IoT project template

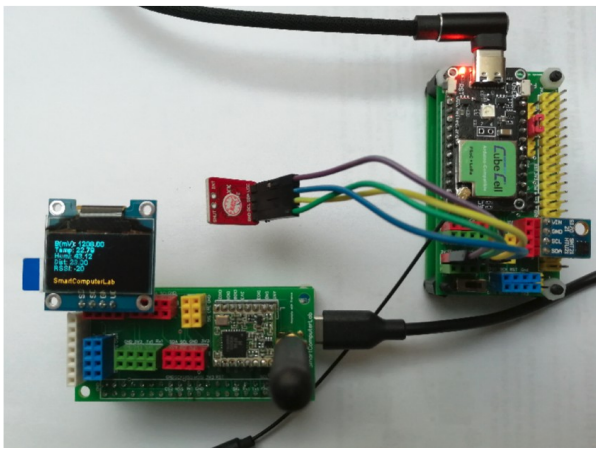
IoT : Sensors, LoRa link & Gateway

Content

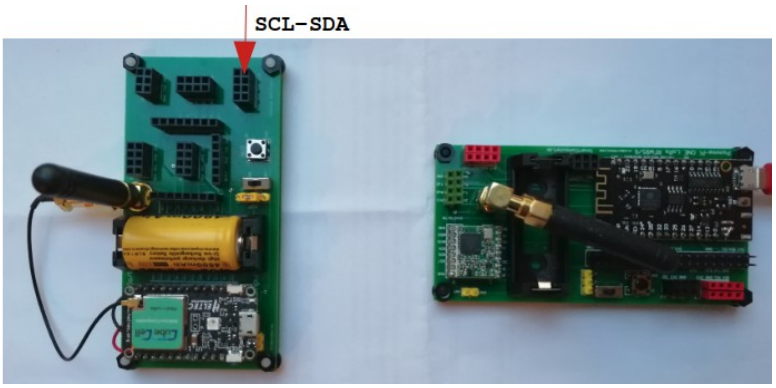
In this exercise we are going to use two different IoT DevKits

- Pomme-Pi ZERO (**`AirM2M-CORE_ESP32C3`**) with LoRa HAT DevKit and
- CubleCell: Pico Devkit and ESP32C3-CORE based board – **`CubeCell-Board (HTCC-AB01)`**

as shown on the following figure:



We also can take CubeCell board and ESP32-(**`LOLIN D32`**) based boards.



Both boards are programmed with **Arduino IDE** or **PlatformIO**.

Attention:
The code for CubeCell and CubeCell Pico DevKits operating as terminal nodes are completely compatible.
The receiver/gateway boards Pomme-Pi ONE with **`ESP32 LOLIN D32`** main board and Pomme-Pi ZERO with **`AirM2M-CORE_ESP32C3`** main board have **different pin-outs for LoRa modem**.

```
//For LOLIN D32:
#define SCK     18   // GPIO18 -- SX127x's SCK
#define MISO    19   // GPIO19 -- SX127x's MISO
#define MOSI    23   // GPIO23 -- SX127x's MOSI
#define SS       5   // GPIO05 -- SX127x's CS
#define RST     15   // GPIO15 -- SX127x's RESET
#define DI0     26   // GPIO26 -- SX127x's IRQ(Interrupt Request)

//For  AirM2M-CORE_ESP32C3
#define SS 7          // GPIO07 -- SX127x's CS
#define RST 6         // GPIO06 -- SX127x's RESET
#define DI0 11        // GPIO11 -- SX127x's IRQ(Interrupt Request)
#define SCK 2         // GPIO02 -- SX127x's SCK
#define MISO 10       // GPIO10 -- SX127x's MISO
#define MOSI 3        // GPIO03 -- SX127x's MOSI
```

1.1 Sender side on CubeCell based boards

The proposed code operates with the “transparent” gateway to ThingSpeak .com (TS) server. The LoRa packet format contains the place for TS **channel number** that **identifies** at the same time the **terminal application** and the place for a **16-byte write key** to this channel. The total size of the packet is 36 bytes. Four floating point variables are reserved for the sensor data, that are:

- battery state
- temperature (SHT21 on any other temperature sensor)
- humidity (SHT21 on any other humidity sensor)
- distance from the detected surface (VL53L1X ToF sensor)

1.1.1 Code explanation

Note that the sender operates in 4 stages:

1. low_power stage (seconds, minutes, hours,..)
2. sensing stage
3. transmission stage (data), and
4. reception stage (ACK)

```
typedef union
{
 uint8_t frame[36]; // frames with bytes
 struct
 {
 unsigned int channel; // 4 bytes - device and channel identifier
 char   wkey[16];      // 16 bytes
 float  sensor[4];     // 16 bytes
 } pay;                // payload
} pack_t ;  // packet type
```

The LoRa (SX1262) modem is connected internally via SPI bus. We need however define Lora radio link parameters:

```
#define RF_FREQUENCY                                868500000 // Hz
#define TX_OUTPUT_POWER                             14        // dBm
#define LORA_BANDWIDTH                              0         // [0: 125 kHz,
                                                              //  1: 250 kHz,
                                                              //  2: 500 kHz,
                                                              //  3: Reserved]
#define LORA_SPREADING_FACTOR                       9         // [SF7..SF12]
#define LORA_CODINGRATE                             1         // [1: 4/5,
                                                              //  2: 4/6,
                                                              //  3: 4/7,
                                                              //  4: 4/8]
#define LORA_PREAMBLE_LENGTH                        8         // Same for Tx and Rx
#define LORA_SYMBOL_TIMEOUT                         0         // Symbols
#define LORA_FIX_LENGTH_PAYLOAD_ON                  false
#define LORA_IQ_INVERSION_ON                        false
```

You can modify these values but they should be the same at the receiver side.

The synchronization/communication between the MCU (ARM-CORTEX-M0) and the modem is controlled by the **radio events**. To these events 3 "**automatic**" functions are attached. They allow to signal that the end of transmission (BUSY line) or transmission/reception timeouts.

```
#define RX_TIMEOUT_VALUE                            2000   // time to wait for ACK packet
static RadioEvents_t RadioEvents;

void OnTxDone( void );
void OnTxTimeout( void );
void OnRxTimeout( void );
```

In the **`setup()`** function the code uses the defined parameters and functions to initialize the LoRa modem and direct radio events to the execution routines.

```
void setup()
{
  Serial.begin(9600);
  rssi=0;
// activated radio events ! With links (addresses) to routines
  RadioEvents.TxDone = OnTxDone;
  RadioEvents.TxTimeout = OnTxTimeout;
  RadioEvents.RxDone = OnRxDone;
  RadioEvents.RxTimeout = OnRxTimeout;
  Radio.Init( &RadioEvents );
  Radio.SetChannel( RF_FREQUENCY );
  Radio.SetTxConfig( MODEM_LORA, TX_OUTPUT_POWER, 0, LORA_BANDWIDTH,
                                 LORA_SPREADING_FACTOR, LORA_CODINGRATE,
                                 LORA_PREAMBLE_LENGTH, LORA_FIX_LENGTH_PAYLOAD_ON,
                                 true, 0, 0, LORA_IQ_INVERSION_ON, 3000 );

  Radio.SetRxConfig( MODEM_LORA, LORA_BANDWIDTH, LORA_SPREADING_FACTOR,
                                  LORA_CODINGRATE, 0, LORA_PREAMBLE_LENGTH,
                                  LORA_SYMBOL_TIMEOUT, LORA_FIX_LENGTH_PAYLOAD_ON,
                                  0, true, 0, 0, LORA_IQ_INVERSION_ON, true );

}
```

In the **first operational stage** the sender enters into **low_power** mode (deep-sleep). This function is provided via **`lowPowerSleep(uint32_t sleeptime)`** function. This function uses a **`sleepTimer`** and **`wakeUp`** function that captures the end of sleep period.
When **`sleepTimerExpired`** is set to **`true`**., the **low_power** stage terminates.

```
TimerEvent_t sleepTimer;
bool sleepTimerExpired;  //Records whether our sleep/low power timer expired

static void wakeUp()
{
  sleepTimerExpired=true;
}

static void lowPowerSleep(uint32_t sleeptime)
{
  sleepTimerExpired=false;
  TimerInit( &sleepTimer, &wakeUp );
  TimerSetValue( &sleepTimer, sleeptime );
  TimerStart( &sleepTimer );
  //Low power handler also gets interrupted by other timers
  //So wait until our timer had expired
  while (!sleepTimerExpired) lowPowerHandler();
  TimerStop( &sleepTimer );
}
```

The **sensing stage** starts with the reading of battery state (analog value):

```
uint16_t read_Bat()
{
  uint16_t v;
  delay(40);
  pinMode(VBAT_ADC_CTL,OUTPUT);
  digitalWrite(VBAT_ADC_CTL,LOW);
  v=analogRead(ADC)+550; //*2;
  pinMode(VBAT_ADC_CTL, INPUT);
  return v;
}
```

Note that the **`analogRead(ADC)`** function **should be calibrated** in order to correspond to the real voltage values. Then follows the sequence of sensor readings. The sensors are attached (mainly) to I2C bus that should be alimented before the activation and use:

```
  pinMode(Vext, OUTPUT); delay(100);
  digitalWrite(Vext, LOW);  delay(100);
  Wire.begin();delay(100);

  // sensor readings
```

Depending on your application, it is provided TS channel number you may provide up to 4 sensor values. We have prepared 3 functions to get : **temperature** (T), **humidity** (H), and the **distance** (D).

```
float read_Temp()
{
  float t;
  t = sht.getTemperature();  // get temp from SHT
  return t;
}

int distance=0;

void read_Dist()
{
  if (distanceSensor.begin() != 0) //Begin returns 0 on a good init
  {
    Serial.println("Sensor failed to begin. Please check wiring. Freezing...");
    while (1) ;
  }
  Serial.println("Sensor VL53L1X online!");
  //distanceSensor.setDistanceModeShort();
  distanceSensor.setDistanceModeLong();
  distanceSensor.startRanging(); //Write configuration bytes to initiate measurement
  while (!distanceSensor.checkForDataReady())
  {
    delay(1);
  }
  distance = distanceSensor.getDistance(); //Get the result of the measurement from the sensor
  distanceSensor.clearInterrupt();
  distanceSensor.stopRanging();
}
float read_Humi()
{
  float h;
  h = sht.getHumidity();  // get temp from SHT
  return h;
}
```

The sensing stage terminates with:

```
  Wire.end();  delay(100);
  digitalWrite(Vext, HIGH); delay(100);
```

to **cutoff the power** for sensors.
Then starts the **transmission stage** to send our data packet with 36 bytes:

```
  turnOnRGB(COLOR_SEND,0);           // red color
  Radio.Send((uint8_t *)sdp.frame,36);
  delay(400);  // min sending time before the reception
```

The transmission stage **ends** with:

```
void OnTxDone( void )
{
  Serial.print("TX done!\n");
  turnOnRGB(0,0);
  lora_idle = true;
}
```

to **signal** the **correct** transmission

or with:

```
void OnTxTimeout( void )
{
    turnOffRGB();
    Radio.Sleep( );
    Serial.print("TX Timeout......");
}
```

to **signal** transmission **problem** and **timeout**

The **last**, **fourth stage**, is the **reception** of the **ACK** packet. This is done within the following loop waiting for the packet:

```
  lora_idle = true;
  while(lora_idle)  // waiting for ACK packet or timeout
    {
    turnOffRGB();
    lora_idle = false;
    Serial.println("RX mode");
    Radio.Rx(0);
    }
```

As in the transmission stage , we can have here two issues, the correct one, or the timeout. They are captured by the following “automatic” functions:

```
void OnRxDone( uint8_t *payload, uint16_t size, int16_t rssi, int8_t snr )
{
  pack_t rp;
    memcpy(rp.frame, payload, size );
    turnOnRGB(COLOR_RECEIVED,0);
    Radio.Sleep( );
    Serial.printf("\nrssi=%d ,length=%d, timeout=%d\n",rssi,size,(int)rp.pay.sensor[0]);
    ttsleep = (int)rp.pay.sensor[0]*200;delay(100);
    lora_idle = true;turnOffRGB();
}

void OnRxTimeout( void )
{
    turnOnRGB(COLOR_RECEIVED,0);
    Radio.Sleep( );
    Serial.print("RX Timeout......");delay(100);
    lora_idle = true;turnOffRGB();
}
```

1.1.2 Complete code

```
#include "LoRaWan_APP.h"
#include "Arduino.h"
#include <SHT21.h>
SHT21 sht;
#include "SparkFun_VL53L1X.h"
//Click here to get the library: http://librarymanager/All#SparkFun_VL53L1X
SFEVL53L1X distanceSensor;

static uint16_t counter=0;

#ifndef LoraWan_RGB
#define LoraWan_RGB 0
#endif
#define RF_FREQUENCY                                868500000 // Hz
#define TX_OUTPUT_POWER                             14        // dBm
#define LORA_BANDWIDTH                              0         // [0: 125 kHz,
                                                              //  1: 250 kHz,
                                                              //  2: 500 kHz,
                                                              //  3: Reserved]
#define LORA_SPREADING_FACTOR                       9         // [SF7..SF12]
#define LORA_CODINGRATE                             1         // [1: 4/5,
                                                              //  2: 4/6,
                                                              //  3: 4/7,
                                                              //  4: 4/8]
#define LORA_PREAMBLE_LENGTH                        8         // Same for Tx and Rx
#define LORA_SYMBOL_TIMEOUT                         0         // Symbols
#define LORA_FIX_LENGTH_PAYLOAD_ON                  false
#define LORA_IQ_INVERSION_ON                        false

#define RX_TIMEOUT_VALUE                            2000   // time to wait for ACK packet
```

```
static RadioEvents_t RadioEvents;
void OnTxDone( void );
void OnTxTimeout( void );
void OnRxTimeout( void );
int16_t rssi,rxSize;
uint8_t sent=0;

//Some utilities for going into low power mode
TimerEvent_t sleepTimer;
bool sleepTimerExpired;  //Records whether our sleep/low power timer expired

static void wakeUp()
{
  sleepTimerExpired=true;
}

static void lowPowerSleep(uint32_t sleeptime)
{
  sleepTimerExpired=false;
  TimerInit( &sleepTimer, &wakeUp );
  TimerSetValue( &sleepTimer, sleeptime );
  TimerStart( &sleepTimer );
  //Low power handler also gets interrupted by other timers
  //So wait until our timer had expired
  while (!sleepTimerExpired) lowPowerHandler();
  TimerStop( &sleepTimer );
}

typedef union
{
 uint8_t frame[36]; // frames with bytes
 struct
 {
 unsigned int channel; // 4 bytes - device and channel identifier
 char   wkey[16];      // 16 bytes
 float  sensor[4];     // 16 bytes
 } pay;                // payload
} pack_t ;  // packet type
pack_t sdp;  // packet to send
uint16_t read_Bat()

{
  uint16_t v;
  delay(40);
  pinMode(VBAT_ADC_CTL,OUTPUT);
  digitalWrite(VBAT_ADC_CTL,LOW);
  v=analogRead(ADC)+550; //*2;
  pinMode(VBAT_ADC_CTL, INPUT);
  return v;
}

float read_Temp()
{
  float t;
  t = sht.getTemperature();  // get temp from SHT
  return t;
}

int distance=0;

void read_Dist()
{
  if (distanceSensor.begin() != 0) //Begin returns 0 on a good init
  {
    Serial.println("Sensor failed to begin. Please check wiring. Freezing...");
    while (1) ;
  }
  Serial.println("Sensor VL53L1X online!");
  //distanceSensor.setDistanceModeShort();
  distanceSensor.setDistanceModeLong();
  distanceSensor.startRanging(); //Write configuration bytes to initiate measurement
  while (!distanceSensor.checkForDataReady())
  {
    delay(1);
  }
```

```
  distance = distanceSensor.getDistance(); //Get the result of the measurement from the sensor
  distanceSensor.clearInterrupt();
  distanceSensor.stopRanging();
}

float read_Humi()
{
  float h;

  h = sht.getHumidity();  // get temp from SHT
  return h;
}

unsigned long myChannelNumber =1538804;
const char *myWriteAPIKey="YOX31M0EDKO0JATK" ;
const char *myReadAPIKey="20E9AQVFW7Z6XXOM" ;

void setup()
{
  Serial.begin(9600);
  rssi=0;
// radio events links to routines
  RadioEvents.TxDone = OnTxDone;
  RadioEvents.TxTimeout = OnTxTimeout;
  RadioEvents.RxDone = OnRxDone;
  RadioEvents.RxTimeout = OnRxTimeout;

  Radio.Init( &RadioEvents );
  Radio.SetChannel( RF_FREQUENCY );
  Radio.SetTxConfig( MODEM_LORA, TX_OUTPUT_POWER, 0, LORA_BANDWIDTH,
                                  LORA_SPREADING_FACTOR, LORA_CODINGRATE,
                                  LORA_PREAMBLE_LENGTH, LORA_FIX_LENGTH_PAYLOAD_ON,
                                  true, 0, 0, LORA_IQ_INVERSION_ON, 3000 );

  Radio.SetRxConfig( MODEM_LORA, LORA_BANDWIDTH, LORA_SPREADING_FACTOR,
                                   LORA_CODINGRATE, 0, LORA_PREAMBLE_LENGTH,
                                   LORA_SYMBOL_TIMEOUT, LORA_FIX_LENGTH_PAYLOAD_ON,
                                   0, true, 0, 0, LORA_IQ_INVERSION_ON, true );

}

int ttsleep=20000;
bool lora_idle = true;

void loop()
{
  counter++;
  Serial.printf("\ntime_to_sleep=%d ms\n", ttsleep);delay(100);
  lowPowerSleep(ttsleep);
  Serial.printf("\nBack from sleep %d, counter=%d\n", millis(),counter);
  sdp.pay.sensor[0]=(float)read_Bat();Serial.println(sdp.pay.sensor[0]);
  pinMode(Vext, OUTPUT); delay(100);
  digitalWrite(Vext, LOW);  delay(100);
  Wire.begin();delay(100);
  sdp.pay.channel=myChannelNumber;
  memcpy(sdp.pay.wkey,myWriteAPIKey,16);
  sdp.pay.sensor[1]=(float)read_Temp();Serial.println(sdp.pay.sensor[1]);delay(100);
  sdp.pay.sensor[2]=(float)read_Humi();Serial.println(sdp.pay.sensor[2]);delay(100);
  read_Dist();
  sdp.pay.sensor[3]=(float)distance;Serial.println(sdp.pay.sensor[3]);delay(100);

  Wire.end();  delay(100);
  digitalWrite(Vext, HIGH); delay(100);
  turnOnRGB(COLOR_SEND,0);
  Radio.Send((uint8_t *)sdp.frame,36);
  delay(400);  // min sending time before the reception
  lora_idle = true;
  while(lora_idle)
    {
    turnOffRGB();
    lora_idle = false;               // works without timeout
    Serial.println("ACK wait");
    Radio.Rx(0);
    }
}
```

```
void OnRxDone( uint8_t *payload, uint16_t size, int16_t rssi, int8_t snr )
{
  pack_t rp;
    memcpy(rp.frame, payload, size );
    turnOnRGB(COLOR_RECEIVED,0);
    Radio.Sleep( );
    Serial.printf("\nrssi=%d,length=%d, id=%u, timeout=%d\n",rssi,size,rp.pay.channel,
(int)rp.pay.sensor[0]);
    ttsleep = (int)rp.pay.sensor[0]*200;delay(100);
    lora_idle = false;turnOffRGB();
}

void OnTxDone( void )
{
  Serial.print("TX done!\n");
  turnOnRGB(0,0);
  lora_idle = true;
}

void OnTxTimeout( void )
{
    turnOffRGB();
    Radio.Sleep( );
    Serial.print("TX Timeout......");
}

void OnRxTimeout( void )
{
    turnOnRGB(COLOR_RECEIVED,0);
    Radio.Sleep( );
    Serial.print("RX Timeout......");delay(100);
    lora_idle = false;turnOffRGB();
}
```

1.2 Receiver/gateway side on Pomme-Pi ZERO

(AirM2M–CORE_ESP32C3)

At the receiver side we have a complete Lora-WiFi gateway to ThingSpeak server.
The gateway is "**transparent**" , it knows neither the channel number nor the write key necessary to register the data packet on the ThingSpeak server. He knows however the **IP address** of the **ThingSpeak.com** server (port number **80**);

1.2.1 Code explanation

The initialization of the gateway communication links is done in setup() function. The WiFi connection is activated via WiFiManager that creates a soft AP on the board and a simple WEB server to capture the WiFi credentials (SSID,PASS) using an external client (PC,smartphone,..).

```
  WiFi.mode(WIFI_STA);
  WiFiManager wm;
  // wm.resetSettings();  // to clear the registered credentials in EEPROM
  bool res;
  res = wm.autoConnect("ESP32AP",NULL); // no password
  if(!res) Serial.println("Failed to connect");// ESP.restart();
  else  Serial.println("connected...yeey :)");
  ThingSpeak.begin(client); // TCP connection to ThingSpeak server
  delay(1000);
```

After the initialization , the board WiFi switches to station mode and is ready to send the data to external server such as **ThinSpeak.com**.
LoRa modem (RFM-SX1276) is "**connected**" and activated in the next step by:

```
#define BAND 8685E5
int sf=9;
long sbw=125E3;

...
  SPI.begin(2,10,3,7); // SCK, MISO, MOSI, SS
  LoRa.setPins(7, 6, 11);  // SS, RST, DI0 – interrupt
  delay(1000);
  Serial.println();Serial.println();
  Serial.println("Starting LoRa Receiver");
  if (!LoRa.begin(BAND)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  else
   {
   Serial.println("Starting LoRa ok!");
   LoRa.setSpreadingFactor(sf);
   LoRa.setSignalBandwidth(sbw);
   LoRa.setCodingRate4(5);
```

Finally, if LoRa modem is correctly initialized, we create **ParseTask** to read cyclically the LoRa modem input buffer looking for a new packet. When a new packet is found it is read and sent to the **ts_queue**.
In this queue we can store up to 4 messages 36 byte each.

```
   ts_queue = xQueueCreate(4,36); // queue for 4 data packets
   xTaskCreate(
                   ParseTask,   /* Function to implement the task */
                   "ParseTask", /* Name of the task */
                   10000,      /* Stack size in words */
                   NULL,       /* Task input parameter */
                   0,          /* Priority of the task */
                   NULL);       /* Task handle. */
  Serial.println("Task Parse created...");
```

The **ParseTask** looks in the input buffer, and if a new packet with the correct number of bytes is found (**packetLen==36**), the task reads data to the **pack_t** type **union**. Then, after a short delay, the task sends back the corresponding ACK packet.
In the ACK packet the task sends back the channel number (**sdp.pay.channel=rdp.pay.channel;**) and a new **timeout value** in **sdp.pay.sensor[0]** that is a random value (**random(60,80)**). This value is multiplied at the receiver/terminal side by 20 to get a number in milliseconds.

Note that **only the terminals** with the same identifier – channel number, should be concerned by the received ACK packet.

```
void ParseTask( void * pvParameters )
{
  pack_t rdp,sdp;
  while(true)
  {
    int packetLen;
    packetLen=LoRa.parsePacket();
    if(packetLen==36)
    {
      int i=0;
      while (LoRa.available()) {
      rdp.frame[i]=LoRa.read();i++;
    }
    rssi=LoRa.packetRssi();
    delay(1000);
    LoRa.beginPacket();  // sending ACK packet
    sdp.pay.channel=rdp.pay.channel;
    sdp.pay.sensor[0]=(float)random(60,80);sdp.pay.sensor[1]=(float)random(10,80);
    LoRa.write(sdp.frame,36);
    LoRa.endPacket();
    xQueueReset(ts_queue); // to keep only the last element
    xQueueSend(ts_queue, &rdp, portMAX_DELAY);
    }
  }
  }
```

The last instruction in the **`ParseTask`** is :

```
    xQueueSend(ts_queue, &rdp, portMAX_DELAY);
```

It allows the task to communicate the received packet to another task that is the main **`loop()`** task.

```
void loop()
{
pack_t rp;
char d1[32],d2[32],d3[32],d4[32], d5[32];
  xQueueReceive(ts_queue,rp.frame,portMAX_DELAY); // 6s,default:portMAX_DELAY
  if(millis()> lastsent+15000)
    {
    ThingSpeak.setField(1,rp.pay.sensor[0]);
    ThingSpeak.setField(2,rp.pay.sensor[1]);
    ThingSpeak.setField(3,rp.pay.sensor[2]);
    ThingSpeak.setField(4,rp.pay.sensor[3]);
    ThingSpeak.setField(5,rssi);
    Serial.printf("d1=%2.2f,d2=%2.2f,d3=%2.2f,d4=%2.2f\n",rp.pay.sensor[0],rp.pay.sensor[1],
                                                          rp.pay.sensor[2],rp.pay.sensor[3]);
    while (WiFi.status() != WL_CONNECTED) { delay(500); }
    int x = ThingSpeak.writeFields(rp.pay.channel, rp.pay.wkey);
    if(x == 200){Serial.println("Channel update successful.");}
    else { Serial.println("Problem updating channel. HTTP error code " + String(x));}
    sprintf(d1,"B(mV): %2.2f",rp.pay.sensor[0]);sprintf(d2,"Temp: %2.2f",rp.pay.sensor[1]);
    sprintf(d3,"Humi: %2.2f",rp.pay.sensor[2]);sprintf(d4,"Lumi: %2.2f",rp.pay.sensor[3]);
    sprintf(d5,"RSSI: %d",rssi);
    disp(d1,d2,d3,d4,d5);
    lastsent=millis();
    }
}
```

The main task reads the received packets in the **`ts_queue`** and if time moment allows sends the sensor values from the packet to ThingSpeak server.
The **time moment** is controlled by :

```
        if(millis()> lastsent+15000)
        {
..
        lastsent=millis();
        }
```

That provides a **dynamic delay** of 15 seconds.

1.2.2 Complete code

```
#include <LoRa.h>
#include <Wire.h>
#include <WiFiManager.h>
#include "ThingSpeak.h"
#include "SSD1306Wire.h"
SSD1306Wire display(0x3c, 4, 5); // OLED: ADDRESS, SDA, SCL

// the following parameters are not required for transparent gateway
//unsigned long myChannelNumber =1697980;
//const char *myWriteAPIKey="4K897XNNHTW7I4NO" ;
//const char *myReadAPIKey="V9GT1RN2FIP0YRAY" ;

WiFiClient  client;

// with LoRa modem RFM95 (HAT) -  AirM2M-CORE_ESP32C3

#define SS 7
#define RST 6
#define DI0 11
#define SCK 2
#define MISO 10
#define MOSI 3
#define BAND 8685E5

int sf=9;
long sbw=125E3;
typedef union
{
 uint8_t frame[36]; // frames with bytes
 struct
 {
 unsigned int channel; // 4 bytes - device and channel identifier
 char   wkey[16];      // 16 bytes
 float  sensor[4];     // 16 bytes
 } pay;                // payload
} pack_t ;  // packet type

QueueHandle_t ts_queue;  // queues for data packets
int rssi=0;

void ParseTask( void * pvParameters )
{
  pack_t rdp,sdp;
  while(true)
  {
    int packetLen; char b1[32];
    packetLen=LoRa.parsePacket();
    if(packetLen==36)
    {
      int i=0;
      while (LoRa.available()) {
      rdp.frame[i]=LoRa.read();i++;
    }
    rssi=LoRa.packetRssi();
    delay(1000);
    LoRa.beginPacket();  // sending ACK packet
    sdp.pay.sensor[0]=(float)random(60,80);sdp.pay.sensor[1]=(float)random(10,80);
    LoRa.write(sdp.frame,36);
    LoRa.endPacket();
    xQueueReset(ts_queue); // to keep only the last element
    xQueueSend(ts_queue, &rdp, portMAX_DELAY);
    }
  }
  }

void disp(char *l1, char *l2, char *l3, char *l4, char *l5)
{
  display.init();
  //display.flipScreenVertically();
  display.setFont(ArialMT_Plain_10);
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.setFont(ArialMT_Plain_10);
  display.drawString(0, 0, l1);
  display.drawString(0, 9, l2);
```

```
  display.drawString(0, 18, l3);
  display.drawString(0, 27, l4);
  display.drawString(0, 36, l5);
  display.drawString(0, 52, "SmartComputerLab");
  display.display();
}

void setup() {
  Serial.begin(9600);
  WiFi.mode(WIFI_STA);
  WiFiManager wm;
  // wm.resetSettings();
  bool res;
  res = wm.autoConnect("ESP32AP",NULL); // no password
  if(!res) Serial.println("Failed to connect");// ESP.restart();
  else  Serial.println("connected...yeey :)");

  ThingSpeak.begin(client); // connexion (TCP) du client au serveur
  delay(1000);
  Serial.println("ThingSpeak begin");
  SPI.begin(2, 10, 3, 7); // SCK, MISO, MOSI, SS
  LoRa.setPins(SS, RST, DI0);
  delay(1000);
  Serial.println();Serial.println();
  Serial.println("Starting LoRa Receiver");
  if (!LoRa.begin(BAND)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  else
   {
   Serial.println("Starting LoRa ok!");
   LoRa.setSpreadingFactor(sf);
   LoRa.setSignalBandwidth(sbw);
   LoRa.setCodingRate4(5);
   ts_queue = xQueueCreate(4,36); // queue for 4 data packets
   xTaskCreate(
                   ParseTask,   /* Function to implement the task */
                   "ParseTask", /* Name of the task */
                   10000,      /* Stack size in words */
                   NULL,       /* Task input parameter */
                   0,          /* Priority of the task */
                   NULL);       /* Task handle. */
  Serial.println("Task Parse created...");
   }
}

uint32_t mindel=15000;  // 15 seconds
long lastsent=0;
void loop()
{
pack_t rp;
char d1[32],d2[32],d3[32],d4[32], d5[32];

  xQueueReceive(ts_queue,rp.frame,portMAX_DELAY); // 6s,default:portMAX_DELAY

  if(millis()> lastsent+15000)
    {
    ThingSpeak.setField(1,rp.pay.sensor[0]);
    ThingSpeak.setField(2,rp.pay.sensor[1]);
    ThingSpeak.setField(3,rp.pay.sensor[2]);
    ThingSpeak.setField(4,rp.pay.sensor[3]);
    ThingSpeak.setField(5,rssi);
    Serial.printf("d1=%2.2f,d2=%2.2f,d3=%2.2f,d4=%2.2f\n",rp.pay.sensor[0],rp.pay.sensor[1],
rp.pay.sensor[2],rp.pay.sensor[3]);
    while (WiFi.status() != WL_CONNECTED) { delay(500);  }
    int x = ThingSpeak.writeFields(rp.pay.channel, rp.pay.wkey);
    if(x == 200){Serial.println("Channel update successful.");}
    else { Serial.println("Problem updating channel. HTTP error code " + String(x));}
    sprintf(d1,"B(mV): %2.2f",rp.pay.sensor[0]);sprintf(d2,"Temp: %2.2f",rp.pay.sensor[1]);
    sprintf(d3,"Humi: %2.2f",rp.pay.sensor[2]);sprintf(d4,"Lumi: %2.2f",rp.pay.sensor[3]);
    sprintf(d5,"RSSI: %d",rssi);
    disp(d1,d2,d3,d4,d5);
    lastsent=millis();
    }
}
```

To do

1. Read the explanations and analyze the presented code solution for your application.
2. Adapt the provided codes to your board (SPI and LoRa modem activation !)
3. Analyze power consumption for your terminal node
4. Add secure communication with AES