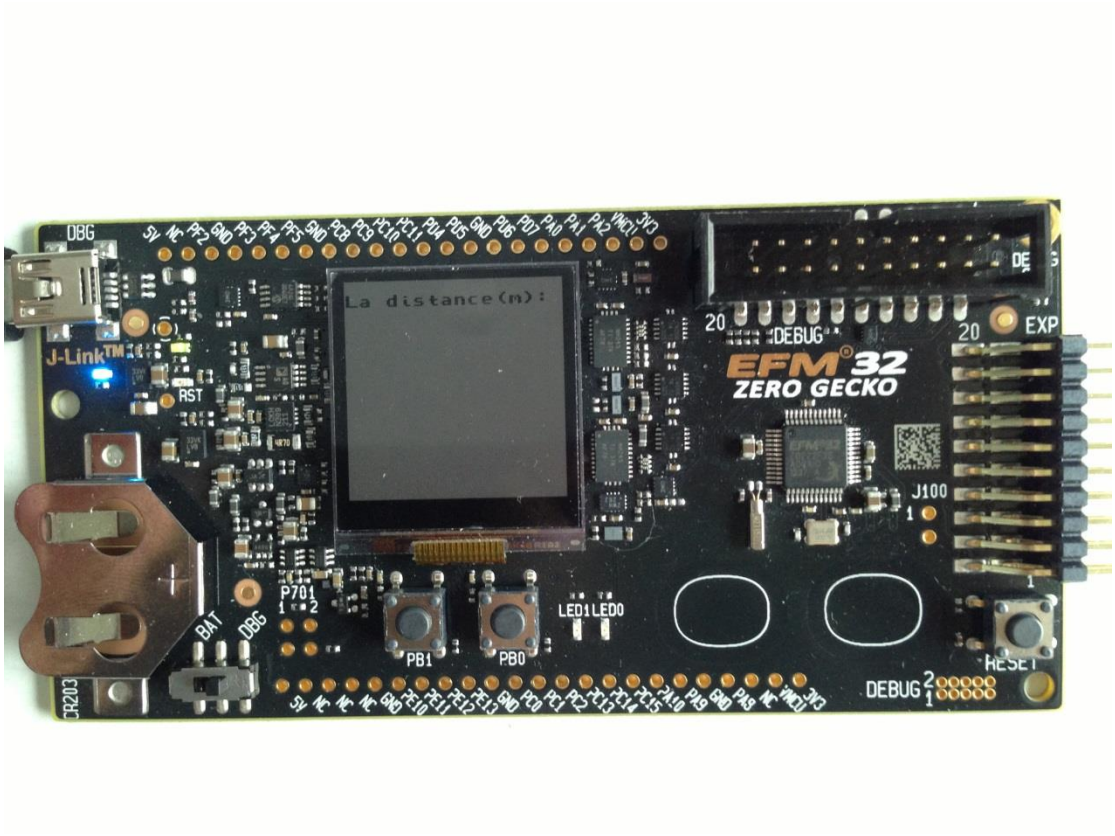
POLYTECH LILLE

Université Lille1 Sciences et Technologies

Rapport de projet IMA4

Mesure de distance par RSSI



CHE XIAODAN

2013-2014

Sommaire

Introduction

Pendant ce semestre, tous les étudiants en IMA informatique-micro électronique-automatique doivent réaliser un projet de 40 heures. J'ai ainsi choisi le projet « mesure de distance par RSSI ».

Bien que je n'ai pas trouvé un camarade intéressé à ce sujet, j'ai mis tous les efforts à réaliser des fonctions autant que possible. J'ai été très intéressé par multi-MCU, mais je n'ai jamais eu la chance de développer quelque chose de moi-même en pratique. Grâce à cette opportunité, j'ai passé beaucoup de temps pour le rechercher et le contrôler. Maintenant, j'ai acquis de nouvelles connaissances sur le micro-contrôleur.

Ce rapport est destiné principalement à vous expliquer les fonctionnements des différents modules utilisés dans le micro-contrôleur pour réaliser ce projet. En même temps, vous montrer la partie que j'ai réalisée, ainsi que les prochaines étapes à réaliser : la partie sur la carte d'émetteur –récepteur.

Présentation du projet

RSSI désigne le signal de réception Strength Indicator. En anglais, il est l'abréviation de "The Receive Signal Strength Indicator". Maintenant, cette technique de mesure est utilisée de plus en plus dans plusieurs domaines. Par exemple, dans le téléphone portable, le bluetooth utilise cette technique. Il mesure la distance en envoyant et en recevant les ondes radio. Puis en fonction de la puissance (dBm), il va nous donner la distance.

L'avantage de ce projet est la petite puissance. Le micro-contrôleur qu'on utilise est plus économe en énergie que tous les autres. En revanche à cause de cette petite puissance, la distance de mesure est bien sûr limitée.

Dans le cadre de mon projet, j'ai réussi à envoyer une donnée sériée via le micro-contrôleur vers l'émetteur -récepteur. Mais parce qu'il y a des problèmes de livraison, quand on achète la carte SX 1211. Du coup, jusqu'à la fin du projet, j'ai eu l'émetteur –récepteur. Dans ce cas, j'ai seulement écrit des fonctions sans la détection.

Cahier des charges

Avant de rentrer au cœur du sujet, il faut définir notre cahier des charges. Dans ce projet, il demande d'utiliser le micro-contrôleur EFM32 à faire l'émetteur-récepteur envoyer et recevoir une donnée, mais il ne force pas d'utiliser quel type d'interface à connecter les deux cartes. Donc, finalement, on se fixe l'interface de SPI (Serial Peripheral Interface). Une liaison SPI est un bus de données série synchrone.

Matériel et outil requis :
Langage: C
Matériels: deux SX1211s (émetteur-récepteur), un EFM32 zerogecko (la carte d'évaluation);
Outil/logiciel pour programmer dans le mirco-contrôleur: Simplicity studio et IAR Embedded Workbench IDE;

Le détail de chaque partie de ce projet va être présenté dans la suite de ce rapport.

Réalisation du projet

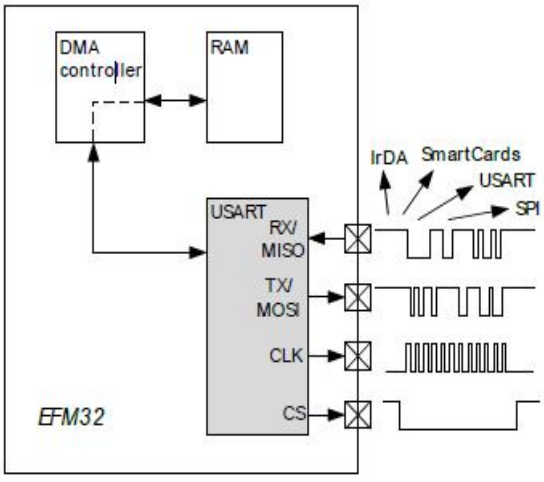
Liste des composants

Micro-contrôleur EFM32

Afin de contrôler l'émetteur-récepteur à envoyer et à recevoir un signal de radio, nous avons besoin d'un micro-contrôleur. Dans ce projet, on a choisi le MCU EFM32.

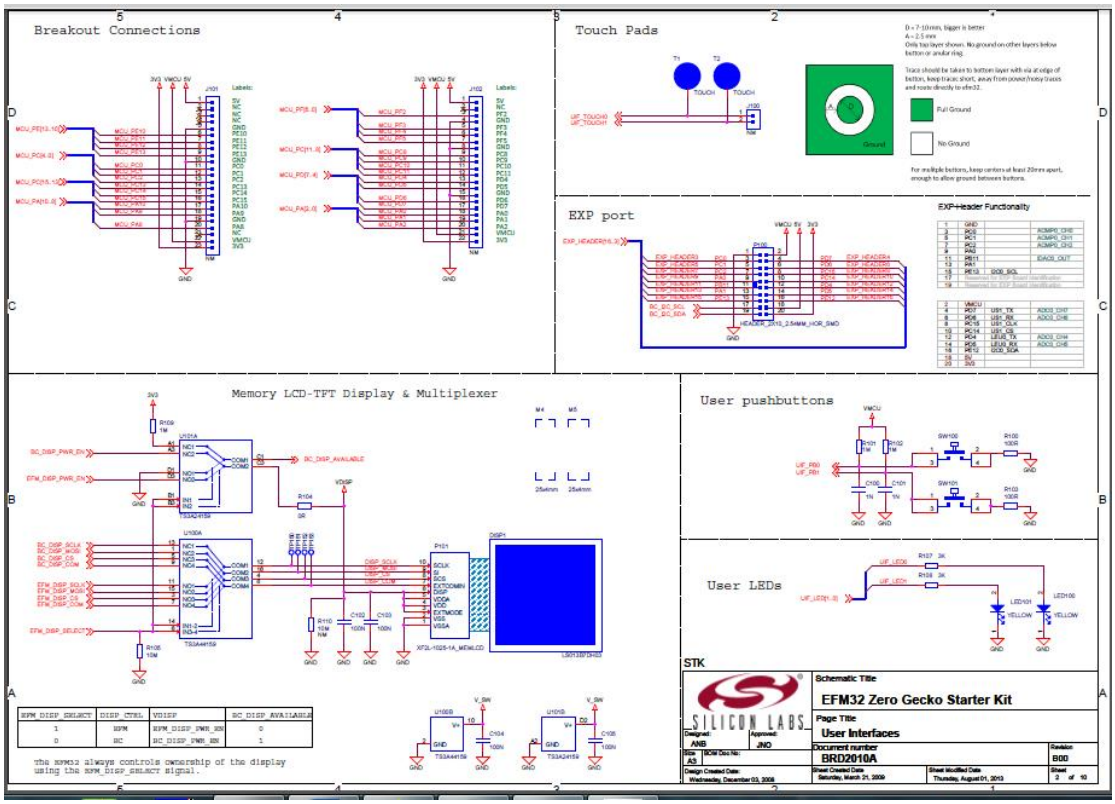
Le EFM32ZG zéro Gecko est le choix idéal pour les exigeants d'applications sensibles à l'énergie avec 8 -, 16 -, et 32-bit. Ces dispositifs sont conçus pour réduire au minimum la consommation d'énergie en réduisant à la fois la puissance et la durée d'activité, au cours de toutes les phases de fonctionnement du microcontrôleur. Cette combinaison unique d'ultra faible consommation d'énergie et la performance du 32 bits ARM Cortex-M0 + processeur, aider les concepteurs à mieux profiter de l'énergie disponible dans une variété d'applications. Le EFM32 est composé de plusieurs modules, par exemple « Core and Memory », « Clock Management », « Energy Management », « Serial Interfaces », « I/ O Ports », « Timers and Triggers », « Analog Interfaces », « 32- bit bus ».

Dans ce projet, on utilise principalement la partie USART dans le module « serail interface ». USART est l'abréviation Universal Synchronous / Asynchronous Receiver / Transmitter. Il supporte la communication SPI. En mode synchrone SPI, un signal d'horloge séparé est transmis avec les données. Ce signal d'horloge est généré par le maître de bus, et l'émission des données du maître et de l'esclave est selon cette horloge.



L'interface SPI est composé de quatre broches, RX, TX, CLK, /CS. RX est la broche de transmettre un signal ; TX est pour recevoir unsignal ; CLK est la broche de l'horloge ; /CS est pour selecter la broche.
Le contrôleur de DMA est capable de transférer des données entre les périphériques et la mémoire RAM.
Carte d'évaluation

La carte d'évaluation EFM32ZG222F32-QFP48 est une carte avec un afficheur LCD, deux LEDs, 40 broches, un port de J-LINK, Cortex-M0+, deux bouttons de touche et un bouton de reset. Il se compose de six parties principales comme dans la photo ci-dessous.

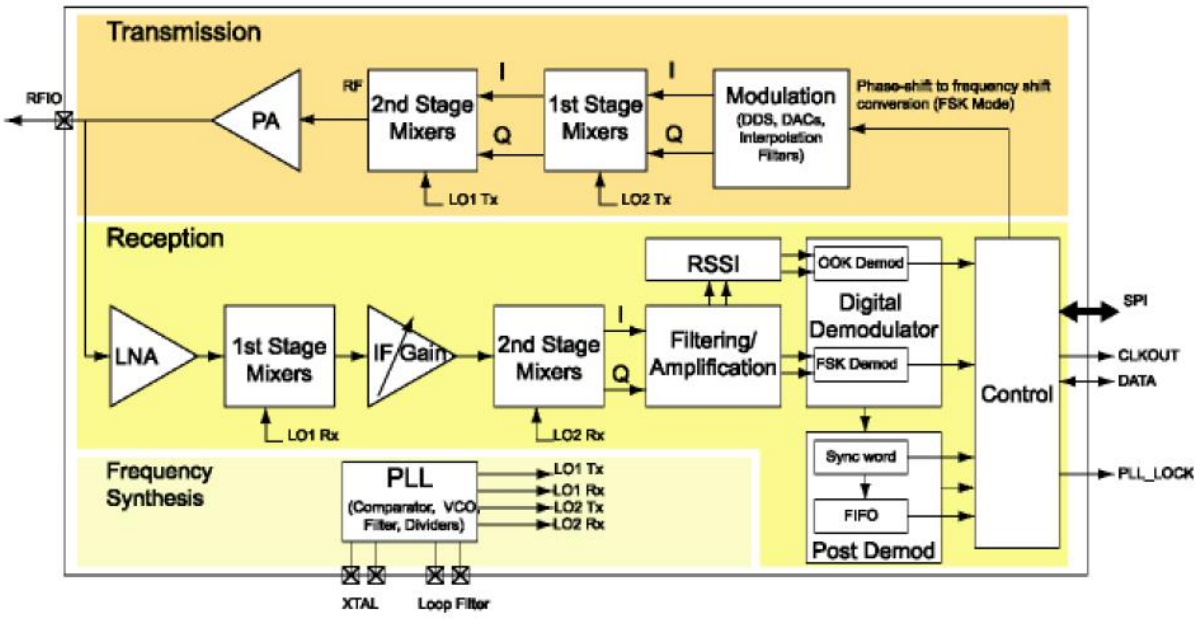


Nous nous intéresserons aux deux parties LED et EXP PORT. Pour utiliser cette carte, premièrement on doit installer le driver interface J-LINK et les logiciels de programmer Simplicity Studio et IAR embedded workbench IDE. Dans le graphe schématique, on peut voir les quatre broches de SPI dans la partie EXP PORT, ce sont les ports PC6, PC7, PD14, PD15. Mais nous ne pouvons pas les utiliser directement, il faut configurer les quatre broches. En même temps, nous pouvons seulement utiliser ces quatre broches comme l'interface de SPI. A cause de cette raison, je me suis trompé de la configuration des ports SPI. J'ai pensé que nous pouvons sélectionner arbitrairement quatre broches à configurer. Après j'ai trouvé qu'il y a seulement deux USARTs, USART0 et USART1 dans cette carte que nous pouvons utiliser. Par ailleurs, USART0 ne supporte pas la communication SPI, parce que la carte n'a pas défini les quatre ports pour USART0. Sur la partie d'afficheur LCD, Les écrans à cristaux liquides utilisent la polarisation de la lumière par des filtres polarisants et la biréfringence de certains cristaux liquides en phase nématique, dont on peut faire varier l'orientation en fonction du champ électrique. Du point de vue optique, l'écran à cristaux liquides est un dispositif passif : il n'émet pas de lumière, seul sa transparence varie, et il doit donc disposer d'un éclairage. Pour afficher les contextes sur l'écran de LCD, on peut utiliser la commande « printf » et en même temps, tous

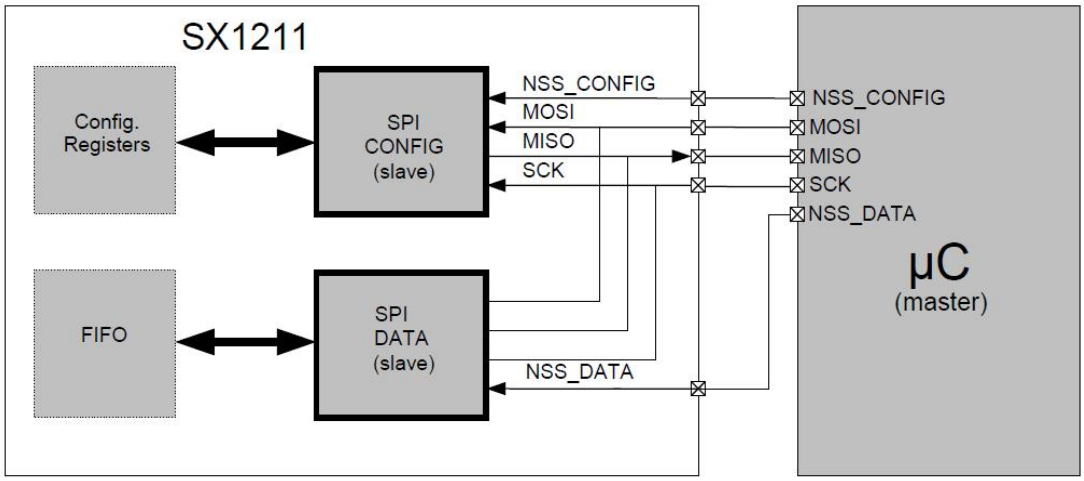
d'abord nous devons activer l'afficheur et définir l'horloge

Emetteur –récepteur

Le SX1211 est à faible coût, une seule puce émetteur-récepteur fonctionnant dans les bandes de fréquences 863-870, 902-928 MHz et 950-960 MHz. Le SX1211 est optimisée pour une très faible consommation d'énergie (3 mA en mode de réception).



Le SX1211 comporte trois modes de fonctionnement différents de données sélectionnables par l'utilisateur : Mode continu, Mode bufferisé, Mode paquet.
Chaque bit transmis ou reçu est accessible en temps réel à la broche de données. Ce mode peut être utilisé en cas de traitement de signal externe adéquat est disponible.
Mode bufferisé: chaque octet transmis ou reçu est stocké dans une mémoire FIFO et accessible par le bus SPI. uC chargé de traitement est donc sensiblement réduite par rapport à un fonctionnement en mode continu. La longueur de paquet est illimitée.
Mode paquet (recommandé): l'utilisateur ne fournit / récupère octets de charge utile à / de la FIFO. Le paquet est automatiquement construit avec préambule, Sync mot, et en option encodage libre CRC, DC et l'opération inverse est effectuée à la réception. La longueur maximale de la charge utile est limitée à la limite maximale de la FIFO de 64 octets.



L'interface SPI du SX1211 se compose de deux sous-blocs :

SPI Config: utilisé dans tous les modes de fonctionnement de données pour lire et écrire les registres de configuration qui contrôlent tous les paramètres de la puce (de mode de fonctionnement, le débit binaire, etc ..).

SPI données: utilisé dans tamponnée et mode paquet d'écrire et de lire des octets de données vers et depuis le FIFO. (Interruptions FIFO peuvent être utilisées pour gérer le contenu de FIFO).

Configuration and Status Registers :

Nom	Taille	Adresse	Description
MCParam	13 x 8	0 - 12	Les principaux paramètres communs pour transmettre et recevoir modes
IRQParam	3 x 8	13 - 15	Interruptteur registers
SYNCParam	4 x 8	22 – 25	Pattern
TXParam	1 x 8	26	Paramètres de l'émetteur
OSCParam	1 x 8	27	Paramètres Cristaloscillateur
PKTParam	4 x 8	28 - 31	Paramètres du gestionnaire de paquets

Programmation

D'abord, j'ai créé un « workspace » dans le logiciel. Ensuite, j'y ai ajouté 4 groupes « CMSIS », « driver », « emlib », « source ». Dans le groupe « CMSIS », j'ai ajouté deux documents de la bibliothèque 'startup_efm32zg.s' celui-ci est le dossier pour démarrager la CMSIS base périphérie et 'systeme_efm32zg.c' qui est le dossier sur la couche du système pour les périphériques EFM32ZG. Dans le groupe « driver », j'ai ajouté tous les documents .c qui définit les pilotes d'interface d'afficheur LCD. Dans le groupe « emlib », j'ai ajouté tous les documents .c qui défini les fonctions que j'ai utilisé dans ma programmation. Le dernier groupe « source » qui contient quatre dossiers de ma programmation. Ce sont les dossiers 'main.c', 'SPI.c', 'usart.c' et 'MRF49XA.c

Programmation sur la configuration SPI

Dans un premier temps, j'ai commencé à rédiger la programmation de configuration sur l'interface SPI. J'ai créé un dossier 'SPI.c'. La première fois que j'ai fait la configuration, j'ai utilisé le USART1 et USART2, un USART est comme l'émetteur qui peut envoyer une donnée, et l'autre comme le récepteur qui peut recevoir underground donnée. Parce que à ce moment-là, j'ai pensé à tester les deux fonctions, USART_sendBuffer() et USART_recevBuffer. Ensuite, quand j'ai essayé de compiler la programmation, il m'a indiqué toujours que USART2 n'a pas été défini. Au début, j'ai pensé que c'est parce que je n'ai pas réussi d'ajouter le dossier qui comporte la définition de USART2. Donc, j'ai commencé à chercher la définition de USART2 partout dans la bibliothèque. Malheureusement, je n'ai pas pu y'arriver. Dans ce cas, j'ai demandé à mon tuteur pour des conseils. Après la recherche, mon tuteur m'a indiqué qu'il y a seulement deux USARTs dans le microcontroleur qu'on utilise. Par conséquent, j'ai pu seulement tester la fonction d'envoyer une donnée.

Avant d' utiliser USART1, il faut d'abord l'initialiser. La variable USART1 est le type de USART_TypeDef qui est une structure. Comme la configuration ci-dessous, on peut configurer la fréquence, le nombre de Bits de chaque donnée et le mode TX，RX etc.

```
init.baudrate      = 1000000;                  //la fréquence 1MHZ
init.databits      = usartDatabits8;           //8 bits donnée
init.master        = 1;                        // le mode master
init.clockMode     = usartClockMode0;
init.prsRxEnable   = 0;
init.autoTx        = 0;
```

Enfin, on utilise la fonction USART_InitSync(USART1, &init) pour donner toute la configuration à USART1.

Des que j'ai fini l'initialisation, j'ai commencé à configurer les quatre ports de l'interface SPI. Premièrement, il faut sélecter une fréquence du microcontrôleur. Deuxièmement, il faut activer l'horloge du module GPIO. Ensuite, on peut configurer les quatre ports comme le port d'entrée ou de sortie par rapport au fonctionnement de chaque port. Par exemple, le port PC15 est le port de l'horloge, donc on l'a configuré comme le type de sortie. Enfin, on doit aussi activer les quatre ports et activer l'horloge de USART1.

```
/* activer TX, RX, CLK, CS */
USART1->ROUTE |= USART_ROUTE_TXPEN | USART_ROUTE_RXPEN | USART_ROUTE_CLKPEN | USART_ROUTE_CSPEN;
```

Programmation d'envoyer et de recevoir une donnée

J'ai créé un nouveau document 'usart.c'. Pour envoyer une donnée par l'intrerface SPI, d'abord on définit une chaîne de caractère, et puis on calcule le nombre de caractère dans cette chaîne. on fait une boucle pour donner en ordre chaque caractère à le variable USART1->TXDATA.

```
for (ii = 0; ii < bytesToSend;  ii++)
  {
    /* Waiting for the usart to be ready */
    while (!(usart->STATUS & USART_STATUS_TXBL)) ;

    if (txBuffer != 0)
    {
      /* Writing next byte to USART */
      usart->TXDATA = *txBuffer;
      txBuffer++;
    }
    else
    {
      usart->TXDATA = 0;
    }
  }
```

Pour recevoir une donnée, c'est presque la même théorie. On a seulement besion de remplacer le commande 'usart->TXDATA = *txBuffer;' par '*rxBuffer1=usart->RXDATA ;' pour recevoir une donnée. En même temps, changer le registre dont on détecte l'état.

Programmation sur l'émetteur-récepteur

Avant de envoyer une donnée à calculer la distance vers l'émetteur-récepteur, il faut envoyer des données comme ci-dessous à configurer la carte :

```
const long      GENCREG         = 0x8038;
// Cload=12.5pF; TX registers & FIFO are disabled
const long      PMCREG          = 0x8200;
// Everything off, uC clk enabled
const long      RXCREG          = 0x94A1;
// BW=135kHz, DRSSI=-97dBm, pin8=VDI, fast VDI
const long      TXBREG          = 0xB800;
const long      FIFORSTREG = 0xCA81;
// Sync. latch cleared, limit=8bits, disable sensitive reset
const long      BBFCREG         = 0xC22C;
// Digital LPF (default)
const long      AFCCREG         = 0xC4D7;
// Auto AFC (default)
const long      CFSREG      = 0xA7D0;
// Fo=915.000MHz (default)
const long      TXCREG      = 0x9830;
//df=60kHz, Pmax, normalmodulation polarity
const long      DRSREG          = 0xC623;
// 9579Baud (default)
```

Donc, j'ai créé une function qui s'appelle 'MRF49XA_Init' pour envoyer ces données. Ensuite, j'ai rédigé les deux fonctions 'MRF49XA_Send_Packet' et 'void MRF49XA_Receive_Packet'. J'ai fait une boucle pour mettre ou récupérer les données dans le registre de l'émetteur-récepteur.

```
for (a=0;a<length;a++)
    {
      SPI_Write16(TXBREG | data[a]);          // write a byte to tx register
     }
```

Conclusion

Le projet

Tout d'abord je regrette de ne pas atteindre les objectifs souhaités. Tout au long du processus, j'ai rencontré beaucoup de problèmes. Dans le processus de résolution du problème, j'ai également acquis beaucoup d'expérience et de méthodes.

J'ai choisi ce projet parce que j'ai été très intéressé par la communication d'interface SPI. Il ya longtemps, j'avais entendu parler de la technologie, mais, je n'en connais pas les principes spécifiques, et ne savais non plus comment la contrôler ni la mettre en œuvre. Avec le développement de la technologie électronique, l'interface SPI a été utilisé dans de nombreux types de communications électroniques. Donc, comme un ingénieur en électronique, nous devrions saisir l'interface technique SPI, qui fournira aux travaux futurs et à l'apprentissage une aide importante.

Tout au long du projet, nous avons manqué quelques composantes. Pour une raison quelconque, nous recevions les composantes avec du retard. Je regrette de ne pas avoir beaucoup de temps pour mener à terme ce projet qui me passionne énormément.

Bibliographie

http://www.silabs.com/Pages/default.aspx;
http://community.silabs.com/;
http://www.silabs.com/support/pages/contacttechnicalsupport.aspx;
http://www.alldatasheet.com/view.jsp?Searchword=Sx1211;
http://www.alldatasheet.com/view.jsp?Searchword=MRF49;
http://blog.csdn.net/playwolf719/article/details/8863673;
http://www.silabs.com/products/mcu/pages/simplicity-studio.aspx;

Annexes

/*Le dossier SPI.c*/

```
#include "em_device.h"
#include "spi.h"
#include "em_gpio.h"
#include "spi_project.h"
#include "usart.h"
#include "em_cmu.h"
#include "em_usart.h"

void USART1_setup(void)
{
  USART_InitSync_TypeDef init = USART_INITSYNC_DEFAULT;

  init.baudrate     = 1000000;
  init.databits     = usartDatabits8;
  init.msbf         = 0;
  init.master       = 1;
  init.clockMode    = usartClockMode0;
  init.prsRxEnable  = 0;
  init.autoTx       = 0;

  USART_InitSync(USART1, &init);
}

void eADesigner_Init(void)
{
  /* Using HFRCO at 14MHz as high frequency clock, HFCLK */
  CMU_ClockSelectSet(cmuClock_HF, cmuSelect_HFRCO);

  /* No low frequency clock source selected */

  /* Enable GPIO clock */
  CMU_ClockEnable(cmuClock_GPIO, true);

  /* Pin PA1 is configured to Push-pull */
  GPIO->P[0].MODEL = (GPIO->P[0].MODEL & ~_GPIO_P_MODEL_MODE1_MASK) |
GPIO_P_MODEL_MODE1_PUSHPULL;
  /* To avoid false start, configure output US1_CS as high on PC14 */
  GPIO->P[2].DOUT |= (1 << 14);
  /* Pin PC14 is configured to Push-pull */
```

```
  GPIO->P[2].MODEH = (GPIO->P[2].MODEH & ~_GPIO_P_MODEH_MODE14_MASK) |
GPIO_P_MODEH_MODE14_PUSHPULL;
  /* Pin PC15 is configured to Push-pull */
  GPIO->P[2].MODEH = (GPIO->P[2].MODEH & ~_GPIO_P_MODEH_MODE15_MASK) |
GPIO_P_MODEH_MODE15_PUSHPULL;
  /* Pin PD6 is configured to Input enabled */
  GPIO->P[3].MODEL = (GPIO->P[3].MODEL & ~_GPIO_P_MODEL_MODE6_MASK) |
GPIO_P_MODEL_MODE6_INPUT;
  /* To avoid false start, configure output US1_TX as high on PD7 */
  GPIO->P[3].DOUT |= (1 << 7);
  /* Pin PD7 is configured to Push-pull */
  GPIO->P[3].MODEL = (GPIO->P[3].MODEL & ~_GPIO_P_MODEL_MODE7_MASK) |
GPIO_P_MODEL_MODE7_PUSHPULL;

  /* Enable clock for USART1 */
  CMU_ClockEnable(cmuClock_USART1, true);
  /* Custom initialization for USART1 */
  USART1_setup();

  USART1->CMD=USART_CMD_MASTEREN|USART_CMD_TXEN|USART_CMD_RXEN;
  /* Module USART1 is configured to location 3 */
  USART1->ROUTE = (USART1->ROUTE & ~_USART_ROUTE_LOCATION_MASK) |
USART_ROUTE_LOCATION_LOC3;
  /* Enable signals TX, RX, CLK, CS */
  USART1->ROUTE |= USART_ROUTE_TXPEN | USART_ROUTE_RXPEN |
USART_ROUTE_CLKPEN | USART_ROUTE_CSPEN;

}
```

/*Le dossier usart.c*/

```
#include "em_device.h"
#include "usart.h"
#include "em_gpio.h"

void USART1_sendBuffer(unsigned char * txBuffer, int bytesToSend)
{
  USART_TypeDef *usart = USART1;
  int                ii;

  /* Sending the data */
  for (ii = 0; ii < bytesToSend;   ii++)
  {
    /* Waiting for the usart to be ready */
```

```
      while (!(usart->STATUS & USART_STATUS_TXBL)) ;

      if (txBuffer != 0)
      {
         /* Writing next byte to USART */
         usart->TXDATA = *txBuffer;
         txBuffer++;
      }
      else
      {
         usart->TXDATA = 0;
      }
   }

   /*Waiting for transmission of last byte */
   while (!(usart->STATUS & USART_STATUS_TXC)) ;
}


void USART1_recevBuffer(unsigned char * rxBuffer, int bytesToSend)
{
   USART_TypeDef *usart = USART1;
   int                  ii;
   unsigned char * rxBuffer1=rxBuffer;

   /* recieving the data */
   for (ii = 0; ii < bytesToSend;    ii++)
   {
      /* Waiting for the incoming data */
      while (!(usart->STATUS & USART_STATUS_RXBLOCK)) ;

         /* receiving next byte to USART */
         *rxBuffer1=usart->RXDATA ;
         rxBuffer1++;

   }

}

void SPI_Command(unsigned char txBuffer)
{
   /*configure output US1_CS as LOW on PC14 */
  GPIO->P[2].DOUT &= (0xFE << 14);
  USART1_sendBuffer(&txBuffer,2);
```

```
  GPIO->P[2].DOUT |= (1 << 14);
}

void SPI_Write16(unsigned char spicmd)
{
   USART1_sendBuffer(&spicmd,2);
}
```

/*Le dossier MRF49XA.c*/

```
#include "MRF49XA.h"
#include "usart.h"

const long      GENCREG         = 0x8038;       // Cload=12.5pF; TX registers & FIFO are disabled
const long      PMCREG          = 0x8200;       // Everything off, uC clk enabled
const long      RXCREG          = 0x94A1;       // BW=135kHz, DRSSI=-97dBm, pin8=VDI, fast VDI
const long      TXBREG          = 0xB800;
const long      FIFORSTREG      = 0xCA81;       // Sync. latch cleared, limit=8bits, disable sensitive reset
const long      BBFCREG         = 0xC22C;       // Digital LPF (default)
const long      AFCCREG         = 0xC4D7;       // Auto AFC (default)
const long      CFSREG          = 0xA7D0;       // Fo=915.000MHz (default)
const long      TXCREG          = 0x9830;       // df=60kHz, Pmax, normal modulation polarity
const long      DRSREG          = 0xC623;       // 9579Baud (default)

void MRF49XA_Init(){

//---- Send init cmd
   SPI_Command( FIFORSTREG);
   SPI_Command( FIFORSTREG | 0x0002);
   SPI_Command( GENCREG);
   SPI_Command( CFSREG);
   SPI_Command( PMCREG);
   SPI_Command( RXCREG);
   SPI_Command( TXCREG);
//---- antenna tunning
   SPI_Command( PMCREG | 0x0020);          // turn on tx

//---- end of antenna tunning
   SPI_Command( PMCREG | 0x0080);          // turn off Tx, turn on receiver
   SPI_Command( GENCREG | 0x0040);         // enable the FIFO
```

```
   SPI_Command( FIFORSTREG);
   SPI_Command( FIFORSTREG | 0x0002);                    // enable syncron latch

}

void MRF49XA_Send_Packet(unsigned char *data, unsigned int length)
{
      int a;
      //---- turn off receiver , enable Tx register
      SPI_Command(PMCREG);                    // turn off the transmitter and receiver
      SPI_Command(GENCREG | 0x0080);          // Enable the Tx register
      //---- Packet transmission
      // Reset value of the Tx regs are [AA AA], we can start transmission
      //---- Enable Tx
      SPI_Command(PMCREG |0x0020);            // turn on tx

            GPIO->P[2].DOUT &= (0xFE << 14);                              // chip select low
            for (a=0;a<length;a++)
                        {
                              SPI_Write16(TXBREG | data[a]);                        // write a byte to
tx register
                        }


            GPIO->P[2].DOUT |= (1<< 14);                                  // chip select high, end
transmission

      //---- Turn off Tx disable the Tx register
      SPI_Command(PMCREG | 0x0080);           // turn off Tx, turn on the receiver
      SPI_Command(GENCREG | 0x0040);          // disable the Tx register, Enable the FIFO
}

void MRF49XA_Receive_Packet(unsigned char *data)
{
      SPI_Command(PMCREG);                        // turn off tx and rx
      SPI_Command(GENCREG | 0x0040);               // enable the FIFO
      SPI_Command(PMCREG | 0x0080);                // turn on receiver
      GPIO->P[2].DOUT &= (0xFE << 14);             // chip select low
      USART1_recevBuffer(data,2);
      GPIO->P[2].DOUT |= (1<< 14);                  // chip select high, end recieving

}
```

/*le dossier main.c*/

```
#include <string.h>
#include "em_device.h"
#include "em_chip.h"
#include "em_cmu.h"
#include "spi.h"
#include "usart.h"
#include <stdio.h>
#include "em_emu.h"
#include "em_gpio.h"
#include "em_rtc.h"
#include "display.h"
#include "textdisplay.h"
#include "retargettextdisplay.h"
#include "MRF49XA.h"

/* Frequency of RTC clock. */
#define RTC_FREQUENCY     (64)
#define SLEEP_TIME          (1)
int valdis=10;
/* RTC callback parameters. */
static void (*rtcCallback)(void*) = NULL;
static void*    rtcCallbackArg          = 0;
static volatile bool          displayEnabled = false; /* Status of LCD display. */
static volatile bool          enterEM4 = false;
static volatile uint32_t    seconds = 0;         /* Seconds elapsed since reset.    */
static volatile int           rtcIrqCount = 0; /* RTC interrupt counter                */
static DISPLAY_Device_t displayDevice;       /* Display device handle.                */
static void RtcInit( void );
/* Buffers */
unsigned char transmitBuffer[]="A";
#define                     BUFFERSIZE       (sizeof(transmitBuffer) / sizeof(char))
unsigned char receiveBuffer[BUFFERSIZE];
unsigned char receiveBuffer2[BUFFERSIZE];

int main(void)
{
   /* Initialize chip */
   CHIP_Init();

  /* Initialize the display module. */
   displayEnabled = true;
   DISPLAY_Init();
```

```
 /* Retrieve the properties of the display. */
  if ( DISPLAY_DeviceGet( 0, &displayDevice ) != DISPLAY_EMSTATUS_OK )
  {
  /* Unable to get display handle. */
     while( 1 );
  }

  /* Retarget stdio to the display. */
  if ( TEXTDISPLAY_EMSTATUS_OK != RETARGET_TextDisplayInit() )
  {
     /* Text display initialization failed. */
     while( 1 );
  }

  /* Set RTC to generate an interrupt every second. */
  RtcInit();

  printf( "\n La distance(m):\n\n\t\t%d",valdis );


  /* Initalize and configure SPI */
  eADesigner_Init();

  /* Initialize and configure the radio chip*/
  MRF49XA_Init();


  /* Transmitting data */
    while(1)
    {
        GPIO->P[2].DOUT &= (0xFE << 14);
       USART1_sendBuffer(transmitBuffer, BUFFERSIZE);
        GPIO->P[2].DOUT |= (1 << 14);
       // MRF49XA_Send_Packet(transmitBuffer,2);
    }
}
```