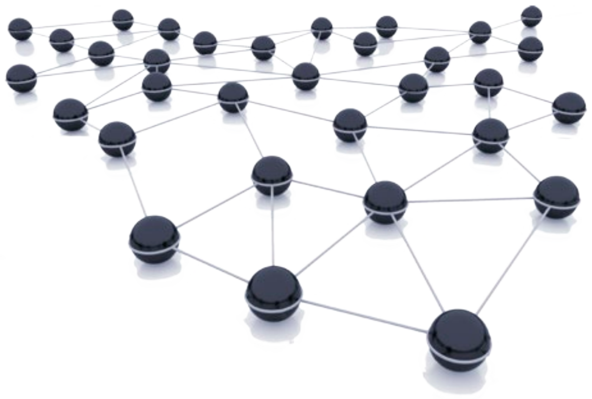
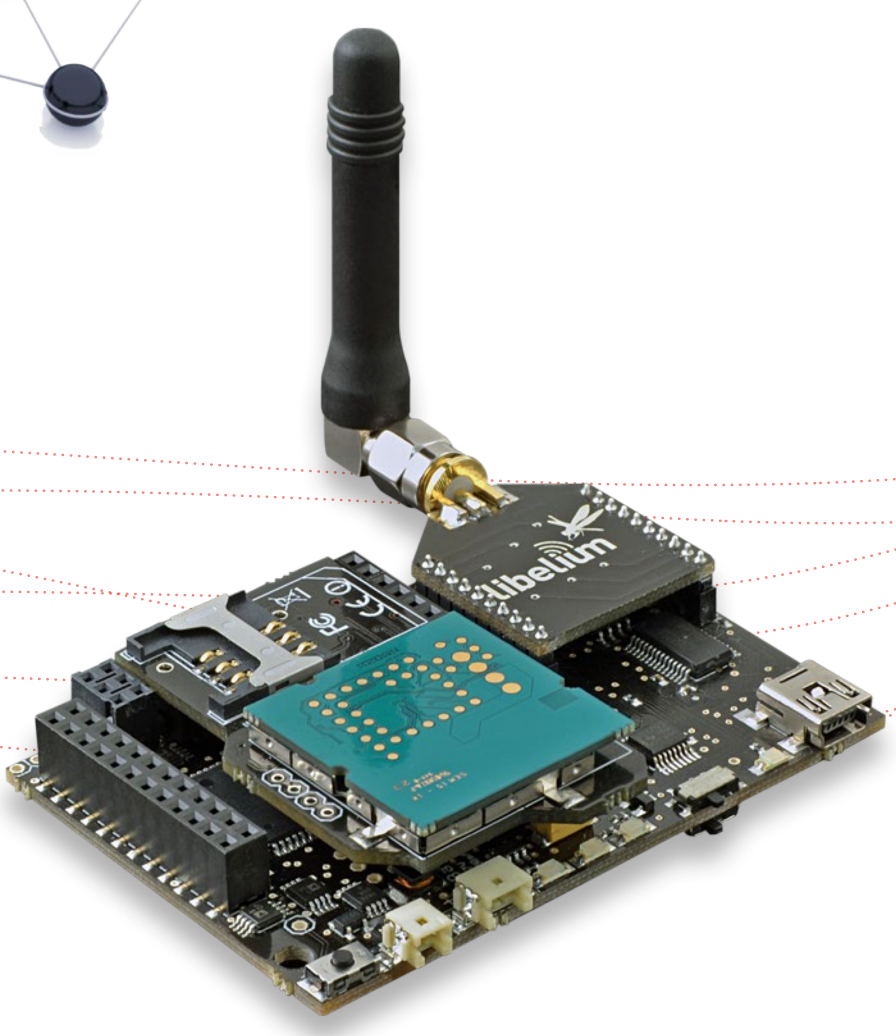
# RFID/NFC 13.56MHz

## Networking Guide





Document version: v4.2 - 02/2014

# INDEX

# 1. Introduction

RFID (Radio Frequency Identification) is a technology that uses electromagnetic fields to identify objects in a contactless way; it is also called proximity identification. There are 2 elements in RFID communications: the RFID module (or reader/writer device) and an RFID card (or tag). The RFID module acts as the master and the card acts as the slave; this means the module queries the card and sends instructions to it. In a normal RFID communication, the RFID module is fixed and the user takes his card near it when he needs to start the interaction.

*Figure : Waspmote with RFID/NFC module on socket 0 (below antenna) and ZigBee radio on socket 1*

An RFID card can be understood as a remote storage unit where we can read and write information without contact. Most of the RFID tags are passive, which implies that the RFID module must create an electromagnetic field in order to power the tag. The RFID card's antenna (in fact it is an inductive coupler) gets the power from this field. Also, an RFID card has a very basic micro-controller which manages the communications and memory access.

Many RFID standards have been released by the industry. Most of them have as operating frequency 13.56 MHz or 125 KHz. Libelium has integrated one module for each frequency; for more information read chapter "RFID: 13.56 MHz or 125 KHz?". In the present document we will be explaining the RFID/NFC module, which works at 13.56 MHz, and any reference to RFID must be understood as RFID/NFC at 13.56 MHz.

In particular, Libelium has created an ISO/IEC 14443-A and NFC compliant module for Waspmote. The ISO/IEC 14443-A protocol is widely accepted as the *de facto* RFID at 13.56 MHz standard. Billions of ISO/IEC14443-A cards have been sold over the world. NFC (Near Field Communication) is a extension of RFID which focuses on communications between smartphones and other advanced devices. NFC is a set of standards based on previous RFID protocols like ISO/IEC 14443-A.

There are basically 3 ways to interact with an RFID card; Libelium's RFID/NFC module allows the developer to implement the 3 of them:

1. the RFID/NFC module reads the RFID card's unique identification (UID)
2. the RFID/NFC module reads the RFID card's internal memory (16 bytes each time)
3. the RFID/NFC module writes in the RFID card's internal memory (16 bytes each time)

Among the RFID/NFC applications, the most common are:

- access control/security
- events ticketing
- public transport
- equipment and personnel tracking
- logistics
- real-time inventories
- marketing information (kiosks)

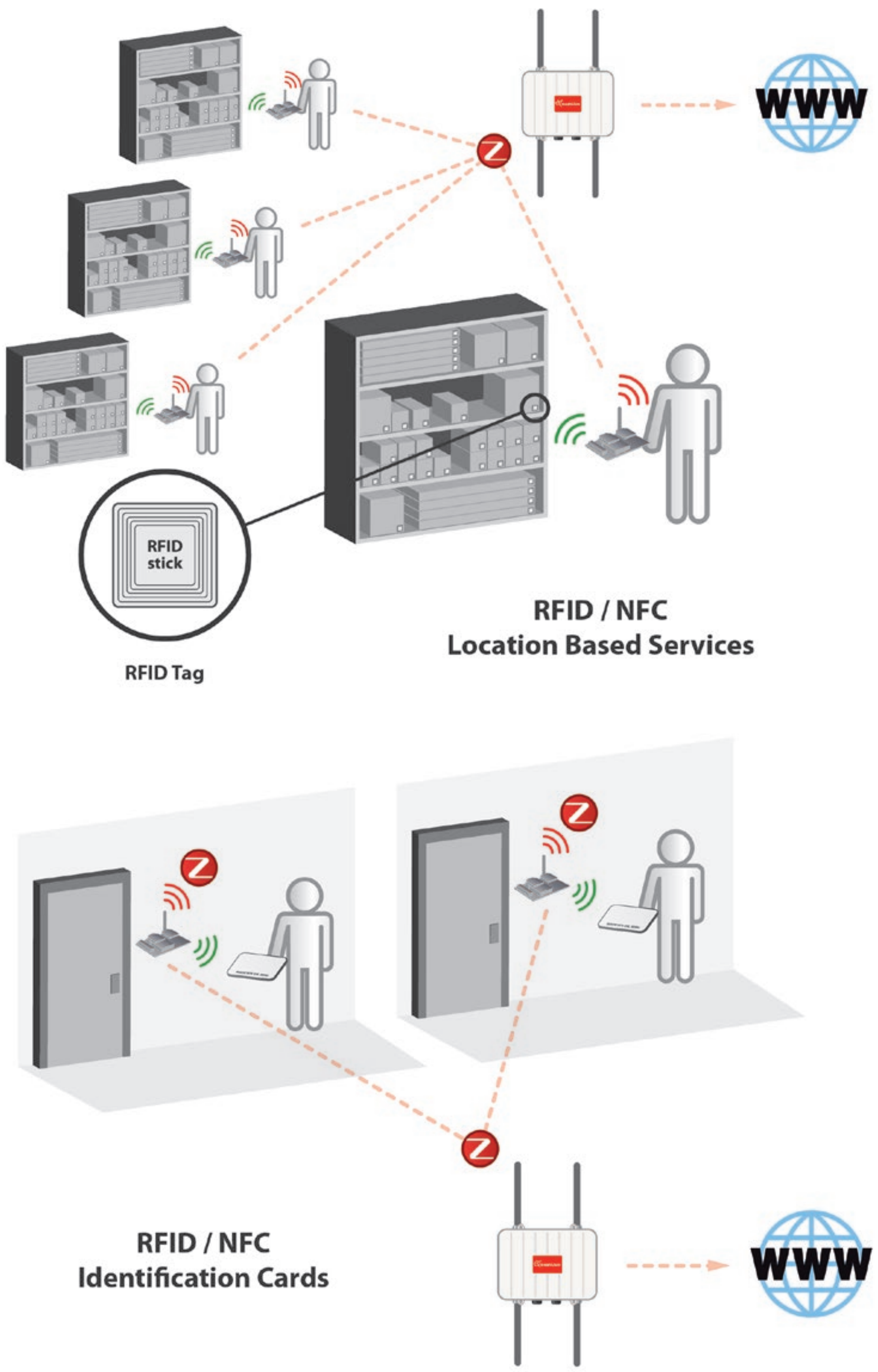


*Figure : Some application examples*

# 2. Hardware

Libelium's RFID/NFC module integrates an RFID/NFC reader/writer integrated circuit.

**Features**

- **Compatibility:** Reader/writer mode supporting ISO 14443A / MIFARE / FeliCaTM / NFCIP-1
- **Distance:** 5cm
- **Max capacity:** 4KB
- **Tags:** cards, keyrings, stickers

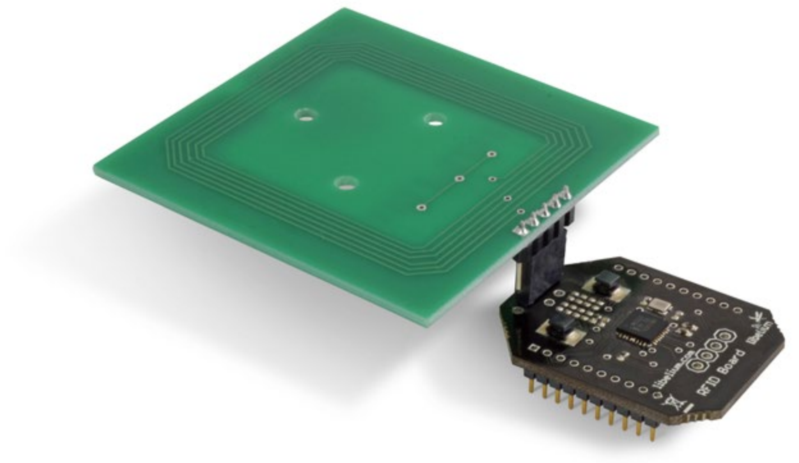


*Figure : RFID/NFC module for 13.56 MHz and antenna*

Among several RFID standards; Libelium's RFID/NFC module is compliant with ISO/IEC 14443-A/Mifare®, which probably are the most successful international standards in RFID technology. Mifare® is a proprietary technology by NXP which is based on the ISO/IEC 14443-A standard. This technology operates at the 13.56 MHz ISM frequency band, so it can be deployed anywhere without authorization or license.

The RFID/NFC module implements the radio-frequency, analog and digital tasks in transmission and reception, while Waspmote's role is to control these operations. Waspmote is connected to the RFID/NFC module through its UART interface.

The ISO/IEC 14443-A/Mifare® standard offers a 3-pass authentication method to enable secure, encrypted communications between the RFID/NFC module and the card. Also, there are different mechanisms (CRC, parity, bit encoding, etc) to ensure the data integrity.

The RFID/NFC module is also compatible with NFC, in particular with the standards ISO/IEC 18098 and NFCIP-1. The NFC technology allows two-way communications between end-points that normally are smartphones ar other advanced electronic devices. An NFC communication can be set in a very quick way, which is a benefit with regard to WiFi or Bluetooth.

The antenna is 57x57x10 mm and can be plugged in 2 directions, but for the best range we suggest to place it like in the photos, standing out of the surface of Waspmote. Obviously, for projects with strict form factor specifications, the antenna can be placed over Waspmote. The antenna has a gain of around 33 dB and is configurable by the user.

# 3. Dual Radio with the Expansion Board

Before starting to use a module, it needs to be initialized. During this process, configuration parameters are sent to the module. USB and SD card are also initialized.

## 3.1. Expansion Radio Board

The RFID/NFC module can use the Expansion Radio Board. The new Expansion Board allows connecting two radios at the same time in the Waspmote sensor platform. This means a lot of different combinations are now possible using any of the 10 radios available for Waspmote: 802.15.4, ZigBee, 868 MHz, 900 MHz, Bluetooth, RFID, RFID/NFC, Wifi, GPRS and 3G/GPRS.

Some of the possible combinations are:

- ZigBee - Bluetooth
- ZigBee – RFID/NFC
- RFID - Wifi
- ZigBee – 3G/GPRS
- Wifi – RFID/NFC
- Wifi - 3G/GPRS
- Wifi – Bluetooth
- etc.

***Remark:*** *the GPRS module and 3G/GPRS module do not need the Expansion Board to be connected to Waspmote. It can be plugged directly in the GPRS socket.*

Next image shows the sockets available along with the UART assigned. On one hand, SOCKET0 allows to plug any kind of radio module through the UART0. On the other hand, SOCKET1 permits to connect a radio module through the UART1.

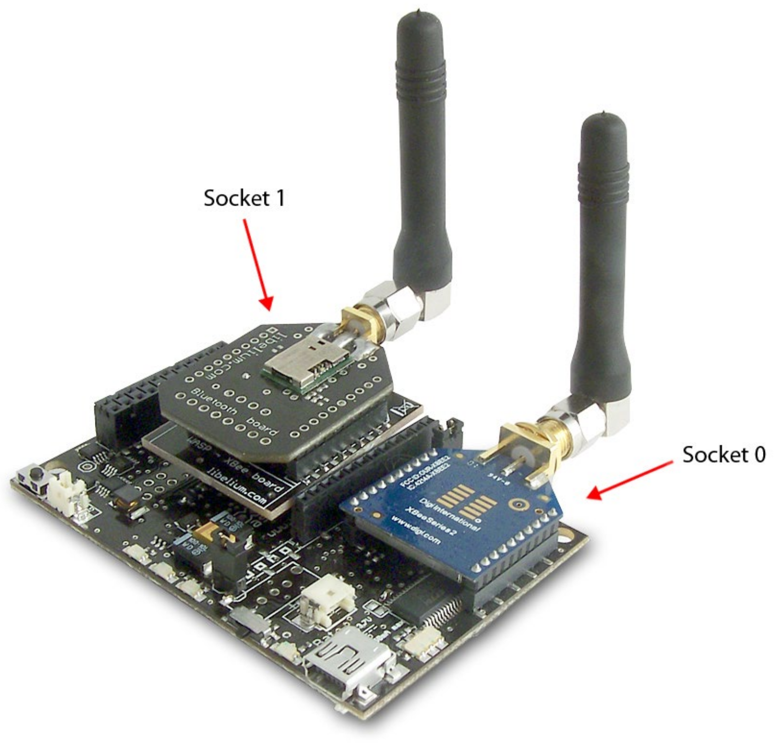


*Figure : Waspmote with XBee radio on socket 0 and Bluetooth module on socket 1*

This API provides a function in order to initialize the RFID/NFC module module called `RFID13.ON(socket)`. This function supports a new parameter which permits to select the socket. It is possible to choose between socket0 or socket1

An example of use the initialization function is the following:

- Selecting socket0: `RFID13.ON(SOCKET0);`
- Selecting socket1: `RFID13.ON(SOCKET1);`

The rest of functions are used the same way as they are used with older API versions. In order to understand them we recommend to read this guide.

WARNING:

- Avoid to use DIGITAL7 pin when working with Expansion Board. This pin is used for setting the XBee into sleep.
- Avoid to use DIGITAL6 pin when working with Expansion Board. This pin is used as power supply for the Expansion Board.

Incompatibility with Sensor Boards:

- Gases Board: Incompatible with SOCKET3B and $NO_2$ sensor.
- Agriculture Board: Incompatible with Sensirion and the atmospheric pressure sensor.
- Smart Metering Board: Incompatible with SOCKET2, SOCKET3, SOCKET4 and SOCKET5.
- Smart Cities Board: Incompatible with microphone and the CLK of the interruption shift register.
- Events Board: Incompatible with interruption shift register.

## 3.2. Setting ON

With the function ON() module is powered and the UART is opened to communicate with the module. It also enters in command mode and sets some default configurations.

```
// switches ON the RFID/NFC module,using expansion board

            RFID13.ON(SOCKET1);
```

## 3.3. Setting OFF

The RFID/NFC API function `OFF()` exits from the radio RFID/NFC command-mode securely, closes the UART and switches the module off.

```
// closes the UART and powers off the module

            RFID13.OFF();
```

# 4. RFID Tags

Libelium offers Mifare® Classic 1k cards tags and stickers along with the Waspmote RFID/NFC module. More than 3.5 billions of this type of cards have been sold around the world up now.

An RFID card has a thin form factor (81x54x1 mm), like a driving license or identification card, and can be kept in the user's wallet. However, an RFID tag is smaller and thicker (around 20x20x5 mm) but can be kept in the user's keyring. Last, an RFID sticker has an intermediate size (50x50) and it is really thin and flexible. Of course, a sticker can be stuck in any object so it focuses on the Internet of Things (IoT) and logistics applications.

It is important to note that a card always has a longer range than a tag or sticker because the antenna of a card has more surface.

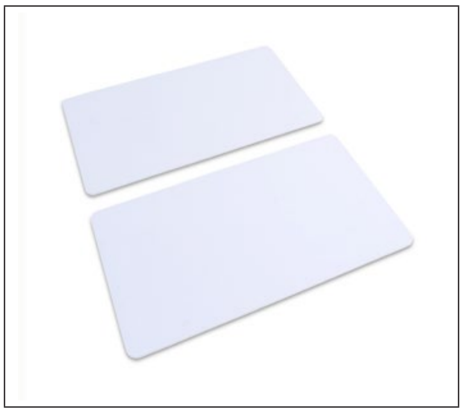


*Figure : RFID cards*

*Figure : RFID keyrings*

*Figure : RFID sticker*

A Mifare® Classic 1k card has 1024 bytes of internal storage capacity, divided into 16 sectors. Each sector is composed of 4 blocks, and each block is composed of 16 bytes. That is to say, each card has 64 blocks, from number 0 to 63. Let's describe the structure of a card:

1. In each sector, the last block (blocks number 3, 7, 11..) is called the sector trailer. This block stores the 2 keys or passwords and controls the access to the rest of the blocks in the sector (e.g. block number 7 controls block 4, 5 and 6). The first 6 bytes (0..5) are the key A, and the last 5 bytes (10..15) are the key B. As we will see, before reading or writing in any block, we must first authenticate us to that sector by providing one of the two keys (normaly the A key). The bytes number 6, 7 and 8 store the control bytes, which control the way the the blocks may be accessed (read/write). Byte number 9 is not defined in all sector trailers.

2. The very first block (number 0) is called the manufacturer block and has special characteristics. The first 4 bytes (0..3) are the unique identification (UID). The UID can be seen as the serial number of the card and identify the card in a univocal way (read more in the "Security with RFID/NFC at 13.56 MHz" chapter). The next byte (number 4) is the CRC check byte so it can be used to check the correct read; it is calculated with the XOR of the 4-byte UID. The rest of the bytes in the block number 0 (bytes 5..15) are the manufacturer data. This block 0 has read-only access for security reasons. We need to authenticate before reading the block number 0 but the UID can be obtained anyway, without the key.

3. The rest of the blocks (1 and 2; 4, 5 and 6; 8, 9 and 10; etc) are known as data blocks and they are available to read and write operations, after the needed authentication process.

To sum up, the Mifare® 1k cards have a net storage capacity of:

(16 sectors/card x 3 data blocks/sector X 16 bytes/block) – 16 bytes (first block) = 752 bytes/card

The cards have been tested to conserve data for 10 years and to have a write endurance of 100,000 cycles.

# 5. Usage

The maximum operational range is about 7 cm for a card and 5 cm for a tag or a sticker. A target beyond this distance will not be detected, or the communications with it will be full of errors. Some RF-related registers in the module can be trimmed to try to improve this range, but it is not really recommended.

Besides, the minimum range is of a few mm. A card which is closer than this to the RFID/NFC module's antenna could not be detected.

Optimal results are obtained when the card is placed parallel to the surface of the antenna, still and at a distance of 1 cm.

It is advised to execute one `init()` function each time that we need to do some RFID operation, just like shown in the examples. That will reset the module to the correct configuration and avoid problems.

As shown in the examples, it is good to use "if's" to control which functions are executed. For example, if the `authenticate()` function was not successful, it does not make sense to execute one `read()` after that. Also, if the `init()` function was not successful, or if it did not deliver an UID and a valid ATQ-A, it does not make sense to execute an `authenticate()` afterwards. All functions should deliver '0' when executed correctly.

When any of the functions does not deliver a successful execution, it is advised to end the loop and start the RFID/NFC process from the beginning (from the `init()` function).

A typical read or write process can take between 80 ms. Make sure the card remains within the field until the whole process is completed.

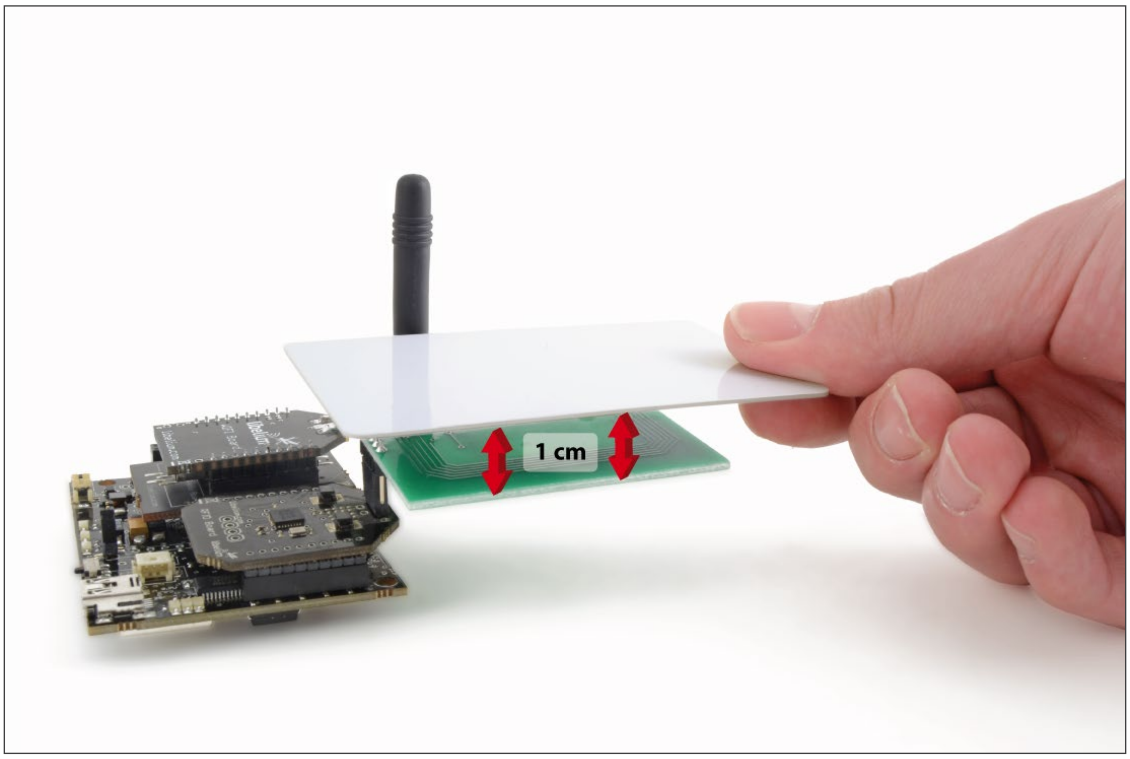


*Figure : Typical RFID/NFC operation*

Some kind of indicator is strongly advised to let the user know that data exchange was successful. Waspmote can control a LED (built-in or external) or a buzzer for this goal.

After a successful operation, it is advised to execute a short delay (e.g. 2 seconds) to let the user take his card away. This way we avoid to execute the same operation more than one time.

The RFID/NFC antenna should be located in an accessible and comfortable place for the user.

The RFID/NFC antenna, the RFID/NFC module, the battery or Waspmote are not waterproof, so an enclosure is mandatory when the system is going to work outdoors. The antenna should be fixed really close to the wall of the enclosure. Metallic enclosures are not advised.

In general, avoid placing the RFID/NFC antenna near metallic objects, since the communication range could decrese.

It can be a good idea to place the antenna near the top of the enclosure, in parallel with the ground. This way the user can just let the card lie still on top of the case for a while. Wherever the antenna is, do not forget to indicate in your enclosure where it is.

Use the Waspmote's Expansion Board to enable another communication module to transmit certain (or all) RFID/NFC operations to another device or to a data center.

Use a micro SD card to store certain (or all) RFID/NFC operations. This can be a security back up.

Add one Meshlium device to your ecosystem for advanced features such as Linux OS tools, MySQL data base storing system, WiFi and Ethernet connectivity, remote web server or higher computing speed.

# 6. Security with RFID/NFC at 13.56 MHz

The standard RFID ISO/IEC 14443-A cards have an UID with a length of 4 bytes, so there are 4,200 millions of different UIDs.

Besides, the A or B key has a length of 6 bytes. That means there are $2.8 \cdot 10^{14}$ different possible passwords.

These numbers, along with the three pass authentication and the data integrity mechanisms, demonstrate RFID/NFC is a pretty secure technology.

**FAQs:**

**Q: Can I change the UID in a given card?**

A: No, it is impossible. The block number 0, where the UID is stored, has read-only access. There are security reasons to do so: if the block number 0 could be written, it would be possible to duplicate or forge cards.

**Q: I heard that lately there are 7-byte UID cards, so there could exist a 7-byte UID card with the same beggining UID than the standard 4-byte UID card. So the UID is not so "unique". Is that true?**

**A:** Yes, the manufacturers started producing cards with a UID of 7 bytes and there could be a 7-byte UID with its first 4 bytes equal than a the ones in a standard 4-byte card.

**Q: So can I consider a 4-byte UID card as unique or not?**

**A:** No, but there is just one possibility among thousands of millions that you find another card like yours.

**Q: Can I order or select a specific UID for my card?**

**A:** No, the cards' UIDs are set in a random way.

**Q: I do not know/remember the key for a certain block, can I read or write in that block?**

**A:** No, it is not possible to access a block unless we have authenticated us in it. All cards are provided with both A and B keys by default (0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF). If the user changes them, he must remember the change. The only thing we can do without the keys of a card is to read its UID and ATQ.

**Q: Are the RFID ISO/IEC 14443-A/Mifare® standards a 100% secure system?**

**A:** No. Any security system has bugs that can be hacked. Besides, there are "security enhancers" integrated chips for RFID/NFC that Libelium does not implement.

**Q: Does Libelium recommend its RFID/NFC module for electronic money exchange?**

**A:** No. The RFID/NFC module by Libelium is not intended for payment applications but for control of usage.

**Q: Should I change the key to the cards?**

**A:** Yes, you should if it is possible that someone is interested in reading or changing the stored information. Setting a new key is a quick process and will ensure only the authorized agents can read or write the data. Avoid sharing or losing this information.

We advise to set a random key. As a tip, it would be an even more secure system if each card has its own key (maybe depending on its own UID).

Since each sector can have different access keys, note it is possible to store public data in certain sectors and private, protected data in other sectors. One example:

- sector 2: personal, general info (key only known by all the institutions of the city council)
- sector 3: city's transport info (key only known by the bus company)
- sector 4: city's library info (key only known by the library)
- sector 5: city's gym (key only known by the gym)
- ...

On the other hand, if your system is just going to read UIDs, there is no point in changing keys.

# 7. RFID: 13.56 MHz or 125 KHz?

Libelium has developed 2 different RFID modules: 13.56 MHz and 125 KHz. They are not compatible because they comply with different standards. As a result their working frequencies are very different, and the communication protocols are not the same. So:

- 13.56 MHz RFID cards cannot work with the 125 KHz RFID module
- 125 KHz RFID cards cannot work with the 13.56 MHz RFID/NFC module

Everything said here about 13.56 MHz RFID technology is also applicable to the NFC technology.

The decision to adopt one system or the other depends on several points. First of all, if you are purchasing Libelium's RFID solution to fit an existing system you will want to to keep the compatibility.

Both frequency bands are suitable to be used in any country. 125 KHz (LF) is unregulated and 13.56 MHz (HF) is an ISM band.

In general, the industry has used the 125 KHz RFID technology in projects that identify and track objects (like in a supply chain), while the 13.56 MHz RFID/NFC technology is used in applications that identify objects or people and store information.

A 125 KHz RFID card, T5557 type, has a memory of just 20 bytes, while a 13.56 MHz RFID Mifare® card, 1k type, has up to 752 available bytes. This seems a great difference, but bear in mind 20 bytes are enough to store information such a long ID code.

Regarding maximum ranges, a 125 KHz RFID card has a similar read range (about 6 or 7 cm) than a 13.56 MHz RFID card (3 cm). However, when the operation includes writing, the range of a 125 KHz RFID card is smaller (1 cm).

The 13.56 MHz RFID/NFC module and its cards implement more secure communications, which includes advanced methods for safer authentication, counterfeiting prevention, encryption and data integrity. Besides, a 13.56 MHz RFID/NFC key is longer and safer than a 125 KHz RFID password. Therefore, 13.56 MHz RFID/NFC is the best option if security is important.

As said before, it is not advised to place the RFID system near metallic objects; however the 125 KHz RFID technology seems to be the most adaptive to a metal environment. However, in industrial environments, the 125 KHz RFID communications could be affected by the noise generated by motors, while the 13.56 MHz technology would not.

The 13.56 MHz RFID/NFC module has much faster data rates in read or write operations.

# 8. RFID 13.56 MHz and NFC

The aim of this chapter is to explain the differences between RFID @ 13.56 MHz and NFC.

As said before, the NFC technology is defined by protocols like ISO/IEC 18098 and NFCIP-1. NFC is an extension of RFID @ 13.56 MHz (ISO/IEC 14443-A/Mifare® protocols) and bases on those protocols to build a technology which focuses on allowing an instant and easy dialog between advanced electronic devices like smartphones. While RFID is meant to interchange very little amounts of data (few bytes), NFC is meant to interchange bigger amounts of data.

Libelium's RFID/NFC module supports both operations in a native way: RFID @ 13.56 MHz (ISO/IEC 14443-A/Mifare® protocols) and NFC (NFCIP-1), but Libelium has not implemented software functions to handle the NFC operations. The reason to that is that the most important applications of the RFID/NFC module for Wireless Sensor Networks are identifying and reading/writing cards and tags, and not interchanging big amounts of data in very short range communications. Besides, only the most advanced smartphones support NFC, so there are not many potential users yet with NFC-capable phones.

To sum up, the applications which benefit the most from the RFID/NFC module are those related to access control, logistics, assets tracking, supply chain, etc.

# 9. Libelium's API

It is mandatory to include the RFID library when using this module. The following line must be introduced at the beginning of the code:

```
#include <WaspRFID13.h>
```

Waspmote's API RFID/NFC files:

- WaspRFID13.cpp
- WaspRFID13.h

**APIs funcions**

- **Private functions:**
  The following functions are executed inside the API functions. In normal conditions, the user must NOT manage or use them.

| Function | Brief description |
|---|---|
| `bool  configureSAM(void);` | set internal parameters of the PN532 |
| `sendTX(uint8_t *dataTX, uint8_t length, uint8_t outLength);` | Send data stored in dataTX |
| `getACK(void);` | Wait for ACK response |
| `waitResponse(void);*` | Wait the response of the module |
| `getData(uint8_t outLength);` | Get data from the module |
| `checkSum(uint8_t *dataTX);` | Calculates the checksum |
| `uint8_t lengthCheckSum(uint8_t *dataTX);` | Calculates the length checksum |

** To avoid the hang of the module this function will wait for 5 seconds aproximately. If no tag is detected the progam will continue after this time. The waiting time can be configured in the library.*

## 9.1. Library constructor

It is the class constructor. It is only executed inside the API’s function `init()`.

- **Public functions:**
  The following functions are public and therefore they can be executed in the `loop()` or `setup()` functions. They provide the tools to perform all the features of the RFID/NFC module.

| Function | Brief description |
|---|---|
| `OFF()` | switches the module off |
| `ON(socket)` | switches the module on, in one of the 2 sockets |
| `init(*UID, *ATQ)` | inits the module, configures it and searches for new cards |
| `authenticate(*UID, ad, *key)` | authenticates one card’s sector |
| `bool getFirmware(void)` | The PN532 sends back the version of the embedded firmware. |
| `read(ad ,*data)` | reads one card’s block |
| `readWithAuth(*UID, *key, *data, ad)` | authenticates one card's sector and reads one block |
| `write(ad, *data)` | writes one card’s block |
| `writeWithAuth(*UID, *key, *data, ad)` | authenticates one card's sector and writes one block |
| `writeAndCheck(*data, ad)` | writes one card's block and checks that |
| `writeAndCheckWithAuth(*UID, *key, *data, ad)` | authenticates one card's sector, writes one block and checks that |
| `uint8_t powerDown(void)` | put the module into Power Down mode |
| `wakeUp(void)` | wake up from power dowm mode. |
| `print( *data, length)` | print data stored in vectors |
| `setKeys(*UID, *keyOld, *keyA, *keyB, *cfg, *data, ad)` | changes both keys and access conditions to one card's sector |
| `equalUIDs(*UID1, *UID2)` | compares 2 UIDs |
| `searchUID(*vCards, *UID, nCards)` | searches one UID inside one group of UIDs |
| `string2vector(*inp, *outp)` | converts from a string to a uint8_t vector |
| `vector2int(*inp)` | converts from a uint8_t vector to a integer |

## 9.2. Switching the module on

It takes the RFID/NFC module out of the power-down mode by starting the wake up procedure.

If you connected the RFID/NFC module into the usual XBee socket, please use SOCKET0; if you connected the RFID/NFC module into the Expansion Board, you must use SOCKET1.

Example of use

```
{
RFID13.ON(SOCKET0); // switches the module on with the socket 0 (the normal socket, without
                   // using the Expansion Board)
}
```

**http://www.libelium.com/development/waspmote/examples/RFID1356-01-basic-example**

## 9.3. Switching the module off

It takes the RFID/NFC module to the software power-down mode.

Example of use

```
{
RFID13.OFF(); // switches the module off
}
```

**http://www.libelium.com/development/waspmote/examples/RFID1356-01-basic-example**

## 9.4. Initiating the module

It initializes the necessary variables, requests if there is an ISO/IEC 14443-A card in the field, does the anticollision loop and selects the card to operate.

This function is enough for those applications which just require to get the UID of cards, since most of the ISO/IEC 14443-A processes are executed inside.

It is highly advised to execute this function in the beginning of each loop, just as shown in any of the examples. Even if the module is already initialized, it is good to start from the same point every time.

This function initializes many internal registers and variables. All the settings are optimized for the best operation. Changing those settings may result in a defective operation.

Example of use

```
{
uint8_t state;  // stores the status of the executed command
uint8_t ATQ[2]; // stores the ATQ-A (answer to request, type 14443- A). Generally stored in
                // a bigger buffer (16B)
uint8_t UID[4]; // stores the UID (unique identification) of a card
...
state = RFID13.init(UID, ATQ); // The goal of this command is to detect as many targets
                               // (maximum MaxTg) as possible in passive mode.
}
```

**http://www.libelium.com/development/waspmote/examples/RFID1356-01-basic-example**

## 9.5. Authenticating a sector

It authenticates a sector of the card thanks to the key or password. It is important to remark that before doing any read or write in a block, it is mandatory to authenticate us in it.

The only exception is the UID, which can be read without authentication; it is returned by the `init()` function.

One authentication is enough to validate all the 4 blocks in a sector.

Example of use

```
{
uint8_t state;  // stores the status of the executed command
uint8_t ATQ[2]; // stores the ATQ-A
uint8_t UID[4]; // stores the UID (unique identification) of a card
uint8_t key[6]; // stores the key or password
...
memset(key, 0xFF, 6); // the key by default, edit if needed
state = RFID13.init(UID, ATQ); // inits systems and look for cards
state = RFID13.authenticate(UID, 2, key); // authenticates block number 2 (all sector 0
                                   // actually)of the selected card. State of the executed
                                   // function is stored.
}
```

**http://www.libelium.com/development/waspmote/examples/RFID1356-01-basic-example**

## 9.6. Reading a block

It reads the 16 bytes stored in a block. This block must have been authenticated before reading.

Example of use

```
{
uint8_t state;   // stores the status of the executed command
uint8_t aux[16]; // auxiliar buffer
uint8_t UID[4];  // stores the UID (unique identification) of a card
uint8_t key[6];  // stores the key or password
...
memset(key, 0xFF, 6); // the key by default, edit if needed
state = RFID13.init(UID, aux); // inits systems and look for cards
state = RFID13.authenticate(UID, 2, key); // authenticates block 2
state = RFID13.read(2, aux);// reads the block 2. Data is stored in aux.
}
```

**http://www.libelium.com/development/waspmote/examples/RFID1356-02-read-all-blocks**

## 9.7. Reading a block with authentication

It authenticates a block of the card first, and then reads the 16 bytes stored in a block. This function is useful to do these 2 steps in just one.

Example of use

```
{
uint8_t state;   // stores the status of the executed command
uint8_t aux[16]; // auxiliar buffer
uint8_t UID[4];  // stores the UID (unique identification) of a card
uint8_t key[6];  // stores the key or password
...
memset(key, 0xFF, 6); // the key by default, edit if needed
state = RFID13.init(UID, aux); // inits systems and look for cards
state = RFID13.readWithAuth(UID, key, aux, 2); // authenticates block 2, then reads it.
                                                // Data is stored in aux.
}
```

## 9.8. Writing in a block

It writes 16 bytes in a block. This block must have been authenticated before writing.

Example of use

```
{
uint8_t state;   // stores the status of the executed command
uint8_t aux[16]; // auxiliar buffer
uint8_t UID[4];  // stores the UID (unique identification) of a card
uint8_t key[6];  // stores the key or password
...
memset(key, 0xFF, 6); // the key by default, edit if needed
state = RFID13.init(UID, aux); // inits systems and look for cards
state = RFID13.authenticate(UID, 2, key); // authenticates block 2
memset(aux, 0x00, 16); // stores 0's in aux, 16 bytes
state = RFID13.write(2, aux);  // writes the content of aux in block number 2
}
```

**http://www.libelium.com/development/waspmote/examples/RFID1356-03-bus-ticketing**

## 9.9. Writing in a block with authentication

It authenticates a block of the card first, and then writes 16 bytes in the block. This function is useful to do these 2 steps in just one.

Example of use

```
{
uint8_t state;   // stores the status of the executed command
uint8_t aux[16]; // auxiliar buffer
uint8_t UID[4];  // stores the UID (unique identification) of a card
uint8_t key[6];  // stores the key or password
...
memset(key, 0xFF, 6); // the key by default, edit if needed
state = RFID13.init(UID, aux); // inits systems and look for cards
memset(aux, 0x00, 16); // stores 0's in aux, 16 bytes
state = RFID13.writeWithAuth(UID, key, aux, 2); // authenticates block 2, then writes the
                                                // content of aux in it
}
```

## 9.10. Writing in a block and checking it

It writes 16 bytes in a block, and then checks the correct data was written. This block must be authenticated before writing. This function is useful to do these 2 steps in just one.

Example of use

```
{
uint8_t state;   // stores the status of the executed command
uint8_t aux[16]; // auxiliar buffer
uint8_t UID[4];  // stores the UID (unique identification) of a card
uint8_t key[6];  // stores the key or password
...
memset(key, 0xFF, 6); // the key by default, edit if needed
state = RFID13.init(UID, aux); // inits systems and look for cards
state = RFID13.authenticate(UID, 2, key); // authenticates the block
memset(aux, 0x00, 16); // stores 0's in aux, 16 bytes
state = RFID13.writeAndCheck(aux, 2); // writes the content of aux in block 2, then checks
                                      // its content is OK
}
```

## 9.11. Writing in a block with authentication and checking it

It authenticates a block of the card first, then writes 16 bytes in the block, and then checks the correct data was written. It is useful to do these 3 steps in just one.

Example of use

```
{
uint8_t state;   // stores the status of the executed command
uint8_t aux[16]; // auxiliar buffer
uint8_t UID[4];  // stores the UID (unique identification) of a card
uint8_t key[6];  // stores the key or password
...
memset(key, 0xFF, 6); // the key by default, edit if needed
state = RFID13.init(UID, aux); // inits systems and look for cards
state = RFID13.authenticate(UID, 2, key); // authenticates the block
memset(aux, 0x00, 16); // stores 0's in aux, 16 bytes
writeAndCheckWithAuth(UID, key, aux, 2); // authenticates block 2, then writes the content of
                                         // aux in it and checks its content is OK
}
```

## 9.12. Setting keys in a sector

It authenticates a block of the card first, then sets the new keys A and B, and the access bits for a sector. After that, it authenticates again and reads the trailer block. This function is useful to do these 4 steps in just one.

This function should be executed with care because if any of the internal functions is not successfully executed, then the sector could be damaged or left unaccessible. That is why it is highly recommended to read the output status in order to check if all the process was completed. Please place the card still and close distance from the RFID/NFC module's antenna when setting new keys; if the whole process is not properly completed, the access to that sector might be lost forever.

There are many options for the configuration of the access bits. We suggest to use the same options that the card has by default: {0xFF, 0x07, 0x80, 0x69}. With this configuration:

- data blocks must be authenticated with the key A, and then can be read/written
- key A can only be changed (written) after an authentication with key A
- access bits can be read or written after an authentication with key A

- in the sector trailer, the key B has no use and can be read/written after an authentication with key A. It can be used as additional storage space, for instance.

With any configuration, the key A can never be read (the module returns 0's instead of the key) for security reasons. As we can see, the key A of each RFID/NFC card is the master key and we must avoid losing or sharing this information.

Any new card is delivered with the same A and B keys by default: 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF. That is to say, 48 logic 1's.

As shown, the key length is 6 bytes so t here are more than $2.8 \cdot 10^{14}$ different possible passwords.

Example of use

```
{
uint8_t state;   // stores the status of the executed command
uint8_t aux[16]; // auxiliar buffer
uint8_t UID[4];  // stores the UID (unique identification) of a card
uint8_t keyA_old[6];  // stores the old key A
uint8_t keyA_new[6];  // stores the new key A
uint8_t keyB_new[6];  // stores the new key B
uint8_t config_new[4] = {0xFF, 0x07, 0x80, 0x00}; // same cfg
...
memset(keyA_old, 0xFF, 6); // the key by default, edit if needed
memset(keyA_new, 0x88, 6); // the new key A, edit if needed
memset(keyB_new, 0x99, 6); // the new key B, edit if needed
state = RFID13.init(UID, aux); // inits systems and look for cards
state = RFID13.setKeys(UID, keyA_old, keyA_new, keyB_new, config_new, aux, 2); //
        // authenticates block 2, then writes the new trailer block, with new key A
        // and B. The access bits are the same.
}
```

**http://www.libelium.com/development/waspmote/examples/RFID1356-05-password-simple**

## 9.13. Powering down

This function puts the module into Power Down mode in order to save power consumption.

Example of use

```
{
        RFID.powerDown(); // After this function you must wake up the module.
}
```

## 9.14. Waking up

Wake up from power dowm mode.

Example of use

```
{
        RFID.wakeUp(); //Wake Up from Power Down mode.
}
```

## 9.15. Printing data

This function is very useful  for viewing data content in vectors when you are debugging the code.

Example of use

```
 {
       RFID.print(readData,16); //Print data read and stored in readData vector in the
       serial monitor
 }
```

**http://www.libelium.com/development/waspmote/examples/RFID1356-01-basic-example**

## 9.16. Comparing UIDs

It compares 2 UIDs to check if they are equal. It is useful to check if the UID we have just detected is the same than a predetermined UID. One logic application could be access control for a single person.

Example of use

```
{
uint8_t state;  // stores the status of the executed command
uint8_t ATQ[2]; // stores the ATQ-A
uint8_t UID[4]; // stores the UID (unique identification) of a card
uint8_t card[] = {0xXY, 0xXY, 0xXY, 0xXY}; // the UID of the card that we want to find
boolean tr = false;
...
state = RFID13.init(UID, ATQ); // inits systems and look for cards
tr = RFID13.equalUIDs(card, UID); // if the read UID is the same than the original card, tr
                                  // will be true
}
```

**http://www.libelium.com/development/waspmote/examples/RFID1356-06-single-cards-counter**

## 9.17. Searching a UID among a group of UIDs

It tests if a certain UID is contained in a set or vector of UIDs. It is useful to check if the UID we have just detected is inside a predetermined set or group of UIDs. One logic application could be access control for a group of people.

Example of use

```
{
#define nCards 3 // edit: number of possible cards
uint8_t state;   // stores the status of the executed command
uint8_t ATQ[2];  // stores the ATQ-A
uint8_t UID[4];  // stores the UID (unique identification) of a card
// edit: vector with the UIDs of all the cards
uint8_t vCards [nCards*4] = {0xXY, 0xXY, 0xXY, 0xXY,
                             0xXY, 0xXY, 0xXY, 0xXY,
                             0xXY, 0xXY, 0xXY, 0xXY};
int card = -1;  // stores the index of the present card
...
state = RFID13.init(UID, ATQ); // inits systems and look for cards
card = RFID13.searchUID(vCards, UID, nCards); // looks for the read card inside the data
                                   // base. If so, it returns its index in the vector.
                                   // If not, -1.
}
```

**http://www.libelium.com/development/waspmote/examples/RFID1356-06-single-cards-counter**

## 9.18. Converting from a string to a uint8_t pointer

It converts from a string to a pointer to uint8_t's. When we want to write a word or phrase in an RFID card, this function is useful to convert from ASCII code to the format the RFID/NFC module can understand.

Example of use

```
{
uint8_t aux[16]; // auxiliar uint8_t pointer
char text [16];  // auxiliar string
int number = 3;  // stores numbers
...
sprintf(text ,"Its a test %d ", number*4); // add a number to a string (dynamic !! )
RFID13.string2vector(text, aux); // converts from string to a uint8_t pointer
}
```

**http://www.libelium.com/development/waspmote/examples/RFID1356-03-bus-ticketing**

## 9.19. Converting from a uint8_t pointer to an integer

It converts from a pointer to uint8_t's to an integer. When we want to read a number frim an RFID card, this function is useful to convert from ASCII conde to the format we can understand, an integer.

Example of use

```
{
uint8_t aux[16] = {0x51, 0x52, 0x53, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}; // auxiliar uint8_t pointer
int number = 0;  // stores numbers
...
number = vector2int(aux); // converts from a uint8_t pointer to an integer
}
```

**http://www.libelium.com/development/waspmote/examples/RFID1356-03-bus-ticketing**

# 10. Time of execution of the functions

The following is a list of the time it takes to execute the main functions:

- init(): 25-30 ms
- authenticate(): 20 ms
- read(): 25 ms
- write(): 25 ms

So a full read or write in a block process (init+authenticate+read or init+authenticate+write) could take around 75 ms. However, a standard UID read (init) can take just 25 ms.

There is a mechanism inside the `init()` function so that if no card is detected, then not all the function is executed. This feature allows to search for a new card faster.

In the read_all_blocks example, all the 64 blocks in a Mifare® Classic 1k card are read in about 2.6 s. This indicates that the maximum read rate is 393 bytes per second.

# 11. Code examples and extended information

In the Waspmote Development section you can find complete examples:

**http://www.libelium.com/development/waspmote/examples**

```
/*
 *  ------ [RFID1356_03] RFID Bus Example --------
 *
 *  Explanation: This sketch shows how Libelium’s RFID/NFC 13.56 can be
 *  used for ticketing solutions, in this case we use the RFID/NFC to
 *  simulate a RFID access control inside a bus.
 *  First, we assignate a quantity of money to the card (with a secret
 *  password of course). After that, each trip will substract the fare.
 *  This can be also used for a gym or a parking, for example.
 *
 *  Copyright (C) 2012 Libelium Comunicaciones Distribuidas S.L.
 *  http://www.libelium.com
 *
 *  This program is free software: you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation, either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program. If not, see <http://www.gnu.org/licenses/>.
 *
 *  Version:                0.1
 *  Design:                 David Gascon
 *  Implementation:         Ahmad Saad, Javier Solobera
 */


#include <WaspRFID13.h>

// stores the status of the executed command:
uint8_t state;
// auxiliar buffer:
uint8_t aux[16];
// stores the UID (unique identifier) of a card:
uint8_t UID[4];
// stores the key or password:
uint8_t keyAccess[] = {
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
// EDIT: initial credit to put in the card (in cents!)
// it is also possible to put “credits”
int initialCredit = 800; // this is 8 euros (8*100 = 800)
//int initialCredit = 32000; // this is 320 euros (8*100 = 800)

// EDIT: the price per trip (in cents !)
// it is also possible to put 1 (if they are credits)
int tripFare = 105; // this is 1.05 euros (no problems with floats now)

boolean firstDone = false; // signal if the first credit write was done
int credit = 0; // stores the money inside the card
char text[16]; // for changing from one format to the other
boolean completed = false; // signals if all the process was successfuly completed
```

```
void setup()
{
  USB.ON();
  USB.println("RFID/NFC @ 13.56 MHz module started");
  // switchs ON the module, type B, and asigns the socket
  RFID13.ON(SOCKET0);
  delay(1000);
}



void loop()
{
  USB.print("\r\n++++++++++++++++++++++++++++++++++++");
  // **** init the RFID/NFC reader
  state = RFID13.init(UID, aux);
  if (aux[0] == aux[1])  // if so, there is no card on the EM field
  {
    USB.print("\r\nRequest error - no card found");
  }
  else // a card was found
  {
    // **** authenticate the key A of sector 1
    state = RFID13.authenticate(UID, 1, keyAccess);
    if (firstDone == false)  // this part is for adding credit
    {
      // **** if passed authentication in block 1, we are able to set the inital credit
      sprintf(text, "%d", initialCredit);
      RFID13.string2vector(text, aux);
      // **** write aux in block number 1, and check afterwards
      state = RFID13.writeAndCheck(aux, 1);
      if (state == 0)
      {
        for (int i=0; i<sizeof(text); i++) // clear this variable
        {
          text[i] = '\0';
        }
        firstDone = true;
        // **** check the 16 bytes in block 1
        state = RFID13.read(1, aux);
        if (state == 0)  // if the read command was successful, we show the data (16 bytes)
        {
          USB.print("\r\n  Initial credit set:  ");
          for (int i=0; i<16; i++)
          {
            if (aux[i] == '\0') // to avoid showing voids
              break;
            USB.print(aux[i], BYTE); // print the 16 bits of block number 1, now in ASCII 
code
          }
        }
      }
    } // end of "adding credit part"
    // **** if passed authentication in block 1, we will be able to read the data in this 
block (the credit)
    state = RFID13.read(1, aux);
    if (state == 0)  // if the read command was successful, we show the data (16 bytes)
    {
      USB.print("\r\n  Credit before the trip: ");
      for (int i=0; i<16; i++)
      {
        if (aux[i] == '\0') // to avoid showing voids
          break;
        USB.print(aux[i], BYTE); // print the 16 bytes of the block 1 (the first ones are 
the credit, the rest are NULLs)
      }
      credit = RFID13.vector2int(aux); // convert the card data to int
      if (credit >= tripFare)
      {
        sprintf(text, "%d", credit - tripFare);
        RFID13.string2vector(text, aux);
        for (int i=0; i<sizeof(text); i++)  // clear this variable
        {
          text[i] = '\0';
        }
        // **** write aux in block number 1, and check afterwards
        state = RFID13.write(1, aux);
        if (state == 0)
        {
          //miSerial.print(" -->  correct write checked");
          completed = true;
          credit = credit - tripFare; // only substract if success happened
          // **** check the 16 bytes in block 1
          state = RFID13.read(1, aux);
          if (state == 0)  // if the read command was successful, we show the data (16 
bytes)
          {
            USB.print("\r\n  Credit after the trip:  ");
            for (int i=0; i<16; i++)
            {
              if (aux[i] == '\0')  // to avoid showing voids
                break;
              USB.print(aux[i], BYTE); // print the 16 bits of block number 1, now in ASCII 
code
            }
          }
        }
      }
      else
      {
        USB.print("\r\n  ** Not enough credit");
      }
    }
  }
  if (completed == true)  // a complete operation was done !!!
  {
    USB.print("\r\n  ACCESS GRANTED !!");
    completed = false;
    for (int i=0; i<2; i++)  // blink the LED fast to show not-OK
    {
      Utils.blinkLEDs(50);
    }
  }
  else
  {
    for (int i=0; i<10; i++) // blink the LED fast to show not-OK
    {
      Utils.blinkLEDs(50);
    }
  }
  delay(2000); // wait some time in each loop
}
```

# 12. API changelog

| Function / File | Changelog | Version |
|---|---|---|
| `ON()` | internal change in function | v006 → v007 |
| `getFirmware()` | internal change in function | v006 → v007 |
| `read()` | internal change in function | v006 → v007 |
| `print()` | internal change in function | v006 → v007 |
| `configureSAM()` | internal change in function | v006 → v007 |
| `WaspRFID::ON(uint8_t socket, uint8_t _type)` | The function have now only the parameter for selecting the socket. In older version there were two parameters, `WaspRFID::ON(uint8_t socket, uint8_t_type)`, where _type could be A or B. | v.31 → v1.0 |
| `uint8_t WaspRFID::readData(uint8_t address, uint8_t *readData)` | This function changes his name. The new declaration is "`uint8_t WaspRFID::read(uint8_t address, uint8_t *readData)`" | v.31 → v1.0 |
| `uint8_t WaspRFID::writeData(uint8_t address, uint8_t *readData)` | This function changes his name. The new declaration is "`uint8_t WaspRFID::write(uint8_t address, uint8_t *readData)` " | v.31 → v1.0 |

# 13. Documentation changelog

**From v4.1 to v4.2:**

- API changelog updated to API v007

**From v4.0 to v4.1:**

- Added references to 3G/GPRS Board in section: Expansion Radio Board.