

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.

Table des matières

Lab 7.....	1
Communication LoRa (Long Range) au niveau de la couche physique avec le modem SX127X.....	1
7.1 Liaison radio LoRa.....	2
7.1.1 Communication avec un spectre étalé.....	2
7.2 LoRa communication with SX1276/78 modem.....	6
7.2.1 LoRa - Packet Mode.....	6
7.2.2 LoRa - format de paquet de longueur variable (SX1276/78).....	6
7.2.2.1 Préambule.....	7
7.2.2.2 En-tête (<i>header</i>).....	7
7.3 Programmation LoRa (SX1276/78) avec bibliothèque LoRa.h.....	8
7.3.1 Configuration des paramètres de liaison physique.....	10
7.3.1.1 Paramètres de modulation et de contrôle des erreurs.....	10
7.3.1.2 Création et envoi des paquets LoRa (trames).....	10
7.3.1.3 Réception des paquets LoRa (trames).....	10
7.3.1.4 Protéger le canal de communication.....	11
7.4 Nœuds émetteurs et récepteurs simples.....	12
7.4.1 Expéditeur simple.....	12
7.4.2 Récepteur simple.....	13
7.4.3 Récepteur simple avec fonction de rappel (<i>callback</i>).....	14
7.5 Communication en mode duplex.....	15
7.5.1 Duplex avec la fonction <i>parsePacket()</i>	15
7.5.1.1 Code complet pour la communication LoRa duplex avec <i>parsePacket()</i>	15
7.5.1.2 Duplex simple avec rappel.....	16
7.6 Communication LoRa simple en liaison montante et descendante avec inversion IQ.....	18
7.6.1 Le code du Terminal avec inversion IQ à la réception.....	18
7.6.2 Le code du Maître avec inversion IQ lors de la transmission.....	19
7.7 À faire:.....	20
7.8 Annexe – LoRa_Para.h.....	21
7.8.1 Terminal code avec IQ inversion et fichier LoRa_Para.h.....	23
7.8.2 Gateway code avec IQ inversion et fichier LoRa_Para.h.....	24

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.

$$C = B \cdot \log_2(1 + S/N)$$

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 **log_e** naturel, et en notant que **ln = log_e** on obtient l'équation suivante :

$$C/B = 1.433 \cdot S/N$$

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 :

$$C/B \approx S/N \text{ ou } N/S \approx B/C$$

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.

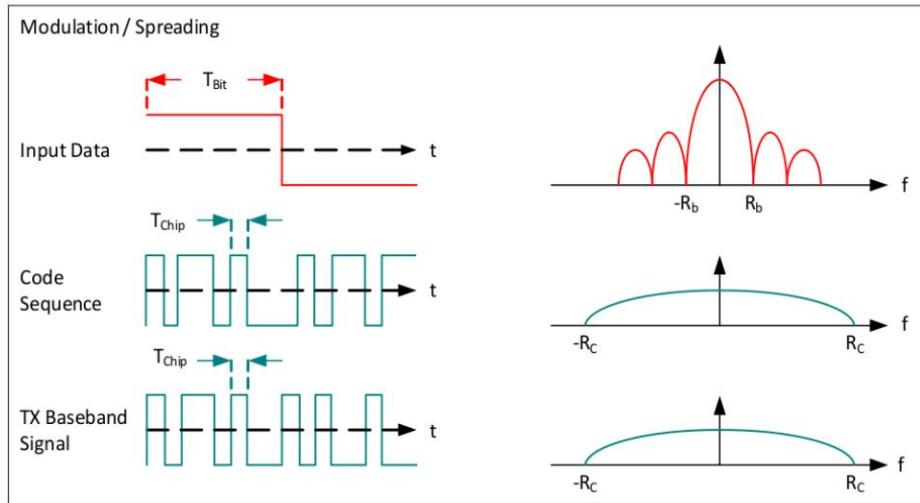


Figure 7.1 Modulation avec étalement du signal

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.

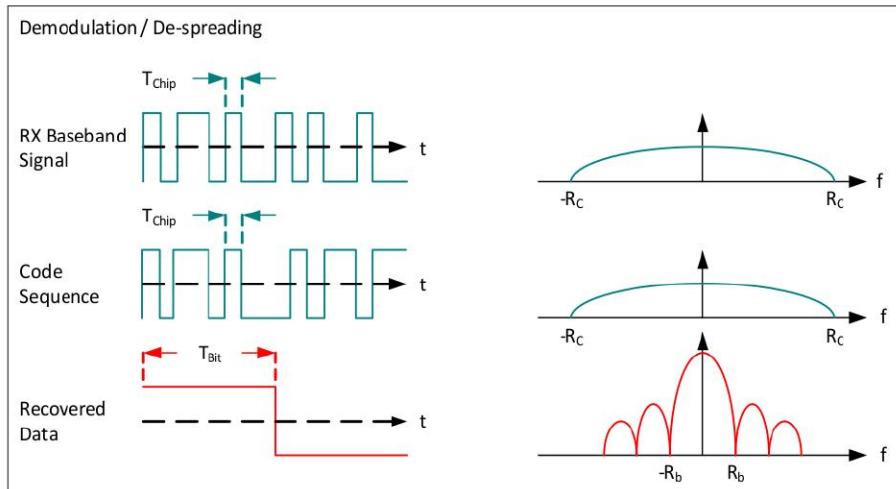


Figure 7.2 Démodulation avec étalement du signal

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**.

$$G_p = 10 \cdot \log_{10}(R_c/R_b) \text{ (dB)}$$

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.

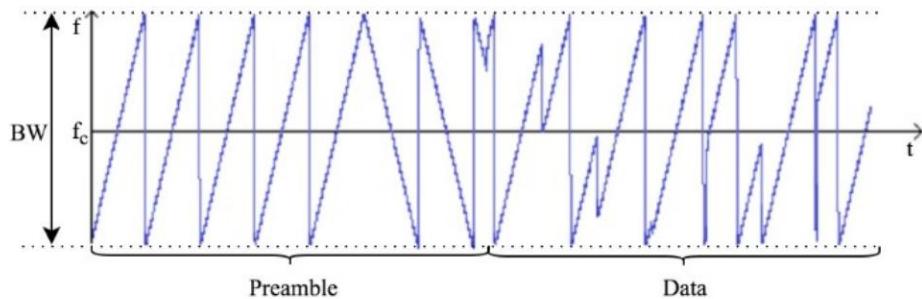


Figure 7.3 Variation de fréquence dans la bande passante du signal

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 constituent un **symbole**.

Le nombre de symboles possibles est donc égal à **M=2^{SF}**.

Par conséquent, la relation entre le débit binaire **R_b** et le débit de symboles **R_s** peut être écrite comme suit :

$$R_b = SF * R_s$$

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:

$$R_b = SF * BW/2^{SF} \text{ bits/sec}$$

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 :

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

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

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

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

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

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.

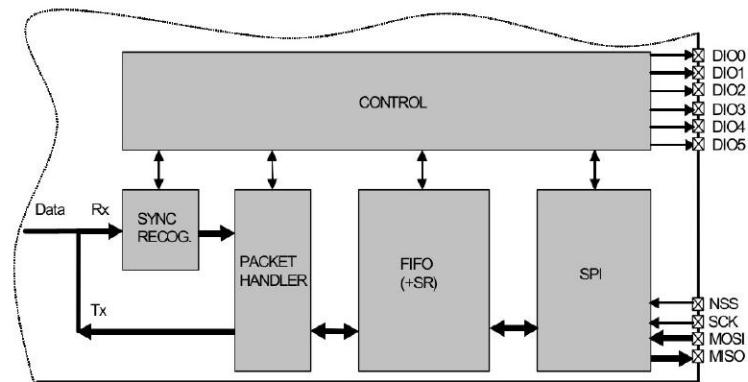


Figure 7.5 Modem SX1278 fonctionnant en mode paquet avec taille variable des paquets

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.

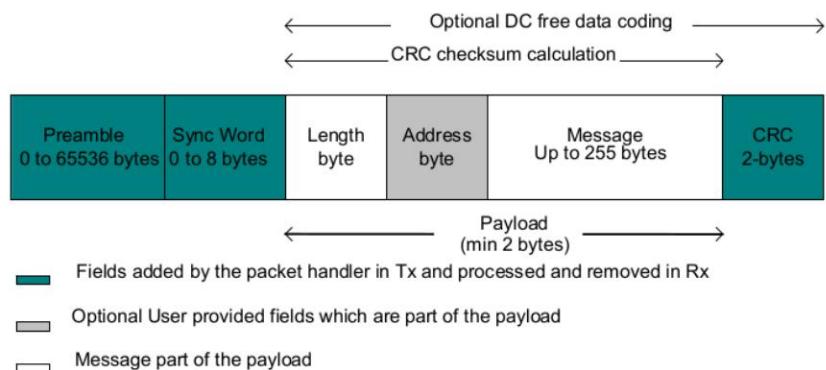


Figure 7.6 Format du paquet de la taille variable

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 modifiée, 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 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

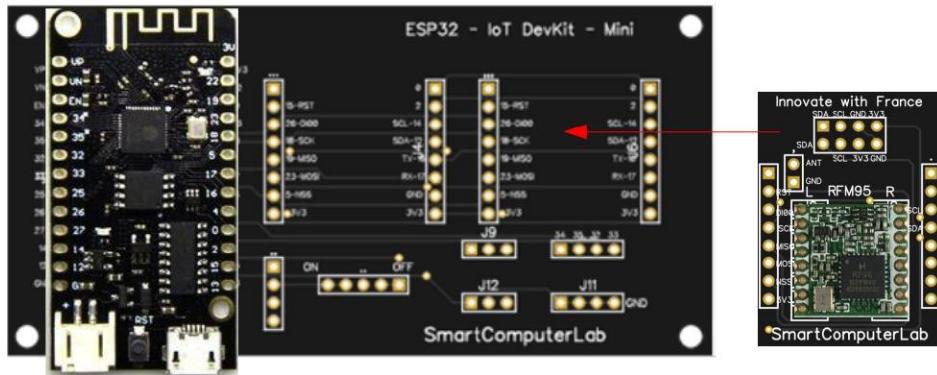
Nos exemples de programmation sont basés sur la bibliothèque `LoRa.h` développée pour les modules RFM95/96. Le câblage avec le bus SPI dépend du type de carte ESP32.

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 DIO    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();

    // from Print
    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("Callback - 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.

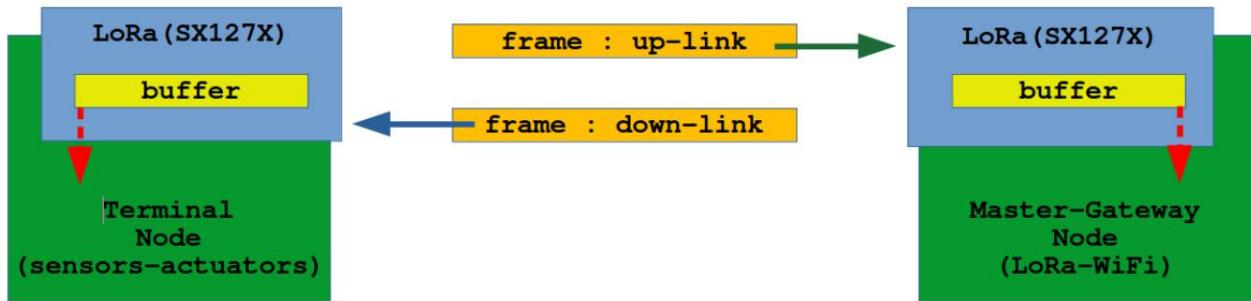


Figure 7.7 Canal de communication LoRa bidirectionnel entre un nœud Terminal et Maître

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 DIO0   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, DIO0);
  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 DIO0    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, DIO0);
  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 DIO0    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, DIO0);
  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("Callback - 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 DIO0   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, DIO0);
  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 DIO0   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, DIO0);
    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++;
    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 DIO0    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, DIO0);
  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 DIO0    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, DIO0);
    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. Expérimitez 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 DIO0    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, DIO0);
    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, 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);
}

// Terminal IQ mode
#ifndef 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
#ifndef 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!");
    }
}
```

Table des matières

Lab 7.....	1
Communication LoRa (Long Range) au niveau de la couche physique avec le modem SX127X.....	1
7.1 Liaison radio LoRa.....	2
7.1.1 Communication avec un spectre étalé.....	2
7.2 LoRa communication with SX1276/78 modem.....	6
7.2.1 LoRa - Packet Mode.....	6
7.2.2 LoRa - format de paquet de longueur variable (SX1276/78).....	6
7.2.2.1 Préambule.....	7
7.2.2.2 En-tête (<i>header</i>).....	7
7.3 Programmation LoRa (SX1276/78) avec bibliothèque LoRa.h.....	8
7.3.1 Configuration des paramètres de liaison physique.....	10
7.3.1.1 Paramètres de modulation et de contrôle des erreurs.....	10
7.3.1.2 Création et envoi des paquets LoRa (trames).....	10
7.3.1.3 Réception des paquets LoRa (trames).....	10
7.3.1.4 Protéger le canal de communication.....	11
7.4 Nœuds émetteurs et récepteurs simples.....	12
7.4.1 Expéditeur simple.....	12
7.4.2 Récepteur simple.....	13
7.4.3 Récepteur simple avec fonction de rappel (<i>callback</i>).....	14
7.5 Communication en mode duplex.....	15
7.5.1 Duplex avec la fonction parsePacket().....	15
7.5.1.1 Code complet pour la communication LoRa duplex avec parsePacket().....	15
7.5.1.2 Duplex simple avec rappel.....	16
7.6 Communication LoRa simple en liaison montante et descendante avec inversion IQ.....	18

7.6.1 Le code du Terminal avec inversion IQ à la réception.....	18
7.6.2 Le code du Maître avec inversion IQ lors de la transmission.....	19
7.7 À faire:.....	20
7.8 Annexe – LoRa_Para.h.....	21
7.8.1 Terminal code avec IQ inversion et fichier LoRa_Para.h.....	23
7.8.2 Gateway code avec IQ inversion et fichier LoRa_Para.h.....	24