Moteino RFM69 Library

Library Version: 9 May 2015
Library Reference: https://github.com/LowPowerLab/RFM69
Document Version: 0.0
Save Date: 19 June 2015

RFM69/SX1231 networking

See:
http://www.hoperf.cn/upload/rf/RFM69-V1.3.pdf
http://www.semtech.com/images/datasheet/sx1231.pdf

OSI Layer reference

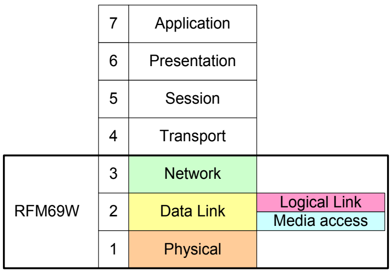

Figure 1: RFM69 and OSI model

RFM69(H)W[1] transceivers are assimilated to OSI layer 3 devices (physical, logical, network layers).

- **Layer 1**- *Physical Layer* (layer 1 802.15.4.g) uses radio unlicensed ISM Band frequencies (315, 433, 868 and 915MHz) with FSK, GFSK, MSK, GMSK and OOK modulations based on Manchester (DC free), NRZI, or Data Whitening encoding techniques at various baud rates
- **Layer 2** - The *Media Access* sub layer uses preamble for frame synchronization and a frame check sequence of 2 Bytes (CRC16) to guarantee the frame integrity. There is no inbuilt media multiple access protocol such as CSMA/CD and unique media access control (MAC) addresses configuration
  The *Logical Link* sub layer allows characterizes the datagram packet with various payload formats
- **Layer** 3 - *Network Layer* uses synchronization or network ID Bytes (a locally administered Network address) and node addresses. There is no advanced routing protocol (only best effort delivery).

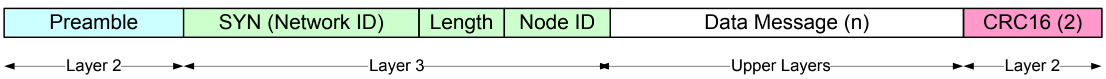

Figure 2: RFM69 and OSI model frame structure

There are 3 types of Logical Link frames:

- Fixed length packet format

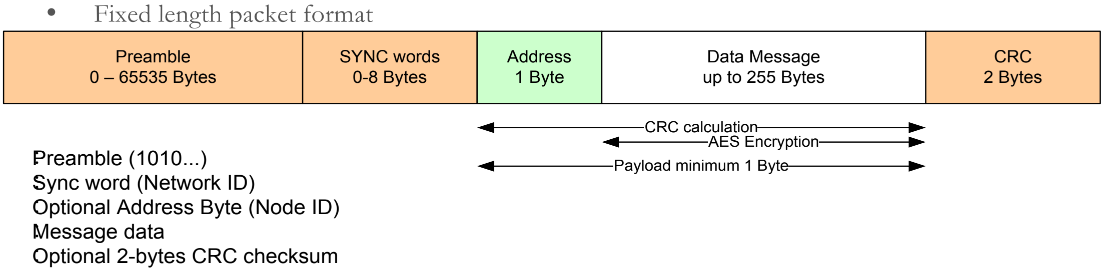

Figure 3: Fixed length packet format

[1] See http://www.hoperf.com/rf/ and http://www.semtech.com/wireless-rf/rf-transceivers/

Moteino RFM69 Library

- Variable length packet format

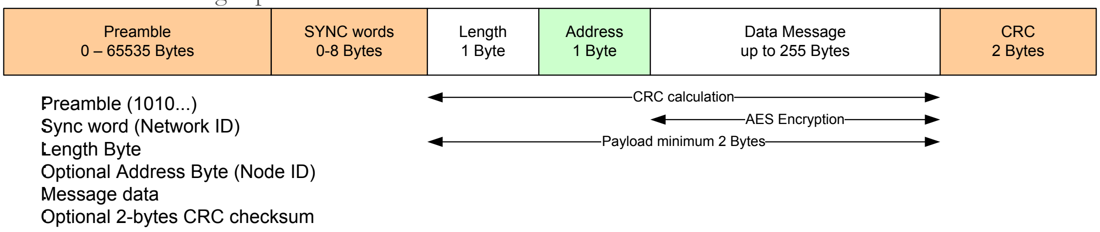

Figure 4: Variable length packet format

- Unlimited length packet format

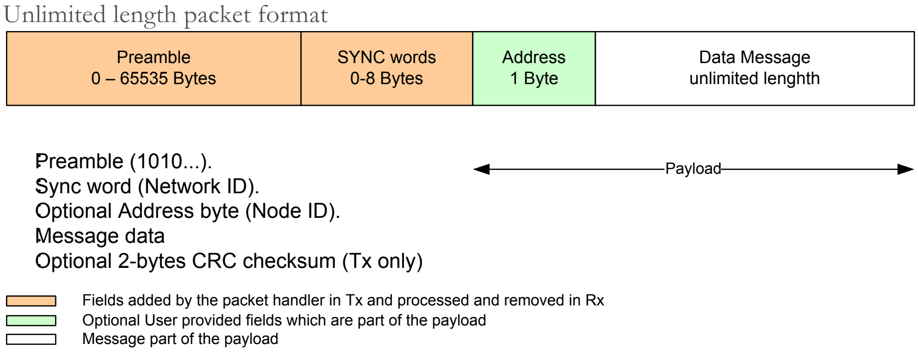

Figure 5: Unlimited length packet format

Notes:

- The Network ID (Sync words), if used is up to 4 words (8 Bytes)
- Frames not matching the Sync words are rejected
- The Node Address, if used is 1 Byte, allowing 256 different addresses, however one address is be dedicated for broadcasting
- The Payload Length, if used is 1 Byte, covering, itself, the Node Address (if used) and the Data part.
    - Internal FIFO used in packet mode is 66 Bytes
    - AES hardware encryption is done by module of 16 Bytes with a maximum of 64 Bytes
      The maximum lengths are therefore limited to:
        - For a fixed packet length:
            - No Node Address filtering:
                - Payload is 64 Bytes for 64 Bytes Data message (no Length, no Node address)
            - Node Address Filtering
                - Payload is 65 Bytes for a 64 Bytes Data message (no Length, with Node address)
        - For variable packet length:
            - No Node Address filtering:
                - Payload is 65 Bytes for 64 Bytes Data message (Length no Node address)
            - Node Address Filtering
                - Payload is 50 Bytes for a 48 Bytes Data message (Length and Node address)
            - If CRC calculation is configured, erroneous data messages are rejected

Moteino RFM69 Library

Moteino[2] implementation

Through the Moteino library several RFM69 configuration options are taken, the main characteristics (implemented by default) are the following:

1. Frequency ranges (transceiver dependent) 433MHz, 868 and 915 MHz. See transceiver bottom sides

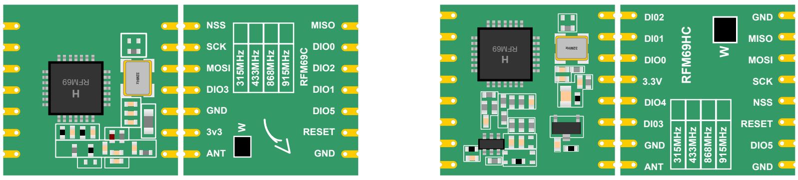

Figure 6: RFM69CW and RFM69HCW Transceivers layout (Top and Bottom sides)

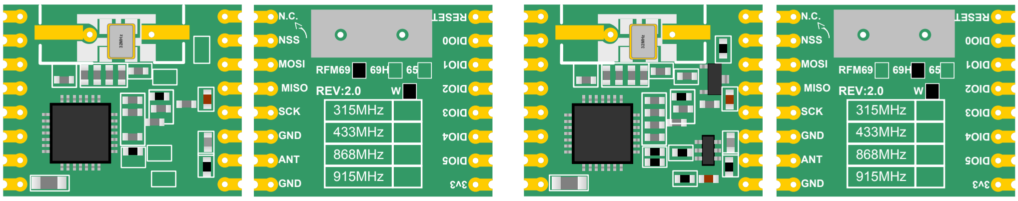

Figure 7: RFM69W and RFM69HW Transceivers layout (Top and Bottom sides)

2. 55.55Kbps Bit rate (default)
3. Adjustable power output capability (1dBm step); from – 18 to +13 dBm with +13dBm default for RFM69W modules and from -18 to +20 dBm with +13dBm default for RFM69**H**W modules
4. Various low power mode operation modes (default is STANDBY or 1.5 mA)
5. No Layer 1 encoding method (default is NRZI)
6. Basic Media channel access implementation through carrier detection CS without Carrier detection and back-off algorithm
7. Only variable frame length packet format is used
8. Data reception mode is packet (no continuous mode) up to 64 user data Bytes
9. CRC calculation enabled
10. 3 Bytes preamble (0xAAAAAA)
11. 2 Bytes for synchronization (1st Byte = 0x2D, 2nd Byte Network ID) including de facto Network ID filtering.
    - 256 networks are possible (0 to 255)
12. 4 Bytes Header; length, destination node ID, source node ID and control Byte (for ACK request and reply)
    - 255 unicast node addresses, with one broadcast address (default 255)
13. Basic Network layer routing implementation using header source and destination node address detection
14. Basic Transport layer implementation through optional usage of a acknowledge request and reception control byte in datagrams
15. Encoding / Decoding AES-128 application option
16. Payload length in packet mode should not exceed the FIFO length of 66 Bytes
    - Packets longer that the maximum payload length are rejected
17. Data message length in a variable frame length, taking into account the AES limitations (not conform frames are rejected by the RFM69 hardware) are limited to:
    - With no node filtering, the **maximum user Data is 61 Bytes** (64 – 3 header Bytes; the length field is not part of the data).
    - With node filtering, the **maximum user Data is 46 Bytes** (48 – 2 header Bytes; the length and the node address are not part of the data)
18. No Node ID and broadcast filtering done by RFM69 hardware, actual filtering is done by the receiver function
    - The following nodes screening are possible:

[2] http://lowpowerlab.com

    - Promiscuous mode, i.e., no node ID nor broadcast are filter-out, all packets of the same network are accepted
    - Unicast/ Broadcast screening: matching the receiver node ID or broadcast address.

19. Hardware Node ID and broadcast filtering are possible though transceiver registers configuration

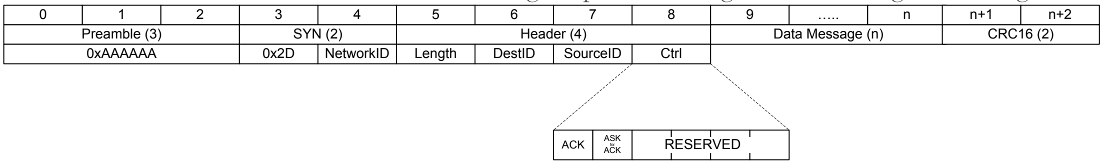

Figure 8: Moteino RFM69 frame format

RFM69 library[3]

The RFM69 library allows the Moteino board to communicate with RFM69 transceiver through the SPI bus as a SPI Master.

Signal	Moteino	Moteino Mega
SCK	13	7
MISO	12	6
MOSI	11	5
SS	10	4

Table 1: MOTEINO SPI I/O pins configuration

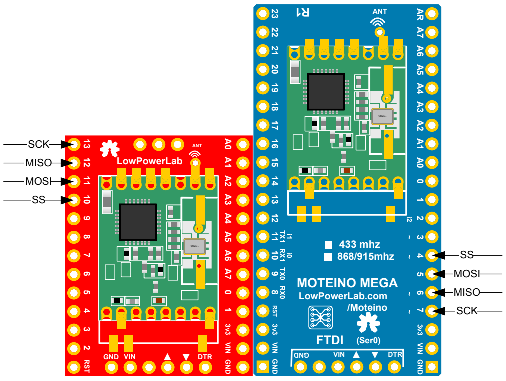

Figure 9: Moteino and Moteino MEGA SPI pin connections

RFM variables and constants

Configuration Constants

- Operation Modes
    - RF69_MODE_SLEEP
      RF69 power consumption: ~1µA
    - RF69_MODE_STANDBY
      RF69 power consumption: ~1,5mA
    - RF69_MODE_SYNTH
      RF69 power consumption: ~9mA
    - RF69_MODE_RX
      RF69 power consumption: ~16mA
    - RF69_MODE_TX
      RF69 power consumption: 16 … 45mA
- Carrier Frequencies
    - RF69_FSTEP
      Frequency synthesizer step: Default 32MHz/2^19 (=61,03515625)
    - RF69_315MHZ
      ISM frequency range 315MHz

[3] See https://github.com/LowPowerLab/RFM69

    - RF69_433MHZ
      ISM frequency range 433MHz
    - RF69_868MHZ
      ISM frequency range 868MHz
    - RF69_915MHZ
      ISM frequency range 915MHz
- Payload Length
    - RF69_MAX_DATA_LEN
      Maximum Data payload length in Bytes: Value is 61
- SPI Configuration
    - RF69_SPI_CS
      SPI Slave Select PIN: Value is; see Table 1: MOTEINO SPI I/O pins configuration
    - RF69_IRQ_PIN
      Interrupt request pin number: Value is pin 2
    - RF69_IRQ_NUM
      Interrupt request number: Value is 0
- Media Access
    - CSMA_LIMIT
      Carrier Sense or RSSI threshold for clear channel before transmitting data: Value is -90 dBm
    - RF69_CSMA_LIMIT_MS
      Excessive Carrier Sense watchdog timer: Value is 1000ms
    - RF69_TX_LIMIT_MS
      Transmission watchdog timer: Value is 1000ms
- Network Access
    - RF69_BROADCAST_ADDR
      Destination Node ID broadcast address: Value is 255
- RF69 CMOS temperature sensor control
    - COURSE_TEMP_COEF
      Temperature reading realistic offset: Value is -90 see Temperature calculation on page 19

Working Variables

- Operation Modes
    - volatile uint8_t RFM69::_mode
      Current transceiver power operating mode
- Payload and Data
    - volatile uint8_t RFM69::DATA[RF69_MAX_DATA_LEN]
      Transfer buffer array
    - volatile uint8_t RFM69::PAYLOADLEN
      Payload length
    - volatile uint8_t RFM69::DATALEN
      Effective data length (PAYLOADLEN-3)
- Network Access
    - volatile uint8_t RFM69::SENDERID
      Sender Node ID
    - volatile uint8_t RFM69::TARGETID
      Receiver Node ID
- Transport Control
    - volatile uint8_t RFM69::ACK_REQUESTED
      Control Byte Acknowledge request
    - volatile uint8_t RFM69::ACK_RECEIVED;
      Control Byte Acknowledge Received
- Media Access
    - volatile int16_t RFM69::RSSI;
      Last reception RSSI value

Moteino RFM69 Library

RFM69 Class

Syntax

RFM69
RFM69.initialize
RFM69.send
RFM69.sendWithRetry
RFM69.receiveDone
RFM69.ACKReceived
RFM69.encrypt
RFM69.promiscuous
RFM69.setFrequency
RFM69.getFrequency
RFM69.setAddress
RFM69.setCS
RFM69.readRSSI
RFM69.setPowerLevel
RFM69.setHighPower
RFM69.sleep
RFM69.readReg
RFM69.witeReg
RFM69.readAllRegs
RFM69.readTemperature
RFM69.rcCalibration

Moteino RFM69 Library

RFM69

Description

A call to RFM69 creates a RFM69 object, whose name needs to be provided while calling the class.

Notes

While creating an RFM69 instance the RFM69 the output power (for RFM69W and RFM69HW) is set by default to 31.

Syntax

RFM69;
RFM69 (uint8_t slaveSelectPin, uint8_t interruptPin, bool isRFM69HW, uint8_t interruptNum);

Parameters

- *none:*
  Default slaveSelectPin is RF69_SPI_CS, default interruptPin is RF69_IRQ_PIN, default isRFM69HW is false (RFM69W), default interruptNum is RF69_IRQ_NUM.
- *slaveSelectpin:*
  Used to select an SS pin. Default is determined by the hardware,
  interruptPin:
  Used to select an interrupt pin. By default defined by RFM69.h, i.e. 2 for ATmega328P, ATmega644P ATmega1284P
- *isRFM69HW:*
  Used to indicate the type of RFM69 module (normal RFM69W or high power RFM69HW). Default is false (i.e. RFM69W)
- *interruptNum:*
  Used to select the interrupt number. The default from RFM69.h is 0

See Table 1: MOTEINO SPI I/O pins configuration on page 4.

Example

```
RFM69 radio;
RFM69 radio(10,2,false,0);    // Setup one RFM69W on Moteino instance with SS on pin 10 and an IRQ pin 2, IRQ
number 0
```

RFM69.initialize

Description

This public function initializes the RFM69 registers and returns a Boolean status "true" when terminated. This function must be invoked before any attempt to send or receive data.

Syntax

RFM69.initialize(uint8_t freqBand, uint8_t ID, uint8_t networkID)

Parameters

- *freqBand:*
  Frequency band to use, this one should match the module specification. Possible values are:
  - RF69_433MHZ
  - RF69_868MHZ
  - RF69_915MHZ
- *ID:*
  - The Node Identification number. Possible values are from 0 to 254 with 255 as the broadcast node ID.
- *networkeID:*
  - The Network Identification number. Possible values are fro 0 to 255

Returns

True when initialization is terminated

Note

The node ID is not saved into the NodeAddress Register (RegNodeAdrs), because address filtering is by default not used.

Example

```
#define NODEID      99
#define NETWORKID   100
#define FREQUENCY   RF69_433MHZ
...
radio.initialize(FREQUENCY,NODEID,NETWORKID);
```

RFM69.canSend

Description

This Boolean function senses the RF channel, in particular by verifying that the RSSI level is lower than the CSMA_LIMIT.

Syntax

RFM69.canSend()

Parameters

None

Returns

True if the RSSI level is less than the CSMA_LIMIT

See also

RFM69.send
RFM69.sendACK

Notes

1. CSMA_LIMIT (currently -90dBm) is a constant, that can't be modified by the sketch
2. This function is used by the *send* and *sendACK* functions
3. This function should NOT BE USED separately because it requires the transceiver in RX mode. This one is per default in STBY mode.

Example

```
uint32_t now = millis();
while (!canSend() && millis() - now < RF69_CSMA_LIMIT_MS) receiveDone();// The
receiveDone function ensures that the transciever is in receive mode while testing the
candSend conditions
```

RFM69.send

Description

This function checks for a clear channel (*canSend*) for duration less than RF69_CSMA_LIMIT_MS, before attempting to send (for a duration that should not exceed RF69_TX_LIMIT_MS) the contents of a data buffer (length is restricted to the RF69_MAX_DATA_LEN). Setting the acknowledgement request bit of the control Byte is optional.

Syntax

RFM69.send(uint8_t toAddress, const void* buffer, uint8_t bufferSize, bool requestACK)

Parameters

- *toAddress:*
  The destination node ID
- **buffer:*
  A pointer to the data buffer
- *bufferSize:*
  The size of the relevant data part of the buffer. Data will be truncated to the RF69_MAX_DATA_LEN.
- *requestACK:*
  To enforce the acknowledge request bit of the control Byte

Returns

None

Notes

1. RF69_MAX_DATA_LEN (currently 61 Bytes) is a constant, that can't be modified by the sketch, however sending data with a Node Address Filtering configuration and encryption should limit the User data to 46 Bytes, the sketch has to take care of this limitation. See Node Address Filtering on page 18.
2. RF69_BROADCAST_ADDR (currently 255) is a constant, that can't be modified by the sketch unless using Node Address Filtering configuration and *promiscuous*
3. RF69_CSMA_LIMIT_MS (currently 1000ms) is a constant, that can't be modified by the sketch
4. RF69_TX_LIMIT_MS (currently 1000ms) is a constant, that can't be modified by the sketch.

See also

RFM69.sendWithRetry

Example

```
radio.send (0,"Hello node 0", 12,0);  // Send 12 Bytes of the string "Hello node 0" to node 0 without ACK request
```

or

```
#define NODEID          99
#define NETWORKID       100
#define COORDINATORID   1
. . . . .
typedef struct
{
  Int                   nodeId; //store this nodeId
  unsigned long         uptime; //uptime in ms
  float                 temp;   //temperature maybe?
} Payload;

Payload theData;

theData.nodeId = NODEID;
theData.uptime = millis();
theData.temp = 91.23;
 . . . . . . . .
radio.send(COORDINATORID, (const void*)(&theData), sizeof(theData), true);
```

RFM69.sendWithRetry

Description

This is a Boolean function that checks for a clear channel (*canSend*), before attempting to send (for a duration that should not exceed RF69_TX_LIMIT_MS) the contents of a data buffer (length is restricted to the RF69_MAX_DATA_LEN), while setting the acknowledgement request bit of the control Byte. It then waits during an optional time for the reception of an acknowledgement from the target node before attempting to retry an optional number of times.

Syntax

RFM69.sendWithRetry(uint8_t toAddress, const void* buffer, uint8_t bufferSize, uint8_t retries=2, uint8_t retryWaitTime=40)

Parameters

- *toAddress*:
  The destination node ID
- **buffer*:
  A pointer to the data buffer
- *bufferSize*:
  The size of the relevant data part of the buffer. Data will be truncated to the RF69_MAX_DATA_LEN.
- *retires*:
  The number or retries for ACK reception (maximum 255), default is 2 (= 3 times in total)
- *retryWaitTime*:
  The time in ms between each retries if no acknowledge was received (maximum 255ms), default is 40ms

Returns

True if acknowledge was received; else, false.

See also

RFM69.send
RFM69.ACKReceived

Notes

1. RF69_MAX_DATA_LEN (currently 61 Bytes) is a constant, that can't be modified by the sketch, however sending data with a Node Filtering configuration and encryption should limit the User data to 46 Bytes, the sketch has to take care of this limitation. See Node Address Filtering on page 18
2. RF69_BROADCAST_ADDR (currently 255) is a constant, that can't be modified by the sketch unless using Node Address Filtering configuration and *RFM69.promiscuous*
3. RF69_CSMA_LIMIT_MS (currently 1000ms) is a constant, that can't be modified by the sketch
4. RF69_TX_LIMIT_MS (currently 1000ms) is a constant, that can't be modified by the sketch
5. Wait Time before retries(default is 40ms):
   With the maximum frame length of 72 Bytes at a bit rate of 55.55Kbps the duration is about 10ms.
   With the minimum frame length of 11 Bytes at a bit rate of 55.55Kbps the duration is about 1,5ms.
   With the worst case that an acknowledgement contains data, the reply should take at most 10ms (excluding; processing time, busy channels, etc.…) so 40ms is reasonable. WARNING: A too fast retries may generates duplicated data packets which requires the implementation of a packet sequencing numbering

Example

```
  if (radio.sendWithRetry(1,"Hello node 1",12))
    Serial.print(" ok!");
  else Serial.print("No acknowledge received");
```

RFM69.sendACK

Description

A variant of the *send* function that sends to the last received source node ID a data packet with the control byte bit set to ACK.

Syntax

RFM69.sendACK(const void* buffer = "", uint8_t bufferSize=0)

Parameters

- **buffer:*
  A pointer to the acknowledgement data buffer. By default data is empty
- *bufferSize*:
  The size of the relevant data part of the buffer. Data will be truncated to the RF69_MAX_DATA_LEN. By default the size is set to 0.

Returns

None

See also

RFM69.send
RFM69.ACKReceived

Note

The original RSSI value of the last received frame is preserved, by this function

Example

```
if (radio.ACKRequested())
    {
      radio.sendACK();
      Serial.println(" - ACK sent");
    }
```

RFM69.receiveDone

Description

This function sets the RFM69 in receiving mode, allowing RFM69 reception interrupt.

Moteino RFM69 Library

Once interrupt is usefully terminated (on correctly formatted frame) data of the received frames are accessible through the working variables.

Syntax

RFM69. receiveDone()

Parameters

None

Example

```
  if (radio.receiveDone())
  {
    Serial.print('[');Serial.print(radio.SENDERID, DEC);Serial.print("] ");
    Serial.print(" [RX_RSSI:");Serial.print(radio.readRSSI());Serial.print("]");
    Serial.print(" to [");Serial.print(radio.TARGETID, DEC);Serial.print("] ");
    Serial.print(" ACK Requested: "), Serial.print(radio.ACKRequested());
    Serial.print(" Data Length: "), Serial.print (radio.DATALEN, DEC);
    Serial.print(" Data: "); for (byte i = 0; i < radio.DATALEN; i++)
Serial.print((char)radio.DATA[i]);
    Serial.println();
  }
```

See also

RFM69.ACKReceived

RFM69.ACKReceived

Description

Boolean function that sets the RFM69 transceiver in receiving mode (see *receiveDone*) and checks the frame reception for the ACK bit of control Byte from the sender node address (that was specified as parameter) or from the broadcast address.

Syntax

RFM69.ACKReceived(uint8_t fromNodeID)

Parameters

- *fromNodeID*:
  Node ID for which a ACK is expected

Returns

True if expected ACK is received; else, false

See also

RFM69.receiveDone

Example

```
radio.send(1,"Hello node 1",12,1);
while (!radio.ACKReceived(1));
Serial.println ("ACK Received");
```

RFM69.encrypt

Description

Use of this function enables the data message encryption part of the payload. Data are encrypted when posted to the RFM69 FIFO during the *send* / *sendWithRetry* functions.

Syntax

RFM69.encrypt(const char* key);

Parameters

- **key*:
  Cypher bytes string array that contains exactly **16 characters** use to encrypt the data

Returns

- True if expected ACK is received, else; false

Notes

1. The data message length is limited to 61 user data Bytes in case of No Node address filtering and 46 user Bytes in case of Node address filtering. Destination address, Control Byte and optionally Source Node Address (depending of the Node filtering settings), which are not part of the user data are also encrypted.
2. See *send*, *sendWithRetry* and *sendACK* functions, protect the maximum user data length by a constant RF69_MAX_DATA_LEN (currently 61 Bytes), this one can't be modified by the sketch, however sending data with a node filtering configuration AND encryption should limit the user data to 46 Bytes, the sketch has to take care of this limitation.

Example

```
#define KEY          "thisIsEncryptKey"
. . .
radio.encrypt(KEY);
radio.send(1,"Hello node 1",12,0);
```

RFM69.promiscuous

Description

Setting the promiscuous mode allows reception of any node destination datagram within the current network.
This function disables in the receiver interrupt handler the test of matching the current node ID with the destination node ID of the payload.

Syntax

RFM69.promiscuous(bool onOff=true);

Parameters

- *onOff*:
  State of the promiscuous option. Default is true (ON) or while calling this function.

Note

This function is not working if node filtering is configured, because the destination node ID matching is done by the RF69 transceiver.

Example

```
bool promiscuousMode = true; //set to 'true' to sniff all packets on the same network.
radio.promiscuous(promiscuousMode);
```

RFM69.setFrequency

Description

Allow changing the RFM69 RF carrier frequency register(s) using a chosen frequency in Hertz

Syntax

RFM69.setFrequency(uint32_t freqHz)

Parameters

- *freqHz*:
  The chosen frequency in Hertz

Returns

None

See also

RFM69.getFrequency

Note

Changing the frequency should be done according to the specific module frequency tolerance, see RFM69 specifications (typically: RFM69W-V1.3.pdf *§2.4.2. Frequency Synthesis, Table 5 Frequency Synthesizer Specification*)

Example

```
radio.setFrequency (432000000);
```

Moteino RFM69 Library

RFM69.getFrequency

Description

Dump the RF69 module RF Frequency carrier register and convert it in to Hertz value

Syntax

RFM69.getFrequency()

Parameters

None

Returns

None

See also

RFM69.setFrequency

Example

```
Serial.print("RF Frequency: "), Serial.println (radio.getFrequency());
```

RFM69.setAddress

Description

Allow changing the node address after initialization.

Syntax

RFM69. setAddress(uint8_t addr)

Parameters

- *addr*:
  The node address to filter, after

Returns

None

See also

RFM69.initialization

Note

Setting the node address also changes the RFM69 *NodeAddress* register, however, the *AdressFiltering* option bits are not affected, and therefore the Node address / Broadcast address filtering is not activated.

Example

```
#define NODEID      99
#define NETWORKID   100
#define FREQUENCY   RF69_433MHZ
...
radio.initialize(FREQUENCY,NODEID,NETWORKID);
radio.setAddress (200); // Change node address from 99 to 200
```

RFM69.setCS

Description

Allow changing the RFM69 SPI slave select pin.

Syntax

RFM69.setCS(uint8_t newSPISlaveSelect)

Parameters

- *newSPISlaveSelect*:
  The new pin used as RFM69 SPI slave Select

Returns

None

See also

RFM69.initialization

Note

This pin is hardwired according to the Moteino model (see Table 1: MOTEINO SPI I/O pins configuration).
WARNING: Changing this pin requires also a hardware modification.

Example

```
radio.setCS (10); // Change the Slave Select from 10 (default) to 10!!!
```

RFM69.readRSSI

Description

This function is used to dump the Received Signal Strength Indicator. This value currently saved by the RFM69 transceiver, after each data reception.
A dedicated test of the RSSI may be forced using the "forceTrigger" option.
One RFM parameter is the RSSI threshold. If this threshold is reached, the transceiver will NOT send a frame. By default this value is set by the library to -110 dBm, (the maximum is -114dBm), this means that data may be send if the RSSI is above the threshold.

Syntax

RFM69.readRSSI(bool forceTrigger=false)

Parameters

- *forceTrigger:*
  - false: The function returns the last RSSI value in dBm
  - true: The function starts the RSSI sampling until the value is above the RFM69 RSSI threshold

Returns

- A negative word value that correspond to the RSSI value in dBm

See also

RFM69.cantSend

Note

The *candSend* function uses software variable CSMA_LIMIT to validate the transmission, this value should be above the RSSI threshold (typically -100 or -90 dBm)

Example

```
Serial.print ("RSSI : "), Serial.println (radio.readRSSI(false));
Serial.print ("RSSI triggered: "), Serial.println (radio.readRSSI(true));
```

RFM69.setPowerLevel

Description

Used to modify the sender output power level for RFM69W and RFM69HW.
Acceptable values are from 0 to 31 for RFM69W / RFM69HW, however the actual values or RM69HW is 0 to 15. The function automatically divides the power parameter by 2. (0 = 0; 2 = 1, ….30 = 15)

Syntax

RFM69.setPowerLevel (uint8_t level)

Parameters

- *level:*
  The power level value to apply;
  - RFM69W and RFM69HW (low power): 0 corresponds to -18 dBm and 31 to +13dBm. Default is 31 (or +13dBm)
  - RFM69HW (with high power set): 0 correspond to +2dBm and 30 to +17dBm. Default is 31 (or +17dBm)

Returns

None

See also

RFM69.setHighPower

Note

Decreasing the power level results in a "weaker" transmitted signal, and directly results in a lower RSSI at the receiver, and decreases the operational distances between sender and receiver.

Setting the power level to different value may be used in a test environment for instance.

Example

```
radio.setPowerLevel(10); // Set the output power level to -8 dBm for a RFM69W
```

RFM69.setHighPower

Description

Activate the RFM69HW transceiver sender output power amplifiers.

Syntax

RFM69.setHighPower(bool onOff=true)

Parameters

- *onOff:*
  - false: for a RFM69W, power amplifier not activated
  - true: for a RFM69HW, power amplifier are activated.

Returns

None

See also

RFM69.setPowerLevel

Notes

It must be set to ON for a RFM69HW transceiver.
The actual power value MUST be adjusted by using the *setPowerLevel* function.
Decreasing the power level results in a "weaker" transmitted signal, and directly results in a lower RSSI at the receiver, or decreases the operational distances between sender and receiver.

Setting the power level to different value may be used in a test environment for instance.

Example

```
/* Transceiver is a RFM69HW
radio.setHighPower(); // Activate the power amplifiers
radio.setPowerLevel(16);       // Set the output power level to 17 dBm
```

RFM69.sleep

Description

Used to set the RFM69 transceiver is low consumption mode, with all oscillators disabled.

Syntax

RFM69.sleep()

Parameters

None

Returns

None

Note

To wake-up from a sleep mode a send or receive command has to be initiated

Example

```
radio.sleep();          // All RFM activities are stopped
```

RFM69.readReg

Description

Read a RFM69 register. See RFM69 datasheets: 6. Configuration and Status Registers

Syntax

RFM69.readReg(uint8_t addr)

Parameters

- *addr*:
  One RW69 register address fro 0x00 to 0x7F

Returns

Register value

Example

```
radio.readReg(0x01);          Read the Operation Mode Register
```

RFM69.readAllRegs

Description

Dump to the serial interface the contents of the RFM69 registers from 0x01 to 0x4F (with the exception of the Test Registers). See datasheets: 6. Configuration and Status Registers.

Syntax

RFM69.reaAllRegs()

Parameters

None

Returns

All Registers value

Example

```
radio.readAllRegs();          Read the RFM69 registers

<1 - 4 - 100
2 - 0 - 0
3 - 2 - 10
4 - 40 - 1000000
5 - 3 - 11
6 - 33 - 110011
7 - 6C - 1101100
8 - 40 - 1000000
9 - 0 - 0
A - 41 - 1000001
B - 0 - 0>
```

RFM69.witeReg

Description

Modify the contents of a RFM69 register. See datasheets: 6. Configuration and Status Registers

Syntax

RFM69.writeReg(uint8_t addr, uint8_t value)

Parameters

- *addr*:
  The register address; valid value are from 0x00 to 7F
- *value*:
  The value to write to the specified register

Returns

None

Example

```
#define REG_PACKETCONFIG1     0x37    // Packet configuration 1 register address
#define RF_PACKET1_ADRSFILTERING_NODEBROADCAST  0x04 //Node and broadcast filter
       address mask
…….
radio.writeReg(REG_PACKETCONFIG1, radio.readReg (REG_PACKETCONFIG1) |
       RF_PACKET1_ADRSFILTERING_NODEBROADCAST); // Activate the Node filtering mode
```

RFM69.readTemperature

Description

Gives an approximate value of the internal CMOS temperature. This temperature influences the accuracy of the RC oscillator. This one should be recalibrate (see *rcCalibration*) regularly for large temperature variations.

Syntax

RFM69. readTemperature(uint8_t calFactor=0)

Parameters

- *calFactor*:
  The correction factor to applied to the temperature measurement allowing the measurement to reflect the actual CMOS temperature

Returns

CMOS relative temperature in °C

See also

RF69M.rcCalibration

Note

Because the result of temperature reading is inaccurate, it should be corrected by a calibration factor (specific for each transceiver) that tunes the actual reading with the ambient temperature. The calibration factor is obtained by comparing the read temperature with the ambient one when the transceiver is powered on.

Example

```
byte cmosTempCur;      // Stores the CMOS current temperature read
byte cmosTempPrev;     // Stores the CMOS previous temperature read
byte calTemp = 2;      // Calibration factor to correct the CMOS actual temperature
byte tempThreshold = 3;// Temperature delta for RC recalibration

void setup()
{
Serial.begin(SERIAL_BAUD);
Serial.print ("Raw CMOS Temperature: ");
Serial.println (cmosTempCur = radio.readTemperature()); // Print the temperature
without correction
cmosTempPrev = radio.readTemperature(calTemp);       // Save the corrected temperature
Serial.println ("Corrected CMOS Temperature: ");
Serial.println (cmosTempPrev),                       // Print the corrected temperature
delay(5000);

}

void loop ()
{
cmosTempCur = radio.readTemperature(calTemp);        // Check the current temperature
Serial.print ("Previous measured temperature :");
Serial.println (cmosTempPrev);               // Print the previous temperature
Serial.print ("Current temperature is :");
Serial.println (cmosTempCur);                // Print the current temperature
if (abs(cmosTempCur - cmosTempPrev)>=tempThreshold ) // Check if the current
temperature is not above or below the threshold
        {
                Serial.println ("RC Calibration in progress");
                radio.rcCalibration();
                cmosTempPrev = cmosTempCur;   // Save the new reference value
        }
        delay (1000);
}
```

RFM69.rcCalibration

Description

Calibrate the internal RC oscillator when used in wide temperature variations environment - see datasheet section [4.3.5. RC Timer Accuracy].

Syntax

RFM69.rcCalibration()

Parameters

None

Returns

None

See also

RFM69.readTemperature

Note

"For applications enduring large temperature variations, and for which the power supply is never removed, RC calibration" can be performed upon user request".

Example

```
radio.rcCalibration(); // start RC oscillator calibration
```

Node Address Filtering

Node address and Broadcast address filtering is inbuilt feature of the RFM69 transceiver; however, this feature is not used by the current library. To activate this function a dedicate register setup is necessary. Note that the promiscuous and the address filtering are mutually exclusive, in other words, promiscuous is only pertinent if node address filtering is not activated. The table below summarizes the option.

Node Address Filtering	Promiscuous Mode	Actual Node Filtering Activation
No	No	No
No	Yes	No
Yes	No	Yes
Yes	Yes	No

Note that the broadcast address should always be 255 to cope with the library definition

See Sketch below as example.

Data Encryption and Node Address Filtering

Once node address filtering is activated, the size of the used data field is reduced (from 61 to 46), the constant RF69_MAX_DATA_LEN which is controlling the size of the data to send is only valid for packet that are encrypted or not in a non node filtering configuration. The user sketch shall verify that in a node filtered configuration, the actual data is no longer than 46 Bytes.

Node Address Filtering	Encryption Mode	User Data packet length [Bytes]
No	No	64
No	Yes	64
Yes	No	64
Yes	Yes	46

See Sketch below as example.

Node Address filtering and Data Encryption

```
#include <RFM69.h>

#define NODEID                              1            // Local Node ID
#define NETWORKID                           100          // Local Network ID
#define FREQUENCY                           RF69_433MHZ  // Node working frequency
#define RF69_BROADCAST_ADDR                 255          // Node broadcast ID
#define REG_PACKETCONFIG1                   0x37         // Packet configuration 1 register address
#define REG_NETWADRS                        0x30          // Network ID address register
#define REG_NODEADRS                        0x39          // Node ID register address
#define REG_BROADCASTADRS                   0x3A          // Broadcast address register address
#define RF_PACKET1_ADRSFILTERING_NODEBROADCAST 0x04       // Node and broadcast filter address mask

#define SERIAL_BAUD                         115200         // Serial baud rate
#define DEBUG                               1              // Debugging option
#define KEY          "thisIsEncryptKey"

#define dataEncryption                       0
#define promiscuousMode                      0
#define nodeFiltering                        0

RFM69 radio;                                // Declare a radio instance

boolean filterState;
```

```
void setup()
{
  Serial.begin(SERIAL_BAUD);

  if (promiscuousMode & nodeFiltering)          // Actual Node Filtering state check
  {
          filterState = false;
  }
  else filterState = nodeFiltering;

  nodeInitialisation();                         // Initialize the node
  Serial.println ("Begin Receive");
  Serial.print ("Promiscuous Mode: "), Serial.print (promiscuousMode), Serial.print (" Filter State: "),
Serial.print (filterState ), Serial.print(" Encryption: "), Serial.println (dataEncryption);
  Serial.print("RF Frequency: "), Serial.println (radio.getFrequency());
 radio.readTemperature();
  Serial.print("Tempertature: "), Serial.println (radio.readTemperature(),HEX);

 delay(5000);

}

void loop ()
{
if (radio.receiveDone())
  {
    Serial.print('[');Serial.print(radio.SENDERID, DEC);Serial.print("] ");
    Serial.print(" [RX_RSSI:");Serial.print(radio.readRSSI(false));Serial.print("]");
    Serial.print(" to [");Serial.print(radio.TARGETID, DEC);Serial.print("] ");
    Serial.print(" ACK Requested: "), Serial.print(radio.ACKRequested());
    Serial.print(" Data Length: "), Serial.print (radio.DATALEN, DEC);
    Serial.print(" Data: "); for (byte i = 0; i < radio.DATALEN; i++) Serial.print((char)radio.DATA[i]);
    Serial.println();
    if (radio.ACKRequested())
    {
      radio.sendACK();
      Serial.println(" - ACK sent");
    }
  }
}
void nodeInitialisation()
{
    radio.initialize(FREQUENCY,NODEID,NETWORKID);
    if (filterState)
    { // If not in promiscuous mode do filter the node and the broadcast addresses via hardware
    if (DEBUG) Serial.print ("Network ID: "), Serial.println (radio.readReg (REG_NETWADRS));
    radio.writeReg(REG_PACKETCONFIG1, radio.readReg (REG_PACKETCONFIG1) | RF_PACKET1_ADRSFILTERING_NODEBROADCAST);
    if (DEBUG) Serial.print("Packet Configuration Register 1: "), Serial.println (radio.readReg
(REG_PACKETCONFIG1),HEX);
    radio.writeReg(REG_NODEADRS,NODEID);
    if (DEBUG) Serial.print("Node Address Register: "),Serial.println (radio.readReg (REG_NODEADRS),HEX);
       radio.writeReg(REG_BROADCASTADRS,RF69_BROADCAST_ADDR);
    if (DEBUG) Serial.print("Node Broadcast Register: "),Serial.println (radio.readReg (REG_BROADCASTADRS),HEX);
    }
    if (dataEncryption) radio.encrypt(KEY);
    radio.promiscuous(promiscuousMode);          // activate the promiscuous mode if necessary
}
```

Temperature calculation

Actual temperature measurement is an inverse ramp; while temperature is increasing the measured value decreases. The slop is one digit by °C.
From measurement; 20°C gives a measurement of 145, while 24°C gives a result of 141. Corollary, a value of 255 indicates a temperature of -90°C and 0 indicates a temperature of +165°C.
To have results that are increasing with a rising the temperature, we have to reverse the slope by inverting the value. In this example 20°C is equal to 255-145=110 and 24°C is 255-141 = 114.
To express this value in actual degree, we have to subtract 90 of the value; 110-90 = 20°C and 114-90 = 24°C.

-ooOoo-