

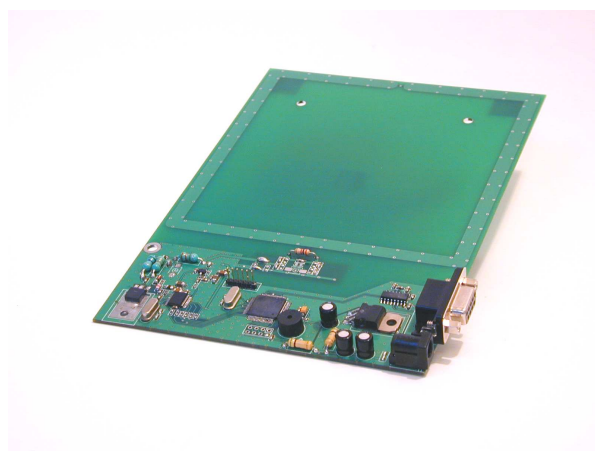
### 1. Scope

This application note is a description of the firmware FW90121 intended to help developers to implement ISO compliant communications with the MLX90121 RFID transceiver. It will guide the developers to build and customize a MLX90121 based RFID reader device and get its best performances. This document describes the structure of the firmware made and represents the structure of the firmware from different points of view, possibility of use the part of code as a library and describes how to expand the code.

### 2. Applications

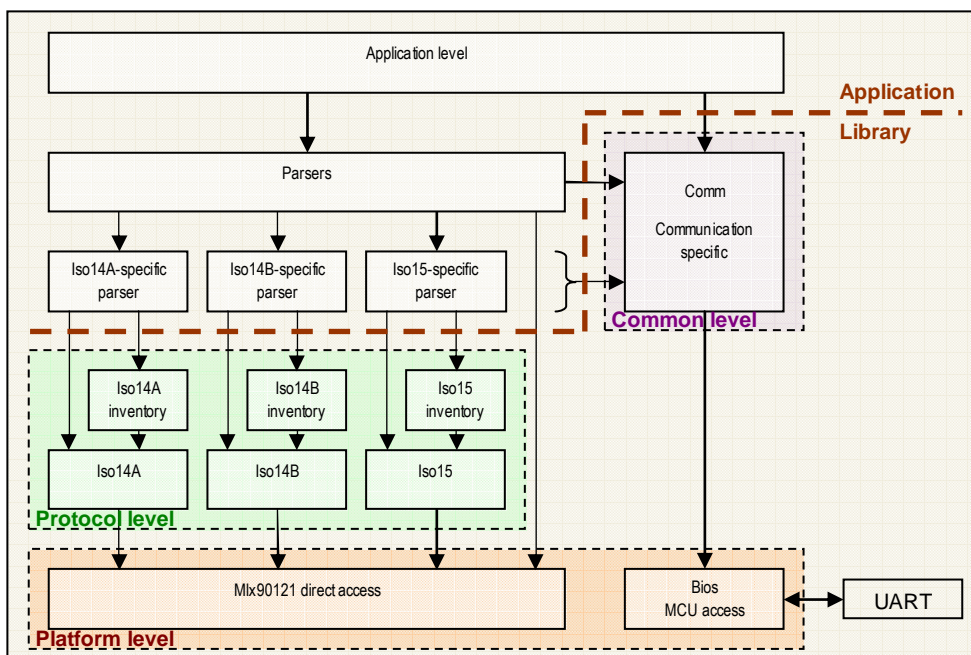
The firmware described in this application note could be use for the development of the following applications:

- Portable data terminals
- Access control readers
- Contact-less payment terminals
- Smart label printers
- Electronic passport readers



### 3. Related Melexis Products

This firmware addresses the use of the MLX90121 with a 8 bit microcontroller. Its maximum need of CPU power is around 8MIPS.



*Important note: This document and the related software may be downloaded onto a computer, stored and duplicated as necessary to support the use of the related Melexis product. Any other type of duplication, circulation or storage on data carriers in any manner are not authorized by Melexis unless specific agreement.*

## 4. Table of Content

1. Scope .....	1
2. Applications .....	1
3. Related Melexis Products .....	1
4. Table of Content.....	2
5. Code optimization criteria.....	3
6. Structure models.....	3
6.1. Location model.....	3
6.2. Logical model .....	6
6.3. Data flow model.....	6
7. Hardware resources.....	8
7.1. Platform dependency avoidance.....	8
7.2. Microcontroller external pins .....	8
7.3. Timer .....	8
7.4. UART.....	9
8. Parser .....	10
8.1. Theory of working.....	10
8.2. How to make own parser and extend functionality.....	12
9. Communicating with a TAG.....	9
9.1. Sending data, receiving data, inventory .....	9
9.2. Error treatment .....	9
10. Function description.....	10
10.1. Module Mlx90121.c.....	14
10.2. Module Crt.c.....	15
10.3. Module Comm.c .....	16
10.4. Modules ErrorCodes.c and ErrorCodes.h .....	19
10.5. Communication module Iso15.c .....	19
10.6. Communication module for inventory Iso15Inv.c .....	20
10.7. Communication module Iso14B.c.....	20
11. Conclusion.....	22

## **5. Code optimization criteria**

The code must be as simple and as portable as possible. It must be clear to read and easy to expand. User must be able to add his own features and commands.

Atmel AVR microcontroller is taken as a base. Though the way the code is implemented is theoretically portable. Meanwhile, a certain effort is needed to transfer the code to another microcontroller.

The part of the code may be taken as a library and the whole structure is hierarchical and doesn't contain cross-module dependencies.

The top-level interface of some modules is fixed, though the algorithm implemented is a subject of constant change and improvement. Some minor internal service functions can also be changed and user is discouraged of using them directly. This gives the certain flexibility to improve or optimize the code later.

## **6. Structure models**

This chapter represents the structure of code from different points: how it is organized, distributed by folders and how different modules and functions depend on each other.

### ***6.1. Location model***

This structure describes the code 'as is', how modules are located in folders and dependencies between modules. It does not describe dependencies between functions which can be different.

In the following picture folders are shown in yellow, files in green. Note, that each branch folder has 2 subfolders: Src and Inc, where sources (\*.c) are located in Src and headers (\*.h) in Inc. These folders are not shown for simplicity.

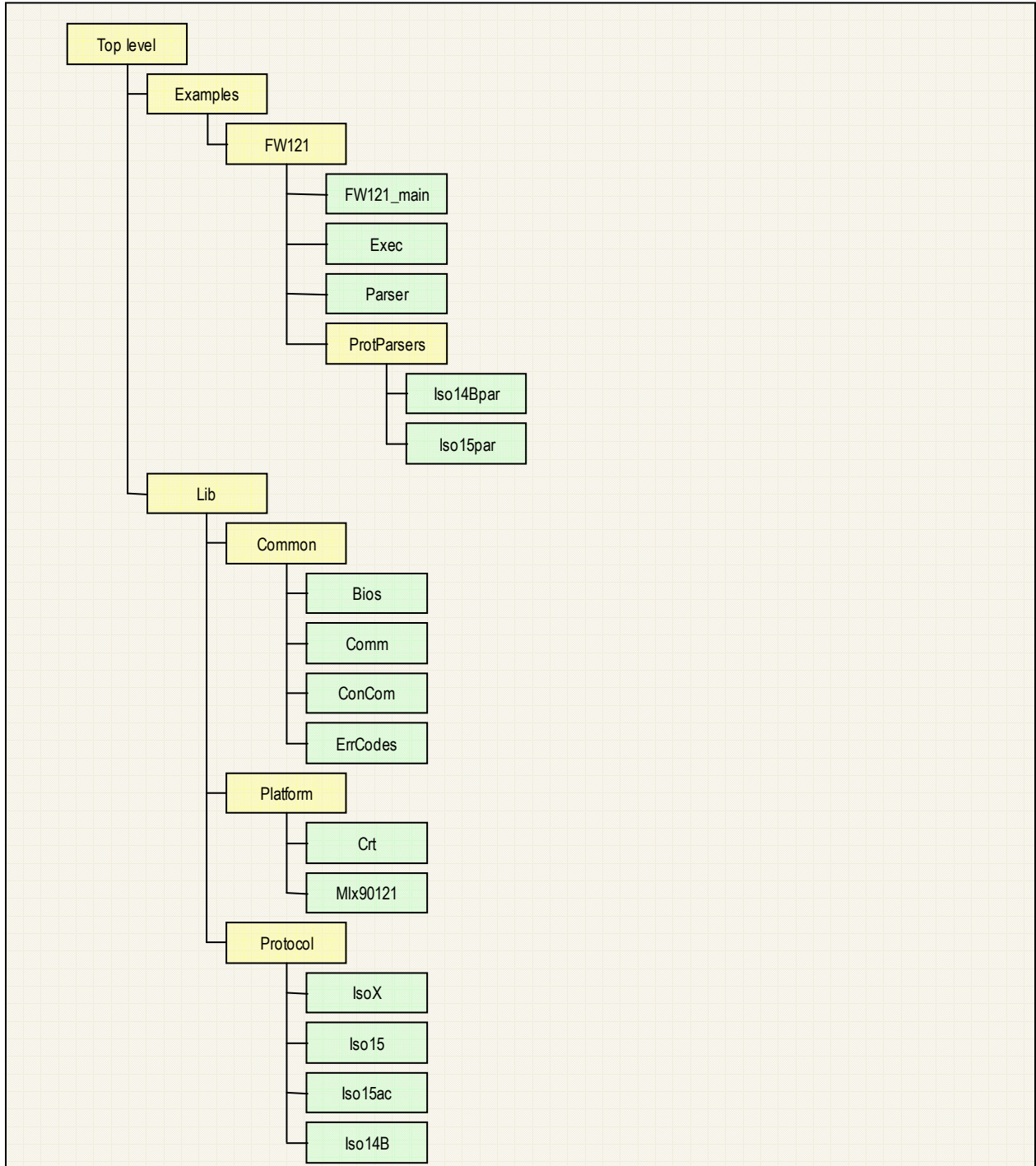


Figure 6.1.1. Location structure

The top level contains 2 subfolders: 'Examples' and 'Lib'.

The folder 'Examples' is a collection of application examples which have been developed using the modules provided in the folder 'Lib'.

The folder 'Lib' is a collection of modules which can be used by different applications. These modules are independent and can be used separately.

The folder 'Lib' contains 3 subfolders: 'Common', 'Platform' and 'Protocol'.

The folder 'Common' is a collection of some common routines, like timer interrupt routines, UART communication functions, data conversion functions. They can be used separately for different applications.

The folder 'Platform' has 2 files: 'Crt' and 'Mlx90121' which contain microcontroller specific and mlx90121-specific functions respectively.

The folder 'Protocol' is a collection of protocol-specific functions. File 'IsoX' is used by all protocols. It contains the description of TAG reception buffer, common for all protocols in order to minimize the usage of RAM in case multiple protocols are used. It also contains the CRC calculator, since for many protocols it is the same. Each of the files 'IsoXX.c' (where 'XX' corresponds to a protocol) contains 2 basic interface functions: IsoXSend and IsoXRecv which are used to send and receive data respectively. The interface for these functions is fixed and users are encouraged to use them rather than intermediate functions. Apart of 'IsoXX.c' for each protocol there is also a file 'IsoXXac.c' which contains inventory function. 'IsoXXac.c' uses functions described in 'IsoXX.c' and cannot be used separately. For example, even if application is supposed to do only inventory, it needs to contain both 'IsoXXac.c' (where inventory functions are described) and also 'IsoXX.c', which is implicitly used by inventory.

### 6.2. Logical model

This structure represents the firmware structure from application point of view: data flow and function call stack. Dependencies between function calls are shown by arrows.

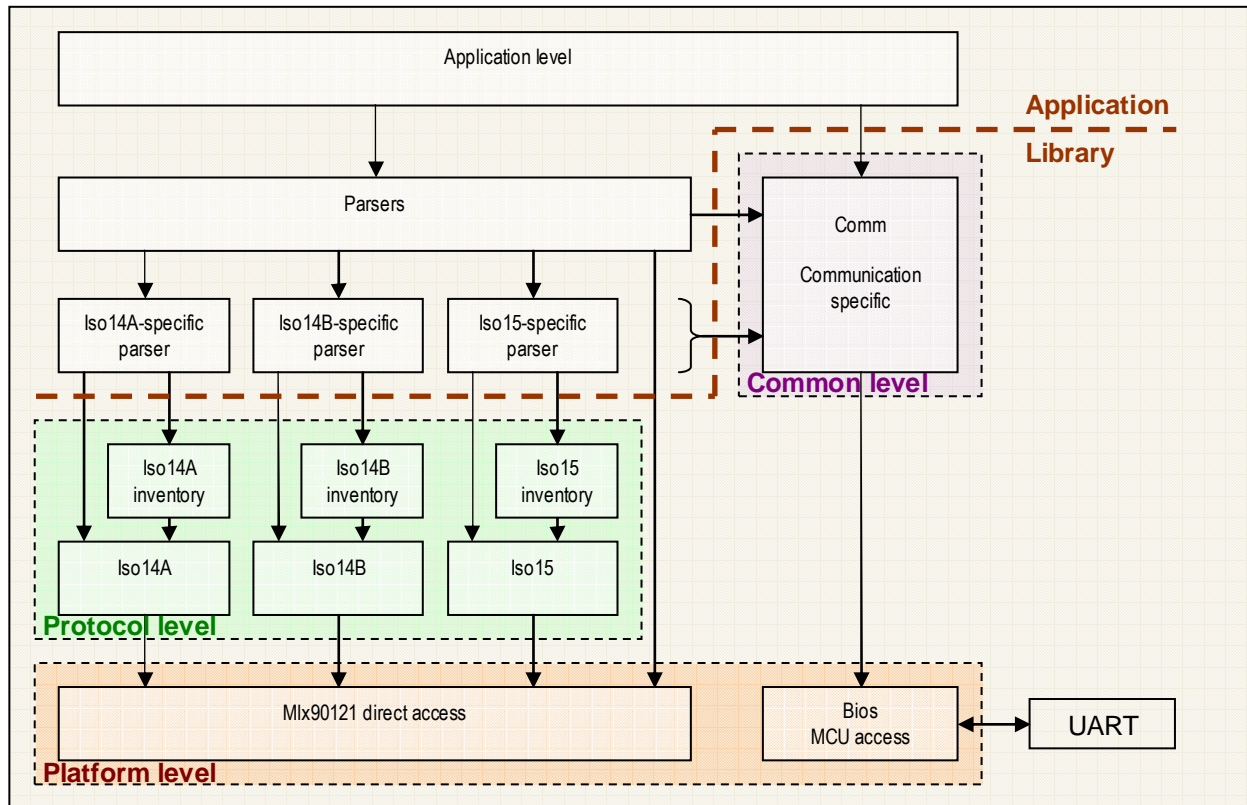


Figure 6.2.1. Logical structure

This model doesn't represent the data flow. It shows only functional and module dependencies. For example, Iso15 protocol specific parser can access functionality of Iso15 protocol level directly. If Inventory functions are not needed, Iso15-Inventory module can be omitted.

### 6.3. Data flow model

This model is very important, because it shows the flow of data through the whole stack from UART to the TAG and back. This model explains the work of command parsers and TAG response checker. This model is somehow similar to the 'Logical model' previously explained, but instead of dependencies shows the direction of data flow.

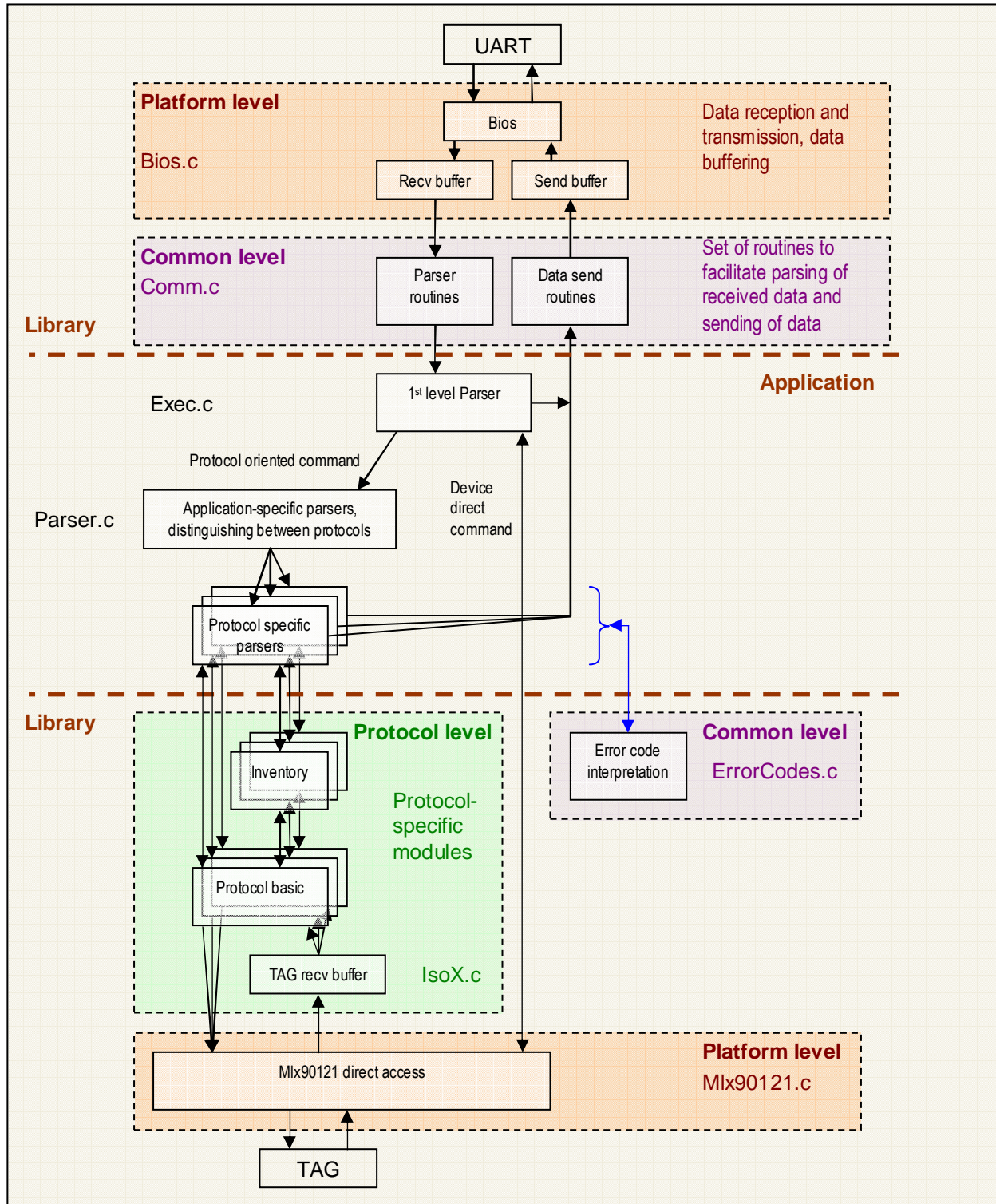


Figure 6.3.1. Data flow

## 7. Hardware resources

The application uses some resources of the microcontroller, such as Timer, communication port UART, microcontroller external pins. All these resources are platform-dependent and need to be changed if developer intends to use another microcontroller.

### 7.1. Platform dependency avoidance

Though it is mostly impossible to avoid platform dependency completely, some rules have been respected to minimize this dependency.

Library modules never use special function registers of the microcontroller directly. There is all the time intermediate function. The good example is timer: application never configures timer directly, it uses a set of functions described in module 'Crt'.

Library modules never use direct access to the microcontroller ports. They use definitions of signals described in header file 'HwDefs.h' which belongs to the application. Developers must edit this file.

At the beginning application needs to properly configure microcontroller to use a certain set of resources.

### 7.2. Microcontroller external pins

The Mlx90121 is connected to the external pins of the microcontroller. In total, 6 signals are used.

Mlx90121 signal name	Function	Direction	Configuration in the microcontroller side
DIN	Input data to be send	Mcu → Mlx	Output
DOUT	Output data which was received	Mcu ← Mlx	Input
CK	Clock signal	Mcu → Mlx	Output
DSYN	Synchro signal	Mcu ← Mlx	Input
MODE	Configuration or communication mode	Mcu → Mlx	Output
RTB	Data transfer direction	Mcu → Mlx	Output

The microcontroller pins must be properly configured at the beginning of the application. File 'HwDefs.h' contains the definition of all the signals.

### 7.3. Timer

The protocol functions actively use timer. The module 'Crt.c' contains functions which work with timer. In the current example Timer-2 is used. If developers intend to use another timer, they must make changes in the module 'Crt.c' and in the header file 'Crt.h'.

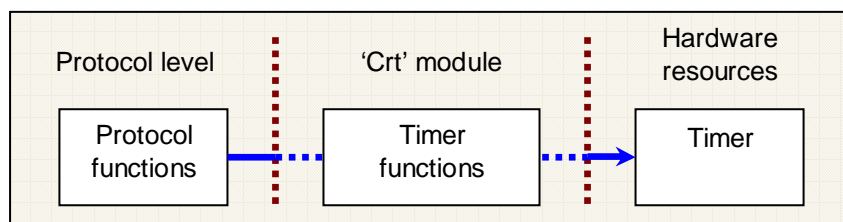


Figure 7.3.1. Access to the resources

## **7.4. UART**

Serial communication resource is similar to the Timer. The rule is the same: application does not access the hardware resource directly. Using communication resource is even simpler, because it is already transparent to the application. Indeed, application uses communication functions from module 'Comm' to communicate. See following explanation for details.

## **8. Communicating with a TAG**

### **8.1. Sending data, receiving data, inventory**

Each protocol supposes to have 2 basic functions: IsoXXSend and IsoXXRecv (where 'XX' corresponds to the protocol). Also there are 2 inventory functions: IsoXXInvStart and IsoXXInvGetNext.

- **IsoXXSend** – sends data to the TAG
- **IsoXXRecv** – receives a response
- **IsoXXInvStart** – starts inventory procedure
- **IsoXXInvGetNext** – returns the ID of the next TAG; application must call this function in a loop until it returns an error code 'NoMoreTags'

Most of these functions require a buffer where data to send is stored or where function can store data received from TAG. The application can use a buffer FTagRecvBuf declared in 'IsoX.c' file. This buffer can be used for multiple purposes and is declared to save expensive RAM resource of the microcontroller. Application can also declare and use own buffers.

Application must prepare a data packet before sending. IsoXXSend is responsible only for encoding data and sending it correctly. It will add SOF and EOF and optionally will calculate CRC but is not responsible for protocol level. During reception function IsoXXRecv decodes TAG response and fills the reception buffer with data. This buffer is returned to the application, and application is responsible for the interpretation of this data.

### **8.2. Error treatment**

All TAG communication functions return a code. It can be difficult for the application to interpret this code. File 'ErrorCodes.h' contains all error codes description and file 'ErrorCodes.c' contains a function ErrGetCodeStr, which helps to convert an error code to its readable representation. It returns a pointer to a string which can be used to send as a comment to the user. Application developers are encouraged to use this function rather than making own error code interpreter.

## 9. Parser

### 9.1. Theory of working

After incoming data is received, it is placed into the UART reception buffer. All further operations are performed on this buffer, even in application level. The operation on data interpretation starts when CR-symbol is received (#13). Of course, during interpretation more data can arrive. This data is stored in the same buffer after the end of CR symbol. Program keeps 2 pointers: PacketSize and DataSize which point to the end of packet received and to the end of data received respectively. See following example.

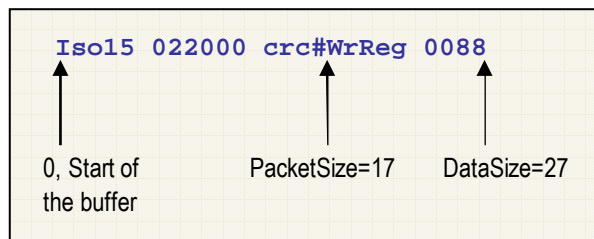


Figure 8.1.1. UART reception buffer example 1

Here by symbol # we show CR (#13). PacketSize=17; DataSize=27.

PacketSize is 0, when no packet is received. In the picture above, the packet is received. During interpretation new data starts to arrive. After received packet is interpreted and operation is performed, program deletes the packet from the buffer. Data, which is received later, copies to the beginning of the buffer.

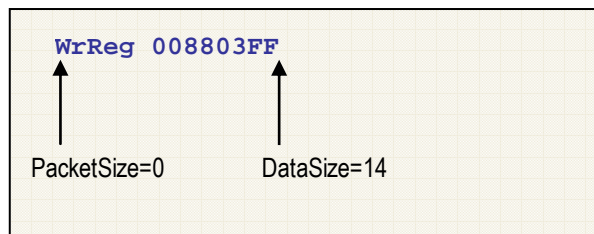


Figure 8.1.2. UART reception buffer example 2

Now PacketSize is 0, indicating that there is no completely received packet yet.

The main function for parse data is called [CommCmpStrC](#). This function compares the string given as a parameter with the first word found in the reception buffer. It returns TRUE, if specified word is found and FALSE otherwise. Comparison is case-insensitive. Let's look to the following example.

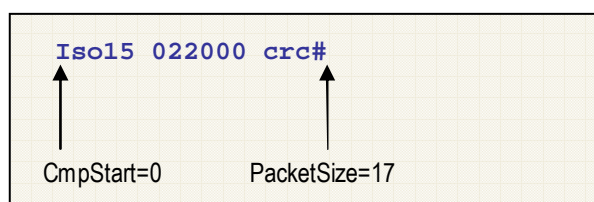


Figure 8.1.3. Parsing example

CommCmpStrC("Iso15") returns TRUE because the 1<sup>st</sup> word in the buffer is 'Iso15';  
 CommCmpStrC("IDN") returns FALSE;  
 CommCmpStrC("crc") returns FALSE. Though word 'crc' is in the buffer, it is not at the 1<sup>st</sup> place;  
 CommCmpStrC("Iso") returns FALSE. Words are supposed to be separated by spaces, 'Iso' <> 'Iso15'.

If specified word is not found (function returned FALSE) the buffer will stay unchanged. If it is found (function returned TRUE) then the first word will be deleted. This is the main idea of working of this function; it allows simple and effective parsing of the parameters. Telling that 'first word will be deleted' means only moving a pointer CmpStart to a new position. Buffer will not be changed at this time, but will be updated later with a deletion of a complete packet. After executing CommCmpStrC("Iso15") buffer will look like following

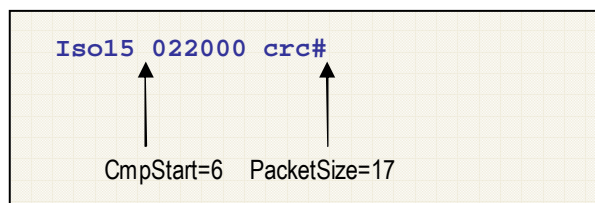


Figure 8.1.4. During comparison only pointer will change

CommCmpStrC("") returns TRUE at the end of packet, i.e. there is no more parameters to compare.

The fact that unknown parameter stays in the buffer and pointers do not move simplifies error reporting. For example, there is a function called [CommSendStrC/CommSendStrC13](#). This function sends data to the UART. It analyzes the data before sending and when it finds #1, it replaces it with the first parameter from the UART reception buffer. For example, we are trying to parse the string

**Iso15 022000 crc blablabla**

Algorithm will look like

```
do
{
  if (CommCmpStrC("")) { exit loop, no more parameters }
  else if (CommCmpStrC("crc")) { Calculate CRC flag = TRUE }
  else { CommSendStrC13("Error: Invalid command [\1]") exit with error }
} while(1);
```

Underlined line will be executed and string 'Error: Invalid command [\1]' will be passed to a function as a parameter. Prior to send it will analyze a string and replace '\1' with a parameter from the UART receiver buffer. 'Error: Invalid command [blablabla]' will be sent through the UART.

Another important function which is used during command interpretation is `CommDecodeHex`. It decodes hexadecimal value and copies it to the beginning of the UART reception buffer. Function returns TRUE if value was successfully decoded and FALSE otherwise. Data in the beginning of buffer will be overwritten. There is no danger, though, that we can lose any information: hexadecimal representation of byte takes 2 bytes in the buffer, so during decoding value will be twice smaller than before. See an example.

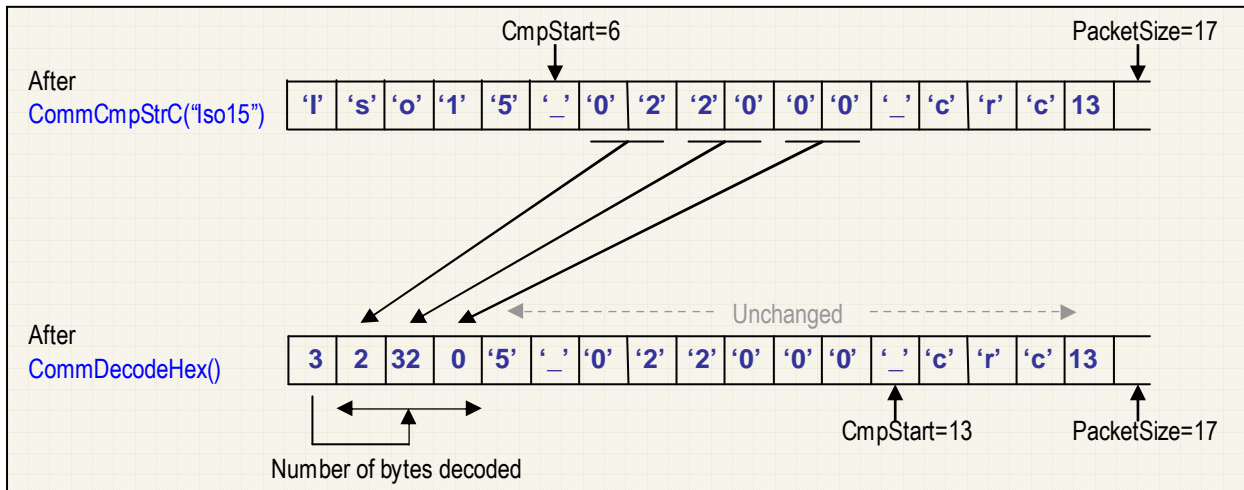


Figure 8.1.5. Hexadecimal data decoding

For some more function and their parameters see header file 'Comm.h'.

## 9.2. How to make own parser and extend functionality

File 'Parser.c' contains a function 'UserParser' which interprets user commands. This function must return TRUE if command was interpreted and ELSE otherwise. Let's add a command called 'MyCommand'.

```

#include <avr/pgmspace.h>
#include "Defs.h"
#include "Comm.h"

void Isol5 ( void );
void Isol4B ( void );
void MyCommand ( void );

BYTE UserParser ( void )
{
    BYTE BResult;

    BResult=TRUE;

    if (CommCmpStrC("Iso15")) Isol5();
    else if (CommCmpStrC("Iso14B")) Isol4B();
    else if (CommCmpStrC("MyCommand")) MyCommand();
    else BResult=FALSE;

    return BResult;
}

```

Let's imagine that MyCommand is supposed to have hexadecimal data and 2 optional parameters: Param1 and Param2. It has the following format

MyCommand <Hexadecimal data> [Param1] [Param2]

Function can look like following

```
void MyCommand ( void )
{
    BYTE BParam1,
        BParam2;
    BYTE BResult;

    BParam1=FALSE;
    BParam2=FALSE;
    BResult=FALSE;

    do
    {
        if (CommDecodeHex()==FALSE)
        {
            CommSendStrC13("Invalid HEX value [\1]");
            break;
        }

        do
        {
            if (CommCmpStrC("")) { BResult=TRUE; break; }
            else if (CommCmpStrC("Param1")) { BParam1=TRUE; }
            else if (CommCmpStrC("Param2")) { BParam2=TRUE; }
            else { CommSendStrC13("Error: Invalid command [\1]"); break; }
        } while(1);

        if (BResult==FALSE) break;

        <Execute command>

    } while(0);
}
```

## 10. Function description

Only the main interface functions are listed below. There are few minor functions used by the main interface functions but not describes in this document. Module Mlx90121.c  
 This module is a collection of functions to access Mlx90121 reader.

<b>Syntax</b>	
<code>void MlxWrReg ( BYTE AIndex, BYTE AData );</code>	
<b>Usage</b>	Writing MLX90121 configuration register.
<b>Parameters</b>	<i>AIndex</i> – register index <i>AData</i> – value to write
<b>Return value</b>	<b>none</b>

The following group of functions is for the users who desire to implement their own communication protocols. Otherwise the using of these functions is discouraged and users are advised to use protocol-specific functions (see chapter '5.1 Sending data, Receiving data, Inventory').

<b>Syntax</b>	
<code>void MlxSendStart ( BYTE ATimerFlags, BYTE ATimerCmp, BYTE ASymbol );</code>	
<b>Usage</b>	Starts transmission. This function prepares timer and sends the 1 <sup>st</sup> symbol
<b>Parameters</b>	<i>ATimerFlags</i> – timer configuration register <i>ATimerCmp</i> – timer compare value <i>ASymbol</i> – the 1 <sup>st</sup> symbol to send
<b>Return value</b>	<b>none</b>

<b>Syntax</b>	
<code>void MlxSendStop ( BYTE ATimerFlags, BYTE ATimerCmp );</code>	
<b>Usage</b>	Stop transmission. This function waits the end of transmission and prepares timer for the reception
<b>Parameters</b>	<i>ATimerFlags</i> – timer configuration register <i>ATimerCmp</i> – timer compare value
<b>Return value</b>	<b>none</b>

<b>Syntax</b>	
<code>BYTE MlxSyncStartASK ( BYTE ATimerFlags, BYTE ATimerCmp );</code>	
<b>Usage</b>	Start synchronization. Function polls DOUT during interval, specified by parameters ATimerFlags and ATimerCmp, tries to filter glitches on DOUT and starts the synchronization (see Majority-Voting description in the datasheet)
<b>Parameters</b>	<i>ATimerFlags</i> – timer configuration register <i>ATimerCmp</i> – timer compare value
<b>Return value</b>	<b>TRUE</b> – if synchronization was started successfully <b>FALSE</b> – otherwise

<b>Syntax</b>	
<code>void MlxSyncStop ( void );</code>	
<b>Usage</b>	Stop synchronization
<b>Parameters</b>	<b>none</b>
<b>Return value</b>	<b>none</b>

### 10.1. Module Crt.c

This module is related to the microcontroller.

<b>Syntax</b>	
<code>void TimerSet ( BYTE ATimerFlags, BYTE ATimerCmp );</code>	
<b>Usage</b>	Timer configuration and timer start
<b>Parameters</b>	<i>ATimerFlags</i> – timer configuration register <i>ATimerCmp</i> – timer compare value
<b>Return value</b>	<b>none</b>

<b>Syntax</b>	
<code>void TimerStop ( void );</code>	
<b>Usage</b>	Timer stop
<b>Parameters</b>	<b>none</b>
<b>Return value</b>	<b>none</b>

<b>Syntax</b>	
<code>BYTE IntDisable ( void );</code>	
<b>Usage</b>	Disable interrupts
<b>Parameters</b>	<b>None</b>
<b>Return value</b>	Old interrupt register to be restored when interrupts are enabled

<b>Syntax</b> <code>void IntRestore ( BYTE AIntSave );</code>	
<b>Usage</b>	Restore interrupts. There is no 'Interrupt enable' function. Application is discouraged to 'enable interrupts', because it cannot be sure that interrupts were disabled by previous function. When disabling interrupts, Application must save the old interrupt flags, and then restore interrupts
<b>Parameters</b>	<code>AIntSave</code> – old interrupt flags
<b>Return value</b>	None

<b>Syntax</b> <code>void WDRreset ( void );</code>	
<b>Usage</b>	Reset watchdog timer
<b>Parameters</b>	none
<b>Return value</b>	none

## 10.2. *Module Comm.c*

Contains communication functions and functions to facilitate parse.

<b>Syntax</b> <code>void CommInit ( void );</code>	
<b>Usage</b>	Initialize module Comm. This function MUST be called once at the beginning of the application
<b>Parameters</b>	none
<b>Return value</b>	none

<b>Syntax</b> <code>void CommDeletePacket ( void );</code>	
<b>Usage</b>	Deletes the packet from the UART reception buffer and analyzes data stored in the buffer after the end of the packet. This function is called at the end of parsing routine. Application should not call this function directly
<b>Parameters</b>	none
<b>Return value</b>	none

<b>Syntax</b>	
<code>void CommSendStrC ( CBytePtr AStr );</code>	
<b>Usage</b>	Sends a zero-terminated string by UART. String must be stored in microcontroller flash memory (constant)
<b>Parameters</b>	<b>AStr</b> – pointer to the zero-terminated string stored in flash
<b>Return value</b>	<b>none</b>

<b>Syntax</b>	
<code>void CommSendStrC13 ( CBytePtr AStr );</code>	
<b>Usage</b>	Sends a zero-terminated string by UART, then sends symbol CR (Caret Return, #13). String must be stored in microcontroller flash memory (constant)
<b>Parameters</b>	<b>AStr</b> – pointer to the zero-terminated string stored in flash
<b>Return value</b>	<b>None</b>

<b>Syntax</b>	
<code>void CommSendHex ( BYTE AData );</code>	
<b>Usage</b>	Sends hexadecimal byte by UART
<b>Parameters</b>	<b>AData</b> – byte to be sent
<b>Return value</b>	<b>none</b>

<b>Syntax</b>	
<code>void CommSendDec ( BYTE AData );</code>	
<b>Usage</b>	Sends decimal byte by UART
<b>Parameters</b>	<b>AData</b> – byte to be sent
<b>Return value</b>	<b>None</b>

<b>Syntax</b>	
<code>void CommSendBufHex ( BYTE *ABuf, BYTE ALen );</code>	
<b>Usage</b>	Sends data in the buffer by UART. Data is represented as hexadecimal
<b>Parameters</b>	<b>ABuf</b> – buffer where data is stored <b>ALen</b> – number of bytes to send
<b>Return value</b>	<b>None</b>

<b>Syntax</b>	
<b>BYTE</b> CommCmpStrC ( <b>CBytePtr</b> AStr );	
<b>Usage</b>	Compares data stored at the beginning of UART reception buffer with a zero-terminated string AStr. If comparison is correct, returns TRUE and deletes the first word from UART reception buffer
<b>Parameters</b>	AStr – zero-terminated string to compare. Must be stored in MCU flash memory (constant)
<b>Return value</b>	<b>TRUE</b> if the word at the beginning of the UART reception buffer matches with AStr <b>FALSE</b> – otherwise
<b>Remarks</b>	<ul style="list-style-type: none"> <li>• This function is case-insensitive</li> <li>• If function returns TRUE, the first word will be deleted from the buffer</li> </ul>

<b>Syntax</b>	
<b>BYTE</b> CommDecodeHex ( <b>void</b> );	
<b>Usage</b>	Checks and converts data stored at the beginning of UART reception buffer <b>FUartRecvBuf</b> from hexadecimal to binary and stores the result in the same buffer. If conversion is successful, FUartRecvBuf[0] will contain the length of data, FUartRecvBuf[1] and further will contain the result of conversion
<b>Parameters</b>	None
<b>Return value</b>	<b>TRUE</b> if conversion is successful <b>FALSE</b> – otherwise
<b>Remarks</b>	<ul style="list-style-type: none"> <li>• If function returns TRUE, the original data will be deleted from the buffer</li> </ul>

<b>Syntax</b>	
<b>BYTE</b> CommGetDataD ( <b>DWORD</b> *AData );	
<b>Usage</b>	Checks and converts data stored at the beginning of UART reception buffer <b>FUartRecvBuf</b> from decimal to binary and copies it to *AData
<b>Parameters</b>	*AData – pointer to 4-bytes variable where the result of conversion is copied
<b>Return value</b>	<b>TRUE</b> if conversion is successful <b>FALSE</b> – otherwise
<b>Remarks</b>	<ul style="list-style-type: none"> <li>• If function returns TRUE, the original data will be deleted from the buffer</li> </ul>

### 10.3. Modules *ErrorCodes.c* and *ErrorCodes.h*

Header file *ErrorCodes.h* contains the list of predefined error codes returned by TAG communication functions. *ErrorCodes.c* contains a function which returns a pointer to the readable error code explanation.

<b>Syntax</b>	
<code>BYTE *ErrGetCodeStrC ( BYTE AErrorCode );</code>	
<b>Usage</b>	Converts error code to its readable explanation
<b>Parameters</b>	<b>AErrorCode</b> – Error code
<b>Return value</b>	Pointer to a readable explanation of the error. Explanation string is zero-terminated and is stored in flash memory <b>Note:</b> This function may return NULL if error code is invalid or unknown
<b>Remarks</b>	Application must analyze NULL-reply from the function and handle it, for example, give “Unknown error” message

### 10.4. Communication module *Iso15.c*

Module contains 2 functions to send data to the TAG and receive data from the TAG using communication protocol ISO-15693.

<b>Syntax</b>	
<code>BYTE Iso15Send ( BYTE *ABuf, BYTE ASize, BYTE AAutoCrc );</code>	
<b>Usage</b>	Sends data to the TAG by ISO-15693 protocol
<b>Parameters</b>	<b>*ABuf</b> – pointer to a buffer where data to send is stored <b>ASize</b> – number of bytes to send <b>AAutoCrc</b> – forces function to calculate CRC and add it at the end
<b>Return value</b>	<b>ENoError</b> code if performed successfully Otherwise see error code explanation

<b>Syntax</b>	
<code>BYTE Iso15Recv ( BYTE *ABuf, BYTE ABufSize, BYTE *ABytesReceived, BYTE *AFlags );</code>	
<b>Usage</b>	Receives data from TAG by ISO-15693 protocol
<b>Parameters</b>	<b>*ABuf</b> – pointer to a buffer where data to send is stored <b>ABufSize</b> – length of the buffer <b>*ABytesReceived</b> – number of bytes actually received <b>*AFlags</b> – 0 if there is no collision
<b>Return value</b>	<b>EFrameRecvOK</b> code if data is received <b>EFrameWaitTOut</b> code if there is no response Otherwise see error code explanation

### 10.5. Communication module for inventory Iso15Inv.c

Module contains 2 functions to perform inventory for ISO-15693 protocol.

<b>Syntax</b>	
<code>void Iso15InvStart ( BYTE AMode );</code>	
<b>Usage</b>	Initializes the inventory routine. Must be called by the application to start inventory
<b>Parameters</b>	<b>AMode</b> – shows which communication mode to use: Dual or Single subcarrier
<b>Return value</b>	None

<b>Syntax</b>	
<code>BYTE Iso15InvGetNext ( BYTE *ABuffer );</code>	
<b>Usage</b>	Searches the next TAG for ISO-15693 Inventory
<b>Parameters</b>	<b>*ABuffer</b> – pointer where a response of the next found TAG is placed <b>note:</b> ABuffer must be at least 12 bytes long
<b>Return value</b>	<b>EFrameRecvOK</b> – when next TAG is found and its response is placed to the buffer <b>ENoTag</b> – when there is no more TAGs Otherwise see error code explanation
<b>Remarks</b>	<ul style="list-style-type: none"> <li>• A full TAG response is places into the buffer, not only UID. Usually, UID is 8 bytes starting with ABuffer[2]</li> <li>• Application must call this function in a loop until ENoTag (or other error code) is received</li> </ul>

### 10.6. Communication module Iso14B.c

Module contains 2 functions to send data to the TAG and receive data from the TAG using communication protocol ISO-14443-B.

<b>Syntax</b>	
<code>BYTE Iso14BSend ( BYTE *ABuf, BYTE ASize, BYTE AAutoCrc );</code>	
<b>Usage</b>	Sends data to the TAG by ISO-14443-B protocol
<b>Parameters</b>	<b>*ABuf</b> – pointer to a buffer where data to send is stored <b>ASize</b> – number of bytes to send <b>AAutoCrc</b> – forces function to calculate CRC and add it at the end
<b>Return value</b>	<b>ENoError</b> code if performed successfully Otherwise see error code explanation

**Syntax**  
**BYTE** Iso14BRecv ( **BYTE** \*ABuf, **BYTE** ABufSize, **BYTE** \*ABytesReceived );

**Usage**

Receives data from TAG by ISO-14443-B protocol

**Parameters**

\***ABuf** – pointer to a buffer where data to send is stored  
**ABufSize** – length of the buffer  
\***ABytesReceived** – number of bytes actually received

**Return value**

**EFrameRecvOK** code if data is received  
**EFrameWaitTOut** code if there is no response  
Otherwise see error code explanation

## **11. Conclusion**

The firmware described here will allow customers to quickly develop their application without the need to understand deeply the internal structure and behavior of the MLX90121. By this way, they can focus on the development of their applications and take benefit of the best performances of the Melexis RFID transceiver IC. Thanks to the firmware flexibility, developers can choose the protocol they want to address and adapt it to fit their applications and the minimum code size for their applications.