École Publique d'Ingénieurs en 3 ans

Rapport de projet industriel

PROJET « I AM ENSICAEN »

le 7 février 2022

Lucile VOISIN
lucile.voisin@ecole.ensicaen.fr

Tuteurs école :
Hugo DESCOUBES
Philippe LEFEBVRE

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

www.ensicaen.fr

TABLE DES MATIÈRES

INTRODUCTION

Le projet industriel se veut dans la continuité du projet immersif « I am Ensicaen » dont la vision à terme est de développer des solutions de mesure nomades sur batterie et connectées par protocole de communication wireless. L'objectif étant de mesurer le comportement des salles d'enseignement en temps réel pour aider l'usager à comprendre le comportement de son environnement direct de travail afin de lui permettre d'interagir avec lui et d'en avoir un usage plus responsable.

1. Présentation du projet

Dans le cadre du projet industriel, le cahier des charges s'est recentré autour de l'objectif de la transmission d'une donnée.

Les développements ont conduit à la réalisation d'un démonstrateur effectuant la transmission des données au travers d'une communication LoRa sans implémentation du protocole LoRaWAN.

La question de l'autonomie énergétique n'a pas été abordée lors de ce projet.

2. Présentation du sous-système

2.1. Place et rôle du sous-système dans le projet « I am Ensicaen »

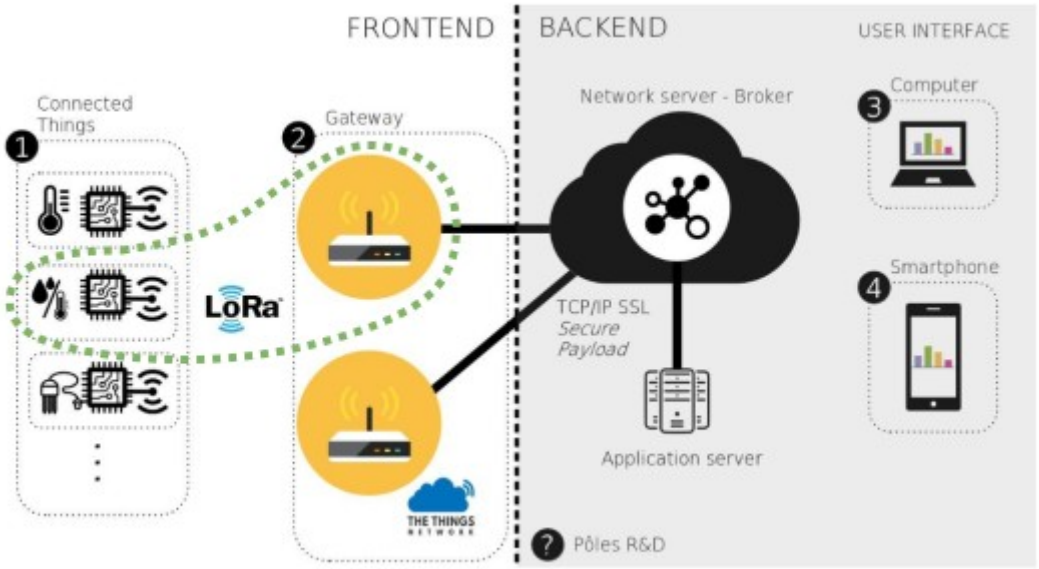

Illustration 1: Entouré en vert, le sous-système ayant fait l'objet de développements lors du projet industriel

Le sous-système se compose d'un objet connecté et d'une gateway, ces deux entités sont mises en relation par communication wireless LoRa. Le sous-système constitue la partie frontend du projet « I am Ensicaen » et englobe les fonctionnalités de mesure des grandeurs physiques et de transmission des données dans l'objectif de les mettre à disposition d'un broker.

2.2. Architecture matérielle du sous-système

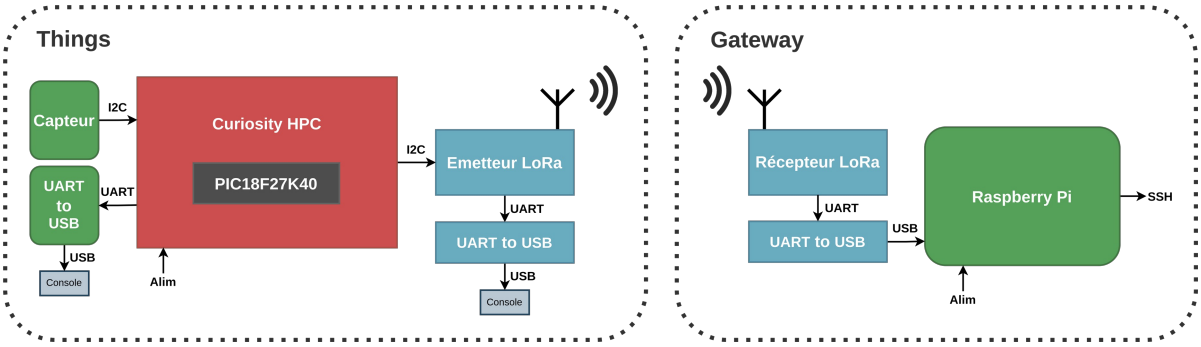

Illustration 2: Architecture matérielle du sous-système

L'objet connecté se compose d'un microcontrôleur, d'un capteur et d'un émetteur LoRa. Ces différents composants communiquent entre eux par protocole I2C. Des bridges UART–USB sont également présents pour de l'affichage en console.

La gateway est constituée d'un récepteur LoRa et d'une Raspberry Pi communiquant ensemble par liaison série au moyen d'un bridge UART-USB.

2.3. Matériel

2.3.1. Objet connecté

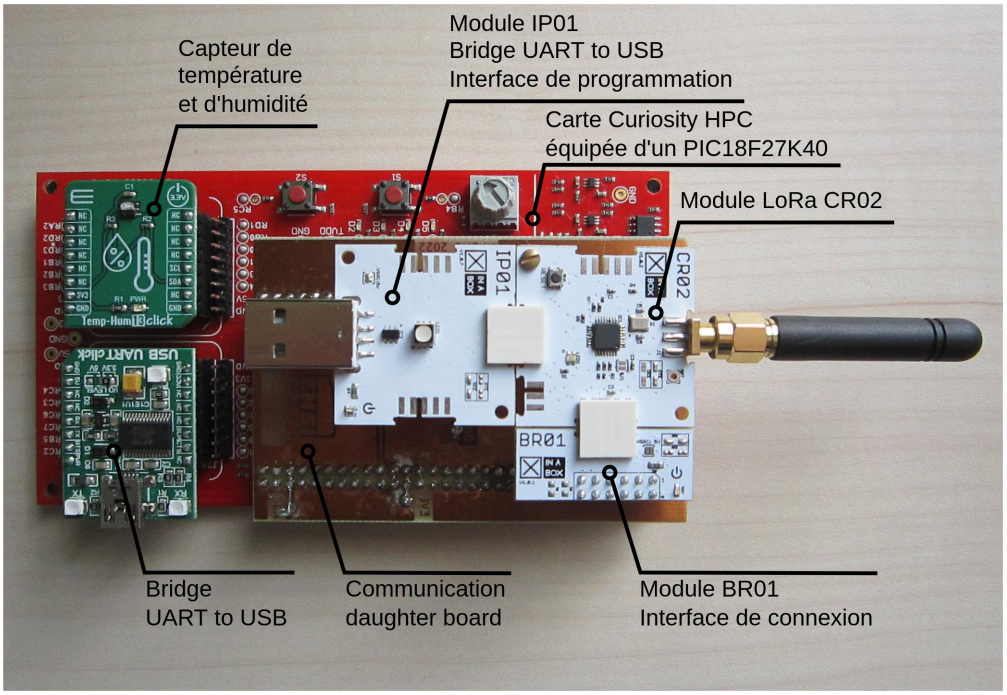

Illustration 3: Objet connecté

- Microcontrôleur 8bits PIC18F27K40 Microchip
- Carte de développement Curiosity HPC Microchip
- Clickboard capteur de température et d'humidité Mikroe
- Clickboard bridge UART to USB Mikroe
- Module CR02 LoRa 868 MHz Xinabox utilisé en émetteur
- Module IP01 Xinabox
- Module BR01 Xinabox
- Communication daughter board Ensicaen

Le PIC18 inséré dans la carte Curiosity HPC se situe sous la communication daughter board. La combinaison du module BR01 et de la communication daughter board (composants non représentés sur l'illustration 2) permet de connecter simplement la carte Curiosity avec le module LoRa CR02.

2.3.2. Gateway

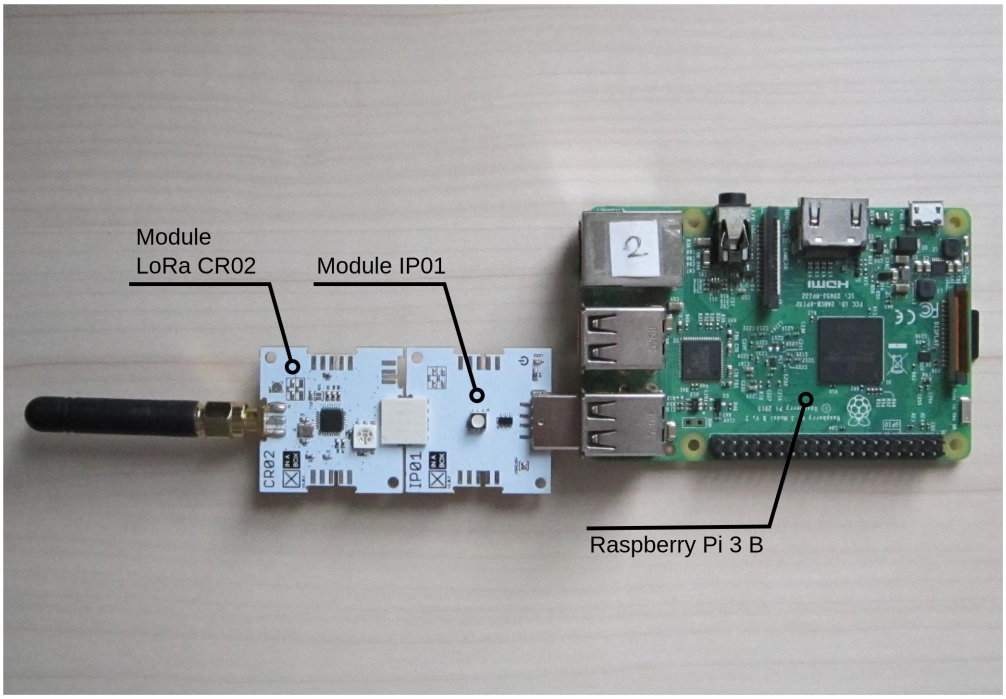

Illustration 4: Gateway

- Module CR02 LoRa 868 MHz Xinabox utilisé en récepteur
- Module IP01 Xinabox
- Raspberry Pi 3 B

Le module IP01 sert à connecter le module CR02 sur un port USB de la Raspberry Pi permettant ainsi une liaison série entre ces deux composants. Il est utile également en tant qu'interface de programmation lors des phases de développement.

2.3.3. Branchements

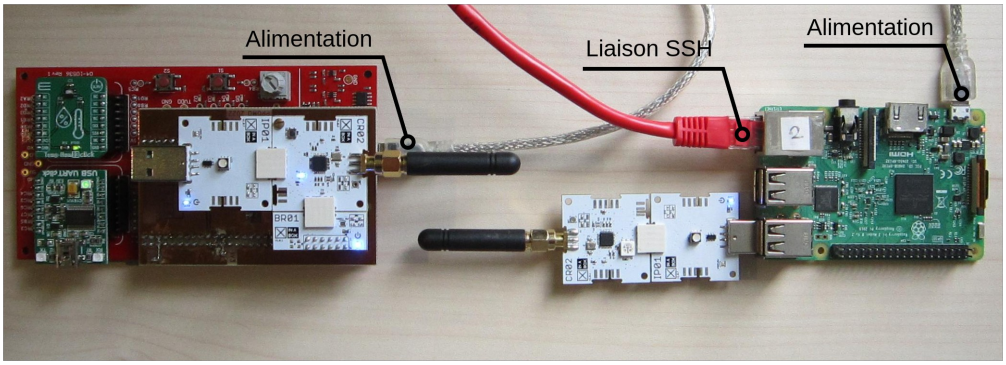

Illustration 5: Branchements

L'alimentation des cartes est réalisée par câble microUSB-USB. La liaison Ethernet (câble RJ45) permet de connecter la Raspberry Pi à internet et d'interagir avec celle-ci par liaison SSH. Il est également possible de connecter un écran, un clavier et une souris à la Raspberry Pi.

Les modules UART to USB et IP01 de l'objet connecté peuvent être reliés aux ports USB d'un ordinateur, ils sont utiles seulement lors des phases de développement et de debug pour programmer les microcontrôleurs et réaliser des affichages en console via minicom.

THINGS

L'objectif de la partie « Things » est l'envoi des données mesurées par un capteur au travers d'une transmission wireless LoRa vers une gateway.

1. Applications de l'objet connecté

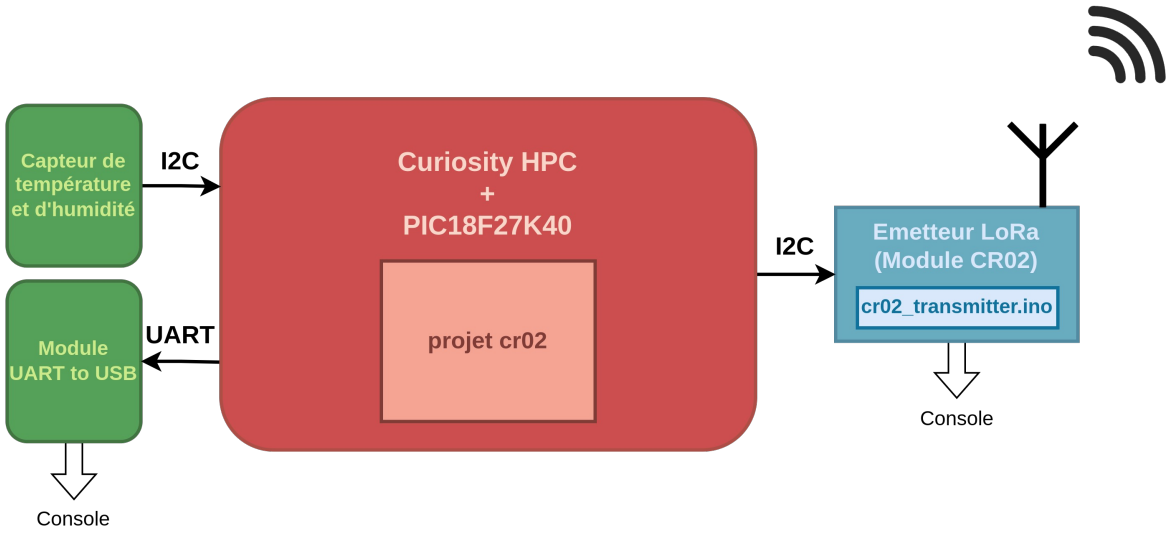

Illustration 6: Applications de l'objet connecté

L'objet connecté possède deux fonctions, effectuer des mesures et émettre les données par communication LoRa.

La donnée est produite par un capteur de température et d'humidité et mise à disposition de l'émetteur LoRa par un firmware embarqué dans le PIC18, le projet cr02.

La donnée est ensuite émise par l'émetteur LoRa possédant son propre programme, cr02_transmitter.ino.

2. Modules LoRa étudiés

Les deux types de modules ayant été étudiés sont le module RL01 constitué d'un bridge I2C-SPI et d'un transmetteur LoRa 433 MHz et le module CR02 composé d'un microcontrôleur 8bits ATmega328P ainsi que d'un transmetteur LoRa 868 MHz.

Les développements ont été réalisés en langage C sous IDE MPLABX v5.15 et toolchain C XC8 v1.45 de Microchip, pour les projets rl01 et cr02, ainsi qu'en langage C++ sous IDE Arduino, uniquement pour le module CR02.

Pour le module RL01, les développements se sont arrêtés au stade de la tentative de configuration du transmetteur LoRa.

Le module CR02 a finalement été retenu pour le démonstrateur.

2.1. Module RL01

Le module RL01 étant composé d'un bridge I2C–SPI et d'un transmetteur LoRa, les documentations de ces deux composants ont été étudiées puis résumées pour faciliter les développements par la suite.

Afin de pouvoir configurer et utiliser le module RL01, la première étape consistait à configurer le bridge et à mettre au point des fonctionnalités de lecture et d'écriture vers son buffer.

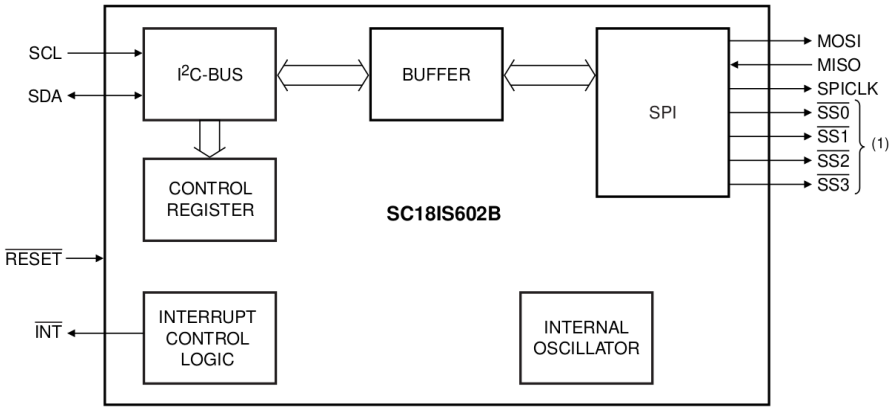

Illustration 7: Schéma bloc du bridge I2C-SPI

Les fonctions permettant l'initialisation du bridge, l'écriture et la lecture du buffer ont été écrites à partir des fonctions I2C du bsp.

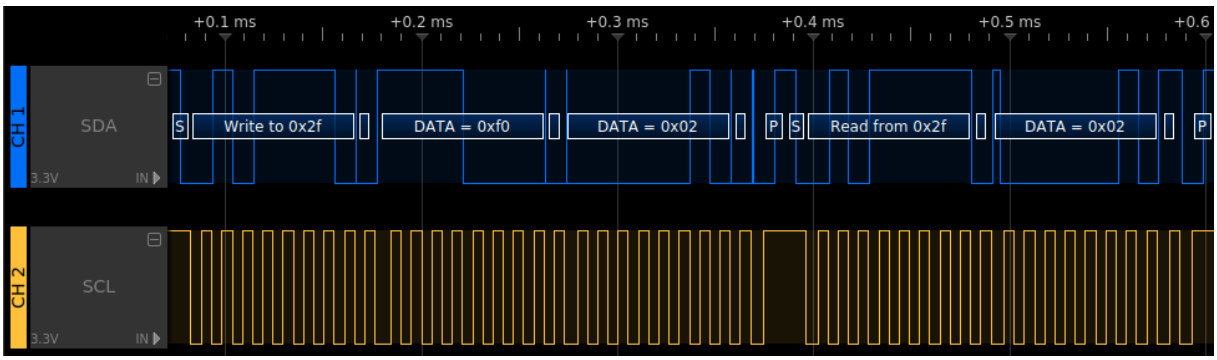

Illustration 8: Trames I2C d'écriture et de lecture du buffer du bridge

La capture de trames I2C a permis de confirmer le bon fonctionnement des écritures et lectures du buffer. L'illustration ci-dessus montre une écriture à l'adresse du buffer (0x2F), suivie d'une commande (0xF0) et de la donnée à écrire (0x02). Vient ensuite une lecture à l'adresse du buffer puis la donnée lue, identique à la donnée écrite.

Cette partie ayant été validée, il fallait ensuite pouvoir configurer le transmetteur LoRa.

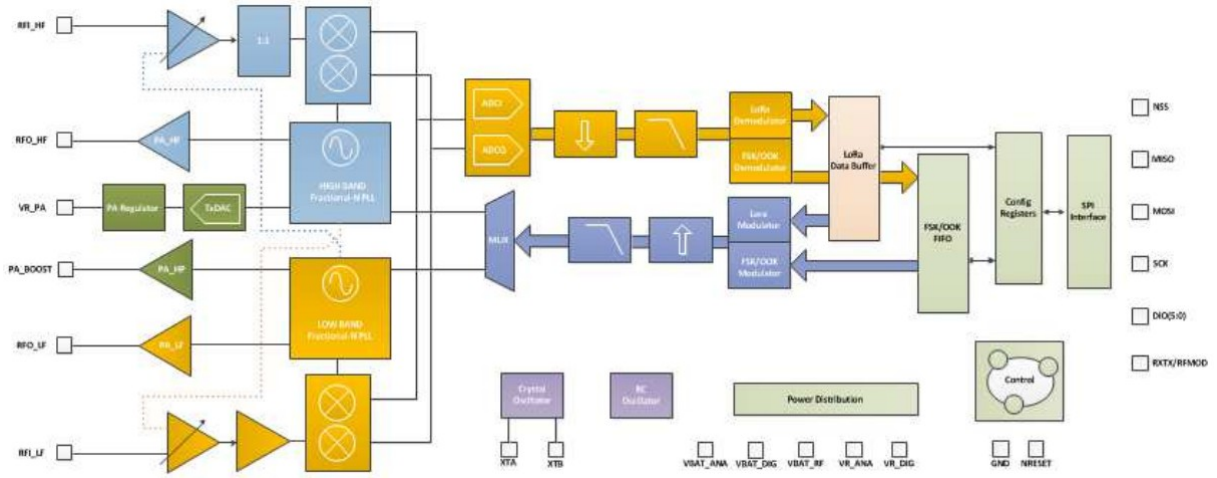

Illustration 9: Schéma bloc du transmetteur

Les tests ont consisté en l'écriture puis la lecture d'une valeur dans un des registres de configuration du transmetteur LoRa.

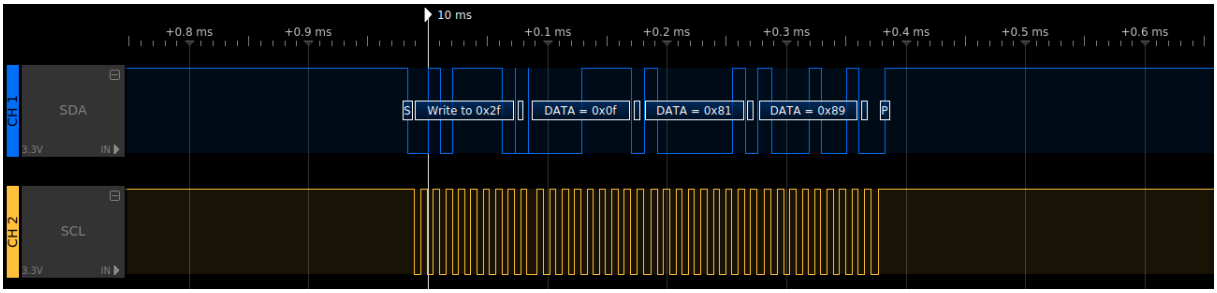

Illustration 10: Trames I2C d'écriture dans le registre RegOpMode du transmetteur

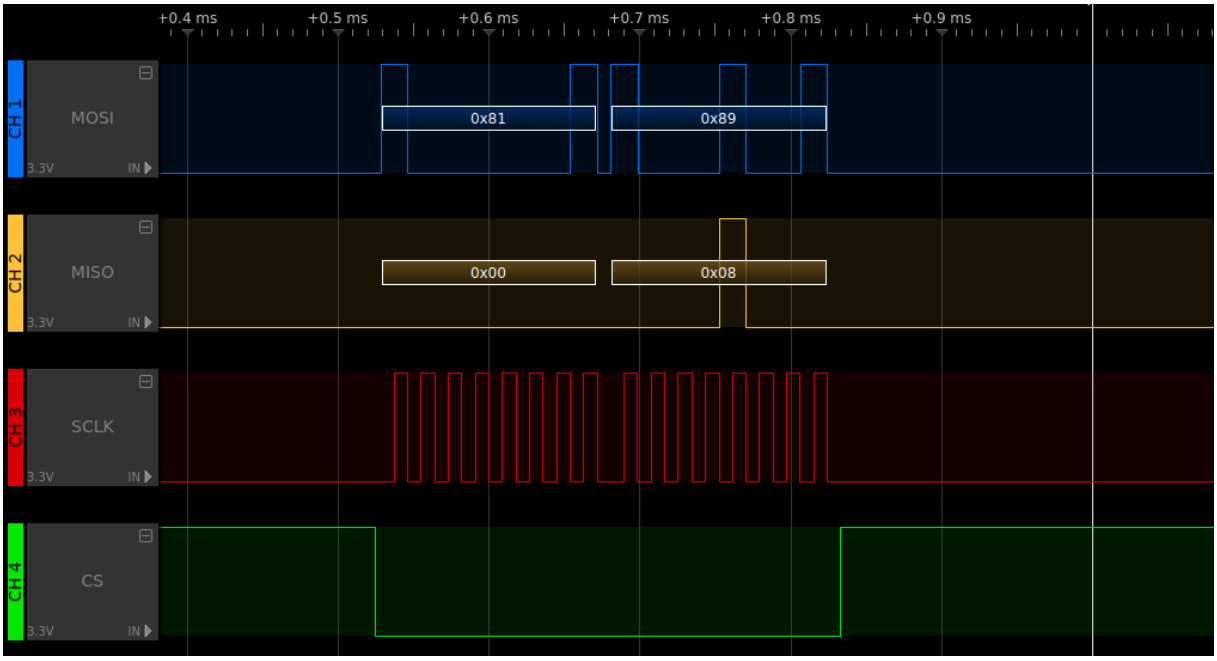

Illustration 11: Trames SPI d'écriture dans le registre RegOpMode du transmetteur

La trame I2C montre la demande d'écriture à l'adresse du bridge (0x2F), une commande d'écriture vers l'interface SPI (0x0F), l'adresse du registre en écriture (0x81 = registre 0x01 + écriture) et la donnée à écrire (0x89).

La trame SPI montre les données, sans le protocole I2C, arrivant au niveau du transmetteur.

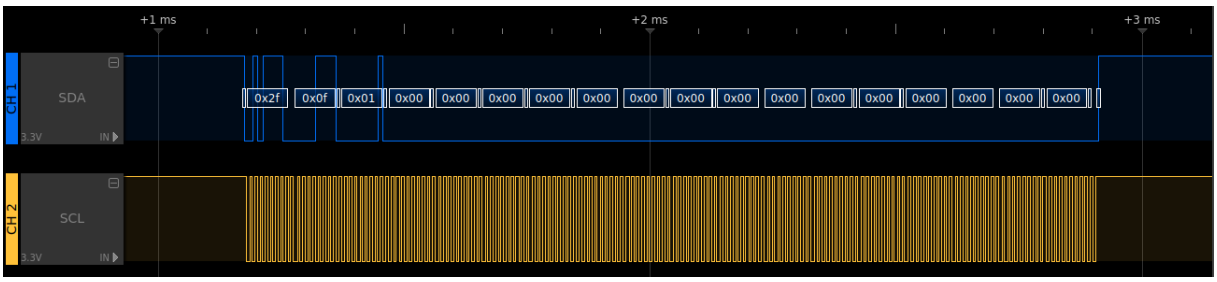

Illustration 12: Trames I2C de demande de lecture des registres

La trame I2C ci-dessus correspond à la demande de lecture du registre RegOpMode (0x01 = 0x01 + lecture).

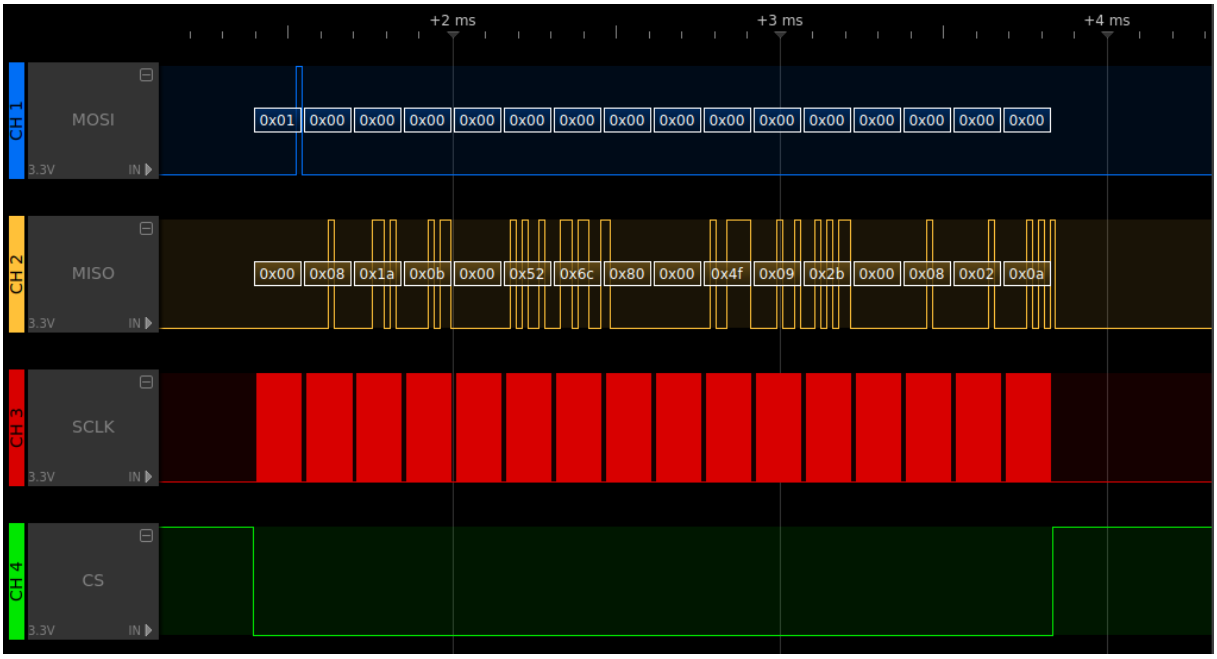

Illustration 13: Trames SPI de lecture des registres

La demande de lecture est transmise par l'intermédiaire du bridge. La trame SPI montre également les données retournées suite à cette demande.

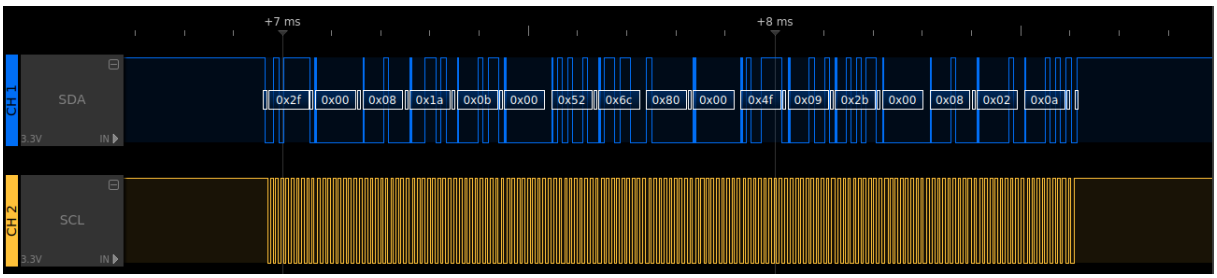

Illustration 14: Trames I2C de lecture des registres

Les données renvoyées sous protocole SPI, sont ensuite transmises par le bridge et converties au format I2C.

L'analyse des trames I2C et SPI de lecture des registres du transmetteur permet de mettre en évidence que l'écriture dans le registre n'a pas été effectuée. Le registre RegOpMode prend la valeur 0x08 correspondant à la transmission FSK en mode sleep.

Name (Address)	Bits	Variable Name	Mode	Default value	FSK/OOK Description
RegFifo (0x00)	7-0	Fifo	rw	0x00	FIFO data input/output
Registers for Common settings					
RegOpMode (0x01)	7	LongRangeMode	r	0x00	0 → FSK/OOK Mode 1→ LoRa™ Mode This bit can be modified only in Sleep mode. A write operation on other device modes is ignored.
	6-5	ModulationType	rw	0x00	Modulation scheme: 00 → FSK 01 → OOK 10 → 11 → reserved
	4	reserved	r	0x0	reserved
	3	LowFrequencyModeOn	rw	0x01	Access Low Frequency Mode registers (from address 0x61 on) 0 → High Frequency Mode (access to HF test registers) 1 → Low Frequency Mode (access to LF test registers)
	2-0	Mode	rw	0x01	Transceiver modes 000 → Sleep mode 001 → Stdby mode 010 → FS mode TX (FSTx) 011 → Transmitter mode (Tx) 100 → FS mode RX (FSRx) 101 → Receiver mode (Rx) 110 → reserved 111 → reserved
RegBitrateMsb (0x02)	7-0	BitRate(15:8)	rw	0x1a	MSB of Bit Rate (chip rate if Manchester encoding is enabled)
RegBitrateLsb (0x03)	7-0	BitRate(7:0)	rw	0x0b	LSB of bit rate (chip rate if Manchester encoding is enabled) $BitRate = \frac{FXOSC}{BitRate(15,0) + \frac{BitrateFrac}{16}}$ Default value: 4.8 kb/s
RegFdevMsb (0x04)	7-6	reserved	rw	0x00	reserved
	5-0	Fdev(13:8)	rw	0x00	MSB of the frequency deviation
RegFdevLsb (0x05)	7-0	Fdev(7:0)	rw	0x52	LSB of the frequency deviation $Fdev = Fstep \times Fdev(15,0)$ Default value: 5 kHz

Illustration 15: Première partie des registres du transmetteur en mode FSK

La comparaison des données retournées avec les valeurs des registres dans la documentation permet de constater qu'il s'agit des valeurs par défaut des registres lorsque le module se trouve en mode FSK.

Les développements réalisés sur le module RL01 n'ont pas permis de le configurer. La méthode d'écriture dans les registres ne fonctionne pas et est à revoir. La lecture des registres semble fonctionner pour une suite de registres mais ne permet pas de lire un seul registre comme voulu.

Il a finalement été décidé de poursuivre avec le module CR02.

2.2. Module CR02

2.2.1. Mesure et mise à disposition des données pour l'émetteur

Cette partie repose sur le travail déjà réalisé lors du projet immersif. Le projet cr02 reprend donc en grande partie le projet htu21d (capteur de température et d'humidité). Celui-ci a simplement été amélioré et augmenté de quelques instructions permettant l'envoi des données au module CR02. Celles-ci sont expédiées avec un intervalle de temps d'environ 3s.

Le PIC18 embarque un firmware réalisant :

- l'initialisation des communications (UART et I2C)
- la configuration du capteur de température et d'humidité
- la récupération des données du capteur
- l'envoi des données sur le bus I2C à destination du module CR02
- l'affichage en console pour les phases de développement et de debug

La donnée retournée par le capteur tient sur deux octets. L'architecture 8bits impose de transmettre octet par octet, la donnée est donc décomposée en deux parties pendant tout le processus de transmission et ne sera reconstituée qu'une fois réceptionnée au niveau de la Raspberry Pi.

2.2.2. Émission des données par communication LoRa

Le module CR02 étant muni d'un microcontrôleur ATmega328P, il a été possible de le programmer, sous environnement Arduino, au moyen de fonctions incluses dans les bibliothèques Wire, pour le protocole I2C, et RadioHead pour la communication LoRa, en s'inspirant des exemples slave_receiver.ino pour la lecture de données I2C et CR02_client.ino pour l'envoi de données par communication LoRa.

Le programme développé pour l'émetteur LoRa comporte dans un premier temps une phase d'initialisation permettant de configurer les communications UART et I2C ainsi que le transmetteur LoRa. La présence d'une donnée adressée au module CR02 sur le bus I2C déclenche un événement effectuant la lecture de deux octets. Les octets sont ensuite émis au travers d'une communication LoRa. Un affichage console est également possible pour les phases de développement et de debug.

Illustration 16: Affichage console des données au niveau du PIC18 et de l'émetteur LoRa

L'affichage en console à chaque étape de la transmission permet de vérifier la validité et la cohérence des données transmises.

GATEWAY

L'objectif de la partie « Gateway » est la réception des données transportées par protocole de communication wireless LoRa, ainsi que leur transmission à un serveur permettant de les rendre accessibles pour les interfaces utilisateur smartphone et PC.

1. Applications de la gateway

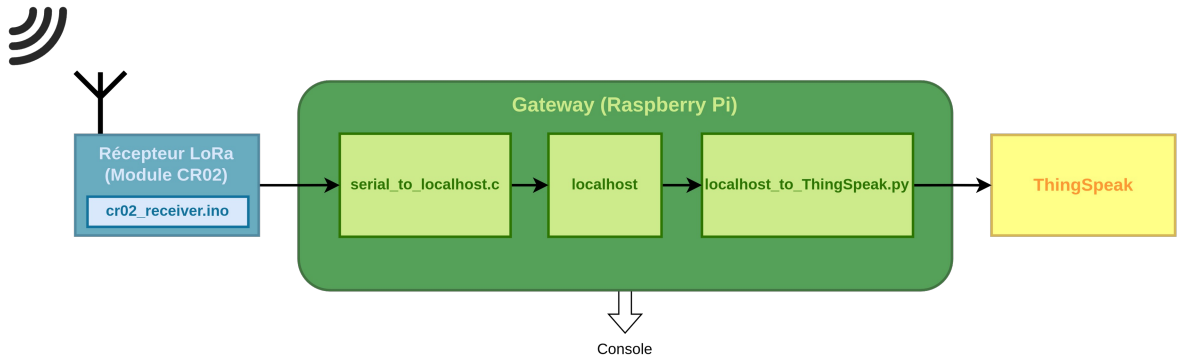

Illustration 17: Applications de la gateway

Les deux missions de la gateway résident en la réception puis la transmission des données.

Pour la réception des données, le programme cr02_receiver.ino écrit pour le module LoRa récepteur permet la lecture des données transmises par communication LoRa puis leur écriture sur le port série. La Raspberry Pi effectue ensuite une lecture du port série, puis met en forme les données pour les préparer à leur transmission.

Pour la transmission des données, la Raspberry Pi dispose de deux applications, serial_to_localhost.c et localhost_to_ThingSpeak.py, permettant de transmettre les données en local dans un premier temps puis vers ThingSpeak dans un second temps.

2. Réception des données

2.1. Module CR02 récepteur

Les données sont réceptionnées, côté gateway, au moyen d'un module CR02 dont l'application a été développée à partir des fonctions provenant de la bibliothèque RadioHead. L'exemple CR02_server.ino a servi de base et a été modifié de façon à lire les deux octets de donnée arrivant par communication LoRa, puis à les écrire sur le port série dans le bon ordre. Un affichage console est réalisé pour le debug.

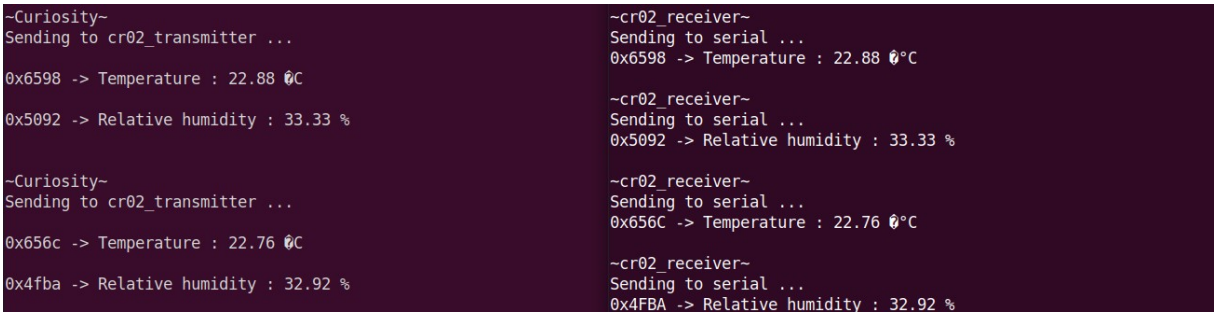

Illustration 18: Affichage console des données au niveau du PIC18 et du récepteur LoRa

La majorité du temps, les données reçues par le récepteur LoRa sont identiques aux données d'origine en provenance du capteur.

Illustration 19: Apparition de valeurs erronées lors de la réception des données

Toutefois, un décalage peut parfois survenir lors de la réception des octets produisant ainsi des valeurs erronées se propageant ensuite dans le reste de la chaîne de transmission.

2.2. Lecture du port série

Les données arrivent sur le port série mais elles ne sont pas encore utilisables. Elles doivent encore être récupérées puis traitées avant de pouvoir être transmises.

Le programme écrit en langage C permettant la lecture du port série repose sur l'utilisation d'une structure termios configurée en mode non canonique de façon à récupérer le nombre d'octets voulu.

Dans un premier temps, l'application a été testée en prenant pour donnée une chaîne de caractères, écrite sur le port série toutes les 3s.

Illustration 20: Lecture d'une chaîne de caractères sur le port série

Le programme a ensuite été modifié pour lire les deux octets de donnée arrivant toutes les 3s en provenance du récepteur LoRa. Les deux octets de donnée lus sur le port série sont assemblés de façon à reconstituer la donnée du capteur. La valeur physique correspondant à la donnée est ensuite calculée. Un affichage de cette valeur physique dans un terminal permet de vérifier le bon fonctionnement du programme.

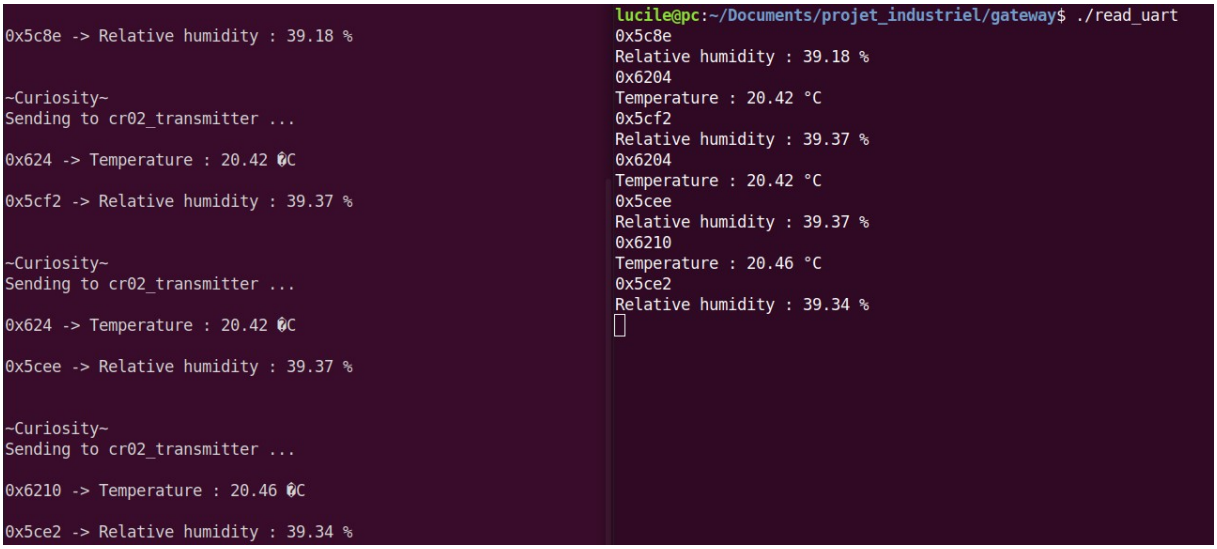

Illustration 21: Lecture des données du module CR02 sur le port série

Les données après lecture du port série correspondent bien aux données de départ, elles ont été correctement lues puis traitées. Elles sont maintenant prêtes pour être transmises.

3. Transmission des données vers un broker

3.1. Protocole MQTT

MQTT (MQ Telemetry Transport) est un protocole de transport de messages basé sur TCP/IP, dont le principe de fonctionnement repose sur les mécanismes de publication et d'abonnement à des topics entre des clients et un serveur. Il s'agit d'un protocole léger, adapté à une utilisation dans des environnements contraints comme l'Internet des objets où l'empreinte du code doit être faible et lorsque les ressources en énergie et bande passante sont limitées.

Le modèle de publish/subscribe offre un découplage spatial entre les différents clients, ceux-ci ne se contactent jamais directement. L'intermédiaire entre les clients se fait au moyen d'un broker.

Les clients s'abonnent à un ou plusieurs topics. Un client peut publier des messages sur un topic choisi à destination d'un broker. Le rôle du broker est de filtrer tous les messages entrant, en fonction des topics, et de les distribuer correctement aux abonnés.

3.2. ThingSpeak

Dans le cadre du projet, ThingSpeak est utilisé pour vérifier le bon fonctionnement du programme localhost_to_ThingSpeak.py (envoi de messages par protocole MQTT vers un broker). ThingSpeak permet également de mettre les données à disposition pour les interfaces utilisateur.

Le channel ThingSpeak est configuré de manière à disposer d'un champ pour chaque type de donnée.

La création d'un device MQTT permet d'obtenir les identifiants nécessaires pour la configuration du client MQTT localhost_to_ThingSpeak.py.

3.3. Transmission des données en deux temps

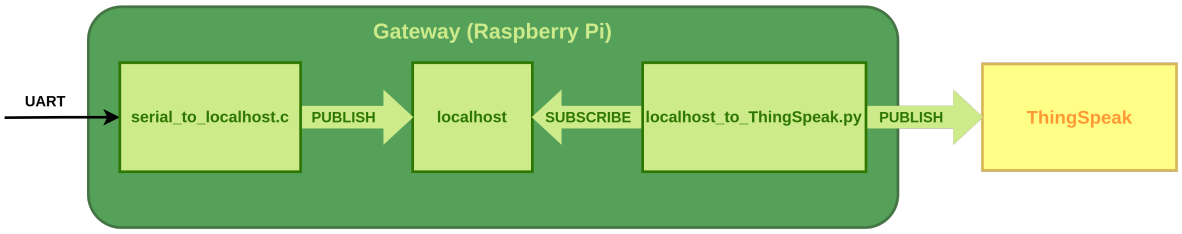

Illustration 22: Décomposition de la transmission

La décomposition de la transmission en deux étapes est la conséquence de difficultés rencontrées pour développer un programme en langage C transmettant directement les données vers ThingSpeak. La solution de contournement a consisté en l'écriture de deux programmes au moyen de fonctions fournies par les bibliothèques MQTT Paho C et Paho python, permettant ainsi d'envoyer finalement la donnée vers ThingSpeak.

Les développements réalisés pour la transmission via protocole MQTT sont le résultat d'une collaboration avec Briac Panchot et Thomas Boduch.

3.3.1. Première étape : serial_to_localhost.c

Cette première étape reprend le travail effectué précédemment pour la lecture du port série et y ajoute l'envoi de messages par protocole MQTT vers localhost. Les fonctions mises à disposition par la bibliothèque Paho C permettent de créer un client MQTT, de le configurer, puis de publier des messages sur un topic. L'exemple MQTTClient_publish.c fourni avec la bibliothèque Paho C a servi de base pour développer ce programme.

Dans l'objectif de les rendre exploitables, les messages sont des chaînes de caractères mises sous la forme « type de la mesure : valeur ».

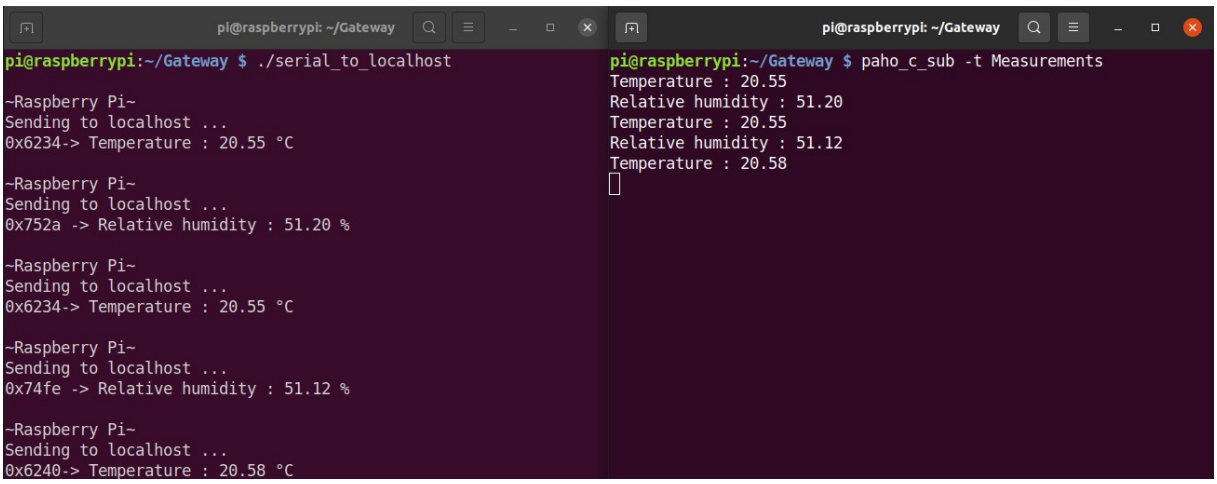

Illustration 23: Publication des messages vers localhost

Le programme est testé au moyen de la ligne de commande paho_c_sub permettant de s'abonner au topic « Measurements » sur lequel les messages sont publiés par le client MQTT. Il est ainsi possible de vérifier que les messages ont été correctement publiés vers localhost.

3.3.2. Deuxième étape : localhost_to_ThingSpeak.py

Ce second programme réalisé au moyen de la bibliothèque Paho python permet dans un premier temps de souscrire au topic local « Measurements » de façon à récupérer les messages qui y sont publiés par le premier programme, ceux-ci sont ensuite publiés vers ThingSpeak.

Les messages reçus dans le format « type de la mesure : valeur » sont décodés de façon à déterminer le type de la donnée pour l'envoyer dans le champ correspondant du channel ThingSpeak.

Les messages sont publiés vers ThingSpeak sur le topic « channels/channel_ID/publish » sous la forme « fieldX=valeur ».

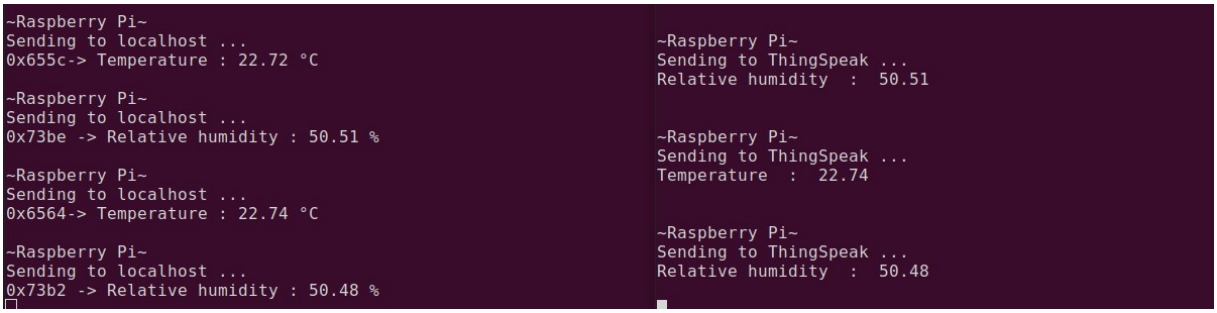

Illustration 24: Publication des messages vers ThingSpeak

L'affichage en console permet de vérifier que les données récupérées au niveau du localhost par le second programme sont identiques à celles publiées par le premier programme.

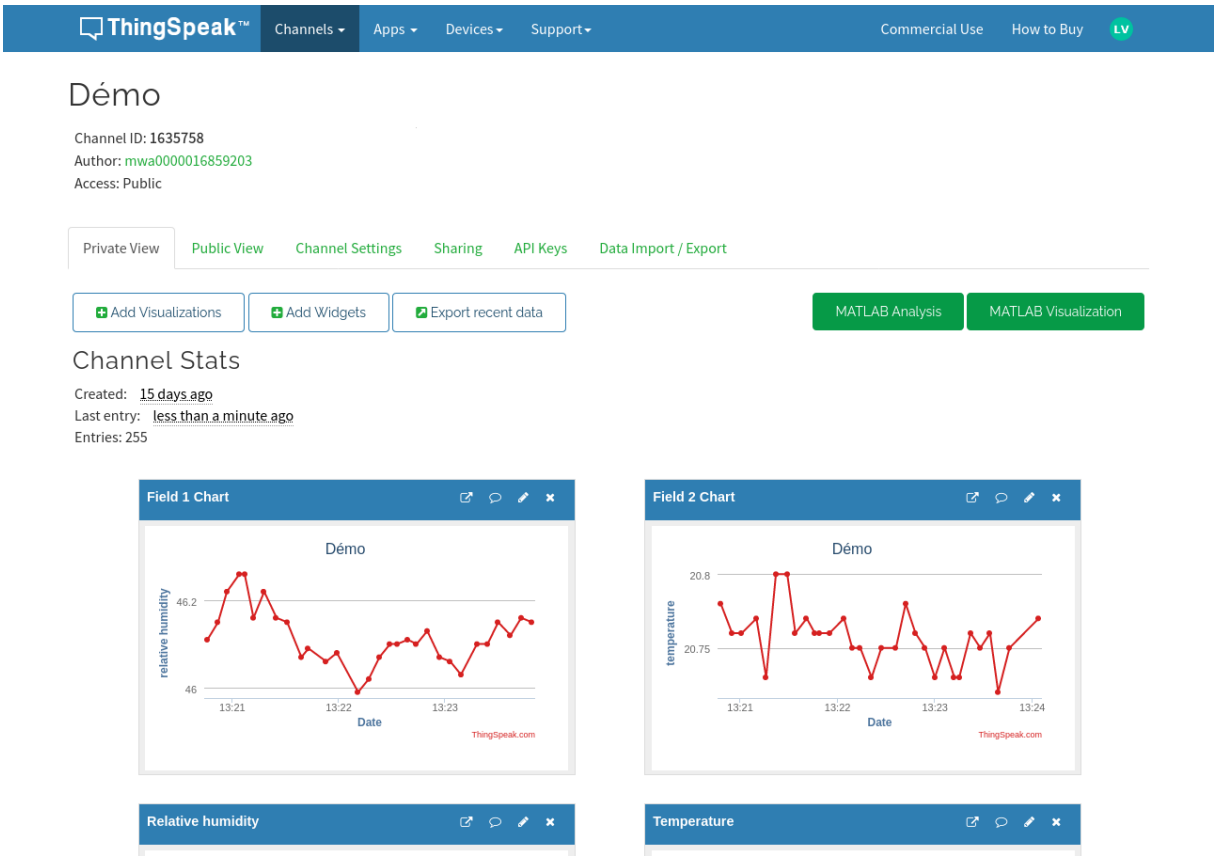

Illustration 25: Visualisation des données sous ThingSpeak

La visualisation des données sous ThingSpeak permet de vérifier que celles-ci ont été correctement transmises et récupérées dans les bons champs.

CONCLUSION

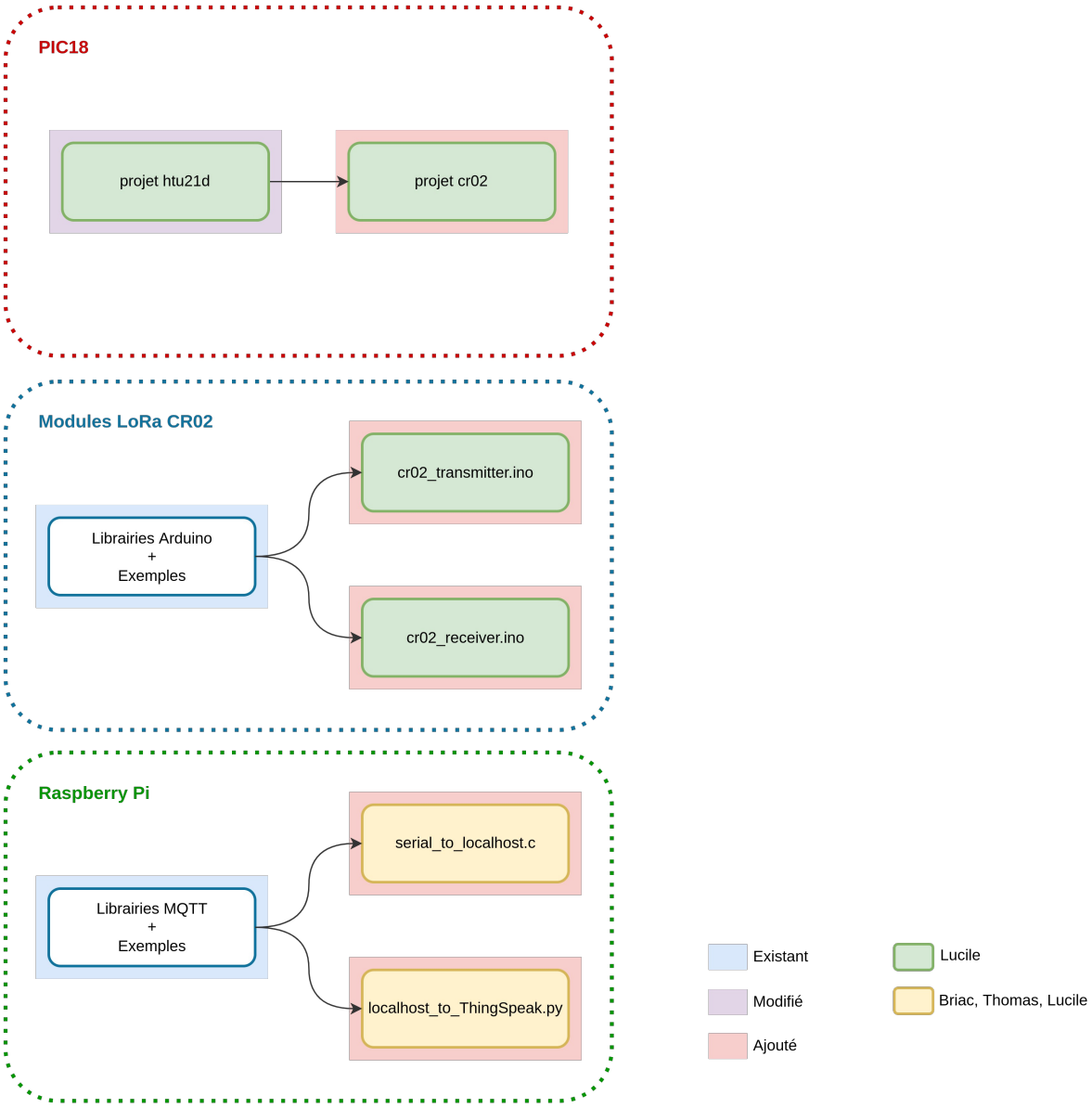

Illustration 26: Développements réalisés

Les développements réalisés ont permis d'obtenir un démonstrateur effectuant la transmission des données du capteur vers ThingSpeak. Celui-ci pourra servir de base de travail pour la prochaine génération d'ingénieurs qui travaillera sur le projet « I am Ensicaen ».

ANNEXES

```c
/*
 * test program for cr02
 * author
 */

#include <stdio.h>
#include <bsplib.h>
#include <htu21d.h>
#include <cr02.h>

#define  DELAY_3S()    for (int i = 0; i < 15; i++) delay_200ms()

#define DEBUG_CONSOLE   1

void main(void)
{
    uInt16_t temperature;
    uInt16_t humidity;
    uInt8_t checksum;

#if DEBUG_CONSOLE
    float humidity_physical_value;
    float temperature_physical_value;
    uInt8_t tmp[50];
#endif

    uInt8_t temperature_msb;
    uInt8_t temperature_lsb;
    uInt8_t humidity_msb;
    uInt8_t humidity_lsb;

    /* system init */
    i2c1_init( I2C1_BAUDRATE );
#if DEBUG_CONSOLE
    uart1_init( UART1_BAUD_RATE );
#endif

    /* soft reset */
    soft_reset( HTU21D_ADD );

    /* choice of measurement resolution */
    write_register( HTU21D_ADD, MEASUREMENT_RESOLUTION_11_11 );

    while (1) {
        read_measurement( HTU21D_ADD, TEMPERATURE_MEASUREMENT_HOLD_MASTER, &temperature, &checksum );
        delay_10us();
        read_measurement( HTU21D_ADD, HUMIDITY_MEASUREMENT_HOLD_MASTER, &humidity, &checksum );
        delay_10us();

        temperature_msb = (uInt8_t) (temperature >> 8) & 0x00FF;
        temperature_lsb = (uInt8_t) temperature & 0x00FF;

        humidity_msb = (uInt8_t) (humidity >> 8) & 0x00FF;
        humidity_lsb = (uInt8_t) humidity & 0x00FF;

#if DEBUG_CONSOLE
        humidity &= 0xfffc;
        temperature_physical_value = ( ( temperature * 175.72 ) / 65536 ) - 46.85;
        humidity_physical_value = ( ( humidity * 125.0 ) / 65536 ) - 6;

        uart1_puts( "\n~Curiosity~\n\r" );
        uart1_puts( "Sending to cr02_transmitter ...\n\n\r" );

        sprintf( tmp, "0x%x%x", temperature_msb, temperature_lsb );
        uart1_puts( tmp );
        sprintf( tmp,
                 " -> Temperature : %d.%02u °C",
                 (uInt8_t) temperature_physical_value,
                 (uInt8_t) ((temperature_physical_value - (uInt8_t) temperature_physical_value ) * 100) );
        uart1_puts( tmp );
        uart1_puts( "\n\n\r" );

        sprintf( tmp, "0x%x%x", humidity_msb, humidity_lsb );
        uart1_puts( tmp );
        sprintf( tmp,
                 " -> Relative humidity : %d.%02u %%",
                 (uInt8_t) humidity_physical_value,
                 (uInt8_t) ((humidity_physical_value - (uInt8_t) humidity_physical_value ) * 100) );
        uart1_puts( tmp );
        uart1_puts( "\n\n\r" );
#endif

        i2c1_write(CR02_ADD, temperature_msb, temperature_lsb);

        DELAY_3S();

        i2c1_write(CR02_ADD, humidity_msb, humidity_lsb);

        DELAY_3S();
    }
}
```

projet cr02

slave_receiver

```
// Wire Slave Receiver
// by Nicholas Zambetti <http://www.zambetti.com>

// Demonstrates use of the Wire library
// Receives data as an I2C/TWI slave device
// Refer to the "Wire Master Writer" example for use with this

// Created 29 March 2006

// This example code is in the public domain.


#include <Wire.h>

void setup() {
  Wire.begin(8);                // join i2c bus with address #8
  Wire.onReceive(receiveEvent); // register event
  Serial.begin(9600);           // start serial for output
}

void loop() {
  delay(100);
}

// function that executes whenever data is received from master
// this function is registered as an event, see setup()
void receiveEvent(int howMany) {
  while (1 < Wire.available()) { // loop through all but the last
    char c = Wire.read(); // receive byte as a character
    Serial.print(c);         // print the character
  }
  int x = Wire.read();    // receive byte as an integer
  Serial.println(x);         // print the integer
}
```

CR02_client

```
#include <RH_RF95.h>

#define LED_BUILTIN 16

#define CR02_FREQUENCY 868.0

uint8_t tempdata[30];

/*
  Slave Select mapped to D13/PD2
  Interrupt mapped to D14/PB2
*/
RH_RF95 CR02;

void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);

  Serial.begin(115200);

  if (!CR02.init()){
    Serial.println("init failed");
  }

  // The default transmitter power is 13dBm, using PA_BOOST.
  // If you are using RFM95/96/97/98 modules which uses the PA_BOOST transmitter pin, then
  // you can set transmitter powers from 5 to 23 dBm:
  // Failure to do that will result in extremely low transmit powers.

  //CR02.setModemConfig(CR02.Bw31_25Cr48Sf512);
  CR02.setFrequency(CR02_FREQUENCY);
  CR02.setTxPower(23, false);
}

void loop()
{
  Serial.println("Sending to CR02_server");
  // Send a message to CR02_server

  sprintf(tempdata, "%s", "Hello Server");

  CR02.send(tempdata, sizeof(tempdata));

  CR02.waitPacketSent();
  // Now wait for a reply
  uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
  uint8_t len = sizeof(buf);

  if (CR02.waitAvailableTimeout(3000))
  {
    // Should be a reply message for us now
    if (CR02.recv(buf, &len))
    {
      digitalWrite(LED_BUILTIN, HIGH);
      Serial.print("got reply: ");
      Serial.println((char*)buf);
      Serial.print("RSSI: ");
      Serial.println(CR02.lastRssi(), DEC);
    }
    else
    {
      Serial.println("recv failed");
    }
  }
  else
  {
    Serial.println("No reply, is CR02_server running?");
  }
  digitalWrite(LED_BUILTIN, LOW);
  delay(400);
}
```

cr02_transmitter

```
#include <Wire.h>
#include <RH_RF95.h>

#define LED_BUILTIN 16

#define CR02_FREQUENCY 868.0

#define DEBUG_CONSOLE 1

uint8_t payload_msb;
uint8_t payload_lsb;

/*
  Slave Select mapped to D13/PD2
  Interrupt mapped to D14/PB2
*/
RH_RF95 CR02;

void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);

#if DEBUG_CONSOLE
  Serial.begin(9600, SERIAL_8N1);   // start serial for output
#endif

  Wire.begin(4);                    // join i2c bus with address #4
  Wire.onReceive(receiveEvent);     // register event

  if (!CR02.init()) {
#if DEBUG_CONSOLE
    Serial.println("init failed");
#endif
  }

  // The default transmitter power is 13dBm, using PA_BOOST.
  // If you are using RFM95/96/97/98 modules which uses the PA_BOOST transmitter pin, then
  // you can set transmitter powers from 5 to 23 dBm:
  // Failure to do that will result in extremely low transmit powers.

  //CR02.setModemConfig(CR02.Bw31_25Cr48Sf512);
  CR02.setFrequency(CR02_FREQUENCY);
  CR02.setTxPower(23, false);
}

void loop()
{
#if DEBUG_CONSOLE
  Serial.println("\n~cr02_transmitter~");
  Serial.println("Sending to cr02_receiver ...");
  Serial.print("0x");
  Serial.print(payload_msb, HEX);
  Serial.println(payload_lsb, HEX);
#endif

  CR02.send(&payload_msb, sizeof(payload_msb));
  CR02.waitPacketSent();
  CR02.send(&payload_lsb, sizeof(payload_lsb));
  CR02.waitPacketSent();

  delay(3000);
}

// function that executes whenever data is received from master
// this function is registered as an event, see setup()
void receiveEvent(int howMany)
{
  payload_msb = Wire.read();
  payload_lsb = Wire.read();
}
```

CR02_server

```
#include <RH_RF95.h>

#define LED_BUILTIN 16

#define CR02_FREQUENCY 868.0

uint8_t tempdata[30];

/*
  Slave Select mapped to D13/PD2
  Interrupt mapped to D14/PB2
*/
RH_RF95 CR02;

void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);

  Serial.begin(115200);

  if (!CR02.init()) {
    Serial.println("init failed");
  }
  // The default transmitter power is 13dBm, using PA_BOOST.
  // If you are using RFM95/96/97/98 modules which uses the PA_BOOST transmitter pin, then
  // you can set transmitter powers from 5 to 23 dBm:
  // Failure to do that will result in extremely low transmit powers.

  //CR02.setModemConfig(CR02.Bw31_25Cr48Sf512);
  CR02.setFrequency(CR02_FREQUENCY);
  CR02.setTxPower(23, false);
}

void loop()
{
  if (CR02.available())
  {
    // Should be a message for us now
    uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
    uint8_t len = sizeof(buf);
    if (CR02.recv(buf, &len))
    {
      digitalWrite(LED_BUILTIN, HIGH);
      Serial.print("got request: ");
      Serial.println((char*)buf);
      Serial.print("RSSI: ");
      Serial.println(CR02.lastRssi(), DEC);

      // Send a reply
      sprintf(tempdata, "%s", "Hello Client");
      CR02.send(tempdata, sizeof(tempdata));
      CR02.waitPacketSent();
      Serial.println("Sent a reply");
      digitalWrite(LED_BUILTIN, LOW);
    }
    else
    {
      Serial.println("recv failed");
    }
  }
}
```

cr02_receiver

```
#include <RH_RF95.h>

#define LED_BUILTIN 16

#define CR02_FREQUENCY 868.0

#define TRUE  1
#define FALSE 0

#define DEBUG_CONSOLE 0

uint8_t payload;
uint8_t payload_msb;
uint8_t payload_lsb;
uint8_t len = sizeof(payload);

uint8_t is_msb = TRUE;
uint8_t nb_byte = 0;

char buf[50];
uint8_t status_measurement;
uint16_t data;
float physical_value;

/*
  Slave Select mapped to D13/PD2
  Interrupt mapped to D14/PB2
*/
RH_RF95 CR02;

void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);

  Serial.begin(9600, SERIAL_8N1);   // start serial for output

  if (!CR02.init()) {
#if DEBUG_CONSOLE
    Serial.println("init failed");
#endif
  }

  // The default transmitter power is 13dBm, using PA_BOOST.
  // If you are using RFM95/96/97/98 modules which uses the PA_BOOST transmitter pin, then
  // you can set transmitter powers from 5 to 23 dBm:
  // Failure to do that will result in extremely low transmit powers.

  //CR02.setModemConfig(CR02.Bw31_25Cr48Sf512);
  CR02.setFrequency(CR02_FREQUENCY);
  CR02.setTxPower(23, false);
}

void loop()
{
  if (CR02.available()) {

    if (CR02.recv(&payload, &len)) {

      digitalWrite(LED_BUILTIN, HIGH);

      if (is_msb) {
        payload_msb = payload;
        is_msb = FALSE;
        nb_byte ++;
      } else {
        payload_lsb = payload;
        is_msb = TRUE;
        nb_byte ++;
      }

      digitalWrite(LED_BUILTIN, LOW);

    } else {
#if DEBUG_CONSOLE
      Serial.println("recv failed");
#endif
    }

    if (nb_byte == 2) {

#if DEBUG_CONSOLE
      data = payload_msb << 8;
      data |= payload_lsb;

      Serial.println("\n~cr02_receiver~");
      Serial.println("Sending to serial ...");
      Serial.print("0x");
      Serial.print(payload_msb, HEX);
      Serial.print(payload_lsb, HEX);

      status_measurement = payload_lsb & 0x02;
      if (status_measurement == 0x02) {
        data &= 0xfffc;
        physical_value = ( ( data * 125.0 ) / 65536 ) - 6;
        Serial.print(" -> Relative humidity : ");
        sprintf(buf,
                "%d.%02u",
                (uint8_t) physical_value,
                (uint8_t) ((physical_value - (uint8_t) physical_value ) * 100));
        Serial.print(buf);
        Serial.println(" %");
      }
      if (status_measurement == 0x00) {
        physical_value = ( (data * 175.72 ) /65536 ) - 46.85;
        Serial.print(" -> Temperature : ");
        sprintf(buf,
                "%d.%02u",
                (uint8_t) physical_value,
                (uint8_t) ((physical_value - (uint8_t) physical_value ) * 100));
        Serial.print(buf);
        Serial.println(" °C");
      }
#endif

#if !DEBUG_CONSOLE
      Serial.write(payload_lsb);
      Serial.write(payload_msb);
#endif

      nb_byte = 0;

    }
  }
}
```

cr02_receiver.ino avec exemple ayant servi de base

MQTTClient_publish.c

```c
/*******************************************************************************
 * Copyright (c) 2012, 2020 IBM Corp.
 *
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License v2.0
 * and Eclipse Distribution License v1.0 which accompany this distribution.
 *
 * The Eclipse Public License is available at
 *   https://www.eclipse.org/legal/epl-2.0/
 * and the Eclipse Distribution License is available at
 *   http://www.eclipse.org/org/documents/edl-v10.php.
 *
 * Contributors:
 *    Ian Craggs - initial contribution
 *******************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "MQTTClient.h"

#define ADDRESS     "tcp://mqtt.eclipse.org:1883"
#define CLIENTID    "ExampleClientPub"
#define TOPIC       "MQTT Examples"
#define PAYLOAD     "Hello World!"
#define QOS         1
#define TIMEOUT     10000L

int main(int argc, char* argv[])
{
    MQTTClient client;
    MQTTClient_connectOptions conn_opts = MQTTClient_connectOptions_initializer;
    MQTTClient_message pubmsg = MQTTClient_message_initializer;
    MQTTClient_deliveryToken token;
    int rc;

    if ((rc = MQTTClient_create(&client, ADDRESS, CLIENTID,
        MQTTCLIENT_PERSISTENCE_NONE, NULL)) != MQTTCLIENT_SUCCESS)
    {
        printf("Failed to create client, return code %d\n", rc);
        exit(EXIT_FAILURE);
    }

    conn_opts.keepAliveInterval = 20;
    conn_opts.cleansession = 1;
    if ((rc = MQTTClient_connect(client, &conn_opts)) != MQTTCLIENT_SUCCESS)
    {
        printf("Failed to connect, return code %d\n", rc);
        exit(EXIT_FAILURE);
    }

    pubmsg.payload = PAYLOAD;
    pubmsg.payloadlen = (int)strlen(PAYLOAD);
    pubmsg.qos = QOS;
    pubmsg.retained = 0;
    if ((rc = MQTTClient_publishMessage(client, TOPIC, &pubmsg, &token)) != MQTTCLIENT_SUCCESS)
    {
        printf("Failed to publish message, return code %d\n", rc);
        exit(EXIT_FAILURE);
    }

    printf("Waiting for up to %d seconds for publication of %s\n"
            "on topic %s for client with ClientID: %s\n",
            (int)(TIMEOUT/1000), PAYLOAD, TOPIC, CLIENTID);
    rc = MQTTClient_waitForCompletion(client, token, TIMEOUT);
    printf("Message with delivery token %d delivered\n", token);

    if ((rc = MQTTClient_disconnect(client, 10000)) != MQTTCLIENT_SUCCESS)
        printf("Failed to disconnect, return code %d\n", rc);
    MQTTClient_destroy(&client);
    return rc;
}
```

serial_to_localhost.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <termios.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

#include "MQTTClient.h"

#define ADDRESS     "localhost:1883"
#define CLIENTID    "ClientPubLocalhost"
#define TOPIC       "Measurements"
#define QOS         1
#define TIMEOUT     6000L

#define DEBUG_CONSOLE   1

int main()
{
    int serial_port = open( "/dev/ttyUSB0", O_RDONLY );
    if ( serial_port == -1 ) {
        printf( "Error %i from open : %s\n", errno, strerror(errno) );
        exit(EXIT_FAILURE);
    }

    struct termios tty_options;

    if ( tcgetattr( serial_port, &tty_options ) != 0 ) {
        printf( "Error %i from tcgetattr : %s\n", errno, strerror(errno) );
        exit(EXIT_FAILURE);
    }

    tty_options.c_cflag &= ~PARENB;         // Clear parity bit, disabling parity (most common)
    tty_options.c_cflag &= ~CSTOPB;         // Clear stop field, only one stop bit used in communication (most common)
    tty_options.c_cflag &= ~CSIZE;          // Clear all bits that set the data size
    tty_options.c_cflag |= CS8;             // 8 bits per byte (most common)
    tty_options.c_cflag &= ~CRTSCTS;        // Disable RTS/CTS hardware flow control (most common)
    tty_options.c_cflag |= CREAD | CLOCAL;  // Turn on READ & ignore ctrl lines (CLOCAL = 1)

    tty_options.c_lflag &= ~ICANON;
    tty_options.c_lflag &= ~ECHO;           // Disable echo
    tty_options.c_lflag &= ~ECHOE;          // Disable erasure
    tty_options.c_lflag &= ~ECHONL;         // Disable new-line echo
    tty_options.c_lflag &= ~ISIG;           // Disable interpretation of INTR, QUIT and SUSP
    tty_options.c_iflag &= ~(IXON | IXOFF | IXANY);    // Turn off s/w flow ctrl
    tty_options.c_iflag &= ~(IGNBRK|BRKINT|PARMRK|ISTRIP|INLCR|IGNCR|ICRNL);    // Disable any special handling of received bytes

    tty_options.c_oflag &= ~OPOST; // Prevent special interpretation of output bytes (e.g. newline chars)
    tty_options.c_oflag &= ~ONLCR; // Prevent conversion of newline to carriage return/line feed
    // tty_options.c_oflag &= ~OXTABS; // Prevent conversion of tabs to spaces (NOT PRESENT ON LINUX)
    // tty_options.c_oflag &= ~ONOEOT; // Prevent removal of C-d chars (0x004) in output (NOT PRESENT ON LINUX)

    tty_options.c_cc[VTIME] = 1;
    tty_options.c_cc[VMIN] = 2;

    cfsetispeed( &tty_options, B9600 );
    cfsetospeed( &tty_options, B9600 );

    if ( tcsetattr( serial_port, TCSANOW, &tty_options ) != 0 ) {
        printf( "Error %i from tcsetattr: %s\n", errno, strerror(errno) );
        exit(EXIT_FAILURE);
    }


    MQTTClient client;
    MQTTClient_connectOptions conn_opts = MQTTClient_connectOptions_initializer;
    MQTTClient_message pubmsg = MQTTClient_message_initializer;
    MQTTClient_deliveryToken token;
    int rc;

    if ((rc = MQTTClient_create(&client, ADDRESS, CLIENTID, MQTTCLIENT_PERSISTENCE_NONE, NULL)) != MQTTCLIENT_SUCCESS) {
        printf("Failed to create client, return code %d\n", rc);
        exit(EXIT_FAILURE);
    }

    conn_opts.keepAliveInterval = 20;
    conn_opts.cleansession = 1;
    if ((rc = MQTTClient_connect(client, &conn_opts)) != MQTTCLIENT_SUCCESS) {
        printf("Failed to connect, return code %d\n", rc);
        exit(EXIT_FAILURE);
    }


    u_int16_t payload;
    u_int8_t measurement_id;
    float humidity_physical_value;
    float temperature_physical_value;

    char tmp[50];
    int num_bytes = 0;

    while (1) {
        num_bytes = read( serial_port, &payload, sizeof(payload) );

        if ( num_bytes < 0 ) {
            printf( "Error reading: %s", strerror(errno) );
            exit(EXIT_FAILURE);
        }

#if DEBUG_CONSOLE
        printf( "\n~Raspberry Pi~\n");
        printf( "Sending to localhost ...\n" );
        printf( "0x%x", payload );
#endif

        measurement_id = payload & 0x02;

        if (measurement_id == 0x02) {
            payload &= 0xfffc;
            humidity_physical_value = ( ( payload * 125.0 ) / 65536 ) - 6;
            sprintf( tmp,
                     "Relative humidity : %d.%02u",
                     (u_int8_t) humidity_physical_value,
                     (u_int8_t) ((humidity_physical_value - (u_int8_t) humidity_physical_value) * 100 ) );
#if DEBUG_CONSOLE
            printf( " -> %s %%\n", tmp );
#endif
        }
        if (measurement_id == 0x00) {
            temperature_physical_value = ( (payload * 175.72 ) /65536 ) - 46.85;
            sprintf( tmp,
                     "Temperature : %d.%02u",
                     (u_int8_t) temperature_physical_value,
                     (u_int8_t) ((temperature_physical_value - (u_int8_t) temperature_physical_value) * 100) );
#if DEBUG_CONSOLE
            printf( "-> %s °C\n", tmp);
#endif
        }

        pubmsg.payload = tmp;
        pubmsg.payloadlen = (int)strlen(tmp);
        pubmsg.qos = QOS;
        pubmsg.retained = 0;
        if ((rc = MQTTClient_publishMessage(client, TOPIC, &pubmsg, &token)) != MQTTCLIENT_SUCCESS) {
            printf("Failed to publish message, return code %d\n", rc);
            exit(EXIT_FAILURE);
        }

        rc = MQTTClient_waitForCompletion(client, token, TIMEOUT);
    }

    return EXIT_SUCCESS;
}
```

serial_to_localhost.c avec exemple ayant servi de base

localhost_to_ThingSpeak.py

```python
#!/usr/bin/env python3
# 20/01/2022
# Publish data with mqtt to a ThingSpeak server
# OPEN-SOURCE

import paho.mqtt.publish as publish
import paho.mqtt.client as mqtt
import string
import socket
import time

#############
# on_message #
#############
def on_message(client,userdata, msg):
    message_received=str(msg.payload.decode("utf-8"))
    channel_ID = "1635758"
    t_port = 80
    t_transport="websockets"
    mqtt_host = "mqtt3.thingspeak.com"
    mqtt_client_ID = "ORYNAwQDASUICCE9IAotHi8"
    mqtt_username  = "ORYNAwQDASUICCE9IAotHi8"
    mqtt_password  = "Uew5ZBCRq0jae84GAbjrN/ad"
    topic_ = "channels/" + channel_ID + "/publish"
    data_type=message_received.split(" :")[0]
    data=message_received.split(": ")[1]
    if(data_type == "Relative humidity"):
        payload="field1="+data
    elif(data_type == "Temperature"):
        payload="field2="+data

    try:
       print("\n~Raspberry Pi~")
       print("Sending to ThingSpeak ...")
       print(data_type, " : ", data, "\n")
       publish.single(topic_, payload, hostname=mqtt_host, transport=t_transport,
                      port=t_port, client_id=mqtt_client_ID, auth={'username':mqtt_username,'password':mqtt_password})
    except Exception as e:
        print (e)


if __name__ == '__main__':
    topic = "Measurements"
    client = mqtt.Client("test")
    client.on_message = on_message
    client.connect("localhost",1883,60)
    client.subscribe(topic)

    try:
       client.loop_forever()
    except KeyboardInterrupt:
        client.disconnect()
        client.loop_stop()

```

localhost_to_ThingSpeak.py

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

École Publique d'Ingénieurs en 3 ans

6 boulevard Maréchal Juin, CS 45053
14050 CAEN cedex 04

ministère éducation nationale enseignement supérieur recherche

Normandie Université

Cti Commission des titres d'ingénieur

cdefi Conférence des Directeurs des Écoles Françaises d'Ingénieurs

European Accreditation of Engineering Programmes EUR-ACE®

CONFÉRENCE DES GRANDES ÉCOLES