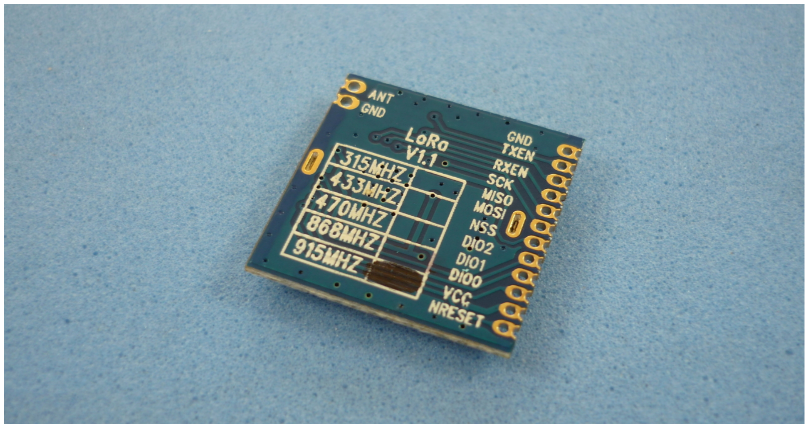
INTERNET OF THINGS
USING NICERF LORA1276



LPWAN and LORA

Internet of things (IoT) has evolved in such way that every day is cheaper, smaller and less power hungry to communicate "things" that were impossible a few years ago. Now technologies like GSM and WiFi are the most used for that task, but has their disadvantages:

- GSM, 3G, LTE: High costs: A data plan is needed for communications, high energy consumption.

  WiFi, BLUETOOTH : Low distance coverage (tens of meters inside buildings) , high energy consumption!

To overcome the disadvantages of previous technologies, LPWAN (Low Power Wide Area Network) has been proposed, and must fill some requirements:

- Oriented to devices with low data volume and sporadic communications.
- Low power consumption, so battery operated devices could work for many years.
-
- Wide area coverage to send and receive data without problems inside buildings, basements, industrial boxes, etc
-
-

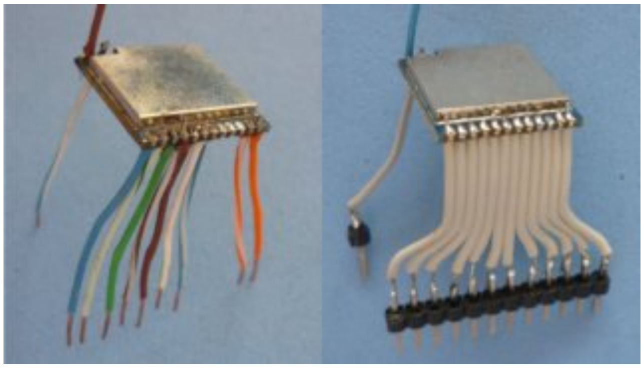
There are now a lot of technologies in the same path like LoRa, LTE-MTC, RPMA, UNB, and others more

LoRa alliance (CISCO, IBM, SEMTECH, MICROCHIP and others) aims to create a wireless communication standard for battery operated devices with regional, national or global coverage.

The main advantage of LoRa is the use of ISM (Industrial, Scientific and Medical) bands, so every person could create their own LPWAN without paying royalties or spectrum fees, as longs as stays inside nations's regulations.

NiceRF LoRa1276

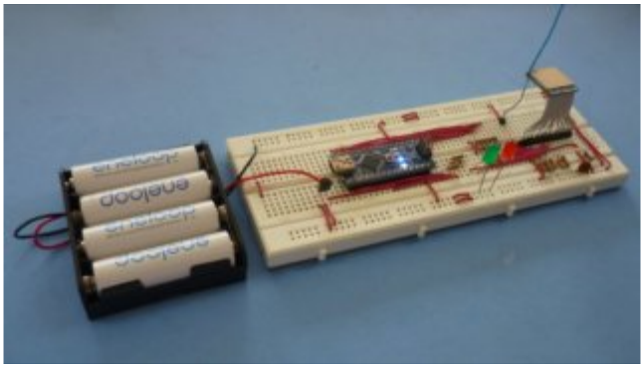
There are a lot of manufacturers of LoRa compatible modules. All of them uses SEMTECH chips, guaranteeing better c between different manufacturer's modules. The module used here is LoRa1276, manufactured by NiceRF. The modules cost about 20 dollars/pair (worldwide shipping included!) and incorporates SX1276 chip. The manufacturer claims 10km range in line-of-sight, and 1k in urban environments with a max transmission power of 120mW. The modules' package aren't hobbyist friendly due to distance between pads (1.27 mm) isn't compatible with standard breadboard (2.54 mm), however is possible to make a homemade adapter with a bit of creativity.



The SX1276 chip embedded in the module is able to operate from 100 MHz to 1050 MHz. However, this chip requires some external components between RF input/output pins and antenna. To minimize component burden and design complexity, the integrator put some passive and active components (capacitors, inductors, RF switch) to form a matched network, that will operate efficiently only in a small frequency range. There are different versions of modules manufactured to work in different ISM bands (a,b,c,d) to match local radio spectrum regulations, because not all countries allow to use the same ISM bands. In this example the 915Mhz version is used.

Lora 1276 and Arduino

Communication with the module is done via SPI, using and Arduino Nano (Clone) for the setup. Additional to the SPI signals, this module requires handling of additional signals. As the module works with 3.3 V max and Arduino Nano with 5 V, some signal level translation is required between Arduino and SX1276. If you don't have a signal level translation chip in your parts bin, you can work with resistors to make a voltage divider and work with the SPI in the lowest speed. The module can work as a transmitter and as a receiver, but not simultaneously. The sample code is based on an example provided by NiceRF. The module can works in different modes:



Modulation: OOK, FSK and LoRa
Error detection and correction: FEC and Cheksum
Power modes: Low power modes (for battery operated devices) and full power modes.
Interference suppression: Multiple chirp and spread factors

In the example presented, the highest power and sensibility settings were used, and works in the following way:

The transmitter sends a periodic message, on each packet sent a LED is toggled

The receiver is listening for the message, if it's received without errors a LED is toggled, if na

error is detected another different LED will be toggled

That's a very easy way to test the range of the devices. Put the transmitter in a fixed site and move the receiver until a bad message received or no message received at all in the expected interval of time.
The software is just a little example, but could be expanded to make a more robust system

CONCLUSIONS

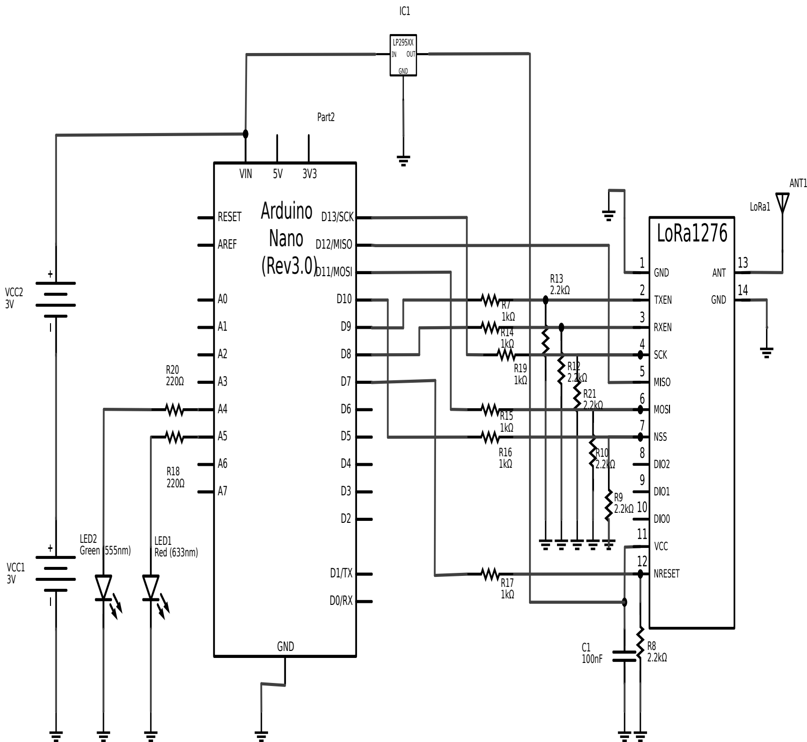
- In the example presented, a wire was cut to 1/4 wavelength (915 MHz) as an antenna. With a better designed antenna and in a higher position like building roof, or a tower a much more range could be obtained in urban environments.

- The module only provides the lowest OSI model layers, it's up to the user to choose an already developed communication stack or made their own according their needs.

- To test the maximum sensitivity of the receiver (and the longest transmission range) a Temperature Compensated Crystal Oscillator (TCXO) is mandatory. The module only incorporates a low cost crystal.

- The example shown, not exactly communicates a device with internet, but adding an Ethernet module to the Arduino is a relative simple task

SCHEMATIC


IC1
LP295XX
IN
OUT
GND
Part2
VIN
5V
3V3
Arduino
Nano
(Rev3.0)
RESET
AREF
A0
A1
A2
A3
A4
A5
A6
A7
D13/SCK
D12/MISO
D11/MOSI
D10
D9
D8
D7
D6
D5
D4
D3
D2
D1/TX
D0/RX
GND
VCC2
3V
VCC1
3V
R20
220Ω
R18
220Ω
LED2
Green (555nm)
LED1
Red (633nm)
R7
1kΩ
R14
1kΩ
R19
1kΩ
R15
1kΩ
R16
1kΩ
R17
1kΩ
R13
2.2kΩ
R12
2.2kΩ
R21
2.2kΩ
R10
2.2kΩ
R9
2.2kΩ
R8
2.2kΩ
C1
100nF
LoRa1276
LoRa1
ANT1
GND
TXEN
RXEN
SCK
MISO
MOSI
NSS
DIO2
DIO1
DIO0
VCC
NRESET
ANT
GND
1
2
3
4
5
6
7
8
9
10
11
12
13
14

ARDUINO SOFTWARE TX

LORA1276TX.ino

```
/*
NiceRF LoRa1276 Module Arduino NANO Clone V3

NANO        LoRa1276
D11 MOSI    6  MOSI
D12 MISO    5  MISO
D13 SCK     4  SCK
D10         7  NSS

by absolutelyautomation.com

*/

// using SPI library:
#include <SPI.h>

// Digital pins definition
#define MOSI   11
#define MISO   12
#define SCK    13
#define SS     10

#define NRESET 7
#define TXEN   9
#define RXEN   8
#define LED1   A4
#define LED2   A5

// register definition

#define LR_RegFifo                    0x00
#define LR_RegOpMode                  0x01
#define LR_RegBitrateMsb              0x02
#define LR_RegBitrateLsb              0x03
#define LR_RegFdevMsb                 0x04
#define LR_RegFdMsb                   0x05
#define LR_RegFrMsb                   0x06
#define LR_RegFrMid                   0x07
#define LR_RegFrLsb                   0x08
#define LR_RegPaConfig                0x09
#define LR_RegPaRamp                  0x0A
#define LR_RegOcp                     0x0B
#define LR_RegLna                     0x0C
#define LR_RegFifoAddrPtr             0x0D
#define LR_RegFifoTxBaseAddr          0x0E
#define LR_RegFifoRxBaseAddr          0x0F
#define LR_RegFifoRxCurrentaddr       0x10
#define LR_RegIrqFlagsMask            0x11
#define LR_RegIrqFlags                0x12
#define LR_RegRxNbBytes               0x13
#define LR_RegRxHeaderCntValueMsb     0x14
#define LR_RegRxHeaderCntValueLsb     0x15
#define LR_RegRxPacketCntValueMsb     0x16
#define LR_RegRxPacketCntValueLsb     0x17
#define LR_RegModemStat               0x18
#define LR_RegPktSnrValue             0x19
#define LR_RegPktRssiValue            0x1A
#define LR_RegRssiValue               0x1B
#define LR_RegHopChannel              0x1C
#define LR_RegModemConfig1            0x1D
#define LR_RegModemConfig2            0x1E
```

```
#define LR_RegSymbTimeoutLsb            0x1F
#define LR_RegPreambleMsb               0x20
#define LR_RegPreambleLsb               0x21
#define LR_RegPayloadLength             0x22
#define LR_RegMaxPayloadLength          0x23
#define LR_RegHopPeriod                 0x24
#define LR_RegFifoRxByteAddr            0x25
#define LR_RegModemConfig3              0x26
#define REG_LR_DIOMAPPING1              0x40
#define REG_LR_DIOMAPPING2              0x41
#define REG_LR_VERSION                  0x42
#define REG_LR_PLLHOP                   0x44
#define REG_LR_TCXO                     0x4B
#define REG_LR_PADAC                    0x4D
#define REG_LR_FORMERTEMP               0x5B
#define REG_LR_AGCREF                   0x61
#define REG_LR_AGCTHRESH1               0x62
#define REG_LR_AGCTHRESH2               0x63
#define REG_LR_AGCTHRESH3               0x64

// payload length
#define payload_length  7

// tx packet
unsigned char txbuf[payload_length]={'t','e','s','t','i','n','g'};
// rx packet
unsigned char rxbuf[30];


// Initialization
void setup() {
  byte temp = 0;


  // Initializing serial port, usefull for debuging
  Serial.begin(9600);


  // Initializing SPI pins

  pinMode(MOSI, OUTPUT);
  pinMode(MISO, INPUT);
  pinMode(SCK,OUTPUT);
  pinMode(SS,OUTPUT);
  digitalWrite(SS,HIGH); //disabling LoRa module

  // Initializing other I/O pins
  pinMode(NRESET, OUTPUT);
  pinMode(TXEN, OUTPUT);
  pinMode(RXEN, OUTPUT);
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);

  digitalWrite(NRESET,HIGH); // Deassert reset
  digitalWrite(TXEN,LOW); // Disabling tx antenna
  digitalWrite(RXEN,LOW); // Disabling rx antenna
  digitalWrite(LED1,LOW);
  digitalWrite(LED2,LOW);

  /* Initializing SPI registers
  description of every SPCR register bits
  | 7    | 6    | 5    | 4    | 3    | 2    | 1    | 0    |
  | SPIE | SPE  | DORD | MSTR | CPOL | CPHA | SPR1 | SPR0 |

  SPIE - Enable SPI interupt (logic 1)
  SPE  - Enable SPI (logic 1)
  DORD - Send Least Significant Bit (LSB) first (logic 1) , Send Most Significant Bit (MSB) first (logic 0)
  MSTR - Enable SPI master mode (logic 1), slave mode (logic 0)
  CPOL - Setup clock signal inactive in high (logic 1), inactive in low (logic 0)
  CPHA - Read data on Falling Clock Edge (logic 1), Rising edge (logic 0)
```

```
  SPR1 y SPR0 - Setup SPI data rate: 00 Fastest (4MHz), 11 Slowest (250KHz)

  // SPCR = 01010011
  //interupt disabled,spi enabled,most significant bit (msb) first,SPI master,clock inactive low,
  data fech rising clock edge, slowest data rate*/

  SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR1)|(1<<SPR0);
  temp=SPSR;  //Reading and discarding previous data
  temp=SPDR;  //Reading and discarding previous data
  delay(10);


}

void loop() {

  digitalWrite(LED1,LOW);
  digitalWrite(LED2,LOW);

  reset_sx1276();
  Config_SX1276();  // intializing RF module


  while(1){
    mode_tx();                  // transmit packet
    delay(300);

  }

}



byte SPIreadRegister(byte addr) {

  byte result;

  digitalWrite(SS, LOW);          // Select LoRa module

  SPDR = addr;                    // Send address & Start transmission. In READ mode bit 7 of address is always 0! for sx1276
  while (!(SPSR & (1<<SPIF)))     // Wait for transmission to finish
  {
  };
  result = SPDR;                  // Discard first reading

  SPDR = 0x0;                     // Sending dummy byte to get the result
  while (!(SPSR & (1<<SPIF)))     // Wait for transmission to finish
  {
  };
  result = SPDR;                  // Reading register value

  digitalWrite(SS, HIGH);         // Deselect LoRa module

  return (result);

}

byte SPIwriteRegister(byte addr,byte value) {

  byte result;

  digitalWrite(SS, LOW);          // Select LoRa module

  SPDR = addr | 0x80;             // Send address & Start transmission. In WRITE mode bit 7 of address is always 1! for sx1276
  while (!(SPSR & (1<<SPIF)))     // Wait for transmission to finish
  {
  };
  result = SPDR;                  // Discard first reading
```

```
  SPDR = value;                    // Sending byte
  while (!(SPSR & (1<<SPIF)))      // Wait for transmission to finish
  {
  };
  result = SPDR;                  // Discard second reading

  digitalWrite(SS, HIGH);        // Deselect LoRa module

}

void SPIwriteBurst(unsigned char addr, unsigned char *ptr, unsigned char len)
{
        unsigned char i;
        unsigned char result;

    digitalWrite(SS, LOW);         // Select LoRa module

    SPDR = addr | 0x80;            // Send address & Start transmission. In WRITE mode bit 7 of address is always 1! for sx1276
    while (!(SPSR & (1<<SPIF)))    // Wait for transmission to finish
    {
    };
    result = SPDR;              // Discard first reading

    for (i=0; i <= len; i++){

      SPDR = *ptr;                    // Sending bytes
      while (!(SPSR & (1<<SPIF)))    // Wait for transmission to finish
      {
      };
      result = SPDR;              // Discard second reading

      //DEBUG DEBUG DEBUG
      Serial.print(*ptr, HEX);
      //DEBUG DEBUG DEBUG

      ptr++;
    }

    //DEBUG DEBUG DEBUG
    Serial.print("\n");
    // DEBUG DEBUG DEBUG

    digitalWrite(SS, HIGH);        // Deselect LoRa module

}
void SPIreadBurst(unsigned char addr, unsigned char *ptr, unsigned char len)
{
        unsigned char i;
        unsigned char result;

    digitalWrite(SS, LOW);         // Select LoRa module

    SPDR = addr;               // Send address & Start transmission. In READ mode bit 7 of address is always 0! for sx1276
    while (!(SPSR & (1<<SPIF)))    // Wait for transmission to finish
    {
    };
    result = SPDR;              // Discard first reading

    for (i=0; i <= len; i++){

      SPDR = 0;                    // Sending dummy byte to get the result
      while (!(SPSR & (1<<SPIF)))  // Wait for transmission to finish
      {
      };
      *ptr = SPDR;              // move pointer


      ptr++;
    }
```

```
        //DEBUG DEBUG DEBUG
        Serial.print("\n");
        // DEBUG DEBUG DEBUG

        digitalWrite(SS, HIGH);         // Deselect LoRa module

}

void reset_sx1276(void)
{
   digitalWrite(TXEN, LOW);
   digitalWrite(RXEN, LOW);

   digitalWrite(NRESET, LOW);
   delay(10);
   digitalWrite(NRESET, HIGH);
   delay(20);
}

void Config_SX1276(void)
{
                // put in sleep mode to configure

                SPIwriteRegister(LR_RegOpMode,0x00);                    // sleep mode, high frequency
                delay(10);

                SPIwriteRegister(REG_LR_TCXO,0x09);                     // external crystal
        SPIwriteRegister(LR_RegOpMode,0x80);                            // LoRa mode, high frequency

        SPIwriteRegister(LR_RegFrMsb,0xE4);
                SPIwriteRegister(LR_RegFrMid,0xC0);
                SPIwriteRegister(LR_RegFrLsb,0x00);                     // frequency：915 MHz


                SPIwriteRegister(LR_RegPaConfig,0xFF);   // max output power PA_BOOST enabled


                SPIwriteRegister(LR_RegOcp,0x0B);                       // close over current protection  (ocp)
                SPIwriteRegister(LR_RegLna,0x23);                       // Enable LNA

        SPIwriteRegister(LR_RegModemConfig1,0x72);   // signal bandwidth：125kHz,error coding= 4/5, explicit header mode

                SPIwriteRegister(LR_RegModemConfig2,0xC7);          // spreading factor：12

                SPIwriteRegister(LR_RegModemConfig3,0x08);          // LNA? optimized for low data rate

                SPIwriteRegister(LR_RegSymbTimeoutLsb,0xFF);     // max receiving timeout

                SPIwriteRegister(LR_RegPreambleMsb,0x00);
                SPIwriteRegister(LR_RegPreambleLsb,16);         // preamble 16 bytes

                SPIwriteRegister(REG_LR_PADAC,0x87);            // transmission power 20dBm
                SPIwriteRegister(LR_RegHopPeriod,0x00);         // no frequency hoping

                SPIwriteRegister(REG_LR_DIOMAPPING2,0x01);       // DIO5=ModeReady,DIO4=CadDetected
                SPIwriteRegister(LR_RegOpMode,0x01);            // standby mode, high frequency
}

void mode_tx(void)
{
                unsigned char addr,temp;

                digitalWrite(TXEN,HIGH);                        // open tx antenna switch
        digitalWrite(RXEN,LOW);


                SPIwriteRegister(REG_LR_DIOMAPPING1,0x41);                      // DIO0=TxDone,DIO1=RxTimeout,DIO3=ValidHeader

                SPIwriteRegister(LR_RegIrqFlags,0xff);                  // clearing interupt
                SPIwriteRegister(LR_RegIrqFlagsMask,0xf7);                              // enabling txdone
```

```
        SPIwriteRegister(LR_RegPayloadLength,payload_length);          // payload length

        addr = SPIreadRegister(LR_RegFifoTxBaseAddr);       // read TxBaseAddr
        SPIwriteRegister(LR_RegFifoAddrPtr,addr);           // TxBaseAddr->FifoAddrPtr

        SPIwriteBurst(0x00,txbuf,payload_length);  // write data in fifo
        SPIwriteRegister(LR_RegOpMode,0x03);                            // mode tx, high frequency


    digitalWrite(LED1, !digitalRead(LED1));


        temp=SPIreadRegister(LR_RegIrqFlags);               // read interput flag
        while(!(temp&0x08))                                             // wait for txdone flag
        {
                temp=SPIreadRegister(LR_RegIrqFlags);

        }

        digitalWrite(TXEN,LOW);                         // close tx antenna switch
    digitalWrite(RXEN,LOW);

        SPIwriteRegister(LR_RegIrqFlags,0xff);              // clearing interupt
        SPIwriteRegister(LR_RegOpMode,0x01);                // standby mode, high frequency

}

void init_rx(void)
{
        unsigned char addr;

    digitalWrite(TXEN,LOW);                         // open rx antenna switch
    digitalWrite(RXEN,HIGH);

        SPIwriteRegister(REG_LR_DIOMAPPING1,0x01);                      //DIO0=00, DIO1=00, DIO2=00, DIO3=01  DIO0=00--RXDONE

        SPIwriteRegister(LR_RegIrqFlagsMask,0x3f);                      // enable rxdone and rxtimeout
        SPIwriteRegister(LR_RegIrqFlags,0xff);              // clearing interupt

        addr = SPIreadRegister(LR_RegFifoRxBaseAddr);       // read RxBaseAddr
        SPIwriteRegister(LR_RegFifoAddrPtr,addr);           // RxBaseAddr->FifoAddrPtr
        SPIwriteRegister(LR_RegOpMode,0x05);                            // rx mode continuous high frequency
}
```

ARDUINO SOFTWARE RX

LORA1276RX.ino

```
/*
NiceRF LoRa1276 Module Arduino NANO Clone V3

NANO        LoRa1276
D11 MOSI    6  MOSI
D12 MISO    5  MISO
D13 SCK     4  SCK
D10         7  NSS

by absolutelyautomation.com

*/

// using SPI library:
#include <SPI.h>

// Digital pins definition
#define MOSI   11
#define MISO   12
#define SCK    13
#define SS     10

#define NRESET 7
#define TXEN   9
#define RXEN   8
#define LED1   A4
#define LED2   A5

// register definition

#define LR_RegFifo                       0x00
#define LR_RegOpMode                     0x01
#define LR_RegBitrateMsb                 0x02
#define LR_RegBitrateLsb                 0x03
#define LR_RegFdevMsb                    0x04
#define LR_RegFdMsb                      0x05
#define LR_RegFrMsb                      0x06
#define LR_RegFrMid                      0x07
#define LR_RegFrLsb                      0x08
#define LR_RegPaConfig                   0x09
#define LR_RegPaRamp                     0x0A
#define LR_RegOcp                        0x0B
#define LR_RegLna                        0x0C
#define LR_RegFifoAddrPtr                0x0D
#define LR_RegFifoTxBaseAddr             0x0E
#define LR_RegFifoRxBaseAddr             0x0F
#define LR_RegFifoRxCurrentaddr          0x10
#define LR_RegIrqFlagsMask               0x11
#define LR_RegIrqFlags                   0x12
#define LR_RegRxNbBytes                  0x13
#define LR_RegRxHeaderCntValueMsb        0x14
#define LR_RegRxHeaderCntValueLsb        0x15
#define LR_RegRxPacketCntValueMsb        0x16
#define LR_RegRxPacketCntValueLsb        0x17
#define LR_RegModemStat                  0x18
#define LR_RegPktSnrValue                0x19
#define LR_RegPktRssiValue               0x1A
#define LR_RegRssiValue                  0x1B
#define LR_RegHopChannel                 0x1C
#define LR_RegModemConfig1               0x1D
#define LR_RegModemConfig2               0x1E
#define LR_RegSymbTimeoutLsb             0x1F
#define LR_RegPreambleMsb                0x20
```

```
#define LR_RegPreambleLsb                0x21
#define LR_RegPayloadLength              0x22
#define LR_RegMaxPayloadLength           0x23
#define LR_RegHopPeriod                  0x24
#define LR_RegFifoRxByteAddr             0x25
#define LR_RegModemConfig3               0x26
#define REG_LR_DIOMAPPING1               0x40
#define REG_LR_DIOMAPPING2               0x41
#define REG_LR_VERSION                   0x42
#define REG_LR_PLLHOP                    0x44
#define REG_LR_TCXO                      0x4B
#define REG_LR_PADAC                     0x4D
#define REG_LR_FORMERTEMP                0x5B
#define REG_LR_AGCREF                    0x61
#define REG_LR_AGCTHRESH1                0x62
#define REG_LR_AGCTHRESH2                0x63
#define REG_LR_AGCTHRESH3                0x64

// payload length
#define payload_length  7

// tx packet
unsigned char txbuf[payload_length]={'t','e','s','t','i','n','g'};
// rx packet
unsigned char rxbuf[30];


// Initialization
void setup() {
  byte temp = 0;


  // Initializing serial port, usefull for debuging
  Serial.begin(9600);


  // Initializing SPI pins

  pinMode(MOSI, OUTPUT);
  pinMode(MISO, INPUT);
  pinMode(SCK,OUTPUT);
  pinMode(SS,OUTPUT);
  digitalWrite(SS,HIGH); //disabling LoRa module

  // Initializing other I/O pins
  pinMode(NRESET, OUTPUT);
  pinMode(TXEN, OUTPUT);
  pinMode(RXEN, OUTPUT);
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);

  digitalWrite(NRESET,HIGH); // Deassert reset
  digitalWrite(TXEN,LOW); // Disabling tx antenna
  digitalWrite(RXEN,LOW); // Disabling rx antenna
  digitalWrite(LED1,LOW);
  digitalWrite(LED2,LOW);

  /* Initializing SPI registers
  description of every SPCR register bits
  | 7    | 6    | 5    | 4    | 3    | 2    | 1    | 0    |
  | SPIE | SPE  | DORD | MSTR | CPOL | CPHA | SPR1 | SPR0 |

  SPIE - Enable SPI interupt (logic 1)
  SPE  - Enable SPI (logic 1)
  DORD - Send Least Significant Bit (LSB) first (logic 1) , Send Most Significant Bit (MSB) first (logic 0)
  MSTR - Enable SPI master mode (logic 1), slave mode (logic 0)
  CPOL - Setup clock signal inactive in high (logic 1), inactive in low (logic 0)
  CPHA - Read data on Falling Clock Edge (logic 1), Rising edge (logic 0)
  SPR1 y SPR0 - Setup SPI data rate: 00 Fastest (4MHz), 11 Slowest (250KHz)
```

```
  // SPCR = 01010011
  //interupt disabled,spi enabled,most significant bit (msb) first,SPI master,clock inactive low,
  data fech rising clock edge, slowest data rate*/

  SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR1)|(1<<SPR0);
  temp=SPSR;  //Reading and discarding previous data
  temp=SPDR;  //Reading and discarding previous data
  delay(10);


}

void loop() {

  unsigned char temp,payload_size;

  digitalWrite(LED1,LOW);
  digitalWrite(LED2,LOW);

  reset_sx1276();
  Config_SX1276();                                    // initialize RF module

  init_rx();                                    // rx mode

  while(1){
    temp=SPIreadRegister(LR_RegIrqFlags);                 // read interupt
    if(temp & 0x40){                                      // wait for rxdone flag
      SPIwriteRegister(LR_RegIrqFlags,0xff);                 // clear interupt
      temp = SPIreadRegister(LR_RegFifoRxCurrentaddr);               // read RxCurrentaddr
      SPIwriteRegister(LR_RegFifoAddrPtr,temp);                      // RxCurrentaddr -> FiFoAddrPtr

      payload_size = SPIreadRegister(LR_RegRxNbBytes);               // read packet size
      SPIreadBurst(0x00, rxbuf, payload_size);                 // read from fifo

      //"testing"
      if( (rxbuf[0] == 't') && (rxbuf[6] == 'g') )             // simple packet verification, please! use CRC flag for more robustness
      {
            digitalWrite(LED2, !digitalRead(LED2));                   // Data OK toggle LED2
            init_rx();
      }
      else
      {
        digitalWrite(LED1, !digitalRead(LED1));                  // Data WRONG toggle LED1
        init_rx();                                             // reinitialize rx when fail
      }
    }

  }


}



byte SPIreadRegister(byte addr) {

  byte result;

  digitalWrite(SS, LOW);          // Select LoRa module

  SPDR = addr;                  // Send address & Start transmission. In READ mode bit 7 of address is always 0! for sx1276
  while (!(SPSR & (1<<SPIF)))     // Wait for transmission to finish
  {
  };
  result = SPDR;              // Discard first reading

  SPDR = 0x0;                   // Sending dummy byte to get the result
  while (!(SPSR & (1<<SPIF)))     // Wait for transmission to finish
  {
```

```
  };
  result = SPDR;              // Reading register value

  digitalWrite(SS, HIGH);        // Deselect LoRa module

  return (result);

}

byte SPIwriteRegister(byte addr,byte value) {

  byte result;

  digitalWrite(SS, LOW);         // Select LoRa module

  SPDR = addr | 0x80;            // Send address & Start transmission. In WRITE mode bit 7 of address is always 1! for sx1276
  while (!(SPSR & (1<<SPIF)))     // Wait for transmission to finish
  {
  };
  result = SPDR;                 // Discard first reading


  SPDR = value;                  // Sending byte
  while (!(SPSR & (1<<SPIF)))       // Wait for transmission to finish
  {
  };
  result = SPDR;                 // Discard second reading

  digitalWrite(SS, HIGH);        // Deselect LoRa module

}

void SPIwriteBurst(unsigned char addr, unsigned char *ptr, unsigned char len)
{
        unsigned char i;
        unsigned char result;

     digitalWrite(SS, LOW);         // Select LoRa module

     SPDR = addr | 0x80;            // Send address & Start transmission. In WRITE mode bit 7 of address is always 1! for sx1276
     while (!(SPSR & (1<<SPIF)))     // Wait for transmission to finish
     {
     };
     result = SPDR;              // Discard first reading

     for (i=0; i <= len; i++){

       SPDR = *ptr;                  // Sending bytes
       while (!(SPSR & (1<<SPIF)))     // Wait for transmission to finish
       {
       };
       result = SPDR;              // Discard second reading

       //DEBUG DEBUG DEBUG
       Serial.print(*ptr, HEX);
       //DEBUG DEBUG DEBUG

       ptr++;
     }

     //DEBUG DEBUG DEBUG
     Serial.print("\n");
     // DEBUG DEBUG DEBUG

     digitalWrite(SS, HIGH);        // Deselect LoRa module

}
void SPIreadBurst(unsigned char addr, unsigned char *ptr, unsigned char len)
{
        unsigned char i;
```

```
            unsigned char result;

        digitalWrite(SS, LOW);          // Select LoRa module

        SPDR = addr;                // Send address & Start transmission. In READ mode bit 7 of address is always 0! for sx1276
        while (!(SPSR & (1<<SPIF)))     // Wait for transmission to finish
        {
        };
        result = SPDR;              // Discard first reading

        for (i=0; i <= len; i++){

          SPDR = 0;                 // Sending dummy byte to get the result
          while (!(SPSR & (1<<SPIF)))   // Wait for transmission to finish
          {
          };
          *ptr = SPDR;              // move pointer


          ptr++;
        }

        //DEBUG DEBUG DEBUG
        Serial.print("\n");
        // DEBUG DEBUG DEBUG

        digitalWrite(SS, HIGH);         // Deselect LoRa module

}

void reset_sx1276(void)
{
   digitalWrite(TXEN, LOW);
   digitalWrite(RXEN, LOW);

   digitalWrite(NRESET, LOW);
   delay(10);
   digitalWrite(NRESET, HIGH);
   delay(20);
}

void Config_SX1276(void)
{
            // put in sleep mode to configure

            SPIwriteRegister(LR_RegOpMode,0x00);                    // sleep mode, high frequency
            delay(10);

            SPIwriteRegister(REG_LR_TCXO,0x09);                     // external crystal
        SPIwriteRegister(LR_RegOpMode,0x80);                        // LoRa mode, high frequency

        SPIwriteRegister(LR_RegFrMsb,0xE4);
            SPIwriteRegister(LR_RegFrMid,0xC0);
            SPIwriteRegister(LR_RegFrLsb,0x00);                     // frequency：915 MHz


            SPIwriteRegister(LR_RegPaConfig,0xFF);   // max output power PA_BOOST enabled


            SPIwriteRegister(LR_RegOcp,0x0B);                       // close over current protection  (ocp)
            SPIwriteRegister(LR_RegLna,0x23);                       // Enable LNA

        SPIwriteRegister(LR_RegModemConfig1,0x72);   // signal bandwidth：125kHz,error coding= 4/5, explicit header mode

            SPIwriteRegister(LR_RegModemConfig2,0xC7);          // spreading factor：12

            SPIwriteRegister(LR_RegModemConfig3,0x08);          // LNA? optimized for low data rate

            SPIwriteRegister(LR_RegSymbTimeoutLsb,0xFF);     // max receiving timeout
```

```
        SPIwriteRegister(LR_RegPreambleMsb,0x00);
        SPIwriteRegister(LR_RegPreambleLsb,16);          // preamble 16 bytes

        SPIwriteRegister(REG_LR_PADAC,0x87);             // transmission power 20dBm
        SPIwriteRegister(LR_RegHopPeriod,0x00);          // no frequency hoping

        SPIwriteRegister(REG_LR_DIOMAPPING2,0x01);       // DIO5=ModeReady,DIO4=CadDetected
        SPIwriteRegister(LR_RegOpMode,0x01);             // standby mode, high frequency
}

void mode_tx(void)
{
        unsigned char addr,temp;

        digitalWrite(TXEN,HIGH);                         // open tx antenna switch
    digitalWrite(RXEN,LOW);


        SPIwriteRegister(REG_LR_DIOMAPPING1,0x41);               // DIO0=TxDone,DIO1=RxTimeout,DIO3=ValidHeader

        SPIwriteRegister(LR_RegIrqFlags,0xff);                   // clearing interupt
        SPIwriteRegister(LR_RegIrqFlagsMask,0xf7);               // enabling txdone
        SPIwriteRegister(LR_RegPayloadLength,payload_length);    // payload length

        addr = SPIreadRegister(LR_RegFifoTxBaseAddr);    // read TxBaseAddr
        SPIwriteRegister(LR_RegFifoAddrPtr,addr);        // TxBaseAddr->FifoAddrPtr

        SPIwriteBurst(0x00,txbuf,payload_length);   // write data in fifo
        SPIwriteRegister(LR_RegOpMode,0x03);                     // mode tx, high frequency


    digitalWrite(LED1, !digitalRead(LED1));


        temp=SPIreadRegister(LR_RegIrqFlags);            // read interput flag
        while(!(temp&0x08))                                      // wait for txdone flag
        {
                temp=SPIreadRegister(LR_RegIrqFlags);

        }

        digitalWrite(TXEN,LOW);                          // close tx antenna switch
    digitalWrite(RXEN,LOW);

        SPIwriteRegister(LR_RegIrqFlags,0xff);           // clearing interupt
        SPIwriteRegister(LR_RegOpMode,0x01);             // standby mode, high frequency

}

void init_rx(void)
{
        unsigned char addr;

    digitalWrite(TXEN,LOW);                              // open rx antenna switch
    digitalWrite(RXEN,HIGH);

        SPIwriteRegister(REG_LR_DIOMAPPING1,0x01);               //DIO0=00, DIO1=00, DIO2=00, DIO3=01  DIO0=00--RXDONE

        SPIwriteRegister(LR_RegIrqFlagsMask,0x3f);               // enable rxdone and rxtimeout
        SPIwriteRegister(LR_RegIrqFlags,0xff);           // clearing interupt

        addr = SPIreadRegister(LR_RegFifoRxBaseAddr);    // read RxBaseAddr
        SPIwriteRegister(LR_RegFifoAddrPtr,addr);        // RxBaseAddr->FifoAddrPtr
        SPIwriteRegister(LR_RegOpMode,0x05);                     // rx mode continuous high frequency
}
```