Bureau d'étude Dimensionnement d'interface radio pour réseaux mobiles - Dimensionnement d'un réseau LoRa pour une application de "stationnement intelligent"

Eléments de programmation

Le but de ce document vous fournit quelques éléments pour rapidement démarrer vos applications sur Arduino MEGA 2560 pour les mesures sur terrain de portée radio d'un réseau LoRa. Ce document décrit brièvement l'environnement Arduino IDE, la structure d'un programme et les librairies principales pour la caractérisation de la portée radio d'un réseau LoRa.

I. Programmation sur Arduino

Les cartes Arduino sont basées sur des microcontrôleurs de la famille ATMEGA développés par ATMEL. Leur programmation est basée sur un langage très proche du C++, utilisant des fonctions haut niveau pour accéder aux différentes ressources matériels du microcontrôleur. Dans cette partie, la structure d'un programme, l'environnement de développement Arduino IDE et quelques librairies natives indispensables à ce BE sont présentées succinctement. Pour plus d'informations, reportez-vous au guide de référence du langage pour Arduino, disponible à l'adresse https://www.arduino.cc/en/main/software, ou directement depuis le menu "Aide > Référence" d'Arduino IDE.

1. Structure d'un programme

Les programmes Arduino, appelés **sketch**, sont écrits dans un langage très proche de C++ et ont une extension .ino. Leur structure est toujours la même et présente deux parties : **setup** et **loop**, comme le montre l'exemple ci-dessous :

```
//import éventuel de librairie
#include <my_lib.h>
//définition éventuelle de constante, variables globales
uint8_t my_var = 1;
//partie 1 : setup (initialisation)
void setup() {
  // put your setup code here, to run once:
/*
*
*/
}
//partie 2 : loop (boucle infinie)
```

```
void loop() {
//put your main code here, to run repeatedly:
/*
*
*/
}
```

La partie **setup** est la première partie du programme. Elle est exécutée en premier et une seule fois. Elle est donc dédiée à l'initialisation des différentes parties de l'application. La partie **loop** s'exécute en boucle une fois la fonction setup exécutée. Elle forme donc le corps de l'application. C'est donc l'équivalent de la fonction main d'un programme C habituel.

Le déroulement de cette fonction va dépendre de stimuli externes. Comme dans n'importe quelle application embarquée développée en C, il existe deux manières de détecter ces stimuli:

- par scrutation périodique de registres et variables (polling). L'utilisation de l'instruction delay(time), où time est exprimé en ms, permet de définir précisément les périodes de polling.
- par interruptions (Interrupt Service Routines ISR). Voir l'instruction attachInterrupt pour associer une ISR à une entrée déclenchant des interruptions externes (digitalPinToInterrupt()).

La compilation du programme, la création et le téléversement des fichiers objets enregistrés en mémoire Flash du microcontrôleur sont assurés par l'environnement de développement Arduino IDE.

2. Description d'Arduino IDE

L'environnement de développement pour Arduino est assez rudimentaire, mais présente les éléments suffisants pour développer rapidement des applications embarquées, ce qui est l'objectif revendiqué d'Arduino. Lorsqu'aucun sketch n'a été chargé, l'écran ci-dessous s'ouvre lors du lancement d'Arduino IDE. Les commandes sont disponibles dans le menu et accessible dans la barre de menu. L'écran fait apparaître la zone d'édition du code et une zone de message qui indiquera les éventuelles erreurs de compilation et les étapes de téléversement.

De nombreux exemples de sketch sont disponibles dans le menu "Fichier > Carnet de croquis", qui peuvent aider lors de la prise en main d'une carte Arduino ou d'une nouvelle librairie. Lorsque un sketch est prêt, enregistrez-le et compilez-le avec la commande "Croquis > Vérifier / Compiler", ou le bouton  ou le raccourci clavier CTRL+R. Dès qu'il est compilé, les fichiers objets pour le microcontrôleur peuvent être préparés et téléversés en cliquant sur "Croquis > Téléverser", ou le bouton  ou le raccourci clavier CTRL+U. A noter qu'en cliquant sur cette commande, le programme est systématiquement recompilé.

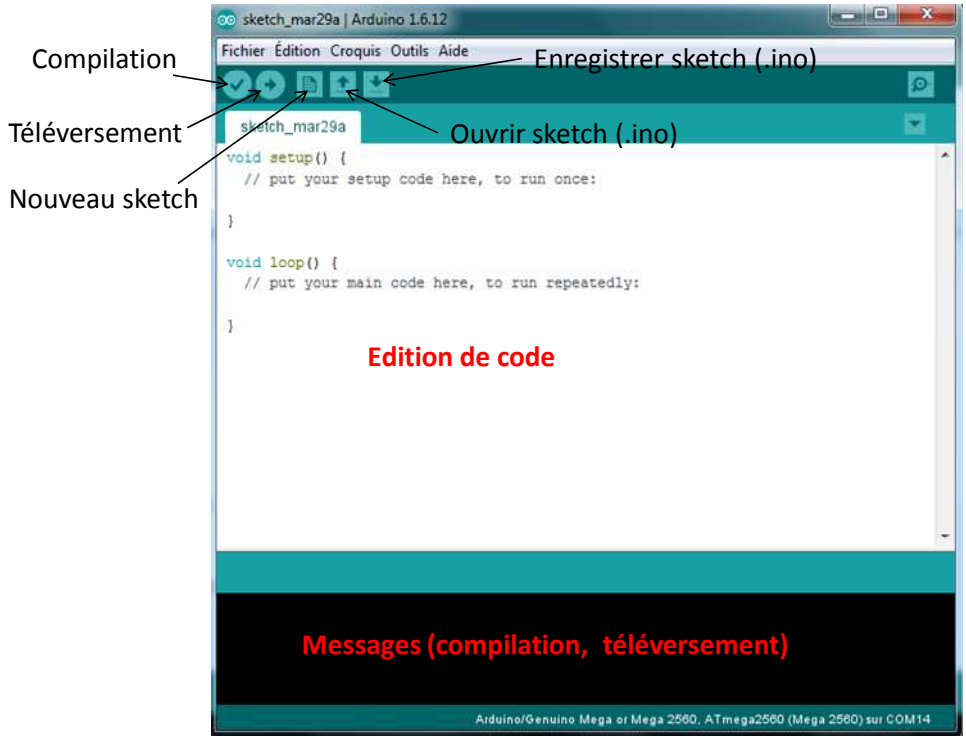

Figure 1 - Fenêtre Arduino IDE avec un sketch vide

Avant de téléverser le programme dans le microcontrôleur, il est nécessaire de connecter une carte Arduino sur un port USB et de s'assurer que Arduino IDE l'a bien détecté. Quelques étapes sont nécessaires pour aider l'environnement Arduino IDE à détecter automatiquement les cartes Arduino connectées sur les ports USB :

1. dans le menu "Outils > Type de cartes", sélectionnez le type de carte connectée. Dans ce BE, il faudra sélectionner "Arduino/Genuino Mega or Mega 2560"
2. dans le menu "Outils > Processor", sélectionnez le type de microcontrôleur monté sur la carte Arduino. Dans ce BE, sélectionnez "ATmega2560 (Mega 2560)".
3. dans le menu "Outils > Port", sélectionnez le port série (USB) sur lequel la carte est connectée.

Une fois le programme téléversé, il est possible de transmettre ou de recevoir des données du microcontrôleur par liaison série via le port USB de la carte. L'environnement Arduino IDE propose un outil de type Hyperterminal, dans le menu "Outils > Moniteur série" ou à l'aide du raccourci clavier CTRL + MAJ + M.

3. Quelques éléments de langage et librairies

Se référer au guide de référence du langage pour Arduino, disponible à l'adresse https://www.arduino.cc/en/Reference/HomePage pour tout détail sur les instructions et les librairies natives d'Arduino.

Ci-dessous, quelques éléments à propos des instructions et bibliothèques que vous utiliserez dans ce bureau d'étude. Elles ne concernent que les périphériques de type port de communication.

a) Gestion des GPIO

On s'intéresse ici aux entrées-sorties digitales à usage général (non associées à un périphérique spécifique). Il est nécessaire de définir si cette broche est une entrée ou une sortie, à l'aide de l'instruction :

```
pinMode(numéro_broche,mode)
```

où numéro_broche est le numéro de la broche de la carte Arduino et mode = 'INPUT' ou 'OUTPUT'

Une écriture de l'état digital d'une sortie est assurée par l'instruction :

```
digitalWrite(numéro_broche,valeur)
```

où numéro_broche est le numéro de la broche de la carte Arduino et valeur = 'HIGH' ou 'LOW'

Une lecture de l'état digital d'une entrée est assurée par l'instruction :

```
digitalRead(numéro_broche
```

où numéro_broche est le numéro de la broche de la carte Arduino. Elle renvoie un entier, égal à 'HIGH' ou 'LOW'

b) Port série ou UART

Afin de communiquer avec un ordinateur ou d'autres équipements électroniques, la carte Arduino MEGA intègre quatre ports série ou UART. Ceux utilisent deux pins TX/RX utilisant des niveaux TTL (0-5V sur la carte Arduino MEGA). Ci-dessous, la liste des quatre ports série et le nom associé :

- `Serial` : dédié au port USB de la carte (voir partie I.3.c)
- `Serial1` : associé aux pins 18 (TX1) et 19 (RX1)
- `Serial2` : associé aux pins 16 (TX2) et 17 (RX2)
- `Serial3` : associé aux pins 14 (TX3) et 15 (RX3)

La liaison `Serial1` sera utilisée entre la carte Arduino MEGA 2560 et le module GPS dans le bureau d'étude. Ci-dessous, une liste des instructions associées à chacun de ces ports qui vous seront le plus utiles durant le bureau d'étude :

Instructions	Description
`Begin(long data_rate)`	Initialise le débit (variable data rate) d'une liaison série.
`End()`	Suspend une liaison série, les pins associés redeviennent des GPIO.
`Print(val,format)`	Transmet des données sous la forme de texte ASCII. La variable val contient les données à transmettre (n'importe quel type). La variable format est optionnelle et spécifie le format d'affichage des données transmises (HEX, DEC, BIN). L'ajout d'uen tabulation se fait à l'aide du caractère "\t", un retour charriot à l'aide de "\r" et une nouvelle ligne à l'aide de "\n". L'instruction renvoie le nombre d'octets transmis (type long).

`Println`	Identique à l'instruction print, excepté le retour charriot et la création d'une nouvelle ligne à l'issue de l'instruction.
`Available`	Indique le nombre d'octets reçus et disponibles sur un port série pour une lecture.
`Read()`	Lit les données reçus sur un port série. Revoie le premier octet reçu (type int).
`parseInt()`	Renvoie le prochain entier valide dans un flux de données série entrant (Stream)

Remarque : vous ne pouvez pas transmettre directement des séries de caractères (strings) à l'aide des instructions `print` ou `println`. Il est nécessaire de les empaqueter à l'aide de l'instruction `F("my_string")`. Exemple : `Serial1.println(F("Configuration reussie !"))`.

c) Port USB

Une carte Arduino dispose de plusieurs ports série (voir partie I.3.b). L'un d'entre eux est connecté pour le port USB de la carte et est réservé aux communications avec un ordinateur. Sur les cartes Arduino MEGA ou UNO, ce port est associé aux pins 0 et 1 (RX0 et TX0) et est dénommé `Serial`. On retrouve les mêmes instructions associées à ce port que celles des autres ports série, décrites dans la partie I.3.b.

d) Port SPI

La liaison SPI (Serial Peripheral Interface) est un bus série couramment utilisé pour des liaisons entre circuits montés sur une même carte électronique. Il s'agit d'une liaison de type full-duplex et est du type maître-esclave. Elle est basée sur 4 fil:

- SCLK : horloge cadençant les transmissions maître-esclave et esclave-maître. Elle est générée par le maître
- SS : Slave Selection, signal généré par le maître.
- MOSI : Master Out Slave In, transmission maître vers l'esclave
- MISO : Master In Slave Out, transmission esclave vers le maître

Un seul port SPI existe sur la carte Arduino MEGA 2560. Les broches SCLK, MISO et MOSI sont disponibles :

- soit sur les pins 50 à 53 de la carte Arduino
- soit sur le connecteur ICSP, où seuls les fil SCLK, MOSI et MISO sont disponibles. Le signal SS peut être délivré par n'importe quel sortie digitale.

Arduino IDE possède une librairie native `SPI` contenant l'ensemble des fonctions nécessaires pour activer, configurer la liaison SPI et transférer des données. La liaison entre la carte Arduino et le module LoRa SX1272 étant assurée par un bus SPI, la librairie sx1272_INSAT utilise les ressources de cette librairie. Vous n'aurez normalement pas besoin d'utiliser cette librairie. Néanmoins, vous pouvez trouver des détails à l'adresse suivante : https://www.arduino.cc/en/reference/SPI.

II. Librairie Adafruit_GPS

La librairie Adafruit_GPS contient l'ensemble des fonctions nécessaires à l'initialisation du module GPS Adafruit Ultimate GPS Breakout v3 et à la réception des coordonnées GPS. La librairie est disponible sur https://learn.adafruit.com/adafruit-ultimate-gps/downloads. Pour plus d'informations sur ce module et son interfaçage avec une carte Arduino, vous pouvez vous référer au lien suivant : https://cdn-learn.adafruit.com/downloads/pdf/adafruit-ultimate-gps.pdf. De nombreux exemples de programmes sont fournis avec cette librairie.

La communication entre la carte Arduino MEGA 2560 et le module GPS est assurée par la liaison série Serial1 (TX1 et RX1). Par défaut, le débit de transfert est de 9600 bauds. Le format des données transmises par le module GPS est NMEA0183 (standard de définition des trames transmises par les équipements GPS). Le module est capable de déterminer sa position jusqu'à 5 fois par secondes.

Les tableaux ci-dessous listent les variables et les fonctions de la classe Adafruit_GPS, associée au module GPS. Le nom GPS est utilisée comme identifiant de la classe sx1272_INSAT.

Variables	Description
`hour, minute, seconds, year, month, day, milliseconds`	Temps universel relevé par le module GPS
`fix, fixquality`	Critères de qualité du signal
`satellites`	Nombre de satellites délivrant un signal valide
`latitude, longitude`	Latitude et longitude en degrés
`latitudeDegrees, longitudeDegrees`	Latitude et longitude en degrés compatibles avec Google Maps
`speed`	Vitesse en nœuds (knots)
`altitude`	Altitude

Fonctions	Description
`begin(uint32_t baud)`	Initialise la liaison série, la variable baud indique le débit de symboles (9600 Bds est une valeur suffisante).
`sendCommand(const char *str)`	Envoie une commande au module GPS via la liaison série. Ci-dessous, les différentes commandes de configuration sont détaillées.
`Read()`	Lit les données série issues du GPS. Renvoie un caractère.
`newNMEAreceived()`	Indique si une trame au format NMEA a été reçu, donc si on est prêt à lire les données issues du module GPS. Renvoie un booléen.
`parse(char *nmea)`	Extrait les différents termes (coordonnées, temps) contenus dans la trame NMEA reçu.

Ci-dessous, quelques-unes des commandes de configuration définies dans la librairie:

- PMTK_SET_NMEA_OUTPUT_RMCGGA: fixe les trames de sortie NMEA au format RMC (recommended minimum) et GGA (fix data including altitude)
- PMTK_SET_NMEA_OUTPUT_RMCONLY: fixe les trames de sortie NMEA au format RMC (recommended minimum) seulement
- PMTK_SET_NMEA_UPDATE_1HZ/5Hz: indique combien de fois par seconde les trames NMEA sont transmises sur la liaison série (ici 1 Hz ou 5 Hz)
- PMTK_API_SET_FIX_CTL_1HZ: indique combien de fois par seconde les données de position GPS sont mises à jour (ici 1 Hz ou 5 Hz)

III. Librairie sx1272_INSAT

La librairie sx1272_INSAT contient l'ensemble des fonctions nécessaires à l'initialisation du module radio SX1272 Libellium, à la transmission et la réception des données. Cette librairie est directement issue de la librairie SX1272.h développée par Libelium, mais adaptée aux besoins du bureau d'étude et assurant une compatibilité avec la carte Arduino MEGA 2560.

La communication entre la carte Arduino MEGA 2560 et le module radio LoRa SX1272 est assurée par une liaison SPI. Physiquement, les signaux MISO, MOSI, SCLK sont transmis via le connecteur ICSP de la carte Arduino MEGA. Le signal SS est transmis via la pin 2 de la carte Arduino MEGA.

Les tableaux ci-dessous listent les variables et les fonctions de la classe sx1272_INSAT, associée au module radio LoRa. Le nom `sx1272` sera utilisée comme identifiant de la classe sx1272_INSAT.

Variables	Description
`_bandwidth`	Bande passante, prend les valeurs prédéfinies BW_125, BW_250 ou BW_500.
`_codingRate`	Taux de codage, prend les valeurs prédéfinies CR_5, CR_6, CR_7 ou CR_8.
`_spreadingFactor`	Facteur d'étalement, prend les valeurs prédéfinies SF_6 à SF_12.
`_channel`	Fréquence centrale du canal (voir sx1272_INSAT.h pour la liste des valeurs prédéfinies). La variable contient directement le contenu du registre 32 bits FRF du SX1272.
`_header`	Header explicite ou implicite, prend les valeurs prédéfinies HEADER_ON ou HEADER_OFF.
`_CRC`	Présence ou non du CRC, prend les valeurs prédéfinies CRC_ON ou CRC_OFF.
`_modem`	Type de modem, prend les valeurs prédéfinies LORA ou FSK.
`_power`	Puissance de sortie (voir sx1272_INSAT.h pour la liste des valeurs prédéfinies). La variable contient directement les 4 bits de poids faible du registre RegPaConfig du SX1272.
`_packetNumber`	Numéro du paquet transmis.
`_reception`	Indique si un paquet valide a été reçu, prend les valeurs prédéfinies CORRECT_PACKET ou INCORRECT_PACKET.
`_retries`	Indique le nombre de tentative ratée de transmission d'un paquet.
`_maxRetries`	Indique le nombre maximale de retransmission d'un paquet.

Fonctions	Description
`ON()`	Initialise la liaison SPI et allume le module radio LoRa SX1272. Renvoie un entier indiquant la bonne exécution de la fonction.
`OFF()`	Suspend la liaison SPI et éteint le module radio LoRa SX1272.
`readRegister(byte address)`	Lit le registre interne du module radio LoRa SX1272 à l'adresse donnée par la variable address. Renvoie le contenu du registre sous la forme d'un octet
`writeRegister(byte address, byte data)`	Ecrit l'octet data dans le registre interne du module radio LoRa SX1272 à l'adresse donnée par la variable address.
`SetLORA()`	Met le modem en mode LORA. Renvoie un entier indiquant la bonne exécution de la fonction.
`setHeaderON(), setHeaderON()`	Header en mode explicite ou implicite. Renvoie un entier indiquant la bonne exécution de la fonction.
`getHeader()`	Indique le mode header. Renvoie un entier indiquant la bonne exécution de la fonction.
`setCRC_ON(), setCRC_OFF()`	Champ CRC actif ou non. Renvoie un entier indiquant la bonne exécution de la fonction.
`getCRC()`	Indique si le champ CRC est actif ou non. Renvoie un entier indiquant la bonne exécution de la fonction.
`setSF(uint8_t spr)`	Fixe le facteur d'étalement, donné par la variable spr. Renvoie un entier indiquant la bonne exécution de la fonction.
`getSF()`	Indique la valeur du facteur d'étalement. Renvoie un entier indiquant la bonne exécution de la fonction.
`setBW(uint16_t band)`	Fixe la bande passante, donné par la variable band. Renvoie un entier indiquant la bonne exécution de la fonction.
`getBW()`	Indique la valeur de la bande passante. Renvoie un entier indiquant la bonne exécution de la fonction.
`setCR(uint8_t cod)`	Fixe le taux de codage, donné par la variable cod. Renvoie un entier indiquant la bonne exécution de la fonction.
`getCR()`	Indique la valeur du taux de codage. Renvoie un entier indiquant la bonne exécution de la fonction.
`setChannel(uint32_t ch)`	Fixe la fréquence centrale, donné par la variable ch. Renvoie un entier indiquant la bonne exécution de la fonction.
`getChannel()`	Indique la valeur de la fréquence centrale. Renvoie un entier indiquant la bonne exécution de la fonction.
`setPower(uint8_t pow)`	Fixe la puissance de sortie, donné par la variable pow. Renvoie un entier indiquant la bonne exécution de la fonction.
`getPower()`	Indique la valeur de la puissance de sortie. Renvoie un entier indiquant la bonne exécution de la fonction.
`setPreambleLength (uint16_t l)`	Fixe la longueur du préambule en nombre de symboles, donné par la variable l. Renvoie un entier indiquant la bonne exécution de la fonction.
`getPreambleLength()`	Indique la longueur du préambule. Renvoie un entier indiquant la bonne exécution de la fonction.
`getPayloadLength()`	Indique la longueur du payload. Renvoie un entier indiquant la bonne exécution de la fonction.
`setNodeAddress(uint8_t addr)`	Fixe l'adresse du module LoRa donnée par la variable addr. Renvoie un entier indiquant la bonne exécution de la fonction.
`getNodeAddress()`	Indique l'adresse du module LoRa. Renvoie un entier indiquant la bonne exécution de la fonction.
`getSNR()`	Indique la valeur du SNR lorsqu'une trame valide a été reçue.

	Renvoie un entier indiquant la bonne exécution de la fonction.
`getRSSI()`	Indique la valeur du RSSI courant. Renvoie un entier indiquant la bonne exécution de la fonction.
`getRSSIpacket()`	Indique la valeur du RSSI lors de la réception du dernier paquet. Renvoie un entier indiquant la bonne exécution de la fonction.
`setRetries(uint8_t ret)`	Fixe le nombre maximale de tentative de ré-émission d'un paquet donné par la variable ret. Renvoie un entier indiquant la bonne exécution de la fonction.
`receivePacketTimeout (uint32_t wait)`	Attente d'une réception valide d'un paquet pendant une durée maximale indiquée par la variable wait (exprimée en ms). Renvoie un entier indiquant la bonne exécution de la fonction.
`receivePacketTimeoutACK (uint32_t wait)`	Même chose que `receivePacketTimeout`, mais le module répond par un accusé de réception. Renvoie un entier indiquant la bonne exécution de la fonction.
`receiveAll(uint32_t wait)`	Même chose que `receivePacketTimeout`, mais le module ne filtre plus les messages qui ne lui sont pas adressées. Renvoie un entier indiquant la bonne exécution de la fonction.
`sendPacketTimeout (uint8_t dest, char *payload, uint32_t wait);`	Transmission d'un paquet de données indiqué par le pointeur payload et adressé à dest. La variable wait indique le temps maximum de transmission avant abandon. Renvoie un entier indiquant la bonne exécution de la fonction.
`sendPacketTimeoutACK (uint8_t dest, char *payload, uint32_t wait);`	Même chose que `sendPacketTimeout`, mais le module répond par un accusé de réception. Renvoie un entier indiquant la bonne exécution de la fonction.

Les paquets à transmettre et à recevoir sont spécifiés par les variables packet_sent et packet_receive. Il s'agit d'une structure de type pack, composé des champs suivants :

Champs	Description
`uint8_t dst`	Adresse de destination
`uint8_t src`	Adresse source
`uint8_t packnum`	Numéro du paquet
`uint8_t length`	Longueur du paquet
`uint8_t data[MAX_PAYLOAD]`	Données constituant le payload
`uint8_t retry`	Numéro de retentative de transmission