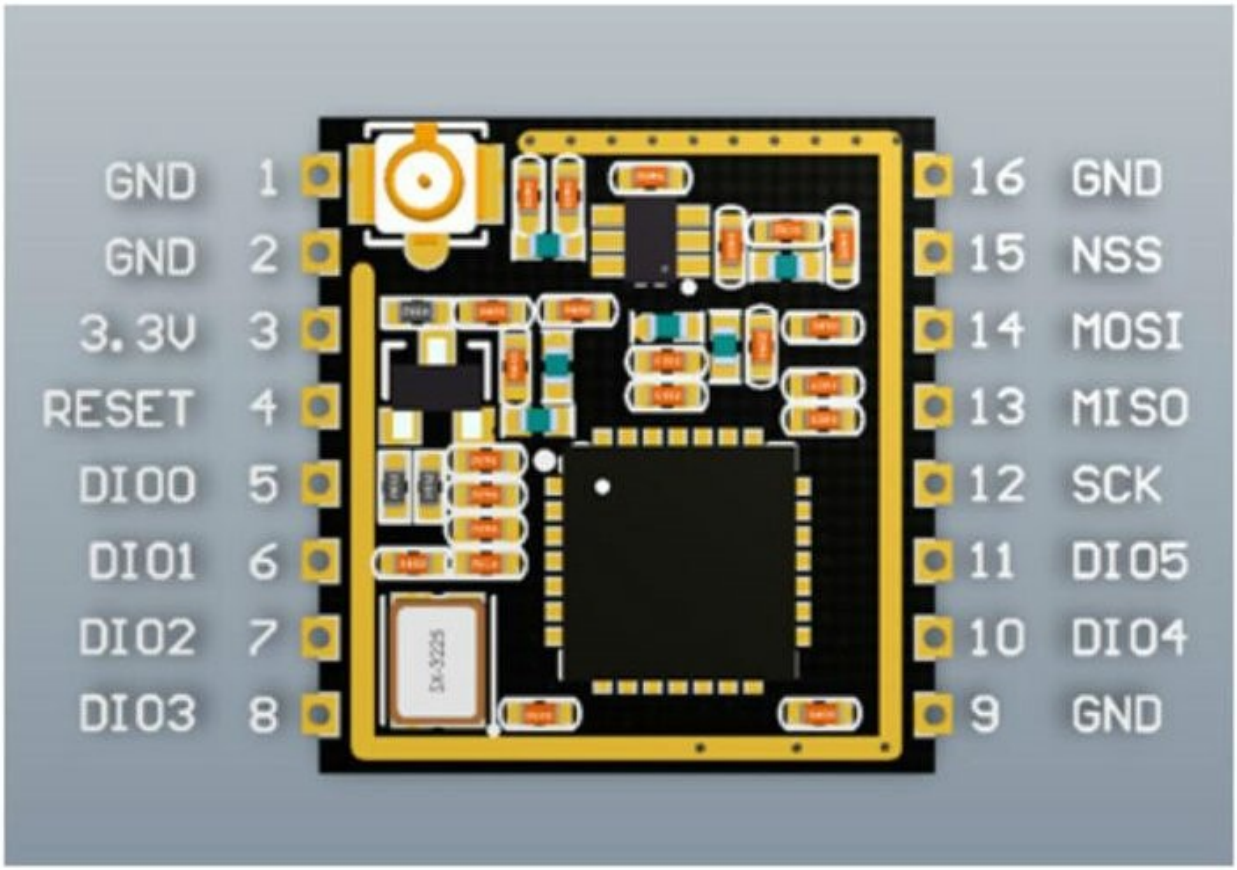
1. Hardware



MODULE LORA 02 SX1278 433 MHZ

Le schéma structurel connecte le modem RA02 de la façon suivante,

GPIO 0 → RESET
GPIO 22 → DIO 0 (configurée avec RegDioMapping1 0x40)
GPIO 17 → DIO 1
GPIO 23 → DIO 2
GPIO 27 → DIO 3
CE0 → NSS utilisée pour sélectionner un périphérique SPI.
MOSI → MOSI
MISO → MISO
CLK → SCK

DIO 4 et DIO 5 ne sont pas connectés.

Le channel de la fonction `wiringPiSPISetup` détermine quelle broche CE sera utilisée pour la communication SPI. channel 0 correspond à la broche CE0 et channel 1 correspond à la broche CE1.

2. Initialisation de la bibliothèque

Avec pigpio

```
12 if (gpioInitialise() < 0)
        {
                printf("Pigpio init error\n");
                return -1;

        }
```

Pigpio est une bibliothèque utilisée pour le contrôle des GPIO (General Purpose Input/Output) sur les Raspberry Pi. Elle est conçue pour fonctionner avec **un seul démon (pigpiod)** en arrière-plan, ce qui rend son utilisation avec plusieurs processus simultanés très délicate.

Avec WiringPi

Il n’y a pas de démon générale pour la biblio WiringPI,

Toutefois assurez-vous que chaque processus contrôle des broches différentes ou utilise des ressources bus SPI ou I2c partagées de manière coordonnée pour éviter les conflits.

```
 // Initialise WiringPi
  if (wiringPiSetupGpio() == -1) {
     printf("Échec de l'initialisation de WiringPi\n");
     return -1;
  }
```

3. Lora_reset

```
18 lora_reset(modem->eth.resetGpioN);
```

modem→eth.resetGpioN est lue dans le fichier de configuration config.ini **valeur 0**

sur le schéma le GPIO0 est effectivement connecté sur la broche reset du RA02

```
400  void lora_reset(unsigned char gpio_n){
        gpioSetMode(gpio_n, PI_OUTPUT);
        gpioWrite(gpio_n, 0);
        usleep(100);
        gpioWrite(gpio_n, 1);
        usleep(5000);
 406    }
```

Avec wirinPI la fct équivalente est :

```
void lora_reset(unsigned char gpio_n){

      pinMode (gpio_n, OUTPUT);
      digitalWrite (gpio_n, LOW);
      usleep(100);
      digitalWrite (gpio_n, HIGH);
      usleep(5000);

}
```

4. Ouverture du bus SPI

Avec pigpio

```
20    if( (modem->spid = spiOpen(modem->spiCS, 32000, 0))<0 )
                return modem→spid;   // return avec le code de l'erreur
```

modem→spiCS est lue dans le fichier de configuration **valeur 0** représente le **channel**
32000 représente la vitesse d'échange **32KHz speed**
0 représente le mode (données capturées sur le front montant de l'horloge)

modem→spid est le handle qui sera utilisé par les fct de transfert.

Avec WiringPi

la fonction équivalente est

```
// Initialise le canal SPI
      if ((modem->spid  = wiringPiSPISetup(channel, speed)) < 0) {
            // Initialisation échouée
            fprintf(stderr, "Impossible d'initialiser le canal SPI: %s\n",
           strerror(errno));
           return errno;
}
```

Dans WiringPi, la fonction `wiringPiSPISetup` utilise le mode 0 par défaut, ce qui signifie que l'horloge SPI est inactive (au niveau bas) lorsque CS0 (Chip Select 0) est inactif, et les données sont capturées sur le front montant de l'horloge comme le montre le chronogramme suivant.

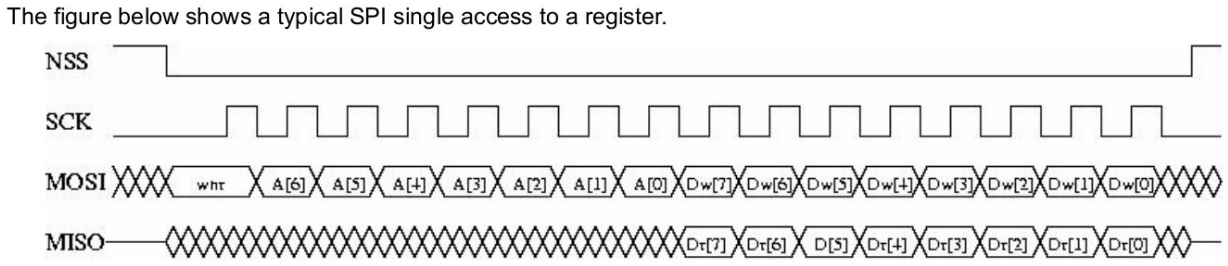


Figure 38. SPI Timing Diagram (single access)

MOSI est généré par le rpi qui envoie deux octets
le premier l'adresse du registre à lire ou écrire avec le bit **whr** qui vaut 1 pour l'accès en écriture et 0 pour l'accès en lecture.
Le second octet la valeur à écrire ou 0x00 si lecture.
MISO est généré par le RA-02 et donne dans le deuxième octet la valeur du registre.

5. fonction d’écriture d’un registre

Avec pigpio, la fonction `spiXfer` est particulièrement utile pour les communications SPI bidirectionnelles, où des données sont envoyées et reçues simultanément.

lora_reg_write_byte

```
431  int lora_reg_write_byte(int spid, unsigned char reg, unsigned char byte){

        char rx[2], tx[2];
        tx[0]=(reg | 0x80);    // Bit whr positionné à 1 pour accés en écriture
        tx[1]=byte;

        rx[0]=0x00;
        rx[1]=0x00;

        return spiXfer(spid, tx, rx, 2);

}
```

Il faut donc remplacer la fonction de pigpio **spiXfer**

spid est le channel 0 ici car CE0 est connecté à NSS
tx Un pointeur vers le buffer de transmission (les données à envoyer).
rx Un pointeur vers le buffer de réception (les données reçues)
2 Le nombre d’octets à transférer.

La fonction retourne le nombre de bytes transférés (c'est-à-dire envoyés et reçus) ou une valeur négative en cas d'erreur.

Avec WiringPi

La fonction équivalente est **wiringPiSPIDataRW** à la différence près que le buffer d’émission et de réception sont les mêmes. Après une émission le buffer contient les données reçues après l'appel à `wiringPiSPIDataRW`.

En utilisant WiringPi, le transfert SPI est assez simple et direct, avec une fonction unique pour effectuer les transferts de données en mode lecture/écriture.

```
Int retour = wiringPiSPIDataRW( spid, tx, 2) ;

return retour ;
```

6. fonction de lecture d’un registre

Ligne 412

```
unsigned char lora_reg_read_byte(int spid, unsigned char reg){

        int ret;
        char rx[2], tx[2];
        tx[0]=reg;
        tx[1]=0x00;

        rx[0]=0x00;
        rx[1]=0x00;

        ret = spiXfer(spid, tx, rx, 2);

        if(ret<0)
                return ret;

        if(ret<=1)
                return -1;

        return rx[1];
}
```

avec WiringPi

```
unsigned char lora_reg_read_byte(int spid, unsigned char reg){

        int ret ;

        char data[2] ;

         data[0] = reg ;
        data[1] = 0x00;

        ret = wiringPiSPIDataRW( spid, data, 2) ;

        if (ret == -1)
                return -1 ;

       return data[1] ;

}
```

7. Gestion des interruptions

La fonction pour gérer la réception d'un paquet de données est appelée sur interruption. Même principe pour gérer la fin de l'émission d'un paquet.

Le fichier de configuration défini l'association de la broche DI0 avec Gpio **dio0GpioN=22**

Dans le code on retrouve ligne 119 la définition d'une fonction pour associer une fonction de rappel appelé par interruption au front montant de gpio_n.

lora_set_rxdone_dioISR(int gpio_n, rxDoneISR func, LoRa_ctl *modem)

le type **rxDoneISR** est définie dans le fichier **lora.h**

```
typedef void (*rxDoneISR)(int gpio_n, int level, uint32_t tick, void *userdata);
```

cette ligne crée un type nommé `rxDoneISR` qui est un pointeur vers une fonction prenant quatre paramètres (`int`, `int`, `uint32_t`, `void *`) et ne retournant rien (`void`).

func est la fonction de rappel (callback) qui sera exécutée lorsque l'interruption se produit. Cette fonction doit accepter quatre arguments : le numéro de la broche GPIO, le niveau logique (0 ou 1), le temps de l'interruption, et un pointeur vers des données utilisateur.

La fonction **lora_set_rxdone_dioISR** est appelé ligne 171

171 lora_set_rxdone_dioISR(modem->eth.dio0GpioN, rxDoneISRf, modem)**;**

- La variable **modem.eth.dio0GpioN** est lu dans le fichier de configuration pour être fixée à 22.
- La variable **rxDoneISRf** est une fonction de rappel définie **ligne 195** de type **rxDoneISR**
- modem est un pointeur sur une structure

elle configure la broche **GPIO22** en entrée, puis configure la fonction de rappel **rxDoneISRf** appelé par interruption au front montant avec la ligne de code suivante

```
gpioSetISRFuncEx(gpio_n, RISING_EDGE, 0, func, (void *)modem);
```

La fonction équivalente dans WiringPi pour configurer une interruption est `wiringPiISR`. Toutefois la fonction de rappel doit être de type `void` et ne doit accepter aucun paramètre

La fonction pour gérer la fin d’émission d’un paquet de données est aussi appelée sur interruption.

lora_set_txdone_dioISR(int gpio_n, rxDoneISR func, LoRa_ctl *modem)

elle configure aussi la broche GPIO22 en entrée ! Puis associe la fonction txDoneISRf appelé par interruption au front montant.

Les prototypes des fonctions de rappel sont:

void txDoneISRf(int gpio_n, int level, uint32_t tick, void *modemptr) ;

void rxDoneISRf(int gpio_n, int level, uint32_t tick, void *modemptr) ;

8. Le mappage des broches DIO du RA 02

Dans le code on retrouve deux fonctions pour gérer le mappage du RA02

111 **void lora_set_dio_rx_mapping(int spid)**

115 **void lora_set_dio_tx_mapping(int spid)**

Ces fonctions écrivent dans le registre **REG_DIO_MAPPING** cette constante est définie dans le fichier LoRa.h registre 0x40

#define REG_DIO_MAPPING_1 **0x40**

Ce registre gère le **Mappage pour les broches DIO0 à DIO3**

La page 105 de la documentation du composant SX1278 nous donne la signification des bits

Name (Address)	Bits	Variable Name	Mode	Default value	FSK/OOK Description
RegDioMapping1 (0x40)	7-6	Dio0Mapping	rw	0x00	Mapping of pins DIO0 to DIO5 See Table 18 for mapping in LoRa mode See Table 29 for mapping in Continuous mode SeeTable 30 for mapping in Packet mode
	5-4	Dio1Mapping	rw	0x00	
	3-2	Dio2Mapping	rw	0x00	
	1-0	Dio3Mapping	rw	0x00	
	7-6	Dio4Mapping	rw	0x00	
	5-4	Dio5Mapping	rw	0x00	

le tableau renvoie à la table 18 pour le mode LoRa

Table 18 DIO Mapping LoRa® Mode

Operating Mode	DIOx Mapping	DIO5	DIO4	DIO3	DIO2	DIO1	DIO0
ALL	00	ModeReady	CadDetected	CadDone	FhssChangeChannel	RxTimeout	RxDone
	01	ClkOut	PllLock	ValidHeader	FhssChangeChannel	FhssChangeChannel	TxDone
	10	ClkOut	PllLock	PayloadCrcError	FhssChangeChannel	CadDetected	CadDone
	11	-	-	-	-	-	-

Ce tableau nous dit que les bits 7 et 6 sont utilisés pour configurer la DIO0

si le bit 6 est à 0 DIO0 indique RxDone (réception d'un paquet disponible)
si le bit 6 est à 1 DIO0 indique TxDone (émission d'un paquet terminé)