Lab 7 - Communication LoRa (Long Range) au niveau de la couche physique avec le modem SX127X

Dans ce laboratoire, nous allons étudier et expérimenter la technologie LoRa au niveau de la couche physique à l'aide des modems Semtech **SX1276**/**78** .
Nous étudierons et expérimenterons avec les exemples de transmission et de réception des paquets LoRa (trames) au niveau physique.

7.1 Liaison radio LoRa

LoRa est un schéma de modulation à spectre étalé propriétaire dérivé de **Chirp Spread Spectrum** modulation (**CSS**). Son débit de données est faible mais sa bande passante du canal fixe est d'une grande sensibilité.
LoRa est une implémentation de couche PHY et est indépendante des implémentations des couches supérieures. Cela permet à LoRa de coexister et d'interagir avec les architectures réseau existantes.
Dans ce paragraphe, nous expliquons certains des concepts de base de la modulation LoRa et les avantages de son schéma de modulation.

7.1.1 Communication avec un spectre étalé

En théorie de l'information, le théorème de Shannon-Hartley indique la vitesse maximale à laquelle l'information peut être transmise sur un canal de communication d'une largeur de bande spécifiée en présence de bruit.
Le théorème établit la **capacité** du canal de Shannon pour une liaison de communication et définit le débit de données maximal pouvant être transmis dans une bande passante spécifiée en présence des interférences.

$$\mathbf{C = B*log_2(1+ S/N)}$$

Où:

- **C** = capacité du canal (bit / s)
- **B** = bande passante du canal (Hz)
- **S** = puissance moyenne du signal reçu (Watts)
- **N** = bruit moyen ou puissance d'interférence (Watts)
- **S**/**N** = rapport signal sur bruit (**SNR**) exprimé sous forme de rapport de puissance linéaire

En passant l'équation ci-dessus de la base logarithmique 2 au $\mathbf{log_e}$ naturel, et en notant que $\mathbf{ln = log_e}$ on obtient l'équation suivante :

$$\mathbf{C/B = 1.433*S/N}$$

Pour les applications à spectre étalé, le rapport signal sur bruit est faible, car la puissance du signal est souvent inférieure au plancher de bruit. En supposant un niveau de bruit tel que **S**/**N<<1**, l'équation ci-dessus peut être réécrite comme :

$$\mathbf{C/B \sim= S/N} \quad \text{ou} \quad \mathbf{N/S \sim= B/C}$$

D'après la dernière équation, on peut voir que pour transmettre des informations sans erreur dans un canal avec le rapport de signal/bruit fixe, seule la largeur de bande du signal transmis doit être augmentée.
En augmentant la bande passante du signal, nous pouvons compenser la dégradation du rapport signal sur bruit (ou bruit sur signal) d'un canal radio.

Dans les systèmes traditionnels à spectre étalé avec la séquence directe (**DSSS**), la **phase** de la porteuse change conformément à une séquence de codes. Ce processus est généralement réalisé en multipliant le signal de données voulu avec un code d'étalement, également connu sous le nom de **chip sequence**.
La **chip sequence** se produit à une vitesse beaucoup plus rapide que le signal de données et étale ainsi la bande passante du signal au-delà de la bande passante d'origine occupée uniquement par le signal d'origine.

Notez que le terme **chip** est utilisé pour distinguer les bits codés des bits non codés.

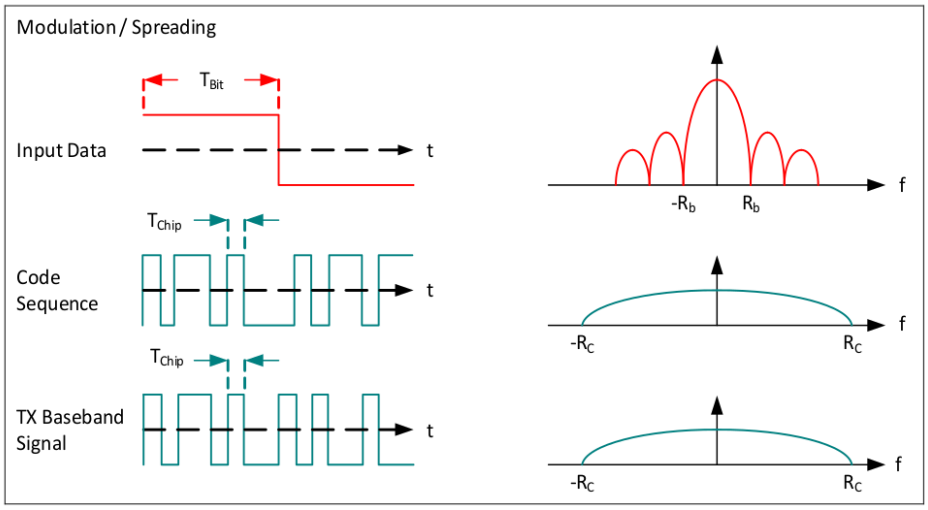


Figure 7.1 Modulation avec étalement du signal

Au niveau du récepteur, le signal de données utiles est récupéré en multipliant à nouveau avec une réplique générée localement de la séquence d'étalement. Ce processus de multiplication dans le récepteur « **comprime** » efficacement les signal non étalée d'origine, comme illustré ci-dessous.

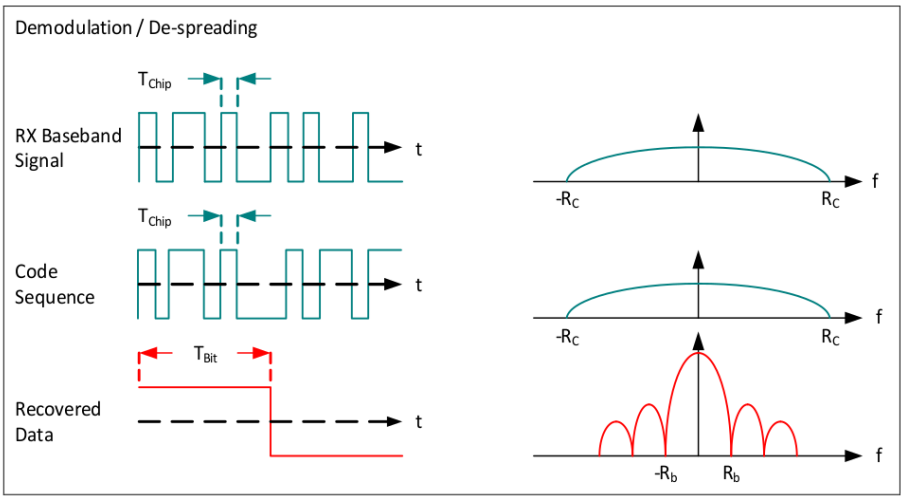


Figure 7.2 Démodulation avec étalement du signal

Il convient de noter que la même séquence ou le même **chip sequence** doit être utilisé dans le récepteur que dans le émetteur pour récupérer correctement les informations.
La quantité d'étalement, pour la séquence directe, dépend du rapport de **chips** par bit - le rapport de **chip sequence (Rc)**au débit de données souhaité (**Rb**), est appelé gain de traitement (**Gp**), communément exprimé en **dB**.

$$G_p = 10*\log_{10}(Rc/Rb) \ \ (dB)$$

Où:

Rc = **chip rate** (chips/second)
Rb = **bit-rate** (bits/second

En plus de fournir un gain de traitement inhérent à la transmission souhaitée (ce qui permet au récepteur de récupérer correctement le signal de données même lorsque le **SNR** du canal est une valeur **négative** (**en dB**); les signaux interférant sont également réduits par le gain de processus du récepteur.
Ceux-ci sont réparties au-delà de la bande passante et peuvent être facilement supprimés par filtrage.

Le **DSSS** est largement utilisé dans les applications de communication de données. Cependant, des défis existent pour les dispositifs à faible puissance d'alimentation.
En général ce système nécessite une source d'horloge de référence très précise.

Plus le code ou la séquence d'étalement est longue, plus long est le temps nécessaire au récepteur pour effectuer une corrélation sur toute la longueur de la séquence du code.

Ceci est particulièrement préoccupant pour les appareils à faible puissance qui ne peuvent pas être «toujours allumés» et qui doivent donc synchroniser rapidement à plusieurs reprises.

La modulation **LoRa** de **Semtech** résout tous les problèmes associés aux systèmes DSSS pour fournir une alternative économique de faible consommation, mais surtout très robuste.

Dans la modulation LoRa, l'étalement du spectre est obtenu en générant un signal de **chirp** qui **varie continuellement en fréquence**.

Un avantage de cette méthode est que les **décalages de synchronisation et de fréquence entre l'émetteur et le récepteur sont équivalents**, ce qui réduit considérablement la complexité de la conception du récepteur.

La largeur de **bande de fréquence** de ce **chip** est équivalente à la largeur de bande spectrale du signal.

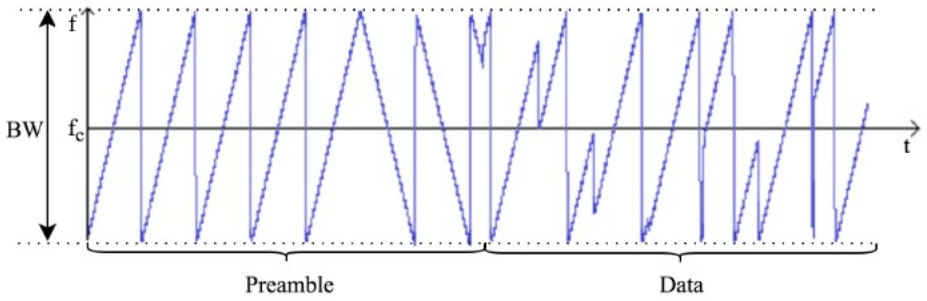


Figure 7.3 Variation de fréquence dans la bande passante du signal

Initialement, le flux d'informations binaires généré à partir de la couche physique est divisé en sous-séquences, chacune de longueur **SF∈ [7 ... 12].** L'ensemble des bits **SF** consécutifs constitue un symbole.
Le nombre de symboles possibles est donc égal à $\mathbf{M=2^{SF}}$.
Par conséquent, la relation entre le débit binaire **Rb** et le débit de symboles **Rs** peut être écrite comme suit :

$$\mathbf{Rb = SF*Rs}$$

Le spectre étalé est obtenu par un signal connu sous le nom de **chirp** dont la fréquence varie de manière continue et linéaire. Lorsque la dérivée de la variation de fréquence est positive, alors nous traitons un **chirp ascendant**, inversement c'est un **chirp descendant**.

La relation entre le débit binaire de données souhaité, le débit de **symboles** et le débit binaire pour la modulation LoRa peut s'exprimer comme suit:

$$\mathbf{Rb = SF * BW/2^{SF}\ \ bits/sec}$$

Où :

SF = facteur d'étalement (**7..12**)
BW = bande passante de la modulation (**Hz**)

La modulation LoRa comprend également un schéma de correction d'erreur variable qui améliore la robustesse du signal transmis.

Le **Rate Code** correspondant est :

$$\mathbf{Rate = 4/(4+CR)}$$

Où CR sont des **bits de correction** (1**..4**) pour **4 bits de données**.
Nous pouvons réécrire le débit binaire nominal comme suit:

$$\mathbf{Rb = Rate * SF * BW/2^{SF}\ \ bits/sec}$$

Par exemple pour: **CR=4**, **SF=7** et **BW** 125 Khz on obtient:

$$\mathbf{Rb = (4/(4+4))*7*125000/128 = \ 3418\ bits/sec}$$

the data rate of **3418 bits/sec**

Ces paramètres influencent également la **sensibilité** du décodeur - récepteur. D'une manière générale, une augmentation de la bande passante diminue la sensibilité du récepteur, tandis qu'une augmentation du facteur d'étalement augmente la sensibilité du récepteur.
La diminution du débit de code permet de réduire le taux d'erreur sur les paquets (**PER**) en présence de courtes rafales d'interférences. Un paquet transmis avec un débit de code de **CR = 4/8** sera plus tolérant aux interférences qu'un signal transmis avec un **CR** de **4/5**.

Les chiffres du tableau 1, issus de la fiche technique **SX1276**, sont donnés à titre indicatif.

BW \ SF	7	8	9	10	11	12
125 kHz	−123	−126	−129	−132	−133	−136
250 kHz	−120	−123	−125	−128	−130	−133
500 kHz	−116	−119	−122	−125	−128	−130

Tableau 7.1 La sensibilité du récepteur en fonction des paramètres **SF** et **BW**

7.2 LoRa communication with SX1276/78 modem

Le protocole de liaison radio LoRa a été implémenté par Semtech dans les modems SX1276 /78. Nous allons utiliser ces circuits dans nos laboratoires.

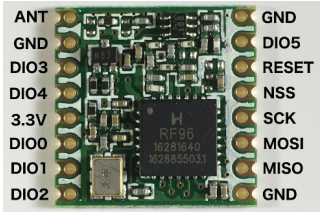


Figure 7.4 RFM96 module avec le modem SX1278

7.2.1 LoRa - Packet Mode

En mode Paquet, les données NRZ vers (depuis) le (dé) modulateur sont stockées dans la FIFO et sont accessibles via l'interface SPI.

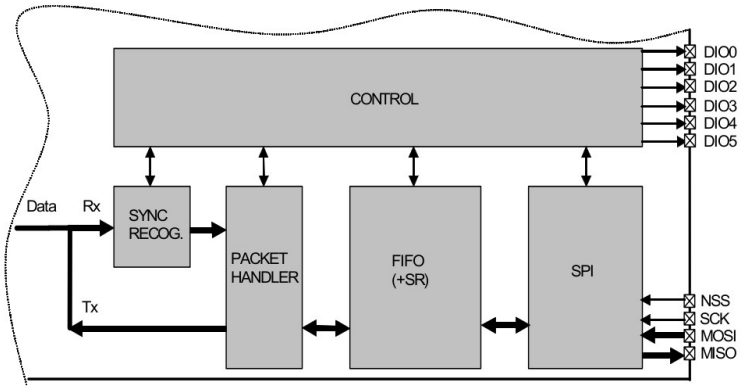


Figure 7.5 Modem SX1278 fonctionnant en mode paquet avec taille variable des paquets

De plus, le gestionnaire de paquets SX1276 / 77/78/79 effectue plusieurs tâches orientées paquets telles que la génération de mots de préambule et de synchronisation, le calcul/contrôle CRC, l'encodage/décodage Manchester, le filtrage d'adresses, etc.
Cela simplifie le logiciel et réduit la surcharge de traitement en effectuant ces tâches répétitives dans le circuit RF lui-même. Une autre caractéristique importante est la capacité de remplir et de vider la FIFO en mode veille (standby) , garantissant une consommation d'énergie optimale et ajoutant plus de flexibilité au logiciel.

7.2.2 LoRa - format de paquet de longueur variable (SX1276/78)

Ce mode est utile dans les applications où la longueur du paquet n'est pas connue à l'avance et peut varier dans le temps. Il est alors nécessaire que l'émetteur envoie les informations de longueur avec chaque paquet pour que le récepteur fonctionne correctement.

Dans ce mode, la longueur de la charge utile, indiquée par l'octet de longueur (Length byte), est donnée par le premier octet du FIFO et est limitée à 255 octets. Notez que l'octet de longueur lui-même n'est pas inclus dans ce calcul. Dans ce mode, la charge utile doit contenir au moins 2 octets, c'est-à-dire longueur + adresse ou octet de message.

Le format de ce type de paquet est illustré dans la figure suivante.

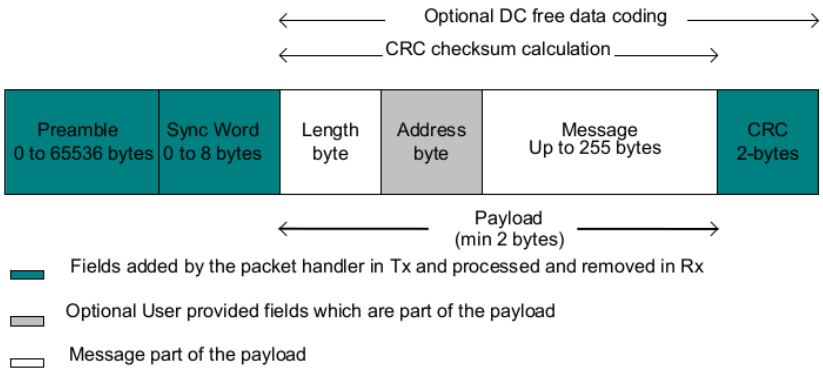


Figure 7.6 Format du paquet de la taille variable

Ce paquet contient les champs suivants:

- Préambule (1010 ...)
- Mot de synchronisation (ID réseau)
- Octet de longueur
- Octet d'adresse facultatif (ID de nœud)
- Données de message

7.2.2.1 Préambule

Le préambule est utilisé pour synchroniser le récepteur avec le flux de données entrant. Par défaut, le paquet est configuré avec une longue séquence de 12 symboles. Il s'agit d'une variable programmable de sorte que la longueur du préambule peut être modifié, par exemple dans l'intérêt de la réduction du cycle de service du récepteur dans les applications de réception intensive.

Cependant, la longueur minimale suffit pour toutes les communications.

La longueur de préambule transmis peut être modifiée en réglant le registre **`PreambleLength`** de **6 à 65535 symboles**,.

Le préambule avec 8 symboles est utilisé dans LoRaWAN.

Le récepteur entreprend un processus de détection de préambule qui redémarre périodiquement. Pour cette raison, la longueur du préambule doit être configurée de la même manière que la longueur du préambule de l'émetteur. Lorsque la longueur du préambule n'est pas connue ou peut varier, la longueur maximale du préambule doit être programmée du côté du récepteur.

Sync Word

Le filtrage/reconnaissance de mots de synchronisation (**SyncWord**) est utilisé pour identifier le début de la charge utile (*payload*) et également pour l'identification du réseau. Le mot de synchronisation peut être ajouté dans dans la transmission et dans la réception.
Chaque paquet reçu qui ne démarre pas avec le mot de synchronisation configuré localement est automatiquement rejeté et aucune interruption n'est générée.

Lorsque le mot **Sync** correspondant est détecté, la réception de la charge utile démarre automatiquement et **`SyncAddressMatch`** est affirmé.

7.2.2.2 En-tête (*header*)

Selon le mode de fonctionnement choisi, deux types d'en-tête sont disponibles. Le type d'en-tête est sélectionné par le bit **`ImplicitHeaderModeOn`** trouvé dans le registre **`RegModemConfig1`**.

Explicit Header Mode
Il s'agit du mode de fonctionnement par défaut. Ici, l'en-tête fournit des informations sur la charge utile, à savoir:

- La longueur de la charge utile en octets.
- Le taux de code de correction d'erreurs direct
- La présence d'un CRC 16 bits optionnel pour la charge utile.

Charge utile (*payload*)
La charge utile du paquet est un champ de longueur variable qui contient les données réelles codées comme spécifié dans l'en-tête en mode explicite ou dans les paramètres de registre en mode implicite.
Un **CRC** facultatif peut être ajouté.

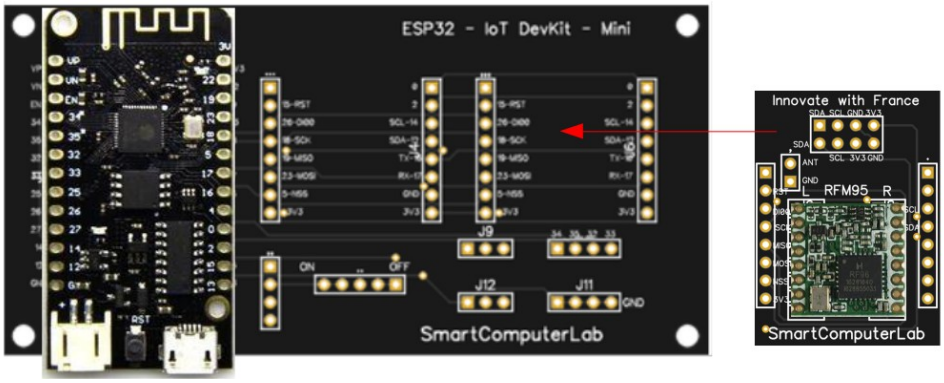
Inversion du QI
L'inversion du **QI** inverse la direction du changement de fréquence au fil du temps. Le début d'un paquet est le préambule qui est transmis avec le **réglage IQ opposé**. À la fin du préambule se trouve le mot de synchronisation, puis la charge utile avec le QI configuré. Les paramètres **IQ inversés** permettent aux paquets de liaison montante et descendante de provoquer très peu d'interférences lorsque le démodulateur suit le décalage de fréquence.

7.3 Programmation LoRa (SX1276/78) avec bibliothèque `LoRa.h`

Nos exemples de programmation sont basés sur la bibliothèque **`LoRa.h`** développée pour les modules RFM95/96. Le câblage avec le bus SPI dépend du type de carte ESP32.
Dans notre cas, nous utilisons la configurations suivante :

```
#include <SPI.h>
#include <LoRa.h>
#define SS       5  //  to NSS - chip select
#define RST     15  //  to - RST - reset
#define DI0     26  //  INTR  - IO interruption
#define SCK     18  //  CLK - SPI clock
#define MISO    19  //  MISO - master in slave out
#define MOSI    23  //  MOSI - master out slave in
```

La carte ESP32 (LOLIN32) est connecté avec un modem externe.



Regardons à l'intérieur de la **`LoRaClass`** - **`LoRa`** définie dans le fichier **`LoRa.h`**.

```
class LoRaClass : public Stream {
public:
  LoRaClass();

  int begin(long frequency);
  void end();

  int beginPacket(int implicitHeader = false);
  int endPacket(bool async = false);

  int parsePacket(int size = 0);
  int packetRssi();
  float packetSnr();
  long packetFrequencyError();

  int rssi();

  virtual size_t write(uint8_t byte);
  virtual size_t write(const uint8_t *buffer, size_t size);

  // from Stream
  virtual int available();
  virtual int read();
  virtual int peek();
  virtual void flush();

  void onReceive(void(*callback)(int));
```

```
  void onTxDone(void(*callback)());

  void receive(int size = 0);
  void idle();
  void sleep();

  void setTxPower(int level, int outputPin = PA_OUTPUT_PA_BOOST_PIN);
  void setFrequency(long frequency);
  void setSpreadingFactor(int sf);
  void setSignalBandwidth(long sbw);
  void setCodingRate4(int denominator);
  void setPreambleLength(long length);
  void setSyncWord(int sw);
  void enableCrc();
  void disableCrc();
  void enableInvertIQ();
  void disableInvertIQ();

  void setOCP(uint8_t mA); // Over Current Protection control

  void setGain(uint8_t gain); // Set LNA gain
  // deprecated
  void crc() { enableCrc(); }
  void noCrc() { disableCrc(); }

  byte random();

  void setPins(int ss = LORA_DEFAULT_SS_PIN, int reset = LORA_DEFAULT_RESET_PIN,
          int dio0 = LORA_DEFAULT_DIO0_PIN);
  void setSPI(SPIClass& spi);
  void setSPIFrequency(uint32_t frequency);

  void dumpRegisters(Stream& out);

private:
  void explicitHeaderMode();
  void implicitHeaderMode();
  void handleDio0Rise();
  bool isTransmitting();
  int getSpreadingFactor();
  long getSignalBandwidth();
  void setLdoFlag();
  uint8_t readRegister(uint8_t address);
  void writeRegister(uint8_t address, uint8_t value);
  uint8_t singleTransfer(uint8_t address, uint8_t value);
  static void onDio0Rise();

private:
  SPISettings _spiSettings;
  SPIClass* _spi;
  int _ss;
  int _reset;
  int _dio0;
  long _frequency;
  int _packetIndex;
  int _implicitHeaderMode;
  void (*_onReceive)(int);
  void (*_onTxDone)();
};

extern LoRaClass LoRa;
```

7.3.1 Configuration des paramètres de liaison physique

Les fonctions suivantes nous permettent de préparer les paramètres de modulation et de contrôle.
Pour démarrer et terminer le fonctionnement du modem, nous utilisons :

```
  int begin(long frequency);  // example: LoRa.begin(868e6)
  void end();
```

7.3.1.1 Paramètres de modulation et de contrôle des erreurs

```
  void setTxPower(int level, int outputPin = PA_OUTPUT_PA_BOOST_PIN);
  void setFrequency(long frequency);  // ex:  434e6, 868e6
  void setSpreadingFactor(int sf);    // ex: 9 (7..11)
  void setSignalBandwidth(long sbw);  // ex: 125e3  -  125 KHz
  void setCodingRate4(int denominator); // ex: 5  (5..8)
  void setPreambleLength(long length);  // ex: 12 (default 8 - symbols)
  void setSyncWord(int sw);             // character - ex: 0x3F (0x00-0xFF)
  void enableCrc();
  void disableCrc();
  void enableInvertIQ();               // use for down and up links
  void disableInvertIQ();              // use for down and up links
```

7.3.1.2 Création et envoi des paquets LoRa (trames)

```
uint8_t frame[32];   // packet - frame to send
int i=0;
  int beginPacket(int implicitHeader = false); // default beginPacket()
  virtual size_t write(uint8_t byte);   // ex: LoRa.write(buff[i]);
  virtual size_t write(const uint8_t *buffer, size_t size);
     // ex: LoRa.write(frame,32);
  int endPacket(bool async = false);   // ex: endPacket(true)
```

`endPacket()` envoie effectivement le paquet si le modem est activé.

7.3.1.3 Réception des paquets LoRa (trames)

Il existe deux façons de signaler la réception des paquets entrants:

- en interrogeant le tampon d'entrée
- via le signal d'interruption sur la broche **`DIO0`**

L'interrogation des paquets avec **`parsePacket()`** permet d'obtenir la taille du paquet transporté dans le champ **`length byte`**. Ensuite, la méthode **`LoRa.read()`** lit les octets du tampon d'entrée tant que les données sont disponibles - **`LoRa.available()`**.

```
uint8_t frame[32];   // packet - frame to send
int i=0;
int packetSize=parsePacket(); // parsing the input buffer
  if (packetSize) {
    while (LoRa.available()) {
      frame[i]=LoRa.read();i++; }
```

La deuxième façon, impliquant une **interruption**, nécessite la redirection du signal **IO** (pin **`DIO0`**) vers la fonction de rappel :

```
void onReceive(void(*callback)(int));

void onReceive(int packetSize) {  // this is ISR asynchronous routine
  Serial.print("Callaback - Received packet '");
  for (int i = 0; i < packetSize; i++) {
    Serial.print((char)LoRa.read());
  }
```

```
  // print RSSI of packet
  Serial.print("' with RSSI ");
  Serial.println(LoRa.packetRssi());
}
```

Dans la fonction **`setup()`**, nous devons préparer l'adresse de la redirection et mettre le modem en mode réception.

```
void setup()
{
  LoRa.onReceive(onReceive);
  LoRa.receive();  // put the radio into receive mode
..
```

Notez l'utilisation de la méthode **`LoRa.packetRssi()`** pour extraire la force du signal reçu.
Sa valeur varie de **-10** à **-120** selon la situation par rapport à l'émetteur et les paramètres de modulation.

7.3.1.4 Protéger le canal de communication

Les paramètres physiques de la modulation LoRa peuvent être complétés par l'**inversion IQ** et l'utilisation de **SyncWord**.
L'utilisation de l'inversion IQ permet la séparation des communications de **liaison montante** (nœuds **T** à **M**) et de **liaison descendante** (nœuds **M** à **T**).
M dénote un nœud **Master** et **T** dénote un nœud **Terminal**.

Par exemple, nous pouvons utiliser dans le nœud Terminal le mode inversé pour envoyer les paquets LoRa au nœud Maître, et le mode normal pour recevoir les paquets envoyés par le nœud Maître.

```
void LoRa_rxMode(){
  LoRa.enableInvertIQ();                // active invert I and Q signals
  LoRa.receive();                       // set receive mode
}

void LoRa_txMode(){
  LoRa.idle();                          // set standby mode
  LoRa.disableInvertIQ();               // normal mode
}
```

Après la transmission, le nœud **retourne implicitement** en mode réception via le rappel (callback) de **`onTxDone()`**.

```
void onTxDone() {
  Serial.println("TxDone");
  LoRa_rxMode();
}
```

Pour fournir une **communication exclusive** entre les nœuds désignés (notre réseau), nous pouvons utiliser **SyncWord** transporté dans chaque paquet LoRa.

```
  LoRa.setSyncWord(0xF3);           // ranges from 0-0xFF
```

Les nœuds qui n'utilisent pas le même SyncWord ne capturent pas les trames physiques.

7.4 Nœuds émetteurs et récepteurs simples

Il est maintenant temps de présenter les exemples complets de nœuds simples.

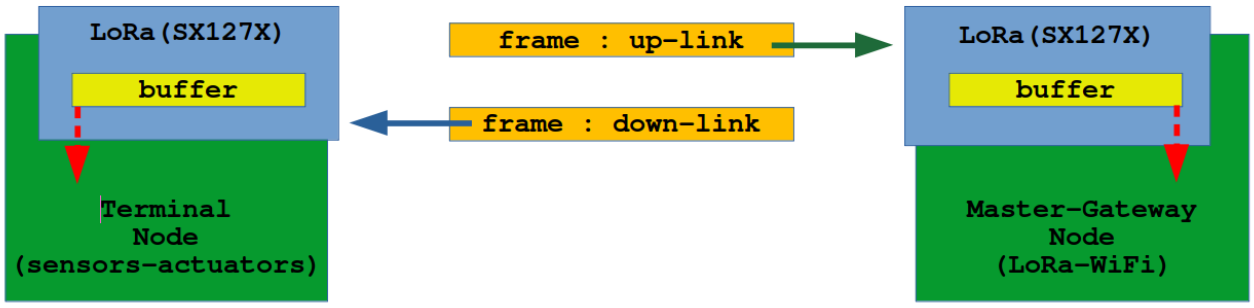


Figure 7.7 Canal de communication LoRa bidirectionnel entre un nœud Terminal et Maître

7.4.1 Expéditeur simple

Le code suivant implémente un simple expéditeur LoRa ; le facteur d'étalement et la bande passante du signal sont définis par les fonctions correspondantes.

```
#include <SPI.h>
#include <LoRa.h>
#define SS       5      //    NSS
#define RST     15      //    RST
#define DI0     26      //   INTR (DIO0)
#define SCK     18      //   CLK
#define MISO    19      //   MISO
#define MOSI    23      //   MOSI
#define BAND    434E6

int sf=7;
long sbw=125E3;

void setup() {
  Serial.begin(9600);  Serial.println();
  SPI.begin(SCK, MISO, MOSI, SS);  // SCK, MISO, MOSI, SS
  LoRa.setPins(SS, RST, DI0);
  if (!LoRa.begin(BAND)) {
    Serial.println("LoRa init failed. Check your connections.");
    while (true);                       // if failed, do nothing
  }
  delay(100);Serial.println();
  Serial.println("LoRa init succeeded.");
  LoRa.setSpreadingFactor(sf);
  LoRa.setSignalBandwidth(sbw);
  Serial.printf("SF set to: %d, Bandwidth set to: %d Hz\n",sf,sbw);
  delay(100);
}

int counter=0;
void loop() {
  Serial.print("Sending packet: ");
  Serial.println(counter);
  LoRa.beginPacket();
  LoRa.print("hello ");
  LoRa.print(counter);
  LoRa.endPacket();
  counter++;
  delay(5000);
}
```

7.4.2 Récepteur simple

Le code suivant implémente un simple récepteur LoRa utilisant les mêmes paramètres (SF, SWB) que l'expéditeur. La réception des trames LoRa se fait via l'interrogation du tampon d'entrée (*buffer polling*) avec **`packetParse()`**.

```
#include <SPI.h>
#include <LoRa.h>
#define SS       5      //  NSS
#define RST     15      //  RST
#define DI0     26      //  INTR
#define SCK     18      //  CLK
#define MISO    19      //  MISO
#define MOSI    23      //  MOSI
#define BAND    434E6
int sf=7;
long sbw=125E3;

void setup() {
  Serial.begin(9600);  Serial.println();
  SPI.begin(SCK, MISO, MOSI, SS);  // SCK, MISO, MOSI, SS
  LoRa.setPins(SS, RST, DI0);
  if (!LoRa.begin(BAND)) {
    Serial.println("LoRa init failed. Check your connections.");
    while (true);                       // if failed, do nothing
  }
  delay(100);Serial.println();
  Serial.println("LoRa init succeeded.");
  LoRa.setSpreadingFactor(sf);
  LoRa.setSignalBandwidth(sbw);
  Serial.printf("SF set to: %d, Bandwidth set to: %d Hz\n",sf,sbw);
  delay(100);
}

void loop() {
  int packetSize = LoRa.parsePacket();    // try to parse packet
  if (packetSize) {
    // received a packet
    Serial.print("Received packet '");
    // read packet
    while (LoRa.available()) {
      Serial.print((char)LoRa.read());
    }
    // print RSSI of packet
    Serial.print("' with RSSI ");
    Serial.println(LoRa.packetRssi());
  }
}
```

L'affichage résultant sur le terminal IDE.

```
LoRa init succeeded.
SF set to: 7, Bandwidth set to: 125000 Hz
Received packet 'hello 144' with RSSI -35
Received packet 'hello 145' with RSSI -35
Received packet 'hello 146' with RSSI -45
Received packet 'hello 147' with RSSI -34
Received packet 'hello 148' with RSSI -34
```

7.4.3 Récepteur simple avec fonction de rappel (*callback*)

Le code suivant implémente un simple récepteur LoRa utilisant les mêmes paramètres (SF, SWB) que l'expéditeur. La réception des trames LoRa se fait via le **signal d'interruption** (`DIO0`) capturé par la fonction de rappel **`onReceive()`**.

```
#include <SPI.h>
#include <LoRa.h>
#define SS       5       // NSS
#define RST     15       // RST
#define DI0     26       // INTR
#define SCK     18       // CLK
#define MISO    19       // MISO
#define MOSI    23       // MOSI
#define BAND    434E6

int sf=7;
long sbw=125E3;
```

```
void setup() {
  Serial.begin(9600);  Serial.println();
  SPI.begin(SCK, MISO, MOSI, SS);  // SCK, MISO, MOSI, SS
  LoRa.setPins(SS, RST, DI0);
  if (!LoRa.begin(BAND)) {
    Serial.println("LoRa init failed. Check your connections.");
    while (true);                       // if failed, do nothing
  }
  delay(100);Serial.println();
  Serial.println("LoRa init succeeded.");
  LoRa.setSpreadingFactor(sf);
  LoRa.setSignalBandwidth(sbw);
  Serial.printf("SF set to: %d, Bandwidth set to: %d Hz\n",sf,sbw);
  delay(100);
  LoRa.onReceive(onReceive);
  LoRa.receive();  // put the radio into receive mode
}

void loop() {
  // do nothing
}

void onReceive(int packetSize) {    // received a packet
  Serial.print("Callaback - Received packet '");
  for (int i = 0; i < packetSize; i++) {
    Serial.print((char)LoRa.read());
  }
  Serial.print("' with RSSI ");  // print RSSI of packet
  Serial.println(LoRa.packetRssi());
}
```

7.5 Communication en mode duplex

Les exemples suivants montrent comment construire les liens bidirectionnels en utilisant une simple interrogation du tampon d'entrée ou la fonction **`onReceive()`.**

7.5.1 Duplex avec la fonction `parsePacket()`

Le programme suivant envoie un message toutes les demi-secondes et interroge continuellement les nouveaux messages entrants. Il implémente un **schéma d'adressage sur un octet**, avec **`0xFF`** comme **adresse de diffusion**.
Il utilise **`readString()`** de la classe **`Stream`** pour lire la charge utile.

7.5.1.1 Code complet pour la communication LoRa duplex avec `parsePacket()`

```
#include <SPI.h>              // include libraries
#include <LoRa.h>

#define SS       5       // NSS
#define RST     15       // RST
#define DI0     26       // INTR
#define SCK     18       // CLK
#define MISO    19       // MISO
#define MOSI    23       // MOSI
#define BAND    434E6

String outgoing;              // outgoing message
byte msgCount = 0;            // count of outgoing messages
byte localAddress = 0xBB;     // address of this device
byte destination = 0xFF;      // destination to send to
long lastSendTime = 0;        // last send time
int interval = 2000;          // interval between sends

int sf=7;
long sbw=125E3;

void setup() {
  Serial.begin(9600);  Serial.println();
  SPI.begin(SCK, MISO, MOSI, SS);  // SCK, MISO, MOSI, SS
  LoRa.setPins(SS, RST, DI0);
  if (!LoRa.begin(BAND)) {
    Serial.println("LoRa init failed. Check your connections.");
    while (true);                       // if failed, do nothing
  }
  delay(100);Serial.println();
  Serial.println("LoRa init succeeded.");
  LoRa.setSpreadingFactor(sf);
  LoRa.setSignalBandwidth(sbw);
  Serial.printf("SF set to: %d, Bandwidth set to: %d Hz\n",sf,sbw);
  delay(100);
}

void loop() {
  if (millis() - lastSendTime > interval) {
    String message = "HeLoRa World!";   // send a message
    sendMessage(message);
    Serial.println("Sending " + message);
    lastSendTime = millis();            // timestamp the message
    interval = random(2000) + 1000;    // 2-3 seconds
  }
  // parse for a packet, and call onReceive with the result:
  onReceive(LoRa.parsePacket());
}

void sendMessage(String outgoing) {
  LoRa.beginPacket();                   // start packet
  LoRa.write(destination);              // add destination address
  LoRa.write(localAddress);             // add sender address
  LoRa.write(msgCount);                 // add message ID
  LoRa.write(outgoing.length());        // add payload length
  LoRa.print(outgoing);                 // add payload
  LoRa.endPacket();                     // finish packet and send it
  msgCount++;                           // increment message ID
}
```

```
void onReceive(int packetSize) {
  if (packetSize == 0) return;          // if there's no packet, return
  int recipient = LoRa.read();          // recipient address
  byte sender = LoRa.read();            // sender address
  byte incomingMsgId = LoRa.read();     // incoming msg ID
  byte incomingLength = LoRa.read();    // incoming msg length
  String incoming = "";
  while (LoRa.available()) {
    incoming += (char)LoRa.read();
  }
  if (incomingLength != incoming.length()) {   // check length
    Serial.println("error: message length does not match length");
    return;                             // skip rest of function
  }
  // if the recipient isn't this device or broadcast,
  if (recipient != localAddress && recipient != 0xFF) {
    Serial.println("This message is not for me.");
    return;                             // skip rest of function
  }
  // if message is for this device, or broadcast, print details:
  Serial.println("Received from: 0x" + String(sender, HEX));
  Serial.println("Sent to: 0x" + String(recipient, HEX));
  Serial.println("Message ID: " + String(incomingMsgId));
  Serial.println("Message length: " + String(incomingLength));
  Serial.println("Message: " + incoming);
  Serial.println("RSSI: " + String(LoRa.packetRssi()));
  Serial.println("Snr: " + String(LoRa.packetSnr()));
  Serial.println();
}
```

7.5.1.2 Duplex simple avec rappel

L'exemple suivant envoie un message toutes les demi-secondes et utilise la **fonction de rappel** pour les nouveaux messages entrants. Il implémente un schéma d'adressage sur un octet, avec **0xFF** comme adresse de diffusion.

Attention :
Lors de l'envoi, la radio LoRa n'écoute pas les messages entrants. Lorsque vous utilisez la méthode de rappel type **ISR (Interrupt Service Routine)**, vous ne pouvez utiliser aucune des fonctions **`Stream`** qui dépendent du **délai d'expiration**, telles que **`readString()`**, **`parseInt()`**, etc.

```
#include <SPI.h>              // include libraries
#include <LoRa.h>
String outgoing;              // outgoing message
byte msgCount = 0;            // count of outgoing messages
byte localAddress = 0xBB;     // address of this device
byte destination = 0xFF;      // destination to send to
long lastSendTime = 0;        // last send time
int interval = 2000;          // interval between sends

#define SS       5      //  NSS
#define RST     15      //  RST
#define DI0     26      // INTR
#define SCK     18      // CLK
#define MISO    19      // MISO
#define MOSI    23      // MOSI
#define BAND    434E6

int sf=7;
long sbw=125E3;

void setup() {
  Serial.begin(9600);  Serial.println();
  SPI.begin(SCK, MISO, MOSI, SS);  // SCK, MISO, MOSI, SS
  LoRa.setPins(SS, RST, DI0);
  if (!LoRa.begin(BAND)) {
    Serial.println("LoRa init failed. Check your connections.");
    while (true);                       // if failed, do nothing
  }
  delay(100);Serial.println();
  Serial.println("LoRa init succeeded.");
  LoRa.setSpreadingFactor(sf);
  LoRa.setSignalBandwidth(sbw);
```

```
  Serial.printf("SF set to: %d, Bandwidth set to: %d Hz\n",sf,sbw);
  delay(100);
  Serial.println("LoRa init succeeded.");
  LoRa.onReceive(onReceive);
  LoRa.receive();
}

void loop() {
  if (millis() - lastSendTime > interval) {
    String message = "HeLoRa World!";
    sendMessage(message);
    Serial.println("Sending " + message);
    lastSendTime = millis();            // timestamp the message
    interval = random(4000) + 2000;     // 2-6 seconds
  }
}

void sendMessage(String outgoing) {
  LoRa.beginPacket();                   // start packet
  LoRa.write(destination);              // add destination address
  LoRa.write(localAddress);             // add sender address
  LoRa.write(msgCount);                 // add message ID
  LoRa.write(outgoing.length());        // add payload length
  LoRa.print(outgoing);                 // add payload
  LoRa.endPacket();                     // finish packet and send it
  msgCount++;                           // increment message ID
  LoRa.receive();
}

void onReceive(int packetSize) {
  if (packetSize == 0) return;          // if there's no packet, return
  int recipient = LoRa.read();          // recipient address
  byte sender = LoRa.read();            // sender address
  byte incomingMsgId = LoRa.read();     // incoming msg ID
  byte incomingLength = LoRa.read();    // incoming msg length

  String incoming = "";                 // payload of packet

  while (LoRa.available()) {       // can't use readString() in callback, so
    incoming += (char)LoRa.read();      // add bytes one by one
  }
  if (incomingLength != incoming.length()) {   // check length for error
    Serial.println("error: message length does not match length");
    return;                             // skip rest of function
  }
  // if the recipient isn't this device or broadcast,
  if (recipient != localAddress && recipient != 0xFF) {
    Serial.println("This message is not for me.");
    return;                             // skip rest of function
  }
  Serial.println("got message");
  // if message is for this device, or broadcast, print details:
  Serial.println("Received from: 0x" + String(sender, HEX));
  Serial.println("Message ID: " + String(incomingMsgId));
  Serial.println("Received message: " + incoming)
}
```

7.6 Communication LoRa simple en liaison montante et descendante avec inversion IQ

Le code suivant utilise la fonction **`InvertIQ`** pour créer une logique de communication Terminal-Master comme suit :

- **Maître** (passerelle) :
  - Envoie des messages avec **`enableInvertIQ()`**
  - Reçoit des messages avec **`disableInvertIQ()`**

- **Terminal** :
  - Envoie des messages avec **`disableInvertIQ()`**
  - Reçoit des messages avec **`enableInvertIQ()`**

Avec cet arrangement, un **Maître** ne reçoit jamais des messages d'un autre **Maître** et un **Terminal** ne reçoit jamais des messages d'un autre **Terminal**. Uniquement **Master** au **Terminal** et **vice versa**.
Le code suivant code reçoit des messages et envoie un message toutes les secondes. La fonction **`EnableInvertIQ()`** inverse les signaux **I** et **Q** de la modulation LoRa.

Les deux nœuds utilisent le même **SyncWord** défini par:

```
LoRa.setSyncWord(0xF3);         // set SyncWord
```

après l'initialisation réussie du modem LoRa.

7.6.1 Le code du Terminal avec inversion IQ à la réception

```
#include <SPI.h>
#include <LoRa.h>
#define SS       5      //  NSS
#define RST     15      //  RST
#define DI0     26      //  INTR
#define SCK     18      //  CLK
#define MISO    19      //  MISO
#define MOSI    23      //  MOSI
#define BAND    434E6

int sf=7;
long sbw=125E3;

void setup() {
  Serial.begin(9600);
  SPI.begin(SCK, MISO, MOSI, SS);  // SCK, MISO, MOSI, SS
  LoRa.setPins(SS, RST, DI0);
  if (!LoRa.begin(BAND)) {
    Serial.println("LoRa init failed. Check your connections.");
    while (true);                // if failed, do nothing
  }
  LoRa.setSyncWord(0xF3);        // set SyncWord
  Serial.println("LoRa init succeeded.");
  Serial.println();
  Serial.println("LoRa Simple Node");
  Serial.println("Only receive messages from gateways");
  Serial.println("Tx: invertIQ disable");
  Serial.println("Rx: invertIQ enable");
  Serial.println();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
}

void loop() {
  if (runEvery(1000)) { // repeat every 1000 millis
    String message = "HeLoRa World! ";
    message += "I'm a Node! ";
    message += millis();
    LoRa_sendMessage(message); // send a message
    Serial.println("Send Message!");
  }
}
```

```
void LoRa_rxMode(){
  LoRa.enableInvertIQ();                // active invert I and Q signals
  LoRa.receive();                       // set receive mode
}

void LoRa_txMode(){
  LoRa.idle();                          // set standby mode
  LoRa.disableInvertIQ();               // normal mode
}

void LoRa_sendMessage(String message) {
  LoRa_txMode();                        // set tx mode - normal mode
  LoRa.beginPacket();                   // start packet
  LoRa.print(message);                  // add payload
  LoRa.endPacket(true);                 // finish packet and send it
}

void onReceive(int packetSize) {
  String message = "";

  while (LoRa.available()) {
    message += (char)LoRa.read();
  }

  Serial.print("Node Receive: ");
  Serial.println(message);
}

void onTxDone() {
  Serial.println("TxDone");
  LoRa_rxMode();        // After transmission - reception mode with inverted IQ
}

boolean runEvery(unsigned long interval)
{
  static unsigned long previousMillis = 0;
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= interval)
  {
    previousMillis = currentMillis;
    return true;
  }
  return false;
}
```

7.6.2 Le code du Maître avec inversion IQ lors de la transmission

Le code suivant utilise la fonction **`InvertIQ`** pour envoyer les paquets Lora au nœud Terminal.

```
#include <SPI.h>              // include libraries
#include <LoRa.h>
#define SS       5      //  NSS
#define RST     15      //  RST
#define DI0     26      //  INTR
#define SCK     18      //  CLK
#define MISO    19      //  MISO
#define MOSI    23      //  MOSI
#define BAND    434E6

int sf=7;
long sbw=125E3;

void setup() {
  Serial.begin(9600);
  SPI.begin(SCK, MISO, MOSI, SS);  // SCK, MISO, MOSI, SS
  LoRa.setPins(SS, RST, DI0);
  if (!LoRa.begin(BAND)) {
    Serial.println("LoRa init failed. Check your connections.");
    while (true);                       // if failed, do nothing
  }
  Serial.println("LoRa init succeeded.");  Serial.println();
  LoRa.setSyncWord(0xF3);   // set SyncWord
  Serial.println("LoRa Simple Gateway");
  Serial.println("Only receive messages from nodes");
  Serial.println("Tx: invertIQ enable");
  Serial.println("Rx: invertIQ disable"); Serial.println();
```

```
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
}

void loop() {
  if (runEvery(5000)) { // repeat every 5000 millis
    String message = "HeLoRa World! ";
    message += "I'm a Gateway! ";
    message += millis();
    LoRa_sendMessage(message); // send a message
    Serial.println("Send Message!");
  }
}

void LoRa_rxMode(){
  LoRa.disableInvertIQ();               // normal mode
  LoRa.receive();                       // set receive mode
}

void LoRa_txMode(){
  LoRa.idle();                          // set standby mode
  LoRa.enableInvertIQ();                // active invert I and Q signals
}

void LoRa_sendMessage(String message) {
  LoRa_txMode();                        // set tx mode
  LoRa.beginPacket();                   // start packet
  LoRa.print(message);                  // add payload
  LoRa.endPacket(true);                 // finish packet and send it
}

void onReceive(int packetSize) {
  String message = "";
  while (LoRa.available()) {
    message += (char)LoRa.read();
  }
  Serial.print("Gateway Receive: ");
  Serial.println(message);
}

void onTxDone() {
  Serial.println("TxDone");
  LoRa_rxMode();
}

boolean runEvery(unsigned long interval)
{
  static unsigned long previousMillis = 0;
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= interval)
  {
    previousMillis = currentMillis;
    return true;
  }
  return false;
}
```

7.7 À faire:

1. Étudiez le schéma de modulation des modems LoRa et calculez le débit de données utile pour :
   SF = 11, BW = 500 kHz CR = 4/8
2. Expérimentez avec les codes d'expéditeur et de destinataire présentés.
3. Modifiez le facteur d'étalement (à 9 et 11) et la bande passante du signal (à 250 KHz et 500 KHz) et observez la force du signal pour les mêmes distances entre les nœuds. Utilisez :
   `Serial.println(LoRa.packetRssi());`
4. Modifiez le **SyncWord** dans un nœud et essayez de communiquer entre les nœuds émetteur et récepteur.

7.8 Annexe – `LoRa_Para.h`

Le fichier d'inclusion **`LoRa_Para.h`** a été préparé pour faciliter le développement des applications avec le modem **LoRa**. Cette bibliothèque permet de configurer les connections du modem et les paramètres radio pour la couche physique de communication.

```
// default values for pins and LoRa physical link – frame parameters
#define SS       5 // 26     // D0 – to NSS
#define RST     15  //16     // D4  – RST
#define DI0     26      // D8 – INTR
#define SCK     18      // D5 – CLK
#define MISO    19      // D6 – MISO
#define MOSI    23      // D7 – MOSI
#define BAND    868E6   // set frequency
#define SF      7       // set spreading factor
#define SBW     125E3   // set signal bandwidth
#define SW      0xF3    // set Sync Word
#define BR      8       // set bit rate (4/5,5/8)

void set_LoRa()  // all default settings
{
  SPI.begin(SCK, MISO, MOSI, SS);  // SCK, MISO, MOSI, SS
  LoRa.setPins(SS, RST, DI0);
  Serial.begin(9600);
  delay(1000);
  Serial.println();
  if (!LoRa.begin(BAND)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  sprintf(buff,"BAND=%f,SF=%d,SBW=%f,SW=%X,BR=%d\n",BAND,SF,SBW,SW,BR);
  Serial.println(buff);
  LoRa.setSpreadingFactor(SF);
  LoRa.setSignalBandwidth(SBW);
  LoRa.setSyncWord(SW);
}

// pins and parameters
void set_LoRa_Pins_Para(int sck,int miso,int mosi,int ss,int rst, int dio0,unsigned long
freq,unsigned sbw, int sf, uint8_t sw)
{
  SPI.begin(sck, miso, mosi, ss);  // SCK, MISO, MOSI, SS
  LoRa.setPins(ss, rst, dio0);
  Serial.begin(9600);
  delay(1000);
  Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  LoRa.setSpreadingFactor(sf);
  LoRa.setSignalBandwidth(sbw);
  LoRa.setSyncWord(sw);
}

// radio settings only
void set_LoRa_Para(unsigned long freq,unsigned sbw, int sf, uint8_t sw)
{
  SPI.begin(SCK, MISO, MOSI, SS);  // SCK, MISO, MOSI, SS
  LoRa.setPins(SS, RST, DI0);
  Serial.begin(9600);
  delay(1000);
  Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  LoRa.setSpreadingFactor(sf);
  LoRa.setSignalBandwidth(sbw);
  LoRa.setSyncWord(sw);
}

// Terminal IQ mode
#ifdef TERMINAL
```

```
void LoRa_rxMode(){
  LoRa.enableInvertIQ();                // active invert I and Q signals
  LoRa.receive();                       // set receive mode
}

void LoRa_txMode(){
  LoRa.idle();                          // set standby mode
  LoRa.disableInvertIQ();               // normal mode
}
#endif

// Gateway IQ mode
#ifdef GATEWAY
void LoRa_rxMode(){
  LoRa.disableInvertIQ();               // normal mode
  LoRa.receive();                       // set receive mode
}

void LoRa_txMode(){
  LoRa.idle();                          // set standby mode
  LoRa.enableInvertIQ();                // active invert I and Q signals
}
#endif

void onTxDone() {
  Serial.println("TxDone");
  LoRa_rxMode();
}

boolean runEvery(unsigned long interval)
{
  static unsigned long previousMillis = 0;
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= interval)
  {
    previousMillis = currentMillis;
    return true;
  }
  return false;
}
```

7.8.1 Terminal code avec IQ inversion et fichier `LoRa_Para.h`

```
#define TERMINAL
#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"

void LoRa_sendMessage(String message) {
  LoRa_txMode();                        // set tx mode - normal mode
  LoRa.beginPacket();                   // start packet
  LoRa.print(message);                  // add payload
  LoRa.endPacket(true);                 // finish packet and send it
}

void onReceive(int packetSize) {
  String message = "";
  while (LoRa.available()) {
    message += (char)LoRa.read();
  }
  Serial.print("Node Receive: ");
  Serial.println(message);
}

void setup() {
  Serial.begin(9600);
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
}

void loop() {
  if (runEvery(1000)) { // repeat every 1000 millis
    String message = "HeLoRa World! ";
```

```
        message += "I'm a Node! ";
    message += millis();
    LoRa_sendMessage(message); // send a message
    Serial.println("Send Message!");
  }
}
```

7.8.2 Gateway code avec IQ inversion et fichier `LoRa_Para.h`

```
#define GATEWAY
#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"

void LoRa_sendMessage(String message) {
  LoRa_txMode();                        // set tx mode
  LoRa.beginPacket();                   // start packet
  LoRa.print(message);                  // add payload
  LoRa.endPacket(true);                 // finish packet and send it
}

void onReceive(int packetSize) {
  String message = "";
  while (LoRa.available()) {
    message += (char)LoRa.read();
  }
  Serial.print("Master Receive: ");
  Serial.println(message);
  Serial.println(LoRa.packetRssi());
}

void setup() {
  Serial.begin(9600);
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
}

void loop() {
  if (runEvery(5000)) { // repeat every 5000 millis
    String message = "HeLoRa World! ";
    message += "I'm a Master! ";
    message += millis();
    LoRa_sendMessage(message); // send a message
    Serial.println("Send Message!");
  }
}
```

Lab 8 - Programmation basse consommation pour les terminaux ESP32 avec LoRa

Ce laboratoire est un guide pour l'ESP32 fonctionnant en mode basse consommation appelé mode veille profonde (*deep_sleep*). Nous allons vous montrer comment mettre l'ESP32 en veille profonde et examiner différents modes pour le réveiller: réveil par minuterie, réveil par les broches tactiles et réveil externe.
Ce laboratoire fournit des exemples pratiques de code et explique comment créer des nœuds de Terminal LoRa basse consommation.

8.1 Présentation des modes de veille (*sleep modes*)

Le SoC ESP32 peut basculer entre différents modes d'alimentation :

- Mode actif
- Mode veille du modem (*WiFi/BT modem-sleep*)
- Mode veille légère (*light sleep*)
- Mode veille profonde (*deep sleep*)
- Mode veille prolongée (*hibernation*)

Power mode	Active	Modem-sleep	Light-sleep	Deep-sleep	Hibernation
Sleep pattern	Association sleep pattern			ULP sensor-monitored pattern	-
CPU	ON	ON	PAUSE	OFF	OFF
Wi-Fi/BT baseband and radio	ON	OFF	OFF	OFF	OFF
RTC memory and RTC peripherals	ON	ON	ON	ON	OFF
ULP co-processor	ON	ON	ON	ON/OFF	OFF

Tableau 8.1 Cinq modes d'alimentation différents du SoC ESP32.

La fiche technique **ESP32 Espressif** fournit également un tableau comparatif de la consommation électrique des différents modes de puissance.

Power mode	Description	Power consumption
Active (RF working)	Wi-Fi Tx packet 14 dBm ~ 19.5 dBm	Please refer to Table 10 for details.
	Wi-Fi / BT Tx packet 0 dBm	
	Wi-Fi / BT Rx and listening	
Modem-sleep	The CPU is powered on.	Max speed 240 MHz: 30 mA ~ 50 mA
		Normal speed 80 MHz: 20 mA ~ 25 mA
		Slow speed 2 MHz: 2 mA ~ 4 mA
Light-sleep	-	0.8 mA
Deep-sleep	The ULP co-processor is powered on.	150 μA
	ULP sensor-monitored pattern	100 μA @1% duty
	RTC timer + RTC memory	10 μA
Hibernation	RTC timer only	5 μA
Power off	CHIP_PU is set to low level, the chip is powered off	0.1 μA

Tableau 8.2 Consommation d'énergie pour différents modes d'alimentation de l'ESP32.

Ci-dessous un tableau pour comparer la consommation électrique en mode actif.

Mode	Min	Typ	Max	Unit
Transmit 802.11b, DSSS 1 Mbps, POUT = +19.5 dBm	-	240	-	mA
Transmit 802.11b, OFDM 54 Mbps, POUT = +16 dBm	-	190	-	mA
Transmit 802.11g, OFDM MCS7, POUT = +14 dBm	-	180	-	mA
Receive 802.11b/g/n	-	95 ~ 100	-	mA
Transmit BT/BLE, POUT = 0 dBm	-	130	-	mA
Receive BT/BLE	-	95 ~ 100	-	mA

Tableau 8.3 Consommation d'énergie en mode actif.

8.1.1 Pourquoi le mode veille profonde ?

Faire fonctionner votre ESP32 en mode actif avec des piles n'est pas idéal, car l'énergie des piles s'épuise très rapidement. Si vous mettez votre ESP32 en mode veille profonde, cela réduira la consommation d'énergie et vos batteries dureront plus longtemps.
Avoir votre ESP32 en mode **veille profonde** signifie réduire les activités qui consomment plus d'énergie pendant le fonctionnement, mais laisser **juste assez d'activité** pour réveiller le processeur lorsque quelque chose d'intéressant se produit.
En mode de veille profonde, ni le processeur ni les activités Wi-Fi n'ont lieu, mais le coprocesseur **Ultra Low Power** (**ULP**) est toujours alimenté.

Pendant que l'ESP32 est en mode de veille profonde, la mémoire de l'horloge en temps réel **RTC** (*Real Time Clock*) reste également allumée, afin que nous puissions écrire un programme pour le coprocesseur ULP et le stocker dans la mémoire RTC pour accéder aux périphériques, aux minuteries internes et internes et aux capteurs.
Ce mode de fonctionnement est utile si vous devez réveiller le processeur principal par un **événement externe**, une **minuterie** ou les deux, tout en maintenant une consommation d'énergie minimale.

8.1.2 Broches `RTC_GPIO`

Pendant le sommeil profond, certaines des broches ESP32 peuvent être utilisées par le coprocesseur ULP, à savoir les broches **`RTC_GPIO`** et les broches tactiles. Jetons un coup d'œil au brochage suivant pour localiser les différentes broches **`RTC_GPIO`**.
Les broches **`RTC_GPIO`** sont mises en évidence par une case rectangulaire orange. Les broches **`GPIO6`** et **`GPIO11`**, non exposées,sont connectées au **flash SPI intégré**.

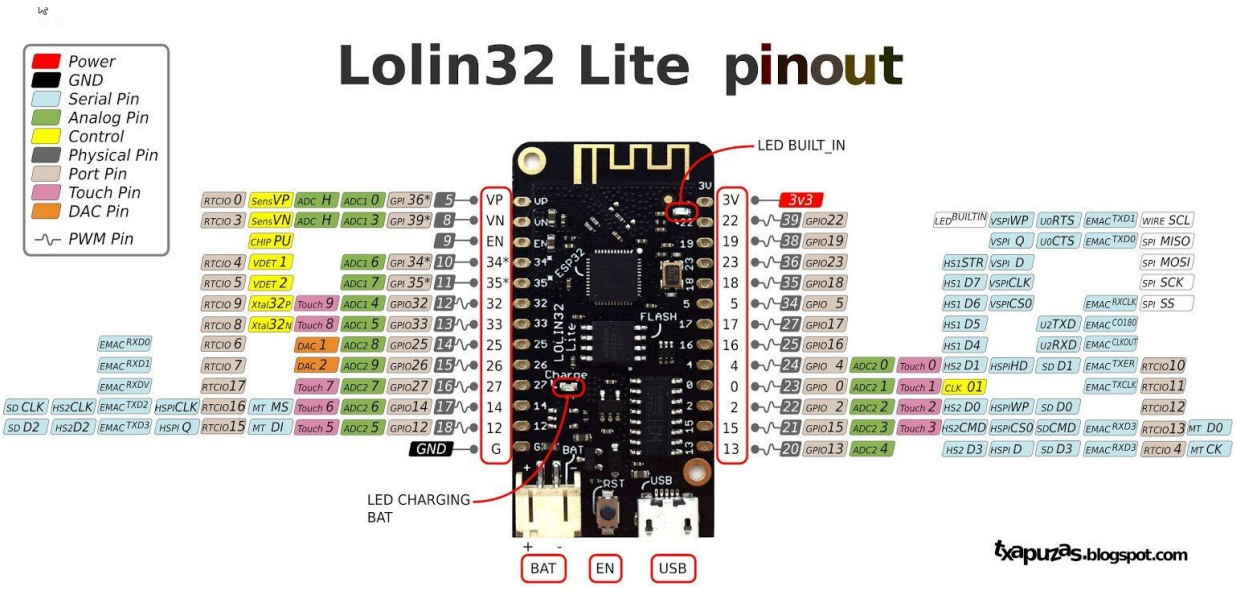


Figure 8.1 Broches **`GPIO`** ESP32 (Lolin32)

8.2 Sources du réveil (*Wake Up*)

Après avoir mis l'ESP32 en mode veille prolongée, il existe plusieurs façons de le réveiller :

1. Vous pouvez utiliser la minuterie pour réveiller votre ESP32 en utilisant des périodes de temps prédéfinies
2. Vous pouvez utiliser les broches tactiles
3. Vous pouvez utiliser deux possibilités de réveil externe : vous pouvez utiliser soit un **réveil externe**, soit plusieurs réveils externes différents
4. Vous pouvez utiliser le coprocesseur ULP - cela ne sera pas couvert dans ce guide.

8.3 Ecrire un programme pour le mode de sommeil profond

Pour écrire un programme pour mettre votre ESP32 en mode de veille profonde, puis le réactiver, vous devez garder à l'esprit que :

- Tout d'abord, vous devez **configurer les sources de réveil**. Cela signifie configurer ce qui réveillera l'ESP32. Vous pouvez utiliser une ou combiner plusieurs sources de réveil.
- Vous pouvez décider des périphériques à arrêter ou à conserver pendant le sommeil profond. Cependant, par défaut, l'ESP32 met automatiquement hors tension les périphériques qui ne sont pas nécessaires avec la source de réveil que vous définissez.
- Enfin, vous utilisez la fonction **`esp_deep_sleep_start()`** pour mettre votre ESP32 en mode sommeil profond.

8.3.1 Réveil par la minuterie (*Timer Wake Up*)

L'ESP32 peut passer en mode veille profonde, puis se réveiller à des périodes prédéfinies. Cette fonction est particulièrement utile si vous exécutez des projets qui nécessitent un horodatage ou des tâches quotidiennes (par exemple scrutation des capteurs), tout en maintenant une faible consommation d'énergie.
Le contrôleur ESP32 RTC dispose d'une minuterie intégrée que vous pouvez utiliser pour réveiller l'ESP32 après un laps de temps prédéfini.

8.3.1.1 Activer le réveil par minuterie

Activer l'ESP32 pour se réveiller après un laps de temps prédéfini est très simple. Dans l'IDE Arduino, il vous suffit de spécifier le **temps de veille en microsecondes** dans la fonction suivante:

```
esp_sleep_enable_timer_wakeup(time_in_us)
```

Exemple de code

Voyons comment cela fonctionne en utilisant un exemple de la bibliothèque. Ouvrez votre IDE Arduino, accédez à **`File`**>**`Examples`**>**`ESP32`**> **`Deep Sleep`** et ouvrez le croquis **`TimerWakeUp`**.

```
#define uS_TO_S_FACTOR 1000000  /* micro seconds to seconds */
#define TIME_TO_SLEEP  5        /* Time ESP32 will go 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("External signal RTC_IO");
         break;
    case ESP_SLEEP_WAKEUP_EXT1 : Serial.println("External RTC_CNTL"); break;
    case ESP_SLEEP_WAKEUP_TIMER : Serial.println("Timer"); break;
    case ESP_SLEEP_WAKEUP_TOUCHPAD : Serial.println("Touchpad"); break;
    case ESP_SLEEP_WAKEUP_ULP : Serial.println("ULP program"); break;
    default : Serial.printf("Not caused by deep sleep: %d\n",wakeup_reason);
         break;
  }
}
```

```
void setup(){
  Serial.begin(9600);
  delay(1000);
  ++bootCount;
  Serial.println("Boot number: " + String(bootCount));
  print_wakeup_reason(); // print the wakeup reason for ESP32
  // We set our ESP32 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");
 // The line below turns off all RTC peripherals in deep sleep.
 // esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH, ESP_PD_OPTION_OFF);
 // Serial.println("All RTC Peripherals to be powered down in sleep");
  Serial.println("Going to sleep now");
  delay(1000);
  Serial.flush();
  esp_deep_sleep_start();
  Serial.println("This will never be printed");
}

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

Jetons un coup d'œil sur ce code. Le premier commentaire décrit ce qui est éteint pendant le sommeil profond avec réveil par minuterie.
Dans ce mode, les processeurs, la plupart de la RAM et tous les périphériques numériques qui sont cadencés à partir d'**APB_CLK** sont mis hors tension. A noter que les minuteries sont cadencées par **APB_CLK** ; le manuel de référence technique indique que **APB_CLK** est dérivée de **CPU_CLK**.
Les seules parties du SoC qui peuvent encore être mises sous tension sont : le contrôleur **RTC**, les périphériques **RTC** et les mémoires **RTC**.
A noter que la fonction **void print_wakeup_reason)** peut être ajoutée à votre programme par

```
#include «ESP32_WakeUp.h»
```

répertoriée dans la partie Annexe de ce laboratoire.

8.3.1.2 Définir le temps de sommeil

Ces deux premières lignes de code définissent la période pendant laquelle l'ESP32 sera en veille.

```
#define uS_TO_S_FACTOR 1000000 /* Conversion from micro seconds to seconds */
#define TIME_TO_SLEEP 5 /* Time ESP32 will go to sleep (in seconds) */
```

Cet exemple utilise un facteur de conversion de microsecondes en secondes, afin que vous puissiez définir le temps de veille dans la variable **TIME_TO_SLEEP** en secondes. Dans ce cas, l'exemple mettra l'ESP32 en mode veille prolongée pendant 5 secondes.

8.3.1.3 Enregistrer les données dans les mémoires RTC

Avec l'ESP32, vous pouvez enregistrer des données dans les mémoires **RTC**. L'ESP32 dispose de 8 Ko de **SRAM** sur la partie **RTC**, appelée mémoire rapide RTC. Les données enregistrées ici ne sont pas effacées pendant le sommeil profond. Cependant, elles seront effacées lorsque vous appuyez sur le bouton de réinitialisation (le bouton étiqueté **RST** sur la carte ESP32).
Pour enregistrer des données dans la mémoire RTC, il suffit d'ajouter **RTC_DATA_ATTR** avant une définition de variable. L'exemple enregistre la variable **bootCount** sur la mémoire RTC. Cette variable comptera le nombre de fois que l'ESP32 s'est réveillé du sommeil profond.

```
RTC_DATA_ATTR int bootCount = 0;
```

```
Boot number: 9
Wakeup caused by timer
Setup ESP32 to sleep for every 5 Seconds
Configured all RTC Peripherals to be powered down in sleep
Going to sleep now
§§§§□D)!11□□§z§1§§□§§

Boot number: 10
Wakeup caused by timer
Setup ESP32 to sleep for every 5 Seconds
Configured all RTC Peripherals to be powered down in sleep
Going to sleep now
```

8.3.1.4 Raison du réveil

La fonction **`print_wakeup_reason()`** imprime la raison pour laquelle l'ESP32 a été réveillé du sommeil.

8.3.1.5 Le `setup()`

Dans le **`setup()`** est l'endroit où vous devez mettre la totalité de votre code. En sommeil profond, l'esquisse n'atteint jamais l'instruction **`loop()`**.
Donc, vous devez écrire tout le croquis dans setup ().
Ensuite, la variable **`bootCount`** est augmentée de un à chaque redémarrage, et ce numéro est imprimé sur le moniteur série.

```
++bootCount;
Serial.println("Boot number: " + String(bootCount));
```

Ensuite, le code appelle la fonction **`print_wakeup_reason()`**, mais vous pouvez appeler n'importe quelle fonction pour effectuer une tâche souhaitée. Par exemple, si vous souhaiterez réveiller votre ESP32 une fois par jour pour lire une valeur d'un capteur.

Ensuite, le code impose la **source de réveil** à l'aide de la fonction suivante :

```
esp_sleep_enable_timer_wakeup(time_in_us)
```

Cette fonction accepte comme argument le temps de sommeil en microsecondes comme nous l'avons vu précédemment.Dans notre cas, nous avons les éléments suivants :

```
esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR);
```

Ensuite, une fois toutes les tâches effectuées, l'ESP32 se met en veille en appelant la fonction suivante:

```
esp_deep_sleep_start()
```

8.3.1.6 La boucle - `loop()`

La section l**`oop()`** est vide, car l'ESP32 va s'endormir avant d'atteindre cette partie du code.

8.3.2 Le réveil par broches tactiles

Vous pouvez réveiller l'ESP32 du sommeil profond à l'aide des broches tactiles.

8.3.2.1 Activer le réveil tactile

Pour activer l'ESP32 pour qu'il puisse se réveiller à l'aide d'une broche tactile est simple. Vous devez utiliser la fonction suivante :

```
esp_sleep_enable_touchpad_wakeup()
```

Exemple de code

```
#include "ESP32_WakeUp.h"  // listing in Annex section
#define Threshold 40 /* Greater the value, more the sensitivity */
RTC_DATA_ATTR int bootCount = 0;
touch_pad_t touchPin;

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("External  RTC_IO");break;
    case ESP_SLEEP_WAKEUP_EXT1 : Serial.println("External  RTC_CNTL"); break;
    case ESP_SLEEP_WAKEUP_TIMER : Serial.println("Timer"); break;
    case ESP_SLEEP_WAKEUP_TOUCHPAD : Serial.println("Touchpad"); break;
    case ESP_SLEEP_WAKEUP_ULP : Serial.println("ULP program"); break;
    default : Serial.printf("Not caused by deep sleep: %d\n",wakeup_reason);
        break;
  }
}
```

```
// present in ESP32_WakeUp.h file
void print_wakeup_touchpad(){
  touchPin = esp_sleep_get_touchpad_wakeup_status();
  switch(touchPin)
  {
    case 0  : Serial.println("Touch detected on GPIO 4"); break;
    case 1  : Serial.println("Touch detected on GPIO 0"); break;
    case 2  : Serial.println("Touch detected on GPIO 2"); break;
    case 3  : Serial.println("Touch detected on GPIO 15"); break;
    case 4  : Serial.println("Touch detected on GPIO 13"); break;
    case 5  : Serial.println("Touch detected on GPIO 12"); break;
    case 6  : Serial.println("Touch detected on GPIO 14"); break;
    case 7  : Serial.println("Touch detected on GPIO 27"); break;
    case 8  : Serial.println("Touch detected on GPIO 33"); break;
    case 9  : Serial.println("Touch detected on GPIO 32"); break;
    default : Serial.println("Wakeup not by touchpad"); break;
  }
}

void callback(){ //placeholder callback function
}

void setup(){
  Serial.begin(9600);
  delay(1000); //Take some time to open up the Serial Monitor
  //Increment boot number and print it every reboot
  ++bootCount;
  Serial.println("Boot number: " + String(bootCount));
  //Print the wakeup reason for ESP32 and touchpad too
  print_wakeup_reason();
  print_wakeup_touchpad();
  //Setup interrupt on Touch Pad 3 (GPIO15) - the tested pad
  touchAttachInterrupt(T3, callback, Threshold);
  //Configure Touchpad as wakeup source
  esp_sleep_enable_touchpad_wakeup();
  //Go to sleep now
  Serial.println("Going to sleep now");
  esp_deep_sleep_start();
  Serial.println("This will never be printed");
}

void loop(){
  //This will never be reached
}
```

```
Boot number: 1
Wakeup was not caused by deep sleep: 0
Wakeup not by touchpad
Going to sleep now
99¶J□L¶1!11□□¶jL¶9¶!¶¶

Boot number: 2
Wakeup caused by touchpad
Touch detected on GPIO 15
Going to sleep now
□□□¶□□¶1¶))))¶¶1□¶9¶¶¶

Boot number: 3
Wakeup caused by touchpad
Touch detected on GPIO 15
Going to sleep now
¶¶¶¶□D)!1q¶□¶1□¶9¶!¶¶
```

8.3.2.2 Définition du seuil

La première chose à faire est de définir une valeur de **seuil** pour les broches tactiles. Dans ce cas, nous définissons le **seuil sur 40**. Vous devrez peut-être modifier la valeur du seuil en fonction de votre projet.

```
#define Threshold 40
```

Lorsque vous touchez un GPIO tactile, la valeur lue par le capteur diminue. Ainsi, vous pouvez définir une valeur de seuil qui provoque un événement lorsque le toucher est détecté.
La valeur de seuil définie ici signifie que lorsque la valeur lue par le GPIO tactile est inférieure à 40, l'ESP32 doit se réveiller. Vous pouvez ajuster cette valeur en fonction de la sensibilité souhaitée.

8.3.2.3 Association des interruptions

Vous devez attacher des interruptions aux broches sensibles au toucher. Lorsque le toucher est détecté sur un GPIO spécifié, une fonction de rappel est exécutée. Comme exemple, regardez la ligne suivante:

```
// Setup interrupt on Touch Pad 3 (GPIO15)
touchAttachInterrupt(T3, callback, Threshold);
```

Lorsque la valeur lue sur **`GPIO 15`** est inférieure à la valeur définie sur la variable **`Threshold`**, l'ESP32 se réveille et la fonction de rappel est exécutée.

La fonction **`callback()`** ne sera exécutée que si l'ESP32 est réveillé.

- Si l'ESP32 est en veille et que vous touchez **`T3 – GPIO 15`**, l'ESP se réveillera - la fonction **`callback()`** ne sera pas exécutée si vous **appuyez et relâchez** simplement la broche tactile;
- Si l'ESP32 est réveillé et que vous touchez **`T3 – GPIO 15`**, la fonction de rappel sera exécutée. Donc, si vous souhaitez exécuter la fonction **`callback()`** lorsque vous réveillez l'ESP32, vous devez maintenir le contact sur cette broche **pendant un moment**, jusqu'à ce que la fonction soit exécutée.

Dans ce cas, la fonction **`callback()`** est vide. I

```
void callback(){
   //placeholder callback function
}
```

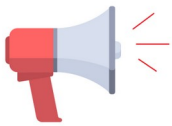
```
Boot number: 5
Wakeup caused by touchpad
Touch detected on GPIO 15
Going to sleep now
ŸŸŸŸ□ŸŸJ!!!Ÿ□ŸJ~ 191ŸŸ

Boot number: 6
Wakeup caused by touchpad
Touch detected on GPIO 15
Going to sleep now
Hello from callback function
Hello from callback function
Hello from callback function
Hello from callback function
Hello from callback function
Hello from callback function
Hello from callback function
Hello from callback function
Hello from callback function
```

Si vous souhaitez réveiller l'ESP32 en utilisant différentes broches tactiles, il vous suffit d'attacher des interruptions à ces broches.
Ensuite, vous devez utiliser la fonction **`esp_sleep_enable_touchpad_wakeup()`** pour définir les broches tactiles comme source de réveil.

```
//Configure Touchpad as wakeup source
esp_sleep_enable_touchpad_wakeup()
```

8.3.3 Réveil externe

Outre la minuterie et les broches tactiles, nous pouvons également réveiller l'ESP32 du sommeil profond en basculant la valeur d'un signal sur une broche, par la pression d'un bouton. C'est ce qu'on appelle un **réveil externe**. Vous avez deux possibilités (broches) de réveil externe : **`ext0`** et **`ext1`**.



8.3.3.1 Réveil externe (`ext0`)

Cette source de réveil vous permet d'utiliser **une broche** pour réveiller l'ESP32. L'option source de réveil **`ext0`** utilise les **GPIO RTC** pour se réveiller. Ainsi, les **périphériques RTC resteront allumés** pendant le sommeil profond si cette source de réveil est demandée.
Pour utiliser cette source de réveil, vous utilisez la fonction suivante:

```
esp_sleep_enable_ext0_wakeup(GPIO_NUM_X, level)
```

Cette fonction accepte comme premier argument la broche que vous souhaitez utiliser, dans ce format **`GPIO_NUM_X`**, dans lequel **`X`** représente le numéro **GPIO** de cette broche.
Le deuxième argument, **`level`**, peut être **`1`** ou **`0`**. Cela représente l'état du GPIO qui déclenchera le réveil.

8.3.3.2 Réveil externe (`ext1`)

Cette source de réveil vous permet d'utiliser **plusieurs GPIO RTC**. Vous pouvez utiliser deux fonctions logiques différentes :

- Réveillez l'ESP32 si l'une des broches que vous avez sélectionnées est haute (**`1 – HIGH`**)
- Réveillez l'ESP32 si toutes les broches que vous avez sélectionnées sont basses (**`0 – LOW`**)

Cette source de réveil est implémentée par le contrôleur RTC. Ainsi, les **périphériques RTC** et les **mémoires** RTC peuvent être mis **hors tension** dans ce mode.

Pour utiliser cette source de réveil, vous utilisez la fonction suivante :

```
esp_sleep_enable_ext1_wakeup(bitmask, mode)
```

Cette fonction accepte deux arguments:

- Un masque de bits (**`bitmask`**) des numéros GPIO qui provoqueront le réveil
- **Mode**: la logique de réveil de l'ESP32. Ça peut être:
  - **`ESP_EXT1_WAKEUP_ALL_LOW`** : se réveiller lorsque tous les GPIO deviennent bas
  - **`ESP_EXT1_WAKEUP_ANY_HIGH`** : réveillez-vous si l'un des GPIO devient haut

Exemple complet :

```
#include "ESP32_WakeUp.h"  // listing in Annex section
#define BUTTON_PIN_BITMASK 0x200000000 // 2^33 in hex
RTC_DATA_ATTR int bootCount = 0;

void setup(){
  Serial.begin(9600);
  pinMode(33,INPUT_PULLUP);  // dafault state of the pin - HIGH
  delay(1000); //Take some time to open up the Serial Monitor
  //Increment boot number and print it every reboot
  ++bootCount;
  Serial.println(); Serial.println();
  Serial.println("Boot number: " + String(bootCount));
  print_wakeup_reason();
  esp_sleep_enable_ext0_wakeup(GPIO_NUM_33,0); //1 = High, 0 = Low
  // use the switch on the board with pin 33 to set signal to LOW
  //If you were to use ext1, you would use it like
  //esp_sleep_enable_ext1_wakeup(BUTTON_PIN_BITMASK,ESP_EXT1_WAKEUP_ANY_LOW);
  Serial.println("Going to sleep now");
  delay(1000);
  esp_deep_sleep_start();
  Serial.println("This will never be printed");
}

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

Cet exemple réveille l'ESP32 lorsque vous déclenchez **`GPIO 33`** sur **`LOW`**. Le code montre comment utiliser les deux méthodes: **`ext0`** et **`ext1`**. Si vous importez le code tel quel, vous utiliserez **`ext0`**.

La fonction à utiliser **ext1** est commentée. Nous allons vous montrer comment fonctionnent les deux méthodes et comment les utiliser.

```
Serial.begin(9600);
delay(1000); //Take some time to open up the Serial Monitor
```

Vous incrémentez un à la variable **bootCount** et imprimez cette variable dans le moniteur série (**Serial**).

```
++bootCount;
Serial.println("Boot number: " + String(bootCount));
```

Ensuite, vous imprimez la raison du réveil en utilisant la fonction print_wakeup_reason () définie précédemment.

```
print_wakeup_reason();
```

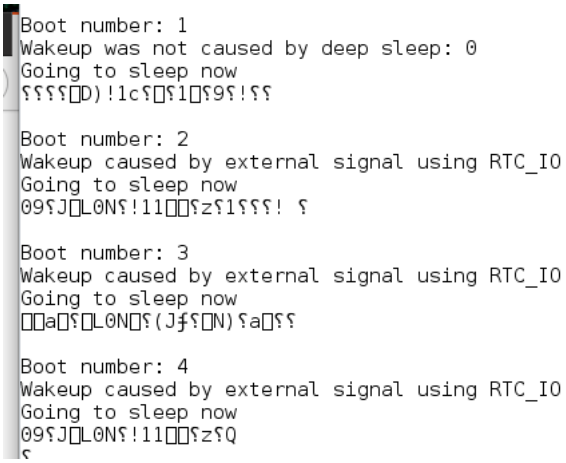
Après cela, vous devez activer les sources de réveil. Nous testerons séparément chacune des sources de réveil, **ext0** et **ext1**.

ext0

Dans cet exemple, l'ESP32 se réveille lorsque le **GPIO 33** est déclenché au niveau bas (**Low**) :

```
esp_sleep_enable_ext0_wakeup(GPIO_NUM_33,0); //1 = High, 0 = Low
```

Au lieu de **GPIO 33**, vous pouvez utiliser n'importe quelle autre broche **GPIO RTC**.



ext1

L'**ext1** vous permet de réveiller l'ESP à l'aide de différents boutons et d'effectuer différentes tâches en fonction du bouton sur lequel vous avez appuyé.
Au lieu d'utiliser la fonction **esp_sleep_enable_ext0_wakeup()**, vous utilisez la fonction fonction **esp_sleep_enable_ext1_wakeup()**.

Dans le code, cette fonction est commentée :

```
//If you were to use ext1, you would use it like
//esp_sleep_enable_ext1_wakeup(BUTTON_PIN_BITMASK,ESP_EXT1_WAKEUP_ANY_HIGH);
```

Dé-commentez cette fonction pour avoir :

```
esp_sleep_enable_ext1_wakeup(BUTTON_PIN_BITMASK,ESP_EXT1_WAKEUP_ANY_HIGH);
```

Le premier argument de la fonction est un masque de bits des GPIO que vous utiliserez comme source de réveil, et le deuxième argument définit la logique pour réveiller l'ESP32.

Dans cet exemple, nous utilisons la variable **`BUTTON_PIN_BITMASK`**, qui a été définie au début du code :

```
#define BUTTON_PIN_BITMASK 0x200000000 // 2^33 in hex
```

Ceci ne définit qu'**une seule broche** comme source de réveil, **`GPIO 33`**. Vous devez modifier le masque de bits pour configurer plus de broches GPIO comme source de réveil.

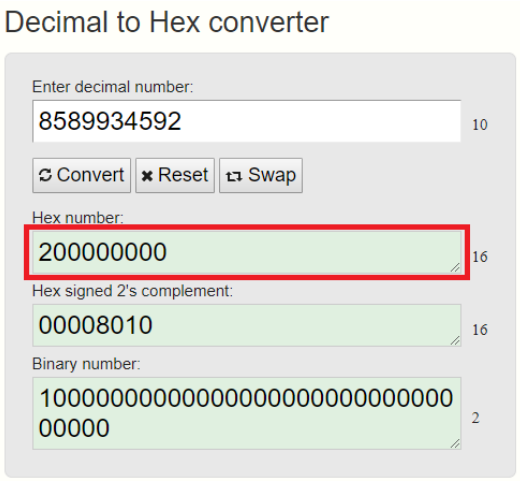
Création de GPIOs `bitmask`

Pour obtenir le masque de bits GPIO (**`bitmask`**), procédez comme suit :

1. Calculez **`2^(GPIO_NUMBER)`**. Enregistrez le résultat en décimal.
2. Allez sur **`http://rapidtables.com/convert/number/decimal-to-hex.html`** et convertissez le nombre décimal en hexadécimal;
3. Remplacez le nombre hexadécimal que vous avez obtenu dans la variable **`BUTTON_PIN_BITMASK`**.

Prenons un exemple. Dans le code de la librairie, le bouton est connecté à **`GPIO 33`**.
Pour obtenir le masque pour **`GPIO 33`** :

1. Calculez **`2^33`**. Vous devriez obtenir **8589934592,**
2. Convertissez ce nombre (**8589934592**) en **hexadécimal**. Vous pouvez accéder à ce convertisseur pour ce faire:



3. Copiez le numéro hexadécimal dans la variable **`BUTTON_PIN_BITMASK`** et vous devriez obtenir :

```
#define BUTTON_PIN_BITMASK 0x200000000 // 2^33 in hex
```

Masque pour plusieurs GPIO

Si vous souhaitez utiliser **`GPIO 34`** et **`GPIO 35`** comme source de réveil, vous devez procéder comme suit :

1. Calculez **`2^35 + 2^34`**. Vous devriez obtenir **51539607552**
2. Convertissez ce nombre en hexadécimal. Vous devriez obtenir : **`C00000000`**
3. Remplacez ce numéro dans le **`BUTTON_PIN_BITMASK`** comme suit :

```
#define BUTTON_PIN_BITMASK 0xC00000000
```

Identifier le GPIO utilisé comme source de réveil

Lorsque vous utilisez plusieurs broches pour réveiller l'ESP32, il est utile de savoir quelle broche a provoqué le réveil. Pour cela, vous pouvez utiliser la fonction suivante :

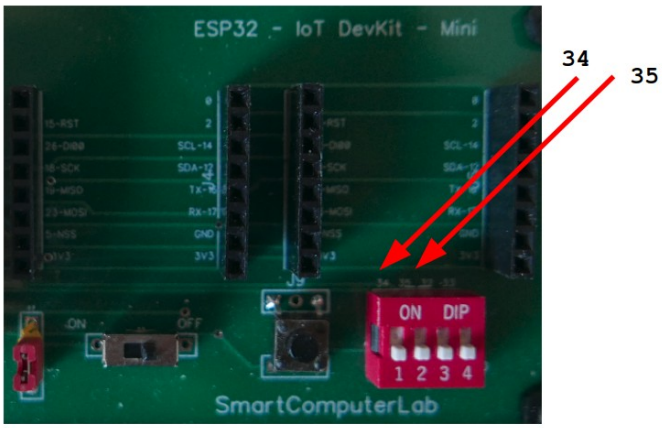
```
esp_sleep_get_ext1_wakeup_status()
```

Cette fonction renvoie un nombre de base 2, avec le nombre GPIO comme exposant: **`2^(GPIO_NUMBER)`**.
Donc, pour obtenir le GPIO en décimal, vous devez effectuer le calcul suivant :

```
GPIO = log(GPIO_NUMBER)/log(2)
```

8.3.3.2 Réveil externe - plusieurs GPIO

Maintenant, vous devriez pouvoir réveiller l'ESP32 à l'aide de différents boutons et identifier le bouton qui a provoqué le réveil. Dans cet exemple, nous utiliserons **`GPIO 34`** et **`GPIO 35`** comme source de réveil.



Le code

Vous devez apporter quelques modifications à l'exemple de code que nous avons utilisé auparavant :

- créer un masque de bits pour utiliser **`GPIO 34`** et **`GPIO 35`**. Nous vous avons montré comment faire cela auparavant;
- activer **`ext1`** comme source de réveil;
- utiliser la fonction **`esp_sleep_get_ext1_wakeup_status()`** pour obtenir le **GPIO** qui a déclenché le réveil.

L'esquisse suivante a tous ces changements mis en œuvre.

```
#include "ESP32_WakeUp.h"  // listing in Annex section
#define BUTTON_PIN_BITMASK 0xC00000000 // GPIOs 34 and 35
RTC_DATA_ATTR int bootCount = 0;

void print_GPIO_wake_up(){
long long GPIO_reason = esp_sleep_get_ext1_wakeup_status();
  Serial.print("GPIO that triggered the wake up: GPIO ");
  Serial.println((log(GPIO_reason))/log(2), 0);
}

void setup(){
  Serial.begin(9600);
  pinMode(34,INPUT_PULLUP); pinMode(35,INPUT_PULLUP);
  delay(1000); //Take some time to open up the Serial Monitor
  //Increment boot number and print it every reboot
  ++bootCount;
  Serial.println("Boot number: " + String(bootCount));
  //Print the wakeup reason for ESP32
  print_wakeup_reason();
  //Print the GPIO used to wake up
  print_GPIO_wake_up();
  //esp_deep_sleep_enable_ext0_wakeup(GPIO_NUM_33,1); //1 = High, 0 = Low
  //If you were to use ext1, you would use it like
  esp_sleep_enable_ext1_wakeup(BUTTON_PIN_BITMASK,ESP_EXT1_WAKEUP_ANY_HIGH);
  //Go to sleep now
  Serial.println("Going to sleep now");
  delay(1000);
  esp_deep_sleep_start();
  Serial.println("This will never be printed");
}

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

8.4 Sommeil profond et modem LoRa

Dans le laboratoire 7 précédent, nous avons étudié et expérimenté avec plusieurs exemples de communication LoRa avec des nœuds terminaux et de passerelle. Afin de réduire la consommation d'énergie, le Terminal fonctionne souvent en mode basse consommation activé sur ESP32 via le mode veille profonde.

L'exemple suivant montre l'utilisation du mode de veille prolongée avec l'interruption par la minuterie dans un Terminal avec modem LoRa (SX1278) connecté via le bus SPI.

8.4.1 Un simple Terminal LoRa avec mode deep_sleep

À chaque cycle de fonctionnement avec l'envoi d'un paquet Lora, le modem est:

- connecté au bus SPI
- activé
- il envoie un paquet - ~ 40 mA (la consommation électrique dépend de la durée de la transmission)
- désactivé (veille) - 0,2 mA et
- déconnecté du bus SPI - ~ 3,3 µA

8.4.1.1 Code complet avec un simple cycle d'envoi de paquet LoRa et un sommeil profond

```
#include <LoRa.h>
#include <Wire.h>
#include "LoRa_Para.h"
#include "ESP32_WakeUp.h"
#define uS_TO_S_FACTOR 1000000  /* micro seconds to seconds */
#define TIME_TO_SLEEP  10       /* Time ESP32 will go to sleep (in seconds) */
RTC_DATA_ATTR int bootCount = 0;

char dbuff[24];

void lora_send()
{
set_LoRa();
  LoRa.beginPacket();
  LoRa.print("hello ");
  LoRa.print(bootCount);
  LoRa.endPacket();
}

void setup(){
  Serial.begin(9600);
  delay(1000); //Take some time to open up the Serial Monitor
  ++bootCount;
  Serial.println();Serial.println();
  Serial.println("Boot number: " + String(bootCount));
  print_wakeup_reason();
  esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR);
  Serial.println("Setup to sleep for every "+String(TIME_TO_SLEEP)+" Seconds");
  esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH, ESP_PD_OPTION_OFF);
  Serial.println("Configured all RTC Peripherals to be powered down in sleep");
  delay(1000);
  lora_send();
  Serial.println("Ending LoRa");
  LoRa.end();    // disconnects SPI
  Serial.println("Going to sleep now");
  delay(300);   // wait to end the SPI bus transaction
  esp_deep_sleep_start();
  Serial.println("This will never be printed");
}


void loop(){}
```

Vous trouverez ci-dessous un fragment de la sortie affichée:

```
..
Boot number: 51
Wakeup caused by timer
Setup to sleep for every 10 Seconds
Configured all RTC Peripherals to be powered down in sleep
```

```
Starting LoRa ok!
Packet sent
Ending LoRa
Going to sleep now
Boot number: 52
Wakeup caused by timer
Setup to sleep for every 10 Seconds
Configured all RTC Peripherals to be powered down in sleep
Starting LoRa ok!
Packet sent
Ending LoRa
Going to sleep now
```

8.4.2 Terminal LoRa simple et liaison de données en mode `deep_sleep`

L'exemple suivant est l'extension de l'exemple étudié dans le laboratoire précédent. Il s'agit d'un nœud **Terminal** qui envoie les données du capteur température-humidité.
Ici, nous appliquons le mode **`deep_sleep`** avec la valeur du délai de réveil. La valeur de temporisation peut être intégrée dans les paramètres du terminal ou peut être fournie par le nœud **Gateway** (passerelle) via le paquet de retour.

8.4.2.1 Le code du Terminal avec le mode deep_sleep

```
#define TERMINAL
#define SHT21_SLEEP
#include <esp_wifi.h>
#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include "SHT21.h"
SHT21 SHT21;

typedef union
{
  uint8_t frame[32];    // data frame to send/receive data
  struct
  {
    uint32_t did;        // destination identifier chipID (4 lower bytes)
    uint32_t sid;        // source identifier chipID (4 lower bytes)
    float   sens[4];     // max 4 values - fields
    uint32_t  tout;      // optional timeout
    uint8_t pad[4];      // padding
  } pack;                // data packet
} DT_t;                  // DaTa type

RTC_DATA_ATTR uint32_t gwID=0, termID;  // gateway identifier stored in RTC memory
RTC_DATA_ATTR uint32_t cycle=8000, rcv_cycle, rcv_ack=0;    // main cycle in millis

void LoRa_send(uint8_t *frame) {
  LoRa_txMode();                        // set tx mode - normal mode
  LoRa.beginPacket();                   // start packet
  LoRa.write(frame,32);                  // add payload
  LoRa.endPacket(true);                 // finish packet and send it
}

void onReceive(int packetSize) {
  DT_t p;int i=0;
  if(packetSize==32)
    {
    while (LoRa.available())  { p.frame[i]= LoRa.read(); i++; }
    Serial.print("Terminal Received Packet:");
    Serial.println(p.pack.tout);
    rcv_cycle=p.pack.tout; rcv_ack=1;
    }
}

float stab[8];
void sensor_Fun()
{
  Wire.begin(12,14);
  SHT21.begin(); delay(200);
  stab[0]=(float) SHT21.getTemperature();
  stab[1]=(float) SHT21.getHumidity();
}
```

```
void setup() {
  Serial.begin(9600);
  termID=(uint32_t)ESP.getEfuseMac();
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  delay(333);
}

void loop() {
  if (runEvery(cycle))
  {
    DT_t p;
    sensor_Fun();
    p.pack.did=(uint32_t)0;
    p.pack.sid=(uint32_t)termID;
    p.pack.sens[0]=stab[0]; p.pack.sens[1]=stab[1];
    p.pack.sens[2]=0; p.pack.sens[3]=0;
    p.pack.tout=(uint32_t)millis();
    LoRa_send(p.frame); // send a packet
    Serial.println("Send Packet!");
    //while(rcv_ack==0)
    delay(2000);
    if(rcv_ack)
      {
      Serial.printf("Go to deep sleep for:%d, sec=%d\n",rcv_cycle,millis()/1000); rcv_ack=0;
      esp_sleep_enable_timer_wakeup(1000*1000*rcv_cycle);  // timeout in seconds
      esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH, ESP_PD_OPTION_OFF); //
      esp_sleep_pd_config(ESP_PD_DOMAIN_MAX, ESP_PD_OPTION_OFF);
      delay(60); LoRa.end();
      delay(300); // necessary to stop LoRa modem
      esp_deep_sleep_start();
      Serial.println("This will never be printed");
      }
  }
}
```

Figure 8.2 Les mesures de courant pour le sommeil profond et le mode actif (en mA) pour l'exécution du code ci-dessus

8.4.2.2 Code de la Gateway

```
#define GATEWAY
#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"

typedef union
{
  uint8_t frame[32];    // data frame to send/receive data
  struct
```

```
  {
    uint32_t did;        // destination identifier chipID (4 lower bytes)
    uint32_t sid;        // source identifier chipID (4 lower bytes)
    float   sens[4];     // max 4 values - fields
    uint32_t  tout;      // optional timeout
    uint8_t pad[4];      // padding
  } pack;                // data packet
} DT_t;                  // DaTa type

RTC_DATA_ATTR uint32_t gwID=0, termID;  // gateway identifier stored in RTC memory
RTC_DATA_ATTR uint32_t rcv_pack=0;

void LoRa_send(uint8_t *frame) {
  LoRa_txMode();                        // set tx mode - normal mode
  LoRa.beginPacket();                   // start packet
  LoRa.write(frame,32);                  // add payload
  LoRa.endPacket(true);                 // finish packet and send it
}

void onReceive(int packetSize) {
  DT_t p;int i=0;
  if(packetSize==32)
    {
    while (LoRa.available())  {  p.frame[i]= LoRa.read(); i++; }
    Serial.println("Gateway Received Packet ");
    Serial.printf("T:%2.2f, H:%2.2f\n",p.pack.sens[0],p.pack.sens[1]);
    rcv_pack=1;
    }
}

void setup() {
  Serial.begin(9600);
  gwID=(uint32_t)ESP.getEfuseMac();
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
}

void loop() {
  if (runEvery(1000)) { // repeat every 5000 millis
    DT_t p;
    p.pack.did=(uint32_t)termID;
    p.pack.sid=(uint32_t)gwID;
    p.pack.sens[0]=23.67; p.pack.sens[1]=67.67;
    p.pack.sens[2]=0; p.pack.sens[3]=0;
    p.pack.tout=(uint32_t)(10 + random(10,40));
    if(rcv_pack)
      {
      LoRa_send(p.frame); rcv_pack=0;
      Serial.printf("Send Packet with timeout=%d, sec=%d\n",p.pack.tout,millis()/1000);
      }/
  }
}
```

Remarque

Sachant que l'état profond réinitialise toutes les données de la mémoire SRAM, nous pouvons stocker l'indicateur **gwID** et autres éléments dans une mémoire active intégrée à l'unité RTC.

```
RTC_DATA_ATTR uint32_t gwID=0;  // gateway identifier stored in RTC memory
RTC_DATA_ATTR uint32_t cycle=8000, rcv_cycle, rcv_ack=0;
```

8.6 Résumé

Dans ce laboratoire, nous vous avons montré comment utiliser le sommeil profond avec l'ESP32 et les différentes façons de le réactiver.
Vous pouvez réactiver l'ESP32 à l'aide d'une minuterie, des broches tactiles ou d'un changement d'état **`GPIO`**.

Résumons ce que nous avons vu à propos de chaque source de réveil:

Minuterie de réveil (*Timer Wake Up*)
Pour activer le réveil du minuteur, vous utilisez la fonction :

```
esp_sleep_enable_timer_wakeup (time_in_us);
```

Utilisez la fonction **`esp_deep_sleep_start()`** pour démarrer le sommeil profond.

Réveil par broches tactiles (*Touch Wake Up*)
Pour utiliser les broches tactiles comme source de réveil, vous devez d'abord attacher des interruptions aux broches tactiles à l'aide de :

```
touchAttachInterrupt (Touchpin, callback, Threshold)
```

Ensuite, vous activez les broches tactiles comme source de réveil en utilisant :

```
esp_sleep_enable_touchpad_wakeup ()
```

Enfin, vous utilisez la fonction **`esp_deep_sleep_start()`** pour mettre l'ESP32 en mode sommeil profond.

Réveil externe (*External Wake Up*)

- Vous ne pouvez utiliser les GPIO RTC que comme réveil externe ;
- Vous pouvez utiliser deux méthodes différentes: **`ext0`** et **`ext1`** ;
  - **`ext0`** vous permet de réveiller l'ESP32 en utilisant une seule broche **`GPIO`**;
  - **`ext1`** vous permet de réveiller l'ESP32 à l'aide de plusieurs broches **`GPIO`**.

Modem externe (`LoRa`)
Pour minimiser la consommation électrique du modem LoRa, nous devons mettre le modem en mode veille et déconnecter le bus SPI.

8.6.1 A faire

1. Testez les exemples ci-dessus
2. Testez le nœud de terminal LoRa avec le nœud de passerelle LoRa avec un écran OLED
3. Utilisez un multimètre pour tester la consommation de courant en mode actif a et sommeil profond; calculer le courant moyen pendant 20 secondes en mode veille profonde et 2 secondes en mode actif. Quelle est l'autonomie de votre Terminal si vous utilisez une batterie LiPo 1500mAh

8.7 Annexe – fichier `ESP32_WakeUp.h`

```
void print_wakeup_reason()
void print_wakeup_touchpad(), and
void print_GPIO_wake_up()


// ESP32_WakeUp.h
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("External signal RTC_IO");
        break;
    case ESP_SLEEP_WAKEUP_EXT1 : Serial.println("External RTC_CNTL"); break;
    case ESP_SLEEP_WAKEUP_TIMER : Serial.println("Timer"); break;
    case ESP_SLEEP_WAKEUP_TOUCHPAD : Serial.println("Touchpad"); break;
    case ESP_SLEEP_WAKEUP_ULP : Serial.println("ULP program"); break;
    default : Serial.printf("Not caused by deep sleep: %d\n",wakeup_reason);
        break;
  }
}

void print_wakeup_touchpad(){
  touchPin = esp_sleep_get_touchpad_wakeup_status();
  switch(touchPin)
  {
    case 0  : Serial.println("Touch detected on GPIO 4"); break;
    case 1  : Serial.println("Touch detected on GPIO 0"); break;
    case 2  : Serial.println("Touch detected on GPIO 2"); break;
    case 3  : Serial.println("Touch detected on GPIO 15"); break;
    case 4  : Serial.println("Touch detected on GPIO 13"); break;
    case 5  : Serial.println("Touch detected on GPIO 12"); break;
    case 6  : Serial.println("Touch detected on GPIO 14"); break;
    case 7  : Serial.println("Touch detected on GPIO 27"); break;
    case 8  : Serial.println("Touch detected on GPIO 33"); break;
    case 9  : Serial.println("Touch detected on GPIO 32"); break;
    default : Serial.println("Wakeup not by touchpad"); break;
  }
}

void print_GPIO_wake_up(){
long long GPIO_reason = esp_sleep_get_ext1_wakeup_status();
  Serial.print("GPIO that triggered the wake up: GPIO ");
  Serial.println((log(GPIO_reason))/log(2), 0);
}
```

Lab 9 - Programmation multitâche avec FreeRTOS pour les Terminaux et Passerelles LoRa

Dans le laboratoire précédent (Lab.8), nous avons étudié la programmation basse consommation avec ESP32 et le développement des terminaux **LoRa** basse consommation. Dans ce laboratoire, nous présentons un certain nombre de mécanismes de programmation nécessaires pour construire des nœuds complexes fonctionnant avec **plusieurs tâches** telles que le traitement, la capture de données (détection), l'affichage des données et les tâches de communication.
ESP32 est fourni avec le système d'exploitation **FreeRTOS** qui permet le déploiement des applications multi-tâches.

9. Introduction à FreeRTOS

FreeRTOS est un **système d'exploitation en temps réel** (RTOS) open source, à faible encombrement et portable. Il fonctionne en **mode préemptif**. Il est aujourd'hui l'un des systèmes d'exploitation en temps réel les plus utilisés sur le marché.

Le **thread contrôlé** par **FreeRTOS** est une **tâche (*task*)**. Le nombre de tâches exécutées simultanément et leur priorité ne sont limités que par le matériel.
La planification est basée sur des **sémaphores**, des mécanismes **mutex** et un système de mise en **file d'attente** (*queue*).
FreeRTOS fonctionne selon le modèle Round-Robin avec gestion des priorités. Conçu pour être très compact, il ne contient que quelques fichiers de langage C et n'implémente aucun pilote matériel.

Les domaines d'application sont assez larges, car les principaux avantages de **FreeRTOS** sont l'exécution en temps réel, le code open source et la très petite taille. Il est donc principalement utilisé pour les systèmes embarqués qui ont des contraintes d'espace pour le code, mais aussi pour les systèmes de traitement vidéo et les applications réseau qui ont des contraintes "temps réel".
Parmi les applications les plus récentes fonctionnant avec des contraintes en temps réel figurent évidemment les **architectures IoT**.

9.1.1 Ordonnanceur de tâches

L'objectif principal du planificateur (ordonnanceur) de tâches est de décider quelles tâches sont à l'état prêt à être exécutées.
Pour faire ce choix, l'ordonnanceur **FreeRTOS** se base uniquement sur la **priorité des tâches**. Les tâches dans FreeRTOS se voient attribuer à leur création un niveau de priorité représenté par un entier.
Le niveau le plus bas est zéro et doit être strictement réservé à la tâche **`Idle`**.
Plusieurs tâches peuvent appartenir au même niveau de priorité. Dans FreeRTOS, il n'y a pas de mécanisme de gestion automatique des priorités (comme pour le système Linux). La priorité d'une tâche ne peut être modifiée qu'à la demande explicite du développeur.
Les tâches sont des fonctions simples qui s'exécutent généralement dans une boucle infinie (**`while(1)`**).

Dans le cas d'un microcontrôleur n'ayant qu'un seul cœur (ESP32 a deux cœurs), il n'y aura à tout moment qu'une seule tâche en cours d'exécution (***run***). Le planificateur s'assurera toujours que la tâche de priorité la plus élevée pouvant être exécutée soit sélectionnée pour entrer dans l'état d'exécution.
Si deux tâches partagent le même niveau de priorité et que les deux peuvent s'exécuter, les deux tâches s'**exécuteront en alternance** en ce qui concerne les réveils du planificateur (***ticks***).
Par défaut, dans ESP32, le planificateur fonctionne sur un noyau (noyau 0) et toutes les autres tâches sont exécutées sur le noyau 1.

Afin de choisir la tâche à exécuter, le planificateur doit lui-même exécuter et préempter la tâche en état d'exécution. Afin d'assurer le réveil de l'ordonnanceur, FreeRTOS définit une interruption périodique appelée "**tick interruption**".
Cette interruption s'exécute indéfiniment à une certaine fréquence qui est définie dans le fichier **`FreeRTOSConfig.h`** par la constante :

```
configTICK_RATE_HZ   // tick frequency interrupt" in Hz
```

FreeRTOS utilise la **planification préemptive** pour gérer les tâches. Cependant, il peut aussi éventuellement utiliser (si la directive lui est donnée) un **ordonnancement coopératif**. Dans ce mode d'ordonnancement, un changement de contexte d'exécution n'a lieu que si la tâche en cours d'exécution **autorise explicitement** l'exécution d'une autre tâche (en appelant un **`yield()`** par exemple) ou en entrant dans un état de blocage.

Les tâches ne sont donc jamais préemptées. Cette méthode de planification simplifie grandement la gestion des tâches, malheureusement elle peut conduire à un système moins efficace et moins sécurisé.

9.1.1.1 Famine (*starvation*)

Dans **FreeRTOS**, la tâche la plus prioritaire prête à être exécutée sera toujours sélectionnée par le planificateur. Cela peut conduire à une situation de **famine**. En effet, si la tâche de priorité supérieure n'est jamais interrompue, toutes les tâches de priorité inférieure ne seront jamais exécutées.

FreeRTOS n'implémente aucun mécanisme automatique pour empêcher le phénomène de famine.
Le développeur doit s'assurer qu'aucune tâche ne monopolise tout le temps d'exécution du microcontrôleur.

Il peut le faire en définissant des **événements** qui interrompront la tâche de priorité supérieure pendant une durée spécifiée ou jusqu'à ce qu'un autre événement se produise, laissant le champ libre pour les tâches de priorité inférieure à exécuter.

Pour éviter la famine, le développeur peut utiliser une **planification de taux monotone**. Il s'agit d'une technique d'attribution de priorité qui donne à chaque tâche une **priorité unique en fonction de sa fréquence d'exécution**.
La priorité la plus élevée est attribuée à la tâche avec la fréquence d'exécution la plus élevée et la priorité la plus faible est accordée à la tâche avec la fréquence la plus basse. L'ordonnancement à des cadences monotones permet d'optimiser l'ordonnancement des tâches, mais cela reste difficile à réaliser en raison de la nature des tâches qui ne sont pas complètement périodiques.

9.1.1.2 La tâche inactive (`Idle`)

Un microcontrôleur doit toujours avoir quelque chose à faire. En d'autres termes, il doit toujours y avoir une tâche en cours d'exécution. FreeRTOS gère cette situation en définissant la tâche **`Idle`** qui est créée au démarrage du planificateur (par exemple la tâche de la fonction **`loop()`**). La priorité du noyau la plus basse est attribuée à cette tâche.
Malgré cela, la tâche **`Idle`** peut avoir plusieurs fonctions à exécuter, notamment :

- libérer l'espace mémoire occupé par une tâche supprimée
- mettre le microcontrôleur en veille afin d'économiser l'énergie du système lorsque aucune tâche d'application n'est en cours d'exécution.
- mesurer le taux d'utilisation du processeur
-

9.1.2 Évolution et statut des tâches

Les tâches dans **FreeRTOS** peuvent exister dans 5 états: supprimées, suspendues, prêtes, bloquées ou en cours d'exécution (***deleted, suspended, ready, blocked , in execution***).

Dans **FreeRTOS**, il n'y a pas de variable pour spécifier explicitement l'état d'une tâche, en retour FreeRTOS utilise des **listes d'états** - **TCB**.
La présence de la tâche dans un t**ype de liste** de statuts détermine son statut (prêt, bloqué ou suspendu).
Lorsque les tâches changent d'état, le planificateur déplace la tâche (l'élément **`xListItem`** appartenant à cette même tâche) d'une liste d'états à une autre.

Lors de la création d'une tâche, **FreeRTOS** crée et remplit le **TCB** correspondant à la tâche, puis il insère directement la tâche dans une **liste prête** (***Ready List***) qui est la liste contenant une référence à toutes les tâches à l'état prêt.

FreeRTOS maintient plusieurs listes prêtes - **une liste pour chaque niveau de priorité**. Lors du choix de la prochaine tâche à exécuter, le planificateur analyse les listes prêtes de la priorité la plus élevée à la plus basse.

Plutôt que de définir explicitement un état en cours d'exécution ou une liste qui lui est associée, le noyau **FreeRTOS** décrit une variable **`pxCurrentTCB`** qui identifie le processus en cours d'exécution. Cette variable pointe vers le **TCB** correspondant au processus trouvé dans l'une des **Ready List**.

Une tâche peut se retrouver dans l'**état bloqué** lors de l'accès à une file d'attente de lecture/écriture si la file d'attente est vide/pleine.
Chaque opération d'accès à la file d'attente est configurée avec un délai d'expiration (**`xTicksToWait`**).
Si ce délai est égal à 0, la tâche n'est pas bloquée et l'opération d'accès à la file d'attente est considérée comme ayant échoué. Si le **`timeout`** n'est pas nul, la tâche passe à l'état bloqué jusqu'à ce qu'il y ait une modification de la file d'attente (par une autre tâche par exemple).

Une fois que l'opération d'accès à la file d'attente est possible, la tâche vérifie que son délai d'expiration n'a pas expiré et termine avec succès son opération.

Dans la figure 9.1, nous voyons le **diagramme d'état d'une tâche**. Une tâche peut être intentionnellement placée dans l'état suspendu, puis elle sera complètement ignorée par le planificateur et ne consommera plus de ressources jusqu'à ce qu'elle soit supprimée de cet état et retournée à un **état prêt**.

Le dernier état qu'une tâche peut prendre est l'**état supprim**é, cet état est nécessaire car une tâche supprimée ne libère pas ses ressources instantanément.
Une fois dans l'état supprimé, la tâche est ignorée par le planificateur et une autre tâche nommée **`Idle`** est chargée de libérer les ressources allouées par les tâches étant à l'état supprimé.

La tâche inactive (**`Idle)`** est créée au démarrage du planificateur et reçoit la priorité la plus faible possible, ce qui entraîne une libération retardée des ressources lorsque aucune autre tâche n'est en cours d'exécution.

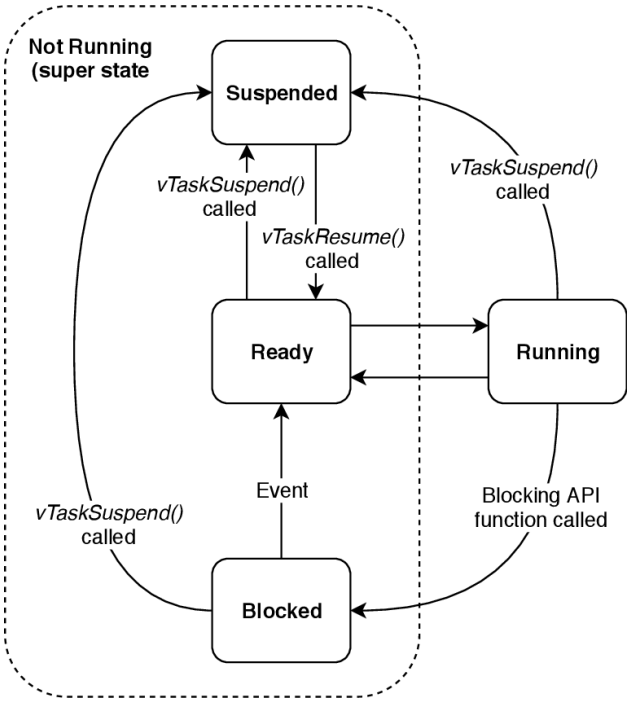


Figure 9.1 Diagramme d'état d'une tâche

9.2 FreeRTOS et programmation en temps réel pour l'IoT sur ESP32

Dans ce laboratoire, nous étudierons comment utiliser **FreeRTOS** fonctionnant sur le **ESP32**. Arduino ESP32 est construit sur FreeRTOS et en fait le programme principal est mis dans une tâche en boucle (`loop()`).

Mais si nous n'utilisons qu'une seule tâche (même en utilisant **Finite State Machine** pour une application Arduino pure), nous ne profiterons pas pleinement de **FreeRTOS** en **mode multitâche**.

FreeRTOS nous permet de déployer plusieurs tâches travaillant en «parallèle». Ces tâches sont gérées par le planificateur RTOS.

Sur l'ESP32, les tâches peuvent être déployées sur **2 cœur**s ce qui, cela dans le **contexte IoT**, donne la possibilité d'exécuter les tâches associées aux capteurs/actionneurs en parallèle avec les **tâches de communication sur les réseaux WiFi, LoRa, GSM, ..**.

Notre courte présentation de **FreeRTOS pour ESP32** est organisée en 4 parties:

1. création et suppression de tâches
2. mécanismes de communication entre les tâches (variables globales et files d'attente)
3. mécanismes de synchronisation (sémaphores, interruptions, ..)
4. déploiement de tâches sur une architecture dual-core

Après cette présentation, nous allons développer/expérimenter avec les nœuds LoRa: Terminaux et Passerelle exécutant plusieurs tâches.

9.2.1 Créer et supprimer une tâche

Nous créons une tâche avec un appel à la fonction **`xTaskCreate()`**.

Les arguments de cette fonction sont les suivants:

- **`TaskCode`**: dans cet argument, nous devons passer un **pointeur vers la fonction** qui implémentera la tâche.
- **`TaskName`**: le nom de la tâche, dans une chaîne. Par exemple "**`Sensor_Task`**".
- **`StackDepth`**: la taille de la pile de tâches, spécifiée comme le **nombre de variables** qu'elle peut contenir (pas le nombre d'octets). Dans cet exemple simple, nous passerons une valeur assez grande (**`1000`**).
- **`Parameter`**: pointeur vers un paramètre que la fonction de tâche peut recevoir. Il doit être de type (**`void *`**). Dans ce cas, pour simplifier le code, nous ne l'utiliserons pas, nous passons donc **`NULL`** dans l'argument.
- **`Priority`**: priorité de la tâche. Nous allons créer la tâche avec la priorité 1.
- **`TaskHandle`**: renvoie un descripteur qui peut être utilisé pour la dernière référence de la tâche lors des appels de fonction (par exemple, pour supprimer une tâche ou modifier sa priorité). Pour cet exemple simple, nous n'allons pas l'utiliser, donc ce sera **`NULL`**.

La fonction **`xTaskCreate()`** renvoie **`pdPASS`** en cas de succès ou un code d'erreur. Pour l'instant, nous supposerons que la tâche sera créée sans aucun problème, nous n'allons donc pas faire de vérification d'erreur. Bien sûr, pour une application plus réelle, nous aurions besoin de faire ce test pour confirmer que la tâche est bien créée.

Pour supprimer une tâche de son propre code, il vous suffit d'appeler la fonction **`vTaskDelete`**.

```
vTaskDelete (NULL);
```

9.2.2 Création et exécution d'une simple tâche supplémentaire

Dans le premier exemple, à côté de la tâche du fond (**`loop() Task`**), nous ajouterons une tâche supplémentaire à notre application.
Notre application a 2 tâches:
La tâche **`loop()`** imprimera le texte "**`this is main loop Task`**" et la deuxième tâche imprimera "c**`this is an extra task`**" sur le terminal série.

```
void setup() {
  Serial.begin(9600);
  /* we create a new task here */
  xTaskCreate(
      additionalTask,           /* Task function. */
      "additional Task",        /* name of task. */
      10000,                    /* Stack size of task */
      NULL,                     /* parameter of the task */
      1,                        /* priority of the task */
      NULL);                    /* Task handle to keep track of created task */
}
/* the forever loop() function is invoked by ESP32 loopTask */
void loop() {
  Serial.println("this is main loop Task");
  delay(1000);
}
/* this function will be invoked when additionalTask was created */
void additionalTask( void * parameter )
{
  /* loop forever */
  for(;;){
    Serial.println("this is additional Task");
    delay(1000);
  }
  /* delete a task when finish,
  this will never happen because this is infinity loop */
  vTaskDelete( NULL );
}
```

Le résultat affiché :

```
this is this is main loop Task
this is additional Task
this is this is main loop Task
this is additional Task
this is this is main loop Task
this is additional Task
```

9.2.3 Création et exécution de deux tâches

Dans cet exemple, nous allons créer deux tâches avec la même priorité d'exécution (1). La tâche principale (**`loop()`**) reste inactive.

```
void taskOne( void * parameter )
{
    for( int i = 0;i<10;i++ ){
        Serial.println("Hello from task 1");
        delay(1000);
    }
    Serial.println("Ending task 1");
    vTaskDelete( NULL );
}

void taskTwo( void * parameter)
{
    for( int i = 0;i<10;i++ ){
        Serial.println("Hello from task 2");
        delay(1000);
    }
    Serial.println("Ending task 2");
    vTaskDelete( NULL );
}
```

```
void setup() {
  Serial.begin(9600);
  delay(1000);
  xTaskCreate(
                    taskOne,          /* Task function. */
                    "TaskOne",        /* String with name of task. */
                    10000,            /* Stack size in words. */
                    NULL,             /* Parameter passed as input of the task */
                    1,                /* Priority of the task. */
                    NULL);            /* Task handle. */

xTaskCreate(
                    taskTwo,          /* Task function. */
                    "TaskTwo",        /* String with name of task. */
                    10000,            /* Stack size in words. */
                    NULL,             /* Parameter passed as input of the task */
                    1,                /* Priority of the task. */
                    NULL);            /* Task handle. */
}

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

Résultat de l'exécution :

```
Hello from task 1
Hello from task 2
Hello from task 1
Hello from task 2
Hello from task 2
Hello from task 1
Hello from task 1
Hello from task 2
```

9.3 Communication entre deux tâches

Il existe plusieurs façons de faire communiquer les tâches entre elles (**variables globales**, **files d'attente**, ..)

9.3.1 Variables globales comme arguments

Pour commencer, nous passerons un seul paramètre à partir d'une variable globale.
Les fonctions de tâche reçoivent un paramètre générique **`(void *)`**. Dans notre code, nous l'interpréterons comme un pointeur vers **`int`**, qui correspond à **`(int *)`**.
Donc, la première chose que nous faisons est de convertir (***cast***) en **`(int *)`**.

```
(int *) parameter;
```

Nous avons maintenant un pointeur vers la position mémoire d'un entier. Cependant, nous voulons accéder au contenu actuel de la position mémoire. Nous voulons donc la valeur de la position mémoire vers laquelle pointe notre pointeur. Pour ce faire, nous utilisons l'opérateur de **déférence**, qui est **`*`**.
Donc, ce que nous devons faire est d'utiliser l'opérateur de déférence sur notre pointeur converti et nous devrions pouvoir **accéder à sa valeur**.

```
*((int *)parameter);
```

Une fois que nous y avons accès, nous pouvons simplement l'imprimer en utilisant la fonction **`Serial.println`**, ce que nous ferons dans les deux fonctions.
Le code source complet peut être vu ci-dessous, ainsi que l'implémentation des deux fonctions.
Notez qu'à des fins de débogage, nous imprimons différentes chaînes dans les fonctions.

```
int globalIntVar = 5;
int localIntVar = 9;

void setup()
{
  Serial.begin(9600);
  delay(1000);
  xTaskCreate(
            globalIntTask,             /* Task function. */
            "globalIntTask",           /* String with name of task. */
            10000,                     /* Stack size in words. */
            (void*)&globalIntVar,      /* Parameter passed as input of the task */
            1,                         /* Priority of the task. */
            NULL);                     /* Task handle. */
  xTaskCreate(
            localIntTask,              /* Task function. */
            "localIntTask",            /* String with name of task. */
            10000,                     /* Stack size in words. */
            (void*)&localIntVar,       /* Parameter passed as input of the task */
            1,                         /* Priority of the task. */
            NULL);                     /* Task handle. */
}

void globalIntTask(void *parameter )
{
    Serial.print("globalIntTask: ");
    Serial.println(*((int*)parameter));
    vTaskDelete( NULL );
}

void localIntTask(void *parameter )
{
    Serial.print("localIntTask: ");
    Serial.println(*((int*)parameter));
    vTaskDelete( NULL );
}

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

Résultat de l'exécution :

```
lobalIntTask: 5
localIntTask: 9
```

9.3.2 Files d'attente (*queues*)

Les files d'attente sont très utiles pour la communication inter-tâches, permettant d'envoyer des messages d'une tâche à une autre. Elles sont généralement organisées en **FIFO** (***First In First Out***), ce qui signifie que de nouvelles données sont insérées à l'arrière de la file d'attente et consommées à l'avant.

Les données placées dans la file d'attente sont copiées plutôt que référencées. Cela signifie que si nous envoyons un entier à la file d'attente, sa valeur sera effectivement copiée et si nous modifions la valeur d'origine par la suite, aucun problème ne devrait se produire.
Un aspect comportemental important est que l'insertion dans une file d'attente complète ou la consommation à partir d'une file d'attente vide peut bloquer les appels pendant une durée donnée (cette **durée est un paramètre** de l'API).

Bien que les files d'attente mentionnées soient généralement utilisées pour la communication entre les tâches, pour cet exemple d'introduction, nous allons insérer et consommer des éléments dans la file d'attente intégrée à la fonction **`loop()`**.

```
QueueHandle_t queue;

void setup() {
  Serial.begin(9600);
  queue = xQueueCreate( 10, sizeof( int ) );
  if(queue == NULL){ Serial.println("Error creating the queue"); }
}
 void loop() {
  if(queue == NULL)return;
  for(int i = 0; i<10; i++){ xQueueSend(queue, &i, portMAX_DELAY); }

  int element;
  for(int i = 0; i<10; i++){
    xQueueReceive(queue, &element, portMAX_DELAY);
    Serial.print(element);
    Serial.print("|");
  }
  Serial.println();
  delay(1000);
}
```

Ci-dessous, le même type de programme est précédé de la création de deux tâches **`ProducerTask`** et **`ConsumerTask`**.

```
QueueHandle_t queue;
int queueSize = 10;

void producerTask( void * parameter )
{
    for( int i = 0;i<queueSize;i++ ){
      xQueueSend(queue, &i, portMAX_DELAY);
    }
    vTaskDelete( NULL );
}

void consumerTask( void * parameter)
{
    int element;
    for( int i = 0;i < queueSize;i++ ){
        xQueueReceive(queue, &element, portMAX_DELAY);
        Serial.print(element);
        Serial.print("|");
    }
    vTaskDelete( NULL );
}

void setup() {
  Serial.begin(9600);
  delay(2000);

  queue = xQueueCreate( queueSize, sizeof( int ) );
  if(queue == NULL){
    Serial.println("Error creating the queue");
  }
```

```
  xTaskCreate(
                    producerTask,     /* Task function. */
                    "Producer",       /* String with name of task. */
                    10000,            /* Stack size in words. */
                    NULL,           /* Parameter passed as input of the task */
                    1,                /* Priority of the task. */
                    NULL);            /* Task handle. */

  xTaskCreate(
                    consumerTask,     /* Task function. */
                    "Consumer",       /* String with name of task. */
                    10000,            /* Stack size in words. */
                    NULL,           /* Parameter passed as input of the task */
                    1,                /* Priority of the task. */
                    NULL);            /* Task handle. */
}

void loop() {
  Serial.println("in the loop");
  delay(6000);
}
```

9.3.3 A faire

Expérimentez avec l'exemple ci-dessus avec différentes tailles de file d'attente (**128, 1000, ..**) et avec différents types de données (**`char`**, **`float`**, **`double`**, ..) et même des **structures**.

9.4 Une application IoT avec deux tâches et la communication par files d'attente

Dans l'exemple suivant, nous utiliserons un capteur de température/humidité **SHT21**.

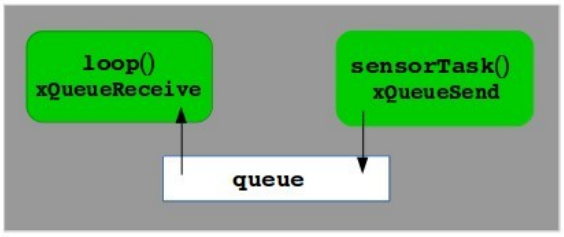


Figure 9.2 `loop()`, `sensorTask` et une file d'attente - `queue`

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

QueueHandle_t queue;
int queueSize = 128;

void sensorTask( void * pvParameters ){
float t,h;
while(true)
  {
  Wire.begin(12,14);
  SHT21.begin(); delay(200);
  t=(float) SHT21.getTemperature();
  h=(float) SHT21.getHumidity();
  xQueueSend(queue, &t, portMAX_DELAY);
  xQueueSend(queue, &h, portMAX_DELAY);
  delay(1000);
  }
}

void setup()
{
  Serial.begin(9600);
  delay(1000);
  queue = xQueueCreate( queueSize, sizeof( int ) );
  Serial.println("SHT21 test");

  xTaskCreate(
                    sensorTask,   /* Function to implement the task */
                    "sensorTask", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL     );  /* Task handle */

  Serial.println("Task created...");
}

void loop() {
   float t,h;
   Serial.println("Starting main loop...");
   while (true){
    xQueueReceive(queue, &t, portMAX_DELAY);
    xQueueReceive(queue, &h, portMAX_DELAY);
    Serial.print("temp:");Serial.println(t);
    Serial.print("humi:");Serial.println(h);
    delay(1000);
    }
}
```

Résultat de l'exécution :

```
Task created...
Starting main loop...
temp:24.11
humi:26.25
temp:24.10
humi:26.53
temp:24.13
humi:26.81
temp:24.13
humi:27.23
```

9.4.1 A faire

1. Testez le même programme avec un capteur différent **`BH1750`** par exemple.
2. Remplacer les variables du capteur par une structure :
```
struct
{
float t;
float h;
} sens;
```
3. Ajouter une deuxième tâche et une deuxième fille pour le deuxième capteur

9.5 Sémaphores

Les sémaphores sont utilisés pour créer des schémas de synchronisation et de protection pour les sections partagées (critiques) du code.
Dans ce paragraphe, nous présenterons plusieurs types de sémaphores:

- **sémaphore binaire**
- **mutex**
- **sémaphore - compteur**

9.5.1 Sémaphore binaire

C'est le moyen le plus simple de contrôler l'accès aux ressources qui n'ont que 2 états: verrouillé/déverrouillé ou indisponible/disponible.
La tâche qui veut accéder à la ressource appellera **`xSemaphoreTake()`**.
Il y a 2 cas:

1. S'il réussit à accéder à la ressource, il la conservera jusqu'à ce qu'elle appelle **`xSemaphoreGive()`** pour libérer la ressource afin que d'autres tâches puissent la récupérer.
2. En cas d'échec, il attendra que la ressource soit libérée par une autre tâche.

Les sémaphores binaires peuvent être appliqués pour interrompre (terminer) le traitement (**Interrupt Sub-Routine ISR**) où la fonction type **ISR** appellera **`xSemaphoreGiveFromISR()`** pour déléguer le traitement d'interruption à la tâche qui attend sur **`xSemaphoreTake()`**.

Lorsque **`xSemaphoreTake()`** est appelé, la tâche sera bloquée et attendra un événement d'interruption.

Remarque :

Les fonctions API appelées depuis l'ISR doivent avoir le préfixe "**`FromISR`**" (**`xSemaphoreGiveFromISR`**).
Elles sont conçus pour les fonctions de l'API ***Interrupt Safe***.

9.5.1.1 Exemple de sémaphore binaire avec `xSemaphoreGiveFromISR`

Nous réutiliserons l'exemple de code suivant en utilisant le style **FreeRTOS** et le sémaphore binaire pour traiter l'interruption.

```
byte ledPin = 22;
byte interruptPin = 0;  // touch pin
/* hold the state of LED when toggling */
volatile byte state = LOW;

void setup() {
  pinMode(ledPin, OUTPUT);
  /* set the interrupt pin as input pullup*/
  pinMode(interruptPin, INPUT_PULLUP);
  /* attach interrupt to the pin
  function blink will be invoked when interrupt occurs
  interrupt occurs whenever the pin change value */
  attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
}

void loop() {
}

/* interrupt function toggle the LED */
void blink() {
  state = !state;
  digitalWrite(ledPin, state);
}
```

Le code avec une interruption de traitement de tâche à haute priorité et appelé-activé depuis la **fonction ISR** par:

```
xSemaphoreGiveFromISR(xBinarySemaphore,&xHigherPriorityTaskWoken );
```

9.5.1.2 Code complet

```
byte ledPin = 22;
byte interruptPin = 2;
volatile byte state = LOW;
SemaphoreHandle_t xBinarySemaphore;

void setup() {
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);
  /* set the interrupt pin as input pullup*/
  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin), ISRcallback, CHANGE);
  /* initialize binary semaphore */
  xBinarySemaphore = xSemaphoreCreateBinary();
  /* this task will process the interrupt event
  which is forwarded by interrupt callback function */
  xTaskCreate(
    ISRprocessing,           /* Task function. */
    "ISRprocessing",         /* name of task. */
    1000,                    /* Stack size of task */
    NULL,                    /* parameter of the task */
    4,                       /* priority of the task */
    NULL);
}

void loop() {}
/* interrupt function callback */
void ISRcallback() {
  BaseType_t xHigherPriorityTaskWoken = pdFALSE;
  /* un-block the interrupt processing task now */
  xSemaphoreGiveFromISR(xBinarySemaphore,&xHigherPriorityTaskWoken );
}

/* this function will be invoked when additionalTask was created */
void ISRprocessing( void * parameter )
{
  Serial.println((char *)parameter);
  /* loop forever */
  for(;;){
    /* task move to Block state to wait for interrupt event */
    xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );
    Serial.println("ISRprocessing is running");
    /* toggle the LED now */
    state = !state;
    digitalWrite(ledPin, state);
  }
  vTaskDelete( NULL );
}
```

9.5.2 Mutex

Un sémaphore de type **mutex** est comme **une clé** associée à la ressource. La tâche contient la clé, verrouille la ressource, la traite, puis déverrouille et renvoie la clé pour d'autres tâches à utiliser.
Ce mécanisme est similaire au sémaphore binaire sauf que la tâche qui prend la clé **doit libérer la clé (mutex)**.
Supposons que nous ayons 2 tâches: une tâche à faible priorité et une tâche à haute priorité.
Ces tâches attendent la clé et la tâche à faible priorité a une chance de capturer la clé, puis bloque la tâche à haute priorité et continue son exécution.

9.5.2.1 Exemple de mutex et deux tâches (de priorité basse et haute)

Dans cet exemple, nous créons 2 tâches: une tâche à faible priorité et une tâche à haute priorité. La tâche à faible priorité conservera la clé et bloquera la tâche à haute priorité.

```
SemaphoreHandle_t xMutex;

void setup() {
  Serial.begin(9600);
  /* create Mutex */
  xMutex = xSemaphoreCreateMutex();
  xTaskCreate(
      lowPriorityTask,           /* Task function. */
      "lowPriorityTask",        /* name of task. */
      1000,                    /* Stack size of task */
      NULL,                     /* parameter of the task */
      1,                        /* priority of the task */
      NULL);                    /* Task handle to keep track of created task */
  delay(500);
  /* let lowPriorityTask run first then create highPriorityTask */
  xTaskCreate(
      highPriorityTask,           /* Task function. */
      "highPriorityTask",        /* name of task. */
      1000,                    /* Stack size of task */
      NULL,                     /* parameter of the task */
      4,                        /* priority of the task */
      NULL);                    /* Task handle to keep track of created task */
}

void loop() {
}

void lowPriorityTask( void * parameter )
{
  Serial.println((char *)parameter);
  for(;;){
    Serial.println("lowPriorityTask gains key");
    xSemaphoreTake( xMutex, portMAX_DELAY );
    /* even low priority task delay high priority
    still in Block state */
    delay(4000);
    Serial.println("lowPriorityTask releases key");
    xSemaphoreGive( xMutex );
  }
  vTaskDelete( NULL );
}
void highPriorityTask( void * parameter )
{
  Serial.println((char *)parameter);
  for(;;){
    Serial.println("highPriorityTask gains key");
    /* highPriorityTask wait until lowPriorityTask release key */
    xSemaphoreTake( xMutex, portMAX_DELAY );
    Serial.println("highPriorityTask is running");
    Serial.println("highPriorityTask releases key");
    xSemaphoreGive( xMutex );
    /* delay so that lowPriorityTask has chance to run */
    delay(2000);
  }
  vTaskDelete( NULL );
}
```

9.6 Tâches FreeRTOS sur un processeur multicœur (ESP32)

L'utilisation d'un processeur avec des tâches indépendantes pour les applications IoT permet de séparer les opérations de capture (détection) et de communication (WiFi, LoRa, ..). Pour aller plus loin nous pouvons implémenter ces fonctionnalités sur les tâches exécutées sur **plusieurs processeurs**.
Dans cette section, nous montrerons comment introduire la deuxième unité d'exécution et ensuite comment partager des tâches sur ces deux processeurs.

9.6. Créer une tâche épinglée sur un CPU

Tout d'abord, nous allons déclarer une variable globale qui contiendra le numéro du CPU où la tâche **FreeRTOS** que nous allons lancer sera épinglée. Cela garantit que nous pouvons facilement le modifier tout en testant le code.
Notez que nous allons exécuter le code deux fois, en attribuant les valeurs **`0`** et **`1`** à cette variable.

```
static int taskCore = 0;
```

Au début, nous afficherons le numéro du noyau que nous testons ; il est stocké dans la variable globale précédemment déclarée.
Ensuite, nous lancerons la tâche **FreeRTOS**, en l'attribuant à un processeur spécifique de l'ESP32. Nous utiliserons la fonction **`xTaskCreatePinnedToCore`**. Cette fonction prend exactement les mêmes arguments que **`xTaskCreate`** et un **argument supplémentaire** à la fin pour spécifier le processeur sur lequel la tâche doit s'exécuter.

Nous allons implémenter la tâche dans une fonction appelée **`coreTask`**. Nous devrions donner à la tâche la priorité **`0`**, pour qu'elle soit inférieure par rapport aux - **`setup()`** et main loop - **`loop()`**.
Nous n'avons pas besoin de nous soucier de passer des paramètres d'entrée ou de stocker le descripteur de tâche.
Dans **`loop()`**, nous commençons par imprimer un message sur le port série indiquant que nous lançons la boucle principale
Nous programmons une boucle **`while(1)`** infinie, sans code à l'intérieur. Il est essentiel que nous ne mettions aucun type de fonction d'exécution ou de retard à l'intérieur. La meilleure approche consiste à laisser ce champ vide.
La fonction de la tâche est également très simple. Nous allons simplement imprimer un message indiquant le CPU qui lui est attribué. Il est obtenu avec **`xPortGetCoreID`**. Bien entendu, cela doit correspondre au processeur spécifié dans la variable globale.

9.6.1.1 Code avec tâche épinglée

```
static int taskCore = 0;

void coreTask( void * pvParameters ){
    Serial.println("task running on core: ");
    while(true){
        Serial.print(xPortGetCoreID());
        delay(1000);
    }
}

void setup() {
  Serial.begin(9600);
  delay(1000);
  Serial.print("Starting to create task on core ");
  Serial.println(taskCore);

  xTaskCreatePinnedToCore(
                    coreTask,   /* Function to implement the task */
                    "coreTask", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore);  /* Core where the task should run */
  Serial.println("Task created...");
}
```

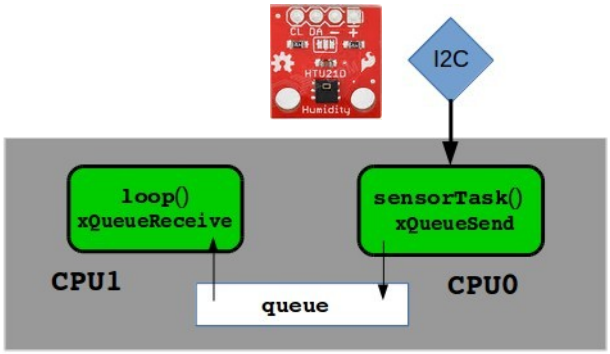
```
void loop() {
  Serial.println("Starting main loop...");
  while (true)
    {
      Serial.print(xPortGetCoreID());
      delay(3000);
     }  // with no delay - CPU is occupied 100%
}
```

Impression du programme :

```
Task created...
Starting main loop...
1task running on core:
00010001000100010001000100
```

9.6.2 Application IoT simple fonctionnant sur 2 cœurs

Dans l'exemple suivant, nous utiliserons 2 cœurs d'exécution, l'un pour lire les données du capteur et l'autre pour afficher ces données sur le terminal.
La communication entre les tâches se fait via une file d'attente.



The code:

```
#include <Wire.h>
#include "SHT21.h"
SHT21 SHT21;
QueueHandle_t queue;
int queueSize = 128;
static int taskCore = 0;

void sensorTask( void * pvParameters ){
float t,h;
while(true)
  {
  Wire.begin(12,14);
  SHT21.begin(); delay(200);
  Serial.printf("In sensor task:%d\n",xPortGetCoreID());
  t=(float) SHT21.getTemperature();
  h=(float) SHT21.getHumidity();
  xQueueSend(queue, &t, portMAX_DELAY);
  xQueueSend(queue, &h, portMAX_DELAY);
  delay(1000);
  }
}

void setup()
{
  Serial.begin(9600);
  delay(1000);
  queue = xQueueCreate( queueSize, sizeof( int ) );
  Serial.println("SHT21 test");
```

```
xTaskCreatePinnedToCore(
                    sensorTask,   /* Function to implement the task */
                    "sensorTask", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,  /* Task handle */
                    taskCore);  /* Core where the task should run */

  Serial.println("Task created...");
}

void loop() {
   float t,h;
   Serial.println("Starting main loop...");
   while (true){

    xQueueReceive(queue, &t, portMAX_DELAY);
    xQueueReceive(queue, &h, portMAX_DELAY);
    Serial.printf("In main task:%d\n",xPortGetCoreID());
    Serial.print("temp:");Serial.println(t);
    Serial.print("humi:");Serial.println(h);
    delay(1000);
        }
}
```

Résultat d'exécution :

```
SHT21 test
Task created...
Starting main loop...
In sensor task:0
In main task:1
temp:28.09
humi:19.76
In sensor task:0
In main task:1
temp:28.09
humi:19.73
In sensor task:0
In main task:1
temp:28.11
humi:19.69
..
```

9.6.3 A faire

1. Testez les exemples de code présentés
2. Ajoutez une **tâche d'affichage** pour afficher les données reçues dans la file d'attente.
   - Utilisez un écran OLED connecté via le bus I2C, puis
   - Écran IPS connecté via le bus SPI.

9.7 Création de nœuds de terminal et de passerelle LoRa avec plusieurs tâches

Dans cette section, nous continuons le développement des nœuds LoRa introduits dans Lab.7 , y compris les nœuds type Terminal et Gateway.
Commençons par compléter les opérations d'un terminal en ajoutant la **tâche de capteur** (**`sensor_Task`**) - pour capturer les données sur le capteur température/humidité (**SHT21**).

9.7.1 Nœud Terminal avec tâche de capteur

Rappelons le code du terminal qui peut envoyer un certain nombre de données de capteur dans le paquet de **`LoRa_send()`**. Le code ci-dessous n'utilise pas **deep_sleep** afin de permettre à la tâche de capteur de fonctionner à tout moment.
Notez que la tâche de capteur peut capturer les données plusieurs fois avant leur émission dans la trame LoRa. Dans ce cas, la valeur des données capturées peut être lissée et la valeur résultante peut être présentée comme une moyenne des lectures sur le capteur donné.
La communication entre le **`sensor_Task ()`** et la tâche principale dans la boucle **`loop()`** se fait via une file d'attente (**`queue`**).
Dans la fonction **`setup()`**, nous activons la file d'attente et nous commençons à créer la **`sensorTask`** qui exécute la fonction **`sensor_Task()`**.

```
QueueHandle_t queue;
int queueSize = 32;
static int taskCore = 0;

queue = xQueueCreate( queueSize, sizeof(float)*8);
xTaskCreatePinnedToCore(
                    sensor_Task,   /* Function to implement the task */
                    "sensor_Task", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore);  /* Core where the task should run */
Serial.println("Task created...");
```

La fonction **`sensor_Task()`** lit les données du capteur et les envoie dans la file d'attente. Le cycle d'exécution de la tâche capteur est quatre fois plus rapide (cycle * 1000/4) que le cycle de la tâche principale dans la **`loop()`**. Cela permet comme toujours de fournir les données. La file d'attente est réinitialisée à chaque fois qu'une nouvelle valeur va être envoyée. Cela empêche la file d'attente de déborder.

```
void sensor_Task( void * pvParameters )
{
float stab[4];
while(true)
  {
  Wire.begin(12,14);
  SHT21.begin(); delay(200);
  stab[0]=(float) SHT21.getTemperature();
  stab[1]=(float) SHT21.getHumidity();
  stab[2]=0.0; stab[3]=0.0;
  Serial.printf("T:%2.2f, H:%2.2f\n",stab[0],stab[1]);
  delay(cycle*1000/4);
  xQueueReset(queue);  // reset queue to store only the last values
  xQueueSend(queue, stab, portMAX_DELAY);
  }
}
}
```

Dans la boucle de tâche principale **`loop()`**, le programme attend les nouvelles données dans la file d'attente avec :

```
xQueueReceive(queue, stab, portMAX_DELAY);
```

9.7.2 Code complet du Terminal LoRa avec la tâche du capteur

Dans le code suivant nous utilisons les éléments de paramétrage fournis dans le fichier **`LoRa_Para.h`**.

```
#define TERMINAL
#define SHT21_SLEEP
#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include "SHT21.h"
SHT21 SHT21;

QueueHandle_t queue;
int queueSize = 32;
static int taskCore = 0;

typedef union
{
  uint8_t frame[32];    // data frame to send/receive data
  struct
  {
    uint32_t did;        // destination identifier chipID (4 lower bytes)
    uint32_t sid;        // source identifier chipID (4 lower bytes)
    float   sens[4];     // max 4 values - fields
    uint32_t  tout;      // optional timeout
    uint8_t pad[4];      // padding
  } pack;                // data packet
} DT_t;                  // DaTa type

RTC_DATA_ATTR uint32_t gwID=0, termID;  // gateway identifier stored in RTC memory
int cycle=16, rcv_cycle, rcv_ack=0;      // main cycle in millis

void LoRa_send(uint8_t *frame) {
  LoRa_txMode();                        // set tx mode - normal mode
  LoRa.beginPacket();                   // start packet
  LoRa.write(frame,32);                  // add payload
  LoRa.endPacket(true);                 // finish packet and send it
}

void onReceive(int packetSize) {
  DT_t p;int i=0;
  if(packetSize==32)
    {
    while (LoRa.available())
      {
      p.frame[i]= LoRa.read(); i++;
      }
    Serial.print("Terminal Received Packet:");
    Serial.println(p.pack.tout);
    gwID=p.pack.sid;
    rcv_cycle=(int)p.pack.tout; rcv_ack=1;
    }
}

void sensor_Task( void * pvParameters )
{
float stab[4];
while(true)
  {
  Wire.begin(12,14);
  SHT21.begin(); delay(200);
  stab[0]=(float) SHT21.getTemperature();
  stab[1]=(float) SHT21.getHumidity();
  stab[2]=0.0; stab[3]=0.0;
  Serial.printf("T:%2.2f, H:%2.2f\n",stab[0],stab[1]);
  delay(cycle*1000/4);
  xQueueReset(queue);  // reset queue to store only the last values
  xQueueSend(queue, stab, portMAX_DELAY);
  }
}
```

```
void setup() {
  Serial.begin(9600);
  termID=(uint32_t)ESP.getEfuseMac();
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  queue = xQueueCreate( queueSize, sizeof(float)*4);
  xTaskCreatePinnedToCore(
                    sensor_Task,   /* Function to implement the task */
                    "sensor_Task", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore);  /* Core where the task should run */
  Serial.println("Task created...");
  delay(333);
}

void loop() {

    DT_t p; float stab[8];
    p.pack.did=(uint32_t)gwID;  // in the first cycle unknown and set to 0
    p.pack.sid=(uint32_t)termID;
    p.pack.sens[0]=stab[0]; p.pack.sens[1]=stab[1];
    p.pack.sens[2]=0; p.pack.sens[3]=0;
    p.pack.tout=(uint32_t)millis();
    xQueueReceive(queue, stab, portMAX_DELAY);
    LoRa_send(p.frame); // send a packet
    Serial.println("Send Packet!");
    delay(2000);
    if(rcv_ack)
      {
      cycle=rcv_cycle; rcv_ack=0;
      }
    Serial.printf("Cycle=%d\n",cycle);
    delay(1000*cycle);
  }
```

9.7.3 Le code complet du nœud de passerelle avec la tâche d'affichage

```
#define GATEWAY
#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include <Wire.h>
#include "SSD1306Wire.h"
SSD1306Wire display(0x3c,12,14);
QueueHandle_t queue;
int queueSize = 32;
static int taskCore = 0;

typedef union
{
  uint8_t frame[32];    // data frame to send/receive data
  struct
  {
    uint32_t did;        // destination identifier chipID (4 lower bytes)
    uint32_t sid;        // source identifier chipID (4 lower bytes)
    float   sens[4];     // max 4 values - fields
    uint32_t  tout;      // optional timeout
    uint8_t pad[4];      // padding
  } pack;                // data packet
} DT_t;                  // DaTa type

RTC_DATA_ATTR uint32_t gwID=0, termID;  // gateway identifier stored in RTC memory
RTC_DATA_ATTR uint32_t rcv_pack=0;

void LoRa_send(uint8_t *frame) {
  LoRa_txMode();                        // set tx mode - normal mode
  LoRa.beginPacket();                   // start packet
  LoRa.write(frame,32);                  // add payload
  LoRa.endPacket(true);                 // finish packet and send it
}
```

```
void onReceive(int packetSize) {
  DT_t p;int i=0;
  if(packetSize==32)
    {
    while (LoRa.available())
      {
      p.frame[i]= LoRa.read(); i++;
      }
    Serial.println("Gateway Received Packet ");
    Serial.printf("T:%2.2f, H:%2.2f\n",p.pack.sens[0],p.pack.sens[1]);
    rcv_pack=1;
    xQueueReset(queue);  // reset queue to store only the last values
    xQueueSend(queue, &p, portMAX_DELAY);
    }
}

void disp_Task(void *pvParameters)
{
float stab[4];
DT_t p;
while(true)
  {
  char buff[32];
  xQueueReceive(queue, &p, portMAX_DELAY);
  Wire.begin(12,14); delay(200);
  display.init();delay(200);
  display.flipScreenVertically();
  display.setFont(ArialMT_Plain_10);
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.drawString(0,0,"Gateway - disp_Task");  // first 16 lines are yellow
  sprintf(buff,"T:%2.2f,H:%2.2f\n",p.pack.sens[0],p.pack.sens[1]);
  display.drawString(0,16,buff);
  Serial.println(buff);
  display.display();
  }
}


void setup() {
  Serial.begin(9600);
  gwID=(uint32_t)ESP.getEfuseMac();
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  queue = xQueueCreate(queueSize, sizeof(float)*4);
  xTaskCreatePinnedToCore(
                    disp_Task,   /* Function to implement the task */
                    "disp_Task", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore);  /* Core where the task should run */
  Serial.println("Task created...");
}

void loop()
{
if(runEvery(1000))
  { // repeat every 5000 millis
  DT_t p;
  p.pack.did=(uint32_t)termID;
  p.pack.sid=(uint32_t)gwID;
  p.pack.sens[0]=23.67; p.pack.sens[1]=67.67;
  p.pack.sens[2]=0; p.pack.sens[3]=0;
  p.pack.tout=(uint32_t)(10 + random(10,40));
  if(rcv_pack)
    {
    LoRa_send(p.frame);
    rcv_pack=0;
    Serial.printf("Send Packet with timeout=%d, sec=%d\n",p.pack.tout,millis()/1000);
    }
  }
}
```

9.8 A faire

1. Testez le code des nœuds Terminal et Gateway
2. Modifiez la tâche du capteur en ajoutant une troisième valeur de capteur (luminosité) et testez les deux nœuds.
3. Modifiez le code du noeud Gateway en remplacant la boucle principale **`loop()`** par une tâche (**`LoRa_Task`**)

Lab 10 - Liaison LoRa et couche réseau pour les services TS et MQTT

Dans ce laboratoire, nous allons étudier les couches **réseau** et **application** développées pour la transmission LoRa et l'association avec des serveurs/brokers IoT traditionnels tels que **ThingSpeak** ou **MQTT**.
Notre travail commence par l'analyse et l'explication des bibliothèques préparées. Cette explication est suivie de la présentation de 4 exemples essentiels fonctionnant avec 4 services: **envoyer** à **ThingSpeak**, **recevoir** de **ThingSpeak**, **publier** sur **MQTT**, **souscrire** et **recevoir** message **MQTT**.

10.1 Les bibliothèques

Dans cette section, nous allons analyser les bibliothèques LoRa prêtes à travailler sur: la couches **physique-liens** et les couches **réseau-applications** :

`LoRa_Para.h` - paramètres et fonctions de la couche physique-liens LoRa
`LoRa_Packets.AES.h` – formats des paquets et fonctions pour envoyer les paquets avec AES
`LoRa_onReceive.AES` - fonctions pour recevoir les paquets

10.1.1 `Lora_Para.h`

Commençons par la bibliothèque **`LoRa_Para.h`** qui a été introduite et étudiée dans **Lab.7** (couche physique LoRa). Dans cette bibliothèque nous fournissons la définition des connexions - broches au modem LoRa (**SX1276/8**) et la définition des paramètres de modulation avec leurs valeurs par valeurs par défaut.
Nous avons 3 **fonctions d'initialisation** :

1. **`set_LoRa()`** - pour définir **tous les paramètres par défaut**
2. **`set_LoRa_Para()`** - pour définir les broches spécifiques et les paramètres de modulation
3. **`set_LoRa_Radio_Para()`** - pour définir les paramètres de modulation et les **broches par défaut**

`LoRa_rxMode()`, **`LoRa_txMode()`** et **`onTxDone()`** sont les fonctions permettant de spécifier le **mode IQ** normal ou inverse. Notez que ces modes sont **symétriquement différents** pour les nœuds **Terminal** et **Gateway**.

```
#define SS       5      // NSS
#define RST     15      // RST
#define DI0     26      // INTR
#define SCK     18      // CLK
#define MISO    19      // MISO
#define MOSI    23      // MOSI

#define BAND    868E6   // set frequency
#define SF      7       // set spreading factor
#define SBW     125E3   // set signal bandwidth
#define SW      0xF3    // set Sync Word
#define BR      8       // set bit rate (4/5,5/8)

void set_LoRa()  // all default settings
{
char buff[128];
  SPI.begin(SCK, MISO, MOSI, SS);  // SCK, MISO, MOSI, SS
  LoRa.setPins(SS, RST, DI0);
  Serial.begin(9600);
  delay(1000);
  Serial.println();
  if (!LoRa.begin(BAND)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  sprintf(buff,"BAND=%f,SF=%d,SBW=%f,SW=%X,BR=%d\n",BAND,SF,SBW,SW,BR);
  Serial.println(buff);
  LoRa.setSpreadingFactor(SF);
  LoRa.setSignalBandwidth(SBW);
  LoRa.setSyncWord(SW);
  }
void set_LoRa_Para(int sck,int miso,int mosi,int ss,int rst, int dio0,unsigned long freq,unsigned
sbw, int sf, uint8_t sw)  // user set pins and parameters
{
  SPI.begin(sck, miso, mosi, ss);  // SCK, MISO, MOSI, SS
```

```
  LoRa.setPins(ss, rst, dio0);
  Serial.begin(9600);
  delay(1000);
  Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  LoRa.setSpreadingFactor(sf);
  LoRa.setSignalBandwidth(sbw);
  LoRa.setSyncWord(sw);
  }
void set_LoRa_Radio_Para(unsigned long freq,unsigned sbw, int sf, uint8_t sw)  // radio settings
{
  SPI.begin(SCK, MISO, MOSI, SS);  // SCK, MISO, MOSI, SS
  LoRa.setPins(SS, RST, DI0);
  Serial.begin(9600);
  delay(1000);
  Serial.println();
  if (!LoRa.begin(freq)) {
    Serial.println("Starting LoRa failed!");
    while (1);
  }
  LoRa.setSpreadingFactor(sf);
  LoRa.setSignalBandwidth(sbw);
  LoRa.setSyncWord(sw);
  }

#ifdef TERMINAL
void LoRa_rxMode(){
  LoRa.enableInvertIQ();                // active invert I and Q signals
  LoRa.receive();                       // set receive mode
}

void LoRa_txMode(){
  LoRa.idle();                          // set standby mode
  LoRa.disableInvertIQ();               // normal mode
}
#endif

#ifdef GATEWAY
void LoRa_rxMode(){
  LoRa.disableInvertIQ();               // normal mode
  LoRa.receive();                       // set receive mode
}

void LoRa_txMode(){rdf.pack.channel
  LoRa.idle();                          // set standby mode
  LoRa.enableInvertIQ();                // active invert I and Q signals
}
#endif

void onTxDone() {
  Serial.println("TxDone");
  LoRa_rxMode();
}

boolean runEvery(unsigned long interval)
{
  static unsigned long previousMillis = 0;
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= interval)
  {
    previousMillis = currentMillis;
    return true;
  }
  return false;
}
```

10.1.2 `Lora_Packets.AES.h`

Ce fichier décrit les types de paquets LoRa utilisés pour communiquer entre les terminaux LoRa et les nœuds de passerelle.
Les identificateurs, source : **`sid`** et destination : **`did`** , sont dérivés des identificateurs du circuit ESP32. Seuls 4 octets inférieurs sont pris en compte (**`uint32_t`**)

Les **paquets de contrôle** contiennent **32 octets** et les **paquets de données** sont construits avec **64** pour les services **ThingSpeak** ou **112** octets pour la communication avec les brokers **MQTT**.
Les 2 octets du **champ de contrôle** **`con[0]`** et **`con[1]`** permettent d'identifier le type du paquet et l'organisation des données (**masque**).

Le premier octet - **`con[0]`** indique le **type du paquet** et le **MODE** (service) à utiliser.
La **première valeur hexadécimale** de cet octet indique le **MODE** de la passerelle/du service:

```
1 – ThingSpeak(TS) send, 2 –ThingSpeak(TS) receive
3 – MQTT – MQP publish, 4 – MQTT – MQS subscribe,
```

La **deuxième valeur hexadécimale** indique le **type du paquet** :

```
0x01 – IDREQ : ID request for any service and ID acknowledge
0x11 – IDREQ, 0x12 – IDACK: ID request and ID acknowledge – TS send – TSS
0x21 – IDREQ, 0x22 – IDACK: ID request and ID acknowledge – TS receive –TSR
0x31 – IDREQ, 0x32 – IDACK: ID request and ID acknowledge – MQTT publish – MQP
0x41 – IDREQ, 0x42 – IDACK: ID request and ID acknowledge – MQTT subscribe – MQS

0x13 – DTSND, 0x14 – DTACK: DATA send and DATA acknowledge – TS send  – TSS
0x23 – DTREQ, 0x24 – DTRCV: DATA request and DATA receive – TS receive – TSR
0x33 – DTPUB, 0x34 – DTPUBACK: DATA publish and DATA publish acknowledge – MQTT publish – MQP
0x43 – DTSUB, 0x44 – DTSUBRCV: DATA subscribe and DATA subscribe receive – MQTT subscribe – MQS
```

Le deuxième octet du champ de contrôle **`con[1]`** est utilisé pour transporter le **masque de champs de données** nécessaire pour marquer les champs valides dans le **canal ThingSpeak**. Chaque bit de ce champ correspond à une valeur de données ou un champ du canal ThingSpeak.
Avec ce masque, les champs peuvent être spécifiés lors de **l'envoi** ou de la **demande** des données.
Le champ **`pass`** contient le mot de passe sur 16 octets à utiliser par les paquets de contrôle type **`IDREQ`** afin de protéger le premier accès au nœud passerelle. Seule une trame avec un mot de passe correct doit être confirmée par le paquet **`IDACK.`** Un paquet de confirmation **`IDACK`** porte le **code de chiffrement AES** à utiliser pour les **paquets de données**.
Le nœud de passerelle peut envoyer un délai calculé (valeur de temporisation) en fonction de son ordonnanceur (agenda) des nœuds Terminal.

Les paquets de données sont cryptés avec l'algorithme AES qui est implémenté/intégré directement dans ESP32. Les fonctions AES **`encrypt()`** et **`decrypt()`** suivantes qui sont définies pour protéger les paquets avec un mot-clé symétrique.

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

void encrypt(unsigned char *plainText,char *key,unsigned char *outputBuffer, int nblocks)
{
  mbedtls_aes_context aes;
  mbedtls_aes_init( &aes );
  mbedtls_aes_setkey_enc(&aes,(const unsigned char*)key,strlen(key)*8);
  for(int i=0;i<nblocks;i++)
    {
    mbedtls_aes_crypt_ecb(&aes,MBEDTLS_AES_ENCRYPT,
                          (const unsigned char*)(plainText+i*16), outputBuffer+i*16);
    }
  mbedtls_aes_free(&aes);
}

void decrypt(unsigned char *chipherText,char *key,unsigned char *outputBuffer, int nblocks)
{
  mbedtls_aes_context aes;
  mbedtls_aes_init( &aes );
  mbedtls_aes_setkey_dec( &aes, (const unsigned char*) key, strlen(key) * 8 );
    for(int i=0;i<nblocks;i++)
    {
    mbedtls_aes_crypt_ecb(&aes,MBEDTLS_AES_DECRYPT,
                        (const unsigned char*)(chipherText+i*16), outputBuffer+i*16);
    }
  mbedtls_aes_free(&aes );
}
```

Le format des paquets de contrôle est le même pour tous les types de services (**TSS**, **TSR**, **MQP**, **MQS**).

Il est représenté par l'union/structure suivante.

```
typedef union
{
  uint8_t frame[32];
  struct
  {
    uint32_t did;       // destination identifier chipID (4 lower bytes)
    uint32_t sid;       // source identifier chipID (4 lower bytes)
    uint8_t  con[2];    // control field: con[0]
    char     pass[16];  // password - 16 characters
    int tout;           // timeout
    uint8_t  pad[2];    // future use
  } pack;               // control packet
} conframe_t;
```

Le format des paquets de données est différent pour les services **ThingSpeak** (TSS/TSR) et **MQTT** (MQP,MQS).
Ils sont représentés par les unions/structures suivants.

```
typedef union
{
  uint8_t frame[64];      // TS frame to send/receive data
  struct
  {
    uint32_t did;         // destination identifier chipID (4 lower bytes)
    uint32_t sid;         // source identifier chipID (4 lower bytes)
    uint8_t  con[2];      // control field: lower byte is used as mask
    int      channel;     // TS channel number
    char     keyword[16]; // write (or read) keyword
    float    sens[8];     // max 8 values - fields
    uint16_t  tout;       // optional timeout
  } pack;                 // data packet
} TSframe_t;

typedef union
{
  uint8_t frame[112];    // MQTT frame to publish on the given topic
  struct
  {
    uint32_t did;        // destination identifier chipID (4 lower bytes)
    uint32_t sid;        // source identifier chipID (4 lower bytes)
    uint8_t  con[2];     // control field
    char     topic[48];  // topic name - e.g. /esp32/Term1/Sens1
    char     mess[48];   // message value
    int      tout;       // optional timeout for publish frame
    uint8_t  pad[2];     // future use
  } pack;                // data packet
} MQTTframe_t;
```

10.2 Fonctions de contrôle pour différents services - MODE

La fonction **`send_IDREQ )`** est utilisée par le nœud Terminal pour démarrer la communication avec un nœud passerelle. L'identifiant de l'adresse de destination n'est pas connu (**`scf.pack.did=(uint32_t) 0;`**).
Le code de contrôle hexadécimal est calculé à partir de la valeur de code (**MODE**).
Par exemple pour **TSS** - **T**hing**S**peak **S**end (MODE = 1), la valeur calculée est **`0x11`**.
Le paquet **`IDREQ`** doit porter un mot de passe à valider par la passerelle.

```
#ifdef TERMINAL              // this packet is sent only by the TERMINAL nodes

void send_IDREQ(char *pass)  // TERM: request ID for all modes: MODE, termID – global
{
conframe_t scf,sccf;
  LoRa_txMode();
  LoRa.beginPacket();
  scf.pack.did=(uint32_t)0;
  scf.pack.sid=(uint32_t)termID;
  scf.pack.con[0]=(uint8_t)(MODE*16+1);   // IDREQ_MODE – type 1 control TERM to GW
  scf.pack.con[1]=0x00;
  if(pass!=NULL)
     strncpy(scf.pack.pass,pass,16);      // password to validate by gateway
  LoRa.write(scf.frame,32);
  LoRa.endPacket(true);
}
```

Le nœud passerelle répond avec le paquet **IDACK** envoyé par la fonction suivante.
Le paquet **IDACK** porte l'identifiant du nœud passerelle - **`gwID`** comme adresse source – **`scf.pack.sid`**.
Il contient également la clé AES à utiliser dans les paquets de données.

```
#ifdef GATEWAY                                 // this packet is sent only by gateway node

void send_IDACK(char *aes, uint16_t tout)      // GW: ID ACK for all modes: code
{
conframe_t scf,sccf;
  LoRa_txMode();
  LoRa.beginPacket();
  scf.pack.did=(uint32_t)termID;
  scf.pack.sid=(uint32_t)gwID;
  scf.pack.con[0]=(uint8_t)16*MODE+2;  // IDACK_MODE – type 2 – control GW to TERM
  scf.pack.con[1]=0x00;
  strncpy(scf.pack.pass,aes,16);       // AES key for data packets
  scf.pack.tout=tout;                  // optional timeout for the next packet
  LoRa.write(scf.frame,32);
  LoRa.endPacket(true);
}
```

Notez que l'argument **`tout`** peut fournir la valeur du délai d'expiration au terminal. Ce délai peut être utilisé pour planifier le moment où la première trame de données doit être envoyée au nœud passerelle.

10.2.1 Fonctions d'envoi des paquets de données pour TSS - MODE 1

Commençons par présenter 2 paquets de données et envoyés par les fonctions utilisées pour implémenter notre premier service (**MODE = 1**) qui est l'**envoi des paquets** (valeurs des capteurs) **au serveur ThingSpeak** via un nœud Gateway (LoRa-WiFi Gateway).

Elles sont:

Étape 1 - le même pour tous les services ou MODES vient d'être présenté ci-dessus.
- Terminal: **`IDREQ`** avec **`send_IDREQ()`**
- Gateway: **`IDACK`** avec **`send_IDACK()`**

Étape 2
- Terminal: **`DTSND`** avec **`send_DTSND()`**
- Gateway: **`DTACK`** avec **`avec_DTACK()`**

Dans cette section nous allons présenter les deux fonctions permettant d'envoyer les données de capteurs à la passerelle et de recevoir la confirmation de leur envoie sur le serveur ThingSpeaK.

10.2.1.1 Fonction `send_DTSND (TERMINAL)`

La fonction **`data_DTSND()`** envoie un paquet vers la passerelle (**`gwID`**). Le type de la fonction est 3 et le mode 1 (**`0x13`**) est enregistré dans l'octet de contrôle **`con[0]`**.
Le paquet porte le numéro du canal (**`chan`**) et la clé d'écriture dans ce canal. Le byte **`mask`** indique les champs de capteurs valides dans cet envoie.
L'ensemble du paquet est encrypté avec la clé (**`CKEY`**) précédemment reçu dans le paquet **`IDACK`**.
Notez que les variables **`gwID`**, **`termID`**, et **`CKEY`** sont globales et enregistrées dan la mémoire RTC non modifiable par **deep_sleep**.

```
void send_DTSND(uint8_t mask,int chan,char *wkey,float *stab)  // TERM: send data to TS, gwID,
termID – global
{
TSframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)gwID;
  sdf.pack.sid=(uint32_t)termID;
  sdf.pack.con[0]=(uint8_t)(MODE*16+3);              // MODE 1 – type 3  data TERM to GW
  sdf.pack.channel= chan;
  strncpy(sdf.pack.keyword,wkey,16);
  sdf.pack.con[1]=mask;
  for(int i=0;i<8;i++) sdf.pack.sens[i]=stab[i];
  sdf.pack.tout=0;
  encrypt(sdf.frame,CKEY,sdcf.frame,4);
  LoRa.write(sdcf.frame,64);
  LoRa.endPacket(true);
}
```

10.2.1.2 Fonction `send_DTACK (GATEWAY)`

Après la réception du paquet **`DTSND`** du côté du nœud de passerelle par un ISR **`onReceive()`**, le nœud passerelle répond par un paquet **`DTACK`** (*Data Acknowledge*).
Dans ce paquet, le nœud passerelle peut envoyer une valeur de temporisation et un message de contrôle supplémentaire. Le paquet est crypté avec la clé AES (**`CKEY`**).
Le message de contrôle indique l'état de réception des données envoyées sur ThingSpeak.

```
void send_DTACK(uint8_t mask,int tout,char *wkey)  // GW: ACK data to TERM
{
TSframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)termID;
  sdf.pack.sid=(uint32_t)gwID;
  sdf.pack.con[0]=(uint8_t)16*MODE+4;
  sdf.pack.con[1]=mask;

  sdf.pack.channel= tout;               // calculated timeout for the next data frame
  if (wkey!=NULL)
    strncpy(sdf.pack.keyword,wkey,16);  // message indicating the reception state at ThingSpeak
  for(int i=0;i<8;i++) sdf.pack.sens[i]=0.0;
  sdf.pack.tout=0;
  encrypt(sdf.frame,CKEY,sdcf.frame,4);
  LoRa.write(sdcf.frame,64);
  LoRa.endPacket(true);
}
```

10.2.2 Fonctions d'envoi des paquets de données pour TSR - MODE 2

Dans le MODE=2 le Terminal envoie la demande des données par la fonction **`send_DTREQ()`** et il attend le résultat à la réception d'un paquet **`DTRCV`** envoyé par la passerelle.

10.2.2.1 Fonction `send_DTREQ() (TERMINAL)`

La fonction **`send_DTREQ()`** exécutée au niveau du nœud Terminal envoie le paquet **`DTREQ`** incluant le champ de masque pour les valeurs requises, le numéro du canal et la clé de lecture permettant de récupérer les données du serveur.

```
#ifdef TERMINAL
void send_DTREQ(uint8_t mask,int chan,char *rkey)  // TERM: send data request to TS
{
TSframe_t sdf,sdcf;
```

```
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)gwID;
  sdf.pack.sid=(uint32_t)termID;
  sdf.pack.con[0]=(uint8_t)(MODE*16+3);               // MODE  2 - type 3  data TERM to GW
  sdf.pack.con[1]=mask;
  sdf.pack.channel= chan;
  strncpy(sdf.pack.keyword,rkey,16);
  for(int i=0;i<8;i++) sdf.pack.sens[i]=0.0;
  sdf.pack.tout=0;
  encrypt(sdf.frame,CKEY,sdcf.frame,4);
  LoRa.write(sdcf.frame,64);
  LoRa.endPacket(true);
}
```

10.2.2.2 Fonction `send_DTRCV() (GATEWAY)`

Du côté de la passerelle, après la réception d'un paquet **`DTREQ`**, nous demandons les données (à recevoir du serveur ThingSpeak). Après leurs réception nous les envoyons dans un paquet **`DTRCV`** (*Data Received*) par la fonction **`send_DTRCV()`**.
Ce paquet transporte le champ de données obtenues marqué par l'octet de masque. Il peut également transmettre la nouvelle valeur de temporisation dans l'argument de canal ainsi qu'un message de contrôle de 16 octets) à la place de la valeur de clé de lecture.

```
#ifdef GATEWAY
void send_DTRCV(uint8_t mask,int tout,char *mess,float stab[])  // GW: send received data in stab[]
{
TSframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)termID;
  sdf.pack.sid=(uint32_t)gwID;
  sdf.pack.con[0]=(uint8_t)16*MODE+4;      // MODE 2 - type 4: GW to TERM
  sdf.pack.con[1]=mask;

  sdf.pack.channel= tout;                  // calculated timeout for the next data frame
  strncpy(sdf.pack.keyword,mess,16);
  for(int i=0;i<8;i++) sdf.pack.sens[i]=stab[i];
  sdf.pack.tout=0;
  encrypt(sdf.frame,CKEY,sdcf.frame,4);
  LoRa.write(sdcf.frame,64);
  LoRa.endPacket(true);
}
```

10.2.3 Fonctions d'envoi des paquets de données pour MQP (MQTT Publish) – MODE 3

Les 2 fonctions suivantes nous permettent de créer un service de publication **MQTT** avec des paquets **`DTPUB`** et **`DTPUBACK`**.

10.2.3.1 Fonction `send_DTPUB() (TERMINAL)`

Du côté du terminal, la fonction **`send_DTPUB()`** prend le sujet (**`topic`**) proposé et le message (**`mess`**) associé et l'envoie dans un paquet crypté AES de **112 octets**.

```
#ifdef TERMINAL
void send_DTPUB(char *topic, char *mess)      // TERM: send DTPUB packet
{
MQTTframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)gwID;
  sdf.pack.sid=(uint32_t)termID;
  sdf.pack.con[0]=(uint8_t)(MODE*16+3);        // MODE  3 - type 3  data TERM to GW
  sdf.pack.con[1]=0x00;
  strcpy(sdf.pack.topic,topic);
  strcpy(sdf.pack.mess,mess);
  sdf.pack.tout=0;
  encrypt(sdf.frame,CKEY,sdcf.frame,7);       // 7*16=112 - 7 16-byte blocks
  LoRa.write(sdcf.frame,112);
  LoRa.endPacket(true);
}
```

10.2.3.2 Fonction `send_DTPUBACK() (GATEWAY)`

Du côté passerelle, nous confirmons la réception du paquet **DTPUB** en envoyant le paquet **DTPUBACK** correspondant.

```
#ifdef GATEWAY
void send_DTPUBACK(char *topic,char *mess, int tout)  // GW: send ACK for PUB message
{
MQTTframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)termID;
  sdf.pack.sid=(uint32_t)gwID;
  sdf.pack.con[0]=(uint8_t)16*MODE+4;               // MODE 3 - type 4: GW to TERM
  sdf.pack.con[1]=0x00;
  strncpy(sdf.pack.topic,topic,48);
  strncpy(sdf.pack.mess,mess,48);
  sdf.pack.tout=tout;
  encrypt(sdf.frame,CKEY,sdcf.frame,7);
  LoRa.write(sdcf.frame,112);
  LoRa.endPacket(true);
}
```

10.2.4 Fonctions d'envoi des paquets de données pour MQS (MQTT Subscribe) – MODE 4

Dans le MODE=4 le Terminal envoie un paquet d'abonnement au sujet donné à la passerelle, puis il attend les données (messages) envoyés vers ce sujet par d'autre terminaux.

10.2.4.1 Fonction `send_DTSUB() (TERMINAL)`

Le nœud Terminal envoie un paquet **DTSUB** pour s'abonner à un sujet (topic) MQTT. Ceci est fait par la fonction **`send_DTSUB()`** suivante :

```
void send_DTSUB(char *topic)                        // TERM:send DTSUB frame - subscribe topic
{
MQTTframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)gwID;
  sdf.pack.sid=(uint32_t)termID;
  sdf.pack.con[0]=(uint8_t)(MODE*16+3);           // MODE  4 - type 3  data TERM to GW
  sdf.pack.con[1]=0x00;
  strncpy(sdf.pack.topic,topic,48);
  memset(sdf.pack.mess,0x00,48);
  sdf.pack.tout=0;
  encrypt(sdf.frame,CKEY,sdcf.frame,7);
  LoRa.write(sdcf.frame,112);
  LoRa.endPacket(true);
}
```

10.2.4.2 Fonction `send_DTSUBRCV() (GATEWAY)`

L'arrivée d'une nouvelle donnée au sujet souscrit et la réception de ces données dans le nœud passerelle est retransmis au nœud Terminal par la fonction send_DTSUBRCV () présentée ci-dessous. Cette fonction prépare et envoie un paquet DTSURCV contenant le nom du sujet (**topic**) et la valeur du message (**mess**).

```
void send_DTSUBRCV(char *topic,char *mess,int tout)  // GW:send received message
{
MQTTframe_t sdf,sdcf;
  LoRa_txMode();
  LoRa.beginPacket();
  sdf.pack.did=(uint32_t)termID;
  sdf.pack.sid=(uint32_t)gwID;
  sdf.pack.con[0]=(uint8_t)16*MODE+4;   // MODE 4 - type 4: GW to TERM
  sdf.pack.con[1]=0x00;
  strncpy(sdf.pack.topic,topic,48);
  strncpy(sdf.pack.mess,mess,48);
  sdf.pack.tout=tout;
  encrypt(sdf.frame,CKEY,sdcf.frame,7);
  LoRa.write(sdcf.frame,112);
  LoRa.endPacket(true);
}
```

10.3 Fichier des fonctions de réception : `Lora_onReceive.AES.h`

La réception des paquets se fait via une ISR (*Interrupt Service Routine*) **`onReceive()`**.
Le fichier suivant contient les données requises et l'**ISR global `onReceive()`** préparé pour la réception de tous les paquets de contrôle et de données.
Les données déclarées sont stockées dans la mémoire RTC qui persistent pendant l'opération deep_sleep de l'ESP32.
Les drapeaux: **`stage1_flag`** et **`stage2_flag`** indiquent l'arrivée d'un paquet de contrôle (**stage1**) ou d'un paquet de données (**stage2**).

Les paquets reçus dans ISR sont envoyés dans la file d'attente (**FreeRTOS**). Le mécanisme de file d'attente permet la communication immédiate des données reçues à la tâche principale (**`loop()`**) ou une autre tâche attendant les données.

```
QueueHandle_t tsrcv_queue, mqrcv_queue, con_queue;   // receive queues
int queueSize = 32;
static int taskCore = 0;

uint8_t mask; int channel; float stab[8];

uint8_t IDREQ_mode,IDACK_mode;
uint8_t DT_mode,ACK_mode;  // data packet receive GW and data packet receive TERM

char *password="passwordpassword";

void onReceive(int packetSize)
{
if(packetSize==32)
  {
  conframe_t rcf; int i=0;
  while (LoRa.available()) { rcf.frame[i] = LoRa.read();i++;}
#ifdef MASTER
  IDREQ_mode=MODE*16+1;
  if(rcf.pack.con[0]==IDREQ_mode && rcf.pack.did==0x00 && !strncmp(password,rcf.pack.pass,16))  //
GW:received IDREQ_MODE packet
    { stage1_flag=1;
      // termID=rcf.pack.sid;
      xQueueReset(con_queue);  // reset queue to keep only the last packet
      xQueueSendFromISR(con_queue, rcf.frame, NULL); Serial.println("Received IDREQ_MODE");
    }
#endif
#ifdef TERMINAL
  IDACK_mode=MODE*16+2;
  if(rcf.pack.con[0]==IDACK_mode && rcf.pack.did==termID )  // TERM:received IDACK_MODE packet
    { stage1_flag=1;
      gwID=rcf.pack.sid; Serial.println("Received IDACK_MODE");
      xQueueReset(con_queue);  // reset queue to keep only the last packet
      xQueueSendFromISR(con_queue, rcf.frame, NULL); Serial.println("Received IDACK_MODE");
    }
#endif
  }

#if MODE<3
if(packetSize==64)
  {
  TSframe_t rdf,rdcf; int i=0;
  char ckey[17];
  strncpy(ckey,CKEY,16);ckey[16]='\0';
  while (LoRa.available()) { rdcf.frame[i] = LoRa.read();i++;}
  decrypt(rdcf.frame,ckey,rdf.frame,4);

  #ifdef MASTER
  DT_mode=MODE*16+3;
  if(rdf.pack.con[0]==DT_mode && rdf.pack.did==gwID)  // GW:received data packet
    { stage2_flag=1; termID=rdf.pack.sid;  Serial.println("GW:Received TS data_MODE");
      xQueueReset(strcv_queue);  // reset queue to keep only the last packet
      xQueueSend(strcv_queue, rdf.frame, portMAX_DELAY);
    }
  #endif

  #ifdef TERMINAL
  ACK_mode=MODE*16+4;
```

```
  if(rdf.pack.con[0]==ACK_mode && rdf.pack.did==termID)  // TERM:received data packet
    { stage2_flag=1;
      gwID=rdf.pack.sid; Serial.println("TERM:Received TS data_MODE");
      xQueueReset(strcv_queue);  // reset queue to keep only the last packet
      xQueueSend(strcv_queue, rdf.frame, portMAX_DELAY);
    }
  #endif
  }
#endif

#if MODE>2
if(packetSize==112)
  {
  MQTTframe_t rdf,rdcf; int i=0;
  char ckey[17];
  strncpy(ckey,CKEY,16);ckey[16]='\0';
  while (LoRa.available()) { rdcf.frame[i] = LoRa.read();i++;}
  decrypt(rdcf.frame,ckey,rdf.frame,7);
  #ifdef MASTER
  DT_mode=MODE*16+3;
  if(rdf.pack.con[0]==DT_mode && rdf.pack.did==gwID)  // GW:received data packet
    { stage2_flag=1; termID=rdf.pack.sid;  Serial.println("GW:Received MQ data_MODE");
      xQueueReset(mqrcv_queue);  // reset queue to keep only the last packet
      xQueueSend(mqrcv_queue, rdf.frame, portMAX_DELAY);
    }
  #endif
  #ifdef TERMINAL
  ACK_mode=MODE*16+4;
  if(rdf.pack.con[0]==ACK_mode && rdf.pack.did==termID)  // TERM:received data packet
    { stage2_flag=1;
      gwID=rdf.pack.sid; Serial.println("TERM:Received MQ data_MODE");
      xQueueReset(mqrcv_queue);  // reset queue to keep only the last packet
      xQueueSend(mqrcv_queue, rdf.frame, portMAX_DELAY);
    }
  #endif
  }
#endif
}
```

10.5 Implémentation des services avec le *front-end* de passerelles

Dans cette deuxième partie du Lab.10, nous présentons les codes simples pour 4 services. Chaque exemple comprend le code pour le terminal et le côté passerelle. Le côté passerelle contient uniquement la partie frontale, y compris l'interface LoRa.

10.5.1 Code du terminal et de la passerelle - MODE 1

Nous commençons par le code du **TERMINAL** fonctionnant en **MODE 1**, il fonctionne comme code émetteur LoRa vers le serveur ThingSpeak par le biais de la passerelle.

Le code commence par la définition de :

le type de nœud : **`TERMINAL`**	[TERMINAL, GATEWAY]
le **`MODE : 1`**	[1- TSS, 2-TSR, 3-MQP, 4-MQS], et
la **`CKEY`** : **`"abcdefghijklmnop"`**	- clé de chiffrement AES

Ensuite, nous incluons un certain nombre de bibliothèques :

- **`SPI.h`** - pour importer les opérations sur le bus SPI
- **`LoRa.h`** - pour inclure les fonctions de contrôle du modem LoRa
- **`LoRa_Para.h`** - pour utiliser les fonctions d'initialisation des broches et du modem LoRa
- **`LoRa_Packets.h`** - pour utiliser les formats de paquets prédéfinis et les fonctions d'envoi
- **`LoRa_onReceive.h`** - pour utiliser la fonction **`onReceive()`** prédéfinie pour capturer différents types de paquets. **`LoRa_onReceive.h`** intègre les déclarations des identifiants du terminal et de la passerelle: **`termID`**, **`gwID`** qui sont stockés dans la **mémoire RTC** et 2 drapeaux (**`stage1_flag`**, **`stage2_flag`**) indiquant l'étape de communication : contrôle (**stage1**) et données (**stage2**).
  Le fichier **`LoRa_onReceive.h`** contient les déclarations des files d'attente pour la communication de différents types de paquets : **`QueueHandle_t tsrcv_queue, mqrcv_queue, con_queue;`** et le **password** à envoyer dans le paquet **`IDREQ`**.

10.5.1.1 Code complet du terminal en MODE 1

Le code suivant intègre une union/structure permettant de grouper les paramètres à envoyer vers le serveur ThingSpeak pour pouvoir y enregistrer des nouvelles données : numéro du canal, clé d'écriture, masque des champs à écrire.

```
#define TERMINAL   // to define TERMINAL or GATEWAY  node
#define MODE 1     // to choose sender (1), receiver (2), publisher (3), subscriber (4) mode
#include <Wire.h>
#include "SHT21.h"
SHT21 SHT21;
#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include "LoRa_Packets.AES.h"
#include "LoRa_onReceive.AES.h"

union               // TS channel send and receive parameters
{
  uint8_t para[24];
  struct
  {
    int chan;      // channel number
    char key[16];  // write key
    uint8_t zero;
    uint8_t mask;  // sensor mask
    uint8_t pad[2];
  } pack;
} ts;

void get_sens()
{
    SHT21.begin();
    stab[0]=SHT21.getTemperature();
    delay(100);
    stab[1]=SHT21.getHumidity();
    Serial.printf("T:%2.2f, H:%2.2f\n",stab[0],stab[1]);
}
```

```
void setup() {
  Serial.begin(9600); delay(100);
  Wire.begin(12,14);
  termID=(uint32_t)ESP.getEfuseMac(); delay(100);
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  con_queue = xQueueCreate(queueSize, 32);
  rcv_queue = xQueueCreate(queueSize, 64);
  stage1_flag=1;stage2_flag=0;
  ts.pack.chan=1243348; strncpy(ts.pack.key,"J4K8ZIWAWE8JBIX7",16);
  ts.pack.zero=0x00;ts.pack.mask=0xC0;
  ts.pack.pad[0]=0x00;ts.pack.pad[1]=0x00;
  // the above parameters may be read from external EEPROM to ts union-structure (24-bytes)
}

int cycle=10;   // 10 seconds

void loop()
{
if(runEvery(cycle*1000))
  {
  if(cycle_cnt>10) { stage1_flag=1; stage2_flag=0;cycle_cnt=0; }
  else { Serial.printf("cycle=%d\n",cycle_cnt); cycle_cnt++; }
  if(stage1_flag==1)  // runs until IDACK sets stage1_flag to 0 - in RTC memory
    {
    conframe_t rcf; int rec=0;
    send_IDREQ("passwordpassword"); // send IDACK MODE 1
    Serial.printf("sent IDREQ MODE=%x sent from TERM: %08X\n",MODE,(uint32_t)termID);
    rec=xQueueReceive(con_queue, rcf.frame, 100); // delay in number of ticks
    if(rec)
      {
        Serial.printf("recv IDACK service=%x got from GW: %08X ckey=%s\n",MODE,
                      (uint32_t)gwID,rcf.pack.pass);
        strncpy(CKEY,rcf.pack.pass,16);
        stage1_flag=0; stage2_flag=1;  // transition from stage 1 to stage2
      }
    }
  if(stage2_flag==1)
    {
    TSframe_t rdf;
    Serial.println("sensor");
    get_sens();  // get stab[] values
    Serial.printf("sent DTSND MODE=%x sent from TERM: %08X\n",MODE,(uint32_t)termID);
    send_DTSND(ts.pack.mask,ts.pack.chan,ts.pack.key,stab);
    xQueueReceive(rcv_queue, rdf.frame, portMAX_DELAY);
    Serial.printf("recv DTACK MODE=%x from GW:%08X, cycle=%dsec, mess=%s\n",MODE,
                  (uint32_t)rdf.pack.sid,rdf.pack.channel,rdf.pack.keyword);
    }
  }
}
```

10.5.1.2 Code complet de la passerelle (partie frontale) en MODE 1

```
#define GATEWAY
#define MODE  1
#define CKEY "abcdefghijklmnop"  // AES key - 16 bytes

#include <SPI.h>
#include <LoRa.h>
#include <Wire.h>
#include "LoRa_Para.h"
#include "LoRa_Packets.AES.h"
#include "LoRa_onReceive.AES.h"  // to capture the packets with MODE and server/broker

int tss_flag=0;

void TS_Task( void * pvParameters ){
float stab[8]; char kbuff[32];
TSframe_t rdf; int r,x,d;
while(true)
  {
  xQueueReceive(rcv_queue, &rdf, portMAX_DELAY);
  Serial.printf("channel:%d,mask=%x,wkey:%16.16s\n",rdf.pack.channel,rdf.pack.con[1],
                rdf.pack.keyword);
```

```
  LoRa.idle(); // during this period the LoRa modem must be idle

  d=1000+random(1000,3000);  // this code section imitates the behaviour of ThingSpeak server
  delay(d);
  r=random(1,100);
  if(r>50) x=200; else x=100;

  if(x == 200)
    { Serial.println("Channel update successful.");tss_flag=1;}
  else
    { Serial.println("Problem updating channel. HTTP error code " + String(x));tss_flag=2;}
  LoRa_rxMode();
  }
}

void setup() {
  Serial.begin(9600); delay(100);
  gwID=(uint32_t)ESP.getEfuseMac(); // we are in MASTER
  delay(100);
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  rcv_queue = xQueueCreate(queueSize, 64);
  con_queue = xQueueCreate(queueSize, 32);
  xTaskCreatePinnedToCore(
                    TS_Task,   /* Function to implement the task */
                    "TS_Task", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore);  /* Core where the task should run */
  Serial.println("Task created...");
  stage1_flag=1;
}

void loop() {
char mess[16];
if(stage1_flag==1)
  {
  conframe_t rcf; int rec=0;
  rec=xQueueReceive(con_queue, rcf.frame, portMAX_DELAY);
  termID=rcf.pack.sid;
  Serial.printf("recv IDREQ MODE=%x from TERM: %08X, pass=%16.16s\n",MODE, termID,rcf.pack.pass);
  send_IDACK(CKEY,0x0000); // send IDACK for service 1 with CKEY
  Serial.printf("IDACK service=%x sent\n",MODE);stage1_flag=0;
  }
if(stage2_flag==1)
  {
  TSframe_t rdf;  int tout=10; char mess[24];
  strcpy(mess,"control message");
  while(tss_flag==0)
    {
      Serial.println("wating for TS return");
      delay(1000);
    }
  //rcv_queue in TS task
  if(tss_flag==1) {Serial.println("got TS return OK");   strcpy(mess,"TS update OK    ");}
  if(tss_flag==2) {Serial.println("got TS return ERROR");strcpy(mess,"TS update    ERR");}
  tss_flag=0; tout=5+random(10,20);
  send_DTACK(rdf.pack.con[1],tout,mess); // send DTACK: mask, channel from DTSND
  Serial.printf("DTACK to TERMINAL: %08X\n",termID);stage2_flag=0;
  }
}
```

10.5.2 Nœuds terminaux et passerelle - MODE 2 (TSR)

MODE 2 est prêt à envoyer les paquets de demandes de données au serveur ThingSpeak et à attendre les données demandées.
Le nœud Terminal commence par établir la liaison logique avec la passerelle à l'étape 1, puis il envoie les paquets **DTREQ**. Après la réception des données demandées, le nœud passerelle les envoie au nœud terminal dans un paquet de type **DTRCV**.

10.5.2.1 Code du Terminal en MODE 2

Du côté du terminal, nous commençons par l'étape1 pour obtenir l'identifiant **gwID** (si un tel service est disponible!).
Puis, à l'étape 2, le terminal envoie le ou les paquets de demande de données - **DTREQ** à la passerelle détectée.

```
#define TERMINAL  // to choose TERMINAL or MASTER (GATEWAY) node
#define MODE 2  // to choose sender (1), receiver (2) , publisher (3), subscriber (4) mode

#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include "LoRa_Packets.AES.h"
#include "LoRa_onReceive.AES.h"  // to capture the packets with MODE and server/broker
#include <Wire.h>
#include "SSD1306Wire.h"
SSD1306Wire display(0x3c, 12, 14);

union
{
  uint8_t para[24];
  struct
  {
    int chan;      // channel number
    char key[16];  // write-read key
    uint8_t zero;
    uint8_t mask;  // sensor mask
    uint8_t pad[2];
  } pack;
} ts;  // TS send and receive para

void disp_sens(uint8_t mask, float *stab)
{
  char buff[32];
  display.init();
  display.flipScreenVertically();
  display.setFont(ArialMT_Plain_10);
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.drawString(0,0,"Terminal TSR");  // first 16 lines are yellow
  if(mask&0x80) { sprintf(buff,"T:%2.2f",stab[0]);display.drawString(0,16,buff); }
  if(mask&0x40) { sprintf(buff,"H:%2.2f",stab[1]);display.drawString(0,28,buff); }
  if(mask&0x20) { sprintf(buff,"L:%4.2f",stab[2]);display.drawString(0,40,buff); }
  display.display();
}



void setup() {
  Serial.begin(9600); delay(100);
  Wire.begin(12,14);
  termID=(uint32_t)ESP.getEfuseMac(); delay(100);
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  con_queue = xQueueCreate(queueSize, 32);
  rcv_queue = xQueueCreate(queueSize, 64);
  stage1_flag=1;stage2_flag=0;
  ts.pack.chan=1243348; strncpy(ts.pack.key,"0XYA1MAWXFGVWDX9",16);
  ts.pack.zero=0x00;ts.pack.mask=0xE0;  // 3rd sensor
  ts.pack.pad[0]=0x00;ts.pack.pad[1]=0x00;
  // the above parameters may be read from external-internal EEPROM as 24 bytes
}

int cycle=10;   // 10 seconds
```

```
void loop()
{
if(runEvery(cycle*1000))
  {
  if(cycle_cnt>10) { stage1_flag=1; stage2_flag=0;cycle_cnt=0; }
  else { Serial.printf("cycle=%d\n",cycle_cnt); cycle_cnt++; }

  if(stage1_flag==1)
      {
      conframe_t rcf; int rec=0;
      send_IDREQ("passwordpassword"); // send IDACK for service 1
      Serial.printf("sent IDREQ MODE=%x sent from TERM: %08X\n",MODE,(uint32_t)termID);
      rec=xQueueReceive(con_queue, rcf.frame, 100);
      if(rec)
      {
        Serial.printf("recv IDACK service=%x got from GW: %08X ckey=%16.16s\n",MODE,(uint32_t)gwID,
                      rcf.pack.pass);
        strncpy(CKEY,rcf.pack.pass,16);
        stage1_flag=0; stage2_flag=1;  // transition from stage 1 to stage2
      }
    }

  if(stage2_flag==1)
    {
    TSframe_t rdf;  int rec=0;
    Serial.printf("DTREQ service=%x sent to GW: %08X\n",MODE,(uint32_t)gwID);
    send_DTREQ(ts.pack.mask,ts.pack.chan,ts.pack.key);  // channel and read key
    rec=xQueueReceive(rcv_queue, rdf.frame, cycle*100);
    if(rec) Serial.printf("\nDTRCV: cycle=%dsec,%s,%x\n", rdf.pack.channel, rdf.pack.keyword,
                          rdf.pack.con[0]);
    if(rdf.pack.channel>5 && rdf.pack.channel< 3600) // filters errors
      {
      disp_sens(ts.pack.mask,rdf.pack.sens);
      cycle=rdf.pack.channel;
      }
    }
  }
}
```

A noter que le champ **`rdf.pack.channel`** dans le paquet **`DTRCV`** peut porter la valeur du **délai d'expiration**. Par conséquent, cette valeur peut être utilisée pour planifier l'émission du prochain paquet **`DTREQ`**.

10.5.2.2 Code de la passerelle en MODE 2

Du côté de la passerelle, le nœud attend d'abord le paquet **`IDREQ`**. Si le service est disponible, le nœud répond avec le paquet **`IDACK`**.
Dans l'étape suivante, le nœud passerelle attend un paquet **`DTREQ`**. Après sa réception, il peut relayer la demande au serveur ThingSpeak ou fournir directement les données demandées comme dans l'exemple suivant.

```
#define MASTER  // to choose TERMINAL or MASTER (GATEWAY) node
#define MODE 2  // to choose sender (1), receiver (2) , publisher (3), subscriber (4) mode
#define CKEY "abcdefghijklmnop"  // AES key - 16 bytes
#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include "LoRa_Packets.AES.h"
#include "LoRa_onReceive.AES.h"  // to capture the packets with MODE and server/broker
int tss_flag=0;

void TS_Task( void * pvParameters ){
char kbuff[32];
TSframe_t rdf; int TSdelay=1000; int x,r;
while(true)
  {
  xQueueReceive(rcv_queue, &rdf, portMAX_DELAY);
  Serial.println(rdf.pack.channel);Serial.println(rdf.pack.keyword);
  LoRa.idle();
  stab[0]= 10.2+random(10,30);
  if(rdf.pack.con[1] & 0x80) { stab[0]=10.2+random(10,10);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x40) { stab[1]=10.2+random(10,20);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x20) { stab[2]=10.2+random(10,30);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x10) { stab[3]=10.2+random(10,40);delay(TSdelay); }
```

```
  if(rdf.pack.con[1] & 0x08) { stab[4]=10.2+random(10,50);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x04) { stab[5]=10.2+random(10,60);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x02) { stab[6]=10.2+random(10,70);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x01) { stab[7]=10.2+random(10,80);delay(TSdelay); }
  r=random(1,100);
  if(r>50) x=200; else x=100;
  if(x == 200) tss_flag=1;
  else tss_flag=2;
  LoRa_rxMode();
  }
}

void setup() {
  Serial.begin(9600); delay(100);
  gwID=(uint32_t)ESP.getEfuseMac();
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  con_queue = xQueueCreate(queueSize, 32);
  rcv_queue = xQueueCreate(queueSize, 64);
  xTaskCreatePinnedToCore(
                    TS_Task,   /* Function to implement the task */
                    "TS_Task", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore);  /* Core where the task should run */
  Serial.println("Task created...");
  stage1_flag=1;
}

void loop() {
 if(stage1_flag==1)
  {
  conframe_t rcf; int rec=0;
  rec=xQueueReceive(con_queue, rcf.frame, portMAX_DELAY);
  termID=rcf.pack.sid;
  Serial.printf("recv IDREQ MODE=%x from TERM: %08X, pass=%16.16s\n",MODE, termID,rcf.pack.pass);
  send_IDACK(CKEY,0x0000);
  Serial.printf("IDACK service=%x sent\n",MODE);stage1_flag=0;
  }
  if(stage2_flag==1)
  {
  TSframe_t rdf;
  int tout=10; char mess[16];
  while(tss_flag==0)
    { Serial.println("wating for TS return"); delay(1000);
    }
  tout=10+random(10,20);
  if(tss_flag==1) strcpy(mess,"last read OK   ");
  if(tss_flag==2) strcpy(mess,"last read error");
  send_DTRCV(rdf.pack.con[1],tout,mess,stab);  tss_flag=0;
  Serial.printf("DTRCV to TERM:%08X;tout=%d,stab[0]=%2.2f,mess=%16.16s\n",termID,tout,stab[0],mess);
  stage2_flag=0;
  }
}
```

10.5.3 Nœuds de terminal et de passerelle – MODE 3 (MQP)

Le MODE 3 est prêt à envoyer les paquets de données au courtier (*broker*) **MQTT**.
A l'étape 1 le nœud Terminal commence par établir la liaison logique avec la passerelle, puis il envoie les paquets **DTPUB** avec le sujet (topic) et les données de message.

A l'étape 2, après la réception du paquet **DTPUB**, le nœud passerelle envoie le paquet **DTPUBACK**. Ce dernier paquet informe le Terminal concerné que les nouveaux message a était publié.

10.5.3.1 Code complet du nœud Terminal en MODE 3 (MQP)

Ce qui suit est le code du nœud terminal pour MODE 3 (**MQ**TT Publish - MQ**P**)

```
#define TERMINAL  // to choose TERMINAL or MASTER (GATEWAY) node
```

```
#define MODE  3  // to choose sender (1), receiver (2) , publisher (3), subscriber (4) mode

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

#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include "LoRa_Packets.AES.h"
#include "LoRa_onReceive.AES.h"  // to capture the packets with MODE and server/broker

char topic[48];   // this variable may be loaded from an EEPROM (external-internal)
char message[48];  // this variable to be loaded with sensor message

void get_sens()
{
    SHT21.begin();
    stab[0]=SHT21.getTemperature();
    delay(100);
    stab[1]=SHT21.getHumidity();
    Serial.printf("T:%2.2f, H:%2.2f\n",stab[0],stab[1]);
}

void setup() {
  Serial.begin(9600); delay(100);
   Wire.begin(12,14);
  termID=(uint32_t)ESP.getEfuseMac(); delay(100);
  strcpy(topic,"/esp32/my_sensors/");
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  con_queue = xQueueCreate(queueSize, 32);
  mqrcv_queue = xQueueCreate(queueSize, 112);
  stage1_flag=1;stage2_flag=0;
}

int cycle=10;   // 10 seconds

void loop()
{
if(runEvery(cycle*1000))
  {
  if(cycle_cnt>10) { stage1_flag=1; stage2_flag=0;cycle_cnt=0; }
  else { Serial.printf("cycle=%d\n",cycle_cnt); cycle_cnt++; }
  if(stage1_flag==1)  // runs until IDACK sets stage1_flag to 1 - in RTC memory
    {
    conframe_t rcf; int rec=0;
    send_IDREQ("passwordpassword"); // send IDACK for service 1
    Serial.printf("sent IDREQ MODE=%x sent from TERM: %08X\n",MODE,(uint32_t)termID);
    rec=xQueueReceive(con_queue, rcf.frame, 100); // 100 ms
    if(rec)
      {
      Serial.printf("recv IDACK service=%x got from GW: %08X ckey=%16.16s\n",MODE,
                    (uint32_t)gwID,rcf.pack.pass);
      strncpy(CKEY,rcf.pack.pass,16);
      stage1_flag=0; stage2_flag=1;  // transition from stage 1 to stage2
      }
    }
  if(stage2_flag==1) // runs until DTPUBACK sets stage2_flag to 1
    {
    MQTTframe_t rdf;
    get_sens();
    sprintf(message, "T:%2.2f, H:%2.2f",stab[0],stab[1]);
    //strcpy(message, "T:24.67, H:56.98");
    Serial.printf("DTPUB MODE=%x sent from TERM: %08X\n",MODE,(uint32_t)termID);
    send_DTPUB(topic,message);
    xQueueReceive(mqrcv_queue, rdf.frame, 600);
    Serial.printf("recv DTPUBACK MODE=%x from GW:%08X,cycle=%dsec,topic=%s\n",MODE,
                  (uint32_t)rdf.pack.sid, rdf.pack.tout,rdf.pack.topic);
    }
  }
}
```

10.5.3.1 Nœud de passerelle – MODE 3 (MQP)

Ce qui suit est le code du nœud de passerelle pour MODE 3 (MQTT Publish - MQP).

```
#define GATEWAY  // to choose TERMINAL or MASTER (GATEWAY) node
#define MODE 3  // to choose sender (1), receiver (2) , publisher (3), subscriber (4) mode
#define CKEY "abcdefghijklmnop"  // AES key - 16 bytes
#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include "LoRa_Packets.AES.h"
#include "LoRa_onReceive.AES.h"  // to capture the packets with MODE and server/broker
int mqpub_flag=0;  // MQTT publish flag

void MQTT_Task( void * pvParameters ){
char kbuff[32];
MQTTframe_t rdf;
while(true)
  {
  xQueueReceive(mqrcv_queue, &rdf.pack, portMAX_DELAY);
  LoRa.idle();
  Serial.printf("Publish topic:%s\n",rdf.pack.topic);
  Serial.printf("Publish message:%s\n",rdf.pack.mess);
  delay(2000);mqpub_flag=1;
  LoRa_rxMode();
  }
}

void setup() {
  Serial.begin(9600); delay(100);
  gwID=(uint32_t)ESP.getEfuseMac();
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  con_queue = xQueueCreate(queueSize, 32);
  mqrcv_queue = xQueueCreate(queueSize, 112);
  xTaskCreatePinnedToCore(
                    MQTT_Task,   /* Function to implement the task */
                    "MQTT_Task", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore);  /* Core where the task should run */
  Serial.println("Task created...");
  stage1_flag=1;
}

void loop() {
 if(stage1_flag==1)
  {
  conframe_t rcf; int rec=0;
  rec=xQueueReceive(con_queue, rcf.frame, portMAX_DELAY);
  termID=rcf.pack.sid;
  Serial.printf("recv IDREQ MODE=%x from TERM: %08X, pass=%16.16s\n",MODE, termID,rcf.pack.pass);
  send_IDACK(CKEY,0x0000); // send IDACK for service 1 with CKEY
  Serial.printf("IDACK service=%x sent\n",MODE);stage1_flag=0;
  }
  if(stage2_flag==1)
  {
  MQTTframe_t rdf;
  int tout=10; char mess[48];char control[24];
  strcpy(mess,"message"); strcpy(control,"control");
  while(mqpub_flag==0)
    {
    Serial.println("wating for MQTT return");
    delay(1000);
    }
  tout=10+random(10,20);
  send_DTPUBACK(control,mess,tout);  mqpub_flag=0;
  Serial.printf("DTPUBACK to TERMINAL: %08X ; timeout=%d\n",termID,tout);
  stage2_flag=0;
  }
}
```

10.5.4 Nœuds de terminal et de passerelle – MODE 4 (MQS)

Le MODE 4 est conçu pour envoyer la demande de données via la commande d'abonnement au sujet (topic) du courtier MQTT.
Dans notre cas, le nœud Terminal, après la réception de l'identifiant de la passerelle (**gwID**) et de la clé AES, envoie le paquet **DTSUB**. Ce paquet contient le sujet à souscrire.
Le nœud passerelle fait appel à cette demande au courtier MQTT et attend les nouvelles données publiées sur le sujet spécifié. Ces données sont ensuite envoyées au nœud Terminal avec le paquet **DTSUBACK**.

Dans notre cas, nous générons les données localement dans le nœud de passerelle.

10.5.4.1 Code complet du nœud Terminal en MODE 4

Voici le code du Terminal fonctionnant en mode 4 avec abonnement au sujet donné.

```
#define TERMINAL  // to choose TERMINAL or MASTER (GATEWAY) node
#define MODE 4  // to choose sender (1), receiver (2) , publisher (3), subscriber (4) mode

#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include "LoRa_Packets.AES.h"
#include "LoRa_onReceive.AES.h"
#include <Wire.h>
#include "SSD1306Wire.h"
SSD1306Wire display(0x3c, 12, 14);

void disp_sens(char *topic, char *mess)
{
  char buff[32];
  display.init();
  display.flipScreenVertically();
  display.setFont(ArialMT_Plain_10);
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.drawString(0,0,"Terminal MQSUB");  // first 16 lines are yellow
  display.drawString(0,16,topic);
  display.drawString(0,28,mess);
  display.display();
}

void setup() {
  Serial.begin(9600); delay(100);
  Wire.begin(12,14);
  termID=(uint32_t)ESP.getEfuseMac(); delay(100);
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  con_queue = xQueueCreate(queueSize, 32);
  mqrcv_queue = xQueueCreate(queueSize, 112);
  stage1_flag=1;stage2_flag=0;
}

int cycle=20;   // 20 seconds
void loop()
{
if(runEvery(cycle*1000))
  {
  if(cycle_cnt>10) { stage1_flag=1; stage2_flag=0;cycle_cnt=0; }
  else { Serial.printf("cycle=%d\n",cycle_cnt); cycle_cnt++; }
  if(stage1_flag==1)  // runs until IDACK sets stage1_flag to 1 - in RTC memory
    {
    conframe_t rcf; int rec=0;
    send_IDREQ("passwordpassword"); // send IDACK for service 1
    Serial.printf("sent IDREQ MODE=%x sent from TERM: %08X\n",MODE,(uint32_t)termID);
    rec=xQueueReceive(con_queue, rcf.frame, 10000);
    if(rec)
      {
      Serial.printf("recv IDACK MODE=%x got from GW: %08X ckey=%16.16s\n",MODE,(uint32_t)gwID,
                    rcf.pack.pass);
      strncpy(CKEY,rcf.pack.pass,16);
      stage1_flag=0; stage2_flag=1;  // transition from stage 1 to stage2
      }
    }
```

```
  if(stage2_flag==1) // runs until DTPUBACK sets stage2_flag to 1
    {
    MQTTframe_t rdf; int rec=0;
    send_DTSUB("/esp32/my_sensors/");
    Serial.printf("DTSUB MODE=%x sent from TERM: %08X\n",MODE,(uint32_t)termID);
    if(xQueueReceive(mqrcv_queue, rdf.frame, cycle*1000)== pdTRUE)
      {
      Serial.printf("\nDTSUBRCV: %s:%s, %d\n",rdf.pack.topic,rdf.pack.mess,rdf.pack.tout);
      disp_sens(rdf.pack.topic, rdf.pack.mess);
      if(rdf.pack.tout>10 && rdf.pack.tout <3600) cycle=rdf.pack.tout; else cycle=20;
      }
    else Serial.println("rcv_queue timeout");
     }
   }
}
```

10.5.4.2 Code complet du nœud passerelle en MODE 4

Voici le code du nœud de passerelle fonctionnant en mode 4 avec abonnement au sujet donné.

```
#define GATEWAY  // to choose TERMINAL or MASTER (GATEWAY) node
#define MODE 4  // to choose sender (1), receiver (2) , publisher (3), subscriber (4) mode
#define CKEY "abcdefghijklmnop"  // AES key - 16 bytes
#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include "LoRa_Packets.AES.h"
#include "LoRa_onReceive.AES.h"  // to capture the packets with MODE and server/broker

int mqsub_flag=0;  // MQTT subscribe get message flag
QueueHandle_t mqtt_queue;   // receive queue to get out the data packets form onReceive() ISR

typedef struct
{
  char topic[48];
  char mess[48];
} SUB_t;  // structure for topic and message

void SUB_Task( void * pvParameters) {
SUB_t mqtt_sub;
while(true)
  {
  delay(4000);
  strcpy(mqtt_sub.topic,"incoming topic");
  strcpy(mqtt_sub.mess,"incoming message");
  xQueueReset(mqtt_queue);  // reset queue to keep only the last packet
  xQueueSend(mqtt_queue, &mqtt_sub, 100000);
  mqsub_flag=1;
  }
}

void MQTT_Task( void * pvParameters ){  // subscribe task
char kbuff[32];
MQTTframe_t rdf;
while(true)
  {
  xQueueReceive(mqrcv_queue, &rdf.pack, portMAX_DELAY);
  LoRa.idle();
  Serial.printf("%s\n",rdf.pack.topic);
  Serial.println("topic subscribed");
  delay(100);
  LoRa_rxMode();
  }
}

void setup() {
  Serial.begin(9600); delay(100);
  gwID=(uint32_t)ESP.getEfuseMac();
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  con_queue = xQueueCreate(queueSize, 32);
  mqtt_queue = xQueueCreate(queueSize, 96);
  mqrcv_queue = xQueueCreate(queueSize, 112);
```

```
  xTaskCreatePinnedToCore(
                    MQTT_Task,   /* Function to implement the task */
                    "MQTT_Task", /* Name of the task */
                    10000,       /* Stack size in words */
                    NULL,        /* Task input parameter */
                    0,           /* Priority of the task */
                    NULL,        /* Task handle. */
                    1);  /* Core where the task should run */
  Serial.println("Task created...");

  xTaskCreatePinnedToCore(
                    SUB_Task,   /* Function to implement the task */
                    "SUB_Task", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    1);  /* Core where the task should run */
  Serial.println("Task created...");
  stage1_flag=1; stage2_flag=0;
}


void loop()
{
int tout=10;
if(stage1_flag==1)
  {
  conframe_t rcf; int rec=0;
  rec=xQueueReceive(con_queue, rcf.frame, portMAX_DELAY);
  termID=rcf.pack.sid;
  Serial.printf("recv IDREQ MODE=%x from TERM: %08X, pass=%16.16s\n",MODE, termID,rcf.pack.pass);
  send_IDACK(CKEY,0x0000); // send IDACK for service 1 with CKEY
  Serial.printf("IDACK service=%x sent\n",MODE);stage1_flag=0;stage2_flag=1;
  }
  if(stage2_flag==1)
  {
  SUB_t mqtt_sub; int rec=0;
  if(xQueueReceive(mqtt_queue, &mqtt_sub, 80000)== pdTRUE)
    {
    tout=10+random(10,30);
    send_DTSUBRCV(mqtt_sub.topic,mqtt_sub.mess,tout);
    Serial.printf("DTSUBRCV to TERMINAL: topic:%s, message:%s\n",mqtt_sub.topic,mqtt_sub.mess);
    }
  else  Serial.println("mqtt_sub queue timeout");
  stage2_flag=0;
  }
}
```

10.6 A faire

1. Testez les 4 modes de services
2. Utilisez
   `set_LoRa_Radio_Para(unsigned long freq,unsigned sbw,int sf, uint8_t sw)`
   pour modifier les paramètres de la radio
3. Au nœud Terminal, ajoutez une fonction ou tâche de capteur pour fournir les données pour les modes 1 et 3 (**`DTSND`**, **`DTPUB`**)
4. Ajouter un écran OLED dans le nœud de passerelle pour afficher les données des paquets **`DTSND`** et **`DTPUB`**

Lab 11 - Protocole et passerelles LoRa TS (ThingSpeak)

11.1 Introduction

Dans ce laboratoire, nous allons créer une double (émetteur-récepteur) passerelle IoT pour la communication LoRa avec les serveurs de type ThingSpeak (TS). Les passerelles fournissent le relais entre les liaisons LoRa et les modems WiFi vers/depuis le serveur (ThingSpeak).
Une architecture IoT correspondante comprend deux cartes ESP32 fonctionnant comme des passerelles LoRa-TS. Une carte passerelle relaie les messages des nœuds terminaux vers le serveur TS et la seconde reçoit les données demandés de TS et les relaie vers les nœuds terminaux.

Les nœuds terminaux fonctionnant en tant qu'expéditeur et récepteur de données ont déjà été présentés en détail dans le laboratoire précédent.
Dans ce laboratoire, nous allons compléter les fonctionnalités des passerelles en émission et en réception des données vers/à partir du serveur ThingSpeak. Plus précisément, nous allons ajouter la partie *back-end* communiquant par **WiFi** avec le serveur ThingSpeak.

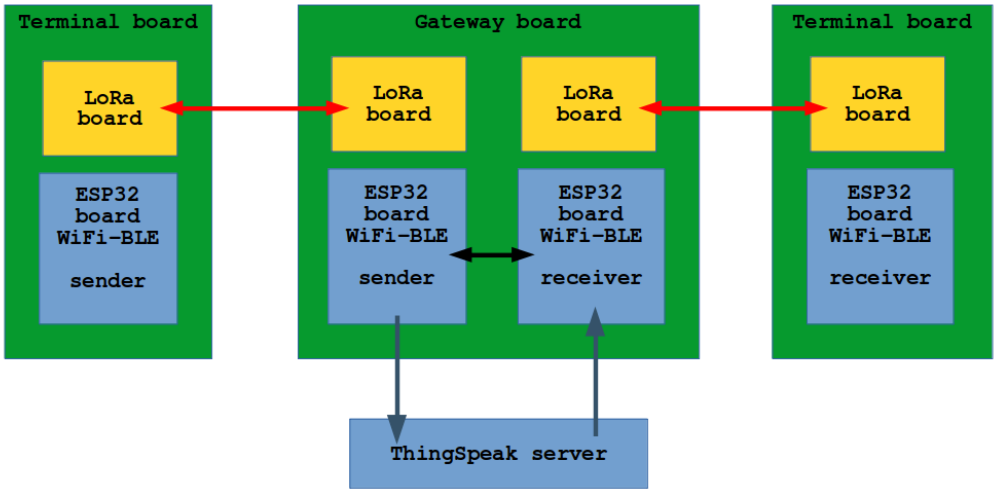


Figure 11.1 Architecture IoT avec deux passerelles vers le serveur ThingSpeak (envoi-réception) et les nœuds terminaux

Dans le Lab.10 précédent, nous avons présenté les nœuds Terminal et Gateway communiquant avec les paquets de contrôle et de données prédéfinis. Dans ce laboratoire, nous utilisons les mêmes bibliothèques et les mêmes fonctions **`send_XXXX()`** et **`onReceive()`**.
La principale différence est que nous complétons nos passerelles avec les tâches de communication basées sur le **WiFi**. Ces tâches nous permettent d'envoyer/recevoir les données vers/depuis le serveur ThingSpeak.

Comme dans les configurations présentées dans le laboratoire précédant, les terminaux et les passerelles fonctionnent en deux étapes.

- la première étape permet aux terminaux de se présenter à la passerelle via un paquet **`IDREQ`** (**`IDREQ`** porte le mot de passe) et de recevoir l'identifiant de la passerelle (**`gwID`**) et la clé AES (**`CKEY`**) via un paquet **`IDACK`**. Cette étape est réactivée périodiquement pour tester la disponibilité de la passerelle.

- la deuxième étape est utilisée pour envoyer les données avec un paquet **`DTSND`** ou pour demander les données avec un paquet **`DTREQ`**. Les paquets répondantes sont les accusés de réception portant la valeur de temporisation (**`DTACK`**) ainsi que les données pour **`DTREQ`** - **`DTRCV`**. La valeur de temporisation peut être utilisée par le nœud terminal pour imposer la période d'attente avant d'envoyer les trames **`DTSND`** ou **`DTREQ`** suivantes.

Comme nous l'avons déjà vu dans le laboratoire précédent, la première étape opérationnelle avec les paquets **`IDREQ/IDACK`** permet de configurer le fonctionnement au niveau de la couche **liaison/réseau** LoRa.

Cette couche **liaison/réseau** est la même pour tous les protocoles applicatifs, y compris: envoyer vers TS (MODE 1), recevoir depuis TS (MODE 2), publier vers MQTT (MODE 3) et souscrire à MQTT (MODE 4).

Le tableau suivant montre les codes utilisés pour identifier les paquets dans le champ **con[0]** des paquets de contrôle et de données.

```
1 - TS send, 2 - TS receive, 3 - MQTT
PUBlish, 4 - MQTT Suscribe

Link
con[0]=11:IDREQ(1),con[0]=12:IDACK(1)
con[0]=21:IDREQ(2),con[0]=22:IDACK(1)
con[0]=31:IDREQ(3),con[0]=32:IDACK(1)
con[0]=41:IDREQ(4),con[0]=42:IDACK(1)

ThingSpeak
con[0]=13:DTSND(1)
con[0]=14:DTACK(1)
con[0]=23:DTREQ(2)
con[0]=24:DTRCV(2)

MQTT
con[0]=33:DTPUB(3)
con[0]=34:DTACK(3)
con[0]=43:DTSUB(4)
con[0]=44:DTRCV(4)
```

Tableau 11.1 Les codes pour les **types** et **modes** des paquets LoRa

Le deuxième champ de contrôle **con[1]** est utilisé pour marquer les champs ou identifiants des capteurs/actionneurs pour les canaux TS.
L'union/la structure suivante définit les paquets de contrôle. Elle commence par les identificateurs de destination/source (**did**, **sid**) dérivés du **chipID** de chaque nœud.

```
typedef union
{
  uint8_t frame[32];
  struct
  {
    uint32_t did;       // destination identifier chipID (4 lower bytes)
    uint32_t sid;       // source identifier chipID (4 lower bytes)
    uint8_t  con[2];    // control field: con[0]
    char     pass[16];  // password or AES key - 16 characters
    int tout;           // timeout
    uint8_t  pad[2];    // future use
  } pack;               // control packet
} conframe_t;              // send control frame , receive control frame
```

Etape 1

Le paquet **IDACK** qui répond au **IDREQ** contient l'identifiant de destination - **did**, c'est-à-dire l'identifiant du terminal qui a envoyé la trame **IDREQ**. Le champ **sid** contient la partie inférieure (4 octets) de l'identifiant de la passerelle (**gwID**).

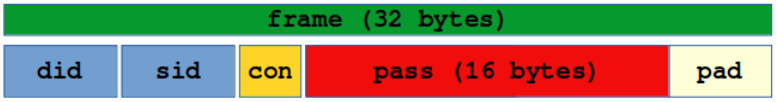


Figure 11.2 Format des paquets **IDREQ**, **IDACK**

Ce sont les paramètres du paquet de contrôle **IDREQ**.

```
scf.pack.did=(uint32_t)0;     // destination is not known !!!
scf.pack.sid=(uint32_t)termID;
scf.pack.con[0]=0x11; scf.pack.con[1]=0x00;  // IDREQ frame for TS sender
```

Notez qu'en fonction des services requis, le champ de contrôle contient le MODE de service :

```
scf.pack.con[0]=0x11;scf.pack.con[1]=0x00; // IDREQ packet for TS sender (TSS)
scf.pack.con[0]=0x21;scf.pack.con[1]=0x00; // IDREQ packet for TS receiver (TSR)
```

Après l'envoi du paquet **IDREQ**, le terminal attend qu'un paquet **IDACK** soit générée par la passerelle.

Notez que la passerelle n'enverra le paquet **IDACK** correspondant que si le champ mot de passe - **pass[16]** fourni par le terminal correspond à la valeur du mot de passe enregistré dans la passerelle et au service demandé (émetteur/récepteur).

```
  scf.pack.did=(uint32_t)termID;  // destination is the requesting terminal
  scf.pack.sid=(uint32_t)gwD;     // gateway identifier
  scf.pack.con[0]=0x12; scf.pack.con[1]=0x00;  // IDACK - MODE 1
```

Etape 2

Après la réception de l'identifiant de passerelle, le terminal peut envoyer son paquet de données – **DTSND** ou demander les données via un paquet **DTREQ**.

Les données sont envoyées dans le type d'union/structure suivante :

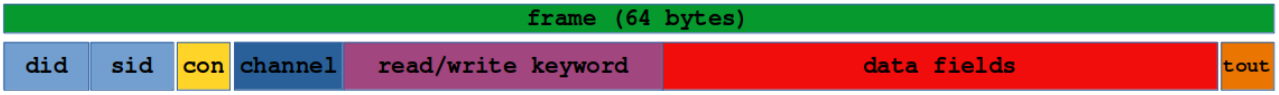


Figure 11.3 Format de paquet de données (**DTSND**, **DTACK**)

```
typedef union
{
  uint8_t frame[64];        // TS frame to send/receive data
  struct
  {
    uint32_t did;           // destination identifier chipID (4 lower bytes)
    uint32_t sid;           // source identifier chipID (4 lower bytes)
    uint8_t  con[2];        // control field: lower byte is used as mask
    int      channel;       // TS channel number
    char     keyword[16];   // write (or read) keyword
    float    sens[8];       // max 8 values - data fields
    uint16_t tout;          // optional timeout
  } pack;                   // data packet
} TSframe_t;
```

La valeur de masque envoyée dans l'octet **con[1]** est un masque qui indique les valeurs de capteur valides. A chaque valeur de capteur **sens[8]** correspond un **bit** dans le **masque** de capteur/actionneur.
Par exemple, si le terminal envoie deux valeurs Température et Humidité dans les deux premiers champs, la valeur du masque doit être **0xC0** (en binaire: **11000000**)

```
  sdf.pack.did=(uint32_t)gwID;
  sdf.pack.sid=(uint32_t)termID;
  sdf.pack.con[0]=0x13; sdf.pack.con[1]=mask;  // DTSND packet - MODE 1
```

Ensuite, le terminal attend un paquet **DTACK**.

```
  sdf.pack.did=(uint32_t)termID;
  sdf.pack.sid=(uint32_t)sendID;
  sdf.pack.con[0]=0x14; sdf.pack.con[1]=mask;  // DTACK packet - MODE 1
```

Ce paquet peut contenir une information supplémentaire qui porte la valeur de temporisation pour la trame de données suivante. Elle sera enregistré dans le champ de numéro du canal (**channel**).
Voici la séquence de code requise pour initialiser l'état **deep_sleep** dans le nœud terminal :

```
      esp_sleep_enable_timer_wakeup(1000*1000*timeout);
      esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH, ESP_PD_OPTION_OFF);
      LoRa.end(); delay(200);  // necessary to stop the LoRa modem
```

```
      esp_deep_sleep_start();
```

La valeur de temporisation reçue est convertie en le nombre de microsecondes pendant lesquelles le circuit ESP32 passe en état de sommeil profond. Notez que pendant cette période, toutes les variables volatiles liées à deux processeurs principaux sont perdues.
Si nous devons les conserver, ils peuvent être stockés dans la RAM statique intégrée dans le bloc RTC ou dans la mémoire EEPROM locale.

Notez que les valeurs stockées dans le bloc RTC sont effacées si nous réinitialisons le SoC ESP32 avec le bouton **RST**.
Côté réception des paquets, **`onReceive()`** est une fonction définie dans le fichier **`LoRa_onReceive.AES.h`** et utilisée pour capturer les interruptions IO du modem LoRa et pour recevoir les paquets de contrôle et des données envoyées par le terminal et les nœuds de passerelle (réception **`GATEWAY`**: **`IDREQ`**, **`DTSND`**; réception **`TERMINAL`**:**`IDACK`**, **`DTRCV`**).

Les indicateurs de réception **`stage1_flag`** et **`stage2_flag`** indiquent le type du paquet reçu.
Dans le même fichier (**`LoRa_onReceive.AES.h`**), nous déclarons la file d'attente de réception utilisée pour communiquer le contenu des paquets reçus à d'autres tâches.

Notez que **`onReceive()`** est une fonction **ISR** exécutée de manière asynchrone et ne doit pas contenir d'opérations de traitement supplémentaires. La fonction **`decrypt()`** est ici exécutée par l'accélérateur matériel interne et peut être incluse dans les opérations de l'ISR.

Voici le contenu du fichier **`LoRa_onReceive.AES.h`** (le même que celui utilisé dans le Lab.10)

```
//   0x11 – IDREQ, 0x12 – IDACK : ID request and ID acknowledge – TS send – TSS
//   0x21 – IDREQ, 0x22 – IDACK : ID request and ID acknowledge – TS receive –TSR
//   0x31 – IDREQ, 0x32 – IDACK : ID request and ID acknowledge – MQTT publish – MQP
//   0x41 – IDREQ, 0x42 – IDACK : ID request and ID acknowledge – MQTT subscribe – MQS
//   0x13 – DTSND, 0x14 – DTACK : DATA send and DATA acknowledge – TS send  – TSS
//   0x23 – DTREQ, 0x24 – DTRCV : DATA request and DATA receive – TS receive – TSR
//   0x33 – DTPUB, 0x34 – DTPUB ACK : DATA publish and DATA publish acknowledge – MQTT publish – MQP
//   0x43 – DTSUB, 0x44 – DTSUB RCV DATA subscribe and DATA subscribe receive – MQTT subscribe – MQS

QueueHandle_t tsrcv_queue, mqrcv_queue, con_queue;
int queueSize = 32;
static int taskCore = 0;
uint8_t mask; int channel; float stab[8];
uint8_t IDREQ_mode,IDACK_mode;
uint8_t DT_mode,ACK_mode;  // data packet receive GW and data packet receive TERM
char *password="passwordpassword";  // default password

void onReceive(int packetSize)
{
if(packetSize==32)
  {
  conframe_t rcf; int i=0;
  while (LoRa.available()) { rcf.frame[i] = LoRa.read();i++;}
#ifdef GATEWAY
  IDREQ_mode=MODE*16+1;
  if(rcf.pack.con[0]==IDREQ_mode && rcf.pack.did==0x00 && !strncmp(password,rcf.pack.pass,16))  //
GW:received IDREQ_MODE packet
    { stage1_flag=1;
      // termID=rcf.pack.sid;
      xQueueReset(con_queue);  // reset queue to keep only the last packet
      xQueueSendFromISR(con_queue, rcf.frame, NULL); Serial.println("Received IDREQ_MODE");
    }
#endif
#ifdef TERMINAL
  IDACK_mode=MODE*16+2;
  if(rcf.pack.con[0]==IDACK_mode && rcf.pack.did==termID )  // TERM:received IDACK_MODE packet
    { stage1_flag=1;
      gwID=rcf.pack.sid; Serial.println("Received IDACK_MODE");
      xQueueReset(con_queue);  // reset queue to keep only the last packet
      xQueueSendFromISR(con_queue, rcf.frame, NULL); Serial.println("Received IDACK_MODE");
    }
#endif
  }
#if MODE<3
if(packetSize==64)
  {
```

```
  TSframe_t rdf,rdcf; int i=0;
  char ckey[17];
  strncpy(ckey,CKEY,16);ckey[16]='\0';
  while (LoRa.available()) { rdcf.frame[i] = LoRa.read();i++;}
  decrypt(rdcf.frame,ckey,rdf.frame,4);
  #ifdef GATEWAY
  DT_mode=MODE*16+3;
  if(rdf.pack.con[0]==DT_mode && rdf.pack.did==gwID)  // GW:received data packet
    { stage2_flag=1; termID=rdf.pack.sid;  Serial.println("GW:Received TS data_MODE");
      xQueueReset(strcv_queue);  // reset queue to keep only the last packet
      xQueueSend(strcv_queue, rdf.frame, portMAX_DELAY);
    }
  #endif
  #ifdef TERMINAL
  ACK_mode=MODE*16+4;
  if(rdf.pack.con[0]==ACK_mode && rdf.pack.did==termID)  // TERM:received data packet
    { stage2_flag=1;
      gwID=rdf.pack.sid; Serial.println("TERM:Received TS data_MODE");
      xQueueReset(strcv_queue);  // reset queue to keep only the last packet
      xQueueSend(strcv_queue, rdf.frame, portMAX_DELAY);
    }
  #endif
  }
#endif
#if MODE>2
if(packetSize==112)
  {
  MQTTframe_t rdf,rdcf; int i=0;
  char ckey[17];
  strncpy(ckey,CKEY,16);ckey[16]='\0';
  while (LoRa.available()) { rdcf.frame[i] = LoRa.read();i++;}
  decrypt(rdcf.frame,ckey,rdf.frame,7);
  #ifdef GATEWAY
  DT_mode=MODE*16+3;
  if(rdf.pack.con[0]==DT_mode && rdf.pack.did==gwID)  // GW:received data packet
    { stage2_flag=1; termID=rdf.pack.sid;  Serial.println("GW:Received MQ data_MODE");
      xQueueReset(mqrcv_queue);  // reset queue to keep only the last packet
      xQueueSend(mqrcv_queue, rdf.frame, portMAX_DELAY);
    }
  #endif
  #ifdef TERMINAL
  ACK_mode=MODE*16+4;
  if(rdf.pack.con[0]==ACK_mode && rdf.pack.did==termID)  // TERM:received data packet
    { stage2_flag=1;
      gwID=rdf.pack.sid; Serial.println("TERM:Received MQ data_MODE");
      xQueueReset(mqrcv_queue);  // reset queue to keep only the last packet
      xQueueSend(mqrcv_queue, rdf.frame, portMAX_DELAY);
    }
  #endif
  }
#endif
}
```

11.2 Terminal émetteur (*sender*) - MODE 1

Dans cette section, nous présentons deux versions du terminal émetteur, l'une fonctionnant toujours en mode actif, la seconde fonctionne avec la fonction de sommeil profond (mode **deep_sleep**) et le **timeout** fourni par le nœud de passerelle.

11.2.1 Code du terminal émetteur – toujours actif et avec état `deep_sleep`

Le code suivant implémente un terminal qui envoie les données à la passerelle pour être relayées vers le serveur ThingSpeak.
La partie du code commenté suivant peut être utilisée pour mettre le mode terminal en mode sommeil profond après chaque itération (cycle).

```
esp_sleep_enable_timer_wakeup(1000*1000*cycle);  // cycle in seconds
esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH, ESP_PD_OPTION_OFF);
Serial.println("Going to sleep now");
Serial.flush(); LoRa.end();delay(160);
esp_deep_sleep_start();
Serial.println("This will never be printed");
```

11.2.1.1 Code complet du terminal avec SHT21 - capteur de température/humidité

```
#define TERMINAL  // to choose TERMINAL or MASTER (GATEWAY) node
#define MODE 1     // to choose sender (1), receiver (2) , publisher (3), subscriber (4) mode
#include <Wire.h>
#include "SHT21.h"
SHT21 SHT21;
#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include "LoRa_Packets.AES.h"
#include "LoRa_onReceive.AES.h"  // to capture the packets with MODE and server/broke

union
{
  uint8_t para[24];
  struct
  {
    int chan;      // channel number
    char key[16];  // write key
    uint8_t zero;
    uint8_t mask;  // sensor mask
    uint8_t pad[2];
  } pack;
} ts;  // TS send and receive para

void get_sens()
{
    SHT21.begin();
    stab[0]=SHT21.getTemperature();
    delay(100);
    stab[1]=SHT21.getHumidity();
    Serial.printf("T:%2.2f, H:%2.2f\n",stab[0],stab[1]);
}

void setup() {
  Serial.begin(9600); delay(100);
  Wire.begin(12,14);
  termID=(uint32_t)ESP.getEfuseMac(); delay(100);
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  con_queue = xQueueCreate(queueSize, 32);
  rcv_queue = xQueueCreate(queueSize, 64);
  stage1_flag=1;stage2_flag=0;
  ts.pack.chan=1243348; strncpy(ts.pack.key,"J4K8ZIWAWE8JBIX7",16);
  ts.pack.zero=0x00;ts.pack.mask=0xC0;
  ts.pack.pad[0]=0x00;ts.pack.pad[1]=0x00;
  // the above parameters may be read from external EEPROM to ts union-structure (24-bytes)
}
int cycle=10;   // 10 seconds

void loop()
{
if(runEvery(cycle*1000))
  {
  if(cycle_cnt>10) { stage1_flag=1; stage2_flag=0;cycle_cnt=0; }
  else { Serial.printf("cycle=%d\n",cycle_cnt); cycle_cnt++; }

  if(stage1_flag==1)  // runs until IDACK sets stage1_flag to 0 - in RTC memory
    {
    conframe_t rcf; int rec=0;
    send_IDREQ("passwordpassword"); // send IDACK MODE 1
    Serial.printf("sent IDREQ MODE=%x sent from TERM: %08X\n",MODE,(uint32_t)termID);
    rec=xQueueReceive(con_queue, rcf.frame, 100); // portMAX_DELAY);  // delay in number of tics
    if(rec)
      {
        Serial.printf("recv IDACK service=%x got from GW: %08X ckey=%s\n",MODE,
                      (uint32_t)gwID,rcf.pack.pass);
        strncpy(CKEY,rcf.pack.pass,16);
        stage1_flag=0; stage2_flag=1;  // transition from stage 1 to stage2
      }

    }
```

```
  if(stage2_flag==1)
    {
    TSframe_t rdf;
    Serial.println("sensor");
    get_sens();  // get stab[] values
    Serial.printf("sent DTSND MODE=%x sent from TERM: %08X\n",MODE,(uint32_t)termID);
    send_DTSND(ts.pack.mask,ts.pack.chan,ts.pack.key,stab);
    xQueueReceive(rcv_queue, rdf.frame, portMAX_DELAY);
    Serial.printf("recv DTACK MODE=%x from GW:%08X, cycle=%dsec, mess=%s\n",MODE,
                  (uint32_t)rdf.pack.sid,rdf.pack.channel,rdf.pack.keyword) ;

    // cycle value in rdf.pack.channel may be used to change the cycle or timeout for deep_sleep
    //        esp_sleep_enable_timer_wakeup(1000*1000*cycle);  // cycle in seconds
    //        esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH, ESP_PD_OPTION_OFF);
    //        Serial.println("Going to sleep now");
    //        Serial.flush(); LoRa.end();delay(160);
    //        esp_deep_sleep_start();
    //        Serial.println("This will never be printed");
    }
  }
}
```

11.2.1.2 Le code du terminal avec BH1750 - capteur de luminosité

La différence essentielle entre le code précédent et le code du terminal avec capteur de luminosité BH1750 réside dans le choix et l'utilisation du capteur :

```
#include <BH1750.h>
BH1750 lightMeter;
..

void get_sens()  // the sensor function
{
    lightMeter.begin();
    stab[2]=lightMeter.readLightLevel();
    delay(100);
    Serial.printf("L:%2.2f\n",stab[2]);
}



void setup() {
..
  ts.pack.zero=0x00;ts.pack.mask=0x20;  // luminosity sensor as 3rd sensor
  ts.pack.pad[0]=0x00;ts.pack.pad[1]=0x00;
..
}
```

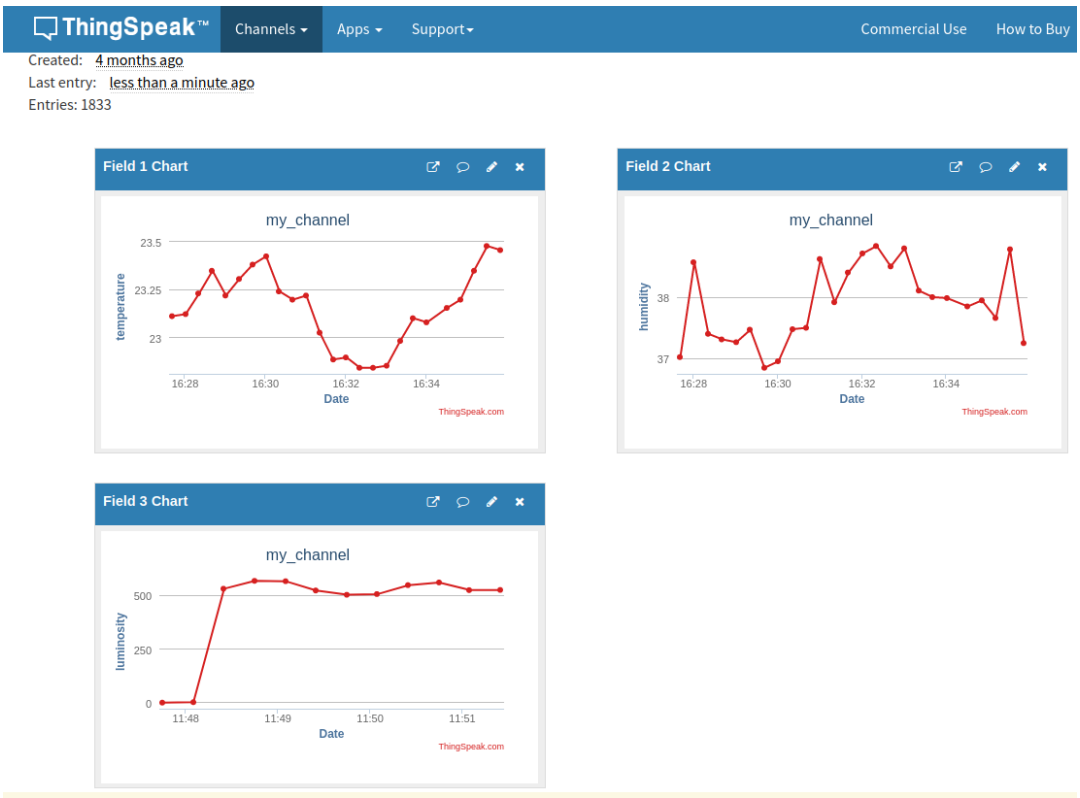


Figure 11.6 Diagramme ThingSpeak avec les données envoyées au même canal à partir de deux cartes séparées

11.3 Passerelle émetteur vers ThingSpeak - MODE 1

La passerelle d'envoi pour ThingSpeak relaie les trames LoRa reçues (**DTSND**) vers le serveur ThingSpeak (par exemple: **ThingSpeak.com**). La passerelle extrait la clé d'écriture, le numéro de canal et décode l'octet de masque - **con[1]** afin de préparer les champs du canal.

Toutes ces opérations sont exécutées séparément dans la tâche **TS_Task** présentée ci-dessous. La tâche attend le paquet de données sur **tsrcv_queue** :

```
xQueueReceive(tsrcv_queue,&rdf,portMAX_DELAY);
```

La file **tsrcv_queue** est écrite par les lignes de code correspondantes:

```
xQueueReset(tsrcv_queue);  // reset queue to keep only the last packet
xQueueSend(tsrcv_queue, rdf.frame, portMAX_DELAY);
```

exécuté dans dans la routine **onReceive()**.

La tâche **TS_Task** prend les données de la **tsrcv_queue** et les envoie au serveur ThingSpeak via une connexion WiFi. Pendant cette période de communication, les terminaux peuvent envoyer leurs paquets LoRa à la passerelle.

Afin de protéger la communication avec le serveur ThingSpeak, **TS_Task** met le modem LoRa en état inactif (**LoRa.idle)**), cet état est maintenu jusqu'à la fin du cycle **TS_Task** où l'on trouve la fonction **LoRa_rxMode()**. Cette fonction désactive le mode **InvertIQ** et met le modem LoRa en état de réception (côté passerelle).

```
void TS_Task( void * pvParameters ){
float stab[8]; char kbuff[32];
TSframe_t rdf;
while(true)
  {
  xQueueReceive(tsrcv_queue, &rdf, portMAX_DELAY);
  Serial.printf("channel:%d, mask=%x, wkey:%16.16s\n",rdf.pack.channel,
                rdf.pack.con[1],rdf.pack.keyword);

  LoRa.idle();
  if(rdf.pack.con[1] & 0x80) ThingSpeak.setField(1, rdf.pack.sens[0]);
  if(rdf.pack.con[1] & 0x40) ThingSpeak.setField(2, rdf.pack.sens[1]);
  if(rdf.pack.con[1] & 0x20) ThingSpeak.setField(3, rdf.pack.sens[2]);
  if(rdf.pack.con[1] & 0x10) ThingSpeak.setField(4, rdf.pack.sens[3]);
  if(rdf.pack.con[1] & 0x08) ThingSpeak.setField(5, rdf.pack.sens[4]);
  if(rdf.pack.con[1] & 0x04) ThingSpeak.setField(6, rdf.pack.sens[5]);
  if(rdf.pack.con[1] & 0x02) ThingSpeak.setField(7, rdf.pack.sens[6]);
  if(rdf.pack.con[1] & 0x01) ThingSpeak.setField(8, rdf.pack.sens[7]);
  memset(kbuff,0x00,32); strncpy(kbuff,rdf.pack.keyword,16);
  // write to the ThingSpeak channel
  int x = ThingSpeak.writeFields((uint32_t)rdf.pack.channel,kbuff);
  if(x == 200)
    { Serial.println("Channel update successful.");tss_flag=1;}
  else
    { Serial.println("Problem updating channel. HTTP error code " + String(x));tss_flag=2;}
  LoRa_rxMode();
  }
}
```

11.3.1 Code complet de la passerelle d'envoi TS - MODE 1

```
#define GATEWAY  // to choose TERMINAL or MASTER (GATEWAY) node
#define MODE  1  // to choose sender (1), receiver (2) , publisher (3), subscriber (4) mode
#define CKEY "abcdefghijklmnop"  // AES key - 16 bytes

#include <WiFi.h>
#include "ThingSpeak.h"
#include <SPI.h>
#include <LoRa.h>
#include <Wire.h>
#include "LoRa_Para.h"
#include "LoRa_Packets.AES.h"
#include "LoRa_onReceive.AES.h"  // to capture the packets with MODE and server/broker
#include "SSD1306Wire.h"        // legacy: #include "SSD1306.h"
SSD1306Wire display(0x3c, 12, 14);   // ADDRESS, SDA, SCL

const char* ssid = "Livebox-08B0";
const char* pass = "G79ji6dtE..VTPWmZP";
WiFiClient client;
int tss_flag=0;

// task code presented above
void TS_Task( void * pvParameters )
{
float stab[8]; char kbuff[32];
..
}

void connect() {
  Serial.print("checking wifi...");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print(".");
    delay(1000);
  }
  Serial.println("\nIoT.GW1 - connected!");
}

void setup() {
  Serial.begin(9600); delay(100);
  gwID=(uint32_t)ESP.getEfuseMac(); // we are in MASTER
  delay(100);

  WiFi.begin(ssid, pass);
  if (!client.connected()) { connect(); }
  Serial.println();Serial.println();
  Serial.println("WiFi connected");

  ThingSpeak.begin(client);  // Initialize ThingSpeak

  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  tsrcv_queue = xQueueCreate(queueSize, 64);
  con_queue = xQueueCreate(queueSize, 32);
  xTaskCreatePinnedToCore(
                    TS_Task,   /* Function to implement the task */
                    "TS_Task", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore);  /* Core where the task should run */
  Serial.println("Task created...");
  stage1_flag=1;
}

void loop() {
char mess[16];
if(stage1_flag==1)
  {
  conframe_t rcf; int rec=0;
  rec=xQueueReceive(con_queue, rcf.frame, portMAX_DELAY);
  termID=rcf.pack.sid;
  Serial.printf("recv IDREQ MODE=%x from TERM: %08X, pass=%16.16s\n",MODE, termID,rcf.pack.pass);
```

```
  send_IDACK(CKEY,0x0000); // send IDACK for service 1 with CKEY
  Serial.printf("IDACK service=%x sent\n",MODE);stage1_flag=0;
  }
  if(stage2_flag==1)
  {
  TSframe_t rdf;  int tout=10; char mess[24];
  strcpy(mess,"control message");
  while(tss_flag==0)
    {
      Serial.println("wating for TS return");
      delay(1000);
    }
  if(tss_flag==1) {Serial.println("got TS return OK");   strcpy(mess,"TS update OK    ");}
  if(tss_flag==2) {Serial.println("got TS return ERROR");strcpy(mess,"TS update    ERR");}
  tss_flag=0; tout=5+random(10,20);
  send_DTACK(rdf.pack.con[1],tout,mess); // send DTACK: mask, channel from DTSND
  Serial.printf("DTACK to TERMINAL: %08X\n",termID);stage2_flag=0;
  }
}
```

11.3.2 Passerelle - planificateur des terminaux

Le paramètre **timeout** à envoyer par la passerelle dans le numéro du canal (**`channel`**) peut être contrôlé par la passerelle afin de planifier le prochain paquet d'envoi depuis le nœud terminal.
L'ordonnanceur peut prendre en compte le nombre de nœuds terminaux enregistrés et décomposer le cycle global en tranches de temps, une tranche par terminal. La durée de chaque créneau doit être plus longue que la durée de la transaction, y compris la communication avec le terminal (LoRa) et le serveur ThingSpeak (WiFi). Avec cette solution, le numéro du terminal correspond au numéro de su créneau.

11.3.3 Paramètres de canal ThingSpeak stockés dans l'EEPROM

Le terminal et les nœuds de passerelle ont besoin d'un certain nombre de paramètres pour communiquer entre eux et entre la passerelle et le serveur ThingSpeak. Par exemple, le terminal a besoin de connaître les paramètres du canal ThingSpeak avant d'envoyer un paquet **`DTSND`**.

Afin de fournir ces paramètres depuis l'extérieur, nous pouvons utiliser l'**EEPROM externe** et **interne**. Tout d'abord, l'EEPROM programmée en externe peut fournir les paramètres requis à la carte qui lit ces paramètres et les stocke dans la mémoire EEPROM interne.
La prochaine fois que le terminal ou la passerelle est activé il utilisera les paramètres stockés en interne pour démarrer les opérations.

Ce qui suit est le code (fonctions) dans le fichier **`EEPROM_TS.h`** qui peut être utilisé pour écrire et lire les paramètres du canal ThingSpeak vers et depuis les mémoires EEPROM externes et internes.

```
// EEPROM_TS.h
#include "EEPROM.h"
#include "Wire.h"
#include "I2C_eeprom.h"
I2C_eeprom ee(0x50, I2C_DEVICESIZE_24LC256);
union
{
  uint8_t para[24];
  struct
  {
    int chan;      // channel number
    char key[16];  // write key
    uint8_t zero;
    uint8_t mask;  // sensor mask
    uint8_t pad[2];
  } pack;
} ts,ts_test;  // TS send and receive

void write_ee_ts(int channel,char *key,uint8_t mask)
{
  Serial.begin(9600);
  Wire.begin(12,14);
  ee.begin();
  if (! ee.isConnected())
  {    Serial.println("ERROR: Can't find eeprom\nstopped..."); while (1);  }
  ts.pack.chan=channel;
  strncpy(ts.pack.key,key,16);ts.pack.zero=0x00;
```

```
  ts.pack.mask=mask;
  ts.pack.pad[0]=0x00;  ts.pack.pad[1]=0x00;
  Serial.println(); Serial.println("Write TS parameters to external EEPROM");
  for(unsigned int i=0;i<24;i++)    { ee.writeByte(i,ts.para[i]); }
}
void read_ee_ts()
{
  Serial.begin(9600);  // included for completeness
  Wire.begin(12,14);
  ee.begin();
  if (! ee.isConnected())
  {
    Serial.println("ERROR: Can't find eeprom\nstopped..."); while (1);
  }
  Serial.println("TS parameters in external EEPROM");
  for(unsigned int i=0;i<24;i++)
    {  ts_test.para[i]=ee.readByte(i); }
}

void write_ie_ts(int channel,char *key,uint8_t mask)
{
  Serial.begin(9600);
  if(!EEPROM.begin(24))
  {
    Serial.println("failed to initialise internal EEPROM"); while (1);
  }
  ts.pack.chan=channel;
  strncpy(ts.pack.key,key,16);ts.pack.zero=0x00;
  ts.pack.mask=mask;
  ts.pack.pad[0]=0x00;  ts.pack.pad[1]=0x00;
  Serial.println();
  Serial.println("Write TS parameters to internal EEPROM");
  for(unsigned int i=0;i<24;i++)
     { EEPROM.write(i,ts.para[i]); } // write to EEPROM buffer
  EEPROM.commit();  // send the EEPROM buffer to memory
}

void read_ie_ts()
{
  Serial.begin(9600);  // included for completeness
  if(!EEPROM.begin(24))
  {
    Serial.println("failed to initialise internal EEPROM");  while (1);
  }
  Serial.println("TS parameters in internal EEPROM");
  for(unsigned int i=0;i<24;i++)  {  ts_test.para[i]=byte(EEPROM.read(i)); }
  Serial.println();
}
```

11.3.3.Code pour tester les fonctions de la bibliothèque `EEPROM_TS.h`

```
#include "EEPROM_TS.h"

void setup()
{
  Serial.begin(9600);
  Wire.begin(12,14);
  Serial.println(" ");
  write_ee_ts(1243348,"J4K8ZIWAWE8JBIX7",0xC0);
  delay(1000);
  read_ee_ts();
  Serial.printf("Channel number: %d\n",ts_test.pack.chan);
  Serial.printf("Write/read key: %s\n",ts_test.pack.key);
  Serial.printf("mask value: %0X\n",ts_test.pack.mask);
  Serial.println(" ");
  delay(3000);
  write_ie_ts(1243348,"J4K8ZIWAWE8JBIX7",0xC0);
  delay(1000);
  read_ie_ts();
  Serial.printf("Channel number: %d\n",ts_test.pack.chan);
  Serial.printf("Write/read key: %s\n",ts_test.pack.key);
  Serial.printf("mask value: %0X\n",ts_test.pack.mask);
}

void loop() { }
```

11.3.4 A faire

1. Utilisez l'EEPROM externe et interne pour charger les paramètres ThingSpeak.
   Commencez avec une EEPROM externe
   - s'il est disponible, charger les paramètres dans l'EEPROM interne
   - s'il n'est pas disponible prendre les paramètres enregistrés dans l'EEPROM interne

2. Expérimentez avec la valeur du délai d'expiration pour «planifier» l'activation du terminal émetteur
3. Utilisez **`WiFiManager`** pour définir le point d'accès WiFi dans la passerelle

11.4 Terminal récepteur - MODE 2

Comme le terminal émetteur, le terminal récepteur fonctionne en 2 étapes:

1. la première étape permet aux terminaux de s'enregistrer à la passerelle et de recevoir la clé EAS et l'identifiant et de la passerelle via les paquets de contrôle **IDREQ** et **IDACK.** Il s'agit de la même transaction qu'en MODE 1
2. la deuxième étape est utilisée pour envoyer la requête - **DTREQ** et pour recevoir les données demandées du canal ThingSpeak via un paquet **DTRCV**

Le terminal récepteur (**TSR**) envoie les paquets de demande de données (**DTREQ**) et attend les paquets de réception de données (**DTRCV**). Chaque paquet de requête porte le **numéro du canal**, le mot-**clé de lecture** du canal et le **masque des champs** de données (8 bits) indiquant quels sont les champs de données d'intérêt.

Notez que les paquets **DTRCV** contiennent les données demandées ainsi qu'un nouveau timeout ou valeur de cycle dans **rdf.pack.channel** et un message de contrôle dans **rdf.pack.keyword** envoyé par la passerelle.

11.4.1 The code of receiver (data request) terminal - MODE 2

```
#define TERMINAL  // to choose TERMINAL or GATEWAY node
#define MODE 2    // to choose sender (1), receiver (2), publisher (3), subscriber (4) mode
#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include "LoRa_Packets.AES.h"
#include "LoRa_onReceive.AES.h"  // to capture the packets with MODE and server/broker
#include <Wire.h>
#include "SSD1306Wire.h"
SSD1306Wire display(0x3c, 12, 14);

union
{
  uint8_t para[24];
  struct
  {
    int chan;      // channel number
    char key[16];  // write-read key
    uint8_t zero;
    uint8_t mask;  // sensor mask
    uint8_t pad[2];
  } pack;
} ts;  // TS send and receive para

void disp_sens(uint8_t mask, float *stab)
{
  char buff[32];
  display.init();
  display.flipScreenVertically();
  display.setFont(ArialMT_Plain_10);
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.drawString(0,0,"Terminal TSR");  // first 16 lines are yellow
  if(mask&0x80) { sprintf(buff,"T:%2.2f",stab[0]);display.drawString(0,16,buff); }
  if(mask&0x40) { sprintf(buff,"H:%2.2f",stab[1]);display.drawString(0,28,buff); }
  if(mask&0x20) { sprintf(buff,"L:%4.2f",stab[2]);display.drawString(0,40,buff); }
  display.display();
}

void setup() {
  Serial.begin(9600); delay(100);
  Wire.begin(12,14);
  termID=(uint32_t)ESP.getEfuseMac(); delay(100);
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  con_queue = xQueueCreate(queueSize, 32);
  tsrcv_queue = xQueueCreate(queueSize, 64);
  stage1_flag=1;stage2_flag=0;
  ts.pack.chan=1243348; strncpy(ts.pack.key,"0XYA1MAWXFGVWDX9",16);
  ts.pack.zero=0x00;ts.pack.mask=0xE0;  // 3rd sensor
  ts.pack.pad[0]=0x00;ts.pack.pad[1]=0x00;
  }
```

```
int cycle=10;   // 10 seconds

void loop()
{
if(runEvery(cycle*1000))
  {
  if(cycle_cnt>10) { stage1_flag=1; stage2_flag=0;cycle_cnt=0; }
  else { Serial.printf("cycle=%d\n",cycle_cnt); cycle_cnt++; }
  if(stage1_flag==1)  // runs until IDACK sets stage1_flag to 1 - in RTC memory
      {
      conframe_t rcf; int rec=0;
      send_IDREQ("passwordpassword"); // send IDACK for service 1
      Serial.printf("sent IDREQ MODE=%x sent from TERM: %08X\n",MODE,(uint32_t)termID);
      rec=xQueueReceive(con_queue, rcf.frame, 100);
      if(rec)
      {
        Serial.printf("recv IDACK service=%x got from GW: %08X ckey=%16.16s\n",MODE,
                      (uint32_t)gwID,rcf. pack.pass);
        strncpy(CKEY,rcf.pack.pass,16);
        stage1_flag=0; stage2_flag=1;  // transition from stage 1 to stage2
      }
    }

  if(stage2_flag==1)
    {
    TSframe_t rdf;  int rec=0;
    Serial.printf("DTREQ service=%x sent to GW: %08X\n",MODE,(uint32_t)gwID);
    send_DTREQ(ts.pack.mask,ts.pack.chan,ts.pack.key);    // channel and read key
    rec=xQueueReceive(tsrcv_queue, rdf.frame, cycle*100);
    if(rec) Serial.printf("\nDTRCV: cycle=%dsec,%s,%x\n",rdf.pack.channel,rdf.pack.keyword,
                          rdf.pack.con[0]);
    if(rdf.pack.channel>5 && rdf.pack.channel< 3600) // eliminates eventual error
      {
      disp_sens(ts.pack.mask,rdf.pack.sens);
      cycle=rdf.pack.channel;
      }
    }
  }
}
```

11.5 Passerelle récepteur - MODE 2

La passerelle de réception fonctionne en 2 étapes:

1. la première étape permet aux terminaux de s'enregistrer (**IDREQ**) à la passerelle et de recevoir la clé AES et l'identifiant de la passerelle via un paquet **IDACK**
2. dans la deuxième étape, la passerelle reçoit les requêtes de données (**DTREQ**) qui sont relayées vers le serveur ThingSpeak; après la réception des données du serveur ThingSpeak, la passerelle les envoie au terminal via un paquet de type **DTRCV**.

La passerelle de réception reçoit les demandes de données et les met dans la file d'attente (**tsrcv_queue**). Cette file d'attente est lue par le **TS_Task** et les données demandées sont extraites du serveur ThingSpeak. Ensuite, le paquet de type **DTRCV** avec les données reçues est envoyée au terminal concerné.

11.5.1 Le code de la passerelle du récepteur

```
#define GATEWAY  // to choose TERMINAL or GATEWAY node
#define MODE 2  // to choose sender (1), receiver (2) , publisher (3), subscriber (4) mode
#define CKEY "abcdefghijklmnop"  // AES key - 16 bytes
#include <WiFi.h>
#include "ThingSpeak.h"
#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include "LoRa_Packets.AES.h"
#include "LoRa_onReceive.AES.h"  // to capture the packets with MODE and server/broker

const char* ssid = "Livebox-08B0";
const char* pass = "G79ji6dtE..VTPWmZP";
WiFiClient client;

int statusCode=0,tss_flag=0;
void TS_Task( void * pvParameters ){
char kbuff[32];
TSframe_t rdf; int TSdelay=1000;
while(true)
  {
  xQueueReceive(tsrcv_queue, &rdf, portMAX_DELAY);
  Serial.println(rdf.pack.channel);Serial.println(rdf.pack.keyword);
  LoRa.idle();
    memset(kbuff,0x00,32); strncpy(kbuff,rdf.pack.keyword,16);
  if(rdf.pack.con[1] & 0x80)
    { stab[0]=ThingSpeak.readFloatField(rdf.pack.channel,1,kbuff);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x40)
    { stab[1]=ThingSpeak.readFloatField(rdf.pack.channel,2,kbuff);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x20)
    { stab[2]=ThingSpeak.readFloatField(rdf.pack.channel,3,kbuff);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x10)
    { stab[3]=ThingSpeak.readFloatField(rdf.pack.channel,4,kbuff);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x08)
    { stab[4]=ThingSpeak.readFloatField(rdf.pack.channel,5,kbuff);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x04)
    { stab[5]=ThingSpeak.readFloatField(rdf.pack.channel,6,kbuff);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x02)
    { stab[6]=ThingSpeak.readFloatField(rdf.pack.channel,7,kbuff);delay(TSdelay); }
  if(rdf.pack.con[1] & 0x01)
    { stab[7]=ThingSpeak.readFloatField(rdf.pack.channel,8,kbuff);delay(TSdelay); }
  statusCode = ThingSpeak.getLastReadStatus();
  if(statusCode == 200) tss_flag=1;
  else tss_flag=2;
  LoRa_rxMode();
  }
}


void connect() {
  Serial.print("checking wifi...");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print(".");
    delay(1000);
  }
  Serial.println("\nIoT.GW1 - connected!");
}
```

```
void setup() {
  Serial.begin(9600); delay(100);
  gwID=(uint32_t)ESP.getEfuseMac();
  WiFi.begin(ssid, pass);
  if (!client.connected()) { connect(); }
  Serial.println();Serial.println();
  Serial.println("WiFi connected");
  ThingSpeak.begin(client);  // Initialize ThingSpeak
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  con_queue = xQueueCreate(queueSize, 32);
  tsrcv_queue = xQueueCreate(queueSize, 64);
  xTaskCreatePinnedToCore(
                    TS_Task,   /* Function to implement the task */
                    "TS_Task", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore);  /* Core where the task should run */
  Serial.println("Task created...");
  stage1_flag=1;
}

void loop() {
 if(stage1_flag==1)
  {
  conframe_t rcf; int rec=0;
  rec=xQueueReceive(con_queue, rcf.frame, portMAX_DELAY);
  termID=rcf.pack.sid;
  Serial.printf("recv IDREQ MODE=%x from TERM: %08X, pass=%16.16s\n",MODE, termID,rcf.pack.pass);
  send_IDACK(CKEY,0x0000);
  Serial.printf("IDACK service=%x sent\n",MODE);stage1_flag=0;
  }
  if(stage2_flag==1)
  {
  TSframe_t rdf;
  //xQueueReceive(rcv_queue, rdf.frame, portMAX_DELAY);
  int tout=10; char mess[16];

  while(tss_flag==0)
    {
      Serial.println("wating for TS return");
      delay(1000);
    }
  tout=10+random(10,20);
  // we send the time-out in the channel field
  if(tss_flag==1) strcpy(mess,"last read OK   ");
  if(tss_flag==2) strcpy(mess,"last read error");

  send_DTRCV(rdf.pack.con[1],tout,mess,stab);  tss_flag=0;
  Serial.printf("DTRCV to TERM:%08X;tout=%d,stab[0]=%2.2f,mess=%16.16s\n",termID,tout,stab[0],mess);
  stage2_flag=0;
  }
}
```

11.6 A faire

1. Utilisez deux ou trois terminaux pour tester les codes ci-dessus.
2. Écrivez une application pour programmer l'EEPROM externe avec différents mots-clés et paramètres, notamment:
    - pour terminal et passerelle: paramètres de liaison LoRa - fréquence, bande d'onde du signal, facteur d'étalement, code de synchronisation et mots de passe de liaison
    - pour terminal: paramètres ThingSpeak - numéro de canal, mots-clés d'écriture / lecture, masque de capteur
    - pour passerelle: identifiants WiFi

3. Modifiez les codes du terminal et de la passerelle, y compris l'utilisation de mémoires EEPROM externes et internes. Lors de la phase d'initialisation, le terminal / la passerelle essaie de lire l'EEPROM externe. Si elles sont présentes, les données sont transférées de l'EEPROM externe vers l'EEPROM interne. Sinon, les nœuds Terminal / Gateway sont activés directement avec les données de l'EEPROM interne.

Lab 12 - Protocole et passerelles LoRa MQTT

12.1 Introduction

Dans ce laboratoire, nous allons créer une passerelle MQTT, à savoir la passerelle LoRa-WiFi (MQTT). L'architecture IoT correspondante comprend deux cartes ESP32 fonctionnant en tant que passerelle LoRa – MQTT. Une carte passerelle relaie les messages publiés par les terminaux vers le courtier MQTT et la seconde reçoit les messages des abonnés et les relaie vers les terminaux.

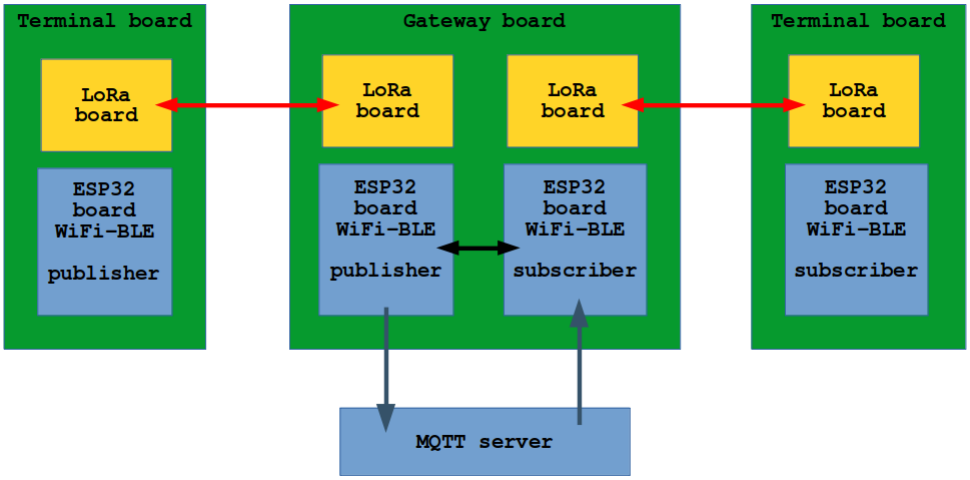


Figure 12.1 Architecture des passerelles LoRa-MQTT avec les terminaux associés.

Les passerelles fonctionnent en deux étapes.

1. la première étape permet aux terminaux de s'enregistrer à la passerelle et de recevoir l'identifiant de la passerelle via le paquet ACK ainsi que le code AES pour le cryptage des paquets de données
2. la deuxième étape est utilisée pour publier les données sur le sujet donné ou pour s'abonner au sujet donné et recevoir les messages publiés par d'autres appareils

L'union/structure suivante représente les éléments essentiels supportant le protocole **LoRa-MQTT**.
Dans la première étape opérationnelle, nous utilisons le paquet de contrôle pour envoyer une demande d'obtention de l'identificateur **`gwID`** et de la clé AES via une requête **`IDREQ`**.

```
typedef union
{
  uint8_t frame[32];
  struct
  {
    uint32_t did;       // destination identifier chipID (4 lower bytes)
    uint32_t sid;       // source identifier chipID (4 lower bytes)
    uint8_t  con[2];    // control field: con[0]
    char     pass[16];  // password or AES key code - 16 characters
    int      tout;      // timeout
    uint8_t  pad[2];    // future use
  } pack;               // control packet
} conframe_t;
```

Le paquet **`IDREQ`** contient une valeur NULL dans le premier champ qui est l'identificateur de destination - **`did`**. Le champ **`sid`** contient la partie inférieure (4 octets) de l'identifiant de puce.

La passerelle envoie le paquet **`IDACK`** correspondant au terminal. L'identifiant de passerelle reçu fourni dans le champ **`sid`** est utilisé dans la deuxième étape pour publier les messages de données et pour s'abonner au sujet donné. Le paquet **`IDACK`** porte également la clé AES fourni pour l'utilisation à l'étape suivante.

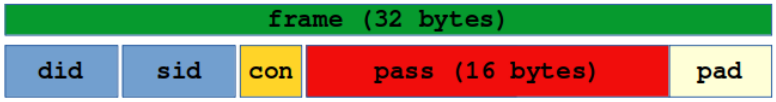


Figure 12.2 Format des paquets de contrôle

Ce sont les paramètres du paquet de contrôle **IDREQ**. L'identifiant du circuit ESP32 (**termID**) est utilisé comme l'adresse source dans le paquet LoRa. Le mot de passe est nécessaire pour pouvoir accéder à la passerelle par l'obtention de son identificateur et de la clé AES.

```
scf.pack.did=(uint32_t)0;
scf.pack.sid=(uint32_t)termID;
scf.pack.con[0]=(uint8_t)(MODE*16+1);
scf.pack.con[1]=0x00;
if(pass!=NULL) strncpy(scf.pack.pass,pass,16);
```

Après l'envoi de son paquet de contrôle, le terminal attend le paquet **IDACK** correspondant généré par la passerelle. Notez que la passerelle n'enverra la trame IDACK correspondante que si le champ de mot de passe fourni par le terminal correspond à la valeur de mot de passe enregistrée dans la passerelle (16 caractères).

```
scf.pack.did=(uint32_t)termID;
scf.pack.sid=(uint32_t)gwID;
scf.pack.con[0]=(uint8_t)16*MODE+2;
scf.pack.con[1]=0x00;
strncpy(scf.pack.pass,aes,16);  // aes points to the AES key
scf.pack.tout=tout;
```

Après la réception de l'identifiant de passerelle, le terminal peut envoyer sa trame de publication. Les données sont envoyées dans le type d'union/structure suivant. Notez que la taille d'un paquet **MQTT** (112 octets) est supérieure à la taille d'un paquet **TS** (64 octets).

```
typedef union
{
  uint8_t frame[112];     // MQTT frame to publish on the given topic
  struct
  {
    uint32_t did;         // destination identifier chipID (4 lower bytes)
    uint32_t sid;         // source identifier chipID (4 lower bytes)
    uint8_t  con[2];      // control field
    char     topic[48];   // topic name - e.g. /esp32/Term1/Sens1
    char     mess[48];    // message value
    int      tout;        // optional timeout for publish frame
    uint8_t  pad[2];      // future use
  } pack;                 // data packet
} MQTTframe_t;
```

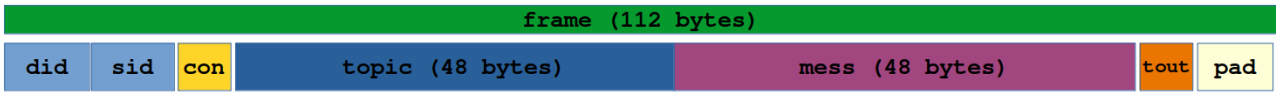


Figure 12.3 Format des paquets de données MQTT

L'exemple d'initialisation d'un paquet **DTPUB**. A noter l'utilisation de la fonction d'encryptage.

```
sdf.pack.did=(uint32_t)gwID;
sdf.pack.sid=(uint32_t)termID;
sdf.pack.con[0]=(uint8_t)(MODE*16+3);
sdf.pack.con[1]=0x00;
strcpy(sdf.pack.topic,topic);
strcpy(sdf.pack.mess,mess);
sdf.pack.tout=0;
encrypt(sdf.frame,CKEY,sdcf.frame,7);  // 7*16=112
strcpy(sdf.pack.mess,mess);
sdf.pack.tout=0;
```

Après l'envoi d'un paquet **DTPUB** le terminal attend que le paquet **DTPUBACK** soit envoyé par la passerelle.

```
sdf.pack.did=(uint32_t)termID;
sdf.pack.sid=(uint32_t)gwID;
sdf.pack.con[0]=(uint8_t)16*MODE+4;
sdf.pack.con[1]=0x00;
strncpy(sdf.pack.topic,topic,48);
strncpy(sdf.pack.mess,mess,48);
sdf.pack.tout=tout;
encrypt(sdf.frame,CKEY,sdcf.frame,4);
```

Ce paquet contient un champ qui porte la valeur du délai d'expiration qui peut être utilisée avant l'envoi du prochain paquet de publication.

La passerelle peut utiliser ce paramètre pour «programmer» l'émission des paquets envoyés par les terminaux afin d'éviter les collisions.

L'extrait de code suivant montre l'utilisation du paramètre **timeout** (secondes) dans le terminal fonctionnant en mode **deep_sleep** :

```
esp_sleep_enable_timer_wakeup(1000*1000*timeout);
esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH, ESP_PD_OPTION_OFF);
esp_deep_sleep_start();
```

La valeur de temporisation reçue est convertie en le nombre de microsecondes pendant lesquelles le circuit ESP32 passe en état de sommeil profond. Notez que pendant cette période, toutes les variables volatiles sont perdues. Si nous devons les conserver, ils doivent d'abord être stockées dans la mémoire RTC locale qui est alimentée pendant la période de **deep_sleep**.

La passerelle de la publication utilise la même union/structure pour recevoir et envoyer les messages LoRa. Le nœud de passerelle utilise sa boucle principale pour rechercher les paquets LoRa entrants et pour générer les paquets de retour (ACK) correspondants.

Les paquets entrants sont détectés et reçues par une ISR **onReceive()**. Cette ISR est définie dans le fichier **LoRa_onReceive.h** à intégrer dans les codes de terminaux et de passerelles.
Ensuite, les drapeaux correspondants sont affectés à la réception des paquets :

- **stage1_flag** après la réception d'un paquet **IDREQ** et
- **stage2_flag** après la réception d'un paquet **DTPUB**.

La boucle principale appelle les fonctions de confirmation pour envoyer les paquets **IDACK** et **DTPUBACK**.

Voici la boucle principale du terminal :

```
int cycle=10;   // 10 seconds

void loop()
{
if(runEvery(cycle*1000))
  {
  if(cycle_cnt>10) { stage1_flag=1; stage2_flag=0;cycle_cnt=0; }
  else { Serial.printf("cycle=%d\n",cycle_cnt); cycle_cnt++; }
  if(stage1_flag==1)
      {
      conframe_t rcf; int rec=0;
      send_IDREQ("passwordpassword"); // send IDACK for service 1
      Serial.printf("sent IDREQ MODE=%x sent from TERM: %08X\n",MODE,(uint32_t)termID);
      rec=xQueueReceive(con_queue, rcf.frame, 100);
      if(rec)
      {
        Serial.printf("recv IDACK service=%x got from GW: %08X ckey=%16.16s\n",MODE,(uint32_t)gwID,
                      rcf.pack.pass);
        strncpy(CKEY,rcf.pack.pass,16);
        stage1_flag=0; stage2_flag=1;  // transition from stage 1 to stage2
      }
    }
  if(stage2_flag==1)
    {
    TSframe_t rdf;  int rec=0;
    Serial.printf("DTREQ service=%x sent to GW: %08X\n",MODE,(uint32_t)gwID);
```

```
    send_DTREQ(ts.pack.mask,ts.pack.chan,ts.pack.key);  // channel and read key
    rec=xQueueReceive(tsrcv_queue, rdf.frame, cycle*100);
    if(rec) Serial.printf("\nDTRCV: cycle=%dsec,%s,%x\n", rdf.pack.channel,rdf.pack.keyword,
                          rdf.pack.con[0]);
    if(rdf.pack.channel>5 && rdf.pack.channel< 3600) // eliminates error
      {
      disp_sens(ts.pack.mask,rdf.pack.sens);
      cycle=rdf.pack.channel;  // eliminates errors
      }
    }
  }
}
```

Dans le nœud de passerelle, lorsqu'un paquet **DTPUB** est détecté par **on_Receive()**, la passerelle transfère sa charge utile à la tâche **MQTT_Task** via une file d'attente :

```
#if MODE>2
if(packetSize==112)
  {
  MQTTframe_t rdf,rdcf; int i=0;
  char ckey[17];
  strncpy(ckey,CKEY,16);ckey[16]='\0';
  while (LoRa.available()) { rdcf.frame[i] = LoRa.read();i++;}
  decrypt(rdcf.frame,ckey,rdf.frame,7);
  #ifdef GATEWAY
  DT_mode=MODE*16+3;
  if(rdf.pack.con[0]==DT_mode && rdf.pack.did==gwID)  // GW:received data packet -  DTPUB
    { stage2_flag=1; termID=rdf.pack.sid;  Serial.println("GW:Received MQ data_MODE");
      xQueueReset(mqrcv_queue);  // reset queue to keep only the last packet
      xQueueSend(mqrcv_queue, rdf.frame, portMAX_DELAY);
    }
  #endif
  #ifdef TERMINAL
  ACK_mode=MODE*16+4;
  if(rdf.pack.con[0]==ACK_mode && rdf.pack.did==termID)  // TERM:received data packet
    { stage2_flag=1;
      gwID=rdf.pack.sid; Serial.println("TERM:Received MQ data_MODE");
      xQueueReset(mqrcv_queue);  // reset queue to keep only the last packet
      xQueueSend(mqrcv_queue, rdf.frame, portMAX_DELAY);
    }
  #endif
  }
#endif
```

La tâche **MQTT_Task** reçoit le paquet dans la file d'attente et envoie sa charge utile au serveur MQTT via une liaison WiFi préparée et la connexion TCP au courtier MQTT externe.

```
void MQTT_Task( void * pvParameters ){
char kbuff[32];
MQTTframe_t rdf;
while(true)
  {
  xQueueReceive(rcv_queue, &rdf.pack, portMAX_DELAY);
  LoRa.idle();
  Serial.printf("%s\n",rdf.pack.topic);
  Serial.printf("%s\n",rdf.pack.mess);
  if (!client.connected()) { connect(); }
  client.publish(rdf.pack.topic,rdf.pack.mess);  // sends publish message to the broker
  delay(1000);mqpub_flag=1;
  LoRa_rxMode();
  }
}
```

Notez que le modem LoRa est mis à l'état inactif par **LoRa.idle()** afin de ne pas accepter les interruptions d'E/S tant que la communication avec le courtier MQTT n'est pas terminée.
Après la publication du nouveau message sur le courtier MQTT, le modem LoRa revient à l'état de réception précédent via la fonction **LoRa_rxMode()**.

12.2 Terminal de l'éditeur (*publish*)

Le code suivant implémente un nœud terminal fonctionnant en MODE 3 qui envoie les données à publier sur le sujet donné par la passerelle «éditeur».

Voici la boucle principale du terminal. Notez l'utilisation des files d'attente pour les paquets **IDACK** (**con_queue**) et **DTPUBACK** (**mqrcv_queue**). Dans ces files on attend les réponses aux paquets **IDREQ** et **DTPUB** envoyés par notre terminal.

```
void loop()
{
if(runEvery(cycle*1000))
  {
  if(cycle_cnt>10) { stage1_flag=1; stage2_flag=0;cycle_cnt=0; }
  else { Serial.printf("cycle=%d\n",cycle_cnt); cycle_cnt++; }
  if(stage1_flag==1)
    {
    conframe_t rcf; int rec=0;
    send_IDREQ("passwordpassword");  // password to get acces to gateway and obtain AES key
    Serial.printf("sent IDREQ MODE=%x sent from TERM: %08X\n",MODE,(uint32_t)termID);
    rec=xQueueReceive(con_queue, rcf.frame, 100); // 100 ms
    if(rec)
      {
      Serial.printf("recv IDACK service=%x got from GW: %08X ckey=%16.16s\n",MODE,(uint32_t)gwID,
                    rcf.pack.pass);
      strncpy(CKEY,rcf.pack.pass,16);  // obtained AES key from IDACK packet
      stage1_flag=0; stage2_flag=1;  // transition from stage 1 to stage2
      }
    }

  if(stage2_flag==1)
    {
    MQTTframe_t rdf;
    get_sens();
    sprintf(message, "T:%2.2f, H:%2.2f",stab[0],stab[1]); // stab[] - table with sensor values
    Serial.printf("DTPUB MODE=%x sent from TERM: %08X\n",MODE,(uint32_t)termID);
    send_DTPUB(topic,message);
    xQueueReceive(mqrcv_queue, rdf.frame, 600);
    Serial.printf("recv DTPUBACK MODE=%x from GW:%08X, cycle=%dsec, topic=%s\n",MODE,
                  (uint32_t)rdf.pack.sid, rdf.pack.tout,rdf.pack.topic);
    }
  }
}
```

12.2.1 Publisher terminal – complete code

```
#define TERMINAL  // to choose TERMINAL or MASTER (GATEWAY) node
#define MODE  3  // to choose sender (1), receiver (2) , publisher (3), subscriber (4) mode

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

#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include "LoRa_Packets.AES.h"
#include "LoRa_onReceive.AES.h"  // to capture the packets with MODE and server/broker

char topic[48];   // this variable may be loaded from an EEPROM (external-internal)
char message[48];  // this variable to be loaded with sensor message

void get_sens()
{
    SHT21.begin();
    stab[0]=SHT21.getTemperature();
    delay(100);
    stab[1]=SHT21.getHumidity();
    Serial.printf("T:%2.2f, H:%2.2f\n",stab[0],stab[1]);
}

void setup() {
  Serial.begin(9600); delay(100);
```

```
  Wire.begin(12,14);
  termID=(uint32_t)ESP.getEfuseMac(); delay(100);
  strcpy(topic,"/esp32/my_sensors/");
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  con_queue = xQueueCreate(queueSize, 32);
  mqrcv_queue = xQueueCreate(queueSize, 112);
  stage1_flag=1;stage2_flag=0;
}

int cycle=10;   // 10 seconds

void loop()
{
if(runEvery(cycle*1000))
  {
   … // main loop presented above
  }
}
```

12.3 Passerelle de l'éditeur (*publisher*) - MODE 3

La passerelle de l'éditeur MQTT prépare les informations d'identification WiFi par **`WiFi.begin(ssid, pass);`**, initialise la connexion LoRa et crée la tâche **`MQTT_Task`**.

La tâche **`MQTT_Task`** reçoit les paquets **`DTPUB`** via **`mqrcv_queue`** et active la connexion au courtier MQTT par **`connect();`**

Ensuite, elle envoie - publie le message sur le sujet donné. Pendant cette période, le modem LoRa est mis à l'état inactif.

Voici le code de la tâche **`MQTT_Task`** :

```
void MQTT_Task( void * pvParameters ){
char kbuff[32];
MQTTframe_t rdf;
while(true)
  {
  xQueueReceive(mqrcv_queue, &rdf.pack, portMAX_DELAY);
  LoRa.idle();
  Serial.printf("%s\n",rdf.pack.topic);
  Serial.printf("%s\n",rdf.pack.mess);
  if (!client.connected()) { connect(); }
  client.publish(rdf.pack.topic,rdf.pack.mess);
  //client.publish("/esp32/my_sensors/","newtest"); // test line
  delay(1000);mqpub_flag=1;
  LoRa_rxMode();
  }
}
```

12.3.1 Passerelle de l'éditeur - code complet

```
#define MASTER  // to choose TERMINAL or MASTER (GATEWAY) node
#define MODE 3  // to choose sender (1), receiver (2) , publisher (3), subscriber (4) mode
#define CKEY "abcdefghijklmnop"  // AES key - 16 bytes

#include <WiFi.h>
#include <MQTT.h>
#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include "LoRa_Packets.AES.h"
#include "LoRa_onReceive.AES.h"  // to capture the packets with MODE and server/broker

const char* ssid = "Livebox-08B0";
const char* pass = "G79ji6dtEptVTPWmZP";


const char* mqttServer = "broker.emqx.io";
WiFiClient net;
MQTTClient client;

int mqpub_flag=0;  // MQTT publish flag

void MQTT_Task( void * pvParameters ){
char kbuff[32];
MQTTframe_t rdf;
while(true)
  {
  xQueueReceive(mqrcv_queue, &rdf.pack, portMAX_DELAY);
  LoRa.idle();
  Serial.printf("%s\n",rdf.pack.topic);
  Serial.printf("%s\n",rdf.pack.mess);
  if (!client.connected()) { connect(); }
  client.publish(rdf.pack.topic,rdf.pack.mess);
  //client.publish("/esp32/my_sensors/","newtest");
  delay(1000);mqpub_flag=1;
  LoRa_rxMode();
  }
}
void connect() {
  Serial.print("checking wifi...");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print("."); delay(1000);}
```

```
  client.begin(mqttServer, net);
  Serial.println("\nconnecting...");
  while (!client.connect("IoT.GW3", "try", "try")) {
    Serial.print(".");delay(1000);}
  Serial.printf("IoT.GW3 - connected!\n");
}

void setup() {
  Serial.begin(9600); delay(100);
  gwID=(uint32_t)ESP.getEfuseMac();
  WiFi.begin(ssid, pass);

  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  con_queue = xQueueCreate(queueSize, 32);
  mqrcv_queue = xQueueCreate(queueSize, 112);
  xTaskCreatePinnedToCore(
                    MQTT_Task,   /* Function to implement the task */
                    "MQTT_Task", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    taskCore);  /* Core where the task should run */
  Serial.println("Task created...");
  stage1_flag=1;
}

void loop() {
 if(stage1_flag==1)
  {
  conframe_t rcf; int rec=0;
  rec=xQueueReceive(con_queue, rcf.frame, portMAX_DELAY);
  termID=rcf.pack.sid;
  Serial.printf("recv IDREQ MODE=%x from TERM: %08X, pass=%16.16s\n",MODE, termID,rcf.pack.pass);
  send_IDACK(CKEY,0x0000); // send IDACK for service 1 with CKEY
  Serial.printf("IDACK service=%x sent\n",MODE);stage1_flag=0;
  }
  if(stage2_flag==1)
  {
  MQTTframe_t rdf;
  int tout=10; char mess[48];char control[24];
  strcpy(mess,"message"); strcpy(control,"control");
  while(mqpub_flag==0)
    {
    Serial.println("wating for MQTT return");
    delay(1000);
    }
  tout=10+random(10,20);
  // we send the time-out in the channel field
  send_DTPUBACK(control,mess,tout);  mqpub_flag=0;
  Serial.printf("DTPUBACK to TERMINAL: %08X ; timeout=%d\n",termID,tout);
  stage2_flag=0;
  }


}
```

12.3.2 A faire

1. utiliser l'écran OLED pour afficher différents paramètres et messages
2. utiliser le mode deep_sleep dans le nœud Terminal
3. utilisez WiFiManager pour définir le point d'accès WiFi dans la passerelle

12.4 Terminal abonné - MODE 4

Le terminal d'abonné (*subscriber*) fonctionne également en 2 étapes:

1. la première étape permet aux terminaux de s'enregistrer auprès de la passerelle et de recevoir la clé AES et l'identifiant de la passerelle via le paquet **IDREQ**
2. la deuxième étape est utilisée pour abonner le terminal au sujet demandé via un paquet **DTSUB**, puis attendre le paquet de réponse - paquet **DTSUBRCV**

Le cycle de la boucle principale commence par l'envoi du paquet **IDREQ**. Après la réception de l'**IDACK**, le cycle commence à envoyer des paquets **DTSUB** comprenant la demande d'abonnement au sujet donné.

```
int cycle=10;   // 10 seconds

void loop()
{
if(runEvery(cycle*1000))
  {
  if(cycle_cnt>10) { stage1_flag=1; stage2_flag=0;cycle_cnt=0; }
  else { Serial.printf("cycle=%d\n",cycle_cnt); cycle_cnt++; }
  if(stage1_flag==1)
    {
    conframe_t rcf; int rec=0;
    send_IDREQ("passwordpassword");
    Serial.printf("sent IDREQ MODE=%x sent from TERM: %08X\n",MODE,(uint32_t)termID);
    rec=xQueueReceive(con_queue, rcf.frame, 10000);
    if(rec)
      {
      Serial.printf("recv IDACK MODE=%x got from GW: %08X ckey=%16.16s\n",MODE,
                    (uint32_t)gwID,rcf.pack.pass);
      strncpy(CKEY,rcf.pack.pass,16);
      stage1_flag=0; stage2_flag=1;  // transition from stage 1 to stage2
      }
    }

  if(stage2_flag==1) // runs until DTPUBACK sets stage2_flag to 1
    {
    MQTTframe_t rdf; int rec=0;
    send_DTSUB("/esp32/my_sensors/");
    Serial.printf("DTSUB MODE=%x sent from TERM: %08X\n",MODE,(uint32_t)termID);
    if(xQueueReceive(mqrcv_queue, rdf.frame, cycle*1000)== pdTRUE)
      {
      Serial.printf("\nDTSUBRCV: %s:%s, %d\n",rdf.pack.topic,rdf.pack.mess,rdf.pack.tout);
      disp_sens(rdf.pack.topic, rdf.pack.mess);
      if(rdf.pack.tout>0 && rdf.pack.tout <3600) cycle=rdf.pack.tout; else cycle=10;
      }
    else Serial.println("rcv_queue timeout");
     }
   }
}
```

Du côté de la réception, **onReceive()** capture le paquet **IDACK** pour le MODE 4 et **DTSUBRCV** avec le message pour le sujet donné. Le paquet reçu **DTSUBRCV** est envoyé à la boucle principale via **mqrcv_queue** avec **stage2_flag** mis à 1.

```
..
xQueueReset(mqrcv_queue);
xQueueSend(mqrcv_queue, &rdf, portMAX_DELAY);
..
```

La fonction d'abonnement côté terminal est :

```
void send_DTSUB(char *topic)  // TERM:send DTSUB frame - subscribe topic
{
MQTTframe_t sdf,sdcf;
LoRa_txMode();
LoRa.beginPacket();
sdf.pack.did=(uint32_t)gwID;
sdf.pack.sid=(uint32_t)termID;
  sdf.pack.con[0]=(uint8_t)(MODE*16+3);           // MODE  4 - type 3  data TERM to GW
```

```
  sdf.pack.con[1]=0x00;
  strncpy(sdf.pack.topic,topic,48);
  memset(sdf.pack.mess,0x00,48);
  sdf.pack.tout=0;
  encrypt(sdf.frame,CKEY,sdcf.frame,7);
  LoRa.write(sdcf.frame,112);
  LoRa.endPacket(true);
}
```

12.4.1 Code complet du terminal d'abonné – MODE 4

Le terminal d'abonné reçoit le paquet de données de la passerelle et affiche le résultat sur son écran OLED.

```
#define TERMINAL  // to choose TERMINAL or GATEWAY node
#define MODE 4  // to choose sender (1), receiver (2) , publisher (3), subscriber (4) mode
#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include "LoRa_Packets.AES.h"
#include "LoRa_onReceive.AES.h"

#include <Wire.h>
#include "SSD1306Wire.h"

SSD1306Wire display(0x3c, 12, 14);

void disp_sens(char *topic, char *mess)
{
  char buff[32];
  display.init();
  display.flipScreenVertically();
  display.setFont(ArialMT_Plain_10);
  display.setTextAlignment(TEXT_ALIGN_LEFT);
  display.drawString(0,0,"Terminal MQSUB");  // first 16 lines are yellow
  display.drawString(0,16,topic);
  display.drawString(0,28,mess);

  display.display();
}

void setup() {
  Serial.begin(9600); delay(100);
  Wire.begin(12,14);
  termID=(uint32_t)ESP.getEfuseMac(); delay(100);
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  con_queue = xQueueCreate(queueSize, 32);
  mqrcv_queue = xQueueCreate(queueSize, 112);
  stage1_flag=1;stage2_flag=0;
}

int cycle=20;   // 20 seconds

void loop()
{
if(runEvery(cycle*1000))
  {
  if(cycle_cnt>10) { stage1_flag=1; stage2_flag=0;cycle_cnt=0; }
  else { Serial.printf("cycle=%d\n",cycle_cnt); cycle_cnt++; }
  if(stage1_flag==1)  // runs until IDACK sets stage1_flag to 1 - in RTC memory
    {
    conframe_t rcf; int rec=0;
    send_IDREQ("passwordpassword"); // send IDACK for service 1
    Serial.printf("sent IDREQ MODE=%x sent from TERM: %08X\n",MODE,(uint32_t)termID);
    rec=xQueueReceive(con_queue, rcf.frame, 10000); // portMAX_DELAY);  // delay in number of tics -
100 Hz
    if(rec)
      {
      Serial.printf("recv IDACK MODE=%x got from GW: %08X ckey=%16.16s\n",MODE,
(uint32_t)gwID,rcf.pack.pass);
      strncpy(CKEY,rcf.pack.pass,16);
      stage1_flag=0; stage2_flag=1;  // transition from stage 1 to stage2
      }
    }
```

```
  if(stage2_flag==1) // runs until DTPUBACK sets stage2_flag to 1
    {
    MQTTframe_t rdf; int rec=0;
    send_DTSUB("/esp32/my_sensors/");
    Serial.printf("DTSUB MODE=%x sent from TERM: %08X\n",MODE,(uint32_t)termID);
    if(xQueueReceive(mqrcv_queue, rdf.frame, cycle*1000)== pdTRUE)
      {
      Serial.printf("\nDTSUBRCV: %s:%s, %d\n",rdf.pack.topic,rdf.pack.mess,rdf.pack.tout);
      disp_sens(rdf.pack.topic, rdf.pack.mess);
      if(rdf.pack.tout>10 && rdf.pack.tout <3600) cycle=rdf.pack.tout; else cycle=20;
      }
    else Serial.println("rcv_queue timeout");
     }
   }
}
```

12.5 Passerelle d'abonné – MODE 4

La passerelle d'abonnés MQTT fonctionne en 2 étapes:

1. la première étape permet aux terminaux de s'enregistrer à la passerelle et de recevoir l'identifiant de la passerelle (et la clé AES) via les paquets **`IDREQ`** et **`IDACK`** (MODE 4)
2. la deuxième étape est utilisée pour abonner le terminal au sujet demandé par **`DTSUB`** et pour envoyer les données reçues du sujet demandé - paquet **`DTSUBR`**. Dans cette étape la passerelle attend les messages du broker, associés au sujet demandé, sur une fonction de ***callback***.

Voici la boucle principale :

```
void loop()
{
int tout=10;
if(stage1_flag==1)
  {
  conframe_t rcf; int rec=0;
  rec=xQueueReceive(con_queue, rcf.frame, portMAX_DELAY);
  termID=rcf.pack.sid;
  Serial.printf("recv IDREQ MODE=%x from TERM: %08X, pass=%16.16s\n",MODE, termID,rcf.pack.pass);
  send_IDACK(CKEY,0x0000); // send IDACK for service 1 with CKEY
  Serial.printf("IDACK service=%x sent\n",MODE);stage1_flag=0;stage2_flag=1;
  }
  if(stage2_flag==1)
  {
  SUB_t mqtt_sub; int rec=0;
  if(xQueueReceive(mqtt_queue, &mqtt_sub, 80000)== pdTRUE)
    {
    tout=10+random(10,30);
    send_DTSUBRCV(mqtt_sub.topic,mqtt_sub.mess,tout);
    Serial.printf("DTSUBRCV to TERMINAL: topic:%s, message:%s\n",mqtt_sub.topic,mqtt_sub.mess);
    }
  else  Serial.println("mqtt_sub queue timeout");
  stage2_flag=0;
  }
}
```

La réception du paquet **`DTSUB`** active la tâche **`MQTT_Task`** qui envoie la demande d'abonnement reçue au serveur **MQTT** désigné via la fonction : **`client.subscribe (rdf.pack.topic);`**

```
void MQTT_Task( void * pvParameters ){  // subscribe task
char kbuff[32];
MQTTframe_t rdf;
while(true)
  {
  xQueueReceive(mqrcv_queue, &rdf.pack, portMAX_DELAY);
  LoRa.idle();   // we send the time-out in the channel field
  Serial.printf("%s\n",rdf.pack.topic);
  if (!client.connected()) { connect();}
  client.subscribe(rdf.pack.topic);
  Serial.println("topic subscribed");
  delay(100);
  LoRa_rxMode();
  }
}
```

La fonction **`connect()`** réactive la liaison WiFi et la connexion TCP au courtier MQTT.

```
void connect() {
  Serial.print("checking wifi");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print("."); delay(1000);}
  Serial.println("\nconnecting");
  while (!client.connect("GW4", "try", "try")) {
    Serial.print(".");delay(1000);}
  Serial.printf("MQTT broker - connected!\n");
}
```

Notez l'utilisation de l'appel **`LoRa.idle ()`** qui éteint le module LoRa pendant la période de communication (abonnement) avec le serveur MQTT.

L'ISR **messageReceived()** capture l'arrivée des messages publiés (sur le sujet donné) et stocke les données reçues: sujet et charge utile (message) dans la structure **MQTTframe_t** qui est utilisée pour préparer le paquet **DTRCVACK** au terminal concerné.

```
void messageReceived(String &topic, String &payload) {
  SUB_t mqtt_sub;
  Serial.println("incoming: " + topic + " - " + payload);
  topic.toCharArray(mqtt_sub.topic,48);
  payload.toCharArray(mqtt_sub.mess,48);
  xQueueReset(mqtt_queue);  // reset queue to keep only the last packet
  xQueueSend(mqtt_queue, &mqtt_sub, portMAX_DELAY);
  mqsub_flag=1;
}
```

12.5.2 Code complet de la passerelle d'abonné - MODE 4

```
#define GATEWAY  // to choose TERMINAL or GATEWAY node
#define MODE 4  // to choose sender (1), receiver (2) , publisher (3), subscriber (4) mode
#define CKEY "abcdefghijklmnop"  // AES key - 16 bytes
#include <WiFi.h>
#include <MQTT.h>
#include <SPI.h>
#include <LoRa.h>
#include "LoRa_Para.h"
#include "LoRa_Packets.AES.h"
#include "LoRa_onReceive.AES.h"  // to capture the packets with MODE and server/broker
const char* ssid = "Livebox-08B0";
const char* pass = "G79ji6dtE..VTPWmZP";
const char* mqttServer = "broker.emqx.io";
WiFiClient net;
MQTTClient client;
int mqsub_flag=0;  // MQTT subscribe get message flag
QueueHandle_t mqtt_queue;   // receive queue to get out the data packets form onReceive() ISR

typedef struct
{
  char topic[48];
  char mess[48];
} SUB_t;  // structure for topic and message

void messageReceived(String &topic, String &payload) {
  SUB_t mqtt_sub;
  Serial.println("incoming: " + topic + " - " + payload);
  topic.toCharArray(mqtt_sub.topic,48);
  payload.toCharArray(mqtt_sub.mess,48);
  xQueueReset(mqtt_queue);  // reset queue to keep only the last packet
  xQueueSend(mqtt_queue, &mqtt_sub, 100000);
  mqsub_flag=1;
}

void MQTT_Task( void * pvParameters ){  // subscribe task
char kbuff[32];
MQTTframe_t rdf;
while(true)
  {
  xQueueReceive(mqrcv_queue, &rdf.pack, portMAX_DELAY);
  LoRa.idle();
  Serial.printf("%s\n",rdf.pack.topic);
  if (!client.connected()) { connect();}
  client.subscribe(rdf.pack.topic);
  Serial.println("topic subscribed");
  delay(100);
  LoRa_rxMode();
  }
}

void connect() {
  Serial.print("checking wifi");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print("."); delay(1000);}
  Serial.println("\nconnecting");
  while (!client.connect("GW4", "try", "try")) {
    Serial.print(".");delay(1000);}
  Serial.printf("MQTT broker - connected!\n");
}
```

```
void setup() {
  Serial.begin(9600); delay(100);
  gwID=(uint32_t)ESP.getEfuseMac();
  WiFi.begin(ssid, pass);
  client.begin(mqttServer, net);
  client.onMessage(messageReceived);
  set_LoRa();
  LoRa.onReceive(onReceive);
  LoRa.onTxDone(onTxDone);
  LoRa_rxMode();
  con_queue = xQueueCreate(queueSize, 32);
  mqtt_queue = xQueueCreate(queueSize, 96);
  mqrcv_queue = xQueueCreate(queueSize, 112);
  xTaskCreatePinnedToCore(
                    MQTT_Task,   /* Function to implement the task */
                    "MQTT_Task", /* Name of the task */
                    10000,      /* Stack size in words */
                    NULL,       /* Task input parameter */
                    0,          /* Priority of the task */
                    NULL,       /* Task handle. */
                    1);  /* Core where the task should run */
  Serial.println("Task created...");
  stage1_flag=1; stage2_flag=0;
}

void loop()
{
int tout=10;
if(stage1_flag==1)
  {
  conframe_t rcf; int rec=0;
  rec=xQueueReceive(con_queue, rcf.frame, portMAX_DELAY);
  termID=rcf.pack.sid;
  Serial.printf("recv IDREQ MODE=%x from TERM: %08X, pass=%16.16s\n",MODE, termID,rcf.pack.pass);
  send_IDACK(CKEY,0x0000); // send IDACK for service 1 with CKEY
  Serial.printf("IDACK service=%x sent\n",MODE);stage1_flag=0;stage2_flag=1;
  }
  if(stage2_flag==1)
  {
  SUB_t mqtt_sub; int rec=0;
  if(xQueueReceive(mqtt_queue, &mqtt_sub, 80000)== pdTRUE)
    {
    tout=10+random(10,30);
    send_DTSUBRCV(mqtt_sub.topic,mqtt_sub.mess,tout);
    Serial.printf("DTSUBRCV to TERMINAL: topic:%s, message:%s\n",mqtt_sub.topic,mqtt_sub.mess);
    }
  else  Serial.println("mqtt_sub queue timeout");
  stage2_flag=0;
  }
}
```

Voici un fragment de la sortie du terminal d'Arduino IDE pour le fonctionnement de la passerelle MQTT d'abonnement:

```
BAND=868000000.000000,SF=7,SBW=125000.000000,SW=F3,BR=8
Task created...
Received IDREQ_MQS
IDACK service=4 sent
TxDone
Received DTSUB
/esp32/my_sensors/
checking wifi...
connecting...
IoT.GW3 - connected!
topic subscribed
Received DTSUB
/esp32/my_sensors/
incoming: /esp32/my_sensors/ - T:24.6,H:98.77
DTSUBRCV to TERMINAL: topic:/esp32/my_sensors/, message:T:24.6,H:98.77
topic subscribed
Received DTSUB
/esp32/my_sensors/
incoming: /esp32/my_sensors/ - T:24.6,H:98.77
DTSUBRCV to TERMINAL: topic:/esp32/my_sensors/, message:T:24.6,H:98.77
topic subscribed
```

Les messages de publication sont générés par le code C suivant. L'argument **a[1]** spécifie le nombre de secondes entre deux messages de publication consécutifs.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int c, char **a)
{
time_t t; float temp,humi;
char command[128];
if(c<2) { printf("usage: %s cycle_in_sec\n",a[0]); exit(1); }
srand((unsigned) time(&t));
while(1)
  {
  temp=(float)(10+rand()%30);
  humi=(float)(10+rand()%90);
  sprintf(command,"mosquitto_pub -h broker.emqx.io -t /esp32/my_sensors/ -m T:%2.2f,H:%2.2f ",
          temp,humi);
  system(command);
  sleep(atoi(a[1]));
  }
}
```

12.5.3 A faire

1. Utilisez deux ou trois terminaux pour tester le code ci-dessus.
2. Utilisez coté passerelle l'écran OLED pour afficher différents paramètres et messages
3. Utilisez le mode deep_sleep dans le nœud Terminal
4. Utilisez **WiFiManager** pour définir le point d'accès WiFi dans la passerelle

Table des matières