LoRa Library LoraNode (V3.0)

Introduction

The new class **LoraNode** has been emplaced to really simplify creation and exploitation of a software node. A software node is an element of a LoRa network (that is not LoRaWAN). This software node is based on hardware LoRa shield for Arduino or on a mini board Maduino or on the our micro LoRa remote command. In any case this software communicates with the radio module SX1278 SEMTECH or any other equivalent as RFM98.

This class uses and conceals the more complex LORA class, witch in turn uses the basic class SX1278 for radio module management. In any case basic functions can be called by LR. or SX prefix.

Before talking about LoraNode class, we need to describe network structure where a LoraNode node works.

About the network:

- is flexible in terms of number of addressable devices; but the more devices addressable is set, the smaller is the range for possible network id; (maximum nodes number can be changed form 7 to 8191)
- is crypted (AES256) and the real key is generated from an integer seed;
- one random byte (marker) is added to the message just for further user applications;

The LoraNode class uses some default values for network and radio parameters to really reduce coding effort. Obviously, that values can be overwritten. Default values in details:

- devices addressable: 15
- network id: 2345 (can be changed choosing from 1 to 4095)
- the crypto key
- radio frequency : 433.6 MHz
- the spreading factor : code 10
- the band width : code 8
- the power : code 2 (10dBm = 10mW)
- the message buffer : 64 char (64 bytes + null byte)

The node id, in other words its address inside the network, is loaded from EEPROM if it was saved, or it can be directly provided by instance function.

Communication protocol

The library uses a proprietary protocol implemented by LORA class. This protocol is contained inside the LoRa radio protocol. In other words, the LoRa payload is split into four segments (crypted, except the first one):

net_id&node_id (to)	net_id&node_id (from)	marker	message

NB The radio communication of module is **half-duplex** (transmit or receive)

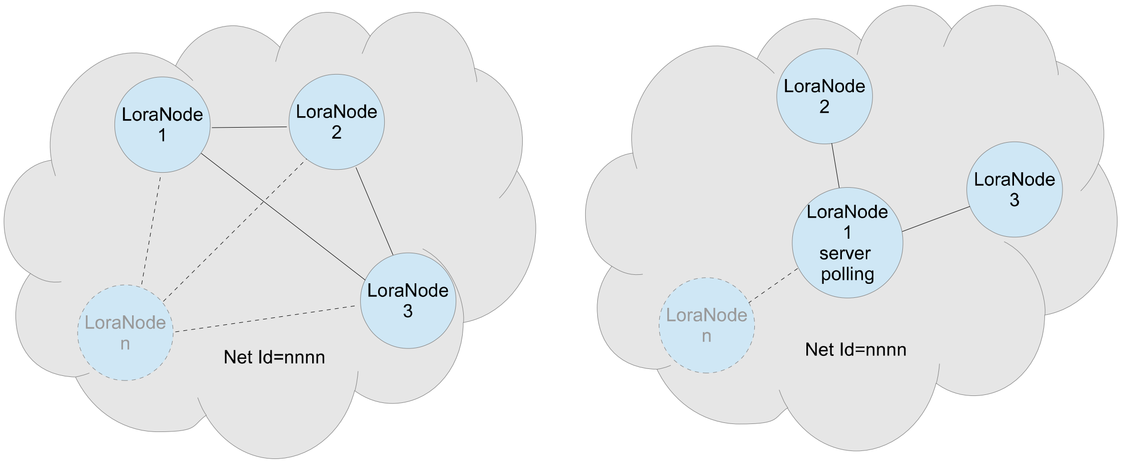

Fig. 1 - Mesh network or Star network

Principal functions

- Instance : **LoraNode()** ; with this format the node id is loaded from EEPROM, if it has been saved. Otherwise is 1 as default and in this case you can use setLoraNode(id), later , for setting it.
  Or directly : **LoraNode(node_number)**.
- Radio module activation: **begin()**; if any problem with module returns "false"
- Send message: **writeMessage(to,string,timeout)**; this function checks if air channel is busy (channel is defined by frequency,spreading factor, band width). If busy, it waits for "timeout" milliseconds at most. In case of timeout it returns "false" and it doesn't send message.
- Try to receive: **newMessAvailable(timeout)**; if a message arrives before "timeout" millisecond it returns "true" else "false", and message can be read using next functions. There is also a function format that waits just for a specific sender (newMessAvailable(sender,timeout);).
  NB Default receiving message length is 64 bytes long (null terminated string of 64 char length) . If you need to send longer messages you have to use changeMessageBufferLen(int maxMesslen) function.
- Read message text and others message features of packet just arrived:
  - **getMessage();**
  - **getSender();**
  - **getMarker();**

In addition of basic functions there are also functions that allow us to overwrite default values and to save node id on EEPROM. It is possible to save, also, all network and radio parameters on EEPROM. All functions are explained inside help pages and are commented on library file "LoraNode.h".

In order to complete description of particular features of library we have to include the function "setAutomaticAck(true/false)" that set on or set off a sort of automatic acknowledge (two char "AK") . If on, library sends acknowledge at any reception, and library waits for acknowledge replay after any sending. Default is off.

We remark that radio module is half-duplex, so it can't receive while it transmits. Therefore the management of this traffic needs some planning to not loose messages.

LoraNode class is string message oriented, but if you need to send binary data you can use byte oriented version of writeMess() and newMessAvailable() function; see help pages, or LoraNode.h file.

Basic sketch example

Here a basic software for a node that execute requests from network and sends notice of local events.

```
/* Basic example and model */
#include "LoraNode.h"           //Include library
LoraNode Node;                  //Instance (if node_id is in EEPROM library loads it)
                                //otherwise use: LoraNode Node(id);
bool SHIELD=true;               //Flag for radio module link checking

void setup() {
Serial.begin(9600);
if (!Node.begin()) {Serial.println("No LoRa module!");SHIELD=false;return;}
//Print (by Serial) configurations just for information (optional)
Node.printConfig();             //Print (by Serial) radio configuration (freq,sf,bw,pwr)
Node.printAddresses();          //Print (by Serial) network configuration (addresses)
}

//Loop: try to receive messages for 1000 milliseconds
// and, after message or timeout, check for any local event
void loop() {
if (!SHIELD) return;
if (Node.newMessAvailable(1000)){execRequest();}     //receiving
if (event()) {sendEvent();}                           //sending
}

// routine that manage request from network
void execRequest()
{
char* text=Node.getMessage();
int sender=Node.getSender();
// put here your code for managing request
}

// routine that detects local event
bool event()
{
// put here your code for local event detecting
// if event too short (<1000 milliseconds)
// better to manage it by interrupt that set a flag
// (no library interference because it doesn't use interrupt)
}

// routine that manages local event for sending
void sendEvent()
{
// put here your code to resolve addressee of message
// and to arrange event message. At last send message
// with a timeout of 300 milliseconds if air is busy
Node.writeMessage(to_id,mess,300);
}
```

Modifying default setting

Nodes can't have id=0 because 0 address has been reserved for possible broadcast message (message to all nodes).

If 15 nodes in a network are too few for your application, you can increase them until:

- 31 using the function: setMaxDevices(5) . But network Id must have a value from 1 and 2047
- 63 using the function: setMaxDevices(6) . But network Id must have a value from 1 and 1023
- 127 using the function: setMaxDevices(7) . But network Id must have a value from 1 and 511
- 255 using the function: setMaxDevices(8) . But network Id must have a value from 1 and 255

For more details see class LORA on help page. But please consider that, with a lot of nodes, because half-duplex transmission, communications can overlap. Using a "server polling" structure (star network), this problem is avoided. Actually, in a "server polling" structure, a specialized node acts as manager and scans each node in sequence.

The network Id can be changed by function setNetId(nuovo_id), and the cryptographic key can be changed using the function setKey(integer_number).

The "spread factor" and band width, define receiving sensitivity and transmission speed, but in a inverse way. High spread (code 12) paired with a narrow band (code 0) causes the best sensitivity (until to -150dBm), but also the smallest speed (11 pbs). On the contrary the lowest spread (code 6) paired with a widest band (code 9) causes a hight speed (23438 bps), but a low sensitivity (-113dBm). The default values (SPR=10 e BW=8) allow -130dBm for sensitivity and 1221 bps as speed. If you need to increase the sensitivity we suggest to increase the spreading factor to code 11 (setSpreadingFactor(11)) and decrease the band width to code 5 (set BandWidth(5)). With this tuning, the sensitivity becomes -140dBm and speed becomes one tenth of previous speed (112 bps).

Power consumption

To increase range you can also augment the transmission power; but consider the greater current consuming in this case. For instance, if power is 10mW (code 2 = default value), the transmission uses about 65mA (radio module only) (about 80/90mA for the entire board with Arduino). If you set power to code 1 (setPower(1)), it means 5mW (7dBm) of power, and a matches 50mA of consuming (70/80mA for the entire board) . The transmission current absorbing is just for the transmission time, that obviously, depend on message length and transmission parameters (SP,BW) seen before.

Special applications can require even lower consuming; in this case you can use a function of SX1278 class : SX.setLowPower(byte dBm); where dBm is the dBm real value from 2 to 6. With the lowest power (2dBm=1.5mW) the current absorbing becomes 35mA (radio module). On the contrary you can increase the power to 100mW (code 5) , but consider legal regulation. If the radio module can have a big range of values for transmission power, the receiving mode of radio module consume, instead, about 10mA always. If the radio module neither transmits nor receives, it is in the “standby” state and consumes just about 2mA. But if you want, you can reduce the current to a negligible value (few uA) putting module on “sleep” state.

Mode Radio Module			Entire System (with Arduino)	Just radio module
Sleep			25mA / 12mA (V>3.7V / battery)	<< 1mA
Standby			27mA / 14mA	1.7 mA
Receive			37mA / 25mA	11 mA
Transmit	2dBm (min)	1.5mW	63 / 53 mA	38 mA
	3dBm	2 mW	65 / 56 mA	40 mA
	4dBm	2.5mW	68 / 59 mA	43 mA
	5dBm	3.1mW	70 / 62 mA	45 mA
	7dBm	5mW	75 / 66 mA	50 mA
	10dBm	10mW	85 / 80 mA	60 mA
	13dBm	20mW	103 / 99 mA	78 mA
	17dBm	50mW	133 / 126 mA	108 mA
	20dBm	100mW	148 / 136 mA	123 mA

Consuming

Transmission consuming is just for transmission time. The transmission time depends on Spreading Factor (SF) and Band Width (BW) parameters. But SF and BW are responsible of sensitivity too. In table 2 are showed measured times for different sensitivity values. As you can see, a bias exists for any message length, because the overhead of LoRa and library protocol.

SF	BW	dBm	bps	fly time (milliseconds)			Bps	1/Bps (mS/B)
				1 to 10 char	50 char	100 char		
7	8	-121	6836	45	121	200	650	1.5
8	7	-127	1953	141	401	664	200	5
10	8	-130	1221	248	641	1032	127	8
9	5	-135	367	740	2116	3488	38	26
11	5	-140	112	2564	6883	10809	12	82
12	4	-144	46	6840	16261	26731	5	200
12	1	-149	15	20514	48781	80200	1.5	650

Transmission time for different sensitivity

EEPROM utilization

More then node id, the complete network configuration (net_id,max_devices,cripto_key) can also saved on EEPROM. The function “saveNetConfig()” uses the first 7 position on EEPROM. Function “loadNetConfig()” loads parameters saved. Radio parameters too can be saved to EEPROM (frequency, spreading factor, band width and power) with function saveRadioConfig() and loaded with function loadRadioConfig(). These values fill additional 7 bytes.

If your application need EEPROM for saving data, we suggest you use bytes from position 20, just to give margin for new versions. The function resetEEPROM() erases (putting 255 value) first 14 positions (so, all data are lost).