

# Tutorial

The art  
of Programming  
for  
OBID i-scan<sup>®</sup>  
and  
OBID<sup>®</sup> *classic-pro*

(based on SDK version up from 4.02.00)

## Note

© Copyright 2012 by  
FEIG ELECTRONIC GmbH  
Lange Straße 4  
D-35781 Weilburg-Waldhausen  
Germany  
<http://www.feig.de>  
[obid-support@feig.de](mailto:obid-support@feig.de)

The indications made in these mounting instructions may be altered without previous notice. With the edition of these instructions, all previous editions become void.

**Copying of this document, and giving it to others and the use or communication of the contents thereof, are forbidden without express authority. Offenders are liable to the payment of damages. All rights are reserved in the event of the grant of a patent or the registration of a utility model or design.**

Composition of the information given in these mounting instructions has been done to the best of our knowledge. FEIG ELECTRONIC GmbH does not guarantee the correctness and completeness of the details given and may not be held liable for damages ensuing from incorrect installation.

Since, despite all our efforts, errors may not be completely avoided, we are always grateful for your useful tips.

FEIG ELECTRONIC GmbH assumes no responsibility for the use of any information contained in this manual and makes no representation that they are free of patent infringement. FEIG ELECTRONIC GmbH does not convey any license under its patent rights nor the rights of others.

The installation-information recommended here relates to ideal outside conditions. FEIG ELECTRONIC GmbH does not guarantee the failure-free function of the OBID®-system in outside environment.

OBID® and OBID i-scan® are registered trademarks of FEIG ELECTRONIC GmbH.

Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

Windows Vista is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries

Oracle and Java® are registered trademarks of Oracle Corporation.

Linux® is a registered Trademark of Linus Torvalds.

Apple, Mac, Mac OS, OS X, Cocoa and Xcode are trademarks of Apple Inc., registered in the U.S. and other countries.

Electronic Product Code (TM) is a Trademark of EPCglobal Inc.

I-CODE® and Mifare® are registered Trademarks of Philips Electronics N.V.

Tag-it (TM) is a registered Trademark of Texas Instruments Inc.

Jewel (TM) is a trademark of Innovision Research & Technology plc.

---

**Content:**

<b>1. Introduction</b>	<b>7</b>
<b>1.1. Standard Software Tools</b>	<b>8</b>
1.1.1. The applications ISOStart and CPRStart	8
1.1.2. Firmware-Update	10
<b>2. Overview of all OBID® Reader families</b>	<b>11</b>
<b>2.1. Working Modes</b>	<b>11</b>
<b>2.2. Communication Interfaces</b>	<b>11</b>
<b>2.3. Transmission Protocol</b>	<b>12</b>
<b>2.4. Reader-API</b>	<b>12</b>
<b>2.5. Transponder-API</b>	<b>12</b>
<b>2.6. Miscellaneous</b>	<b>13</b>
<b>3. Options for integration</b>	<b>14</b>
<b>3.1. Supported Operating Systems</b>	<b>15</b>
<b>3.2. FEIG Kernel Driver for Windows</b>	<b>15</b>
3.2.1. Standard-Driver for USB-Reader	15
3.2.2. PC/SC-Driver for OBID® <i>classic-pro</i> Reader	15
<b>3.3. FEIG-Libraries</b>	<b>16</b>
3.3.1. Function Libraries for the transport layer	16
3.3.2. Function Library for the protocol layer	16
3.3.3. Function Library for external Function Units (Multiplexer, Antenna Tuner)	17
3.3.4. Function Library for T=CL based APDU handling	17
3.3.5. Class Libraries (C++, Java, C#)	17
<b>3.4. Custom-Applications in the Reader</b>	<b>19</b>
<b>4. Overview of all Libraries</b>	<b>20</b>
<b>4.1. Function Libraries</b>	<b>20</b>
4.1.1. FECOM	20
4.1.2. FEUSB	21
4.1.3. FETCP	22
4.1.4. FEISC	23
4.1.5. FETCL	25
4.1.6. FEFU	26
<b>4.2. Class Libraries</b>	<b>27</b>
4.2.1. FEDM	27
4.2.2. OBIDISC4J and OBIDISC4NET	29
<b>4.3. Thread security</b>	<b>30</b>
<b>5. Error Handling</b>	<b>31</b>
<b>6. General preliminary notes to the sections</b>	<b>32</b>
<b>7. Section 1: Basic initializations</b>	<b>33</b>
<b>8. Section 2: Establish a connection to the Reader</b>	<b>35</b>
<b>8.1. Serial Port (RS232 / RS485 / RS422)</b>	<b>35</b>
<b>8.2. Bluetooth</b>	<b>37</b>
<b>8.3. USB</b>	<b>39</b>
<b>8.4. TCP/IP (LAN and WLAN)</b>	<b>43</b>
<b>8.5. Excursion: Secured data transmission with encryption</b>	<b>44</b>

<b>8.6. Excursion: Error handling for TCP/IP communication .....</b>	<b>46</b>
8.6.1. Communication errors.....	46
8.6.2. Errors while establish a connection .....	47
8.6.3. Errors while closing the connection .....	48
8.6.4. Problem with broken communication link – the Keep-Alive option .....	48
<b>8.7. Excursion: Detecting Readers with different Protocol Frames in one App.....</b>	<b>49</b>
8.7.1. Detecting at serial port .....	50
8.7.2. Detecting at USB .....	52
<b>9. Section 3: Basic knowledge about how to use SendProtocol().....</b>	<b>53</b>
<b>10. Section 4: Read of important information from Reader .....</b>	<b>56</b>
10.1. Reader Information: The method ReadReaderInfo().....	56
10.2. Reader Diagnostic: The method ReadReaderDiagnostic().....	58
10.3. Reader Configuration: The method ReadCompleteConfiguration().....	60
<b>11. Section 5: Programming for the Host-Mode .....</b>	<b>62</b>
11.1. Inventory .....	62
11.2. Read / Write Transponder data.....	66
11.2.1. Normal addressed mode .....	66
11.2.2. Extended addressed mode .....	69
11.3. Excursion: Inventory with multiple antennas.....	72
11.4. The application ISOHostSample.....	75
<b>12. Section 6: Using TagHandler classes with Host-Mode.....</b>	<b>76</b>
12.1. Benefit.....	76
12.2. Inventory and Select .....	77
12.3. TagHandler classes.....	80
12.3.1. Life Cycle of TagHandler objects.....	80
12.3.2. Naming Conventions.....	80
12.3.3. Base class FedmlscTagHandler.....	81
12.3.4. Excursion: Class FedmlscTagHandler_ISO15693.....	82
12.3.5. Excursion: Class FedmlscTagHandler_EPC_Class1_Gen2.....	83
12.3.6. Excursion: Classes FedmlscTagHandler_ISO14443.....	84
12.3.7. Excursion: Class FedmlscTagHandler_ISO14443_4_MIFARE_DESFire .....	85
12.4. The application TagHandlerSample .....	86
<b>13. Section 7: APDU Handling with ISO 14443-4 compliant Tags.....</b>	<b>87</b>
<b>14. Section 8: Programming for the Buffered-Read-Mode .....</b>	<b>88</b>
14.1. Method of operation .....	88
14.2. Programming the query cycle .....	88
14.3. Structure of the received data .....	88
14.4. Adjust the method of operation.....	89
14.4.1. Excursion: Filter .....	89
14.4.2. Excursion: Adjust the structure of the data record .....	89
14.4.3. Excursion: Triggered Mode .....	89
14.4.4. Excursion: Automatic activation of outputs .....	89
14.4.5. Excursion: Writing data to the Transponder .....	89
14.5. The application BRMSample .....	90
<b>15. Section 9: Programming for the Notification-Mode .....</b>	<b>91</b>
15.1. Method of operation .....	91
15.2. Register an event .....	92

<b>15.3. Event handling.....</b>	<b>94</b>
<b>15.4. Cancel asynchronous task .....</b>	<b>95</b>
<b>15.5. Adjust the method of operation.....</b>	<b>95</b>
15.5.1. Excursion: Writing data to the Transponder .....	96
<b>15.6. Considerations for fail-safe operation .....</b>	<b>97</b>
15.6.1. Keep-Alive option for detecting broken network link .....	97
15.6.2. Avoiding deadlock situations .....	97
<b>15.7. The application NotifySample.....</b>	<b>98</b>
<b>16. Section 10: Programming for the Scan-Mode .....</b>	<b>99</b>
16.1. Method of operation .....	99
16.2. Select the output interface.....	99
16.3. Register an event .....	99
16.4. Event handling.....	99
16.5. Adjust the method of operation.....	99
16.5.1. Excursion: Setting the data format .....	99
16.5.2. Excursion: HID (Human-Interface-Device) .....	99
16.6. The application ScanSample .....	99
<b>17. Section 11: Management of the Reader configuration .....</b>	<b>100</b>
17.1. Persistence of the Reader configuration .....	100
17.1.1. Physical (old) structure.....	100
17.1.2. Logical (new) structure .....	100
17.2. Read/Modify/Write of the Reader configuration .....	100
17.3. Serialization of the Reader configuration into an XML file .....	100
<b>18. Section 12: Activation of outputs .....</b>	<b>101</b>
<b>19. Section 13: Reading states of digital inputs.....</b>	<b>102</b>
<b>20. Section 14: Reset methods .....</b>	<b>103</b>
<b>21. Section 15: Classes for external Function Units.....</b>	<b>104</b>
21.1. Multiplexer (HF and UHF).....	104
21.2. Automatic Antenna Tuner (HF).....	104
21.3. The Application DATuningTool .....	104
21.4. People-Counter in HF-Gate-Antennas.....	104

## Notes concerning this tutorial

This tutorial describes software libraries which are also described in detail in manuals or document files. For this reason we have limited the documentation to what is absolutely necessary for understanding the functionality and use of the libraries. It is assumed that the reader of this tutorial reads the system manual of the used OBID® RFID-Reader and the OBID®-Library manuals too.

FEIG ELECTRONIC GmbH does not repeat the same information about OBID® RFID-Readers in different manuals or use cross-references to certain pages in a different document. This is necessary due to the constant updating of manuals, and it prevents confusion caused by information in out-of-date documents. The reader of this tutorial is therefore well advised to check regularly that he has the latest manuals. If not, these can of course be obtained whenever needed from FEIG ELECTRONIC GmbH.

Preliminary

---

## 1. Introduction

---

FEIG ELECTRONIC GmbH has developed different, hierarchical structured software libraries to simplify the integration of OBID® RFID-Readers into customers applications.

A common attribute of all components is the support of all OBID®-Reader families with a uniform Application Programming Interface (API).

Class libraries for the most popular programming languages C++, Java and C# (VB.NET) represent the highest level in the software stack and will get the main focus of this Tutorial. Reader- and Transponder management and the serialization of the Reader configuration are some highlights of the class libraries.

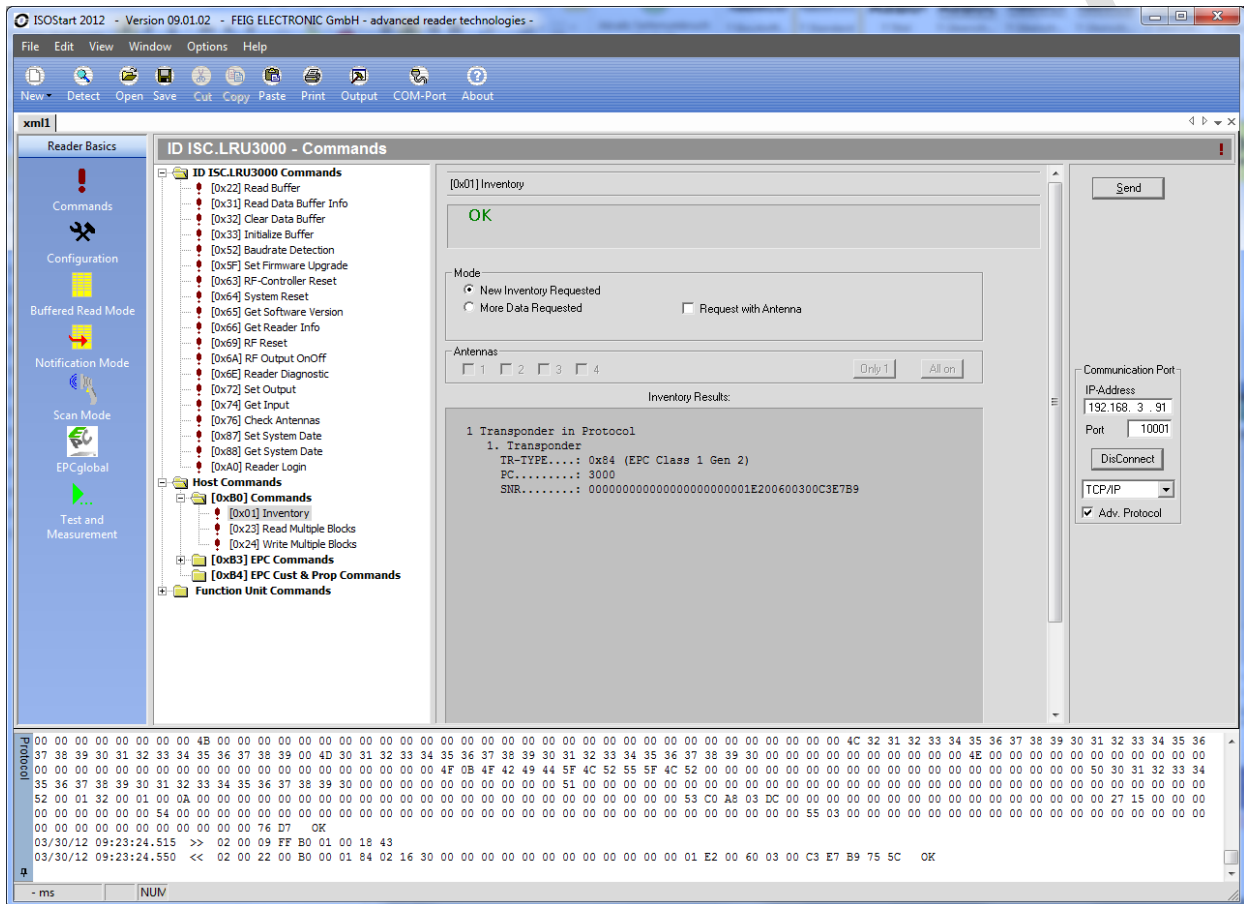
The class libraries are based upon function libraries, realizing the transport and protocol layers. These function libraries have a C interface and can be used with different programming languages.

The main intention of this tutorial is to give application programmers a structured introduction to the APIs for OBID® RFID-Readers with the help of hands-on examples. The executable samples included in the SDKs offer an advanced stadium.

## 1.1. Standard Software Tools

### 1.1.1. The applications ISOStart and CPRStart

The demo programs **ISOStart** and **CPRStart** have been developed to familiarize you with the functionality of the OBID® RFID-Readers. Also, these tools can be used as reference applications to test intended interactions with RFID-Readers and RFID-Transponders or to compare with the results of your application.



Using this software you can:

- Test the communication with RFID-Transponders.
- Read out and modify the configuration of OBID® RFID-Readers.
- Communication with Function Units like Multiplexer or Dynamic Antenna Tuner
- Activate a Firmware Upgrade

With each action the transmission protocols between PC and Reader are displayed on the screen. This transparency guides you to the software interface of the OBID® RFID-Readers. The respective system manuals should be referred for interpreting the protocols and for studying the reader properties.

Unique features are:



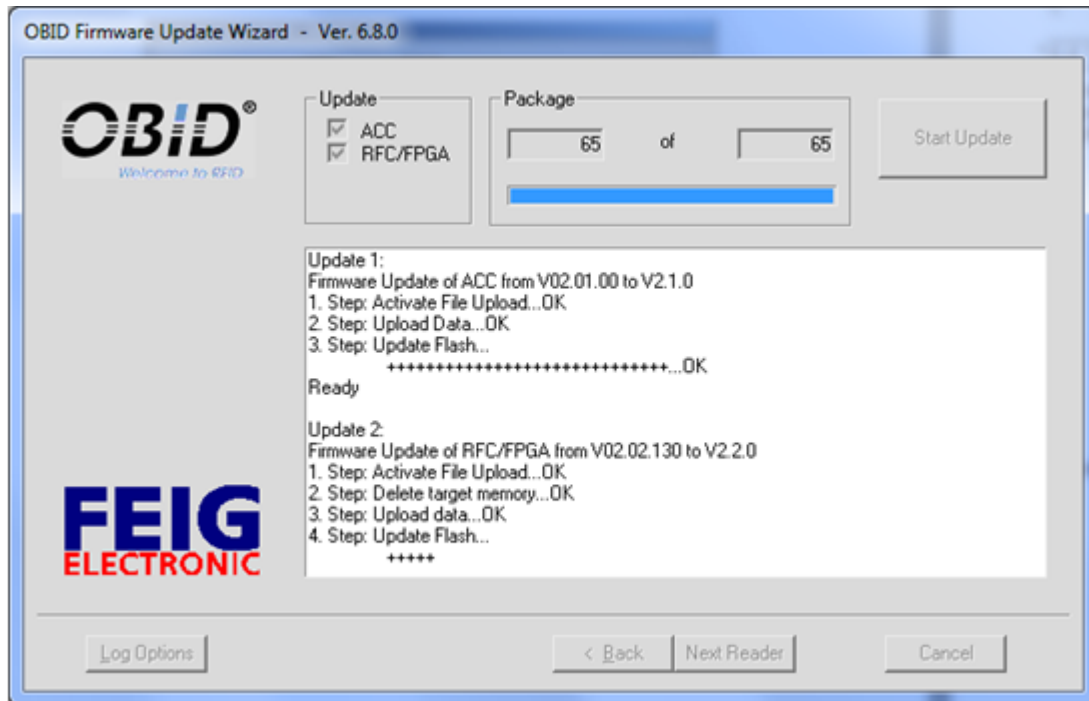
- Reader Editor for editing the OBID® RFID-Reader parameters. You can open any number of reader files and “link” them with various interface types.
- Protocol Editor for manual protocol entry and editing.
- Protocol Window for visualizing the communication.
- Test of the automatic reader modes like: Buffered Read Mode, Scan Mode, Notification Mode.

Preliminary

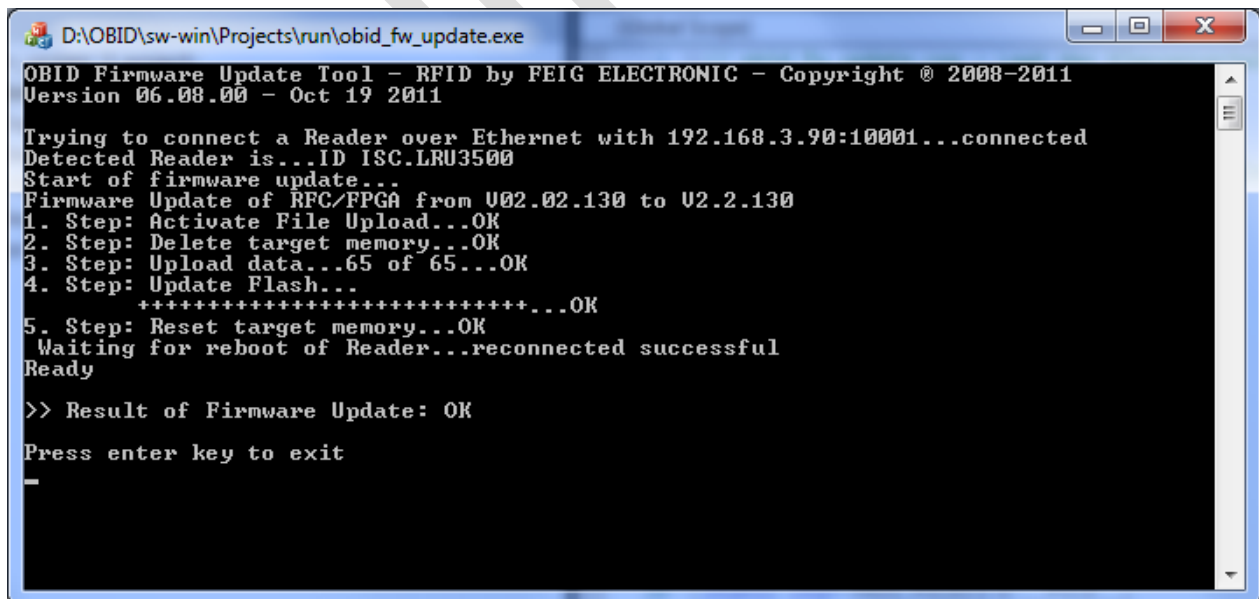
### 1.1.2. Firmware-Update

The application OBIDFirmwareUpdateTool can be used to perform a firmware update with an OBID® RFID-Reader. FEIG offers two versions:

One tool with user interface (only for Windows).



And one tool for the console (Windows, Windows CE and Linux).



---

## 2. Overview of all OBID® Reader families

---

FEIG ELECTRONIC develops and manufactures RFID-Readers for different target applications. Over the years and mainly in intense discussions with customers, the continuous engineering results in three Reader families with familiar command interfaces.

OBID® <i>classic-pro</i>	13,56 MHz Reader for ISO14443 compliant Transponders
OBID i-scan® HF	13,56 MHz Reader for ISO15693 compliant Transponders
OBID i-scan® UHF	860...960 MHz Reader for EPC compliant Transponders

The similarities refer to:

- Working modes
- Communication interfaces
- Organisation of the configuration
- Reader-API: commands for controlling and configuring the Reader
- Transponder-API: standardized, transparent and Custom-specific Transponder commands
- Function-Unit-API: commands for external Function Units

---

### 2.1. Working Modes

---

Every OBID® RFID-Reader supports at least the Host-Mode, which is a polling mode.

Many Readers support additionally autonomous read modes. These modes have an increased read performance, as no host communication is necessary. The read data items are either collected in an internal table (Buffered-Read-Mode or Notification-Mode) or transferred directly over the serial port or USB (Scan-Mode).

The autonomous read modes can be configured for a wide range of applications. Detailed information can be found in the system manuals of the Readers.

---

### 2.2. Communication Interfaces

---

OBID® RFID-Readers have one or up to four communication interfaces. Besides the serial port (TTL-Level, RS232, RS485, RS422), USB or Bluetooth or LAN or WLAN can be implemented too.

Independently of the physical communication interface, the communication protocol is always identical.

## 2.3. Transmission Protocol

All OBID® RFID-Readers have an identical, binary transmission protocol with frame and checksum. The following pictured send and receive protocol are included in actual almost all Readers and are the standard frame for future Readers. The old frame with only one length byte, which is currently present in many Readers, will be no longer supported with future Readers. We would therefore recommend that new applications should use only the Advanced Protocol Frame.

Reader ← Host

1	2	3	4	5	(6...n-2)
STX (0x02)	MSB ALENGTH	LSB ALENGTH	COM-ADR	CONTROL- BYTE	(DATA)

n-1	n
LSB CRC16	MSB CRC16

Host ← Reader

1	2	3	4	5	6	(7...n-2)
STX (0x02)	MSB ALENGTH	LSB ALENGTH	COM-ADR	CONTROL- BYTE	<b>STATUS</b>	(DATA)

n-1	n
LSB CRC16	MSB CRC16

One of the most significant element of the receive protocol is the **Status byte**. Application programmers have to analyze this **Status** after each communication.

## 2.4. Reader-API

All OBID® RFID-Readers support a subset of commands for controlling and configuring the Reader.

Included are read/write/reset of the configuration, read of Reader information, reset commands, RF on/off and mostly diagnostic commands and commands for digital I/O, relays and LEDs.

## 2.5. Transponder-API

The Transponder API is ordered in group of commands and the usage depends on the working mode.

With Host-Mode, these are the obligatory ISO 15693 and ISO 14443 commands. Some Readers support additionally optional RF transparent commands, with which the manufacturer specific tag commands are forwarded by the Reader without modifications. OBID® *classic-pro* Readers support

APDUs via the T=CL protocol with ISO 14443-4 compliant Transponders and have SAM-commands included.

With Buffered-Read-Mode and Notification-Mode a special command set transports the Transponder data from the Reader.

When the Scan-Mode is activated, no command set is required, because the Transponder data is transferred at once as a configurable binary data stream over the selected communication interface.

---

## 2.6. Miscellaneous

---

Some OBID i-scan® HF and OBID i-scan® UHF Reader can communicate with Function Units (Antenna Multiplexer, Antenna Tuner), which are wired into the antenna cable. The provided API for these devices is realized with special RF-Transparent commands.

### 3. Options for integration

Every software project has its own requirements. Similarly, every programmer has its own experience and preferences. FEIG ELECTRONIC responded with flexible, hierarchical libraries, and supports multiple operating systems and programming languages. High-level class libraries for C++, .NET and Java on top of the library stack offers the most comfortable API (s. Fig. 1). On the other side, integration on the lowest level – the transport layer – can be realized without any help of FEIG libraries, because every command is well documented in the system manual of the particular RFID-Reader.

In the following, we will discuss the pros and cons of the various integration options.

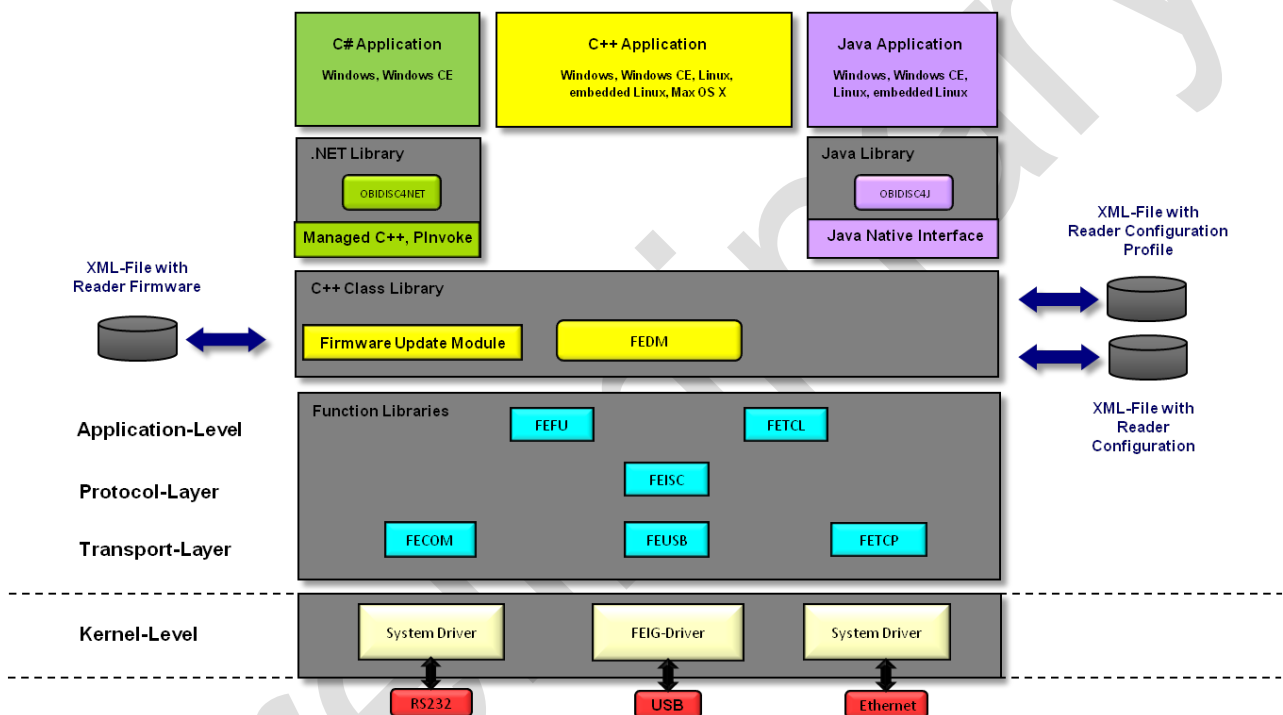


Fig. 1: Library stack from which the particular SDKs are built

As an alternative to the integration with the help of the native libraries, FEIG offers software components for middleware on demand.

### 3.1. Supported Operating Systems

All basic C and C++ libraries are available for different operating systems. Actually, the following OS are supported:

Operating System	Target		Notes
	32-Bit	64-Bit	
Windows XP	X	(X)	with 64-Bit OS: only with 32-Bit Runtime Environment
Windows Vista / 7 / 8	X	X	
Windows CE	X	-	V4.2 or higher
Linux	X	(X)	with 64-Bit OS: only with 32-Bit Runtime Environment
Apple Max OS X	-	X	OS X V10.7.3 or higher Architecture x86_64

With upcoming new OS or on request, FEIG is disposed to adapt the libraries for the requested OS.

Platform independent languages like Java or C# (VB.NET) are supported with special SDKs. These SDKs are compiled only for 32-Bit Frameworks, but can be used with 32-Bit Runtime Environments in 64-Bit OS. Native 64-Bit libraries for Java or .NET are actually not available.

### 3.2. FEIG Kernel Driver for Windows

#### 3.2.1. Standard-Driver for USB-Reader

For driving USB-Readers with Windows and Windows CE, a special kernel driver is required. This WHQL certified driver can be downloaded from the homepage of FEIG.

#### 3.2.2. PC/SC-Driver for OBID® *classic-pro* Reader

PC/SC is a standard for integrating smart cards and smart card readers. With Windows and Windows CE, PC/SC is realized as the Smart Card Library and part of the operating system. FEIG provides a PC/SC kernel driver for some OBID® *classic-pro* Readers for Windows XP, Vista, 7 and 8 and for Windows CE on demand.

Drivers for Linux and Apple's Mac OS X are actually not provided.

---

### 3.3. FEIG-Libraries ---

---

#### 3.3.1. Function Libraries for the transport layer ---

The transport layer is the first and lowest library layer in the library stack. For each communication port type a specialized function library with a C interface is provided: **FECOM** for the serial port and Bluetooth<sup>1</sup>, **FEUSB** for USB and **FETCP** for TCP/IP (IPv4) communication over LAN or WLAN. The main task of these libraries is to manage the transport of data in cooperation with the port drivers of the operating system. The C interface makes these libraries compatible with the most important programming languages and development systems.

Programmers selecting this integration layer have to implement the protocol handling (build and split of frames, CRC check and check of frame length) in their application. This creates considerable programming effort, and it is to be considered whether the entry is imperative at this level.

.NET and Java applications have no access to this layer.

Based on these libraries examples are not provided in the context of this tutorial.

---

#### 3.3.2. Function Library for the protocol layer ---

The protocol layer is one layer above the transport layer and is realized with the function library **FEISC**. The main task is to implement the protocol handling (build and split of frames, CRC check and check of frame length) in cooperation with the libraries in the transport layer. The C interface makes this library compatible with the most important programming languages and development systems.

The libraries of the transport layer are bound dynamically at runtime. A Plug-in mechanism is provided to support vendor specific port types.

Programmers selecting this integration layer (usually with Pascal, Delphi, VB6) can focus their work to the basic communication tasks. May be the writing of standard Reader control commands is more easily, the handling of Transponder commands or especially the programming for the working modes Buffered-Read-Mode or Notification-Mode is more complex and it is to be considered whether the entry is imperative at this level.

.NET and Java applications have no access to this layer.

Based on this library only few examples are provided in the context of this tutorial.

---

<sup>1</sup> Connected with a virtual, serial port based on the Bluetooth stack with the Serial Port Profile (SPP)



### 3.3.3. Function Library for external Function Units (Multiplexer, Antenna Tuner)

---

Support for Function Units which are wired into the antenna cable is provided with the library **FEFU** and depends on FEISC. The C interface makes this library compatible with the most important programming languages and development systems.

Programmers selecting this integration layer (usually with Pascal, Delphi, VB6) can focus their work to the basic communication tasks.

C++ programmers have the free choice to use this library or a specialized C++ class from the class library FEDM, because there is no elementary difference concerning the programming effort.

.NET and Java applications have no direct access to this layer. Instead of that, the control of the Function Units is realized with a specialized class.

---

### 3.3.4. Function Library for T=CL based APDU handling

---

APDU (Application Protocol Data Unit) centric applications find support with the library **FETCL** which depends on FEISC. APDUs are transmissioned on the RF field with T=CL protocols and can be applied with ISO 14443-4 compliant RFID Transponders. The C interface makes this library compatible with the most important programming languages and development systems.

C++ programmers have the free choice to use this library or specialized TagHandler classes from the C++ class library FEDM.

.NET and Java applications have no direct access to this layer. Instead of that, other interfaces are provided like TagHandler classes or an extension in the main reader class.

---

### 3.3.5. Class Libraries (C++, Java, C#)

---

The class libraries for C++ (**FEDM**), Java (**OBDISC4J**) and .NET (**OBIDISC4NET**) represent the highest level in the software stack and support all OBID®-Reader families. Java and .NET libraries are realized as a small wrapper layer above the C++ library FEDM and have almost the same API.

The C++ Class Library **FEDM** is the introduction of an organizational principle for all OBID® Reader families which allows you to create similar program structures for all OBID® Readers regardless of the reader you are using. The libraries for Java and .NET adopt this architecture. These libraries provide the first time persistence of data (e. g. Reader's configuration data as well as data from Transponders).

In spite of the uniform organizational principle, the view of the storable reader and transponder data is still at a very low level. This means that as a programmer you are confronted with reader parameters in bits and bytes and are offered transponder data only in the form of unorganized data quantities. The advantage of this is that you have access to everything, but on the other hand you have to carry out multiple operations in sequence if you want for example to write just a small amount of data to a transponder. Additional simplification with respect to abstraction of data streams and actions remains reserved for a higher-order module layer.

High-level methods in the reader class simplify the reader communication (e. g. ReadReaderInfo, ReadReaderDiagnostic, ReadCompleteConfiguration, TagInventory, TagSelect, etc.) while the

concept of *TagHandler* classes provides an efficient programming model for Transponder communication in the Host-Mode with a collection of proxy classes for a wide range of Transponder standards (ISO 14443, ISO 15693, EPC Class 1 Gen 2) as well as specific Chip types. Each *TagHandler* class offers a specific API for the identified Transponder type.

OBID *i-scan*® and OBID® *classic-pro* Reader have included Transponder commands which can transport data of multiple Transponders (Host-Mode, Buffered-Read-Mode, Notification-Mode) with one command and require the storage in tables in a structured form. The class libraries support this requirement with table classes and a query interface.

The class libraries offer a simple way of serializing data for the reader configuration. This makes it possible to store a complete reader configuration in an XML file, load it again later and transfer it to the reader.

### 3.4. Custom-Applications in the Reader

Some OBID *i-scan*® Reader have included a processor module, called Application and Connectivity Controller (ACC), to support onboard custom specific applications. The operating system is embedded Linux. Typically C++ is used as the preferred and most performant programming language and GCC as compiler. FEIG offers special Software Development Kits (SDKs) with a complete Toolchain.

The special thing about this SDK is the containment of almost all previous discussed and cross-compiled FEIG libraries so that the application programming inside the Reader differs not from outside the Reader.

The figure below (Fig. 2) pictures the principal internal software architecture for the OBID *i-scan*® UHF-Reader ID ISC.LRU3000, which is identical for all other Readers with an ACC onboard.

Writing embedded applications is ambitious and developers should be familiar with Linux and with the Gnu Compiler Collection (GCC). In view of the limited system resources (memory, no hard disk) the design of an embedded application should be more static to prevent increasing memory consumption during runtime.

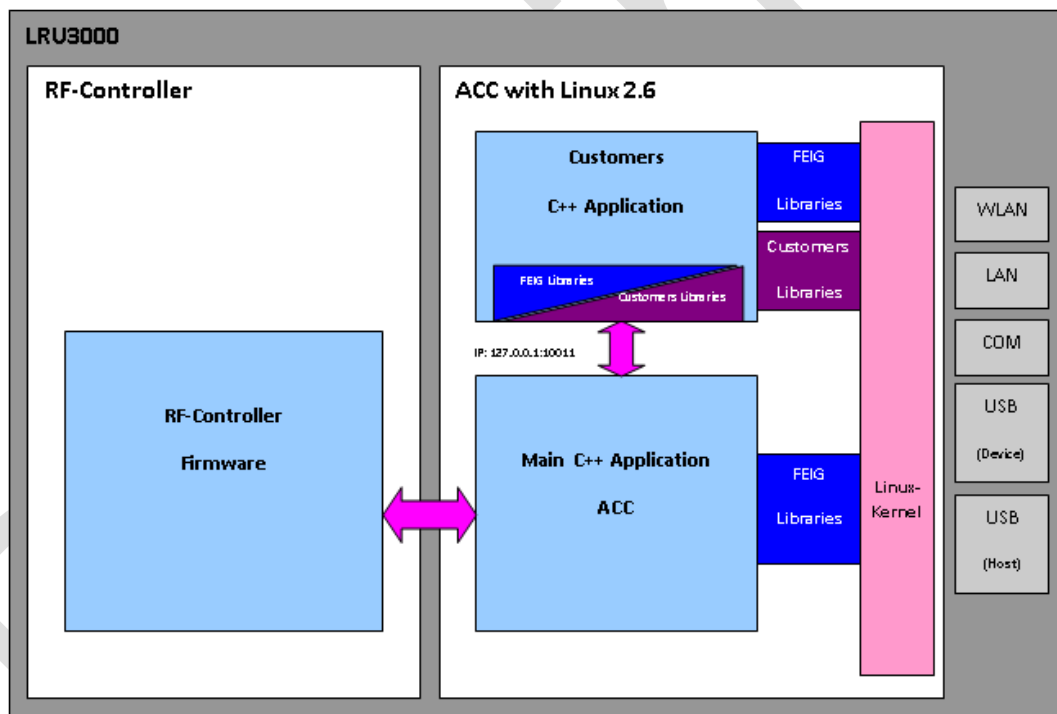


Fig 2: ACC Software Architecture

## 4. Overview of all Libraries

The following overview is for a quick introduction. A detailed description of each library can be found in the particular document.

### 4.1. Function Libraries

#### 4.1.1. FECOM



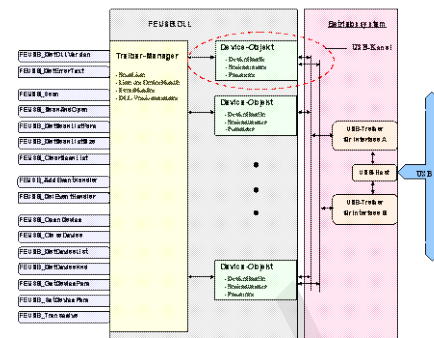
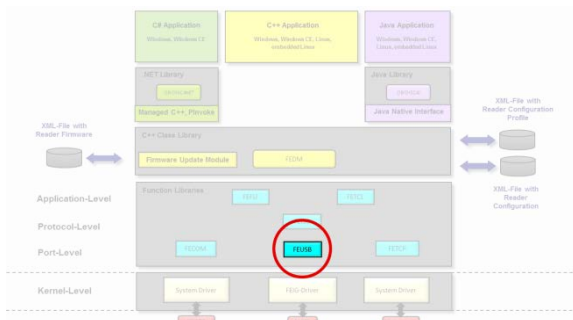
The transport layer library **FECOM** (= **FEIG COMM**unication) encapsulates all the functions and parameters which the user needs in order to manage one or more serial ports open at the same time. These ports are independent from each other and can be used simultaneously. Event handling mechanisms can be installed individually for each control lines of any opened port (e.g. CTS).

The API is identical for all Operating Systems and contains functions for managing of Serial Ports, communication functions and some more helper functions.

A selection of the important functions:

```
int FECOM_OpenPort( char* cPortNr )
int FECOM_ClosePort( int iPortHnd )
int FECOM_GetPortPara( int iPortHnd, char* cPara, char* cValue )
int FECOM_SetPortPara( int iPortHnd, char* cPara, char* cValue )
int FECOM_GetErrorText( int iErrorCode, char* cErrorText )
int FECOM_Transceive( int iPortHnd, unsigned char* cSendProt, int iSendLen, unsigned char* cRecProt, int iRecLen )
int FECOM_Transmit( int iPortHnd, unsigned char* cSendProt, int iSendLen )
int FECOM_Receive( int iPortHnd, unsigned char* cRecProt, int iRecLen )
```

### 4.1.2. FEUSB



The transport layer library **FEUSB** (= **FEIG USB**) manages multiple connections to USB-Readers. These connections are independent from each other and can be used simultaneously. An event handling mechanisms for plug'n play can be installed.

The API is identical for all Operating Systems and contains functions for managing of USB connections, communication functions and some more helper functions.

The first step in establishing a connection with an USB-Reader is to detect (scan procedure) one or all USB-Readers on the USB of the PC. Each found device is entered in the internal scan list but not opened. Before the first communication a USB-Reader must be selected from the scan list and the function `FEUSB_OpenDevice` is used to open a channel to this Reader.

A selection of the important functions:

```
int FEUSB_Scan( int iScanOpt, FEUSB_SCANSEARCH* pSearchOpt )
int FEUSB_ScanAndOpen( int iScanOpt, FEUSB_SCANSEARCH* pSearchOpt )
int FEUSB_OpenDevice( long nDeviceID )
int FEUSB_CloseDevice( int iDevHnd )
int FEUSB_GetDevicePara( int iDevHnd, char* cPara, char* cValue )
int FEUSB_SetDevicePara( int iDevHnd, char* cPara, char* cValue )
int FEUSB_GetLastError( int iDevHnd, int* iErrorCode, char* cErrorText )
int FEUSB_AddEventHandler( int iDevHnd, FEUSB_EVENT_INIT* pInit )
int FEUSB_DelEventHandler( int iPortHnd, FEUSB_EVENT_INIT* pInit )
int FEUSB_Transceive( int iDevHnd, char* cInterface, int iDir, unsigned char* cSendData, int iSendLen, unsigned* cRecData, int iRecLen )
int FEUSB_Transmit( int iDevHnd, char* cInterface, unsigned char* cSendData, int iSendLen )
int FEUSB_Receive( int iDevHnd, char* cInterface, unsigned char* cRecData, int iRecLen )
```

### 4.1.3. FETCP



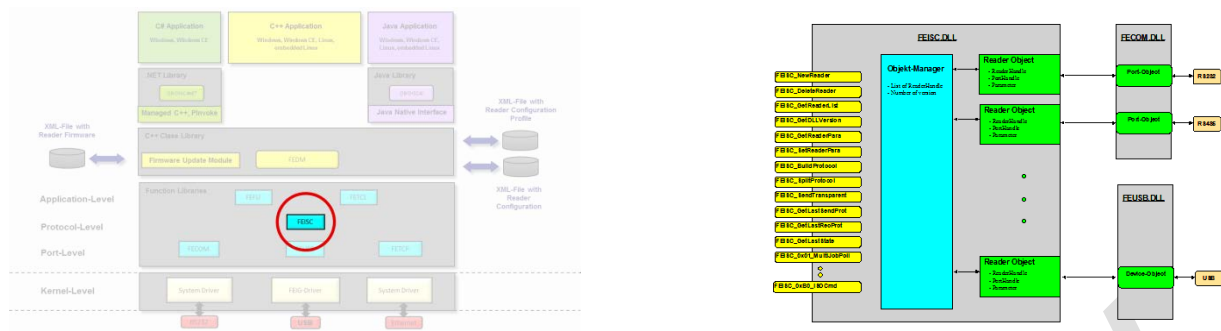
The transport layer library **FETCP** (= **FEIG TCP**) manages multiple TCP/IP (IPv4) connections over LAN or WLAN. These connections are independent from each other and can be used simultaneously.

The API is identical for all Operating Systems and contains functions for managing of TCP/IP connections, communication functions and some more helper functions.

A selection of the important functions:

```
int FETCP_Connect( char* cHostAdr, int iPortNr )
int FETCP_DisConnect( int iSocketHnd )
int FETCP_GetSocketPara( int iPortHnd, char* cPara, char* cValue )
int FETCP_SetSocketPara( int iPortHnd, char* cPara, char* cValue )
int FETCP_GetErrorText( int iErrorCode, char* cErrorText )
int FETCP_Transceive( int iPortHnd, unsigned char* cSendProt, int iSendLen, unsigned char* cRecProt, int iRecLen )
int FETCP_Transmit( int iPortHnd, unsigned char* cSendProt, int iSendLen )
int FETCP_Receive( int iPortHnd, unsigned char* cRecProt, int iRecLen )
```

#### 4.1.4. FEISC



The library **FEISC** (= **FEIG** OBID **i-scan**<sup>®</sup>) is part of the second level of a hierarchical structured, multi-tier FEIG library stack. It is only designed for executing Reader commands over the low-level protocol layer (build/split of frames, check of CRC, check of frame length). Together with transport layer libraries FECOM, FETCP and FEUSB, this makes it possible to run all the protocols in the system manual of the OBID **i-scan**<sup>®</sup> - or OBID<sup>®</sup> *classic-pro* Reader Family directly by invoking a function.

Multiple Readers can be handled independent from each other and can be used simultaneously.

Optional, the library support encrypted data transmission over Ethernet (TCP/IP), if this feature is implemented in the Reader firmware.

For analyzing or visualizing the protocol exchange, the library has implemented a logging mechanism for transferring protocol strings into a file or sending the protocol string to an application.

The API is identical for all Operating Systems and contains functions for managing of Reader objects, communication functions and some more helper functions.

A selection of the important functions:

int FEISC_NewReader( int iPortHnd )
int FEISC_DeleteReader( int iReaderHnd )
int FEISC_GetReaderList( int iNext )
int FEISC_GetReaderPara( int iReaderHnd, char* cPara, char* cValue )
int FEISC_SetReaderPara( int iReaderHnd, char* cPara, char* cValue )
int FEISC_GetErrorText( int iErrorCode, char* cErrorText )
int FEISC_GetStatusText( unsigned char ucStatus, char* cStatusText )
int FEISC_AddEventHandler( int iReaderHnd, FEISC_EVENT_INIT* pInit )
int FEISC_DelEventHandler( int iReaderHnd, FEISC_EVENT_INIT* pInit )
int FEISC_StartAsyncTask( int iReaderHnd, int iTaskID, FEISC_TASK_INIT* pInit, void* pInput )
int FEISC_CancelAsyncTask( int iReaderHnd )
int FEISC_0x63_CPUReset( int iReaderHnd, unsigned char cBusAdr )
int FEISC_0x66_ReaderInfo( int iReaderHnd, unsigned char cBusAdr, unsigned char cMode, unsigned char* cInfo, int iDataFormat )
int FEISC_0x69_RFReset( int ReaderHnd, unsigned char cBusAdr )
int FEISC_0x6A_RFOnOff( int iReaderHnd, unsigned char cBusAdr, unsigned char cRF )
int FEISC_0x72_SetOutput( int iReaderHnd, unsigned char cBusAdr, unsigned char cMode, unsigned char cOutN, unsigned char* pRecords )
int FEISC_0x74_ReadInput( int iReaderHnd, unsigned char cBusAdr, unsigned char* cInput )
int FEISC_0x80_ReadConfBlock( int iReaderHnd, unsigned char cBusAdr, unsigned char cConfAdr, unsigned char* cConfBlock, int iDataFormat )
int FEISC_0x81_WriteConfBlock( int iReaderHnd, unsigned char cBusAdr, unsigned char cConfAdr, unsigned char* cConfBlock, int iDataFormat )
int FEISC_0x83_ResetConfBlock( int iReaderHnd, unsigned char cBusAdr, unsigned char cConfAdr )
int FEISC_0xA0_RdLogin( int iReaderHnd, unsigned char cBusAdr, unsigned char* cRd_PW, int iDataFormat )

---

```
int FEISC_0xAE_ReaderAuthent( int iReaderHnd, unsigned char cBusAdr, unsigned char cMode, unsigned char cKeyType, unsigned char cKeyLen,  
    unsigned char * cKey, int iDataFormat )
```

---

```
int FEISC_0xB0_ISOCmd( int iReaderHnd, unsigned char cBusAdr, unsigned char* cReqData, int iReqLen, unsigned char* cRspData, int* iRspLen,  
    int iDataFormat )
```

---

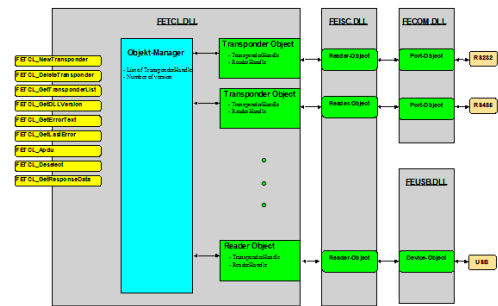
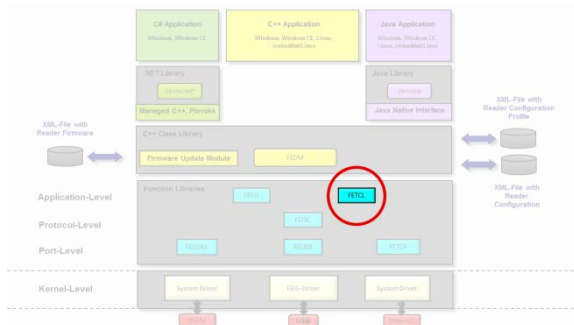
```
int FEISC_0x22_ReadBuffer( int iReaderHnd, unsigned char cBusAdr, int iSets, unsigned char* cTrData, int* iRecSets, unsigned char*  
    cRecDataSets, int iDataFormat )
```

---

Preliminary



#### 4.1.5. FETCL



The library **FETCL** (= **FEIG T=CL**) contains high-level functions for the exchange of APDUs (Application Data Unit Protocol) with ISO 14443-4 compliant Transponders over the T=CL protocol. In principle, multiple APDUs can be handled independent from each other and can be executed simultaneously with different Transponders, if each Reader deals with only one APDU.

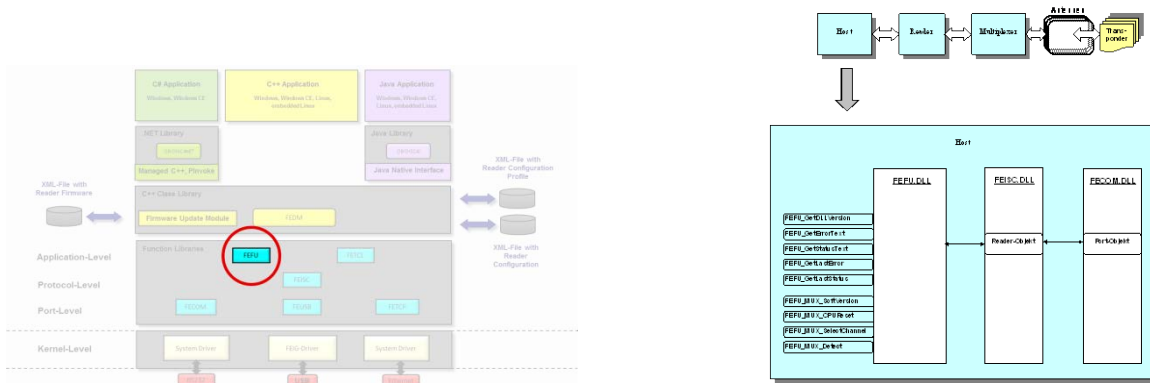
The functions in FETCL are responsible only for internal administration, T=CL protocol cycle handling, like split of uplink data and response data collection and any necessary error outputs. Every other ISO14443 commands, like Inventory, Select or Halt must be executed with FEISC.

The API is identical for all Operating Systems and contains functions for managing of Transponder objects, communication functions and some more helper functions.

A selection of the important functions:

```
int FETCL_NewTransponder( int iReaderHnd, unsigned char ucBusAdr, unsigned char ucCid, unsigned char ucNad, bool bUseCid, bool bUseNad )
int FETCL_DeleteTransponder( int iTrpHnd )
int FETCL_GetErrorText( int iErrorCode, char* cErrorText )
int FETCL_APDU( int iTrpHnd, unsigned char* ucData, int iDataLen, FETCL_EVENT_INIT* pInit )
int FETCL_GetResponseData( int iTrpHnd, unsigned char* ucData, int iDataBufLen )
int FETCL_Ping( int iTrpHnd )
int FETCL_Deselect( int iTrpHnd )
```

#### 4.1.6. FEFU



The library FEFU (= **FEIG Function Unit**) incorporates for the programmer all the necessary functions for easy communication with external Function Units that are accessed by Readers in the OBID i-scan® family. The picture on the right side shows the chain of communication within and outside the host.

Multiple Function Units can be handled, but only one communication at the same time can be executed.

The API is identical for all Operating Systems and contains communication functions and some more helper functions.

Function list:

```
int FEFU_GetErrorText( int iErrorCode, char* cErrorText )
```

```
int FEFU_GetStatusText( unsigned char ucStatus, char* cStatusText )
```

```
int FEFU_MUX_Detect( int iReaderHnd, unsigned char ucReaderBusAdr, unsigned char ucMuxAdr )
```

```
int FEFU_MUX_SoftVersion( int iReaderHnd, unsigned char ucReaderBusAdr, unsigned char ucMuxAdr, unsigned char* ucVersion )
```

```
int FEFU_MUX_SelectChannel( int iReaderHnd, unsigned char ucReaderBusAdr, unsigned char ucMuxAdr, unsigned char ucIn1, unsigned char ucIn2 )
```

```
int FEFU_DAT_Detect( int iReaderHnd, unsigned char ucReaderBusAdr, unsigned char ucDatAdr )
```

```
int FEFU_DAT_GetValues( int iReaderHnd, unsigned char ucReaderBusAdr, unsigned char ucDatAdr, unsigned char* ucValues )
```

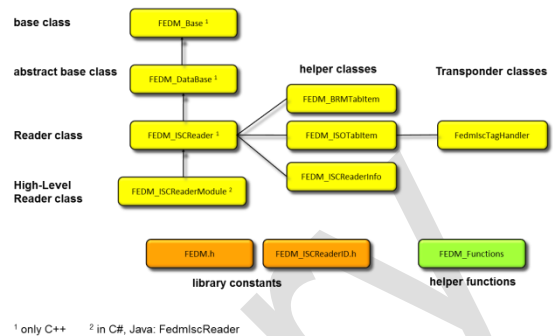
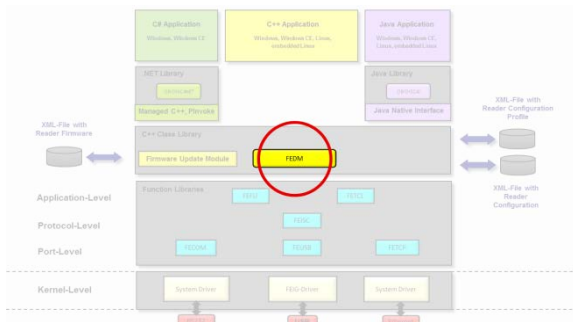
```
int FEFU_UMUX_Detect_GetPower( int iReaderHnd, unsigned char ucReaderBusAdr, unsigned char ucMuxAdr, unsigned char ucFlags, unsigned char* ucData )
```

```
int FEFU_UMUX_SelectChannel( int iReaderHnd, unsigned char ucReaderBusAdr, unsigned char ucMuxAdr, unsigned char ucFlags, unsigned char ucChannelNo )
```

```
int FEFU_UMUX_SoftVersion( int iReaderHnd, unsigned char ucReaderBusAdr, unsigned char ucMuxAdr, unsigned char ucFlags, unsigned char* ucVersion )
```

## 4.2. Class Libraries

### 4.2.1. FEDM



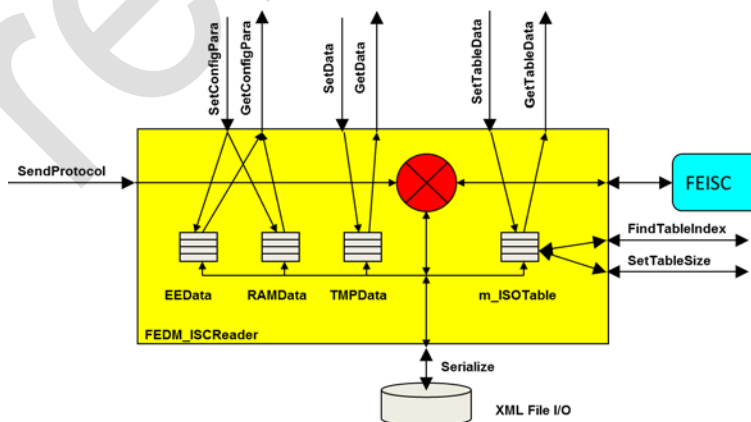
<sup>1</sup> only C++    <sup>2</sup> in C#, Java: FedIntcReader

The C++ class library **FEDM (FEIG Data Modul)** offers a comfortable API for RFID-Readers and – Transponders and should be the best choice for C++ programmers.

Primarily in the year 2000, the library was designed for five, completely different RFID product families with different binary communication protocols to unify the high-level handling with OBID®-Readers. Since then, the library is permanently developed and specialized for the OBID i-scan® and OBID® classic-pro Reader family. A lot of operations can be realized with few code lines by using the high-level Reader class and since 2009/2010 by using Transponder classes.

The library is permanently in development and the support for new Readers will be added in short terms, because of building the basic for the FEIG applications ISOStart/CPRStart and FirmwareUpdateTool.

The principle method of operation of the Reader class can be clearly seen in the following illustration.



The horizontal axis shows the control flow that is generated by the `SendProtocol` method, the only communication method. It independently retrieves all the necessary data from the integrated data containers before transmitting the send protocol and stores the received protocol data there as well. This means the application must write all the data necessary for this protocol to the corresponding data containers in the correct locations before invoking the `SendProtocol`. Likewise the receive data are stored at particular locations in corresponding data containers. The manual for

FEDM (Part B.ISC) contains examples for each Reader command and the following sections demonstrate also a lot of more complex examples.

High-level methods like ReadReaderInfo combine multiple actions and reduce the number of code lines.

The concept of TagHandler classes provides a new library part for more efficient programming with different transponder types. TagHandler classes can be used only when the Reader works in **Host-Mode**.

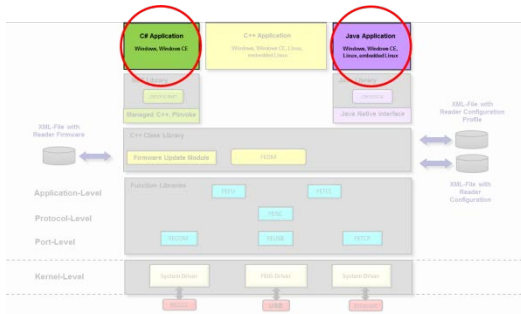
The concept is based on the automatic identification of the type of the transponder after a successful inventory. With ISO 15693 compliant transponders the manufacturer ID and the chipID, which are part of the serial number, are evaluated. With ISO 14443 compliant transponders the type of the TagHandler can be determined after a mandatory Select command based on the returned Card-Info or, in case of the explicit selection of a transponder driver with the Select command, the transponder driver selects the type of the TagHandler.

All TagHandler classes are derived from the base class FedmlscTagHandler. Furthermore, the relationship between the different transponder types is mapped to derivations between TagHandler classes.

A selection of the important high-level methods of the reader class:

```
int ConnectCOMM(int iPortNr)
int ConnectTCP(char* cHostAdr, int iPortNr)
int ConnectUSB(unsigned long dwDeviceID)
int Disconnect()
int ReaderAuthentication(unsigned char ucAuthentType, string sAuthentKey);
FEDM_ISC_READER_INFO* ReadReaderInfo(unsigned int uiProtocolFrame)
FEDM_ISC_READER_DIAGNOSTIC* ReadReaderDiagnostic()
int ReadCompleteConfiguration(bool bEEPROM)
int WriteCompleteConfiguration(bool bEEPROM)
int ResetCompleteConfiguration(bool bEEPROM)
int ApplyConfiguration(bool bEEPROM)
int Serialize(bool bRead, char* sFileName)
int TransferXmlFileToReaderCfg(char* sFileName)
int TransferReaderCfgToXmlFile(char* sFileName)
int StartAsyncTask(FEDM_TASK_INIT* pInit)
int SendProtocol( unsigned char ucCmdByte )
FEDM_ISC_TAG_LIST* TagInventory(bool bAll,unsigned char ucMode, unsigned char ucAntennas)
FedmlscTagHandler* TagSelect(FedmlscTagHandler* pTagHandler, unsigned int uiTagDriver)
FEDM_ISC_TAG_LIST* GetTagList()
FedmlscTagHandler* GetTagHandler(string sSnr)
FedmlscTagHandler* GetSelectedTagHandler()
FEDM_ISOTabItem* GetISOTableItem(unsigned int uIdx)
FEDM_BRMTabItem* GetBRMTTableItem(unsigned int uIdx)
int GetConfigPara( string sParaName, <Typ> Data, bool bEEPROM)
int SetConfigPara( string sParaName, <Typ> Data, bool bEEPROM)
int GetData( const char* ID, <Typ> Data )
int SetData( const char* ID, <Typ> Data )
int GetTableData(int iIdx, unsigned int uiTableID, unsigned int uiDataID, <Typ> Data)
int SetTableData(int iIdx, unsigned int uiTableID, unsigned int uiDataID, <Typ> Data)
int FindTableIndex(int iStartIdx, unsigned int uiTableID, unsigned int uiDataID, <Typ> Data)
```

#### 4.2.2. OBIDISC4J and OBIDISC4NET



The Java library **OBIDISC4J** and the .NET library **OBIDISC4NET** are built upon the C++ class library FEDM. This means, that the most methods are realized in C++ and wrapped into the Java or .NET Reader class. Nevertheless, the full functionality of the C++ class library is accessible for Java or the .NET Framework and results in a similar API.

### 4.3. Thread security

---

In principle, all FEIG libraries are not fully thread safe. But respecting some guidance, a practical thread security can be realized allowing parallel execution of communication tasks. One should keep in mind that all OBID® RFID-Reader works synchronously and can perform commands only in succession.

On the level of the transport layer (FECOM, FEUSB, FETCP) the communication with each port must be synchronized in the application, as the Reader works synchronously. Using multiple ports and so multiple Readers from different threads simultaneously is possible, as the internal port objects acts independently from each other. But it is not possible to communicate independently from different threads with different Readers over one serial port of type RS485 or RS422. Yet another limitation concerns the Scan function of FEUSB library. The scan over the complete USB cannot be thread-safe, as a global kernel action is performed. To prevent mutual interactions, the opening and closing of serial and USB connections must be serialized on application side.

On the level of the protocol layer (FEISC), parallelism can be realized only when each Reader object represents exactly one physical Reader and is bound with an individual communication port. This is not true for the four specialized functions FEISC\_BuildxxProtocol and FEISC\_SplitxxProtocol, which use an internal global buffer for protocol data.

The library FEFU has no precautions for thread-safeness implemented. Thus, only one thread can call FEFU functions at the same time. Thread-safeness must be implemented on application side.

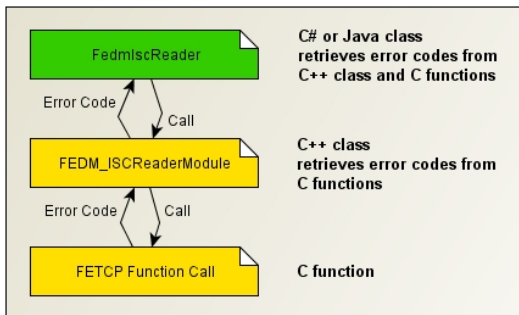
The library FETCL for ISO 14443-4 compliant Transponders is thread-safe, only when each Transponder object is connected with a different Reader object and only one APDU is exchanged with each Reader at the same time. Even if the function FETCL\_Apdu can be called asynchronously, this means not, that multiple calls of FETCL\_Apdu to the same Transponder object are allowed. APDUs are not stored in a stack.

On the level of the class libraries FEDM, OBIDISC4J and OBIDISC4NET parallelism can be realized when with each Reader object only one method call is performed. Thread-safeness for each Reader object must be implemented on application side. Parallelism with non-synchronized opening and closing of serial and USB ports (ConnectCOMM, ConnectUSB) must be avoided. When Function Units are integrated in an application, keep in mind that only one FunctionUnit object can be used at the same time, even if the Function Units are connected on different Readers, as the underlying library FEFU is not thread-safe.

## 5. Error Handling

One of the most important, but likely unattended theme is the error handling. With each received protocol, OBID® RFID-Readers signals with the Status byte the result of the last operation (see [2.3. Transmission Protocol](#)). Every Status byte is listed in the system manual of each Reader.

Beginning with the protocol layer (FEISC), the Status byte is returned with each communication function and **have to be checked** in the application. An operation with the Reader was successful, if the return value is 0, which equates the Status OK. Positive return values relates to the Status byte from the last operation. Negative return values signal an error condition inside the library stack.



The error codes for the class libraries (C++, C#, Java) and the function libraries are organized into sectors such that they cannot overlap. The following ranges are reserved for the libraries:

Library	Value range for error codes	Reference
FEDM, OBIDISC4J, OBIDISC4NET	-101 ... -999	H10102-xx-ID-B, H31101-xx-ID-B, H40301-xx-ID-B
FECOM	-1000...-1099	H80592-xx-ID-B
FEUSB	-1100...-1199	H00501-xx-ID-B
FETCP	-1200...-1299	H30802-xx-ID-B
FEISC	-4000...-4099	H9391-xx-ID-B
FEFU	-4100...-4199	H30801-xx-ID-B
FETCL	-4200...-4299	H50401-xx-ID-B

Calling the method `GetStatusText()` returns a text corresponding to the sent status byte. Calling the method `GetErrorText()` returns a text corresponding to the sent error code, which may also come from the function library FEISC or the underlying communication library FECOM, FETCP or FEUSB.

One of the commonly raised error is the communication timeout caused by too long operation time in the RFID-Reader or too small timeout setting in the library (transport layer). As a general rule, the communication timeout should be set to 5 seconds and only be increased, if the Reader is configured to remain for a longer time (> 5 seconds) in the RF communication with the parameter `AirInterface.TimeLimit` (Transponder Response Time). The communication timeout must always be larger than the setting in `AirInterface.TimeLimit`.

## 6. General preliminary notes to the sections

---

For your better orientation, the source code snippets in the sections are placed in colored boxes and signed with symbols for the programming languages:



C/C++ examples in a **sand colored** box. The snippets are for the C++ class library FEDM. If possible, an alternative snippet is boxed separately with exclusive use of the C function libraries.



Java examples in a **violet** box.



C# examples in a **green** box (with the implicit use for Windows and Windows CE / Windows Compact).

If some examples relate to a specific operating system, the boxes are signed with the following symbols:



Windows and Windows CE / Windows Compact



Windows CE / Windows Compact



Linux (also embedded Linux)

Examples without an operating system symbol are universal.

For reasons of clarity the processing of the return values of the methods is not shown here. Of course it should always be included in applications.

The label *reader* represents an object of type FEDM\_ISCReaderModule (C++) or FedmIsCReader (Java, .NET).

All method names in this Tutorial begin with a capital letter, although methods in Java begin with lower-case.



## 7. Section 1: Basic initializations

---

Some values or instance of classes have to be hold in the application. By exclusive use of the function libraries, this might be the Port Handle as the return value of `FECOM_OpenPort`, `FEUSB_OpenDevice` and `FETCP_Connect` and the Reader Handle as the return value of `FEISC_NewReader`. In object oriented applications (C++, C#, Java) we recommend to instantiate for each physical RFID-Reader one reader instance, which remains the complete application run-time or as long the connection is established.

Before using the reader object (C++, C#, Java) for the first time, some initializing must be performed:

1. Bus address                      The bus address for the Reader is preset in the class for 255. To set a different address, use the `SetBusAddress` method.  
  
Note: The bus address is only relevant for the communication over the serial port.
2. Reader type                      The reader type must be set in the reader class with one of two options:
  1. The call of the method `ReadReaderInfo` after a successful connection (recommended).
  2. Set of reader type with the method `SetReaderType`. The constants of all reader types are listed in the file `FEDM_ISC.h` (C++), in the class `FedmlscReaderConst` (Java) and in the structure `FedmlscReaderConst` (.NET).
3. Table size                        The integrated tables for Buffered Read Mode (BRM) and ISO Host Mode are not initialized. Before the initial communication, you must set the table size using the method `SetTableSize`. The size is selected equal to the maximum number of transponders located in the antenna field at the same time.  
  
Only the size of the table actually being used needs to be set.
4. TagHandler-Support              The support for TagHandler classes is disabled by default. If TagHandler classes are used in an application, the support must be enabled with the method `EnableTagHandler`.  
(only C++)

**FEDM**

```
#include FedmIsCore.h

FEDM_ISCReaderModule reader;

// set a proper busaddress (only for communication over serial port)
reader.SetBusAddress(1); // or other proper bus address

// set of table size
// use the same method with FEDM_ISC_BRM_TABLE when programming for Buffered-Read-Mode
reader.SetTableSize(FEDM_ISC_ISO_TABLE, 100); // max. 100 Tags per Inventory in Host-Mode

// we want to use TagHandler support
reader.EnableTagHandler(true);

// set of Reader-Type after a successful connection
reader.ReadReaderInfo();
```

**FECOM and FEISC**

```
#include fecom.h // and/or feusb.h and/or fetcp.h
#include feisc.h

int iPortHandle = 0;
int iReaderHandle = 0;
```



```
import de.feig.*;

FedmIsReader reader = new FedmIsReader();

// set a proper busaddress (only for communication over serial port)
reader.setBusAddress(1); // or other proper bus address

// set of table size
// use the same method with FEDM_ISC_BRM_TABLE when programming for Buffered-Read-Mode
reader.setTableSize(FEDM_ISC_ISO_TABLE, 100); // max. 100 Tags per Inventory

// the Reader-Type is set automatically with the connectXXX method
```

**C#**

```
using OBID;

FedmIsReader reader = new FedmIsReader();

// set a proper busaddress (only for communication over serial port)
reader.SetBusAddress(1); // or other proper bus address

// set of table size
// use the same method with FEDM_ISC_BRM_TABLE when programming for Buffered-Read-Mode
reader.SetTableSize(FedmIsReaderConst.FEDM_ISC_ISO_TABLE, 100); // max. 100 Tags per Inventory

// the Reader-Type is set automatically with the ConnectXXX method
```

## 8. Section 2: Establish a connection to the Reader

This section gives an introduction to the operations for establishing a connection to the RFID-Reader.

### 8.1. Serial Port (RS232 / RS485 / RS422)

A singular resource and under control of the operating system is the serial port, which can be opened once. The serial interface can handle only one protocol at the same time. Parallelism is therefore not realizable. After opening, a serial port must be configured to adjust the transmission parameters to the connected RFID-Reader.

In the following example the port COM1 (Linux: ttyS0) is opened, the baud rate is set to 38400 and the frame to 8E1 (8 data bits, even parity, 1 stop bit). The timeout is 1000ms by default. It must be increased, if the response time of some Reader commands is larger.



#### FEDM

```
int back = reader.ConnectCOMM(1);
if(back == 0)
{
    // search for the connected Reader
    // the port parameters and the protocol frame (Standard or Advanced) will be set properly internal
    back = reader.FindBaudRate();
    if(back = 0)
    {
        // Reader detected
        reader.SetPortPara("Timeout", "5000"); // 5s timeout
    }
}
```

#### FECOM and FEISC

```
int back = 0;
int iPortHandle = 0;
int iReaderHandle = 0;

iPortHandle = FECOM_OpenPort("1");
if(iPortHandle > 0)
{
    FECOM_SetPortPara(iPortHandle, "Baud", "38400"); // or other proper baud rate
    FECOM_SetPortPara(iPortHandle, "Frame", "8E1"); // or other proper frame

    // create new Reader object and connect with COM1 Port object in FECOM
    iReaderHandle = FEISC_NewReader(iPortHandle);
    if(iReaderHandle > 0)
    {
        // for most OBID® Reader (not for ISC.MR/PR/PRH100, ISC.LR200, ISC.M02, CPR.M02, CPR.02,
        // CPR.04)
        FEISC_SetReaderPara(iReaderHandle, "FrameSupport", "Advanced");
        // search for the connected Reader
        // with busaddress 255 each Reader is answering
        back = FEISC_0x52_GetBaud(iReaderHandle, 255);
        if(back = 0)
        {
            // Reader detected
            FECOM_SetPortPara(iPortHandle, "Timeout", "5000"); // 5s timeout
        }
    }
}
```



```
import de.feig.*;

try
{
    // search for the connected Reader
    // the port parameters will be set properly internal with FindBaudRate()
    // use parameter with Detect=true only, if the Reader is normally connected
    reader.connectCOMM(1, true);
    // Reader detected
    reader.setPortPara("Timeout", "5000"); // 5s timeout
}
catch (Exception ex)
{
    ...
}
```



```
C#

using OBID;

try
{
    // search for the connected Reader
    // the port parameters will be set properly internal with findBaudRate()
    // use parameter with Detect=true only, if the Reader is normally connected
    reader.ConnectCOMM(1, true);
    // Reader detected
    reader.SetPortPara("timeout", "5000");
}
catch (Exception ex)
{
    ...
}
```

## 8.2. Bluetooth

For Windows and Windows CE, OBID® RFID-Readers with Bluetooth interface are connected at a virtual serial port (e.g. COM15) and with Linux as a /dev/rfcomm device.

The Bluetooth interface is a singular resource and under control of the operating system. It can be opened once and handle only one protocol at the same time. Parallelism is therefore not realizable. After opening, a Bluetooth interface must not be configured.

In the following example the port COM1 (Linux: /dev/rfcomm1) is opened. The timeout is 1000ms by default. It must be increased, if the response time of some Reader commands is larger.

Note: For Windows and Windows CE there are no modifications against the handling of serial ports. Linux programmers have to adjust the device name first.



### FEDM

```
// same operations as for serial ports
```



### FECOM and FEISC

```
// same operations as for serial ports
```



### FEDM

```
int back = 0;

// adjust device name
reader.SetPortPara("PortPrefix", "/dev/rfcomm");
back = reader.ConnectCOMM(1); // opens dev/rfcomm0 and not /dev/rfcomm1 !!
// reset device name to default
reader.SetPortPara("PortPrefix", "/dev/ttyS");
if(back == 0)
{
    // search for the connected Reader
    back = reader.FindBaudRate();
    if(back = 0)
    {
        // Reader detected
        reader.SetPortPara("Timeout", "5000"); // 5s timeout
    }
}
```



### FECOM and FEISC

```
int back = 0;
int iPortHandle = 0;
int iReaderHandle = 0;

// adjust device name
FECOM_SetPortPara(0, "PortPrefix", "/dev/rfcomm");
iPortHandle = FECOM_OpenPort("1"); // opens dev/rfcomm0 and not /dev/rfcomm1 !!
// reset device name to default
FECOM_SetPortPara(0, "PortPrefix", "/dev/ttyS");
if(iPortHandle > 0)
{
    FECOM_SetPortPara(iPortHandle, "Baud", "38400"); // proper setting for Bluetooth
    FECOM_SetPortPara(iPortHandle, "Frame", "8E1"); // proper setting for Bluetooth

    // create new Reader object and connect with /dev/rfcomm0 Port object in FECOM
    iReaderHandle = FEISC_NewReader(iPortHandle);
    if(iReaderHandle > 0)
    {
```

```
// for all OBID® Reader with Bluetooth interface
FEISC_SetReaderPara(iReaderHandle, "FrameSupport", "Advanced");
// search for the connected Reader
// with busaddress 255 each Reader is answering
back = FEISC_0x52_GetBaud(iReaderHandle, 255);
if(back == 0)
{
    // Reader detected
    FECOM_SetPortPara(iPortHandle, "Timeout", "5000"); // 5s timeout
}
}
```



```
// same operations as for serial ports
```



```
import de.feig.*;

try
{
    // search for the connected Reader
    // the port parameters will be set properly internal with findBaudRate()
    // use parameter with Detect=true only, if the Reader is normally connected
    reader.connectBT(1, true); // opens dev/rfcomm0 and not /dev/rfcomm1 !!
    // Reader detected
    reader.setPortPara("Timeout", "5000"); // 5s timeout
}
catch (Exception ex)
{
    ...
}
```



### C#

```
// same operations as for serial ports
```

### 8.3. USB

---

OBID® RFID-Reader with USB-Interface requires for Windows and Windows CE a FEIG Kernel-Driver and for Linux and Mac OS X the Open-Source library libusb must be installed.

USB is a single-master bus with the PC as master (host). Only this master can generate protocol activities. Up to 127 physical devices can be supported at the same time. The devices differ in their bus addresses, which are automatically assigned by the host. After a peripheral is plugged in, an initialisation phase (enumeration) is automatically started in the host which allows the host to load the appropriate driver(s). This process is always triggered by the operating system.

In physical terms a USB device always consists of at least one logical USB device. This means the communication data can be stacked within the device into several information channels, the so-called pipes. Each pipe has an endpoint assigned to it which corresponds physically to a FIFO.

A logical USB device can combine several pipes into an interface, and the host can install an appropriate driver for such an interface. The host obtains the information about the logical composition of a USB device during enumeration.

USB devices from the OBID® Reader families are characterized in that they all have uniform interfaces. This means the special USB drivers can be categorized as device-independent within the OBID® Reader families. The programmer does not however come into contact with these drivers, interfaces, pipes or bus addresses. For him a programming model has been developed which enables communication with OBID® USB devices in no more than four steps.

1. Scan process: A function invoke detects all OBID® devices on the USB and administers them in a scan list within the DLL.
2. Device selection: In the second step this scan list is used to select a USB device based on its serial number (Device-ID). The serial number is by the way the only feature which distinguishes the devices from each other.
3. Open communications path: In the third step a channel to this USB device is opened. A data structure, the device object, is created internally in the DLL.
4. Data exchange: Beginning with the fourth step data can be exchanged with the USB device.

In the case of only one USB device from the OBID® Reader families is connected, a special function/method in the libraries combines steps 1, 2 and 3 together.

**FEDM**

```

int back = 0;
long dwDeviceID = 0;

// detect single Reader (the first detected will be opened)
back = reader.ConnectUSB(0);
if(back == 0)
{
    // Reader detected
}

// alternatively, the USB must be scanned with the function library FEUSB (see below)
// to identify a specific Reader

// detect Reader with specific Device-ID
// obtain the Device-ID from FEUSB library (see below)
back = reader.ConnectUSB(dwDeviceID);

```

**FEUSB and FEISC**

```

int back = 0;
int iPortHandle = 0;
int iReaderHandle = 0;
long dwDeviceID = 0;
char cDeviceID[16];

// scan USB and open first detected Reader. No search option is used
iPortHandle = FEUSB_ScanAndOpen(FEUSB_SCAN_FIRST, NULL);
if(iPortHandle > 0)
{
    // create new Reader object and connect with USB device object in FEUSB
    iReaderHandle = FEISC_NewReader(iPortHandle);
    if(iReaderHandle > 0)
    {
        // for all OBID® USB-Reader recommended (except for CPR.04-U)
        FEISC_SetReaderPara(iReaderHandle, "FrameSupport", "Advanced");
    }
}

// alternatively, the USB must be scanned to identify a specific Reader

// scan USB and open e.g. the second USB device. No search option is used
back = FEUSB_Scan(FEUSB_SCAN_ALL, NULL);
if(back == 0)
{
    if( FEUSB_GetScanListPara(1, "Device-ID", cDeviceID) == 0 )
    {
        sscanf((const char*)cDeviceID, "%lx", &dwDeviceID);
        iPortHandle = FEUSB_OpenDevice(dwDeviceID);
        if( iPortHandle < 0 )
        {
            // code here for error
        }
        else
        {
            // code here for communication or other
        }
    }
}
}

```





```
import de.feig.*;

int back = 0;
long deviceID = 0;
String scanListPara;
FeUsb usbHelper = new FeUsb();
FeUsbScanSearch scanSearch = null;

try
{
    // detect single Reader (the first detected will be opened)
    reader.connectUSB(0);
    // Reader detected
}
catch (Exception ex)
{
    ...
}

// alternatively, the USB must be scanned with the class FeUsb
// to identify a specific Reader

// scan USB and open e.g. the second USB device. No search option is used
try
{
    back = usbHelper.scan(FeUsbScanSearch.SCAN_ALL, scanSearch);
    if(back == 0)
    {
        scanListPara = usbHelper.getScanListPara(1, "Device-ID");
        deviceID = FeHexConvert.hexStringToLong(scanListPara);
        reader.connectUSB(deviceID);
    }
}
catch (Exception ex)
{
    ...
}
```



```
C#

using OBID;

int back = 0;
long deviceID = 0;
String scanListPara;
FeUsb usbHelper = new FeUsb();
FeUsbScanSearch scanSearch = null;

try
{
    // detect single Reader (the first detected will be opened)
    reader.ConnectUSB(0);
    // Reader detected
}
catch (Exception ex)
{
    ...
}

// alternatively, the USB must be scanned with the class FeUsb
// to identify a specific Reader

// scan USB and open e.g. the second USB device. No search option is used
try
{
    back = usbHelper.Scan(FeUsbScanSearch.SCAN_ALL, scanSearch);
    if(back == 0)
    {
        scanListPara = usbHelper.GetScanListPara(1, "Device-ID");
        deviceID = FeHexConvert.HexStringToLong(scanListPara);
        reader.ConnectUSB(deviceID);
    }
}
catch (Exception ex)
{
    ...
}
```

## 8.4. TCP/IP (LAN and WLAN)

The TCP/IP communication interface is a singular resource in the RFID-Reader. It can be opened once and handles only one protocol at the same time. Parallelism is therefore not realizable. This singularity is consciously willed, in order to bind a RFID-Reader with only one process.

In the following example a connection to a Reader with the IP-Address 192.168.1.100 and Port 10001 is established.



### FEDM

```
int back = 0;

back = reader.ConnectTCP("192.168.1.100", 10001);
if(back == 0)
{
    // Reader detected
    // call of ReadReaderInfo() necessary -> see 10.1. Reader Information: The method ReadReaderInfo\(\)
}
```

### FETCP and FEISC

```
int back = 0;
int iPortHandle = 0;
int iReaderHandle = 0;

iPortHandle = FETCP_Connect("192.168.1.100", 10001);
if(iPortHandle > 0)
{
    // create new Reader object and connect with socket object in FETCP
    iReaderHandle = FEISC_NewReader(iPortHandle);
    if(iReaderHandle > 0)
    {
        // for all OBID® Reader with LAN and WLAN interface recommended
        FEISC_SetReaderPara(iReaderHandle, "FrameSupport", "Advanced");
    }
}
```



```
import de.feig.*;

try
{
    reader.connectTCP("192.168.1.100", 10001);
    // Reader detected
}
catch (Exception ex)
{
    ...
}
```



```
C#

using OBID;

try
{
    reader.ConnectTCP("192.168.1.100", 10001);
    // Reader detected
}
catch (Exception ex)
{
    ...
}
```

## 8.5. Excursion: Secured data transmission with encryption

Some OBID i-scan®- and OBID® classic-pro Reader can secure the data transmission over Ethernet (TCP/IP) with a 256 bit AES algorithm. The Authentication Key (Password) is stored in the Reader and cannot read back. The crypto mode is disabled by default.

The encrypted data transmission is realized with functions of the Open-Source organization openssl (<http://www.openssl.org>), which are part of the library file libeay32.dll (Windows) resp. libcrypto.so (Linux). The binding to the openssl library file will be affected at runtime with the first call of an openssl function. This has the advantage that all applications are freed from the installation of the openssl library file if no encrypted data transmission is used. In the case that encrypted data transmission is used the license issues of openssl have to be considered.

The encrypted data transmission will be enabled by activating the crypto mode in the Reader configuration with a following CPU-Reset. After that, the Reader accepts only enciphered protocols. To get access rights in crypto mode, the first step must be the establishment of a secured connection, transporting the enciphered password (password contains only nulls by default), to open a new session. Every successive protocol will then enciphered automatically.



### FEDM

```
int back = 0;

back = reader.ConnectTCP("192.168.1.100", 10001);
if(back == 0)
{
    // Reader detected
    back = reader.ReaderAuthentication(2, "MyAuthentKeyInHex"); // 1
    if(back == 0)
    {
        // authenticated
    }
}
```

### FETCP and FEISC

```
int back = 0;
int iPortHandle = 0;
int iReaderHandle = 0;

iPortHandle = FETCP_Connect("192.168.1.100", 10001);
if(iPortHandle > 0)
{
    // create new Reader object and connect with socket object in FETCP
    iReaderHandle = FEISC_NewReader(iPortHandle);
    if(iReaderHandle > 0)
    {
        // for all OBID® Reader with LAN and WLAN interface recommended
        FEISC_SetReaderPara(iReaderHandle, "FrameSupport", "Advanced");
        // try to authenticate
        back = FEISC_OxAE_ReaderAuthent( iReaderHandle, 255, 0, 2, 32,
                                         "MyAuthentKeyInHex", 1); // 1

        if(back == 0)
        {
            // authenticated
        }
    }
}
```

<sup>1</sup> The Authent-Key has to be a string with 64 hexadecimal characters, where every two hex characters build a byte. All 32 bytes together build a AES256-Key



```
import de.feig.*;

try
{
    reader.connectTCP("192.168.1.100", 10001, 2, "MyAuthentKeyInHex"); // 1
    // Reader detected and we are authenticated
}
catch (Exception ex)
{
    ...
}
```



```
C#

using OBID;

try
{
    reader.ConnectTCP("192.168.1.100", 10001, 2, "MyAuthentKeyInHex"); // 1
    // Reader detected and we are authenticated
}
catch (Exception ex)
{
    ...
}
```

<sup>1</sup> The Authent-Key has to be a string with 64 hexadecimal characters, where every two hex characters build a byte. All 32 bytes together build a AES256-Key

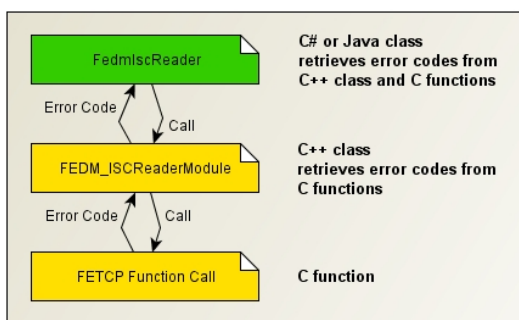
## 8.6. Excursion: Error handling for TCP/IP communication

TCP/IP based communication is normally easy to realize. But in error cases, the handling is different from serial or USB based communication.

In the following, we discuss error handling for:

- Communication errors
- Errors while establish a connection
- Errors while closing the connection
- Problem with broken communication link

The method/function calls inside the libraries is important to be considered to understand the returned error codes or thrown exceptions.



With the principle: the caller reflects the error code of the called method/function, e.g. a call from C# can throw error codes from C++ library and that can return error codes from the protocol layer (FETCP), the following tables list the most important error codes and the recommended error handling. All other error codes are critical errors, cannot be fixed at runtime and must be analyzed by the development team.

### 8.6.1. Communication errors

In general, when a communication with the method `SendProtocol` (C++, Java, .NET) or the function `FETCP_Transceive` (C/C++, VB, ...) fails with error codes -1230 (Timeout), -1232 (Error in read process) or -1237 (error in send process), the connection must be closed at once and established again.

If a timeout is ignored and another OBID protocol is sent afterwards, the timed out receive protocol may be received. After this, a displacement of the receive protocol is permanent existent. Only closing and opening of the connection can fix this situation.

The preset timeout is 3000ms and normally large enough for the most communication tasks. In rare cases it must be enlarged with the method `FEDM_ISCReaderModule::SetPortPara` (C++), `FedmIsrReader.SetPortPara` (Java, .NET) or `FETCP_SetPortPara`.

**For ID ISC.LR/LRU Readers:** It is mandatory to add a short sleep time of 500-1000 ms between closing and opening for RFID-Readers with embedded AC-Controller. Otherwise, the connection a) may fail or b) may be successful, but the first communication may fail.

### 8.6.2. Errors while establish a connection

When an error occurs with the method `ConnectTCP` (C++, Java, .NET) or `FETCP_Connect` (C/C++, VB, ...), the error code must be analyzed in detail while error handling.

Function / Method	Error code	Error handling
Function in transport layer <b>FETCP_Connect</b>	-1211	Timeout for establishing a connection to the TCP/IP server. Cause may be that another client is blocking the connection.  This is a normal runtime problem. The repetitive call can be applied until the Server (RFID-Reader) can be connected.  Other reasons: the RFID-Reader is not powered on or not switched into the subnet or not configured properly concerning the TCP parameters. This must be analyzed by the installation team
	-1212	The parameter <code>cHostAdr</code> in the function is structurally defective.  This is a critical error and must be analyzed by the development team.
	-1251	Pass parameter too large or too small, here: the transferred port number is out of range.  This is a critical error and must be analyzed by the development team.
Method in C++ class library <b>FEDM_ISCReaderModule::ConnectTCP</b>	-106	Unknown transfer parameter, here: the transferred port number is out of range.  This is a critical error and must be analyzed by the development team.
	-137	Reader object is already connected with a communication port.  This is a runtime problem with multiple reasons: <ul style="list-style-type: none"> <li>a) Another application is connected with this RFID-Reader. This is a normal runtime problem. The repetitive call can be applied until the Server (RFID-Reader) can be connected.</li> <li>b) Multiple call of <code>ConnectTCP</code> with the same Reader objects from the same application. This is a critical error in the application structure and must be analyzed by the development team.</li> </ul>
	-157	Reader object is already connected with a communication port.  The reason is a multiple call of <code>ConnectTCP</code> with different Reader objects from the same application. This is a critical error in the application structure which is not supported by the class libraries and must be analyzed by the development team.
Method in Java/.NET class library <b>FedmlscReader::ConnectTCP</b> (includes an internal call of <code>ReadReaderInfo</code> )	-1230 -1232 -1237	Communication error while reading the Reader-Info and the connection is closed internally.  This is an abnormal error and must be analyzed by the development or installation team.

### 8.6.3. Errors while closing the connection

The closing of a connection is realized internally with a call of `closesocket` (Windows) or `close` (Linux) and returns while the process of closing is not finished. Thus, although the disconnection from the application-side is finished, the final TCP status `TIME_WAIT` is probably not yet reached. To indicate this situation, the last TCP status is reflected to inform the application. With successive calls of the static method `GetTcpConnectionState(ip, port)` (.NET) or the function `FETCP_GetSocketState(ip, port)` (C/C++, VB, ...) the closing process can be observed.

Only two important errors can occur with the method `Disconnect` (C++, Java, .NET) or `FETCP_Disconnect` (C/C++, VB, ...):

Function / Method	Error code	Error handling
Function in transport layer <b>FETCP_Disconnect</b>	-1213	The socket in the Operating System cannot be closed and remains open.  The disconnection must be repeated, until it is successful.
	1 - 10	The socket is closed, but the last TCP status is returned as the final status <code>TIME_WAIT</code> is not reached.
Method in C++ class library <b>FEDM_ISCReaderModule::Disconnect</b>	-138	No connection is enabled and nothing is to be closed.  This error code indicates a structural code problem in the application and should be analyzed by the development team.  At runtime, this error can be ignored,
Method in Java/.NET class library <b>FedmlscReader::Disconnect</b>	-	no additional error codes

### 8.6.4. Problem with broken communication link – the Keep-Alive option

When the Ethernet cable gets broken while an active communication, the server-side application (Reader) may not indicate an error while it is listening for new transmissions. On the other side, the host application will run in an error with the next transmission and can close and reopen the socket. But the close and reopen will never be noticed by the Reader, as he is listening at a half-closed port.

The solution for this very realistic scenario is the activating of the Keep-Alive option on the server-side. Every OBID *i-scan*® and OBID® *classic-pro* Reader with Ethernet interface has parameters for Keep-Alive and it is recommended to enable this option.



## 8.7. Excursion: Detecting Readers with different Protocol Frames in one App

In constellations when OBID i-scan®- and/or OBID® *classic-pro* Readers supporting only Standard Protocol Frame or only Advanced Protocol Frame must be detected together in one application, the detection algorithm must be prepared with more logic to prevent long-term timeouts.

The following table illustrates the Protocol Frame support situation for the Reader Families:

Only Standard Protocol Frame	Both Protocol Frames	Only Advanced Protocol Frame
<u>1<sup>st</sup> and 2<sup>nd</sup> gen. Short-Range HF</u> ID ISC.M01, ID ISC.M02	-	future Short-Range HF gen.
<u>1<sup>st</sup> gen. Mid-Range HF</u> ID ISC.MR/PR/PRH100	<u>2<sup>nd</sup> gen. Mid-Range HF</u> ID ISC.MR101	<u>3<sup>rd</sup> gen. Mid-Range HF</u> ID ISC.MR102
<u>1<sup>st</sup> gen. Long-Range HF</u> ID ISC.LR200	<u>2<sup>nd</sup> gen Long-Range HF</u> ID ISC.LR2000	<u>3<sup>rd</sup> gen. Long-Range HF</u> ID ISC.LR2500-x
<u>1<sup>st</sup> gen. classic-pro</u> ID CPR.M02, ID CPR.02, ID CPR.04	<u>2<sup>nd</sup> gen. classic-pro</u> ID CPR40, ID CPR44, ID CPR50, ID CPR52, ID MAX50	all future OBID® <i>classic-pro</i> Readers
	all not named RFID-Readers	

In the ambition to provide the markets with high performance RFID-Products, future RFID-Readers will only support the Advanced Protocol Frame.

High-level APIs like C++, Java or .NET class libraries have integrated a flexible algorithm with the methods `FindBaudRate()`<sup>1</sup> and `ReadReaderInfo()`. Programmers using these Methods have nothing to do.

Applications based on Low-level APIs (up to the protocol layer realized with FEISC) must implement this logic separately. For serial port and USB, different strategies must be applied. For TCP/IP Readers the protocol frame can always be set to Advanced.

<sup>1</sup> only for serial or Bluetooth connection

### 8.7.1. Detecting at serial port

Detecting a Reader at the serial port needs protocol transmissions. The program flow illustrated below works with the simple [0x52] Get Baud protocol, which is probed with different port frames, different port baud rates and alternating protocol frames. All this with reduced timeout to speed up the detection process. When all transmission settings are fit, the Reader returns a response, otherwise, the transmission runs in a timeout.

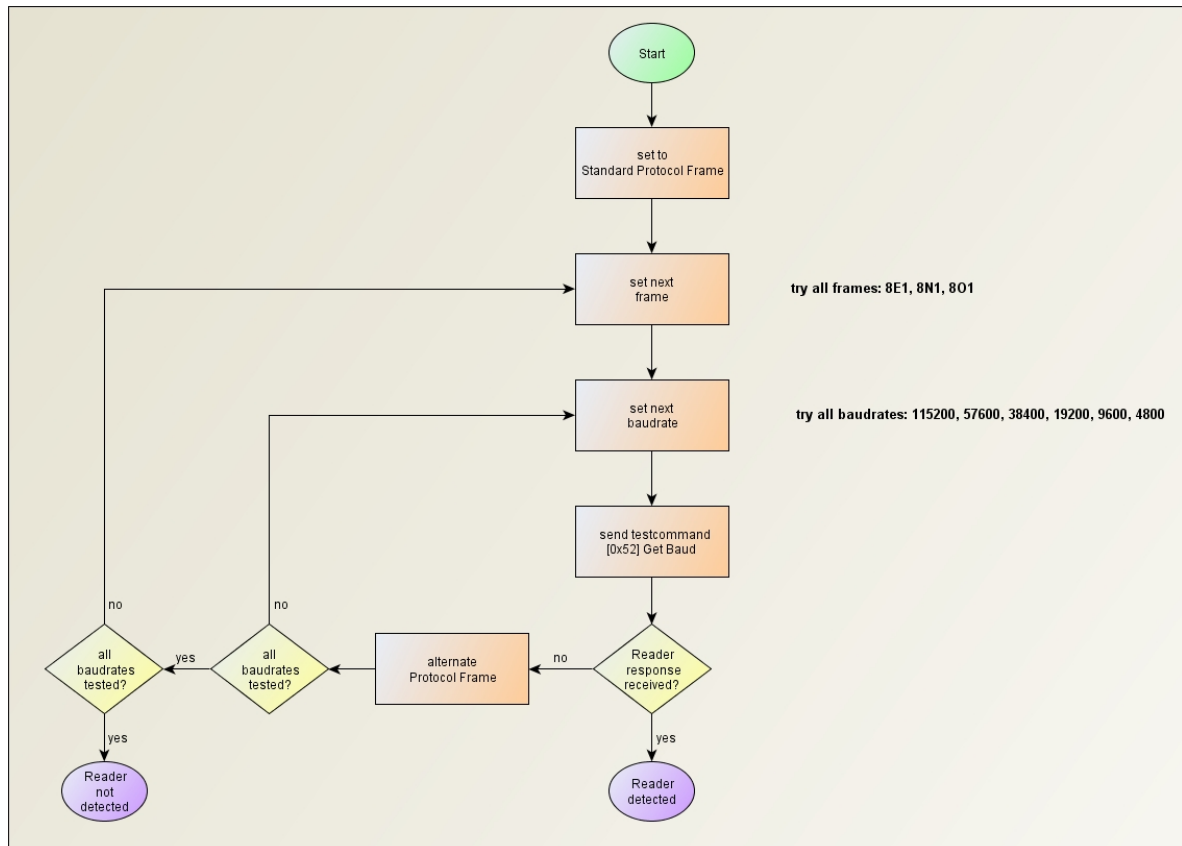


Figure 3: Detection algorithm



#### FECOM and FEISC

```

// NOTE: this sample is without error handling. In real applications, the error handling for every
//       FECOM and FEISC function call must be added!!

int back = 0;
int iPortHandle = FECOM_OpenPort("1");
int iReaderHandle = FEISC_NewReader(iPortHandle);
char cTimeout[8];
char cOldBaud[8];
char cOldFrame[8];
char cOldTimeout[8];
char cPrtFrame[10];
char* baud[] = { "115200", "57600", "38400", "19200", "9600", "4800" };
char* frame[] = { "8E1", "8N1", "8O1" };

if(iPortHandle > 0 && iReaderHandle > 0)
{
    // call detect function
    back = MyDetectCOM(iPortHnd, iReaderHandle);
}
  
```

```
// detect function for serial port
int MyDetectCOM(int iPortHnd, int iReaderHandle)
{
    // save actual baud rate, frame, timeout
    FECOM_GetPortPara( iPortHnd, "baud", cOldBaud );
    FECOM_GetPortPara( iPortHnd, "frame", cOldFrame );
    FECOM_GetPortPara( iPortHnd, "timeout", cOldTimeout );

    // reduce timeout to 300ms
    FECOM_SetPortPara( iPortHnd, "timeout", "300");

    // for some OBID® Readers (ISC.MR/PR/PRH100, ISC.LR200, ISC.M02, CPR.M02, CPR.02, CPR.04)
    // we setup Standard Protocol Frame to give these Readers a chance
    FEISC_SetReaderPara(iReaderHandle, "FrameSupport", "STANDARD");

    for( int j=0; j<3; j++ )    // try all frames "8E1", "8N1", "8O1"
    {
        // set the next frame
        FECOM_SetPortPara( iPortHnd, "frame", frame[j]);

        for( int i=0; i<3; i++ )    // try all baud rates, beginning with "115200"
        {
            // set the next baud rate
            FECOM_SetPortPara( iPortHnd, "baud", baud[i]);

            // search for the connected Reader
            // with busaddress 255 each Reader is answering
            back = FEISC_0x52_GetBaud(iReaderHandle, 255);
            if(back == 0)
            {
                // Reader detected, restore communication timeout
                FECOM_SetPortPara(iPortHandle, "Timeout", cOldTimeout);
                return -1;
            }
        }

        // alternate protocol frame
        FEISC_GetReaderPara(iReaderHandle, "FrameSupport", cPrtFrame);
        if(strcmp(cPrtFrame, "STANDARD") == 0)
            FEISC_SetReaderPara(iReaderHandle, "FrameSupport", "ADVANCED");
        else
            FEISC_SetReaderPara(iReaderHandle, "FrameSupport", "STANDARD");
    }

    // no Reader detected, restore previous port settings
    FECOM_SetPortPara(iPortHandle, "baud", cOldBaud);
    FECOM_SetPortPara(iPortHandle, "frame", cOldFrame);
    FECOM_SetPortPara(iPortHandle, "Timeout", cOldTimeout);
    return 0;
}
```

### 8.7.2. Detecting at USB

Detecting a Reader at USB needs no protocol transmissions and it makes the setting of the protocol frame much easier. When a Reader is detected by the kernel driver, the Reader's name is requested and evaluated. If the Reader name is found in a positive list (CPR.04, ISC.MR100, ISC.PR100), the protocol frame is set to Standard, otherwise to Advanced.



#### FEUSB and FEISC

```
// NOTE: this sample is without error handling. In real applications, the error handling for every
// FEUSB and FEISC function call must be added!!

int back = 0;
int iReaderHandle = FEISC_NewReader(0);
char cPortHandle[11];
char cDeviceName[32];

if(iReaderHandle > 0)
{
    // call detect function
    back = MyDetectUSB(iReaderHandle);
}

// detect function for USB
int MyDetectUSB(int iReaderHandle)
{
    int iPortHandle = FEUSB_ScanAndOpen(0); // connect first found OBID® Reader
    if(iPortHandle < 0)
        return -1; // no Reader connected

    // set port handle in reader object
    sscanf(cPortHandle, "%d", &iPortHandle);
    FEISC_SetReaderPara(iReaderHandle, "PortHnd", cPortHandle);

    // request Reader name to detect ISC.MR/PR100-U or CPR.04-USB,
    // which supports only Standard Protocol Frame
    // all other USB-Readers supports Advanced Protocol Frame

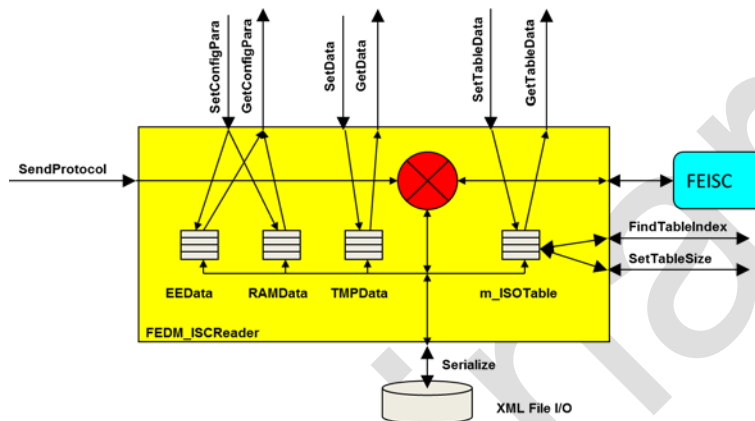
    FEUSB_GetDevicePara(iPortHandle, "", cDeviceName);
    if(strcmp(cDeviceName, "ID CPR.04-USB") == 0)
        FEISC_SetReaderPara(iReaderHandle, "FrameSupport", "STANDARD");
    else if(strcmp(cDeviceName, "ID ISC.MR100-U") == 0)
        FEISC_SetReaderPara(iReaderHandle, "FrameSupport", "STANDARD");
    else if(strcmp(cDeviceName, "ID ISC.PR100-U") == 0)
        FEISC_SetReaderPara(iReaderHandle, "FrameSupport", "STANDARD");
    else
        FEISC_SetReaderPara(iReaderHandle, "FrameSupport", "ADVANCED");

    // Reader detected
    return 0;
}
```

## 9. Section 3: Basic knowledge about how to use SendProtocol()

More than 40 different Readers with together more than 65 commands, many of them individual adjustable with a mode byte, a lot of commands with subcommands and finally some commands with reader dependent content have to be represented by a simple communication process. This is the intention of the method `SendProtocol()`. It transfers only a single parameter: the command byte.

The principle use of `SendProtocol()` is following a schema represented by the picture below.



The horizontal axis shows the control flow that is generated by `SendProtocol()`. It independently retrieves all the necessary data from the integrated data containers before transmitting the send protocol and stores the received protocol data there as well. This means the application must write all the data necessary for this protocol to the corresponding data containers in the correct locations before invoking the `SendProtocol()`. Likewise the receive data are stored at particular locations in corresponding data containers.

In the vertical axis are the data streams which are moved using the overlaid methods:

GetData / SetData	<p>transfer methods for all common communication protocols. The key to the protocol data are so-called access constants and can be found in <code>FEDM_ISCReaderID.h</code> (C++) and <code>FedmlscReaderID</code> (Java, .NET).</p> <p>An example for an access constant is</p> <p>C++            <code>FEDM_ISC_TMP_READER_INFO_MODE</code></p> <p>Java/.NET    <code>FedmlscReaderID.FEDM_ISC_TMP_READER_INFO_MODE</code></p>
GetConfigPara / SetConfigPara	<p>transfer methods for modifying the Reader's configuration in the Reader object. Each configuration parameter is identified by an identifier string from a namespace (C++), interface (Java) or structure (.NET).</p> <p>An example for an identifier is:</p> <p>C++            <code>ReaderConfig::DigitalIO::Relay::No1::IdleMode</code></p> <p>Java/.NET    <code>ReaderConfig.DigitalIO.Relay.No1.IdleMode</code></p>
GetTableData / SetTableData	<p>transfer methods for table oriented commands in the Host-Mode, Buffered-Read-Mode and Notification-Mode. Each table element is identified by an constant and can be found in <code>FEDM_ISC.h</code> (C++) and <code>FedmlscReaderConst</code> (Java, .NET).</p> <p>An example for a constant is:</p> <p>C++            <code>FEDM_ISC_DATA_SNR</code></p>

Summarized: a bulk of Set methods have to be placed in front of `SendProtocol()`, followed optionally by a bulk of Get methods to retrieve the received data.

The following example shows the reading, modification and rewriting of one block of the reader configuration.



#### FEDM

```
int back = 0;
unsigned char ucCfgAdr = 2; // Address of the configuration block
bool bEEProm = false; // Configuration data from/in RAM of the reader
unsigned int uiIdleModeRel1 // Parameter IDLE-MODE
// settings for the next SendProtocol
reader.SetData(FEDM_ISC_TMP_READ_CFG, (unsigned char)0x00); // reset mode byte
reader.SetData(FEDM_ISC_TMP_READ_CFG_ADR, ucCfgAdr); // set address
reader.SetData(FEDM_ISC_TMP_READ_CFG_LOC, bEEProm); // set memory location on RAM
// read configuration data
back = reader.SendProtocol(0x80);
// evaluate back here!
uiIdleModeRel1 = 3; // REL1 alternating on
// (Note: set frequency in Parameter IDLE-FLASH)
// change configuration parameter in RAM
reader.SetConfigPara(ReaderConfig::DigitalIO::Relay::No1::IdleMode, uiIdleMode, false);
// settings for the next SendProtocol
reader.SetData(FEDM_ISC_TMP_WRITE_CFG, (unsigned char)0x00); // reset mode byte
reader.SetData(FEDM_ISC_TMP_WRITE_CFG_ADR, ucCfgAdr); // set address
reader.SetData(FEDM_ISC_TMP_WRITE_CFG_LOC, bEEProm); // set memory location on EEPROM
// rewrite configuration data
back = reader.SendProtocol(0x81);
```



```
import de.feig.*;
import de.feig.ReaderConfig.*;

int back = 0;
byte cfgAdr = 2; // Address of the configuration block
boolean eeProm = false; // Configuration data from/in RAM of the reader
int idleModeRel1 // Parameter IDLE-MODE

try
{
    // settings for the next SendProtocol
    reader.setData(FedmIsedReaderID.FEDM_ISC_TMP_READ_CFG, (byte)0x00); // reset mode byte
    reader.setData(FedmIsedReaderID.FEDM_ISC_TMP_READ_CFG_ADR, cfgAdr); // set address
    reader.setData(FedmIsedReaderID.FEDM_ISC_TMP_READ_CFG_LOC, eeProm); // RAM memory
    // read configuration data
    back = reader.sendProtocol(0x80);
    // evaluate back here!
    idleModeRel1 = 3; // REL1 alternating on
    // (Note: set frequency in Parameter IDLE-FLASH)
    // change configuration parameter in RAM
    reader.setConfigPara(DigitalIO.Relay.No1.IdleMode, idleMode, false);
    // settings for the next SendProtocol
    reader.setData(FedmIsedReaderID.FEDM_ISC_TMP_WRITE_CFG, (byte)0x00); // reset mode byte
    reader.setData(FedmIsedReaderID.FEDM_ISC_TMP_WRITE_CFG_ADR, cfgAdr); // set address
    reader.setData(FedmIsedReaderID.FEDM_ISC_TMP_WRITE_CFG_LOC, eeProm); // RAM memory
    // rewrite configuration data
    back = reader.sendProtocol((byte)0x81);
}
catch (Exception ex)
{
    ...
}
```

**C#**

```
using OBID;
using OBID.ReaderConfig;

int back = 0;
byte cfgAdr = 2;           // Address of the configuration block
boolean eeProm = false;    // Configuration data from/in RAM of the reader
int idleModeRel1           // Parameter IDLE-MODE

try
{
    // settings for the next SendProtocol
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_READ_CFG, (byte)0x00); // reset mode byte
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_READ_CFG_ADR, cfgAdr); // set address
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_READ_CFG_LOC, eeProm); // RAM memory
    // read configuration data
    back = reader.SendProtocol(0x80);
    // evaluate back here!
    idleModeRel1 = 3;        // REL1 alternating on
                           // (Note: set frequency in Parameter IDLE-FLASH)
    // change configuration parameter in RAM
    reader.SetConfigPara(DigitalIO.Relay.No1.IdleMode, idleMode, false);
    // settings for the next SendProtocol
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_WRITE_CFG, (byte)0x00); // reset mode byte
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_WRITE_CFG_ADR, cfgAdr); // set address
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_WRITE_CFG_LOC, eeProm); // RAM memory
    // rewrite configuration data
    back = reader.SendProtocol((byte)0x81);
}
catch (Exception ex)
{
    ...
}
```

## 10. Section 4: Read of important information from Reader

This section presents same valuable methods to simplify application programming.

### 10.1. Reader Information: The method ReadReaderInfo()

Immediately after establishing the connection - but only once - an application should read important information from the Reader to initialize the reader object with at least the reader type. The easiest and best way is to call the method `ReadReaderInfo()`. This method executes successive all [0x66] Reader Info commands to collect the complete reader information in a structure (C++) or class (Java, .NET) and to set the reader type for internal initializations.

The ReaderInfo structure/class collects all info fields of all Readers. Thus, the structure/class is very big. To get more clearness about the info fields, please have a quick look to the system manual of the Reader. The ReaderInfo structure/class is hold internally in the Reader object and can later be get with `GetReaderInfo()`.

Applications, based on Function Libraries up to FEISC needs not to read information from the Reader. Inside FEISC, nothing is to be initialized. But if an application should be fit for different RFID-Readers, at least the reader type should be read with the command [0x65] or [0x66] with mode 0x00 and be stored in a variable for later decisions.



C/C++

#### FEDM

```
FEDM_ISC_READER_INFO* info = NULL;

info = reader.ReadReaderInfo();
if(reader.GetLastError() == 0)
{
    // complete Reader info read and reader object initialized

    if(info->bIsMode0x00Read)
    {
        // all values for mode 0x00 are valid
        // info->ucReaderType contains the value of the reader type
    }
}
```

#### FEISC

```
int back = 0;
int iReaderHandle = 0;
unsigned char ucInfo[30];
unsigned char ucReaderType = 0;

// read mode 0x00 in binary format (last parameter is 0)
// the supported modes can be found in the system manual of the Reader
back = FEISC_0x66_ReaderInfo( iReaderHandle, 255, 0x00, ucInfo, 0);
if(back == 0)
{
    // OK
    // content of ucInfo is in the same order as described in the
    // system manual of the Reader
    ucReaderType = ucInfo[4];
}
```





```
import de.feig.*;

FedmIscReaderInfo info = null;

try
{
    reader.readReaderInfo();

    // complete Reader info read and reader object initialized

    if(info.isMode0x00Read)
    {
        // all values for mode 0x00 are valid
        // info.readerType contains the value of the reader type
    }
}
catch (Exception ex)
{
    ...
}
```



```
C#

using OBID;

FedmIscReaderInfo info = null;

try
{
    reader.ReadReaderInfo();

    // complete Reader info read and reader object initialized

    if(info.IsMode0x00Read)
    {
        // all values for mode 0x00 are valid
        // info.ReaderType contains the value of the reader type
    }
}
catch (Exception ex)
{
    ...
}
```

Nice to know, that the Java/.NET class FedmIscReaderInfo contains a method GetReport() to return a report string like it is used in ISOStart/CPRStart. C++ Programmers have to use the class FedmIscReport\_ReaderInfo to generate the report string.

## 10.2. Reader Diagnostic: The method ReadReaderDiagnostic()

Some Reader supports the query of diagnostic data. These diagnostic data can be valuable for analyzing problems inside the Reader or on the RF channel(s). More information about the diagnostic data can be found in the system manuals of the Readers.

The method ReadReaderDiagnostic executes successive all [0x6E] Reader Diagnostic commands to collect all available reader diagnostic data in a structure (C++) or class (Java, .NET).

Applications, based on Function Libraries up to FEISC have to read the diagnostic data in a loop of commands [0x6E] Reader Diagnostic with modes depends on the Reader type.



C/C++

### FEDM

```
FEDM_ISC_READER_DIAGNOSTIC* diag = NULL;

diag = reader.ReadReaderDiagnostic();
if(reader.GetLastError() == 0)
{
    // complete Reader diagnostic data read

    if(diag->bIsMode0x01Read)
    {
        // all values for mode 0x01 are valid
    }
}
```

### FEISC

```
int back = 0;
int iReaderHandle = 0;
unsigned char ucDiag[30];

// read mode 0x01
// the supported modes can be found in the system manual of the Reader
back = FEISC_0x6E_RdDiag( iReaderHandle, 255, 0x01, ucInfo);
if(back == 0)
{
    // OK
    // content of ucDiag is in the same order as described in the
    // system manual of the Reader
}
```



```
import de.feig.*;

// readReaderDiagnosotic is actually not implemented.
// alternatively, the diagnostic data can be read in a loop of the following sequence

byte[] diag = null;

try
{
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_DIAG_MODE, (byte)0x01);
    reader.sendProtocol((byte)0x6E);

    // diagnostic data read

    // OK
    // content of diag is in the same order as described in the
    // system manual of the Reader
    diag = reader.getByteArrayData(FedmIscReaderID.FEDM_ISC_TMP_DIAG_DATA);
}
catch (Exception ex)
{
    ...
}
```



```
C#

using OBID;

// ReadReaderDiagnosotic is actually not implemented.
// alternatively, the diagnostic data can be read in a loop of the following sequence

byte[] diag = null;

try
{
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_DIAG_MODE, (byte)0x01);
    reader.SendProtocol((byte)0x6E);

    // diagnostic data read

    // OK
    // content of diag is in the same order as described in the
    // system manual of the Reader
    reader.GetData(FedmIscReaderID.FEDM_ISC_TMP_DIAG_DATA, out );
}
catch (Exception ex)
{
    ...
}
```

### 10.3. Reader Configuration: The method ReadCompleteConfiguration()

Each OBID i-scan® and OBID® classic-pro Reader is controlled by parameters which are stored grouped in blocks in an EEPROM and are described in detail in the system manual for the respective Reader. After switching on or resetting the Reader, all parameters are loaded into RAM, evaluated and incorporated in the controller.

All parameters can be modified using a protocol so that the behavior of the Reader can be adapted to the application. Ideally, the program ISOStart/CPRStart is used for this adaptation and normally no parameters have to be changed in the application. Despite this, it can happen that one or more parameters from a program have to be changed.

A common characteristic of all Readers is the grouping in blocks of thematically related parameters to 14 or 30 bytes per configuration block. Each parameter cannot be addressed individually but must always be retrieved together with a configuration block using the protocol [0x80] or [0x8A] Read Configuration, then modified and finally written back to the reader with the protocol [0x81] or [0x8B] Write Configuration. This cycle must always be complied with and is also checked by the reader class. This means that writing a configuration block without previously reading the same block is not possible.

[17. Section 11: Management of the Reader configuration](#) discusses the programming for Reader configuration in more detail. But if your application has to modify the configuration, it is recommended to use the method ReadReaderConfiguration once after the connection is established and after ReadReaderInfo().



#### FEDM

```
int back = 0;

back = reader.ReadCompleteConfiguration(true); // read configuration from EEPROM
if(back == 0x13 || // Login-Request
   back == 0x19 ) // crypto processing error
{
    // login or authentication is required
    // and then read must be repeated
}
back = reader.ReadCompleteConfiguration(false); // read configuration from RAM
```



```
import de.feig.*;

int back = 0;

try
{
    back = reader.readCompleteConfiguration(true); // read configuration from EEPROM
    if(back == 0x13 || // Login-Request
       back == 0x19 ) // crypto processing error
    {
        // login or authentication is required
        // and then read must be repeated
    }
    back = reader.readCompleteConfiguration(false); // read configuration from RAM
}
catch (Exception ex)
{
    ...
}
```



```
C#

using OBID;

int back = 0;

try
{
    back = reader.ReadCompleteConfiguration(true); // read configuration from EEPROM
    if(back == 0x13 || // Login-Request
        back == 0x19 ) // crypto processing error
    {
        // login or authentication is required
        // and then read must be repeated
    }
    back = reader.ReadCompleteConfiguration(false); // read configuration from RAM
}
catch (Exception ex)
{
    ...
}
```

## 11. Section 5: Programming for the Host-Mode

The Host-Mode or Polling-Mode is the basic working mode and supported by all RFID-Readers. One of the main advantages is the use of an anti-collision algorithm to detect multiple Transponders with different RF-Protocols in one cycle.

### 11.1. Inventory

The Inventory is the most important command for Transponder identification in the RF field. The command [0xB0][0x01] Inventory is controlled by a mode byte and returns, if Transponders are found, one or multiple record sets. The structure of each record set depends on the Transponder type.

For the mode byte, the following flags are defined, but not all are applicable with each Reader type:

Bit Number	Flag	Notes
7	MORE	query of more data from last Inventory cycle
6	NTFC	Notification (only <i>classic-pro</i> Reader)
5	PRESC	no Inventory, only presence check (only <i>classic-pro</i> Reader)
4	ANT	request of additional antenna information (only <i>i-scan</i> Reader)
3	-	
2	-	
1	-	
0	-	

The received record sets are copied into the internal ISOTable, where each Transponder record resides in one table item. The following Host commands (in addressed and selected mode) are then based on this table item and new Transponder data like read data blocks are added. Also data blocks to be written must first be set into the table item before the Host command can be executed.

A table item has data members for all ISO 15693, ISO 14443 and EPC Class1 Gen2 compliant Transponders. Thus, not every table element is applicable with every Transponder type.

The following table collects some important table items returned by an Inventory (all elements can be viewed by documentation or C++ code):

Element	Reader Support	Notes	
SNR or UID	all	Serial Number or UID or IDD with variable length of up to 96 bytes	
SNR-Length	all	length of SNR/UID in number of bytes	
TrType	all	Transponder type according the system manuals of the Readers	
		TrType	Transponder Type
		0x00	NXP I-Code1
		0x01	National Instruments Tag-it
		0x03	Transponder according ISO15693

		0x04	Transponder according ISO14443A
		0x05	Transponder according ISO14443B
		0x06	NXP I-Code EPC
		0x07	NXP I-Code UID
		0x08	Jewel
		0x09	ISO 18000-3M3
		0x0A	STMicroelectronics SR176
		0x0B	STMicroelectronics SRIxx (SRI512, SRIX512, SRI4K, SRIX4K)
		0x0C	Microchip MCRFxxx
		0x10	Innovatron (ISO 14443B')
		0x11	ASK CTx
		0x80	ISO18000-6 A
		0x81	ISO18000-6 B (UCODE; UCODE EPC 1.19)
		0x83	EM4222, EM4444
		0x84	EPC class 1 Gen 2
Rx Data	ISO 15693 / ISO 14443	Receive data block with variable block size	
Tx Data	ISO 15693 / ISO 14443	Transmit data block with variable block size	
Rx EPC-Bank	EPC Class1 Gen2	Receive data block for EPC memory with block size 2	
Tx EPC-Bank	EPC Class1 Gen2	Transmit data block for EPC memory with block size 2	
Rx TID-Bank	EPC Class1 Gen2	Receive data block for TID memory with block size 2	
Tx TID-Bank	EPC Class1 Gen2	Transmit data block for TID memory with block size 2	
Rx Res-Bank	EPC Class1 Gen2	Receive data block for Reserved memory with block size 2	
Tx Res-Bank	EPC Class1 Gen2	Transmit data block for Reserved memory with block size 2	
AFI	ISO 15693	Application Family Identifier	
DSFID	ISO 15693	Data Storage Family Identifier	
IDDT	EPC Class1 Gen2	Identifier Data Type	
TrInfo	ISO 14443 A	Transponder Info	
OptInfo	ISO 14443 A	Optional Information	
ProtoInfo	ISO 14443 B	Protocol Information	
the following elements are antenna specific records (ANT flag must be set in Mode)			
Flags	ISO 15693/EPC Class1 Gen2	Record information	
AntCount	ISO 15693/EPC Class1 Gen2	Number of antenna records	
AntNumber[]	ISO 15693/EPC Class1 Gen2	array with antenna numbers	
AntStatus[]	ISO 15693/EPC Class1 Gen2	array with antenna status	
AntRSSI[]	ISO 15693/EPC Class1 Gen2	array with RSSI values	

**FEDM**

```
int back = 0;
unsigned char ucTrType = 0;
string sUId;

// settings for the next SendProtocol
reader.SetData(FEDM_ISC_TMP_B0_CMD, (unsigned char)0x01); // sub command Inventory
reader.SetData(FEDM_ISC_TMP_B0_MODE, (unsigned char)0x00); // no option flags set

// clear internal ISOtable for next Inventory
reader.ResetTable(FEDM_ISC_ISO_TABLE);

// execute Inventory
back = reader.SendProtocol(0xB0);
if(back == 0x00)
{
    // query table data
    for(int idx=0; idx<reader.GetTableLength(); idx++)
    {
        reader.GetTableData(idx, FEDM_ISC_ISO_TABLE, FEDM_ISC_DATA_TRTYPE, &ucTrType);
        reader.GetTableData(idx, FEDM_ISC_ISO_TABLE, FEDM_ISC_DATA_SNR, sUId);
    }
}
```





```
import de.feig.*;

int back = 0;
byte trType = 0;
String uid;

try
{
    // settings for the next sendProtocol
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_CMD, (byte)0x01); // sub command
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_MODE, (byte)0x00); // no option flags

    // clear internal ISOtable for next Inventory
    reader.resetTable(FedmIscReaderConst.ISO_TABLE);

    // execute Inventory
    back = reader.sendProtocol((byte)0xB0);
    if(back == 0x00)
    {
        // query table data
        for(int idx=0; idx<reader.getTableLength(); idx++)
        {
            reader.getBytesTableData(idx, FedmIscReaderConst.ISO_TABLE,
                                      FedmIscReaderConst.DATA_TRTYPE);

            uid = reader.getStringTableData( idx, FedmIscReaderConst.ISO_TABLE,
                                             FedmIscReaderConst.DATA_SNR);
        }
    }
}
catch (Exception ex)
{
    ...
}
```



```
C#

using OBID;

int back = 0;
byte trType = 0;
string uid;

try
{
    // settings for the next SendProtocol
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_CMD, (byte)0x01); // sub command
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_MODE, (byte)0x00); // no option flags

    // clear internal ISOtable for next Inventory
    reader.ResetTable(FedmIscReaderConst.ISO_TABLE);

    // execute Inventory
    back = reader.SendProtocol((byte)0xB0);
    if(back == 0x00)
    {
        // query table data
        for(int idx=0; idx<reader.GetTableLength(); idx++)
        {
            reader.getTableData(idx, FedmIscReaderConst.ISO_TABLE, FedmIscReaderConst.DATA_TRTYPE,
                                out trType);

            reader.getTableData(idx, FedmIscReaderConst.ISO_TABLE, FedmIscReaderConst.DATA_SNR,
                                out uid);
        }
    }
}
catch (Exception ex)
{
    ...
}
```

## 11.2. Read / Write Transponder data

All OBID® RFID-Readers support the reading and writing of transponder data. Mostly, the normal addressed mode is implemented, which means, that the block address has a size of one byte and therefore the highest address is limited to 255.

Some few Readers have additionally the extended addressed mode implemented with which the block address is two bytes wide and the UID can have a variable length. For OBID® i-scan UHF, the extended address mode supports also different memory banks and an access password.

The examples below reads some data blocks from the previously detected Transponder, and writes the same data blocks back again. The first block address is 5 and 4 data blocks will be read. After the read operation, the block size of data blocks is returned by the Transponder and stored in the internal ISOTable. The block size is essential for calculating the returned number of bytes.

### 11.2.1. Normal addressed mode



#### FEDM

```
int back = 0;
int idx = 0;
unsigned char ucBlockSize = 0;
unsigned char ucData[16]; // buffer for 4 data blocks of each 4 bytes
string sUId;

// request UID from anywhere
// settings for the next SendProtocol
reader.SetData(FEDM_ISC_TMP_B0_CMD, (unsigned char)0x23); // sub command
reader.SetData(FEDM_ISC_TMP_B0_MODE, (unsigned char)0x01); // addressed mode
reader.SetData(FEDM_ISC_TMP_B0_REQ_UID, sUId); // UID for addressed mode
reader.SetData(FEDM_ISC_TMP_B0_REQ_DB_ADR, (unsigned char)5); // first block address
reader.SetData(FEDM_ISC_TMP_B0_REQ_DBN, (unsigned char)4); // number of blocks to be read

// execute Read Multiple Blocks
back = reader.SendProtocol(0xB0);
if(back == 0x00)
{
    // find the table item for specific UID
    idx = reader.FindTableIndex(0, FEDM_ISC_ISO_TABLE, FEDM_ISC_DATA_SNR, sUId);
    if(idx >= 0)
    {
        // query blocksize from table
        reader.GetTableData(idx, FEDM_ISC_ISO_TABLE, FEDM_ISC_DATA_BLOCKSIZE, ucBlockSize);
        // query table data from rx buffer
        for(int i=0; i<4; i++)
            reader.GetTableData(idx, FEDM_ISC_ISO_TABLE, FEDM_ISC_DATA_RxDB, 5,
                                &ucData[i*ucBlockSize], 4);
        // set data into tx buffer
        for(int i=0; i<4; i++)
            reader.SetTableData(idx, FEDM_ISC_ISO_TABLE, FEDM_ISC_DATA_TxDB, 5,
                                &ucData[i*ucBlockSize], ucBlockSize);

        // execute Write Multiple Blocks
        reader.SetData(FEDM_ISC_TMP_B0_CMD, (unsigned char)0x24); // sub command
        back = reader.SendProtocol(0xB0);
    }
}
```



```
import de.feig.*;

int back = 0;
int idx = 0;
byte blockSize = 0;
byte[] data = null;
String uid;

// request UID from anywhere

try
{
    // settings for the next sendProtocol
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_CMD, (byte)0x23); // sub command
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_MODE, (byte)0x01); // addressed mode
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_REQ_UID, uid); // UID for addressed mode
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_REQ_DB_ADR, (byte)5); // first block addr
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_REQ_DBN, (byte)4); // number of blocks
    // execute Read Multiple Blocks
    back = reader.sendProtocol((byte)0xB0);
    if(back == 0x00)
    {
        // find the table item for specific UID
        idx = reader.findTableIndex( 0, FedmIscReaderConst.ISO_TABLE, FedmIscReaderConst.DATA_SNR,
                                    uid);

        if(idx >= 0)
        {
            // query blocksize from table
            blockSize = reader.getBytesTableData( idx, FedmIscReaderConst.ISO_TABLE,
                                                  FedmIscReaderConst.DATA_BLOCKSIZE);

            // query table data from rx buffer
            data = reader.getBytesArrayTableData( idx, FedmIscReaderConst.ISO_TABLE,
                                                  FedmIscReaderConst.DATA_RxDB, 5, 4);

            // set data into tx buffer
            reader.setTableData(idx, FedmIscReaderConst.ISO_TABLE, FedmIscReaderConst.DATA_TxDB,
                               5, 4, blockSize, data);

            // execute Write Multiple Blocks
            reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_CMD, (byte)0x24); // sub command
            back = reader.sendProtocol((byte)0xB0);
        }
    }
}
catch (Exception ex)
{
    ...
}
```



```

C#

using OBID;

int back = 0;
int idx = 0;
byte blockSize = 0,
byte[] data = null;
string uid;

try
{
    // settings for the next sendProtocol
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_CMD, (byte)0x23); // sub command
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_MODE, (byte)0x01); // addressed mode
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_REQ_UID, uid); // UID for addressed mode
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_REQ_DB_ADR, (byte)5); // first block addr
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_REQ_DBN, (byte)4); // number of blocks
    // execute Read Multiple Blocks
    back = reader.SendProtocol((byte)0xB0);
    if(back == 0x00)
    {
        // find the table item for specific UID
        idx = reader.FindTableIndex( 0, FedmIscReaderConst.ISO_TABLE, FedmIscReaderConst.DATA_SNR,
                                    uid);

        if(idx >= 0)
        {
            // query blocksize from table
            reader.GetTableData(idx, FedmIscReaderConst.ISO_TABLE,
                               FedmIscReaderConst.DATA_BLOCKSIZE, out blockSize);

            // query table data from rx buffer
            reader.GetTableData(idx, FedmIscReaderConst.ISO_TABLE,
                               FedmIscReaderConst.DATA_RxDB, 5, 4, out data);

            // set data into tx buffer
            reader.SetTableData(idx, FedmIscReaderConst.ISO_TABLE, FedmIscReaderConst.DATA_TxDB,
                               5, 4, blockSize, data);

            // execute Write Multiple Blocks
            reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_CMD, (byte)0x24); // sub command
            back = reader.SendProtocol((byte)0xB0);
        }
    }
}
catch (Exception ex)
{
    ...
}

```

### 11.2.2. Extended addressed mode

If using Transponders with a UID length not equal 8 or Transponders with larger memory and a number of data blocks greater than 256, the Extended Addressed Mode must be enabled in the request protocol.

It must be considered, that the ISOTable is configured by default for maximal 256 data blocks with 32 bytes in each block to optimize memory consumption. For Transponders with more than 256 data blocks, the internal buffer sizes of the ISOTable must be modified with the method:

```
SetTableSize(unsigned int uiTableID,int iSize,
             int iRxDB_BlockCount,int iRxDB_BlockSize,
             int iTxDB_BlockCount,int iTxDB_BlockSize).
```

This should be done only once after initializing of the reader object.



#### FEDM

```
int back = 0;
int idx = 0;
unsigned char ucBlockSize = 0;
unsigned char ucData[16]; // buffer for 4 data blocks of each 4 bytes
string sUId;

// request UID from anywhere
// settings for the next SendProtocol
reader.SetData(FEDM_ISC_TMP_B0_CMD, (unsigned char)0x23); // sub command
reader.SetData(FEDM_ISC_TMP_B0_MODE, (unsigned char)0x00); // clear mode byte
reader.SetData(FEDM_ISC_TMP_B0_MODE_ADR, (unsigned char)0x01); // addressed mode
reader.SetData(FEDM_ISC_TMP_B0_MODE_EXT_ADR, true); // extended addressed mode
reader.SetData(FEDM_ISC_TMP_B0_MODE_UID_LEN, true); // UID with variable length
reader.SetData(FEDM_ISC_TMP_B0_REQ_UID, sUId); // UID for addressed mode
reader.SetData(FEDM_ISC_TMP_B0_REQ_UID_LEN, sUId.length()/2); // length of UID in number of bytes
reader.SetData(FEDM_ISC_TMP_B0_BANK, (unsigned char)0x00); // clear bank byte
reader.SetData(FEDM_ISC_TMP_B0_BANK_BANK_NR, (unsigned char)0x03); // user memory bank
reader.SetData(FEDM_ISC_TMP_B0_REQ_DB_ADR_EXT, (unsigned int)5); // first block address
reader.SetData(FEDM_ISC_TMP_B0_REQ_DBN, (unsigned char)4); // number of blocks to be read

// execute Read Multiple Blocks
back = reader.SendProtocol(0xB0);
if(back == 0x00)
{
    // find the table item for specific UID
    idx = reader.FindTableIndex(0, FEDM_ISC_ISO_TABLE, FEDM_ISC_DATA_SNR, sUId);
    if(idx >= 0)
    {
        // query blocksize from table
        reader.GetTableData(idx, FEDM_ISC_ISO_TABLE, FEDM_ISC_DATA_BLOCKSIZE, ucBlockSize);
        // query table data from rx buffer
        for(int i=0; i<4; i++)
            reader.GetTableData(idx, FEDM_ISC_ISO_TABLE, FEDM_ISC_DATA_RxDB, 5,
                               &ucData[i*ucBlockSize], 4);
        // set data into tx buffer
        for(int i=0; i<4; i++)
            reader.SetTableData(idx, FEDM_ISC_ISO_TABLE, FEDM_ISC_DATA_TxDB, 5,
                               &ucData[i*ucBlockSize], ucBlockSize);

        // execute Write Multiple Blocks
        reader.SetData(FEDM_ISC_TMP_B0_CMD, (unsigned char)0x24); // sub command
        back = reader.SendProtocol(0xB0);
    }
}
```



```
import de.feig.*;

int back = 0;
int idx = 0;
byte blockSize = 0,
byte[] data = null;
String uid;

// request UID from anywhere

try
{
    // settings for the next sendProtocol
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_CMD, (byte)0x23); // sub command
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_MODE, (byte)0x00); // clear mode byte
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01); // addressed mode
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_MODE_EXT_ADR, true); // extended addressed mode
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_MODE_UID_LF, true); // UID with variable length
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_REQ_UID, uid); // UID for addressed mode
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_REQ_UID_LEN, uid.length()/2); // length of UID
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_BANK, (byte)0x00); // clear bank byte
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_BANK_BANK_NR, (byte)0x03); // user memory bank
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_REQ_DB_ADR_EXT, (int)5); // first block address
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_REQ_DBN, (byte)4); // number of blocks
    // execute Read Multiple Blocks
    back = reader.sendProtocol((byte)0xB0);
    if(back == 0x00)
    {
        // find the table item for specific UID
        idx = reader.findTableIndex( 0, FedmIscReaderConst.ISO_TABLE, FedmIscReaderConst.DATA_SNR,
                                    uid);

        if(idx >= 0)
        {
            // query blocksize from table
            blockSize = reader.getByteTableData( idx, FedmIscReaderConst.ISO_TABLE,
                                                FedmIscReaderConst.DATA_BLOCKSIZE);

            // query table data from rx buffer
            data = reader.getByteArrayTableData( idx, FedmIscReaderConst.ISO_TABLE,
                                                FedmIscReaderConst.DATA_RxDB, 5, 4);

            // set data into tx buffer
            reader.setTableData(idx, FedmIscReaderConst.ISO_TABLE, FedmIscReaderConst.DATA_TxDB,
                               5, 4, blockSize, data);

            // execute Write Multiple Blocks
            reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_CMD, (byte)0x24); // sub command
            back = reader.sendProtocol((byte)0xB0);
        }
    }
}
catch (Exception ex)
{
    ...
}
```

**C#**

```

using OBID;

int back = 0;
int idx = 0;
byte blockSize = 0,
byte[] data = null;
string uid;

try
{
    // settings for the next sendProtocol
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_CMD, (byte)0x23); // sub command
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_MODE, (byte)0x00); // clear mode byte
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01); // addressed mode
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_MODE_EXT_ADR, true); // extended addressed mode
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_MODE_UID_LF, true); // UID with variable length
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_REQ_UID, uid); // UID for addressed mode
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_REQ_UID_LEN, uid.Length/2); // length of UID
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_BANK, (byte)0x00); // clear bank byte
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_BANK_BANK_NR, (byte)0x03); // user memory bank
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_REQ_DB_ADR_EXT, (int)5); // first block address
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_REQ_DBN, (byte)4); // number of blocks
    // execute Read Multiple Blocks
    back = reader.SendProtocol((byte)0xB0);
    if(back == 0x00)
    {
        // find the table item for specific UID
        idx = reader.FindTableIndex( 0, FedmIscReaderConst.ISO_TABLE, FedmIscReaderConst.DATA_SNR,
                                     uid);

        if(idx >= 0)
        {
            // query blocksize from table
            reader.GetTableData(idx, FedmIscReaderConst.ISO_TABLE,
                               FedmIscReaderConst.DATA_BLOCKSIZE, out blockSize);

            // query table data from rx buffer
            reader.GetTableData(idx, FedmIscReaderConst.ISO_TABLE,
                               FedmIscReaderConst.DATA_RxDB, 5, 4, out data);

            // set data into tx buffer
            reader.SetTableData(idx, FedmIscReaderConst.ISO_TABLE, FedmIscReaderConst.DATA_TxDB,
                               5, 4, blockSize, data);

            // execute Write Multiple Blocks
            reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_CMD, (byte)0x24); // sub command
            back = reader.SendProtocol((byte)0xB0);
        }
    }
}
catch (Exception ex)
{
    ...
}

```

### 11.3. Excursion: Inventory with multiple antennas

The standard Inventory command operates with one antenna. This covers the capability of the most RFID-Readers. For Readers with multiple antenna connectors and an internal multiplexer, the Inventory command is extended to operate with specified antennas. This option can be switched on with a flag in the mode byte.

The antenna number, where the Transponder is detected, is reflected in the returned data records.



#### FEDM

```
int back = 0;
unsigned char ucTrType = 0;
string sUId;
FEDM_ISOTabItem* pTabItem = NULL;

// settings for the next SendProtocol
reader.SetData(FEDM_ISC_TMP_B0_CMD, (unsigned char)0x01); // sub command Inventory
reader.SetData(FEDM_ISC_TMP_B0_MODE, (unsigned char)0x00); // clear mode byte
reader.SetData(FEDM_ISC_TMP_B0_MODE_ANT, true); // enable multiple antenna support
reader.SetData(FEDM_ISC_TMP_B0_REQ_ANT_SEL, (unsigned char)0x03); // antenna 1 and 2

// clear internal ISOTable for next Inventory
reader.ResetTable(FEDM_ISC_ISO_TABLE);

// execute Inventory
back = reader.SendProtocol(0xB0);
if(back == 0x00)
{
    // query table data
    for(int idx=0; idx<reader.GetTableLength(); idx++)
    {
        reader.GetTableData(idx, FEDM_ISC_ISO_TABLE, FEDM_ISC_DATA_TRTYPE, &ucTrType);
        reader.GetTableData(idx, FEDM_ISC_ISO_TABLE, FEDM_ISC_DATA_SNR, sUId);

        // additional antenna related data are only accessible with the table object
        pTabItem = reader->GetISOTableItem((unsigned int)idx);
        if(pTabItem == NULL)
            continue;

        for(unsigned int uiCnt=0; uiCnt<pTabItem->m_ucAntCount; uiCnt++)
        {
            // do anything with pTabItem->m_ucAntNumber[uiCnt];
            // do anything with pTabItem->m_ucAntStatus[uiCnt];
            // do anything with pTabItem->m_ucAntRSSI[uiCnt];
        }
    }
}
```





```

import de.feig.*;

int back = 0;
byte trType = 0;
String uid;
FedmIsoTableItem tabItem = null;

try
{
    // settings for the next sendProtocol
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_CMD, (byte)0x01); // sub command
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_MODE, (byte)0x00); // no option flags
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_MODE_ANT, true); // enable multiple antenna support
    reader.setData(FedmIscReaderID.FEDM_ISC_TMP_B0_REQ_ANT_SEL, (byte)0x03); // antenna 1 and 2

    // clear internal ISOTable for next Inventory
    reader.resetTable(FedmIscReaderConst.ISO_TABLE);

    // execute Inventory
    back = reader.sendProtocol((byte)0xB0);
    if(back == 0x00)
    {
        // query table data
        for(int idx=0; idx<reader.getTableLength(); idx++)
        {
            reader.getBytesTableData(idx, FedmIscReaderConst.ISO_TABLE,
                                      FedmIscReaderConst.DATA_TRTYPE);

            uid = reader.getStringTableData( idx, FedmIscReaderConst.ISO_TABLE,
                                             FedmIscReaderConst.DATA_SNR);

            // additional antenna related data are only accessible with the table object
            tabItem = (FedmIsoTableItem)reader.getTableItem(idx, FedmIscReaderConst.ISO_TABLE);
            if(tabItem == NULL)
                continue;

            for(int cnt=0; cnt<(int)tabItem.antCount; cnt++)
            {
                // do anything with tabItem.antNumber[cnt];
                // do anything with tabItem.antStatus[cnt];
                // do anything with tabItem.antRSSI[cnt];
            }
        }
    }
}
catch (Exception ex)
{
    ...
}

```

**C#**

```

using OBID;

int back = 0;
byte trType = 0;
string uid;
FedmIsoTableItem tabItem = null;

try
{
    // settings for the next SendProtocol
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_CMD, (byte)0x01); // sub command
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_MODE, (byte)0x00); // no option flags
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_MODE_ANT, true); // enable multiple antenna support
    reader.SetData(FedmIscReaderID.FEDM_ISC_TMP_B0_REQ_ANT_SEL, (byte)0x03); // antenna 1 and 2

    // clear internal ISOtable for next Inventory
    reader.ResetTable(FedmIscReaderConst.ISO_TABLE);

    // execute Inventory
    back = reader.SendProtocol((byte)0xB0);
    if(back == 0x00)
    {
        // query table data
        for(int idx=0; idx<reader.GetTableLength(); idx++)
        {
            reader.getTableData(idx, FedmIscReaderConst.ISO_TABLE, FedmIscReaderConst.DATA_TRTYPE,
                                out trType);

            reader.getTableData(idx, FedmIscReaderConst.ISO_TABLE, FedmIscReaderConst.DATA_SNR,
                                out uid);

            // additional antenna related data are only accessible with the table object
            tabItem = (FedmIsoTableItem)reader.getTableItem(idx, FedmIscReaderConst.ISO_TABLE);
            if(tabItem == NULL)
                continue;

            for(int cnt=0; cnt<(int)tabItem.antCount; cnt++)
            {
                // do anything with tabItem.antNumber[cnt];
                // do anything with tabItem.antStatus[cnt];
                // do anything with tabItem.antRSSI[cnt];
            }
        }
    }
}
catch (Exception ex)
{
    ...
}

```

---

#### 11.4. The application ISOHostSample

---

Preliminary

## 12. Section 6: Using TagHandler classes with Host-Mode

Programmers have two alternatives in the Reader class FEDM\_ISCReaderModule (C++) resp. FedmlscReader (Java, .NET) for the communication with Transponders:

1. Table oriented API (s. [11. Section 5: Programming for the Host-Mode](#))
2. TagHandler API, based on specialized Transponder classes

It is recommended to use the 2nd API for new projects. TagHandler classes are specific to Transponder standards like ISO 14443, ISO 15693 and EPC Class 1 Gen 2 or customized for manufacturer specific extended API. Each standard and chip type is implemented as a class and all classes together build a hierarchical system of derived classes. Base class is FedmlscTagHandler.

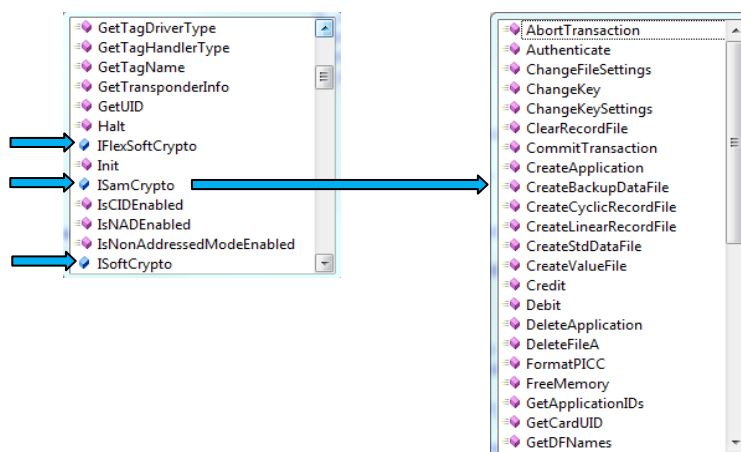
Precondition for the use of TagHandler classes is 1<sup>st</sup> the use of the methods TagInventory and TagSelect from the Reader class and 2<sup>nd</sup> the identifiability of the Transponder standard and/or chip type for the accurate creation of TagHandler classes. Unsupported chip types are assigned automatically to the base class FedmlscTagHandler.

TagHandler objects are managed by the ISOTable. Thus, they use internally the table oriented API.

The method interface of each TagHandler class is made up of the command list of Transponder standards or chip types. Consequently, the programmer has to work with the documentation of a standard or with the Transponder manual from the manufacturer to understand the meaning of the method interfaces.

### 12.1. Benefit

The picture below demonstrates with the example of the ISO 14443-A Transponder MIFARE DESFire the method interface (left) of the TagHandler class and, after selection of the internal interface – here: ISamCrypto – the real method interface of the Transponder. This API corresponds nearly to the manual of the manufacturer and the programmer can apply the operations directly with less code lines.



## 12.2. Inventory and Select

The samples demonstrate the benefit when using the TagHandler concept. The number of code lines is dramatically reduced against the use of the Table oriented API and the impact on the programmer friendly Transponder-API is obvious. The samples support different Transponder types in the RF field. One can see that the program flow for all Transponder types is identical.



### FEDM

```
int back = 0;
unsigned int uiBlockSize = 0;
unsigned int uiRspDataLen = 0;
unsigned char ucData[16] // buffer for 4 data blocks of each 4 bytes
FedmIscTagHandler* pTagHandler = NULL;
FEDM_ISC_TAG_LIST* pTagList = NULL;
FedmIscTagHandler_ISO14443_4_MIFARE_DESFire* pDesFire = NULL;
FEDM_ISC_TAG_LIST_ITOR itor;

// Inventory command with standard options
pTagList = reader.TagInventory();

for(itor = pTagList->begin(); itor != pTagList->end(); itor++)
{
    pTagHandler = itor->second;

    // check specialized tag handler
    if(dynamic_cast<FedmIscTagHandler_ISO15693*>(pTagHandler) != NULL)
    {
        FedmIscTagHandler_ISO15693* th = (FedmIscTagHandler_ISO15693*)pTagHandler;
        // read 4 datablocks from block address 5 and write same data back
        back = th->ReadMultipleBlocks(5, 4, uiBlockSize, ucData);
        back = th->WriteMultipleBlocks(5, 4, uiBlockSize, ucData);
    }
    else if(dynamic_cast<FedmIscTagHandler_ISO14443*>(pTagHandler) != NULL)
    {
        // execute a select command for MIFARE DESFire
        // TagSelect creates a new TagHandler object internally
        pTagHandler = reader.tagSelect(pTagHandler, 9);

        if(dynamic_cast<FedmIscTagHandler_ISO14443_4_MIFARE_DESFire*>(pTagHandler) != NULL)
        {
            FedmIscTagHandler_ISO14443_4_MIFARE_DESFire* th =
                (FedmIscTagHandler_ISO14443_4_MIFARE_DESFire*)pTagHandler;
            // read version information
            // use of the internal Interface IFlexSoftCrypto
            back = th->IFlexSoftCrypto.GetVersion(0, ucData, 16, uiRspDataLen);
        }
    }
    else if(dynamic_cast<FedmIscTagHandler_EPC_Class1_Gen2*>(pTagHandler) != NULL)
    {
        FedmIscTagHandler_EPC_Class1_Gen2* th = (FedmIscTagHandler_EPC_Class1_Gen2*)pTagHandler;
        // write new EPC to Transponder (without Password)
        th->WriteEPC("0102030405060708090A0B0C", "");
    }
}
```



```

import de.feig.*;
import de.feig.TagHandler.*;

int back = 0;
HashMap<String, FedmIsctagHandler> mapTH = null;
FedmIsctagHandler tagHandler = null;
FedmIsctagHandler_Result res = new FedmIsctagHandler_Result();// object containing result values

try
{
    // Inventory with standard options
    mapTH = reader.tagInventory(true, (byte)0, (byte)1);
    for( Map.Entry<String, FedmIsctagHandler> e : mapTH.entrySet() )
    {
        TagHandler = e.getValue(); // TagHandler from HashMap

        if(TagHandler instanceof FedmIsctagHandler_ISO15693)
        {
            FedmIsctagHandler_ISO15693 th = (FedmIsctagHandler_ISO15693) TagHandler;
            // read datablocks and write same data back
            back = th.readMultipleBlocks(4, 4, res);
            back = th.writeMultipleBlocks(4, 4, res.blockSize, res.data);
        }
        else if(TagHandler instanceof FedmIsctagHandler_ISO14443)
        {
            // execute a select command for MIFARE DESFire
            // TagSelect creates a new TagHandler object internally
            TagHandler = reader.tagSelect(TagHandler, 9);

            if(TagHandler instanceof FedmIsctagHandler_ISO14443_4_MIFARE_DESFire)
            {
                FedmIsctagHandler_ISO14443_4_MIFARE_DESFire th =
                    (FedmIsctagHandler_ISO14443_4_MIFARE_DESFire) TagHandler;
                // read version information
                // use of the internal Interface IFlexSoftCrypto
                back = th.IFlexSoftCrypto.getVersion((byte)0, res);
            }
        }
        else if(TagHandler instanceof FedmIsctagHandler_EPC_Class1_Gen2)
        {
            FedmIsctagHandler_EPC_Class1_Gen2 th = (FedmIsctagHandler_EPC_Class1_Gen2) TagHandler;
            // write new EPC to Transponder (without Password)
            th.writeEPC("0102030405060708090A0B0C", "");
        }
    }
}
catch (Exception ex)
{
    ...
}

```

**C#**

```

using OBID;
using OBID.TagHandler;

int back = 0;
byte blockSize = 0;
byte[] data = null;
Dictionary<string, FedmIscTagHandler> TagList;
Dictionary<string, FedmIscTagHandler>.ValueCollection listTagHandler;
FedmIscTagHandler TagHandler = null;

// Inventory command with standard options
TagList = reader.TagInventory(true, 0x00, 1));

if(TagList.Count > 0)
{
    listTagHandler = TagList.Values;
    foreach(FedmIscTagHandler tagHandler in listTagHandler)
    {
        if(tagHandler != null)
        {
            TagHandler = tagHandler;

            if(TagHandler is FedmIscTagHandler_ISO15693)
            {
                FedmIscTagHandler_ISO15693 th = (FedmIscTagHandler_ISO15693)TagHandler;
                // read datablocks and write same data back
                back = th.ReadMultipleBlocks(4, 4, out blockSize, out data);
                back = th.WriteMultipleBlocks(4, 4, blockSize, data);
            }
            else if(TagHandler is FedmIscTagHandler_ISO14443)
            {
                // execute a select command for MIFARE DESFire
                // TagSelect creates a new TagHandler object internally
                TagHandler = reader.TagSelect(TagHandler, 9);

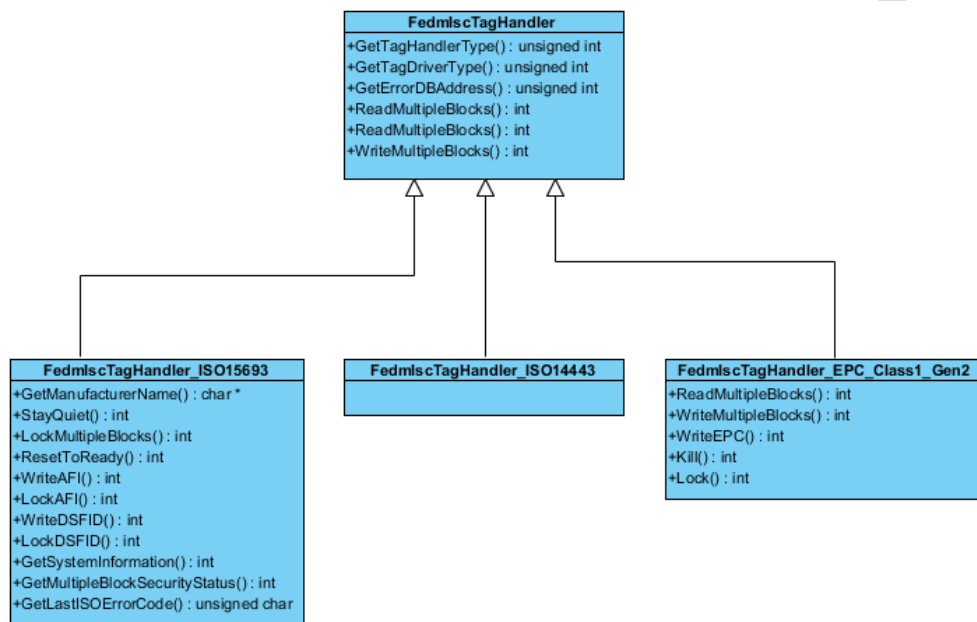
                if(TagHandler is FedmIscTagHandler_ISO14443_4_MIFARE_DESFire)
                {
                    FedmIscTagHandler_ISO14443_4_MIFARE_DESFire th =
                        (FedmIscTagHandler_ISO14443_4_MIFARE_DESFire)TagHandler;
                    // read version information
                    // use of the internal Interface IFlexSoftCrypto
                    back = th.IFlexSoftCrypto.GetVersion((byte)0, out data);
                }
            }
            else if(TagHandler is FedmIscTagHandler_EPC_Class1_Gen2)
            {
                FedmIscTagHandler_EPC_Class1_Gen2 th = (FedmIscTagHandler_EPC_Class1_Gen2)TagHandler;
                // write new EPC to Transponder (without Password)
                th.WriteEPC("0102030405060708090A0B0C", "");
            }
        }
    }
}

```

### 12.3. TagHandler classes

As mentioned above, all TagHandler classes together build a hierarchical system of derived classes with FedmiscTagHandler as base class. This is based on the fact that a Transponder with a manufacturer specific API extension is always based on a standard like ISO 15693 or ISO 14443. It is therefore consequent to derive such a proxy class from a class representing the ISO standard.

The picture below shows the first level of derivation with the base class, the derived classes representing standards and its method interfaces.



#### 12.3.1. Life Cycle of TagHandler objects

TagHandler objects are created with the call of `TagInventory()` and, for some ISO 14443 standard Transponders, with a call of `TagSelect()`. Provided that the next detected Transponder type in the same table index is identical, the TagHandler object is not destroyed, but initialized. This handling prevents continuous memory allocation.

#### 12.3.2. Naming Conventions

The name of a TagHandler class begins always with `FedmiscTagHandler` followed by the standard (like ISO 15693). Manufacturer specific extensions are reflected in the class name with a postfix of a shortening of the manufacturer name followed by the chip type.



### 12.3.3. Base class FedmlscTagHandler

The base class implements all methods which are common for all Transponder standards, like Read- and WriteMultipleBlocks. But if the handling of these both methods must be adapted or the parameter field must be extended, these methods are overwritten from specialized TagHandler classes.

Each TagHandler class can be identified at run-time with class identification operations (*dynamic\_cast* with C++, *instanceof* with Java and *is* with C#) or with a constant returned by the method `GetTagHandlerType()`. All TagHandler type constants are defined in the base class.

FedmlscTagHandler
+TYPE_BASIC : unsigned int = 1
+TYPE_EPC_CLASS1_GEN2 : unsigned int = 10
+TYPE_EPC_CLASS1_GEN2_IDS_SL900A : unsigned int = 11
+TYPE_ISO14443 : unsigned int = 20
+TYPE_ISO14443_2 : unsigned int = 30
+TYPE_ISO14443_2_INNOVISION_JEWEL : unsigned int = 31
+TYPE_ISO14443_2_STM_SR176 : unsigned int = 32
+TYPE_ISO14443_2_STM_SR1xxx : unsigned int = 33
+TYPE_ISO14443_3 : unsigned int = 40
+TYPE_ISO14443_3_INFINEON_MY_D : unsigned int = 41
+TYPE_ISO14443_3_INFINEON_MY_D_MOVE : unsigned int = 42
+TYPE_ISO14443_3_MIFARE_CLASSIC : unsigned int = 43
+TYPE_ISO14443_3_MIFARE_ULTRALIGHT : unsigned int = 44
+TYPE_ISO14443_3_MIFARE_PLUS_SL1 : unsigned int = 45
+TYPE_ISO14443_3_MIFARE_PLUS_SL2 : unsigned int = 46
+TYPE_ISO14443_4 : unsigned int = 50
+TYPE_ISO14443_4_MIFARE_DESFIRE : unsigned int = 51
+TYPE_ISO14443_4_MIFARE_PLUS_SL1 : unsigned int = 52
+TYPE_ISO14443_4_MIFARE_PLUS_SL2 : unsigned int = 53
+TYPE_ISO14443_4_MIFARE_PLUS_SL3 : unsigned int = 54
+TYPE_ISO14443_4_MAXIM : unsigned int = 60
+TYPE_ISO14443_4_MAXIM_MAX66000 : unsigned int = 61
+TYPE_ISO14443_4_MAXIM_MAX66020 : unsigned int = 62
+TYPE_ISO14443_4_MAXIM_MAX66040 : unsigned int = 63
+TYPE_ISO15693 : unsigned int = 0xE0000000
+TYPE_ISO15693_STM : unsigned int = 0xE0020000
+TYPE_ISO15693_STM_LR12K : unsigned int = 0xE0022000
+TYPE_ISO15693_STM_LRIS2K : unsigned int = 0xE0020280
+TYPE_ISO15693_STM_M24LR64R : unsigned int = 0xE00202C0
+TYPE_ISO15693_NXP : unsigned int = 0xE0040000
+TYPE_ISO15693_NXP_ICODE_SL1 : unsigned int = 0xE0040001
+TYPE_ISO15693_NXP_ICODE_SL1_L : unsigned int = 0xE0040002
+TYPE_ISO15693_NXP_ICODE_SL1_S : unsigned int = 0xE0040006
+TYPE_ISO15693_Infineon : unsigned int = 0xE0050000
+TYPE_ISO15693_Infineon_my_d : unsigned int = 0xE005FFFF
+TYPE_ISO15693_TI : unsigned int = 0xE0070000
+TYPE_ISO15693_TI_Tag_it_HFI_Pro : unsigned int = 0xE007E000
+TYPE_ISO15693_TI_Tag_it_HFI_Plus : unsigned int = 0xE0078000
+TYPE_ISO15693_Fujitsu : unsigned int = 0xE0080000
+TYPE_ISO15693_Fujitsu_MB89R1xx : unsigned int = 0xE0080001
+TYPE_ISO15693_EM : unsigned int = 0xE0160000
+TYPE_ISO15693_EM_4034 : unsigned int = 0xE0160004
+TYPE_ISO15693_KSW : unsigned int = 0xE0170000
+TYPE_ISO15693_MAXIM : unsigned int = 0xE02B0000
+TYPE_ISO15693_MAXIM_MAX66100 : unsigned int = 0xE02B0010
+TYPE_ISO15693_MAXIM_MAX66120 : unsigned int = 0xE02B0020
+TYPE_ISO15693_MAXIM_MAX66140 : unsigned int = 0xE02B0030
+TYPE_ISO15693_IDS_SL13A : unsigned int = 0xE036FFFF
+GetTagHandlerType() : unsigned int
+GetTagDriverType() : unsigned int
+GetErrorDBAddress() : unsigned int
+ReadMultipleBlocks() : int
+ReadMultipleBlocks() : int
+WriteMultipleBlocks() : int

#### 12.3.4. Excursion: Class FedmlscTagHandler\_ISO15693

The ISO 15693 standard is represented by the TagHandler class FedmlscTagHandler\_ISO15693. Many Transponder manufacturers have extended the ISO 15693 standard with its own API and are supported with derived classes.

With an Inventory the RFID-Reader specifies and returns for each detected Transponder a Transponder type ([11.1. Inventory](#)). If an ISO 15693 compliant Transponder is detected, the next problem is to identify the manufacturer and chip type to create the proper TagHandler object. The ISO 15693 Part 3 specifies a manufacturer ID as part of the UID.

MSB				LSB			
64	57	56	49	48			1
'E0'		IC Mfg code		IC manufacturer serial number			

A chip identifier is unfortunately not specified, but most of the manufacturers use the bits above 41 to insert a chip ID. Based on this information, the proper TagHandler can be created.

The following manufacturers and Chip-Types are supported:

Manufacturer	Chip-Types
STMicroelectronics SA	LRI2K LRIS2K M24LR64-R
NXP Semiconductors	I-Code SLI I-Code SLI-L I-Code SLI-S
Infineon Technologies AG	my-d Light
Texas Instruments	Tag-it HF Tag-it HF-I Pro Tag-it HF-I Plus
Fujitsu Limited	MB89R116 MB89R118 MB89R119
EM Microelectronic-Marin SA	EM4034
KSW Microtec GmbH	TempSens
Maxim	MAX66100 MAX66120 MAX66140
IDS Microchip AG	SL13A

**12.3.5. Excursion: Class FedmlscTagHandler\_EPC\_Class1\_Gen2**

---

The following manufacturers and Chip-Types are supported:

Manufacturer	Chip-Types
IDS	SL900A

Preliminary

---

**12.3.6. Excursion: Classes FedmlscTagHandler\_ISO14443**

---

The following manufacturers and Chip-Types are supported:

Manufacturer	Chip-Types
STMicroelectronics SA	SR176 SRI512 SRIX512 SRI4K SRIX4K
NXP Semiconductors	MIFARE Classic 1K MIFARE Classic 4K MIFARE Ultralight MIFARE Ultralight C MIFARE DESFire MIFARE Plus SL0..3
Innovision or other	Jewel
Infineon Technologies AG	my-d proximity SLE55Rxx my-d move SLE
Maxim	MAX66000 MAX66020 MAX66040

---

**12.3.7. Excursion: Class FedmlscTagHandler\_ISO14443\_4\_MIFARE\_DESFire**

---

Preliminary

---

## 12.4. The application TagHandlerSample

---

Preliminary

---

**13. Section 7: APDU Handling with ISO 14443-4 compliant Tags**

---

Preliminary

## 14. Section 8: Programming for the Buffered-Read-Mode

The Buffered-Read-Mode is an automatic read mode and the fastest way of scanning Transponder data and is supported by many OBID i-scan® Readers. It should be preferred for applications with short timing conditions to detect and read from a Transponder. The second advantage is the buffering of the collected data in a First-in First-out buffer to provide the discontinuous request of these data records.

The Buffered-Read-Mode needs polling from the host-side application for receiving the automatic scanned Transponder data. When the Notification-Mode is enabled in the Reader (parameter: `OperatingMode.Mode`), the Buffered-Read-Mode Task in the Reader is started and scans continuously for Transponders.

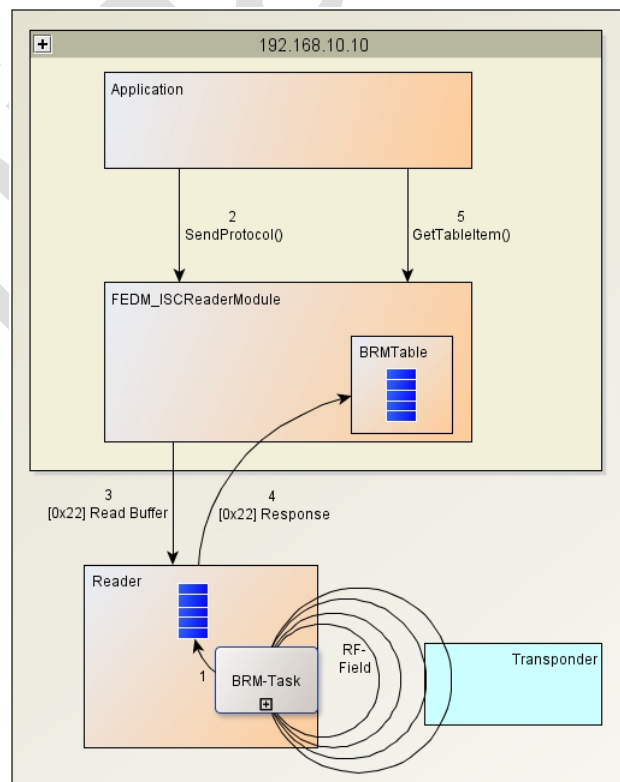
With settings in the parameter group `OperatingMode.BufferedReadMode.DataSelector`, the Reader can be configured to read specific data like UID, data blocks or Date/Time.

### 14.1. Method of operation

Programming for this synchronous operating mode is quite simple. Programmers, following this step-by-step explanation, illustrated with the picture on the right side, should be well prepared for this task.

If the Reader is detecting Transponders, data elements are read and collected in the Reader's internal table (1), one record for each Transponder. Asynchronously, the application can query with a call of `SendProtocol()` (2) all new – but not more than 255 – records from the Reader's table with command [0x22] Read Buffer (3) to save them in the BRMTable of the module `FedMlscReader` (4).

The final step from application-side is to query and process the new table data (5).



### 14.2. Programming the query cycle

### 14.3. Structure of the received data



---

**14.4. Adjust the method of operation**

---

---

**14.4.1. Excursion: Filter**

---

---

**14.4.2. Excursion: Adjust the structure of the data record**

---

---

**14.4.3. Excursion: Triggered Mode**

---

---

**14.4.4. Excursion: Automatic activation of outputs**

---

---

**14.4.5. Excursion: Writing data to the Transponder**

---

---

**14.5. The application BRMSample**

---

Preliminary

## 15. Section 9: Programming for the Notification-Mode

The Notification-Mode is an extension of the [Buffered-Read-Mode](#) and is supported by many OBID i-scan® Readers. While the Buffered-Read-Mode needs polling from the host-side application for receiving the automatic scanned Transponder data, the Notification-Mode transmits these data automatically to a configurable destination (see parameter group: `OperatingMode`.

`NotificationMode.Transmission.Destination.Ipv4`).

When the Notification-Mode is enabled in the Reader (parameter: `OperatingMode.Mode`), the Buffered-Read-Mode Task in the Reader is started and scans continuously for Transponders.

### 15.1. Method of operation

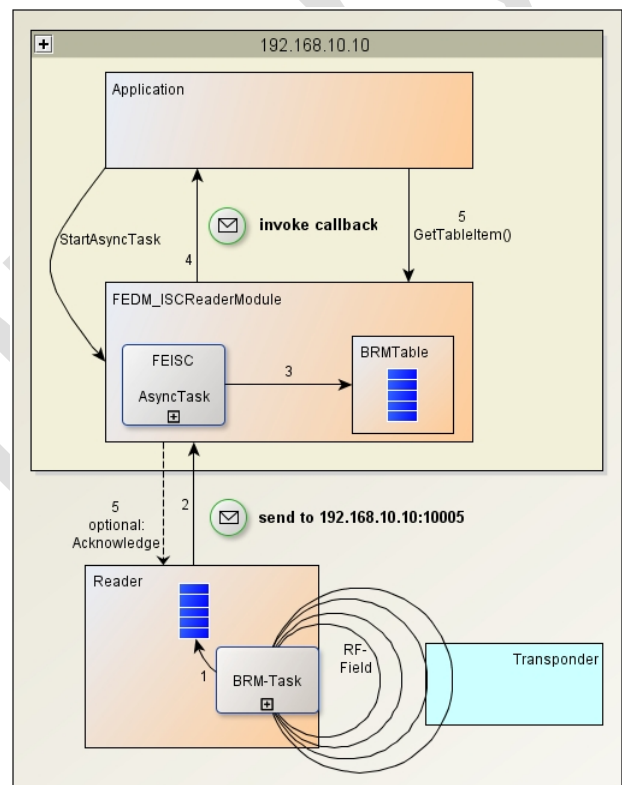
Programming for this asynchronous operating mode needs a deeper understanding of program and data flow inside the Reader and the library. Programmers, following this step-by-step explanation, illustrated with the picture on the right side, should be well prepared for this task.

The initial step has to be the call of the method `StartAsyncTask()` to start an asynchronous task inside the library FEISC (protocol layer) and to install a notification function<sup>1</sup> as an event handler. The asynchronous task is realized as a Thread.

If the Reader is detecting Transponders, data elements are read and collected in the Reader's internal table (1), one record for each Transponder. If new records are added to the internal table, the Reader try to establish a TCP/IP connection (2) to the configured destination address and if this was successful, all new – but not more than 255 – records are transmitted to the host application.

Receiver is the module `FedmlscReader`. He stores the new data elements in the internal `BRMTable` (3)<sup>2</sup> and calls the installed event handler (4) to inform the application. The final step from application-side is to query and process the new table data (5).

If the acknowledge option is configured in the Reader's configuration, the asynchronous task inside FEISC will transmit this protocol after the callback function returns (5). By default, a Reader transmits the data records as fast as possible and is not waiting for an acknowledge<sup>3</sup>.



<sup>1</sup> In C++: callback function; in Java: listener method; in C#: delegate

<sup>2</sup> Step (3) overwrites older data records.

<sup>3</sup> When the secured data transmission is configured, handshake is enabled by default and cannot be disabled.

## 15.2. Register an event

Registering an event handler for notification data is realized together with the start of the asynchronous listener task with the call of `StartAsyncTask()`.



### FEDM

```
void MyClass::StartAsyncTask()
{
    int back = 0;
    FEDM_TASK_INIT taskInit; // task initialization structure (declared in FEDM_ISCReaderModule.h)

    // mandatory: initialize task structure with 0
    memset(&taskInit, 0, sizeof(FEDM_TASK_INIT));

    // settings for the listener task
    taskInit.pAny = (void*)this; // pointer to anything, which is reflected as the first parameter
    taskInit.uiFlag = FEDM_TASKCB2; // specifies the callback to be used to
    taskInit.uiUse = FEDM_TASKID_NOTIFICATION; // defines the task
    taskInit.iPortNr = 10005; // listener port
    taskInit.uiTimeout = 0; // total timeout in seconds after receiving the first bytes of a record
                            // if 0, the timeout is set internally to 1s. This is enough for small
                            // local networks
                            // must be adapted to the network capabilities (Internet, GPRS, ...)
    taskInit.cbFct2 = cbsFct2; // application-side provided callback function
    taskInit.iNotifyWithAck = 1; // Reader waits for an acknowledge, before the next records are
                                // transmitted. See also 15.6. Considerations for fail-safe operation
    // it is strongly recommended to enable the Keep-Alive option
    taskInit.bKeepAlive = true;
    taskInit.uiKeepAliveIdleTime = 500;
    taskInit.uiKeepAliveProbeCount = 5; // applicable only for Linux
    taskInit.uiKeepAliveIntervalTime = 500;

    // and go...
    back = reader.StartAsyncTask(&taskInit);
    if(back != 0)
    {
        // error handling
    }
}
```



```
import de.feig.*;

// MyClass must be derived from the interface FedmTaskListener
void MyClass::StartAsyncTask()
{
    FedmTaskOption taskInit = new FedmTaskOption(); // task initialization class

    // settings for the listener task
    taskInit.setIpPort(10005); // listener port
    // total timeout (in seconds after receiving the first bytes of a record) is set internally to 10s
    // This is enough for small local networks
    // can be adapted to the network capabilities (Internet, GPRS, ...) with setInventoryTimeout()
    taskOpt.setInventoryTimeout((byte)0);
    // Reader waits for an acknowledge, before the next records are transmitted.
    // See also 15.6. Considerations for fail-safe operation
    taskInit.setNotifyWithAck(1);

    try
    {
        // and go...
        reader.startAsyncTask(FedmTaskOption.ID_NOTIFICATION, this, taskInit);
    }
    catch (Exception ex)
    {
        // error handling
    }
}
```

**C#**

```
using OBID;

// MyClass must be derived from the interface FedmTaskListener
void MyClass::StartAsyncTask()
{
    taskOpt = new FedmTaskOption();

    // settings for the listener task
    taskOpt.IPPort = 10005;    // listener port

    // total timeout (in seconds after receiving the first bytes of a record) is set internally to 30s

    // Reader waits for an acknowledge, before the next records are transmitted.
    // See also 15.6. Considerations for fail-safe operation
    taskOpt.NotifyWithAck = 1;

    try
    {
        // and go...
        reader.StartAsyncTask(FedmTaskOption.ID_NOTIFICATION, this, taskOpt);
    }
    catch (Exception ex)
    {
        // error handling
    }
}
```

### 15.3. Event handling



#### FEDM

```
// definition of the callback function; must be declared as a static member in the header
// iError - error code (<0) or OK (0) or reader status byte (>0)
// ucCmd - command byte from notification protocol
// cIPAdr - IP-Address of the reader
// iPortNr - port number of the local port which has received the notification

void MyClass::cbsFct2(void* pAny, int iError, unsigned char ucCmd, char* cIPAdr, int iPortNr)
{
    MyClass* pThis = (MyClass*)pAny;

    if(iError != 0)
    {
        // process error codes, but leave the callback as fast as possible
        return;
    }

    switch(ucCmd)
    {
    case 0x22: // notification data
        // process the notification, but leave the callback as fast as possible
        break;
    case 0x6E: // diagnostic data (keep-alive protocol from Reader)
        // process diagnostic data, but leave the callback as fast as possible
        break;
    case 0x74: // input status
        // process input status, but leave the callback as fast as possible
        break;
    }
}
```



```
import de.feig.*;

/**
 * Listener method for the transponder data coming with notification event.
 * @param error error code (<0) or OK (0) or reader status byte (>0)
 * @param remoteIP IP-Address of the reader
 * @param portNr port number of the local port which has received the notification
 */
public void onNewNotification(int error, String remoteIP, int portNr)
{
    if(iError != 0)
    {
        // process error codes, but leave the callback as fast as possible
        return;
    }

    // process the notification, but leave the callback as fast as possible
    FedmBrmTableItem[] brmItems = null;
    brmItems = (FedmBrmTableItem[])reader.getTable(FedmIscReaderConst.BRM_TABLE);
    ...
}
```



```

C#
using OBID;

// Listener method for the transponder data coming with notification event.
// error - error code (<0) or OK (0) or reader status byte (>0)
// remoteIP - IP-Address of the reader
// portNr - port number of the local port which has received the notification

public void onNewNotification(int error, String remoteIP, int portNr)
{
    if(iError != 0)
    {
        // process error codes, but leave the callback as fast as possible
        return;
    }

    // process the notification, but leave the callback as fast as possible
    FedmBrmTableItem[] brmItems;
    brmItems = (FedmBrmTableItem[])reader.GetTable(FedmIscReaderConst.BRM_TABLE);
    ...
}

```

## 15.4. Cancel asynchronous task

The cancelling of an asynchronous task is realized with the method `CancelAsyncTask()`. Internally, `CancelAsyncTask()` sets a flag for the listener thread to stop the process and to force immediately finishing. `CancelAsyncTask()` is waiting up to 3 seconds for the thread finish event.

If the listener thread is just calling the callback function, `CancelAsyncTask()` returns immediately with the error code -4084 ("FEISC: asynchronous task is busy") and `CancelAsyncTask()` has to be called again, until the return value is 0.

## 15.5. Adjust the method of operation

The Notification-Mode can be adjusted by a lot of parameters, collected in the Reader's configuration. In addition to the data elements to be read from a Transponder, trigger and filter parameter can be set to control and reduce the amount of data records.

Three other parameters control the data transfer over the Ethernet link:

1. A handshake mode can be enabled to let the Reader wait for an acknowledge from the receiver with the parameter `(OperatingMode.NotificationMode.Transmission.Enable_Acknowledge)`. When secured data transmission is configured, the handshake is enabled by default and cannot be disabled.
2. A limitation of the number of data records to be sent with one transmission can be set with `OperatingMode.NotificationMode.Transmission.DataSetsLimit`. Limitation is recommended for slow or bad conditioned links like WLAN or GPRS. In some constellations, the limit can be set to 1 data record to serialize the notifications for the following information processing.
3. A connection can be kept open after a notification transmission. This hold time can be adjusted with `OperatingMode.NotificationMode.Transmission.Destination`.

ConnectionHoldTime. With every connect to a server, a new transmit port is selected in the Reader's TCP stack and is blocked for up to 2 minutes before it gets into the closed state. In high frequented notification situations, this can cause a problem in the Reader, because the number of ports is limited and when all ports are used, a notification cannot be sent until the first port reaches the closed state.

### 15.5.1. Excursion: Writing data to the Transponder

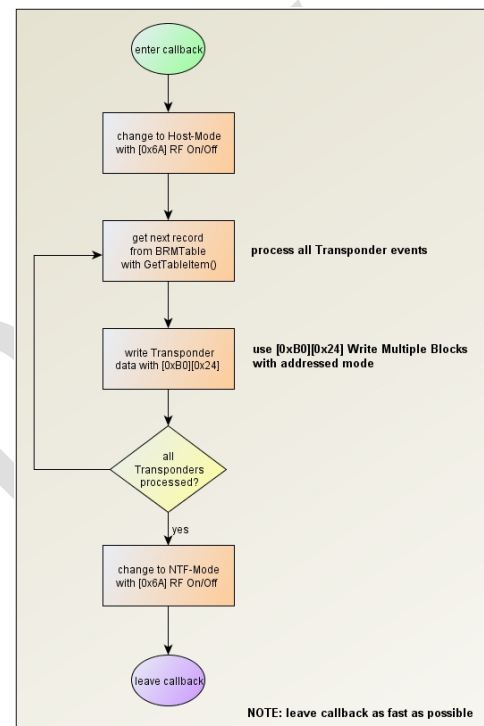
The Buffered-Read-Mode (BRM) and the extension, the Notification-Mode (NTFM), is an interruptible process. The switching from NTFM to Host-Mode (HM) and back to NTFM is controlled by the Reader protocol [0x6A] RF On/Off.

This mode changing is also allowed after receiving a notification and inside a callback function. It enables an event driven writing of data into Transponders.

A typical cycle might be as illustrated in the picture on the right side: after entering the callback function, all Transponder events are processed in a loop. It is important to switch back to Notification-Mode before the callback is left.

When the callback is returned, the next notification can be received by the library FEISC.

Although the execution of the callback interrupts the receiving of the next notification and prevents the loss of events, the execution time of a callback should be as little as possible. A callback is normally a member of another process and during the execution the process scheduler must share the CPU time of this process with the callback. On the other side, as longer the execution time, as longer is the waiting time for `CancelAsyncTask()` and the risk for a deadlock.





---

## 15.6. Considerations for fail-safe operation

---

---

### 15.6.1. Keep-Alive option for detecting broken network link

---

When the Ethernet cable gets broken while an active communication, the server-side application (Host) may not indicate an error while he is listening for new transmissions. On the other side, the Reader will run in an error with the next transmission and can close and reopen the socket. But the close and reopen will never be noticed by the Host, as he is listening at a half-closed port.

The solution for this very realistic scenario is the activating of the Keep-Alive option on the server-side (see code examples in [15.2. Register an event](#)).

---

### 15.6.2. Avoiding deadlock situations

---

When the code execution is just inside a callback function from the main process and the application calls `CanceAsyncTask()` in a loop, the main process will be locked by the loop and the callback function will never return. This is a typical deadlock situation, where a caller is waiting for completion of a process while this process is interrupted by the scheduler.

One solution for this situation is an repetitive asynchronous call (message driven?) of `CanceAsyncTask()` to close the execution path of the main process immediately, when `CanceAsyncTask()` returns with -4084 (busy). This enables the scheduler to continue the callback function.

A second and familiar situation is the closing of an application while the listener thread is still inside the callback function. The application programmer has to be ensure that closing the application does not interrupt the canceling of the listener thread.

---

**15.7. The application NotifySample**

---

Preliminary

---

## **16. Section 10: Programming for the Scan-Mode**

---

---

### **16.1. Method of operation**

---

---

### **16.2. Select the output interface**

---

---

### **16.3. Register an event**

---

---

### **16.4. Event handling**

---

---

### **16.5. Adjust the method of operation**

---

---

#### **16.5.1. Excursion: Setting the data format**

---

---

#### **16.5.2. Excursion: HID (Human-Interface-Device)**

---

---

### **16.6. The application ScanSample**

---

---

## 17. Section 11: Management of the Reader configuration

---

---

### 17.1. Persistence of the Reader configuration

---

Each OBID i-scan® and OBID® *classic-pro* reader is controlled by parameters which are stored grouped in blocks in an EEPROM and are described in detail in the system manual for the respective reader. After switching on or resetting the reader, all parameters are loaded into RAM, evaluated and incorporated in the controller.

All parameters can be modified using a protocol so that the behavior of the reader can be adapted to the application. Ideally, the program ISOStart/CPRStart is used for this adaptation and normally no parameters have to be changed in the application. Despite this, it can happen that one or more parameters from a program have to be changed. This chapter should familiarize you with the procedure using the reader class as an example.

A common characteristic of all readers is the grouping in blocks of thematically related parameters to 14 or 30 bytes per configuration block. Each parameter cannot be addressed individually but must always be retrieved together with a configuration block using the protocol [0x80] or [0x8A] Read Configuration, then modified and finally written back to the reader with the protocol [0x81] or [0x8B] Write Configuration. This cycle must always be complied with and is also checked by the reader class. This means that writing a configuration block without previously reading the same block is not possible.

The reader class manages the configuration data in a byte array for data from the EEPROM and a second byte array for data from the RAM of the reader. The differentiation is important as changes in RAM are used immediately while changes in the EEPROM of the reader do not become active until after a reset. Therefore the reader class has its own byte arrays for both configuration sets.

---

#### 17.1.1. Physical (old) structure

---

---

#### 17.1.2. Logical (new) structure

---

---

### 17.2. Read/Modify/Write of the Reader configuration

---

---

### 17.3. Serialization of the Reader configuration into an XML file

---

---

**18. Section 12: Activation of outputs**

---

Preliminary

---

**19. Section 13: Reading states of digital inputs**

---

Preliminary

---

**20. Section 14: Reset methods**

---

Preliminary

---

## **21. Section 15: Classes for external Function Units**

---

---

### **21.1. Multiplexer (HF and UHF)**

---

---

### **21.2. Automatic Antenna Tuner (HF)**

---

---

### **21.3. The Application DATuningTool**

---

---

### **21.4. People-Counter in HF-Gate-Antennas**

---