NFC Firmware structure for OPTOTRONIC® ECGs

**===================**

**Confidential**

**For internal use only**

**===================**

Content

[1. Change history 2](#_Toc434387733)

[2. Reference documents 3](#_Toc434387734)

[3. Abbreviations 3](#_Toc434387735)

[4. Introduction 3](#_Toc434387736)

[5. RF interface use cases 4](#_Toc434387737)

[6. Tag memory layout 4](#_Toc434387738)

[7. Data management 5](#_Toc434387739)

[8. I2C vs RF port arbitration 6](#_Toc434387740)

[9. Management registers 7](#_Toc434387741)

[10. Passwords 9](#_Toc434387742)

[11. NFC API 9](#_Toc434387743)

[12. Tag API 11](#_Toc434387744)

[13. I2C bus API 12](#_Toc434387745)

[14. I2C driver API 12](#_Toc434387746)

[15. NFC module setup 13](#_Toc434387747)

[16. NFC module test 15](#_Toc434387748)

[Annex A – Todos 17](#_Toc434387749)

[Annex B – Open issues 17](#_Toc434387750)

[Annex C – Tag memory management 18](#_Toc434387751)

# Change history

|  |  |  |
| --- | --- | --- |
| Version | Date | Changes |
| 0.0 | 10/03/2015 | First draft |
| 1.0 | 10/04/2015 | Review after chip (M34LR16) selection |
| 1.1 | 05/05/2015 | Garching meeting 22/04/2015 update |
| 1.2 | 29/06/2015 | Document review |
|  |  |  |

# Reference documents

* 2D1 3149416 EN 00 NFC antenna design guidelines
* 2D1 3149417 EN 00 NFC FW structure
* 2D1 3151038 EN 00 NFC interface specification
* 2D1 3149415 EN 00 NFC BB reports
* ISO/IEC 14443
* ISO/IEC 15693
* IEC 61347-1
* UL safety standards

# Abbreviations

* cfg: configuration
* DALI: digital addressable lighting interface
* ECG: electronic control gear
* FW: firmware
* IC: integrated circuit
* I2C: Inter Integrated Circuit
* IEC: International Electro technical Commission
* ISO: Internationale Organisation für Normung
* DALI MB: DALI memory bank
* NFC: near field communication
* NVM: non volatile memory
* RF: radio frequency
* RFID: radio frequency identification
* SW: software
* UL: Underwriters Laboratories
* µC: microcontroller

# Introduction

This document aims to describe the firmware module that implements the RF functionality on ECGs equipped with the related hardware. **The analysis assumes that only OSRAM tools are used on the RF side to access the tag**. See Annex A – Open issues.

In this document ”RF” and “NFC” words are used as synonyms even though, considered the M24LRxx IC selected based on ISO15693, “NFC” will be appropriate only after the introduction of ISO15693 in NFC forum (2016).

# RF interface use cases

The following use cases list the interactions between an RF reader and the system (ECG).

1. **Write ECG configuration**

RF reader writes ECG cfg data (rated current, CIN, lamp counters ect.) in NFC tag with ECG µC ON or OFF. The RF reader shall first provide a valid password.

1. **Read-back** **ECG data**

RF reader reads-back ECG data from tag. ECG data is not only ECG cfg data but other ECG constant or quasi-constant data too (CIN, EAN, encrypted password etc.).

In tag memory a copy of ECG data is always present. Reading it back can be done with ECG µC ON or OFF. Password needed.

1. **Read ECG diagnostic**

RF reader can retrieve and monitor ECG diagnostic data from NFC tag. After ECG µC has gone off-line diagnostic data is available. Password needed.

1. **ECG testing**

RF reader can send commands to ECG µC via NFC tag for calibration and specific data read/write operations (functionality similar to OSRAM commands). RF reader shall first provide a valid password.

1. **Change RF password**

RF reader can set, change or delete NFC password. An encrypted version of the password is always readable in the tag so that if the password is lost it can be retrieved (see Dali MB password). NFC and Dali passwords are the same i.e. the defined passwords can be accessed by NFC or Dali interface.

# Tag memory layout

M24LR16 provides 2 KBytes of EEPROM. From RF side the memory is divided in 16 sectors of 128 bytes. Each sector can be protected by one of the three passwords. The number of writing cycles (1 million) does not require memory paging methods. 20 years / 1 million cycles = 10 minutes. Parameters that are supposed to change more often than 10 minutes shall have a minimum saving interval of 10 minutes.

Every parameter shall have a unique and fixed position in the tag EEPROM. Whenever a data struct containing several parameter is used, only the starting address of the struct shall be specified. From RF side this means that when the µC firmware is compiled the address of each parameter of the struct shall be derived and specified somewhere (XML file). Other strategies have been considered: see Annex B – Tag memory management.

All EEPROM memory will be accessible from I2C side with read-write rights.

In order to modify parameters from RF side a valid password shall be provided. This simplifies the memory layout and avoids to handle sectors with different access rights.

# Data management

Parameters are divided by the application in two groups: “quasi constant” parameters that are expected to change a limited number of times during ECG lifetime and “frequently changing” parameters that are expected to change several times (more than 1 million i.e. more than once every 10 minutes) in a ECG lifetime. Different saving strategies are adopted for each data group:

1. Whenever a “quasi constant” parameter is erroneously modified from RF side the µC will restore the expected value as soon as possible. A continuous comparison between RAM and tag data is carried out.
2. “Frequently changing” parameters are saved at regular interval of times (~10 minutes) only if a mismatch between RAM and NFC data is found.
3. Between two “frequently changing” data savings the µC checks every 5 seconds data integrity of “frequently changing” group itself. By calculating the tag data CRC, it controls if anything has accidentally changed: in this case the time out (10 minutes) is reduced to 1s and therefore task at point 2) executed so that all data that does not match with RAM copy is replaced in the tag. Moreover every time a new CRC of “frequently changing” parameters is stored in the tag a copy of it is kept in RAM. Every 5 seconds the CRC RAM copy is compared with the CRC tag copy in order to check if RAM data must be re-written in the tag.
4. At power down all parameters are saved in the tag.

The flow-chart summarizes the timing sequence of operations:

# I2C vs RF port arbitration

RF and I2C can access tag memory at the same time giving place to conflicts. When the communication is present on one port the tag HW disables the other.

I2C port is the master port and manages the mutual access to the tag. RF readers access the tag by first forwarding a tag access request to the ECG and by waiting for acknowledgment.

**I2C access**:

The ECG µC checks if any RF access requests is present every 400 ms. If a RF valid request is present the µC acknowledges it (see following chapter) and waits for the RF operations to complete. After that the ECG controls if new data to upload has been written in the tag. In background a process continuously looks for mismatches between RAM and tag “quasi constant” parameters (every 1s) and between RAM and tag “frequently changing” parameters (every 10 minutes).

**RF access**:

A register (ECG\_ON) tells the RF reader whether the ECG is running. If the ECG is OFF the RF reader can directly access the tag.   
When the ECG is ON the RF reader has first to forward to the ECG access requests by writing a register (RF\_REQUEST). After ECG acknowledges the request the RF reader has some seconds to finish the intended operation without any interruption from the I2C bus.

# Management registers

Besides ECG data (configuration data, diagnostics and testing) part of the EEPROM is reserved for the device management and for RF to I2C data exchange. Here a brief description of the registers with the related memory layout.

**Note**: multiple bytes parameters are written with LSB on lowest tag addresses and MSB on highest tag addresses.

**ECG\_ON**Written with **0xAA** at power-on in order to notify the ECG is ON. When the ECG is switching OFF the register is cleared (**0x00**). At run time at least every second its value is checked and overwritten if different from 0xAA.

**RF\_REQUEST**

Two different requests can be received by ECG from RF reader:

1. **Write request**A RF reader that aims to write data in the tag must first forward a request to the ECG. This is done by writing **0x96** in the register. The ECG then acknowledges the request by writing this register with a random byte**.** After thatthe RF reader has 5 seconds to give back the two’s complement of this random byte. The ECG notifies the RF reader that the write request is valid by writing the register with **0xCC**. The RF reader has now 5 seconds to write data in the tag. When the RF is done it has to write the register back to **0x00** in order to close the update process properly, otherwise data will be overwritten back**.** The ECG knows at this point that new data might be present in the tag.

|  |  |  |
| --- | --- | --- |
| **RF tag write - sequence** | | |
| **step** | **RF\_REQUEST value** | **Who writes ?** |
| 1 | 0x96 | RF reader |
| 2 | Random byte | ECG |
| 3 | Random byte 2’s complement | RF reader |
| 4 | 0xCC | ECG |
| Tag update (by RF reader) | | |
| 5 | 0x00 | RF reader |

**Note:** After step 3, whenever needed, the ECG can switch off the power stage in order to improve the RF communication.

1. **Update request**A RF reader that aims to get an updated copy of ECG data must first trigger a tag data update of “frequently changing” parameters otherwise updated only every some minutes (“quasi constant” data is expected to be constantly up to date). This is done by writing **0xC3** in the register. The ECG then acknowledges the request by writing this register with a random byte**.** After thatthe RF reader has 5 seconds to give back the two’s complement of this random byte. The ECG notifies the RF reader that the request has been accepted and the update process started by writing the register with **0xCC**. When the update process is done the ECG writes the register back to **0x00**.The RF reader knows at this point that the tag is updated and can be read.

|  |  |  |
| --- | --- | --- |
| **ECG tag update - sequence** | | |
| **step** | **RF\_REQUEST value** | **Who writes ?** |
| 1 | 0xC3 | RF reader |
| 2 | Random byte | ECG |
| 3 | Random byte 2’s complement | RF reader |
| 4 | 0xCC | ECG |
| Tag update (by ECG) | | |
| 5 | 0x00 | ECG |

**NEW\_RF\_DATA**A RF reader notify the presence of new data in the tag by writing the upper nibble of this register to 0x5. The lowest nibble tells the ECG which data group has been updated:

Bit0 = 1: “quasi constant” data updated

Bit1 = 1: “frequently changing” data updated

Bit3 = unused

Bit4 = unused

**QCONST\_CRC**“quasi constant” data group CRC.

**FCHANG\_CRC**“frequently changing” data group CRC.

|  |  |
| --- | --- |
| **Byte addr** | **Description** |
| 0 | **ECG\_ON** |
| 1 | **RF\_REQUEST** |
| 2 | **-** |
| 3 | **NEW\_RF\_DATA** |
| 4 | **QCONST\_CRC** |
| 5 |
| 6 |
| 7 |
| 8 | **FCHANG\_CRC** |
| 9 |
| 10 |
| 11 |
| 12 | **User data** |
|  |
|  |
|  |
| ... |
| ... |
| ... |
| ... |
|  |
|  |
|  |
| 2047 |

# Passwords

M24LR16 allows to set three different “hardware” passwords from RF side. The 16 memory sectors can be individually associated with one of these passwords with different read/write access rights.

A basic access (read/write) password is used to lock all the memory sectors so that unwanted RF writing operations are discouraged.

On top of this a security software layer with 2 passwords protects data. A RF reader that aims to update an ECG shall provide a valid passwords in order to have data accepted by µC.

An encrypted copy of the password is present in the tag so that a RF reader can determine when new data password matches encrypted password (even if the µC is off) and can eventually give immediate feedback to the operator.

Passwords are shared by RF and Dali side. Whenever a password is modified on one side the change is immediately mirrored on the other.

# NFC API

The µC application layer provides the methods to handle data in NFC tag. Instead of extending nvm module, a new NFC module is proposed in order to have much more flexibility.  
  
The module API is:

**1. nfcInit** ( void )

**2. nfcDataInit** ( void \*pData, uint32\_t aSize, uint32\_t aAddr, data\_type\_t aType )

**3. nfcManager**( void )

**4. nfcTimer** ( void )

**5. nfcWriteAll** ( void )

**7. nfcUpdateStart** ( void )

**8. nfcUpdateStop** ( void )

**6. nfcRefreshWdt** ( void )

A brief description follows:

1. **1. nfcInit**

Initializes the external NFC tag in order to be used as non-volatile memory and RF communication port. It checks whether there is new data to upload from NFC tag and sets “new\_nfc\_data” flag.

1. **2. nfcDataInit**

This function initializes data in NFC tag by filling in a table where RAM addresses are associated with tag addresses. The user can specify any tag address for each parameter: care should be taken so that addresses fall within the valid range and no overlaps occur.

Parameters:

“pData” is a pointer to RAM declared data.

“aSize” size in bytes of data pointed by “pData”.

“aAddr” absolute address where data is saved in NFC tag.

“aType” specifies if this is a “quasi constant” or “frequently changing” parameter.

**Note:** this function addresses the problem of big data structures that contain a single or few elements of a different type. In order to assign a sub-struct with a different data type the function has to be called a second time. An example is “InfoPage” memory bank where besides “quasi constant” parameters the “lamp\_operation\_counter” is of “frequently changing” type. See also example in chapter 15 - NFC module setup.

1. **3. nfc**

This function shall be placed in the main loop and called as frequently as possible. It performs several background tasks:

1. Receives requests from RF reader (write new data requests or update requests)
2. Checks whether new consistent (CRC check) data is present in the tag and copies it in RAM.
3. Compares RAM data with tag data and if any mismatch is found it updates tag data.
4. Checks tag “frequently changing” data integrity and re-writes it if CRC corrupted or if CRC stored in tag differs from latest valid CRC calculated.

**Note:** at power-on the first operation is checking if new RF data is present.

**Note:** during data upload a) all the interfaces (Dali) that potentially modify at interrupt the data shared with NFC interface shall be temporarily stopped.

Writing in NFC tag is done incrementally, one page (4 bytes) at each call (~5 ms duration). For 500 bytes it takes at least 625 ms.

1. **4. nfcTimer**

This function shall be placed in a proper timer call. The timer must have a maximum period of 10 ms. It is used by NFC background tasks and is supposed to be called at interrupt so that even with blocking calls i.e. **nfcWriteAll**() the timer is updated.

1. **5. nfcWriteAll**

Saves RAM data in NFC tag. Data is first read from NFC and compared to RAM data and only if they do not match the operation is executed. This is a suspending function that takes several ms before returning and heavily depends on the amount of data that must be saved. It is intended to be called when an µC power-down event is detected. See also **nfcRefreshWdt**().

1. **6. nfcRefreshWdt**

This function shall be defined somewhere by the application designer in order to compile the NFC module. This function is called inside **nfcWriteAll**() in order to keep the watchdog timer refreshed. It is needed when the a big amount of data must be saved at power-down. A timeout guarantees the program to continue in any case after 5 seconds.

1. **7. nfcUpdateStart**

This function shall be defined somewhere by the application designer in order to compile the NFC module. This function is called by the NFC module just after an RF reader write request has been correctly acknowledged. It can be used, whenever needed, to switch the power stage off during RF reader data update in order to improve the RF communication or to stop any other communication port in order to avoid data modifications during RF update.

1. **8. nfcUpdateStop**

This function shall be defined somewhere by the application designer in order to compile the NFC module. This function is called by the NFC module at the end of a RF tag update (process done or timeout occurred). It can be used to switch the power stage on again after the **nfcUpdateStart**() has switched it off.

# Tag API

Tag interface is implemented for different chips. Here M24LR16 is considered:

1. **M24LRxx\_Init**( void )
2. **M24LRxx\_GetStatus** ( void )
3. **M24LRxx\_ReadReg** ( reg\_addr\_t aRegName, uint32\_t \*pRegValue )
4. **M24LRxx\_ReadByte** (uint16\_t aAddr, uint8\_t \*pBuf)
5. **M24LRxx\_ReadBuffer** (uint16\_t aAddr, uint16\_t aByteNum, uint8\_t \*pBuf)
6. **M24LRxx\_WriteReg** (reg\_addr\_t aRegName, uint8\_t aRegValue)
7. **M24LRxx\_WriteByte** (uint16\_t aAddr, uint8\_t aValue)
8. **M24LRxx\_WritePage** ( uint16\_t aAddr, uint32\_t aValue)
9. **M24LRxx\_WriteBuffer**(uint16\_t aAddr, uint16\_t aByteNum, uint32\_t aValue)
10. **M24LRxx\_Manager** (void)
11. **M24LRxx\_ReadReq** (uint16\_t aAddr, loc\_type\_t aLocType, uint16\_t aByteNum, uint8\_t \*pBuf)
12. **M24LRxx\_WriteReq** (uint16\_t aAddr, loc\_type\_t aLocType, uint16\_t aByteNum, uint8\_t \*pData)
13. **M24LRxx\_GetReqStatus**(void)
14. **M24LRxx\_Timer**(void)

# I2C bus API

The I2C communication interface is:

1. **I2C\_Init** ( void )
2. **I2C\_MasterTransmit**(uint16\_t aSlaveAddr, bool aStop, uint8\_t \*pTxBuf, uint32\_t aTxBufSize)
3. **I2C\_MasterReceive** (uint16\_t aSlaveAddr, bool aStop, uint8\_t \*pRxBuf, uint32\_t aRxBufSize)
4. **I2C\_IsBusy**( void )
5. **I2C\_GetCode**( void )
6. **I2C\_GetDataCount**( void )
7. **I2C\_Timer**( void )

# I2C driver API

The driver I2C implementation is µC dependent but with this common API:

1. **i2c\_drv\_master\_init** ( void );
2. **i2c\_drv\_master\_rx\_irq\_callback** ( void );
3. **i2c\_drv\_master\_tx\_irq\_callback** ( void );
4. **i2c\_drv\_master\_err\_irq\_callback** ( i2c\_code\_t ecode );
5. **i2c\_drv\_master\_start** ( uint32\_t slave\_addr, i2c\_master\_cmd\_t rw, bool rep\_start);
6. **i2c\_drv\_master\_tx\_write** ( uint8\_t data);
7. **i2c\_drv\_master\_tx\_read** (i2c\_ack\_t ack);
8. **i2c\_drv\_master\_get\_rx\_data** ( void );
9. **i2c\_drv\_master\_stop** ( void );

# NFC module setup

This paragraph describes how to properly setup and use the NFC module. For further information refer to the related paragraph.

1. **I2C configuration**

Load in your project the following files:

* I2C.c
* I2c\_driver\_xmc1300.c (XMC1300 µC)

In ConfigI2C.h de-comment the pair of pins used for I2C SDA and SCL signals (example):

* **#define** I2C\_\_SCL\_PIN\_2\_0\_\_SDA\_PIN\_2\_1

In order to make the I2C module working a timer must be used at interrupt. In ConfigI2C.h specify the period of the timer used to call **nfcTimer**() that in turn implicitly calls **I2C\_Timer**(). The period of the timer is defined by:

* **#define** I2C\_TIMER\_PERIOD\_US (1200) // [us]

**Note:** the period must be smaller than 10 ms (example 1.2 ms).

1. **M24LRxx configuration**

Load in your project the following file:

* M24LRxx.c

In ConfigM24LRxx.h de-comment the part number of the chip in use (example M24LR16ER with 2KB of EEPROM):

* **#define** M24LR16E\_R

1. **Nfc configuration**

Load in your project the following file:

* nfc.c

Include in your main.c file:

* **#include** "nfc.h"

In ConfigNfc.h define the maximum number of parameters/data structs that you intend to initialize and the size of the greatest parameter/data struct among these. The first define tells how many times **nfcDataInit**() must be called.

* **#define** NFC\_DATA\_STRUCTS\_CNT\_MAX (64)
* **#define** NFC\_STRUCT\_MAX\_SIZE\_BYTE (128)

**Note:** every time a nested parameter/struct is initialized “NFC\_DATA\_STRUCTS\_CNT\_MAX“ must be incremented by one (see example at the end of the paragraph).

In the application program:

* Place **nfcInit**() in the initialization part i.e. before main loop.
* Place **nfcManager**() in the main loop so that it is called as frequently as possible.
* Place **nfcTimer**() in a timer called at interrupt with a period smaller than 10 ms. See define “I2C\_TIMER\_PERIOD\_US” configuration.
* Place **nfcWriteAll**() in the part of the application program executed when a power-failure is detected. This function saves parameters at power-down.

**Note:** after this function call **nfcManager**() shall not be executed until the µC stops (no power-supply) or until the power-supply is restored.

* Define **nfcRefreshWdt**() with a watchdog refresh call. This function is called during **nfcWriteAll**() (blocking function) for some seconds just to give the time to write data in the tag.
* Define **nfcUpdateStart**(). This function is called when a valid RF reader write request is accepted and the tag is supposed to be updated. It is typically used to switch the ECG output power off in order to improve the RF communication.
* Define **nfcUpdateStop**(). This function is called when the tag update is done. It is typically used to switch back the ECG output power on after it had been switched off by **nfcUpdateStart** ().

Make then a list of the parameters you want to save in the tag with the related size. Identify data that is expected to change frequently i.e. more often than 10 minutes. This data **must** be declared as “frequently changing” data. Constant data or quasi-constant i.e. not changing several times in the ECG life-time belong to the “quasi-constant” class. Define then the address in the tag where you want to place each item.

Once the list of parameters is ready:

* Initialize each parameter by calling **nfcDataInit**() before getting into the main loop.

**Note**: if you declare “quasi constant” a parameter that is frequently changing the **nfcManger**() will write it in the tag every time a mismatch between RAM and tag copy is detected. This might wear the tag EEPROM out.  
It is therefore safer to declare parameters as “frequently changing” in case of uncertainty.

**Note**: the latest valid address of the tag is TAG\_MAX\_SIZE\_BYTE – 12 (bytes).

**Note**: if a data struct contains “quasi-constant” members and “frequently changing” members too, the initialization shall be done in two steps. First the complete struct is initialized as “quasi-constant” data, the “frequently changing” part is then initialized in a second step. The vice-versa is allowed too. An example follows.

**Example:**

// Mixed data struct.   
// Elements of the struct are “quasi-constant” but “life\_cnt\_minutes” that changes frequently i.e. every minute.

typedef struct

{

uint32\_t ECG\_Name;

uint32\_t ratedIout;

uint32\_t life\_cnt\_minutes; // -> “frequently-changing”

uint32\_t life\_cnt\_hours;

uint32\_t life\_cnt\_days;

} mix\_data\_t;

// Data declaration  
mix\_data\_t my\_mixed\_data;

// Here the initialization of the mixed struct. First the complete struct is initialized as “quasi-const” data.  
nfcDataInit ( &my\_mixed\_data, sizeof(mix\_data\_t), 100, NFC\_DATA\_TYPE\_QCONST);

// The “frequently-changing” data is then initialized.  
nfcDataInit ( &my\_mixed\_data .life\_cnt\_minutes, sizeof(uint32\_t), 200, NFC\_DATA\_TYPE\_ FCHANGING);

**Note**: the “quasi constant” part of the struct is placed at address 100 of the tag EEPROM while the “frequently changing” parameter is placed at address 200. This is an arbitrary design decision the application designer takes. The resulting tag memory layout is:

|  |  |
| --- | --- |
| **Variable name** | **Tag address** |
| ECG\_Name | 100 |
| ratedIout | 104 |
| life\_cnt\_minutes | 200 |
| life\_cnt\_hours | 112 |
| life\_cnt\_days | 116 |

**Note:** care must be taken in defining the tag memory layout. Each parameter/struct shall be placed in a unique memory location that does not overlap any other data present in the tag.

# NFC module test

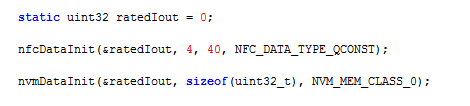
Once the NFC fw module has been integrated in the application program as previously described it must be tested. Please first check the NFC demo software as reference project (**svn rev.835**) from the repository:

[http://app-mchsvn01.mch.osram.de/svn/EC/Platforms/00 Tests and Doc/UnitTestNfc/test\_iar/oti\_xmc\_v472](http://app-mchsvn01.mch.osram.de/svn/EC/Platforms/00%20Tests%20and%20Doc/UnitTestNfc/test_iar/oti_xmc_v472)

selecting *DEMO\_SW\_PC*as workspace configuration.

Here below a quick procedure is proposed to verify whether basic functionalities are working in a real ECG.

* Equip an ECG with the NFC HW.
* Initialize before the main loop an unsigned long variable (here called *ratedIout*) as nfc data, located at tag address 40 and of type “quasi-constant”. Make the variable non-volatile as well by calling nvmDataInit(). See code snippet below:



* The variable *ratedIout* can take values from 100000 to 1000000. Scale this variable and use it to modify your ECG output current at run-time.
* Compile the code and update your ECG equipped with NFC board.
* Download PC demo software from : [N:\!Altre\_Localita\DE\Muenchen\EC\D\NT\LLS EC\_Team 3\WiSet\DemoSW\NFC RFID Tag Reader\_Writer - DEMO v0.1.2.zip](file:///\\osram.de\Net-mch\EC\D\NT\LLS%20EC_Team%203\WiSet\DemoSW\NFC%20RFID%20Tag%20Reader_Writer%20-%20DEMO%20v0.1.2.zip)
* Connect a MB1054 RF reader demo board to your PC and launch *RFID\_wr.exe*.
* Alternatively set 10% and 100% with the demo software (see pic below). The variable *ratedIout* should change from min to max value and the output current according to the relation established.

**Note**: the above described operation can be done with ECG Off or at run-time too.



# Annex A – Todos

1. RF reader is expected to set on the complete NFC chip a HW protection (password) so that data is not RF readable nor writable when not needed. The password in use is one out of three available for all tag’s sectors and is predefined by Osram. When presented, tag’s sectors become RF readable and writable. If anybody who gets the password can change it and make the tag not reachable anymore by Osram tools. In order to get out of this situation NFC driver should continuously check if password has changed. In this case it should remove the protection via I2C.

# Annex B – Open issues

This table summarizes the open questions raised during the first part of the firmware design phase.

|  |  |  |  |
| --- | --- | --- | --- |
| **Owner** | **Question** | **R&D preference** | **Final reply** |
| NXP | When NTAG S sample will be available ? | NA | NA |
| NXP | When NTAG S first datasheet release will be available ? | NA | NA |
| MKT | Is ECG config downloaded in NTAG only off-line (ECG µC off) ? | yes | Not always |
| MKT | Are 200K writing cycles enough  (no paging needed) ? | yes | - |
| MKT | Are only OSRAM tools used to interface with NTAG ? | yes | yes at the moment |
| MKT | A password (KeyA) must always be provided even to read tag (relevant if non OSRAM tools are used) ? | yes | yes |
| MKT | Does tag have same password used for Dali MB ? | no | - |
| R&D | Is 1K version enough for all use cases ? | no | no |
| R&D | Chip selected | M24LR16 | M24LR16 |

NA = not applicable because refers to NXP chip.

# Annex C – Tag memory management

The location of each parameter in the tag memory must be known or derived by both RF reader and µC (I2C) before any read or write operation is done. Different approaches are possible:

1. Each parameter has a fixed predefined address in memory. Simple but not flexible: in the µC each parameter has to be assigned manually with an absolute tag address.
2. Parameters are grouped (memory banks, dali parameters, calibration data etc. ) and for each group a fixed predefined address is given. Not flexible but only group addresses must be specified in µC. From µC point of view this is equivalent to point 1. From RF side this means that when the µC fw is compiled every parameter address of a data group must be derived and specified in XML file.
3. A TOC in the tag gives the address of each parameter. The TOC index defines which is the parameter the address refers to. Memory layout in µC can change without affecting the tag memory. Semi-flexible solution that requires the definition of the TOC index sequence. TOC is quite expensive in terms of size. Complete TOC must be read by RF reader before read/write operations: expensive in terms of access time.
4. A TOC in the tag gives the address of each group of parameters. The TOC index defines which is the group of parameters the address refers to.
5. An identifier is given for each parameter. A TOC associates each identifier with the relative parameter address. Maximum flexibility but a large part of the memory is taken by TOC. Complete TOC must be read by RF reader before read/write operations: expensive in terms of access time.
6. An identifier is given for each group of parameters. A TOC associates each identifier with the relative group address.

The below table highlights which strategy minimizes the T4T updates when the related change is implemented in ECG:

Table 1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Change** | **1** | **2** | **3** | **4** | **5** | **6** |
| Parameter position in ECG mem bank changes (for example a parameter is removed from mem bank) | **X** | **X** | **OK** | **X** | **OK** | **X** |
| Pad byte added in ECG mem bank | **X** | **X** | **OK** | **X** | **OK** | **X** |
| ECG parameter size changes | **X** | **X** | **X** | **X** | **X** | **X** |
| New parameter added in ECG | **X** | **X** | **X** | **X** | **X** | **X** |

Given the following assumptions:

* 200 parameters
* 400 bytes total parameters size
* 50 groups of parameters (mem banks, dali param group, calib data group etc..)
* 16 bits addresses to cover 2KBytes of memory
* 8 bits identifiers (< 256 parameters in any case)

a raw estimation of memory needed for each strategy follows:

1. Data = **400** bytes
2. Data = **400** bytes
3. Data + address = 400 + (200 x 2) = **800** bytes
4. Data + address = 400 + (50 x 2) = **500** bytes
5. Data + identifier + address = 400 + 200 + (200 x 2) = **1000** bytes
6. Data + identifier + address = 400 + 50 + (50 x 2) = **550** bytes

Some considerations from **T4T** side:

* strategy 1 requires to specify tag parameters absolute address and relative size.
* strategy 2 requires to specify tag parameters absolute address or groups absolute address with parameters offset inside the group and relative size.
* strategy 3 requires to specify TOC parameters index and relative size.
* strategy 4 requires to specify TOC group index, parameters offset inside the group and relative size.
* strategy 5 requires to specify parameters ID and size.
* strategy 6 requires to specify groups ID, parameters offset inside the group and relative size.

... and from **µC** side:

* strategy 1 requires to specify tag parameters absolute address.
* strategy 2 requires to specify tag groups absolute address.
* strategy 3 requires to specify TOC parameters index.
* strategy 4 requires to specify TOC group index.
* strategy 5 requires to specify parameters ID.
* strategy 6 requires to specify groups ID.

**Conclusion**:

* **strategy 1**: not flexible but simple. Requires a lot of assignments on µC side i.e. every parameter must be assigned with an address. **Discharded.**
* **strategy 2**: not flexible but simple. Requires a limited number of assignments on the µC side i.e. only the starting addresses of parameters groups are needed. It is assumed that positions of parameters inside a struct (like memory banks) do not change frequently across different fw versions. If a new parameter is added to a mem bank T4T shall be updated in any case. **Adopted.**
* **strategy 3 or 5**: flexible but expensive in term of memory space due to several TOC entries and in terms of access time (TOC reading). In order to shrink TOC size 11 bits addresses might be used but this makes the TOC layout and handling more complex. Requires a lot of assignments on µC side i.e. every parameter must be assigned with TOC indexes or ID. **Discharded.**
* **strategy 4 or 6**: compromise in terms of space needed by the TOC and number of assignments on the µC side. The solutions gives small added value see Table 1. On the T4T side each parameter shall be specified with address, offset and size. **Discharded.**