures, using a private-key that is protected from firmware access. Similarly, the HMAC peripheral can generate a cryptographic digest with a secret that is protected from firmware access. Most of the IoT cloud services use the X.509-certificate-based authentication, and the digital signature peripheral protects the device's private key that defines the device's identity. This provides a strong protection for the device's identity even in case of software vulnerability exploits.

World Controller: ESP32-C3 has a new peripheral called world controller. This provides two execution environments fully isolated from each other. Depending on the configuration, this can be used to implement a Trusted Execution Environment (TEE) or a privilege separation scheme. If the application firmware has a task that deals with sensitive security data (such as the DRM service), it can take advantage of the world controller and isolate the execution.

0.2.2 **Bluetooth 5 (LE) with Long-Range Support**

Typically, connected devices use Wi-Fi connectivity to connect to cloud services. However, Wi-Fi-only devices pose some difficulty to the network configuration of the devices, as these devices fail to provide reliable configuration feedback to the provisioner, while at the same time iOS and Android provisioners have additional complexity when connecting to the network. The availability of Bluetooth LE radio in the device makes the provisioning easy. Also, Bluetooth LE provides easy discovery and control in the local environment.
Previous versions of the Bluetooth LE protocol had a smaller range, and that made it not very suitable a protocol for local control in large spaces, e.g. big homes. ESP32-C3 adds support for the Bluetooth 5 (LE) protocol, with coded PHY and extended advertisement features, while it also provides data redundancy to the packets, thus improving the range (typically 100 meters). Furthermore, it supports the Bluetooth LE Mesh protocol. This makes it a strong candidate for controlling devices in a local network, and for communicating with other Bluetooth 5 (LE) sensor devices directly.

0.2.3 **Memory**

With a large variety in the use-cases and their memory requirements, it is tricky to determine the most suitable memory size for the SoC. However, in our experience, it is important to support use-cases with one or, sometimes, two TLS connections to the cloud, which are Bluetooth-LE-active all the time, while also supporting a reasonable application headroom on top of that. ESP32-C3's **400 KB of SRAM** can meet these requirements, while still keeping the chip's cost within the budget target. Also, ESP32-C3 has **dynamic partitioning** for the instruction (**IRAM**) and data (**DRAM**) memory. So, the usable memory is effectively maximized. It is also important to note here that we have optimized the Bluetooth subsystem's memory requirements, in comparison with ESP32.

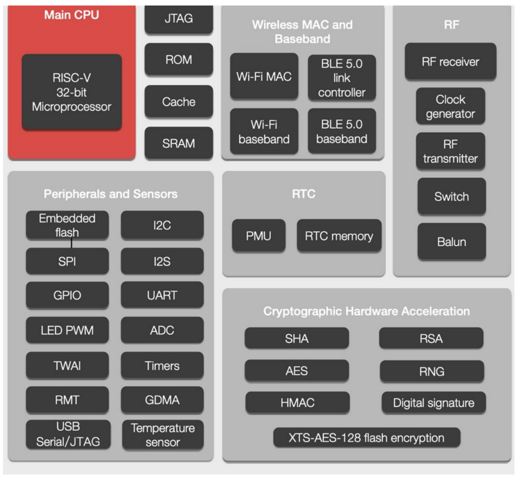


Fig 0.5 Block scheme of the internal architecture of ESP32-C3

The implementation of **Pomme-Pi Zero** within the typical Pi Zero board requires the use of small MCU board. In our case we use M5Stamp C3 board. Below you see the pinout of a M5Stamp C3 board.

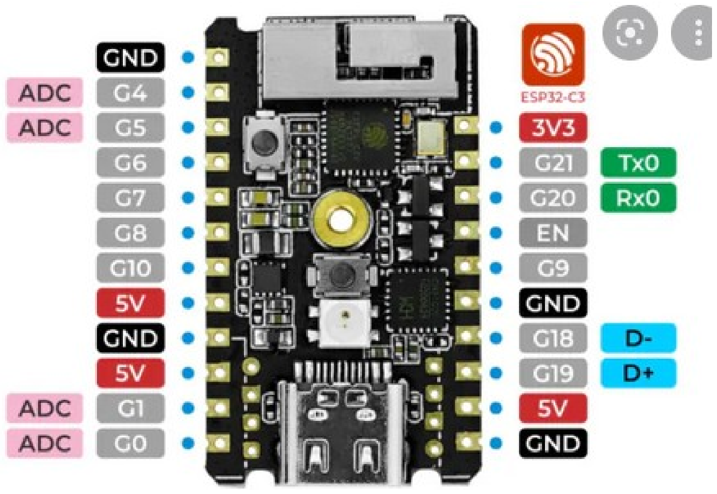


Fig 0.2 M5Stamp MCU board ant its pinout

0.3.2 Functional specifications

- Complete Wi-Fi 802.11b/g/n, 1T1R mode data rate up to 150Mbps
- Support BLE5.0 and rate support: 125Kbps, 500Kbps, 1Mbps, 2Mbps
- Onboard ESP32-C3 chip, 32-bit RISC-V single-core processor, supports a clock frequency of up to 160 MHz, with 400 KB SRAM, 384 KB ROM, 8KB RTC SRAM
- Support UART/PWM/GPIO/ADC/I2C/I2S interface, temperature sensor, pulse counter
- Integrated Wi-Fi MAC/ BB/RF/PA/LNA/BLE
- Support multiple sleep modes, deep sleep electric current is less than 5uA
- UART rate up to 5Mbps
- Support STA/AP/STA+AP mode and mix mode
- Support Smart Config (APP)/AirKiss (WeChat) of Android and IOS One-click network configuration
- Support UART port local upgrade and remote firmware upgrade (FOTA)
- Support secondary development, integrated Linux development environment
- ESP-C3-32S module acquiesce in using the built-in 4MByte Flash, meanwhile support external Flash version

0.4 The Software

1. Install the current upstream Arduino IDE at the 1.8 level or later. The current version is at the Arduino website.
2. Start Arduino and open Preferences window. In additional board manager add url: **`https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_dev_index.json`**
3. Select **`Tools -> Board -> ESP32C3 Dev Module`**

0.4.1 First code example – RGB LED and programmable BUTTON (integrated)

The following example shows the use of the integrated RGB LED and button. The default button state is **`HIGH`** – activated by : **`pinMode(BUT, INPUT_PULLUP);`**

```
#include <Adafruit_NeoPixel.h>
#define PIN        2
#define NUMPIXELS  1
#define BUT   3

Adafruit_NeoPixel pixels(NUMPIXELS, PIN, NEO_GRB + NEO_KHZ800);

int buttonState = 0;         // variable for reading the pushbutton status

void setup() {
  Serial.begin(9600);
  // initialize the pushbutton pin as an input:
  pinMode(BUT, INPUT_PULLUP);
  pixels.begin(); // INITIALIZE NeoPixel strip object (REQUIRED)
  pixels.clear(); // Set all pixel colors to 'off'
}

void loop() {
  // read the state of the pushbutton value:
  buttonState = digitalRead(BUT);
  // Show the state of pushbutton on serial monitor
  Serial.println(buttonState);
  // check if the pushbutton is pressed.
  // if it is, the buttonState is LOW:
  if (buttonState == 1) {
    pixels.setPixelColor(0, pixels.Color(255, 0, 0));
    pixels.show();
    } else {
    pixels.setPixelColor(0, pixels.Color(0, 255, 0));
    pixels.show();
  }
  // Added the delay so that we can see the output of button
  delay(100);
}
```

To do:

1. Analyze and test the above code

Lab 1

Essential ESP32-C3 SoC functionalities

In this first lab with ESP32-C3 and its RISC-V processor we are going to **oversee** the essential features of the ESP32-C3 SoC. We home that this oversee will allow you to **grasp** the available functionalities without going into the details that will be studied in the following labs.

Before starting the analysis of our board let us implement the use of an OLED display that will serve us to show the results of the following examples.

1.1 OLED display on bus I2C

```
#include <Wire.h>              // I2C bus on : SDA-7, SCL-8
#include "SSD1306Wire.h"        // legacy: #include "SSD1306.h"
SSD1306Wire display(0x3c, 7, 8);   // I2C bus ADDRESS, SDA, SCL

void display_SSD1306(float d1,float d2,float d3,float d4)
{
  char buff[64];
  display.init();
  //display.flipScreenVertically();
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.setFont(ArialMT_Plain_16);
  display.drawString(0, 0, "Lab1: IoT DevKit");
  display.setFont(ArialMT_Plain_10);
  sprintf(buff,"d1: %2.2f, d2: %2.2f\nd3: %2.2f, d4: %2.2f",d1,d2,d3,d4);
  display.drawString(0, 22, buff);
  display.drawString(20, 52, "SmartComputerLab");
  display.display();
}

void setup() {
  Serial.begin(9600);
  Wire.begin(7,8);  // required to initialize the I2C bus connection
}

float v1=0.0,v2=0.0,v3=0.0,v4=0.0;
void loop()
{
  Serial.println("in the loop");
  display_SSD1306(v1,v2,v3,v4);
  v1=v1+0.1;v2=v2+0.2;v3=v3+0.3;v4=v4+0.4;
  delay(2000);
}
```

To do:

2. Test the above code
3. Change the type of variables (`float`) to integer (`int`) and test the program

1.2 Chip ID

The **true** ESP32 chip ID is essentially its **MAC** address. The following program provides an alternate chip ID that extarcts a 32-bit integer matching the last 3 bytes of the MAC address. This is less unique than the MAC address chip ID, but is helpful when you need an identifier that can be no more than a 32-bit integer (like for `switch...case`).

```
uint32_t chipId = 0;

void setup()
{
  Serial.begin(9600);
}
```

```
void loop()
{
  for(int i=0; i<17; i=i+8)
    {
    chipId |= ((ESP.getEfuseMac() >> (40 - i)) & 0xff) << i;
    }

Serial.printf("ESP32 Chip model = %s Rev %d\n", ESP.getChipModel(), ESP.getChipRevision());
Serial.printf("This chip has %d cores\n", ESP.getChipCores());
Serial.print("Chip ID: "); Serial.println(chipId);
delay(3000);
}
```

```
#######################################################

…
This chip has 1 cores
Chip ID: 12722108
ESP32 Chip model = ESP32-C3 Rev 3
This chip has 1 cores
Chip ID: 12722108
ESP32 Chip model = ESP32-C3 Rev 3
..
```

To do:

1. Test the above code
2. Display the ID value on OLED screen

1.3 Simple FreeRTOS code with 2 tasks

The ESP32-C3 operates under the control of **FreeRTOS** system. The essential component of program running under **FreeRTOS** is **task**. The following code shows the creation and the execution of 2 tasks. Note that the ESP-C3 SoC is **unicore**.

```
#define RUNNING_CORE 0  // only one core

// define two tasks
void TaskHello( void *pvParameters );
void TaskBonjour( void *pvParameters );

// the setup function runs once when you press reset or power the board
void setup()
{
  Serial.begin(9600);
  // Now set up two tasks to run independently.
  xTaskCreatePinnedToCore(
      TaskHello,
       "TaskHello",   // A name just for humans
       1024,  // stack size can be checked & adjusted by reading the Stack Highwater
       NULL,
       2,  // Priority, with 3 (configMAX_PRIORITIES - 1) being the highest, and 0 being the lowest.
       NULL,
      RUNNING_CORE);

  xTaskCreatePinnedToCore(
      TaskBonjour,
       "TaskBonjour",
       1024,  // Stack size
       NULL,
       1,  // Priority
       NULL,
       RUNNING_CORE);

  // Now the task scheduler, which takes over control of scheduling individual tasks,
  // is automatically started.
}
```

```
void loop()
{
  // Things are done in the background tasks.
  Serial.println("in the loop task ");
  delay(2000);
}

void TaskHello(void *pvParameters)  // This is a task.
{
  (void) pvParameters;

  for (;;) // A Task shall never return or exit.
  {
    Serial.println("Hello 1");
    vTaskDelay(1000);
    Serial.println("Hello 2");
    vTaskDelay(1000);
  }
}

void TaskBonjour(void *pvParameters)  // This is a task.
{
  (void) pvParameters;

  for (;;) // A Task shall never return or exit.
  {
    Serial.println("Bonjour 1");
    vTaskDelay(1000);
    Serial.println("Bonjour 2");
    vTaskDelay(1000);
  }
}
```

To do:

1. Analyze and test the above code.
2. Modify the `loop()` **function**/**task** by changing the `delay` value.

1.4 WiFi scan

The following program demonstrates how to scan WiFi networks. We will see later that the WiFi modem provides many operational modes such as: station mode (**STA**), access point mode (**SoftAP**), long-range mode (**LR**) , direct mode (**ESP-NOW**). The scan program operates in **monitor** or **promiscuous** mode.

```
#include "WiFi.h"

void setup()
{
  Serial.begin(9600);while(!Serial);Serial.println();
  // Set WiFi transmission power to 11dBm
  WiFi.setTxPower(WIFI_POWER_11dBm);
  // Set WiFi to station mode and disconnect from an AP if it was previously connected
  WiFi.mode(WIFI_STA);
  WiFi.disconnect();
  delay(100);
  Serial.println("Setup done");
  WiFi.setTxPower(WIFI_POWER_11dBm);    delay(100);
}

void loop()
{
  Serial.println("scan start");
  // WiFi.scanNetworks will return the number of networks found
  int n = WiFi.scanNetworks();
  Serial.println("scan done");
  if (n == 0) { Serial.println("no networks found"); }
  else
```

```
    {
    Serial.print(n);
    Serial.println(" networks found");
    for (int i = 0; i < n; ++i) {
      // Print SSID and RSSI for each network found
      Serial.print(i + 1);
      Serial.print(": ");Serial.print(WiFi.SSID(i));
      Serial.print(" (");Serial.print(WiFi.RSSI(i));
      Serial.print(")");
      Serial.println((WiFi.encryptionType(i) == WIFI_AUTH_OPEN)?" ":"*");
      delay(10);
      }
    }
    Serial.println("");
    // Wait a bit before scanning again
    delay(5000);
}
```

Execution example:

```
scan start
scan done
15 networks found
1: DIRECT-G8M2070 Series (-48)*
2: VAIO-MQ35AL (-59)*
3: Livebox-08B0 (-60)*
4: PIX-LINK-2.4G (-66)
5: freebox_DZHECV (-75)*
6: FreeWifi_secure (-75)*
7: SFR_E0B0 (-84)*
8: Livebox-C6E0 (-84)*
9: SFR WiFi FON (-84)
10: SFR WiFi Mobile (-84)*
11: SFR_A0A0 (-88)*
12: SFR WiFi FON (-88)
13: FreeWifi_secure (-88)*
14: SFR WiFi Mobile (-89)*
15: freebox_ODTMTH (-89)*
```

1.5 Simple time

The data and time values mys be received via Internet from the time server such as: `pool.ntp.org.` We start the execution by a WiFi connection to the NTP server. To establish the connection we need to provide access point (AP) credentials: **ssid** (name of the AP indicating its physical address – MAC) and the **password**.

```
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
      delay(500);
      Serial.print(".");
 }
```

Note that after the reception of the **data**/**time** value from the **NTP server** the WiFi connection is closed.

```
#include <WiFi.h>
#include "time.h"
const char* ssid       = "PhoneAP";
const char* password   = "smartcomputerlab";

const char* ntpServer = "pool.ntp.org";
const long  gmtOffset_sec = 3600;
const int   daylightOffset_sec = 3600;

void printLocalTime()
{
  struct tm timeinfo;
  if(!getLocalTime(&timeinfo)){
    Serial.println("Failed to obtain time"); return;
  }
  Serial.println(&timeinfo, "%A, %B %d %Y %H:%M:%S");
}

void setup()
{
  Serial.begin(9600);while(!Serial);Serial.println();
  // Set WiFi transmission power to 13dBm
  WiFi.setTxPower(WIFI_POWER_13dBm);  // optional
  Serial.printf("Connecting to %s ", ssid);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
      delay(500);
      Serial.print(".");
  }
  Serial.println(" CONNECTED");
  //init and get the time
  configTime(gmtOffset_sec, daylightOffset_sec, ntpServer);
  printLocalTime();
  //disconnect WiFi as it's no longer needed
  WiFi.disconnect(true);
  WiFi.mode(WIFI_OFF);
}

void loop()
{
  delay(1000);
  printLocalTime();
}
```

To do:

1. Display the data/time value on OLED screen

1.6 Deep sleep

1.6.1 Simple Deep Sleep with Timer Wake Up

ESP32 offers a **deep sleep mode** for effective **power saving** as power **is an important factor** for IoT applications. In this mode CPUs, most of the RAM, and all the digital peripherals which are clocked from APB_CLK are powered off.

The only parts of the chip which can still be powered on are:

- RTC controller,
- RTC peripherals
- RTC memories

This code displays the most **basic deep sleep** with a **timer** to **wake** it up and how to store data in RTC memory to use it over reboots.

We have setup a wake cause and if needed setup the peripherals state in deep sleep, then we can start going to deep sleep. In the case that no wake up sources were provided but deep sleep was started, it will sleep forever unless hardware reset occurs.

```
#define uS_TO_S_FACTOR 1000000  /* Conversion factor for micro seconds to seconds */
#define TIME_TO_SLEEP  5        /* Time  to sleep (in seconds) */

RTC_DATA_ATTR int bootCount = 0;

void print_wakeup_reason()
{
  esp_sleep_wakeup_cause_t wakeup_reason;
  wakeup_reason = esp_sleep_get_wakeup_cause();
  switch(wakeup_reason)
{
```

```
case ESP_SLEEP_WAKEUP_EXT0 : Serial.println("Wakeup caused by external signal using RTC_IO");
                             break;
    case ESP_SLEEP_WAKEUP_EXT1 : Serial.println("External signal using RTC_CNTL"); break;
    case ESP_SLEEP_WAKEUP_TIMER : Serial.println("Wakeup caused by timer"); break;
    case ESP_SLEEP_WAKEUP_TOUCHPAD : Serial.println("Wakeup caused by touchpad"); break;
    case ESP_SLEEP_WAKEUP_ULP : Serial.println("Wakeup caused by ULP program"); break;
    default : Serial.printf("Wakeup was not caused by deep sleep: %d\n",wakeup_reason); break;
  }
}

void setup(){
  Serial.begin(9600);
  delay(1000); //Take some time to open up the Serial Monitor
  Serial.println();
  //Increment boot number and print it every reboot
  ++bootCount;
  Serial.println("Boot number: " + String(bootCount));
  //Print the wakeup reason
  print_wakeup_reason();
  // set the SoC to wake up every 5 seconds
  esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR);
  Serial.println("Sleep for every " + String(TIME_TO_SLEEP) + " Seconds");
  // By default, ESP32-C3 will automatically power down the peripherals.
  // The line below turns off all RTC peripherals in deep sleep.
  //esp_deep_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH, ESP_PD_OPTION_OFF);
  //Serial.println("Configured all RTC Peripherals to be powered down in sleep");
  Serial.println("Going to sleep now");
  Serial.flush();
  esp_deep_sleep_start();
  Serial.println("This will never be printed");
}

void loop(){
  //This is not going to be called
}
```

To do:

1. Analyze the program and test it, then modify the **time to sleep** value.
2. Add some code to `loop()` function/task.

1.7 EEPROM – write/read

EEPROM memory is no volatile, the data are kept even if there is no alimenting current.
In the example that follows we write (store) random values into the local EEPROM (max is 512 bytes). These values will **stay** in the EEPROM **when the board is turned off** and may be retrieved later by the same or **another sketch**.

```
#include "EEPROM.h"
int addr = 0; // the current address in the EEPROM
#define EEPROM_SIZE 32

void setup()
{
  Serial.begin(9600);while(!Serial);Serial.println();
  Serial.println("start...");
  if (!EEPROM.begin(EEPROM_SIZE))
  {
    Serial.println("failed to initialise EEPROM"); delay(1000000);
  }
  Serial.println(" bytes read from Flash . Values are:");
  for (int i = 0; i < EEPROM_SIZE; i++)
  {
    Serial.print(byte(EEPROM.read(i))); Serial.print(" ");
  }
  Serial.println();
  Serial.println("writing random n. in memory");
}
```

```
void loop()
{

  int val = byte(random(10020));
  // write the value to the appropriate byte of the EEPROM.
  // these values will remain there when the board is
  // turned off.
  EEPROM.write(addr, val);
  Serial.print(val); Serial.print(" ");

  addr = addr + 1;
  if (addr == EEPROM_SIZE)
  {
    Serial.println();  addr = 0;
    EEPROM.commit();
    Serial.print(EEPROM_SIZE);
    Serial.println(" bytes written on Flash . Values are:");
    for (int i = 0; i < EEPROM_SIZE; i++)
    {
      Serial.print(byte(EEPROM.read(i))); Serial.print(" ");
    }
    Serial.println(); Serial.println("----------------------------------");
  }
  delay(200);
}
```

To do:

In the previous example (deep sleep) we have used RTC memory to store the boot counter value. Now use the **EEPROM** memory to store and read the counter value int `bootCount` between two deep sleep periods.

1.8 Hardware accelerated AES cipher/decipher

The ESP32-C3 integrates sever units to provide security. These include **Advanced Encryption Standard** - AES cipher/decipher unit. The algorithm described by AES is a symmetric-key algorithm, meaning the same key is used for both encrypting and decrypting the data.
Note that the data are encrypted/decrypted by **16-byte** chunks.

```
#include "mbedtls/aes.h"

void encrypt(char * plainText, char * key, unsigned char * outputBuffer)
{
  mbedtls_aes_context aes;
  mbedtls_aes_init( &aes );
  mbedtls_aes_setkey_enc( &aes, (const unsigned char*) key, strlen(key) * 8 );
  mbedtls_aes_crypt_ecb( &aes, MBEDTLS_AES_ENCRYPT, (const unsigned char*)plainText, outputBuffer);
  mbedtls_aes_free( &aes );
}

void decrypt(unsigned char * chipherText, char * key, unsigned char * outputBuffer)
{
  mbedtls_aes_context aes;
  mbedtls_aes_init( &aes );
  mbedtls_aes_setkey_dec( &aes, (const unsigned char*) key, strlen(key) * 8 );
  mbedtls_aes_crypt_ecb(&aes, MBEDTLS_AES_DECRYPT, (const unsigned char*)chipherText, outputBuffer);
  mbedtls_aes_free( &aes );
}

void setup()
{
  Serial.begin(9600);while(!Serial); Serial.println();
  char * key = "abcdefghijklmnop";  // encrypting and decrypting key
  char *plainText = "SmartComputerLab";
  unsigned char cipherTextOutput[16];
  unsigned char decipheredTextOutput[16];

  encrypt(plainText, key, cipherTextOutput);

  decrypt(cipherTextOutput, key, decipheredTextOutput);
  Serial.println();
  Serial.print("Original plain text: "); Serial.println(plainText);
  Serial.println();
  Serial.println("Ciphered text:");
  for (int i = 0; i < 16; i++)
    {
    char str[3];
    sprintf(str, "%02x", (int)cipherTextOutput[i]);
    Serial.print(str);
  }

  Serial.println("\n\nDeciphered text:");
  for (int i = 0; i < 16; i++)
    { Serial.print((char)decipheredTextOutput[i]);}
}

void loop() {}
```

Execution example:

```
Original plain text: SmartComputerLab

Ciphered text:
968323997128a2e1d08ea60cbff16ecf

Deciphered text:
SmartComputerLab
```

To do:

1. Test the provided program.
2. Write an extended version of this program to encrypt/decrypt the following **`pack union`**:

```
union {
char mybuffer[64];
float sensor[8];
} pack;
```

Lab 2 - Buses and Sensors/Actuators

In this lab we are going to experiment with different kinds of sensors and actuators. The sensors capture analog **physical parameters** such as temperature, humidity, luminosity,.. etc. These analog values are transformed into digital values by ADC (**Analog-Digital Converters**) and sent over serial buses.
There are several types of **standard serial buses**:

1. I2C -
2. UART -
3. SPI -

and many specific serial buses depending on the nature of the sensor.

2.1 I2C bus

2.1.1 The signals and the operational principle

The physical I2C interface consists of the **serial clock** (**SCL**) and **serial data/address** (**SDA**) lines.
Both **SDA** and **SCL** lines must be connected to **VCC** through a **pull-up resistor**. The size of the pull-up resistor is determined by the amount of capacitance on the I2C lines. Data transfer may be initiated only when the bus is idle. A bus is considered idle if both **SDA** and **SCL** lines are **high** after a STOP condition.

The general procedure for a master to access a slave device is the following:

1. Suppose a **master wants to send data** to a slave:
Master-transmitter sends a START condition and **addresses** the slave-receiver
Master-transmitter sends data to slave-receiver
Master-transmitter terminates the transfer with a STOP condition

2. If a **master wants to receive/read data** from a slave:
• Master-receiver sends a START condition and addresses the slave-transmitter
• Master-receiver sends the requested register to read to slave-transmitter
• Master-receiver receives data from the slave-transmitter
• Master-receiver terminates the transfer with a STOP condition

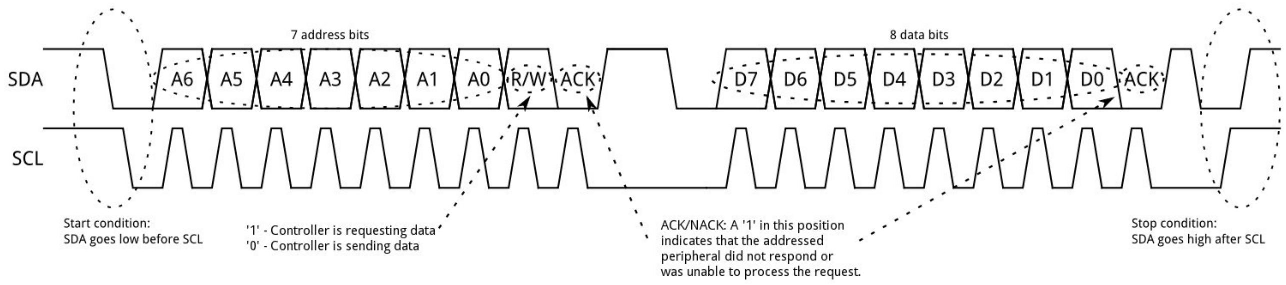

7 address bits
8 data bits
SDA
SCL
A6
A5
A4
A3
A2
A1
A0
R/W
ACK
D7
D6
D5
D4
D3
D2
D1
D0
ACK
Start condition: SDA goes low before SCL
'1' - Controller is requesting data
'0' - Controller is sending data
ACK/NACK: A '1' in this position indicates that the addressed peripheral did not respond or was unable to process the request.
Stop condition: SDA goes high after SCL

Fig 2.1 Operational sequence for I2C bus with 7-bit addresses and 8-bit data transfers

2.1.1 Testing the I2C bus – `I2Cscan`

In the previous lab we have already used the I2C bus to connect the OLED display (SSD1306) and to send the data to be displayed.
In this section we are going to scan for the presence of the devices connected to shared I2C bus. Note that the selected lines (pins) are: **Pin 7** for **SDA** line and **Pin 8** for **SCL** line; both pins are in **`PULL_UP`** state.

```
#include "Wire.h"

void setup() {
  Serial.begin(9600);
  Wire.begin(7,8);   // SDA, SCL for mini D1 plus
}
```

```
void loop() {
  byte error, address;
  int nDevices = 0;
  delay(5000);
  Serial.println("Scanning for I2C devices ...");
  for(address = 0x01; address < 0x7f; address++){
    Wire.beginTransmission(address);
    error = Wire.endTransmission();
    if (error == 0){
      Serial.printf("I2C device found at address 0x%02X\n", address);
      nDevices++;
    } else if(error != 2){
      Serial.printf("Error %d at address 0x%02X\n", error, address);
    }
  }
  if (nDevices == 0) Serial.println("No I2C devices found");
}
```

Execution example:

```
Scanning for I2C devices ...
I2C device found at address 0x3C
```

Note that we are starting the scan at address: **`address = 0x01`**; and we terminate it at: **`address < 0x7f`**;

To do:

1. Test the **`I2Cscan`** program.
2. Connect an SHT21 sensor together with the OLED (SSD1306) display, and load/execute the program.

2.1.2 Using SSD1306 - OLED display

The following code is the same as this given in the first lab. We provide it to remind you how to use (code) this device.

```
#include <Wire.h>               // SDA-7, SCL-8
#include "SSD1306Wire.h"        // legacy: #include "SSD1306.h"
SSD1306Wire display(0x3c, 7, 8);   // ADDRESS, SDA, SCL

void display_SSD1306(float d1,float d2,float d3,float d4)
{
  char buff[64];
  display.init();
  //display.flipScreenVertically();
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.setFont(ArialMT_Plain_16);
  display.drawString(0, 0, "ETN - IoT DevKit");
  display.setFont(ArialMT_Plain_10);
  sprintf(buff,"d1: %2.2f, d2: %2.2f\nd3: %2.2f, d4: %2.2f",d1,d2,d3,d4);
  display.drawString(0, 22, buff);
  display.drawString(20, 52, "SmartComputerLab");
  display.display();
}

void setup() {
  Serial.begin(9600);
  Wire.begin(7,8);
  Serial.println("Wire started");
}

float v1=0.0,v2=0.0,v3=0.0,v4=0.0;

void loop()
{
  Serial.println("in the loop");
  display_SSD1306(v1,v2,v3,v4);
  v1=v1+0.1;v2=v2+0.2;v3=v3+0.3;v4=v4+0.4;
  delay(2000);
}
```

2.1.3 Receiving data - SHT21 sensor on I2C bus

```
#include <Wire.h>
#include "SHT21.h"
SHT21 SHT21;

void setup()
{
  Wire.begin(7,8);
  SHT21.begin();
  Serial.begin(9600);
}

void loop()
{
  Serial.print("Humidity(%RH): ");
  Serial.print(SHT21.getHumidity());
  Serial.print("     Temperature(C): ");
  Serial.println(SHT21.getTemperature());
  delay(1000);
}
```

Execution example:

```
Humidity(%RH): 33.60     Temperature(C): 28.14
Humidity(%RH): 34.47     Temperature(C): 28.00
Humidity(%RH): 33.17     Temperature(C): 27.89
Humidity(%RH): 34.08     Temperature(C): 27.76
Humidity(%RH): 33.28     Temperature(C): 27.68
```

To do:

1. Test the above program.
2. Connect an SHT21 sensor together with the OLED (SSD1306) module and display the results on the screen.

2.1.4 Reading data - BH1750 luminosity sensor

This example initializes the BH1750 object using the high resolution one-time mode and then makes a light level reading every second. The BH1750 component starts up in default mode when it next powers up.

```
#include <BH1750.h>
#include <Wire.h>

BH1750 lightMeter;

void setup() {
  Serial.begin(9600);
  while(!Serial); Serial.println();
  Wire.begin(7,8);
  delay(2000);
  lightMeter.begin(BH1750::ONE_TIME_HIGH_RES_MODE);
  Serial.println("BH1750 One-Time Test");
}

void loop() {
  while (!lightMeter.measurementReady(true)) {   yield();  }  // waiting for the data
  float lux = lightMeter.readLightLevel();
  Serial.print("Light: ");
  Serial.print(lux);
  Serial.println(" lx");
  lightMeter.configure(BH1750::ONE_TIME_HIGH_RES_MODE);
}
```

Execution example:

```
Light: 302.50 lx
Light: 304.17 lx
Light: 36.67 lx
Light: 30.00 lx
```

```
Light: 1474.17 lx
Light: 1482.50 lx
Light: 1493.33 lx
Light: 293.33 lx
```

To do:

1. Test the above program.
2. Connect an BH1750 sensor together with the OLED (SSD1306) module and display the results on the screen.

2.1.5 Reading data - SGP30 Indoor Air Quality sensor

This example reads the sensors calculated **CO2** and **TVOC** values. Note that the correct use of the sensor requires at least 20-30 initial readings before getting to the correct values.

```
#include "SparkFun_SGP30_Arduino_Library.h"
// Click here to get the library: http://librarymanager/All#SparkFun_SGP30
#include <Wire.h>

SGP30 mySensor; //create an object of the SGP30 class

void setup() {
  Serial.begin(9600);
  while(!Serial); Serial.println();
  Wire.begin(19,18);
  delay(2000);
  if (mySensor.begin() == false) {
    Serial.println("No SGP30 Detected. Check connections.");
    while (1);
  }
  //Initializes sensor for air quality readings
  mySensor.initAirQuality();
}

void loop() {
  //First fifteen readings will be
  //CO2: 400 ppm  TVOC: 0 ppb
  delay(1000); //Wait 1 second
  //measure CO2 and TVOC levels
  mySensor.measureAirQuality();
  Serial.print("CO2: ");
  Serial.print(mySensor.CO2);
  Serial.print(" ppm\tTVOC: ");
  Serial.print(mySensor.TVOC);
  Serial.println(" ppb");
}
```

Execution example:

```
CO2: 405 ppm   TVOC: 58 ppb
CO2: 590 ppm   TVOC: 132 ppb
CO2: 494 ppm   TVOC: 114 ppb
CO2: 515 ppm   TVOC: 112 ppb
CO2: 436 ppm   TVOC: 91 ppb
CO2: 468 ppm   TVOC: 87 ppb
CO2: 565 ppm   TVOC: 143 ppb
CO2: 400 ppm   TVOC: 57 ppb
CO2: 400 ppm   TVOC: 34 ppb
CO2: 400 ppm   TVOC: 0 ppb
```

To do:

1. Test the above program.
2. Connect an **SGP30 sensor** together with the OLED (SSD1306) module and display the results on the screen.

2.1.6 Reading data - BMP280 air pressure and temperature

BMP280 sensor, is an environmental sensor with temperature, barometric pressure which is the next . This sensor is ideal for all kinds of weather detection it can be used with I2C .

This precision sensor is the best low-cost precision detection solution for measuring **barometric pressure** with an absolute accuracy of **± 1 hPa** and **temperature** with an accuracy of **± 1.0 ° C**. We can also use it as an **altimeter** with an accuracy of ± 1 meter.

```
// example of using library farmerkeith_BMP280.h for pressure and temperature

#include <farmerkeith_BMP280.h>
#include <Wire.h>

bmp280 bmp0 ; // creates object bmp of type bmp280, base address

void setup() {
  Serial.begin(9600); // use this if you get errors with the faster rate
  Serial.println("\nStart of basicPressureAndTemperature sketch");
  Serial.println("Printing pressure and temperature at 5 second intervals");
  Wire.begin(19,18); // initialise I2C protocol
  bmp0.begin(); // initialise BMP280 for continuous measurements
}

void loop() {
  double temperature;
  double pressure=bmp0.readPressure (temperature); // measure pressure and temperature
  Serial.print("Atmospheric pressure = ");
  Serial.print(pressure,4); // print with 4 decimal places
  Serial.print(" hPa. Temperature = ");
  Serial.print(temperature,2); // print with 2 decimal places
  Serial.println( " degrees Celsius");
  delay(5000);
}
```

Execution example:

```
Start of basicPressureAndTemperature sketch
Printing pressure and temperature at 5 second intervals
Atmospheric pressure = 1018.7044 hPa. Temperature = 28.10 degrees Celsius
Atmospheric pressure = 1018.7126 hPa. Temperature = 28.08 degrees Celsius
```

To do:

1. Test the above program.
2. Connect a **BMP280 sensor** together with the OLED (SSD1306) module and display the results on the screen.

2.1.7 Reading data - PAJ7620 gesture sensor

This demo program can recognize 9 gestures and output the result, including **move up**, **move down**, **move left**, **move right**, **move forward**, **move backward**, **circle-clockwise**, **circle-counter clockwise**, and **wave**. Includes `enum` definition of `GES_*` return values from `readGesture()`.

```
#include "RevEng_PAJ7620.h"
RevEng_PAJ7620 sensor = RevEng_PAJ7620();

void setup()
{
  Serial.begin(9600);
  while(!Serial); Serial.println();
  Wire.begin(19,18);
  delay(2000);
  Serial.println("PAJ7620 sensor demo: Recognizing all 9 gestures.");
  if( !sensor.begin() )           // return value of 0 == success
  {
    Serial.print("PAJ7620 I2C error - halting");
    while(true) { }
  }
          Serial.println("PAJ7620 init: OK");
  Serial.println("Please input your gestures:");
}

void loop()
{
  Gesture gesture;                  // Gesture is an enum type from RevEng_PAJ7620.h
  gesture = sensor.readGesture();   // Read back current gesture (if any) of type Gesture
  switch (gesture)
  {
    case GES_FORWARD:
      { Serial.print("GES_FORWARD");  break;  }
    case GES_BACKWARD:
      {  Serial.print("GES_BACKWARD"); break; }
    case GES_LEFT:
      { Serial.print("GES_LEFT"); break;  }
    case GES_RIGHT:
      {  Serial.print("GES_RIGHT"); break;  }
    case GES_UP:
      {  Serial.print("GES_UP");  break;  }
    case GES_DOWN:
      { Serial.print("GES_DOWN");  break;}
    case GES_CLOCKWISE:
      {  Serial.print("GES_CLOCKWISE"); break; }
    case GES_ANTICLOCKWISE:
      {  Serial.print("GES_ANTICLOCKWISE");  break; }
    case GES_WAVE:
      {  Serial.print("GES_WAVE"); break; }
    case GES_NONE: { break;}
  }

  if( gesture != GES_NONE )
  {
    Serial.print(", Code: ");
    Serial.println(gesture);
  }
  delay(100);
}
```

Execution example:

```
PAJ7620 sensor demo: Recognizing all 9 gestures.
PAJ7620 init: OK
Please input your gestures:
GES_LEFT, Code: 3
GES_RIGHT, Code: 4
GES_LEFT, Code: 3
GES_RIGHT, Code: 4
GES_UP, Code: 1
GES_DOWN, Code: 2
```

To do:

Test the above program.
Connect a **PAJ7620 sensor** together with the OLED (SSD1306) module and display the results on the screen.

2.1.8 Storing and reading data with external EEPROM module

In the Lab1 we have introduced the EEPROM memory to store the data in non volatile memory. The size of the internal EEPROM is limited to 512 bytes. If you need more space or you wish to keep the data in an extenal memory you may use external EEPROM module such as 24LC256 chip.
This chip offers you 32K bytes of non volatile memory. It is connected via an I2C bus.

```
#include <Wire.h>
#define disk1 0x50    //Address of 24LC256 eeprom chip

void writeEEPROM(int deviceaddress, unsigned int eeaddress, byte data )
{
  Wire.beginTransmission(deviceaddress);
  Wire.write((int)(eeaddress >> 8));   // MSB
  Wire.write((int)(eeaddress & 0xFF)); // LSB
  Wire.write(data);
  Wire.endTransmission();
  delay(5);
}

byte readEEPROM(int deviceaddress, unsigned int eeaddress )
{
  byte rdata = 0xFF;
  Wire.beginTransmission(deviceaddress);
  Wire.write((int)(eeaddress >> 8));   // MSB
  Wire.write((int)(eeaddress & 0xFF)); // LSB
  Wire.endTransmission();
  Wire.requestFrom(deviceaddress,1);
  if (Wire.available()) rdata = Wire.read();
  return rdata;
}

void setup(void)
{
  Serial.begin(9600);
  while(!Serial); Serial.println();
  Wire.begin(7,8);
  delay(2000);
}

byte val=0;
int i=0;

void loop()
{
  unsigned int address;
  address=0;val=0;
  for(i=0;i<16;i++)
  {
    writeEEPROM(disk1, address, val);
    address++;val++;
    delay(100);
  }
  Serial.println();
  address=0;
  for(i=0;i<16;i++)
  {
    Serial.print(readEEPROM(disk1, address), DEC);Serial.print(',');
    address++;
    delay(100);
  }
  Serial.println();
  delay(2000);
}
```

To do:

1. Test the above program.
2. Use the same program with random values generated for each stored byte.

2.2 UART – serial bus

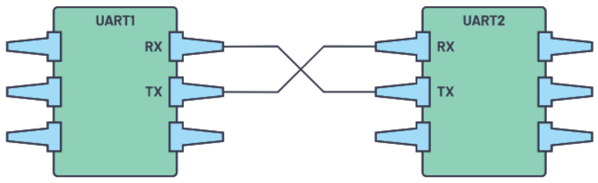


Fig 2.2 UART bus data lines (Rx,Tx) connections

2.2.1 UART – operational principle

By definition, **UART** is a hardware communication protocol that uses **asynchronous serial communication** with configurable speed. Asynchronous means there is **no clock signal** to synchronize the output bits from the transmitting device going to the receiving end.

The two signals of each UART device are named:

- Transmitter (Tx)
- Receiver (Rx)

The main purpose of a transmitter and receiver line for each device is to transmit and receive serial data intended for serial communication.

The transmitting UART is connected to an internal controlling **data bus** that sends data in a parallel form. From this, the data will now be transmitted on the transmission line (wire) serially, **bit by bit**, to the receiving UART. This, in turn, will convert the serial data into parallel for the receiving device.

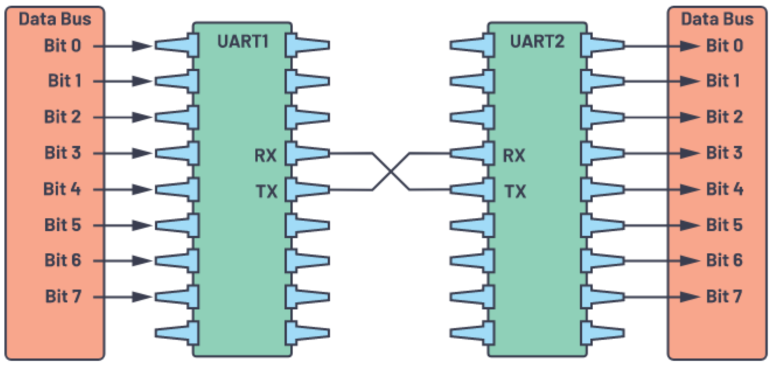
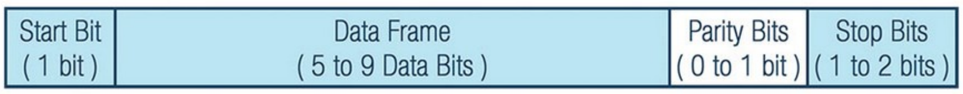


Fig 2.3 Serialization and de-serialization of UART bus data lines (Rx,Tx)

For UART and most serial communications, the **baud rate** needs to be set the same on both the transmitting and receiving device. The baud rate is the rate at which information is transferred to a communication channel. In the serial port context, the set baud rate will serve as the **maximum number of bits per second** to be transferred.



Start Bit (1 bit)	Data Frame (5 to 9 Data Bits)	Parity Bits (0 to 1 bit)	Stop Bits (1 to 2 bits)

Fig 2.4 UART packet (5-8 bits) with start (1) , parity (0-1), and stop bits (1-2 bits)

In UART, the mode of transmission is in the form of a **packet**. The piece that connects the transmitter and receiver includes the creation of serial packets and controls those physical hardware lines. A packet consists of a **start bit**, **data frame (example: 1 byte)** , a **parity bit**, and **stop bits**.

2.2.1.1 Start Bit

The UART data transmission line is **normally held at a high voltage level** (`pull_up`) when it's not transmitting data. To start the transfer of data, the transmitting UART pulls the transmission line from high to low for one (1) clock cycle. When the receiving UART detects the high to low voltage transition, it begins reading the bits in the data frame at the frequency of the baud rate.

2.2.1.2 Data Frame

The data frame contains the actual data being transferred. It can be five (5) bits up to eight (8) bits long if a parity bit is used. If no parity bit is used, the data frame can be nine (9) bits long. In most cases, the data is sent with the least significant bit first

2.2.1.3 Parity

Parity describes the evenness or oddness of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission. Bits can be changed by electromagnetic radiation, mismatched baud rates, or long-distance data transfers.
After the receiving UART reads the data frame, it counts the number of bits with a value of 1 and checks if the total is an even or odd number. If the parity bit is a 0 (even parity), the 1 or logic-high bit in the data frame should total to an even number. If the parity bit is a 1 (odd parity), the 1 bit or logic highs in the data frame should total to an odd number.
When the parity bit matches the data, the UART knows that the transmission was free of errors. But if the parity bit is a 0, and the total is odd, or the parity bit is a 1, and the total is even, the UART knows that bits in the data frame have changed.

2.2.1.4 Stop Bits

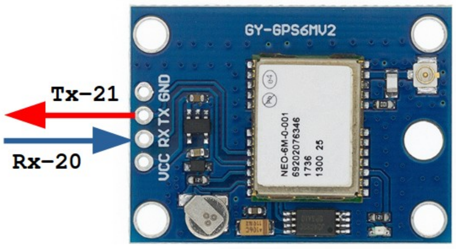
To signal the end of the data packet, the sending UART drives the data transmission line from a low voltage to a high voltage for one (1) to two (2) bit(s) duration

An example of UART parameters specification

```
uart.begin(9600, SERIAL_8N1, 20, 21); //RX0, Tx0
```

means:
speed - 9600 bauds, data_frame - 8 bits (byte), no parity, 1 stop bit

2.2.2 Receiving data - GPS modem (no decoding library)



The following code uses UART0 interface, the same as the terminal interface. The GPS modem sends automatically the UART data flow to the IDE terminal.
To capture the flow for internal needs we may show the received results on the OLED screen.

```
HardwareSerial uart(0);
#include <Wire.h>               // SDA-7, SCL-8
#include "SSD1306Wire.h"        // legacy: #include "SSD1306.h"

SSD1306Wire display(0x3c, 7, 8);

#include <Adafruit_NeoPixel.h>
#define PIN         2
#define NUMPIXELS   1
```

```
Adafruit_NeoPixel pixels(NUMPIXELS, PIN, NEO_GRB + NEO_KHZ800);
// RX0-21, TX0-20

void display_SSD1306(char* d1)
{
  char buff[64];
  display.init();
  //display.flipScreenVertically();
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.setFont(ArialMT_Plain_16);
  display.drawString(0, 0, "Smart IoT DevKit");
  display.drawString(0, 28, d1);
  display.setFont(ArialMT_Plain_10);
  display.drawString(20, 52, "SmartComputerLab");
  display.display();
}

void setup()
{
uart.begin(9600, SERIAL_8N1, 21, 20); //TX0, Rx0
pixels.begin(); // INITIALIZE NeoPixel strip object (REQUIRED)
pixels.clear(); // Set all pixel colors to 'off'
Wire.begin(7,8);
delay(1000);
}

int i=0,ihour;
char gpst[2048],ttime[16],ftime[16],hour[8],nhour[2];
char *ptr=NULL;
uint8_t buff[32];

void loop()
{
i=0;
memset(gpst,0x00,2048);
while (uart.available() > 0)
  {
  char gpsData = uart.read();
  gpst[i]=(char)gpsData;i++;
  }
  if(i && i<100)
  {
    pixels.setPixelColor(0, pixels.Color(0, 255, 0)); pixels.show();
  }
    if(i && i>100)
  {
    pixels.setPixelColor(0, pixels.Color(0, 0, 255)); pixels.show();
    ptr=strstr(gpst,"$GPGGA");
    memcpy(buff,ptr,13); memcpy(buff,"Time :",6);
    display_SSD1306((char*)buff);
  }
  delay(200);
  pixels.setPixelColor(0, pixels.Color(255, 0, 0));
  pixels.show();
  delay(500);
}
```

We use also the integrated RGB LED to signal the arrival of the UART packet.

2.4 One-Wire bus (SIG-9)

One wire bus a solution to **send/receive the data** asynchronously on a **single line**. Some specific devices sensors/actuators are using their own protocol. This is the case for **DHT11/22 (Temperature/Humidity)** sensors.

2.4.1 DHT22 sensor

```
#include "DHT.h"
#define DHTPIN 9    // Digital pin connected to the DHT sensor
#define DHTTYPE DHT22   // DHT 22  (AM2302), AM2321
DHT dht(DHTPIN, DHTTYPE);

void setup() {
  Serial.begin(9600);
  Serial.println("DHTxx test!");
  dht.begin();
}

void loop() {
  delay(2000);
  float h = dht.readHumidity();
  // Read temperature as Celsius (the default)
  float t = dht.readTemperature();
  // Check if any reads failed and exit early (to try again).
  if (isnan(h) || isnan(t)) {
    Serial.println("Failed to read from DHT sensor!");
    return;
  }

  Serial.print("Humidity: ");
  Serial.print(h);
  Serial.print("%  Temperature: ");
  Serial.println(t);
}
```

2.4.2 RGB LED

RGB LADS may be driven vi a single line.

The following code shows the use of a RING LED to display the time values: hours, minutes and seconds with different colors. We exploit a 12-LED ring, so the minutes and the seconds are displayed on the ring with a precision of 5-seconds and 5-minutes.

```
#include <Adafruit_NeoPixel.h>
#include <WiFi.h>
#include "time.h"
long lastUpdate = millis();
long lastSecond = millis();

String hours, minutes, seconds;
int currentSecond, currentMinute, currentHour;

const char* ssid  = "PhoneAP";  //  your network SSID (name)
const char* password  = "smartcomputerlab";       // your network password

const float UTC_OFFSET = 0;
const char* ntpServer = "pool.ntp.org";
const long  gmtOffset_sec = 3600;
const int   daylightOffset_sec = 3600;

Adafruit_NeoPixel led = Adafruit_NeoPixel(12, 9);

int ledh,ledm,leds;

void printLocalTime()
{
  struct tm timeinfo;
  if(!getLocalTime(&timeinfo)){
    Serial.println("Failed to obtain time"); return;
  }
  Serial.println(&timeinfo, "%A, %B %d %Y %H:%M:%S");
  Serial.println(timeinfo.tm_hour);
  Serial.println(timeinfo.tm_min);
  Serial.println(timeinfo.tm_sec);
  ledh=(timeinfo.tm_hour+7)%12;
  ledm=(timeinfo.tm_min/5+7)%12;
  leds=(timeinfo.tm_sec/5+7)%12;
  led.clear();
  led.setPixelColor(leds, led.Color(0, 0, 255));
  led.setPixelColor(ledm, led.Color(0, 255, 0));
  led.setPixelColor(ledh, led.Color(255, 0, 0));
  led.show();
}

void setup()
{
  Serial.begin(9600);while(!Serial);Serial.println();
  led.begin(); // INITIALIZE NeoPixel strip object (REQUIRED)
  led.clear(); // Set all pixel colors to 'off'
  // Set WiFi transmission power to 13dBm
  WiFi.setTxPower(WIFI_POWER_13dBm);  // optional
  Serial.printf("Connecting to %s ", ssid);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
      delay(500);
      Serial.print(".");
  }
  Serial.println(" CONNECTED");
  //init and get the time
  configTime(gmtOffset_sec, daylightOffset_sec, ntpServer);
  printLocalTime();
  //disconnect WiFi as it's no longer needed
  WiFi.disconnect(true);
  WiFi.mode(WIFI_OFF);
}

void loop()
{
  delay(1000);
  printLocalTime();
}
```

2.4.3 PIR sensor – SR602

The **PIR** (**Infra Red** sensor) detects the presence of Infra Red radiation emitted by human skin. The presence of radiation is signaled by an interruption signal carried here on the connected to **Pin 9**.

```
#define PIR 9
bool MOTION_DETECTED = false;

void pinChanged()
{
  MOTION_DETECTED = true;
}

void setup()
{
  Serial.begin(9600);
  attachInterrupt(PIR, pinChanged, RISING);
}

int counter=0;

void loop()
{
  int i=0;
  if(MOTION_DETECTED)
  {
    Serial.println("Motion detected.");
    delay(1000);counter++;
    MOTION_DETECTED = false;
    Serial.println(counter);
  }
}
```

2.4.4 Doppler radar sensor - RCWL-0516

The program demonstrating RCWL-0516 "doppler radar microwave motion sensor module"

```
// the sensor has 3 second latency
int  detectPin = 9;
bool detect    = false;
int  led       = 3;  // GREEN_LED

void setup() {
  Serial.begin(9600);while(!Serial);Serial.println();
  Serial.println("Starting...\n");
  pinMode (detectPin, INPUT);
  pinMode (led, OUTPUT);
}

void loop() {

  detect = digitalRead(detectPin);

  if(detect == true) {
    digitalWrite(led, HIGH);
    Serial.println("Movement detected");
  }
  else {
    digitalWrite(led, LOW);
  }

  delay(1000);
}
```

2.5 SPI bus – operational principles

Serial Peripheral Interface (SPI) is an interface bus commonly used to send data between micro-controllers and peripherals such as displays, sensors, modems, and SD cards. It uses separate clock and data lines, along with a select line to choose the device you wish to talk to.

SPI is a "synchronous" data bus, which means that it uses separate lines for data and a "clock" that keeps both sides in perfect sync. The clock is an oscillating signal that tells the receiver exactly when to sample the bits on the data line. This could be the rising (low to high) or falling (high to low) edge of the clock signal; the datasheet will specify which one to use. When the receiver detects that edge, it will immediately look at the data line to read the next bit (see the arrows in the below diagram). Because the clock is sent along with the data, specifying the speed isn't important, although devices will have a top speed at which they can operate.

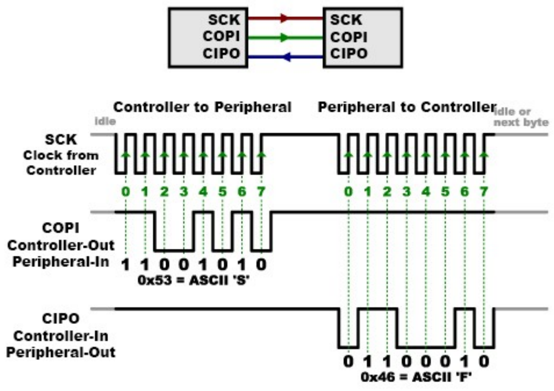


Fig 2.5 Operational principle of SPI bus with SCK-clock, COPI/MOSI, CIPO/MISO lines

2.5.1 SPI Clock and data signals/lines

In SPI, **only one side generates the clock signal** (usually called **CLK** or **SCK** for **Serial ClocK**). The side that generates the clock is called the "controller" or Master, and the other side is called the "peripheral" or Slave. There is always only one controller (which is almost always your micro-controller), but there can be multiple peripherals (more on this in a bit).
When data is sent from the controller to a peripheral, it's sent on a data line called COPI/**MOSI**, for "Controller Out/Peripheral In". If the peripheral needs to send a response back to the controller, the controller will continue to generate a prearranged number of clock cycles, and the peripheral will put the data onto a third data line called CIPO/**MISO**, for "Controller In/Peripheral Out".

2.5.1.1 Chip Select (CS)

There's one last line you should be aware of, called **CS** for Chip Select. This tells the peripheral that it should **wake up** and receive/send data and is also used when multiple peripherals are present to select the one you'd like to talk to

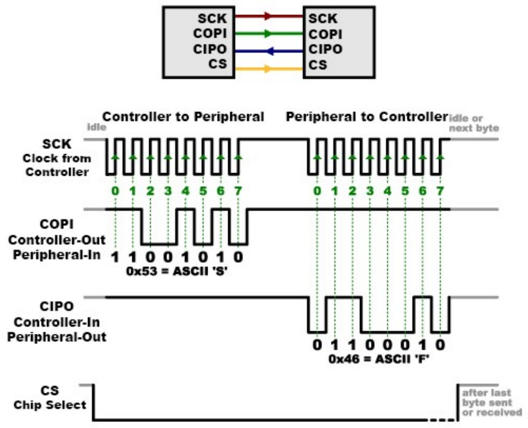


Fig 2.5 Operational principle of selection line (**CS**)

The CS line is normally held high, which disconnects the peripheral from the SPI bus. (This type of logic is known as “active low,” and you’ll often see used it for enable and reset lines.) Just before data is sent to the peripheral, the line is brought low, which activates the peripheral. When you're done using the peripheral, the line is made high again. In a shift register, this corresponds to the "latch" input, which transfers the received data to the output lines.

2.5.1.2 Multiple peripherals

In general, each peripheral will need a **separate CS line**. To talk to a particular peripheral, we'll make that peripheral's CS line low and keep the rest of them high . Lots of peripherals will require lots of CS lines; if you're running low on outputs, there are binary decoder chips that can multiply your CS outputs.

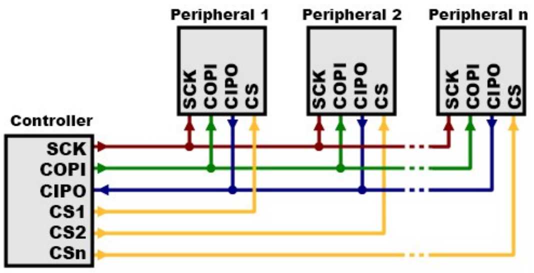


Fig 2.6 Multiple peripherals

2.5.2 SPI bus with LoRa modem – SX1276/8

The SX1276/77/78/79 transceivers feature the LoRaTM long range modem that provides ultra-long range spread spectrum communication and high interference immunity whilst minimising current consumption. Using Semtech’s patented LoRaTM modulation technique SX1276/77/78/79 can achieve a sensitivity of over -148dBm using a low cost crystal and bill of materials. The high sensitivity combined with the integrated +20 dBm power amplifier yields industry leading link budget making it optimal for any application requiring range or robustness.

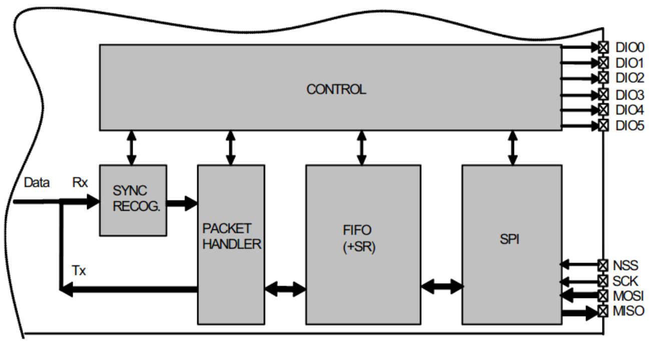
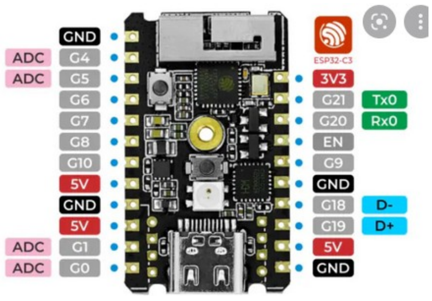


Fig 2.7 Lora modem (SX1276/8) SPI interface (**RST** pin is not visible)

2.5.2.1 The code of a simple LoRa sender

```
#include <SPI.h>
#include <LoRa.h>
// connection lines - SPI bus
#define SCK     6   // GPIO18 -- SX127x's SCK
#define MISO    0   // GPIO19 -- SX127x's MISO
#define MOSI    1   // GPIO23 -- SX127x's MOSI
#define SS     10   // GPIO05 -- SX127x's CS
#define RST     4   // GPIO15 -- SX127x's RESET
#define DI0     5   // GPIO26 -- SX127x's IRQ(Interrupt Request)
// radio parameters
#define freq    434E6
#define sf 7
#define sb 125E3
```



```
union pack
{
  uint8_t frame[16]; // trames avec octets
  float  data[4];    // 4 valeurs en virgule flottante
} sdp ;  // paquet d'émission

void setup() {
  Serial.begin(9600);
  SPI.begin(SCK,MISO,MOSI,SS);
  LoRa.setPins(SS,RST,DI0);
  Serial.println();delay(100);Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
Serial.println("Starting LoRa OK!");
LoRa.setSpreadingFactor(sf);
LoRa.setSignalBandwidth(sb);
}

float d1=0.0, d2=0.0 ;

void loop()  // la boucle de l'emetteur
{
Serial.print("New Packet:") ;
  LoRa.beginPacket();                   // start packet
  sdp.data[0]=d1;
  sdp.data[1]=d2;
  LoRa.write(sdp.frame,16);
  LoRa.endPacket();
  Serial.printf("%2.2f,%2.2f\n",d1,d2);
  d1=d1+0.1; d2=d2+0.2;
  delay(2000);
}
```

2.5.2.2 The code of a simple LoRa receiver

```
#include <SPI.h>
#include <LoRa.h>
#define SCK     6   // GPIO18 -- SX127x's SCK
#define MISO    0 //7   // GPIO19 -- SX127x's MISO
#define MOSI    1 //8   // GPIO23 -- SX127x's MOSI
#define SS      10   // GPIO05 -- SX127x's CS
#define RST     4   // GPIO15 -- SX127x's RESET
#define DI0     5   // GPIO26 -- SX127x's IRQ(Interrupt Request)
#define freq    434E6
#define sf 7
#define sb 125E3

union pack
{
  uint8_t frame[16]; // trames avec octets
  float  data[4];    // 4 valeurs en virgule flottante
} rdp ;  // paquet de réception

void setup() {
  Serial.begin(9600);
  SPI.begin(SCK,MISO,MOSI,SS);
  LoRa.setPins(SS,RST,DI0);
  Serial.println();delay(100);Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
Serial.println("Starting LoRa OK!");
LoRa.setSpreadingFactor(sf);
LoRa.setSignalBandwidth(sb);
}

float d1=0.0, d2=0.0 ;
int rssi;

void loop()
{
int packetLen;
packetLen=LoRa.parsePacket();
if(packetLen==16)
  {
  int i=0;
  while (LoRa.available()) {
    rdp.frame[i]=LoRa.read();i++;
    }
  d1=rdp.data[0];d2=rdp.data[1];
  rssi=LoRa.packetRssi();  // force du signal en réception en dB
  Serial.printf("Received packet:%2.2f,%2.2f\n",d1,d2);
  Serial.printf("RSSI=%d\n",rssi);
  }
}
```

Lab 3

Simple WEB and IoT servers (ThingSpeak, MQTT,..)

This lab deals with IoT servers. First we are going to build a simple WEB server that create a web page used to control the RGB LED colors on our board.
This example may be used to build more complex control of the devices attached to our IoT board.
The second section deals with the "standard" IoT services/servers that allows us to send and receive simple massages carrying IoT data. This service is based on MQTT protocol and the corresponding broker/client programs.
Finally, in the last section of this lab we introduce **ThingSpeak** servers allowing us to send/receive and store the IoT messages. In case of **ThingSpeak.com** server associated to **Matlab**, these data may be further analyzed by statistical tools.

3.1.1 Simple WEB server in station mode (STA)

The following example contains simple WEB server connected to the external access point ("`PhoneAP`"). The user needs to connect the navigator to the obtained IP address.
The default page sends the values of two variables: `t`, `h` in a simple text HTML page.

```
#include <WiFi.h>
#include <WiFiClient.h>
#include <WebServer.h>
#include <ESPmDNS.h>
#include <Adafruit_NeoPixel.h>
#define PIN        2

#define NUMPIXELS 1 // one pixel LED

const char* ssid = "PhoneAP";
const char* password = "smartcomputerlab";
Adafruit_NeoPixel pixels(NUMPIXELS, PIN, NEO_GRB + NEO_KHZ800);
WebServer server(80);

int t=24, h=55;

void handleRoot() {
  char buff[32];
  sprintf(buff,"Temp:%d, Humi:%d", t,h);
  server.send(200, "text/plain", buff);
  pixels.setPixelColor(0, pixels.Color(0, 150, 0));pixels.show();
  }

void handleNotFound() {
  pixels.setPixelColor(0, pixels.Color(150, 0, 0));pixels.show();
  String message = "File Not Found\n\n";
  message += "URI: ";
  message += server.uri();
  message += "\nMethod: ";
  message += (server.method() == HTTP_GET) ? "GET" : "POST";
  message += "\nArguments: ";
  message += server.args();
  message += "\n";
  for (uint8_t i = 0; i < server.args(); i++) {
    message += " " + server.argName(i) + ": " + server.arg(i) + "\n";
  }
  server.send(404, "text/plain", message);
  pixels.setPixelColor(0, pixels.Color(0, 150, 0));pixels.show();
}

void setup(void) {
  Serial.begin(9600);
  pixels.begin();
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);
  Serial.println("");

  // Wait for connection
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.print("Connected to ");
  Serial.println(ssid);
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());
  pixels.setPixelColor(0, pixels.Color(0, 0, 150));pixels.show();
  if (MDNS.begin("esp32")) {
    Serial.println("MDNS responder started");
  }
  server.on("/", handleRoot);
  server.on("/inline", []() {
    server.send(200, "text/plain", "this works as well");
  });
  server.onNotFound(handleNotFound);
  server.begin();
  Serial.println("HTTP server started");
}

void loop(void) {
  server.handleClient();
  delay(2);//allow the cpu to switch to other tasks
}
```

To do:

1. Display the IP address on OLED screen.
2. Modify the content of the WEB page using some HTML tags.

3.1.2 Simple WEB server with soft access point (`SoftAP`)

The following example integrates a simple server and operates in soft access point mode.
The user terminal may connect to this access point ("RV-AP") and use default IP address: 192.168.4.1. This address may be modified with :

```
WiFi.softAPConfig(local_ip, gateway, subnet);
```

fonction.

The **html** page presented by the server is defined in `String SendHTML` function.

```
#include <WiFi.h>
#include <WebServer.h>
#include <Adafruit_NeoPixel.h>
#define PIN        2

#define NUMPIXELS 1 // one pixel LED

const char* ssid = "RV-AP";
const char* password = NULL;

Adafruit_NeoPixel pixels(NUMPIXELS, PIN, NEO_GRB + NEO_KHZ800);

/* Put IP Address details */
//IPAddress local_ip(192,168,1,1);
//IPAddress gateway(192,168,1,1);
//IPAddress subnet(255,255,255,0);

bool LED1status = LOW;
bool LED2status = LOW;

WebServer server(80);

void setup() {
  Serial.begin(9600);
  pixels.begin();

  WiFi.softAP(ssid, password);
  //WiFi.softAPConfig(local_ip, gateway, subnet);
  delay(100);

  server.on("/", handle_OnConnect);
  server.on("/ledredon", handle_ledredon);
  server.on("/ledredoff", handle_ledredoff);
  server.on("/ledgreenon", handle_ledgreenon);
  server.on("/ledgreenoff", handle_ledgreenoff);
  server.onNotFound(handle_NotFound);

  server.begin();
  Serial.println("HTTP server started");
}
void loop() {
  server.handleClient();
  if(LED1status)
  {  pixels.setPixelColor(0, pixels.Color(150, 0, 0));pixels.show();}
  else
  {pixels.setPixelColor(0, pixels.Color(0, 0, 0));pixels.show();}

  if(LED2status)
  {pixels.setPixelColor(0, pixels.Color(0,150, 0));pixels.show();}
  else
  {pixels.setPixelColor(0, pixels.Color(0, 0, 0));pixels.show();}
}

void handle_OnConnect() {
  LED1status = LOW;
  LED2status = LOW;
  Serial.println("GPIO4 Status: OFF | GPIO5 Status: OFF");
  server.send(200, "text/html", SendHTML(LED1status,LED2status));
}

void handle_ledredon() {
  LED1status = HIGH;
  Serial.println("GPIO4 Status: ON");
  server.send(200, "text/html", SendHTML(true,LED2status));
}

void handle_ledredoff() {
  LED1status = LOW;
  Serial.println("GPIO4 Status: OFF");
  server.send(200, "text/html", SendHTML(false,LED2status));
}

void handle_ledgreenon() {
  LED2status = HIGH;
  Serial.println("GPIO5 Status: ON");
  server.send(200, "text/html", SendHTML(LED1status,true));
}

void handle_ledgreenoff() {
  LED2status = LOW;
  Serial.println("GPIO5 Status: OFF");
  server.send(200, "text/html", SendHTML(LED1status,false));
}

void handle_NotFound(){
  server.send(404, "text/plain", "Not found");
}

String SendHTML(uint8_t led1stat,uint8_t led2stat){
  String ptr = "<!DOCTYPE html> <html>\n";
  ptr +="<head><meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0, user-
scalable=no\">\n";
  ptr +="<title>LED Control</title>\n";
  ptr +="<style>html { font-family: Helvetica; display: inline-block; margin: 0px auto; text-align:
center;}\n";
  ptr +="body{margin-top: 50px;} h1 {color: #444444;margin: 50px auto 30px;} h3 {color:
#444444;margin-bottom: 50px;}\n";
  ptr +=".button {display: block;width: 80px;background-color: #3498db;border: none;color:
white;padding: 13px 30px;text-decoration: none;font-size: 25px;margin: 0px auto 35px;cursor:
pointer;border-radius: 4px;}\n";
  ptr +=".button-on {background-color: #3498db;}\n";
  ptr +=".button-on:active {background-color: #2980b9;}\n";
  ptr +=".button-off {background-color: #34495e;}\n";
  ptr +=".button-off:active {background-color: #2c3e50;}\n";
  ptr +="p {font-size: 14px;color: #888;margin-bottom: 10px;}\n";
  ptr +="</style>\n";
  ptr +="</head>\n";
  ptr +="<body>\n";
  ptr +="<h1>RISC-V Web Server</h1>\n";
  ptr +="<h3>Using Access Point(AP) Mode</h3>\n";

   if(led1stat)
  {ptr +="<p>LED_RED Status: ON</p><a class=\"button button-off\" href=\"/ledredoff\">OFF</a>\n";}
  else
  {ptr +="<p>LED_RED Status: OFF</p><a class=\"button button-on\" href=\"/ledredon\">ON</a>\n";}

  if(led2stat)
  {ptr +="<p>LED_GREEN Status: ON</p><a class=\"button button-off\" href=\"/ledgreenoff\">OFF</a>\
n";}
  else
  {ptr +="<p>LED_GREEN Status: OFF</p><a class=\"button button-on\" href=\"/ledgreenon\">ON</a>\n";}

  ptr +="</body>\n";
  ptr +="</html>\n";
  return ptr;
}
```

To do:

1. Display the **ssid** name and IP address on OLED screen.
2. Modify the content of the WEB page adding third button and LED color BLUE.

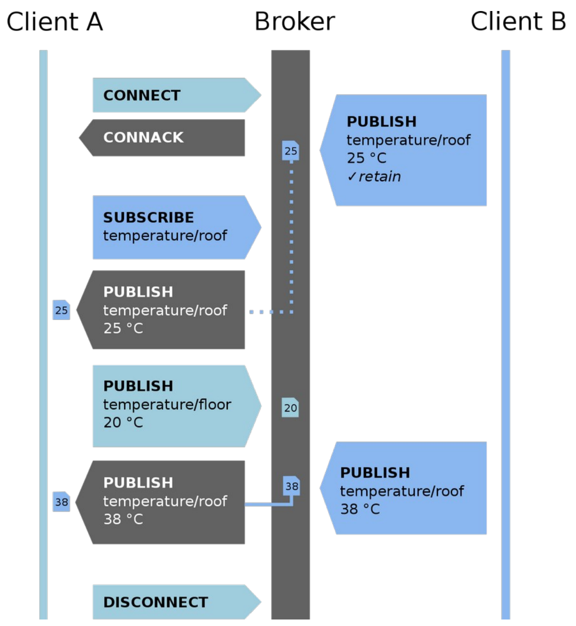
3.2 MQTT – broker and client

Message Queuing Telemetry Transport or **MQTT** is a lightweight, publish-subscribe network protocol that transports messages between devices. The protocol usually runs over TCP/IP; however, any network protocol that provides ordered, lossless, bi-directional connections can support **MQTT**. It is designed for connections with remote locations where a "small code footprint" is required or the network bandwidth is limited.
The **MQTT protocol** defines two types of network entities: a message broker and a number of clients. An MQTT broker is a server that receives all messages from the clients and then routes the messages to the appropriate destination clients. An MQTT client is any device (from a micro controller up to a fully-fledged server) that runs an MQTT library and connects to an MQTT broker over a network.

Information is organized in a hierarchy of **topics**. When a publisher has a new item of data to distribute, it sends a control message with the data to the connected broker. The broker then distributes the information to any clients that have subscribed to that topic. The publisher does not need to have any data on the number or locations of subscribers, and subscribers, in turn, do not have to be configured with any data about the publishers.
If a broker receives a message on a topic for which there are no current subscribers, the broker discards the message.
Each client that subscribes to a topic pattern that matches the topic of the retained message receives the retained message immediately after they subscribe. The broker stores only one retained message per topic. This allows new subscribers to a topic to receive the most current value rather than waiting for the next update from a publisher.
When a publishing client first connects to the broker, it can set up a default message to be sent to subscribers if the broker detects that the publishing client has unexpectedly disconnected from the broker.

A minimal MQTT control message can be **as little as two bytes of data**. A control message can carry nearly 256 megabytes of data if needed. There are fourteen defined message types used to connect and disconnect a client from a broker, to publish data, to acknowledge receipt of data, and to supervise the connection between client and server.
MQTT relies on the TCP protocol for data transmission. MQTT sends connection credentials in plain text format and does not include any measures for security or authentication. This can be provided by using TLS to encrypt and protect the transferred information against interception, modification or forgery.

The default **un-encrypted MQTT** port is **1883**. The **encrypted** port is **8883**.


Client A
Broker
Client B
CONNECT
CONNACK
PUBLISH temperature/roof 25 °C ✓retain
SUBSCRIBE temperature/roof
PUBLISH temperature/roof 25 °C
PUBLISH temperature/floor 20 °C
PUBLISH temperature/roof 38 °C
PUBLISH temperature/roof 38 °C
DISCONNECT
25
20
38

3.2.1 Basic MQTT broker code

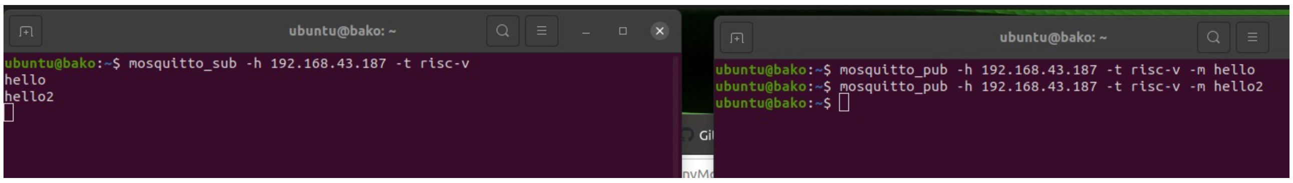
```
#include "TinyMqtt.h"   // https://github.com/hsaturn/TinyMqtt
//#include <my_credentials.h>
#define PORT 1883
//#include <my_credentials.h>
const char *ssid     = "PhoneAP";
const char *pass = "smartcomputerlab";

MqttBroker broker(PORT);


void setup()
{
  pinMode(LED_GREEN, OUTPUT);pinMode(LED_BLUE, OUTPUT);
    Serial.begin(9600);while(!Serial);Serial.println();
  // Set WiFi transmission power to 13dBm
  WiFi.setTxPower(WIFI_POWER_13dBm);
  delay(100);
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, pass);

  while (WiFi.status() != WL_CONNECTED) {
    Serial << '.';
    delay(500);
  }
  digitalWrite(LED_GREEN, HIGH);
  Serial << "Connected to " << ssid << "IP address: " << WiFi.localIP() << endl;
  broker.begin();
  Serial << "Broker ready : " << WiFi.localIP() << " on port " << PORT << endl;
}

void loop()
{
  broker.loop();
}
```

Execution result:

```
..Connected to PhoneAPIP address: 192.168.43.187
Broker ready : 192.168.43.187 on port 1883
```

The screen display of two terminals sending and receiving MQTT messages to our broker.

Note that the **computer host** (ubuntu@bako) is connected to the same wifi network.



```
ubuntu@bako:~$ mosquitto_sub -h 192.168.43.187 -t risc-v
hello
hello2
```

```
ubuntu@bako:~$ mosquitto_pub -h 192.168.43.187 -t risc-v -m hello
ubuntu@bako:~$ mosquitto_pub -h 192.168.43.187 -t risc-v -m hello2
ubuntu@bako:~$
```

To do:

1. Analyze and test the broker code.
2. Display the **obtained** IP address on OLED screen.

3.2.2 Simple MQTT client code with subscribe and publish operations

```
#include <WiFi.h>
#include <PubSubClient.h>
#include <Wire.h>
// Replace the next variables with your SSID/Password combination
const char *ssid     = "PhoneAP";
const char *pass = "smartcomputerlab";

// Add your MQTT Broker IP address, example:
const char* mqtt_server = "192.168.1.81"; // "YOUR_MQTT_BROKER_IP_ADDRESS";
WiFiClient espClient;
PubSubClient client(espClient);
long lastMsg = 0;
char msg[50];
int value = 0;
char *topic="risc-v/test";

void setup() {
  Serial.begin(9600);
  setup_wifi();
  client.setServer(mqtt_server, 1883);
  client.setCallback(callback);
}

void setup_wifi() {
  delay(10);
  // We start by connecting to a WiFi network
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, pass);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500); Serial.print(".");
  }
  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

void callback(char* topic, byte* message, unsigned int length) {
  Serial.print("Message arrived on topic: ");
  Serial.print(topic);
  Serial.print(". Message: ");
  String messageTemp;

  for (int i = 0; i < length; i++) {
    Serial.print((char)message[i]);
    messageTemp += (char)message[i];
  }
  Serial.println();
  // Feel free to add more if statements to control more GPIOs with MQTT

  if (String(topic) == "topic") {
    Serial.print("Changing output to ");
    if(messageTemp == "on"){
      Serial.println("on");
    }
    else if(messageTemp == "off"){
      Serial.println("off");
    }
  }
}

void reconnect() {
  // Loop until we're reconnected
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    // Attempt to connect
    if (client.connect("ESP32Client")) {
      Serial.println("connected");
      // Subscribe
      client.subscribe(topic);

    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      // Wait 5 seconds before retrying
      delay(5000);
    }
  }
}

int counter=0;
char buff[64];

void loop() {
  if (!client.connected()) {
    reconnect();
  }
  client.loop();
  long now = millis();
  if (now - lastMsg > 5000) {
    lastMsg = now;
    counter++;
    // Convert the value to a char array
    sprintf(buff,"Counter:%d",counter);
    Serial.printf("Publish:%s\n",buff);
    client.publish(topic, buff);

  }
}
```

On your Ubuntu host activate **mosquitto** broker:

```
ubuntu@bako:~$ sudo systemctl start mosquitto
```

The find the server host ip with: `ip a`

for example you get: `inet 192.168.43.243`

Use the obtained address in the above program I place of `"192.168.1.81"`

Execution result:

```
...
WiFi connected
IP address:
192.168.43.187
Attempting MQTT connection...connected
Publish:Counter:1
Message arrived on topic: risc-v/test. Message: Counter:1
Publish:Counter:2
Message arrived on topic: risc-v/test. Message: Counter:2
Publish:Counter:3
```

```
Message arrived on topic: risc-v/test. Message: Counter:3
Publish:Counter:4
Message arrived on topic: risc-v/test. Message: Counter:4
Publish:Counter:5
Message arrived on topic: risc-v/test. Message: Counter:5
Publish:Counter:6
..
```

To do:

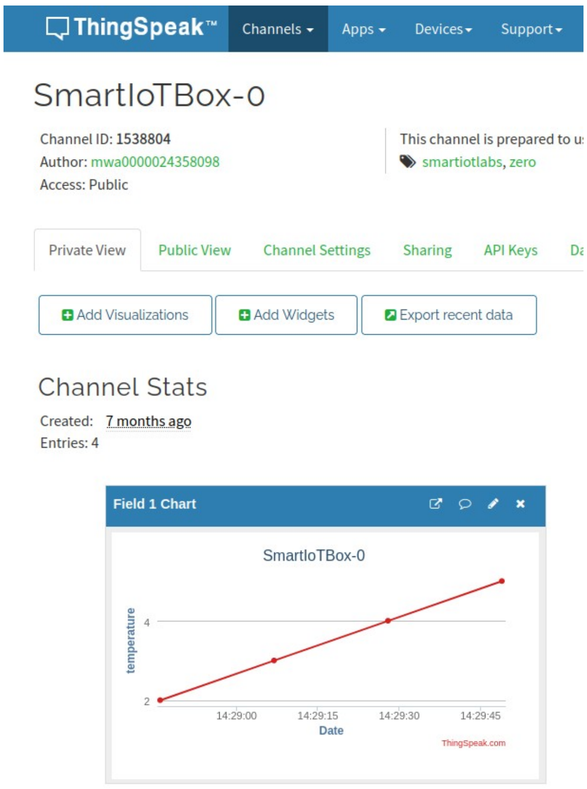
1. Use a **real sensor** (SHT21) and send an MQTT message with temperature and humidity values
2. Display the **receiver MQTT on OLED screen**

3.3 ThingSpeak IoT Server

The following are the programs that send the simple data to **ThingSpeak** server via a WiFi (TCP/HTTP) connection. **`WriteSingleField`** program writes a value to a channel on **ThingSpeak** every 20 seconds.
Note that the initial **`ThingSpeak.h`** file should be modified with your IP address and service port if you use your own **ThingSpeak** server.
Before using the following examples you have to obtain the **ThingSpeak account**.

3.3.1 Write single variable to ThingSpeak channel

There are up to 8 fields in a channel, allowing you to store up to 8 different variables.
Here, we write to **`field 1`**.



SmartIoTBox-0

Channel ID: 1538804
Author: mwa0000024358098
Access: Public

This channel is prepared to u
smartiotlabs, zero

Channel Stats

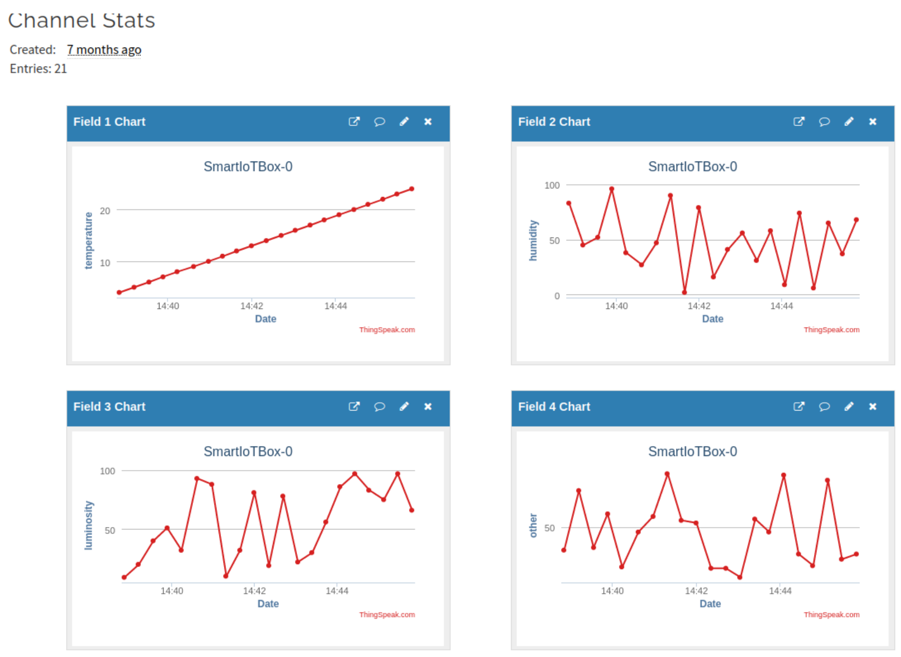
Created: 7 months ago
Entries: 4

```
#include <WiFi.h>
//#include "secrets.h"
#include "ThingSpeak.h"
const char* ssid     = "PhoneAP";
const char* password = "smartcomputerlab";
WiFiClient  client;
unsigned long myChannelNumber = 1538804;
const char * myWriteAPIKey = "4VZFZQSGA9L52B8X";
int number = 0;

void setup() {
    Serial.begin(9600);
    delay(10);
    Serial.println();
    Serial.println();
    Serial.print("Connecting to ");
    Serial.println(ssid);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);  Serial.print(".");
    }
    Serial.println("");
    Serial.println("WiFi connected");
    Serial.println("IP address: ");
    Serial.println(WiFi.localIP());
    delay(100);
  ThingSpeak.begin(client);  // Initialize ThingSpeak
}


void loop() {
  // Connect or reconnect to WiFi
    while(WiFi.status() != WL_CONNECTED){
      Serial.print(".");
      delay(1000);
    }
    Serial.println("\nConnected.");
  int x = ThingSpeak.writeField(myChannelNumber, 1, number, myWriteAPIKey);
  if(x == 200){
    Serial.println("Channel update successful.");
  }
  else{
    Serial.println("Problem updating channel. HTTP error code " + String(x));
  }
  // change the value
  number++;
  if(number > 99){
    number = 0;
  }
  delay(20000); // Wait 20 seconds to update the channel again
}
```

3.3.2 Write multiple variables to ThingSpeak channel

`WriteMultipleFields` writes values to fields **1,2,3,4** in **a single ThingSpeak update** every 20 seconds.

```
#include <WiFi.h>
#include "secrets.h"
#include "ThingSpeak.h"
const char* ssid     = "PhoneAP";
const char* password = "smartcomputerlab";
WiFiClient  client;
unsigned long myChannelNumber = 1538804;
const char * myWriteAPIKey = "4VZFZQSGA9L52B8X";
// Initialize our values
int number1 = 0;
int number2 = random(0,100);
int number3 = random(0,100);
int number4 = random(0,100);
String myStatus = "";

void setup() {
    Serial.begin(9600);
    delay(10);
    Serial.println();
    Serial.println();
    Serial.print("Connecting to ");
    Serial.println(ssid);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println("");
    Serial.println("WiFi connected");
    Serial.println("IP address: ");
    Serial.println(WiFi.localIP());
    delay(100);
  ThingSpeak.begin(client);  // Initialize ThingSpeak
}

void loop() {
  // Connect or reconnect to WiFi
    while(WiFi.status() != WL_CONNECTED){
      Serial.print(".");
      delay(5000);
    }
    Serial.println("\nConnected.");
  // set the fields with the values
  ThingSpeak.setField(1, number1);
  ThingSpeak.setField(2, number2);
  ThingSpeak.setField(3, number3);
  ThingSpeak.setField(4, number4);
  // write to the ThingSpeak channel
  int x = ThingSpeak.writeFields(myChannelNumber, myWriteAPIKey);
  if(x == 200){
    Serial.println("Channel update successful.");
  }
  else{
    Serial.println("Problem updating channel. HTTP error code " + String(x));
  }
  number1++;
  if(number1 > 99){
    number1 = 0;
  }
  number2 = random(0,100);
  number3 = random(0,100);
  number4 = random(0,100);

  delay(20000); // Wait 20 seconds to update the channel again
}
```


Channel Stats

Created: 7 months ago

Entries: 21

To do:

1. Activate your ThingSpeak account and retrieve the **channel number** and **write/read API keys**
2. Use a **real sensor** (SHT21) and send a message with temperature and humidity values to 2 fields

3.3.3 Read single variable (`field`) from ThingSpeak channel

`ReadField` demonstrates reading from a **`private`** channel which requires **`read`** API key

```
#include <WiFi.h>
#include "secrets.h"
#include "ThingSpeak.h"
const char* ssid     = "PhoneAP";
const char* password = "smartcomputerlab";
WiFiClient  client;
unsigned long myChannelNumber = 1538804;
const char * myReadAPIKey = "V7M6I771U8OJ2JSV";

void setup() {
    Serial.begin(9600);
    delay(10);
    Serial.println();
    Serial.println();
    Serial.print("Connecting to ");
    Serial.println(ssid);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println("");
    Serial.println("WiFi connected");
    Serial.println("IP address: ");
    Serial.println(WiFi.localIP());
    delay(100);
  ThingSpeak.begin(client);  // Initialize ThingSpeak
}

int statusCode=0;

void loop() {
   // Connect or reconnect to WiFi
    while(WiFi.status() != WL_CONNECTED){
      Serial.print(".");  delay(1000);
    }
    Serial.println("\nConnected.");
  // Read in field 2 of the private channel
  int counter1 = ThingSpeak.readIntField(myChannelNumber, 2,myReadAPIKey);
  // Check the status of the read operation to see if it was successful
  statusCode = ThingSpeak.getLastReadStatus();
  if(statusCode == 200){
    Serial.println("Counter1: " + String(counter1));
  }
  else{
    Serial.println("Problem reading channel. HTTP error code " + String(statusCode));
  }
  delay(15000); // No need to read the temperature too often.
}
```

Execution result:

```
Connecting to PhoneAP
..
WiFi connected
IP address:
192.168.43.187

Connected.
Counter1: 45

Connected.
Counter1: 45

..
```

3.2.4 Read multiple variables (`field`) from ThingSpeak channel

`ReadMultipleFields` demonstrates reading from a private channel which requires API key and created-at timestamp associated with the latest feed.

```
#include <WiFi.h>
#include "secrets.h"
#include "ThingSpeak.h"
const char* ssid     = "PhoneAP";
const char* password = "smartcomputerlab";
WiFiClient  client;
unsigned long myChannelNumber = 174;
const char * myReadAPIKey = "V7M6I771U8OJ2JSV";
int number = 0;

void setup() {
    Serial.begin(9600);
    delay(10);
    Serial.println();
    Serial.println();
    Serial.print("Connecting to ");
    Serial.println(ssid);
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println("");
    Serial.println("WiFi connected");
    Serial.println("IP address: ");
    Serial.println(WiFi.localIP());
    delay(100);
  ThingSpeak.begin(client);  // Initialize ThingSpeak
}

int field[8] = {1,2,3,4,5,6,7,8};
// Initialize our values
int number1,number2,number3,number4;

void loop() {
   // Connect or reconnect to WiFi
```

```
    while(WiFi.status() != WL_CONNECTED){
      Serial.print(".");   delay(1000);
    }
    Serial.println("\nConnected.");
    int statusCode = ThingSpeak.readMultipleFields(myChannelNumber,myReadAPIKey);
    if(statusCode == 200)
    {
     number1 = ThingSpeak.getFieldAsInt(field[0]); // Field 1
     number2 = ThingSpeak.getFieldAsInt(field[1]); // Field 1
     number3 = ThingSpeak.getFieldAsInt(field[2]); // Field 1
     number4 = ThingSpeak.getFieldAsInt(field[3]); // Field 1
     String createdAt = ThingSpeak.getCreatedAt(); // Created-at timestamp
     Serial.println(number1);Serial.println(number2);
     Serial.println(number3);Serial.println(number4);
     Serial.println(createdAt);
    }
    else{
      Serial.println("Problem reading channel. HTTP error code " + String(statusCode));
    }
    Serial.println();
    delay(5000);
}
```

Execution result:

```
Connecting to PhoneAP
...
WiFi connected
IP address:
192.168.43.187

Connected.
48
45
99
54
2022-05-24T12:54:09Z


Connected.
48
45
99
54
2022-05-24T12:54:09Z
```

To do:

1. Activate your **ThingSpeak** account and retrieve the **channel number** and **write/read API keys**
2. Use an OLED display to show the received values

Lab 4

Simple IoT Gateways

In this lab we are going to build simple IoT gateways that relay the LoRa packets to MQTT broker or to ThingSpeak server.

4.1 LoRa to MQTT broker gateways

First let us take our LoRaSender from point – **2.5.2.1**. We are going to modify it by adding the `union` construct with the MQTT packet structure.

```
typedef union
{
uint8_t  buff[64];   // total data -64 bytes
struct
       {
       char topic[32];  // MQTT topic
       char mess[32];   // MQTT message
       } data;
} mqtt_pack;
```

4.1.1 Adapted version of LoRaSender with MQTT packets

```
#include <SPI.h>
#include <LoRa.h>
// connection lines - SPI bus
#define SCK     6   // GPIO18 -- SX127x's SCK
#define MISO    0   // GPIO19 -- SX127x's MISO
#define MOSI    1   // GPIO23 -- SX127x's MOSI
#define SS     10   // GPIO05 -- SX127x's CS
#define RST     4   // GPIO15 -- SX127x's RESET
#define DI0     5   // GPIO26 -- SX127x's IRQ(Interrupt Request)
// radio parameters
#define freq    434E6
#define sf 7
#define sb 125E3

typedef union
{
uint8_t  buff[64];   // total data -64 bytes
struct
       {
       char topic[32];  // MQTT topic
       char mess[32];   // MQTT message
       } data;
} mqttpack_t;

mqttpack_t spack;

void setup() {
  Serial.begin(9600);
  SPI.begin(SCK,MISO,MOSI,SS);
  LoRa.setPins(SS,RST,DI0);
  Serial.println();delay(100);Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
Serial.println("Starting LoRa OK!");
LoRa.setSpreadingFactor(sf);
LoRa.setSignalBandwidth(sb);
}

strcpy(spack.data.topic,”risc-v/test”);

float d1=0.1, d2=0.2;
```

```
void loop()  // la boucle de l'emetteur
{
Serial.print("New Packet:") ;
  LoRa.beginPacket();                    // start packet
  sprintf(spack.data.mess,"T:%2.2f,H:%2.2f",d1,d2)
  LoRa.write(spack.buff,64);
  LoRa.endPacket();
  Serial.println(spack.data.mess);
  d1=d1+0.1; d2=d2+0.2;
  delay(6000);
}
```

4.1.2 Adapted version of LoRaReceiver with MQTT packets

```
#include <SPI.h>
#include <LoRa.h>
#define SCK     6   // GPIO18 -- SX127x's SCK
#define MISO    0 //7   // GPIO19 -- SX127x's MISO
#define MOSI    1 //8   // GPIO23 -- SX127x's MOSI
#define SS      10   // GPIO05 -- SX127x's CS
#define RST     4   // GPIO15 -- SX127x's RESET
#define DI0     5   // GPIO26 -- SX127x's IRQ(Interrupt Request)
#define freq    434E6
#define sf 7
#define sb 125E3

typedef union
{
uint8_t  buff[64];   // total data -64 bytes
struct
  {
  char topic[32];  // MQTT topic
  char mess[32];   // MQTT message
  } data;
} mqttpack_t;

mqttpack_t rpack;

void setup() {
  Serial.begin(9600);
  SPI.begin(SCK,MISO,MOSI,SS);
  LoRa.setPins(SS,RST,DI0);
  Serial.println();delay(100);Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
Serial.println("Starting LoRa OK!");
LoRa.setSpreadingFactor(sf);
LoRa.setSignalBandwidth(sb);
}

char topic[32],mess[32];
int rssi;

void loop()
{
int packetLen;
packetLen=LoRa.parsePacket();
if(packetLen==64)
  {
  int i=0;
  while (LoRa.available()) {
    rpack.buff[i]=LoRa.read();i++;
    }
  strcpy(topic,rpack.data.topic);strcpy(mess,rpack.data.mess);
  rssi=LoRa.packetRssi();  // force du signal en réception en dB
  Serial.printf("Topic=%s,Message=%s\n",topic,mess);
  Serial.printf("RSSI=%d\n",rssi);
  }
}
```

4.1.3 Simple publisher of MQTT packets via WiFi connection

```
#include <WiFi.h>
#include <PubSubClient.h>
#include <Wire.h>
// Replace the next variables with your SSID/Password combination
const char *ssid     = "PhoneAP";
const char *pass = "smartcomputerlab";
// Add your MQTT Broker IP address, example:
const char* mqtt_server = "broker.emqx.io";  // "YOUR_MQTT_BROKER_IP_ADDRESS";

WiFiClient espClient;
PubSubClient client(espClient);
long lastMsg = 0;
char msg[50];
int value = 0;
char *topic="risc-v/test";

void setup_wifi() {
  delay(10);
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, pass);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500); Serial.print(".");
  }
  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

void setup() {
  Serial.begin(9600);
  setup_wifi();
  client.setServer(mqtt_server, 1883);
}

void reconnect() {
  // Loop until we're reconnected
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    // Attempt to connect
    if (client.connect("PommePi Client")) {
      Serial.println("connected");
      // Subscribe
      client.subscribe(topic);

    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      // Wait 5 seconds before retrying
      delay(5000);
    }
  }
}

int counter=0;
char buff[64];

void loop() {
  if (!client.connected()) {
    reconnect();
  }
  long now = millis();
  if (now - lastMsg > 5000) {
    lastMsg = now;
    counter++;
    // Convert the value to a char array
    sprintf(buff,"Counter:%d",counter);
    Serial.printf("Publish:%s\n",buff);
    client.publish(topic, buff);
  }
}
```

4.1.3 Simple subscriber of MQTT packets via WiFi connection

```
#include <WiFi.h>
#include <PubSubClient.h>
#include <Wire.h>
const char *ssid     = "PhoneAP";
const char *pass = "smartcomputerlab";
//const char* mqtt_server = "192.168.43.243"; // "YOUR_MQTT_BROKER_IP_ADDRESS";
const char* mqtt_server = "broker.emqx.io";
WiFiClient espClient;
PubSubClient client(espClient);
long lastMsg = 0;
char msg[50];
int value = 0;
char *topic="risc-v/test";

void setup() {
  Serial.begin(9600);
  setup_wifi();
  client.setServer(mqtt_server, 1883);
  client.setCallback(callback);
}

void setup_wifi() {
  delay(10);
  Serial.println();Serial.print("Connecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, pass);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500); Serial.print(".");
  }
  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

void callback(char* topic, byte* message, unsigned int length)
{
  Serial.print("Message arrived on topic: ");
  Serial.print(topic);
  Serial.print(". Message: ");
  for (int i = 0; i < length; i++) {
    Serial.print((char)message[i]);
  }
  Serial.println();
}

void reconnect() {
  // Loop until we're reconnected
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    // Attempt to connect
    if (client.connect("PommePiClient")) {
      Serial.println("connected");
      // Subscribe
      client.subscribe(topic);
    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      // Wait 5 seconds before retrying
      delay(5000);
    }
  }
}

int counter=0;
char buff[64];

void loop()
{
  if (!client.connected()) reconnect();
  client.loop();
}
```

4.1.4 Simple upload MQTT gateway (LoRa to WiFi MQTT)

```
#include <WiFi.h>
#include <PubSubClient.h>
#include <Wire.h>
#include <SPI.h>
#include <LoRa.h>
#define SCK     6   // GPIO18 -- SX127x's SCK
#define MISO    0 //7   // GPIO19 -- SX127x's MISO
#define MOSI    1 //8   // GPIO23 -- SX127x's MOSI
#define SS      10   // GPIO05 -- SX127x's CS
#define RST     4   // GPIO15 -- SX127x's RESET
#define DI0     5   // GPIO26 -- SX127x's IRQ(Interrupt Request)
#define freq    434E6
#define sf 7
#define sb 125E3
// Replace the next variables with your SSID/Password combination
const char *ssid     = "PhoneAP";
const char *pass = "smartcomputerlab";

// Add your MQTT Broker IP address, example:
const char* mqtt_server = "broker.emqx.io";  // "192.168.43.243"; // "YOUR_MQTT_BROKER_IP_ADDRESS";

WiFiClient espClient;
PubSubClient client(espClient);
//long lastMsg = 0;
//char msg[50];
//int value = 0;
char ttop[32];
char *top="risc-v/test";

typedef union
{
uint8_t  buff[64];   // total data -64 bytes
struct
  {
  char topic[32];  // MQTT topic
  char mess[32];   // MQTT message
  } data;
} mqttpack_t;

mqttpack_t rpack;

void setup_wifi() {
  delay(10);
  // We start by connecting to a WiFi network251
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, pass);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500); Serial.print(".");
  }
  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

void setup_LoRa()
{
  SPI.begin(SCK,MISO,MOSI,SS);
  LoRa.setPins(SS,RST,DI0);
  Serial.println();delay(100);Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
Serial.println("Starting LoRa OK!");
LoRa.setSpreadingFactor(sf);
LoRa.setSignalBandwidth(sb);
}
```

```
void setup() {
  Serial.begin(9600);
  setup_LoRa(); delay(400);
  setup_wifi();
  client.setServer(mqtt_server, 1883);
  //client.setCallback(callback);
  strcpy(ttop,"risc-v/test");
}


void reconnect() {
  // Loop until we're reconnected
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    // Attempt to connect
    if (client.connect("PommePi")) {
      Serial.println("connected");
      // Subscribe
      client.subscribe(ttop);

    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      // Wait 5 seconds before retrying
      delay(5000);
    }
  }
}

char topic[32],mess[32];
int rssi;

void loop()
{
int packetLen;
packetLen=LoRa.parsePacket();
if(packetLen==64)
  {
  int i=0;
  while (LoRa.available())
    {
    rpack.buff[i]=LoRa.read();i++;
    }
  strcpy(topic,rpack.data.topic);strcpy(mess,rpack.data.mess);
  rssi=LoRa.packetRssi();  // force du signal en réception en dB
  Serial.printf("Topic=%s,Message=%s\n",topic,mess);
  Serial.printf("RSSI=%d\n",rssi);
  if (!client.connected()) { strcpy(ttop,topic);reconnect(); }
  client.publish(topic, mess);
  Serial.println("published");
  }
}
```

4.1.5 Simple download MQTT gateway (MQTT WiFi to LoRa)

```
#include <WiFi.h>
#include <PubSubClient.h>
#include <Wire.h>
#include <SPI.h>
#include <LoRa.h>
#define SCK     6   // SX127x's SCK
#define MISO    0   // SX127x's MISO
#define MOSI    1   // SX127x's MOSI
#define SS     10   // SX127x's CS
#define RST     4   // SX127x's RESET
#define DI0     5   // SX127x's IRQ(Interrupt Request)
#define freq    434E6
#define sf 7
#define sb 125E3
// Replace the next variables with your SSID/Password combination
const char *ssid     = "PhoneAP";
const char *pass = "smartcomputerlab";

// Add your MQTT Broker IP address, example:
//const char* mqtt_server = "192.168.43.243"; // "YOUR_MQTT_BROKER_IP_ADDRESS";
const char* mqtt_server = "broker.emqx.io";
WiFiClient espClient;
PubSubClient client(espClient);
long lastMsg = 0;
char msg[50];
int value = 0;
char *topic="risc-v/test";

typedef union
{
uint8_t  buff[64];   // total data -64 bytes
struct
  {
  char topic[32];  // MQTT topic
  char mess[32];   // MQTT message
  } data;
} mqttpack_t;

mqttpack_t spack;



void setup_wifi() {
  delay(10);
  // We start by connecting to a WiFi network
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, pass);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500); Serial.print(".");
  }
  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

void setup_LoRa()
{
  SPI.begin(SCK,MISO,MOSI,SS);
  LoRa.setPins(SS,RST,DI0);
  Serial.println();delay(100);Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
Serial.println("Starting LoRa OK!");
LoRa.setSpreadingFactor(sf);
LoRa.setSignalBandwidth(sb);
}

void callback(char* topic, byte* message, unsigned int length)
{
  char mess[32];
  Serial.print("Message arrived on topic: ");
  Serial.print(topic);
  Serial.print(". Message: ");
  for (int i = 0; i < length; i++)
    {
    mess[i]=(char)message[i];
    Serial.print((char)message[i]);
    }
  Serial.println();
  LoRa.beginPacket();                   // start packet
  strcpy(spack.data.mess,mess); strcpy(spack.data.topic,topic);
  LoRa.write(spack.buff,64);
  LoRa.endPacket();
  Serial.println("Packet sent");
}
```

```
void reconnect() {
  // Loop until we're reconnected
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    // Attempt to connect
    if (client.connect("PommePiClient")) {
      Serial.println("connected");
      // Subscribe
      client.subscribe(topic);

    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      // Wait 5 seconds before retrying
      delay(5000);
    }
  }
}

void setup() {
  Serial.begin(9600);
  setup_LoRa();delay(400);
  setup_wifi();
  client.setServer(mqtt_server, 1883);
  client.setCallback(callback);
}

int counter=0;
char buff[64];

void loop()
{
  if (!client.connected()) {
    reconnect();
  }
  client.loop();
}
```

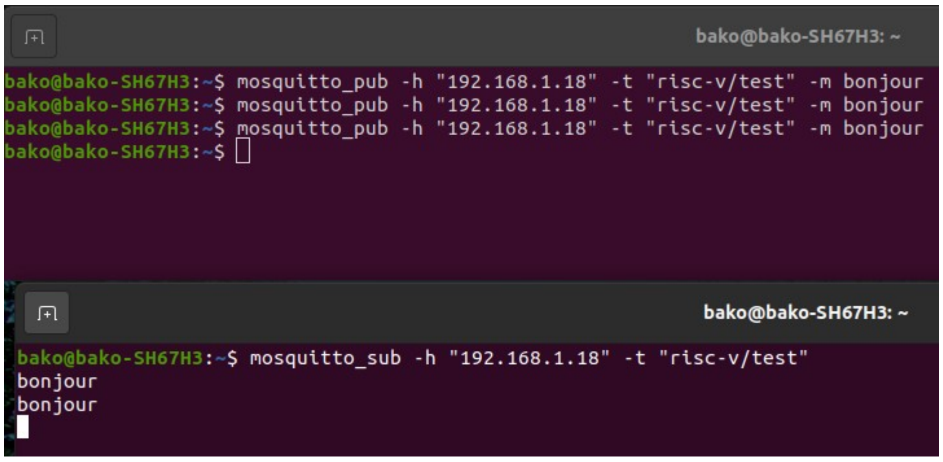
4.1.6 Using Pomme-Pi based MQTT broker

The following is a simple MQTT broker code using **`TinyMqtt`** library.

```
#include <Adafruit_NeoPixel.h>
#define PIN        2
#define NUMPIXELS  1
#include "TinyMqtt.h"   // https://github.com/hsaturn/TinyMqtt
//#include <my_credentials.h>
#define PORT 1883
//#include <my_credentials.h>
const char *ssid     = "PhoneAP";
const char *pass = "smartcomputerlab";

Adafruit_NeoPixel pixels(NUMPIXELS, PIN, NEO_GRB + NEO_KHZ800);

MqttBroker broker(PORT);

void setup()
{
  pixels.begin(); // INITIALIZE NeoPixel strip object (REQUIRED)
  pixels.clear(); // Set all pixel colors to 'off'
  Serial.begin(9600);while(!Serial);Serial.println();
  // Set WiFi transmission power to 13dBm
  WiFi.setTxPower(WIFI_POWER_13dBm);
  delay(100);
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, pass);

  while (WiFi.status() != WL_CONNECTED) {
    Serial << '.';
    delay(500);
  }
  pixels.setPixelColor(0, pixels.Color(0, 150, 0));pixels.show();
  Serial << "Connected to " << ssid << "IP address: " << WiFi.localIP() << endl;
  broker.begin();
  Serial << "Broker ready : " << WiFi.localIP() << " on port " << PORT << endl;
  pixels.setPixelColor(0, pixels.Color(0, 150, 0));pixels.show();
   pixels.setPixelColor(0, pixels.Color(0, 0, 150));pixels.show();
}
void loop()
{
  broker.loop();
}
```

Execution result (terminal):

```
..Connected to Livebox-08B0IP address: 192.168.1.18
Broker ready : 192.168.1.18 on port 1883
Connected client:mosq-TZcOXaNUx4sn5kkNSk, keep alive=60.
Connected client:mosq-VuGiR5StAGNxbBGids, keep alive=60.
Connected client:mosq-QUmeBXFGjThK9E3bMi, keep alive=60.
Connected client:mosq-jVYzUGuqLcS3Swc9nI, keep alive=60.
```



```
bako@bako-SH67H3: ~
bako@bako-SH67H3:~$ mosquitto_pub -h "192.168.1.18" -t "risc-v/test" -m bonjour
bako@bako-SH67H3:~$ mosquitto_pub -h "192.168.1.18" -t "risc-v/test" -m bonjour
bako@bako-SH67H3:~$ mosquitto_pub -h "192.168.1.18" -t "risc-v/test" -m bonjour
bako@bako-SH67H3:~$
```

```
bako@bako-SH67H3: ~
bako@bako-SH67H3:~$ mosquitto_sub -h "192.168.1.18" -t "risc-v/test"
bonjour
bonjour
```

Fig 4.1 Operation of mosquitto with our tiny MQTT broker

Note:
The `mosquitto_pub/sub` clients are operating on the same private network as the Pomme-Pi **MQTT** tiny **broker**.

4.2 LoRa to ThingSpeak server gateways

The **LoRa-ThingSpeak server** gateways operate in a similar way as LoRa-MQTT broker gateways, however instead of topics and text messages we have **channel numbers**, **read/write keys** and **channel fields**.

```
typedef union
{
uint8_t  buff[40];   // 40 or 72 bytes
struct
  {
  uint8_t head[4];  // packet header: address, control, ..
  char key[16];  // read or write APi key
  int chnum;     // channel number
  float sensor[4];  // or sensor[8]
  } data;
} tspack_t;
```

4.2.1 Sending LoRa-TS packet to LoRa-WiFi (ThingSpeak) gateway

```
#include <SPI.h>
#include <LoRa.h>
// connection lines - SPI bus
#define SCK     6   // GPIO18 -- SX127x's SCK
#define MISO    0   // GPIO19 -- SX127x's MISO
#define MOSI    1   // GPIO23 -- SX127x's MOSI
#define SS     10   // GPIO05 -- SX127x's CS
#define RST     4   // GPIO15 -- SX127x's RESET
#define DI0     5   // GPIO26 -- SX127x's IRQ(Interrupt Request)
// radio parameters
#define freq    434E6
#define sf 7
#define sb 125E3

#define wkey "3IN09682SQX3PT4Z"
#define channel 1626377

typedef union
{
uint8_t  buff[40];   // 40 or 72 bytes
struct
  {
  uint8_t head[4];  // packet header: address, control, ..
  char key[16];  // read or write APi key
  int chnum;     // channel number
  float sensor[4];  // or sensor[8]
  } data;
} tspack_t;

tspack_t spack;   // sending packet

void setup() {
  Serial.begin(9600);
  SPI.begin(SCK,MISO,MOSI,SS);
  LoRa.setPins(SS,RST,DI0);
  Serial.println();delay(100);Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  Serial.println("Starting LoRa OK!");
  LoRa.setSpreadingFactor(sf);
  LoRa.setSignalBandwidth(sb);
  spack.data.head[0]=0xFF;spack.data.head[1]=0x01;
  spack.data.head[2]=0x00;spack.data.head[3]=0x00;
  strncpy(spack.data.key,wkey,16);
  spack.data.chnum=channel;
}
```

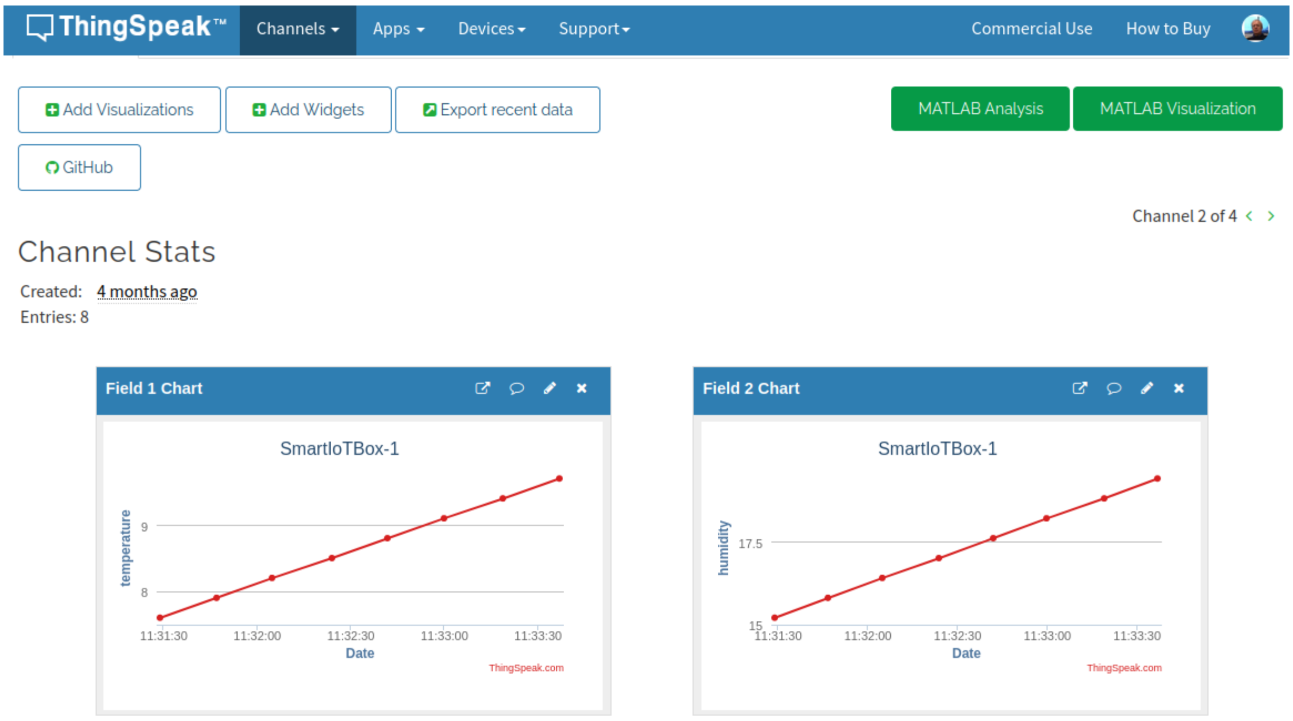
```
float s1=0.1, s2=0.2;

void loop()  // la boucle de l'emetteur
{
Serial.println("New Packet:") ;
  LoRa.beginPacket();                   // start packet
  spack.data.sensor[0]=s1;spack.data.sensor[1]=s2;
  LoRa.write(spack.buff,40);
  LoRa.endPacket();
  Serial.print(s1);Serial.print("  ");Serial.println(s2);
  s1=s1+0.1; s2=s2+0.2;
  delay(6000);
}
```

4.2.2 LoRa-WiFi (ThingSpeak) sender gateway

```
#include <WiFi.h>
#include "ThingSpeak.h"
#include <Wire.h>
#include <SPI.h>
#include <LoRa.h>
#define SCK     6   // GPIO18 -- SX127x's SCK
#define MISO    0 //7   // GPIO19 -- SX127x's MISO
#define MOSI    1 //8   // GPIO23 -- SX127x's MOSI
#define SS      10   // GPIO05 -- SX127x's CS
#define RST     4   // GPIO15 -- SX127x's RESET
#define DI0     5   // GPIO26 -- SX127x's IRQ(Interrupt Request)
#define freq    434E6
#define sf 7
#define sb 125E3
const char *ssid     = "PhoneAP";
const char *pass = "smartcomputerlab";
WiFiClient  client;
char wkey[17];  // wkey[16] to store NULL character
int channel;

typedef union
{
uint8_t  buff[40];   // 40 or 72 bytes
struct
  {
  uint8_t head[4];  // packet header: address, control, ..
  char key[16];  // read or write APi key
  int chnum;     // channel number
  float sensor[4];  // or sensor[8]
  } data;
} tspack_t;

tspack_t rpack;   // sending packet

void setup_wifi() {
  delay(10);
  // We start by connecting to a WiFi network251
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, pass);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500); Serial.print(".");
  }
  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
  ThingSpeak.begin(client);  // Initialize ThingSpeak
}

void setup_LoRa()
{
  SPI.begin(SCK,MISO,MOSI,SS);
  LoRa.setPins(SS,RST,DI0);
  Serial.println();delay(100);Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
Serial.println("Starting LoRa OK!");
LoRa.setSpreadingFactor(sf);
LoRa.setSignalBandwidth(sb);
}

void setup() {
  Serial.begin(9600);
  setup_LoRa(); delay(400);
  setup_wifi();
}

int rssi;

void loop()
{
int packetLen;
packetLen=LoRa.parsePacket();
if(packetLen==40)
  {
  int i=0;
  while (LoRa.available())
    {
    rpack.buff[i]=LoRa.read();i++;
    }
  strncpy(wkey,rpack.data.key,16);wkey[16]=0x00;channel=rpack.data.chnum;
  Serial.println(wkey);Serial.println(channel);
  rssi=LoRa.packetRssi();  // force du signal en réception en dB
  ThingSpeak.setField(1, rpack.data.sensor[0]);
  ThingSpeak.setField(2, rpack.data.sensor[1]);
  ThingSpeak.setField(3, rpack.data.sensor[2]);
  ThingSpeak.setField(4, rpack.data.sensor[3]);
// write to the ThingSpeak channel
  int x = ThingSpeak.writeFields(channel, wkey);
  if(x == 200){
    Serial.println("Channel update successful.");
  }
  else{
    Serial.println("Problem updating channel. HTTP error code " + String(x));
  }
  Serial.println("Packet sent to TS");
  delay(15000);
  }
}
```

Execution example:

```
Starting LoRa OK!

Connecting to Livebox-08B0
..
WiFi connected
IP address:
192.168.1.20
3IN09682SQX3PT4Z
1626377
Channel update successful.
Packet sent to TS
3IN09682SQX3PT4Z
1626377
Channel update successful.
Packet sent to TS
3IN09682SQX3PT4Z
1626377
Channel update successful.
Packet sent to TS
3IN09682SQX3PT4Z
1626377
Channel update successful.
Packet sent to TS
3IN09682SQX3PT4Z
1626377
..
```



Channel Stats

Created: 4 months ago

Entries: 8

Fig 4.2 ThingSpeak channel diagrams for the received LoRa packets with 2 sensor values

4.2.3 LoRa receiver gateway for TS packets

```
#include <WiFi.h>
#include "ThingSpeak.h"
#include <Wire.h>
#include <SPI.h>
#include <LoRa.h>
#define SCK     6   // GPIO18 -- SX127x's SCK
#define MISO    0 //7   // GPIO19 -- SX127x's MISO
#define MOSI    1 //8   // GPIO23 -- SX127x's MOSI
#define SS      10   // GPIO05 -- SX127x's CS
#define RST     4   // GPIO15 -- SX127x's RESET
#define DI0     5   // GPIO26 -- SX127x's IRQ(Interrupt Request)
#define freq    434E6
#define sf 7
#define sb 125E3
const char *ssid     = "PhoneAP";
const char *pass = "smartcomputerlab";
WiFiClient  client;
unsigned long myChannelNumber = 1626377;
const char * myReadAPIKey = "9JVTP8ZHVTB9G4TT";
int number = 0;
typedef union
{
uint8_t  buff[40];   // 40 or 72 bytes
struct
  {
  uint8_t head[4];  // packet header: address, control, ..
  char key[16];  // read or write APi key
  int chnum;     // channel number
  float sensor[4];  // or sensor[8]
  } data;
} tspack_t;
```

```
tspack_t spack;   // sending packet

void setup_LoRa()
{
  SPI.begin(SCK,MISO,MOSI,SS);
  LoRa.setPins(SS,RST,DI0);
  Serial.println();delay(100);Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
Serial.println("Starting LoRa OK!");
LoRa.setSpreadingFactor(sf);
LoRa.setSignalBandwidth(sb);
}

void setup_wifi() {
  delay(10);
  // We start by connecting to a WiFi network251
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, pass);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500); Serial.print(".");
  }
  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
  ThingSpeak.begin(client);  // Initialize ThingSpeak
}

void setup() {
  Serial.begin(9600);
  setup_LoRa(); delay(400);
  setup_wifi();
}

int field[8] = {1,2,3,4,5,6,7,8};
// Initialize our values
float s1,s2,s3,s4;

void loop() {
   // Connect or reconnect to WiFi
    while(WiFi.status() != WL_CONNECTED){
      Serial.print(".");   delay(1000);
    }
    Serial.println("\nConnected.");
    int statusCode = ThingSpeak.readMultipleFields(myChannelNumber,myReadAPIKey);
    if(statusCode == 200)
    {
     s1 = ThingSpeak.getFieldAsFloat(field[0]);
     s2 = ThingSpeak.getFieldAsFloat(field[1]);
     s3 = ThingSpeak.getFieldAsFloat(field[2]);
     s4 = ThingSpeak.getFieldAsFloat(field[3]);
     String createdAt = ThingSpeak.getCreatedAt(); // Created-at timestamp
     Serial.println(s1);Serial.println(s2);
     Serial.println(createdAt);
     LoRa.beginPacket(); // start packet
     spack.data.head[0]=0x01;     // first client terminal
     spack.data.sensor[0]=s1;spack.data.sensor[1]=s2;
     LoRa.write(spack.buff,40);
     LoRa.endPacket();

    }
    else{
      Serial.println("Problem reading channel. HTTP error code " + String(statusCode));
    }
    Serial.println();
    delay(20000);
}
```

4.2.4 LoRa receiver termainal for TS packets

```
#include <SPI.h>
#include <LoRa.h>
// connection lines - SPI bus
#define SCK     6   // GPIO18 -- SX127x's SCK
#define MISO    0   // GPIO19 -- SX127x's MISO
#define MOSI    1   // GPIO23 -- SX127x's MOSI
#define SS     10   // GPIO05 -- SX127x's CS
#define RST     4   // GPIO15 -- SX127x's RESET
#define DI0     5   // GPIO26 -- SX127x's IRQ(Interrupt Request)
// radio parameters
#define freq    434E6
#define sf 7
#define sb 125E3
#define wkey "3IN09682SQX3PT4Z"
#define channel 1626377

typedef union
{
uint8_t  buff[40];   // 40 or 72 bytes
struct
  {
  uint8_t head[4];  // packet header: address, control, ..
  char key[16];  // read or write APi key
  int chnum;     // channel number
  float sensor[4];  // or sensor[8]
  } data;
} tspack_t;

tspack_t rpack;   // sending packet

void setup() {
  Serial.begin(9600);
  SPI.begin(SCK,MISO,MOSI,SS);
  LoRa.setPins(SS,RST,DI0);
  Serial.println();delay(100);Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  Serial.println("Starting LoRa OK!");
  LoRa.setSpreadingFactor(sf);
  LoRa.setSignalBandwidth(sb);
}

float s1=0.1, s2=0.2;
int rssi;

void loop()  // la boucle de l’emetteur
{
int packetLen;
packetLen=LoRa.parsePacket();
if(packetLen==40)
  {
  int i=0;
  Serial.println("packet recv");
  while (LoRa.available()) {
    rpack.buff[i]=LoRa.read();i++;
    }
  if(rpack.data.head[0]==0x01)
    {
    s1=rpack.data.sensor[0]; s2=rpack.data.sensor[1];
    rssi=LoRa.packetRssi();  // force du signal en réception en dB
    Serial.printf("T=%2.2f,H=%2.2f\n",s1,s2);
    Serial.printf("RSSI=%d\n",rssi);
    }
  }
}
```

```
Starting LoRa OK!
packet recv
T=21.10,H=42.20
RSSI=-75
packet recv
T=21.10,H=42.20
```

```
RSSI=-77
packet recv
T=21.10,H=42.20
RSSI=-76
packet recv
T=21.10,H=42.20
RSSI=-77
```

To do:

1. Test the above examples with your board.
2. Add real sensors to the terminal nodes (senders)
3. Add OLED display to the terminal nodes (receivers)
4. Add OLED display to show the behavior of the gateways.

4.3 Summary

In this lab we have done experiments with simple LoRa-WiFi gateways and a limited number of terminals.
To build more complex networks of terminals-gateways we need to add some control and addressing mechanisms.
These kind of problems will be dealt with in the **advanced IoT laboratories**.

Advanced Smart IoT labs

Lab 5

Advanced WiFi : WiFiManager, ESP-NOW , ESP-LR, ..

Lab 6

Over the Air (OTP) and BLE (BT Low Energy)

Lab 7

Advanced LoRa

Lab 8

Real Time IoT Programming

Lab 9

Advanced IoT Gateways