OBID®

MANUAL

Java Class Library

ID ISC.SDK.Java

Version 3.02.06

Software-Support for

OBID i-*scan*®

and

OBID® *classic-pro*

Reader Families

for 32- or 64-Bit Operating Systems

Windows 2000/XP/Vista/7

and Linux

with 32-Bit Java Runtime Environment (JRE) 5 or higher

final
public (B)
2010-10-18
H31101-16e-ID-B.doc

FEIG ELECTRONIC

Note

FEIG ELECTRONIC GmbH
Lange Straße 4
D-35781 Weilburg-Waldhausen
Germany
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.

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 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 relate to ideal outside conditions. FEIG ELECTRONIC GmbH does not guarantee the failure-free function of the OBID® -system in outside environment.

Licensing Agreement for use of the software

This is an agreement between you and FEIG ELECTRONIC GmbH (hereafter "FEIG") for use of provided software **ID ISC.SDK.Java** (application programs, program libraries, source code examples and the connected documents), hereafter called licensing material. By installing and using the licensing material you agree to all terms and conditions of this agreement without exception and without limitation. If you are not or not completely in agreement with the terms and conditions, you may not install the licensing material or use it in any way.

§1 Subject and Scope of the Agreement

1. FEIG grants you the non-exclusive and non-transferable right to install the licensed material and to use it under the conditions specified below.

2. The licensed material is intended for use by an individual developer (single user license). You may install all the components of the licensed material on a hard-disk of a single computer which is intended for your use.

3. Installation and use may also include a network fileserver as long as the use is exclusive to the licensee. A separate license is required for any additional user.

4. You may make a backup copy of the licensed material.

5. FEIG grants you the right to use the documented Java library OBIDISC4J.JAR, as well as the necessary native libraries, for developing your own application program and to sell this Java library OBIDISC4J.JAR, as well as the necessary native libraries, only together with your application programs without payment of licensing fees so long as these application programs are used only to control or operate devices and/or systems which are developed and/or sold by FEIG.

6. FEIG grants you the right to use and modify the source code of the supplied program examples for developing your own application programs and to sell these application programs together with the Java library OBIDISC4J.JAR, as well as the necessary native libraries, without payment of licensing fees so long as these application programs are used only to control or operate devices and/or systems which are developed and/or sold by FEIG.

7. This license material can depend on third-party software. In case of the use of this third-party software the listed license agreements in chapter Third-party Licensing agreements have to be applied.

§2. Protection of the Licensed Material

1. The licensed material is the intellectual property of FEIG and its suppliers. The licensed material is also protected by German copyright law, International Treaty provisions and the laws of the country in which it is used. Structure, organization and code of the licensed material are the valuable trade secrets of FEIG and its suppliers.

2. You agree not to change, modify, adapt, translate, reverse engineer, decompile, disassemble or otherwise attempt to discover the source code of the executable application programs, program libraries and documents.

3. To the extent that FEIG has applied protection marks, such as copyright marks and other legal restrictions in the licensing material, you agree to keep these unchanged and to use them unchanged in all complete or partial copies which you make.

4. The publication and transmission to third parties of licensed material prohibited as long as no explicit agreement to the contrary has been established between you and FEIG . This provision does not apply to such application programs as have been created and sold under §1 Par. 5 of this Agreement.

§ 3 Warranty and Limitation of Liability

1. You agree with FEIG that it is not possible to develop electronic data processing programs such that they are without defect for all application conditions. FEIG calls explicit attention to the fact that the installation of a new program may affect already existing software, including software which does not run simultaneous with the new software. In no event will FEIG be liable to you for any consequential, incidental or special damages, including any lost profits or lost savings. If you want to be sure that no already installed program will be affected, you may not install the licensed material.

2. FEIG calls explicit attention to the fact that the use of the licensed material may result irreversible settings and adjustments to devices which may in turn destroy or otherwise make them unusable. FEIG assumes no liability for such actions whether knowingly or unknowingly.

3. FEIG provides the software "as is“ and without any warranty. FEIG cannot guarantee the performance or the results you obtain from using the licensed material. FEIG assumes no liability or guarantee that the protection rights of third parties are not violated, nor that the software is suitable for a particular purpose.

4. FEIG call explicit attention the licensed material is not designed with components and testing for a level of reliability suitable for use in or in connection with surgical implants or as critical components in any life support systems whose failure to perform can reasonably be expected to cause significant injury to a human.

To avoid damage, injury, or death, the user or application designer must take reasonably prudent steps to protect against system failures.

§ 4 final clause

1. This Agreement contains the complete licensing terms and conditions and supercedes any prior agreements and terms. Changes and additions must be made in writing.

2. If any provision in this agreement is declared to be void, or if for any reason is declared to be invalid or of no effect, the remaining provisions shall be in no manner affected thereby but shall remain in full force and effect. Both parties agree to replace the invalid provision with one which comes closest to its original intention.

3. This agreement is subject to the laws of the Federal Republic of Germany. Place of jurisdiction is Frankfurt a. M.

Please direct any questions pertaining to this agreement to:

FEIG ELECTRONIC GmbH
Lange Strasse 4,
D-35781 Weilburg-Waldhausen
- Germany -
Fon: +49 6471 / 3109-0
Fax: +49 6471 / 3109-99
e-mail: info@feig.de
http://www.feig.de

Third-party Licensing agreements

Licensing agreement of openSSL organization

The following license issues are to be applied in the case that encrypted data transmission is used.

The implementation was written so as to conform with Netscapes SSL.

This library is free for commercial and non-commercial use as long as the following conditions are aheared to. The following conditions apply to all code found in this distribution, be it the RC4, RSA, lhash, DES, etc., code; not just the SSL code. The SSL documentation included with this distribution is covered by the same copyright terms except that the holder is Tim Hudson (tjh@cryptsoft.com).

Copyright remains Eric Young's, and as such any Copyright notices in the code are not to be removed. If this package is used in a product, Eric Young should be given attribution as the author of the parts of the library used.
This can be in the form of a textual message at program startup or in documentation (online or textual) provided with the package.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
1. Redistributions of source code must retain the copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. All advertising materials mentioning features or use of this software must display the following acknowledgement:
"This product includes cryptographic software written by Eric Young (eay@cryptsoft.com)"
The word 'cryptographic' can be left out if the rouines from the library being used are not cryptographic related :-).

4. If you include any Windows specific code (or a derivative thereof) from the apps directory (application code) you must include an acknowledgement:
"This product includes software written by Tim Hudson (tjh@cryptsoft.com)"

THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The licence and distribution terms for any publically available version or derivative of this code cannot be changed. i.e. this code cannot simply be copied and put under another distribution licence [including the GNU Public Licence.]

Contents:

Remarks concerning the documentation for this library

This manual describes a software library for which there is also online documentation. For this reason we have intentionally avoided documenting more than absolutely necessary for understanding the functionality and utilization of the classes. It is presumed that the user of this library will refer to the online documentation for details on the classes and methods.

System manuals for the OBID® Readers actually used must also be referred to for understanding the classes and methods.

FEIG ELECTRONIC GmbH does not duplicate information about OBID® Readers in different manuals or include cross-references to certain page numbers of another document. This is because the manuals are constantly updated, and helps to eliminate mistakes resulting from information obtained from out-of-date documents. We therefore encourage the user of this library to always verify that he is using the current manuals. The newest versions can always be obtained from FEIG ELECTRONIC GmbH.

<u>Important notes:</u>

You may use this library only if you have first agreed to the licensing terms found on the reverse side.

1. Introduction

The Java class library ID OBIDISC4J from FEIG ELECTRONIC GmbH represents yet another component for simplifying the development of application programs in Java for OBID *i-scan*® and OBID® *classic-pro* readers.

This manual is intended as an introduction to the library and supplements the online documentation.

The Java class library ID OBIDISC4J currently supports Windows and Linux[1].

The Java class library ID OBIDISC4J is based on the C++ class library ID FEDM as well as the native function libraries ID FECOM, ID FEUSB[2], ID FETCP, ID FEISC, ID FETCL and ID FEFU. The Java class library therefore consists only of a wrapper. Nevertheless, the full functionality of the C++ class library is accessible for Java:

- A uniform organizational principle for savable data from reader and transponder in data containers and tables.
- Overloaded methods for access to the data containers and tables.
- A single, easy to use communications method.
- Synchronous and asynchronous communication
- Complete error handling using exceptions or return values from methods.
- A simple way of serializing reader configuration data in an XML file.

Important note:

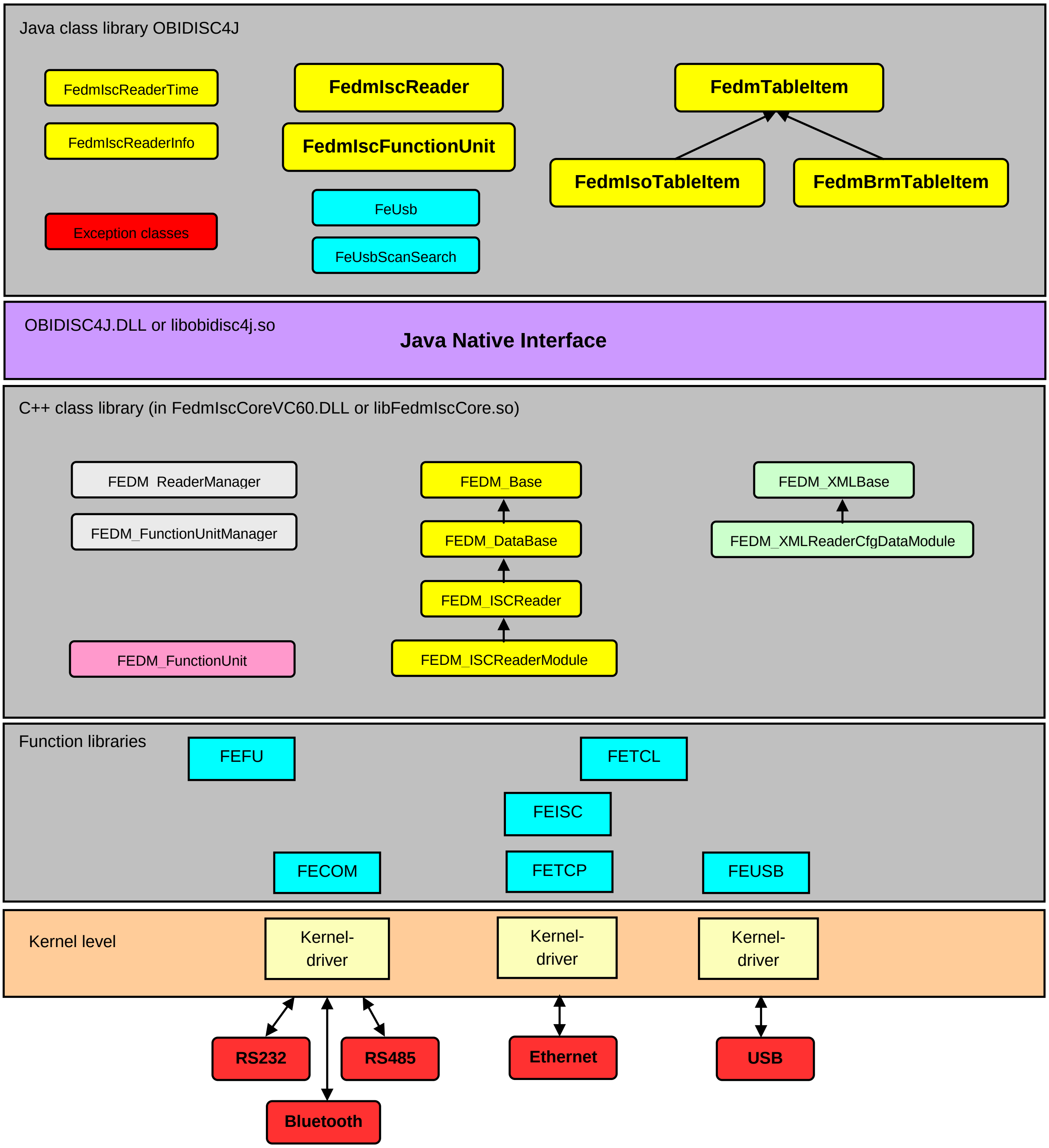
The ID OBIDISC4J class library is being constantly adapted. We make effort to maintain the documented status. Nevertheless, changes cannot be precluded.

[1] For x86 processors only. All others upon request.

[2] Only under Windows 2000/XP/Vista/7 and Linux

1.1. Overview of all software modules

The following illustration shows the individual software modules upon which the ID OBIDISC4J Java class library is based. The **FedmIscReader** class is the main class. Through it the communications channel is opened and the entire communication with the reader is carried out on this channel. FedmIscReader builds directly upon the C++ class FEDM_ISCReaderModule, which contains the implementation of the Java methods.

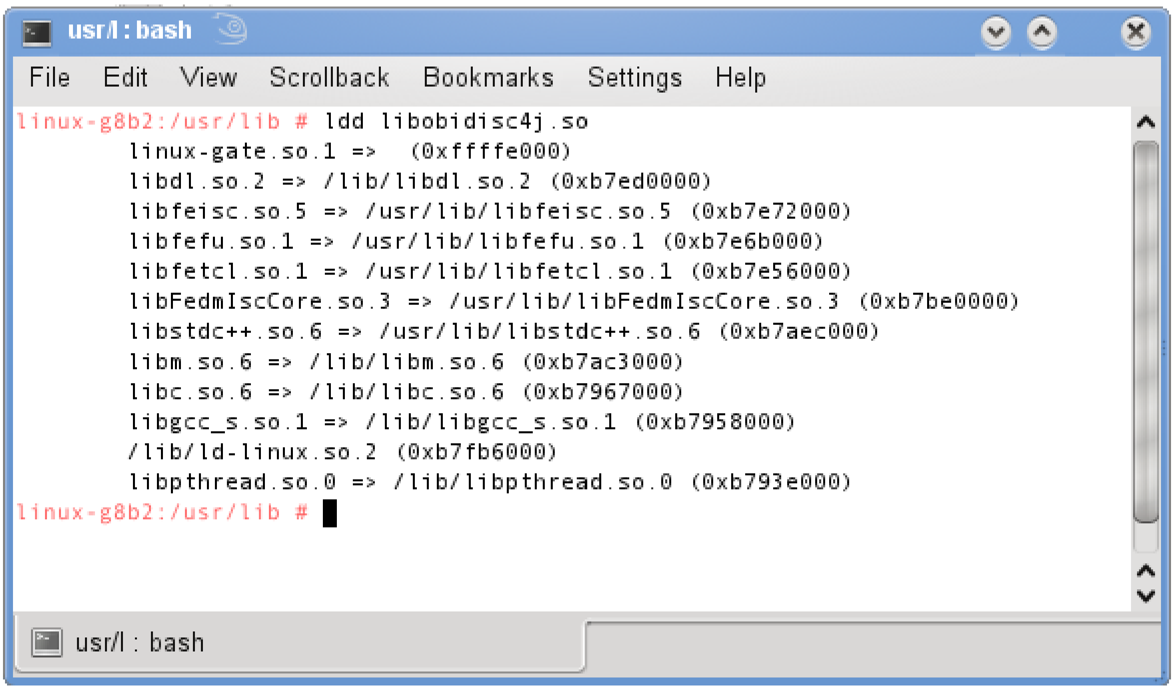


1.2. Supported operating systems

- 32- or 64-Bit Microsoft Windows 2000/XP/Vista/7
- Microsoft Windows CE upon request
- 32- or 64-Bit Linux (x86 processor only, others upon request)

1.3. System requirements

- 32-Bit Java Runtime Environment (JRE) 5 or higher
- Linux: the native libraries are developed and tested under SuSE Linux 11.1 with the GNU Compiler Collection V4.3.2. All native libraries built as 32-Bit library files are linked to 32-Bit system libraries similar to libobidisc4j.so. It is required to have a 32-Bit runtime environment installed on the target system.



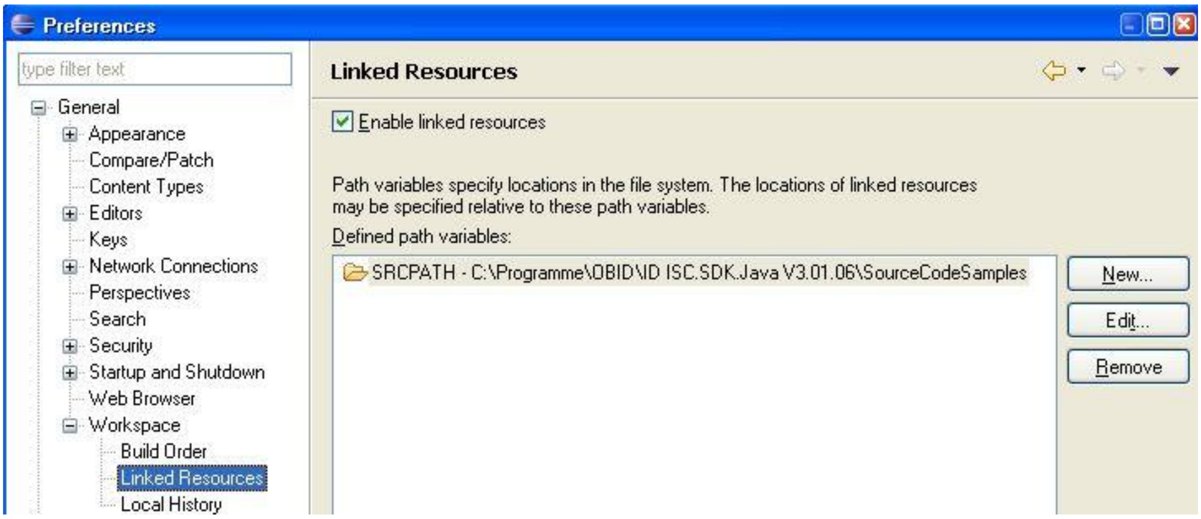
2. Installation

2.1. Installation on Development Computer

2.1.1. Windows 2000/XP/Vista/7

The sample projects shipped with this SDK are properly configured to include dependent libraries, if the SDK is installed in C:\Programe Files\OBID\. If the installation path differs, the configuration for Eclipse must be adapted:

- Set Workspace in *Installation Path*\SourceCodeSamples\Eclipse_3_5\workspace
- Modify Link Resources in Window/Preferences/General/Workspace/Link Resources:



Settings for Netbeans projects remain constant.

If an alternative IDE is preferred, the installation is the same as for a target computer and described in **Fehler! Verweisquelle konnte nicht gefunden werden.**.

2.1.2. Linux

The installation is the same as for a target computer and described in **Fehler! Verweisquelle konnte nicht gefunden werden.**.

2.2. Installation on Target Computer

2.2.1. Windows 2000/XP/Vista/7

The following files are included:

File	Description
FECOM.DLL	Native library for serial interface
FETCP.DLL	Native library for TCP/IP
FEUSB.DLL	Native library for USB
FEISC.DLL	Native library for OBID i-*scan*® and OBID® *classic-pro* Reader
FEFU.DLL	Native library for OBID i-*scan*® external Function Units
FETCL.DLL	Native library for ISO14443-4 T=CL protocols with OBID® *classic-pro* Reader
FedmIscCoreVC60.DLL	Native library for OBID i-*scan*® and OBID® *classic-pro* Reader
OBIDISC4J.DLL	Native library for OBID i-*scan*® and OBID® *classic-pro* Reader
OBIDISC4J.jar	Java library

Installation is quite simple:

Copy all DLL files to the directory: *Java-directory*\jre\bin

Copy the file OBIDISC4J.jar to the directory: *Java-directory*\jre\lib\ext

Alternately you can select any other directory, as long as you tell this to the Java environment: java –classpath *Directory*.

2.2.2. Linux

The following files are included:

File[3]	Description
libfecom.so.x.y.z	Native library for serial interface
libfeusb.so.x.y.z	Native library for USB
libfetcp.so.x.y.z	Native library for TCP/IP
libfeisc.so.x.y.z	Native library for OBID i-*scan*® and OBID® *classic-pro* Reader
libfefu.so.x.y.z	Native library for OBID i-*scan*® external Function Units
libfetcl.so.x.y.z	Native library for ISO14443-4 T=CL protocols with OBID® *classic-pro* Reader
libFedmIscCore.so.x.y.z	Native library for OBID i-*scan*® and OBID® *classic-pro* Reader

[3] x.y.z represents the version number of the library file

File[3]	Description
libobidisc4j.so.x.y.z	Native library for OBID i-*scan*® and OBID® *classic-pro* Reader
OBIDISC4J.jar	Java library

Installation is quite simple:

Copy all so-files to the system directory usr/lib and create the necessary symbolic links in the steps listed below. For this you will need superuser privileges.

1) Change to the directory usr/lib:

   cd /usr/lib

2) Create the symbolic links:

   ln –sf libfecom.so.x.y.z libfecom.so.x

   ln –sf libfecom.so.x libfecom.so

   ln –sf libfeusb.so.x.y.z libfeusb.so.x

   ln –sf libfeusb.so.x libfeusb.so

   ln –sf libfetcp.so. x.y.z libfetcp.so.x

   ln –sf libfetcp.so.x libfetcp.so

   ln –sf libfeisc.so. x.y.z libfeisc.so.x

   ln –sf libfeisc.so.x libfeisc.so

   ln –sf libfefu.so.x.y.z libfefu.so.x

   ln –sf libfefu.so.x libfefu.so

   ln –sf libfetcl.so.x.y.z libfetcl.so.x

   ln –sf libfetcl.so.x libfetcl.so

   ln –sf libFedmIscCore.so.x.y.z libFedmIscCore.so.x

   ln –sf libFedmIscCore.so.x libFedmIscCore.so

   ln –sf libobidisc4j.so. x.y.z libobidisc4j.so.x

   ln –sf libobidisc4j.so.x libobidisc4j.so

3) Invoke the program ldconfig

   ldconfig

Copy the OBIDISC4J.jar file to the directory: *Java-directory*\jre\lib\ext

Alternately you can select any other directory, as long as you tell this to the Java environment: java –classpath *Directory*.

In the case that the JVM cannot find the native library libobidisc4j.so, set a symbolic link to the library in the same directory where OBIDISC4J.jar is located.

If USB is intended to be used, then additional installation steps are necessary. Please follow the installation instructions in the FEUSB manual (H00501-#e-ID-B.pdf).

3. Revisions since the previous version

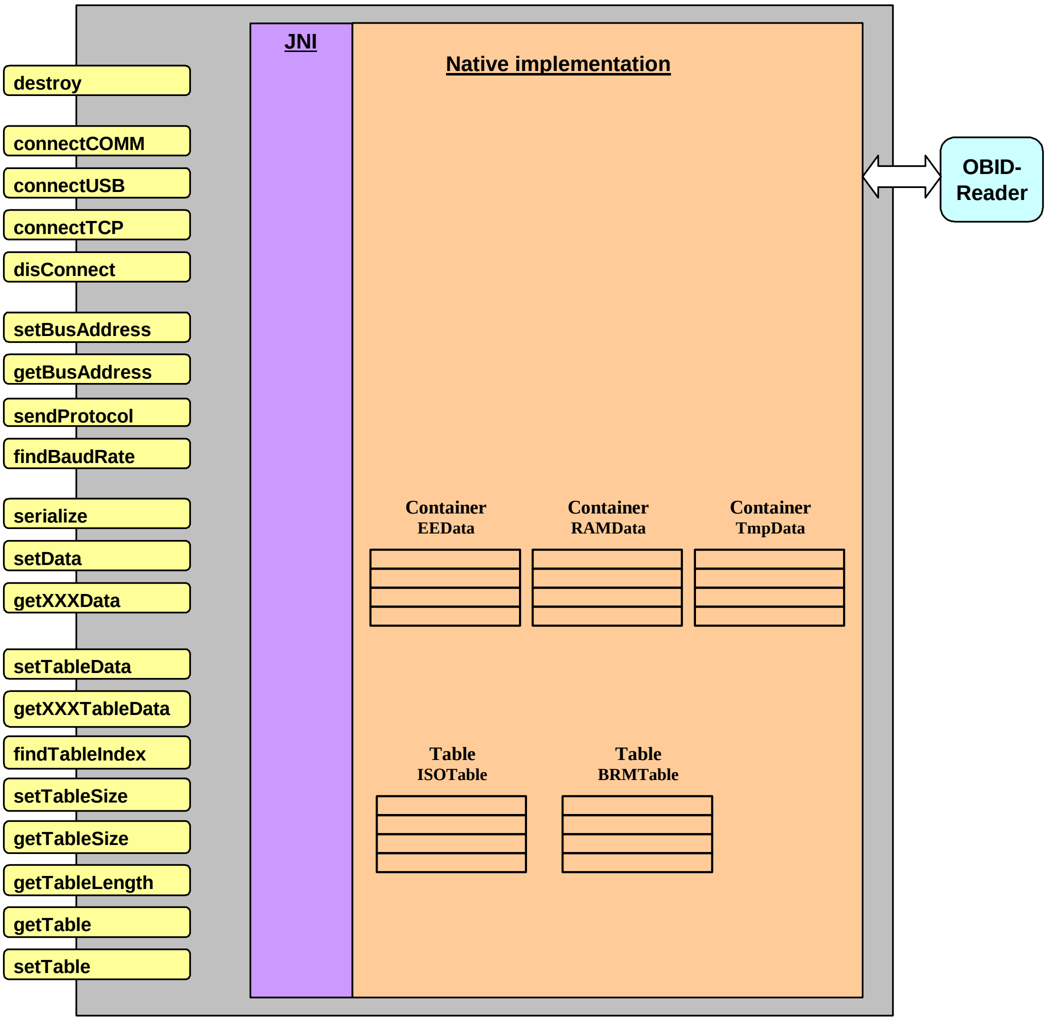
- New reader configuration parameters in the package de.feig.ReaderConfig.

4. Overview of the classes

4.1. Reader class FedmIscReader

The reader class **FedmIscReader** is the main class of the Java library. The component diagram shows an overview of the reader class.

Only the most important methods are shown. Attributes are not contained in the Java class. Refer to the online documentation for a complete description of the methods.



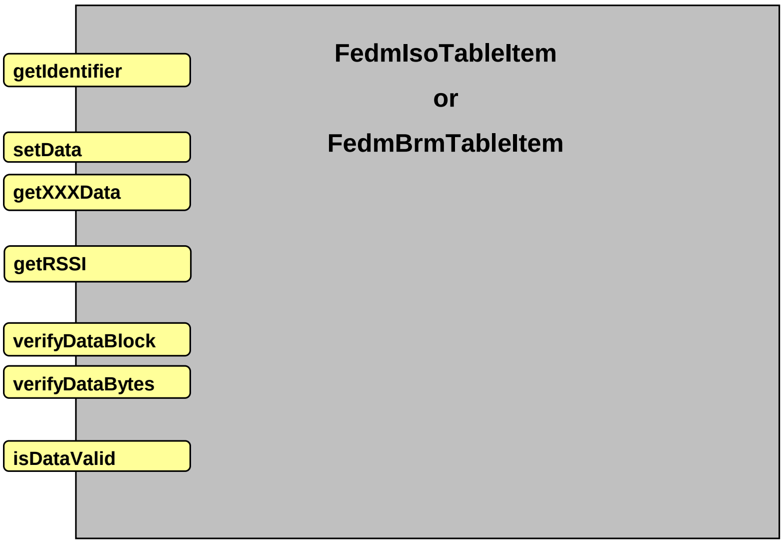
4.2. Table classes FedmIsoTableItem and FedmBrmTableItem

The table classes **FedmIsoTableItem** and **FedmBrmTableItem** are derived from the interface **FedmTableItem** and contain transponder data. An array from these classes forms a table, whereby a mixed table is not allowed.

Both classes are an alternative interface to the transponder for the methods *getXXXTableData*[4] and *setTableData* of the reader class **FedmIscReader**. Data can be exchanged with the transponder using only one of the two interfaces.

FedmIsoTableItem contains transponder data that were read with the ISO host mode reader commands or saved there before writing to the transponder.

FedmBrmTableItem contains transponder data that were read by the reader in Buffered Read Mode or Notification Mode. Since both modes are purely read modes, no data can be written in **FedmBrmTableItem** using *setData*.

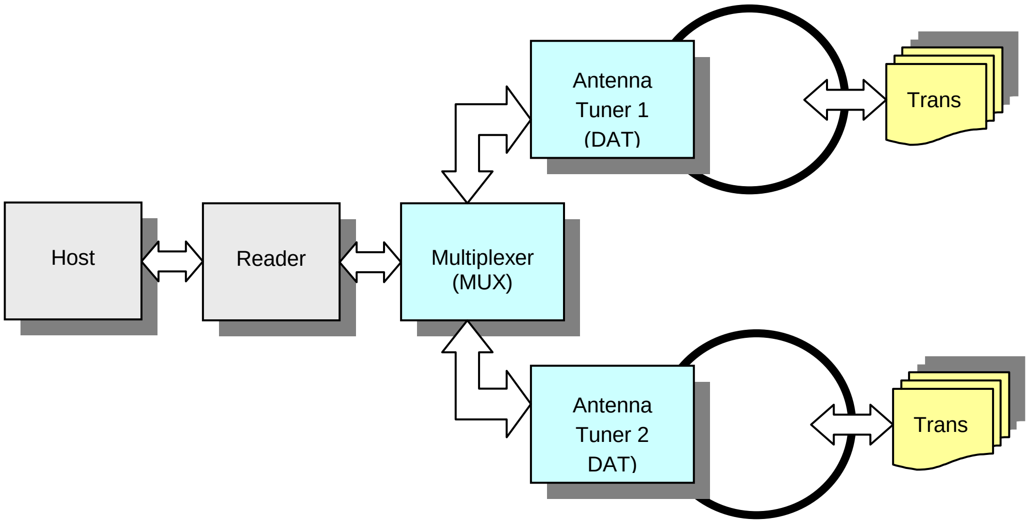


[4] XXX stands for Boolean, Byte, Integer, Long, String and represents the data types of the return value.

4.3. Class FedmIscFunctionUnit

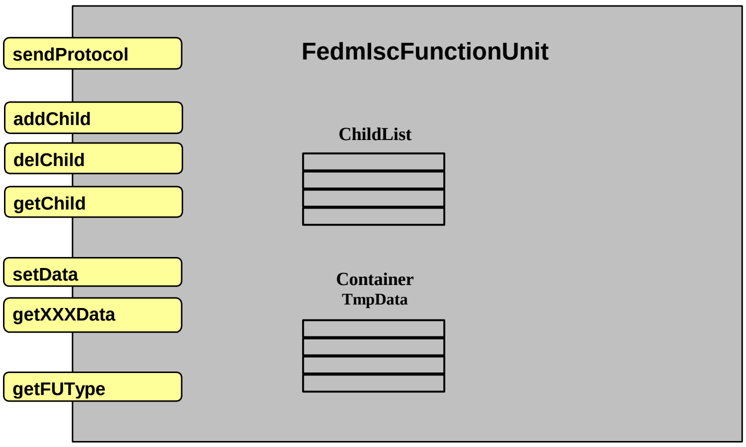
The class **FedmIscFunctionUnit** represents an external function unit (FU) integrated in the antenna cable of the reader. The class has no base class. For a deeper understanding of the possibilities of function units you should read the system manual H30701-xe-ID-B (HF) or H80302-xe-ID-B (UHF). Additional informations can be find in the installation guides of the function units.

In consideration of the fact that a function units needs always a reader as a communication bridge, the class **FedmIscFunctionUnit** can only be instantiated if a reader object of type **FedmIscReader** is previously created.



The picture above demonstrates also that external function units are arranged in hierachical order. The function unit class pattern this topology with a list of successors of type **FedmIscFunctionUnit**. Beginning with the first function unit after the reader one can traverse through the tree of function units.

The component diagram shows an overview of the function unit class.



4.4. Class FedmIscPeopleCounter

The class **FedmIscPeopleCounter** represents an external unit connected at the RS485-Bus of the Reader. The class has no base class. For a deeper understanding of the possibilities of People Counter you should read the system manual H01011-xe-ID-B. Additional information can be found in the installation guides of the gate antennas.

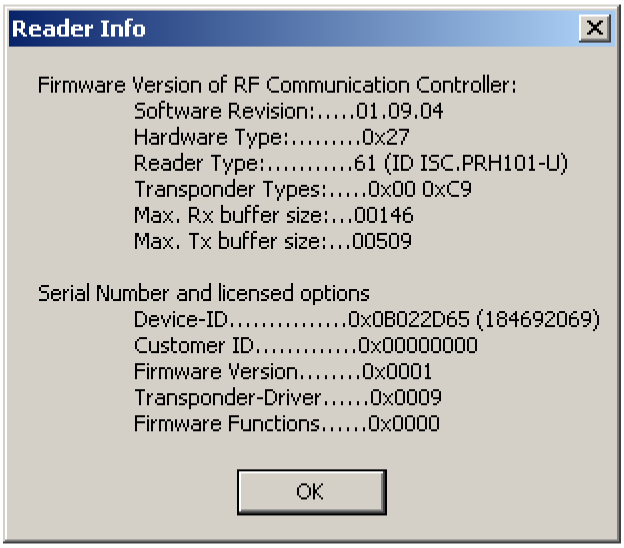
More information can be found in 5.4. Communication with a People Counter.

4.5. Help classes and interfaces

4.5.1. FedmIscReaderInfo

FedmIscReaderInfo is a class collecting all important information of the connected Reader after a call of the method ReadReaderInfo.

The method GetReport() returns a formatted string with all information about the connected reader.



4.5.2. FeIscReaderTime

FeIscReaderTime is a class that represents the reader time in Buffered Read Mode.

The object is obtained only using the method *getReaderTime* of the class **FedmIscBrmTableItem**.

4.5.3. FedmCprApdu

FedmCprApdu is a class supporting the reader class in the asynchronous execution of ISO14443-4 T=CL protocols (APDUs).

4.5.4. FedmCprCommandQueue

FedmCprCommandQueue is a class supporting the reader class in the asynchronous execution of a [0xBC] Command Queue.

4.5.5. FeUsb and FeUsbScanSearch

The class **FeUsb** is a help class for recognizing more than one USB reader when several are connected to the USB at the same time. **FeUsbScanSearch** is a class with search options for a scan procedure on the USB.

If never more than one USB reader is used at a time in your application, you will not need these classes.

4.5.6. FeHexConvert

The class **FeHexConvert** contains useful methods for converting data.

4.5.7. FeMethods

The class **FeMethods** contains useful methods for extracting data values from the access constants (s. 5.5.2. Access constants for temporary protocol data).

4.5.8. The FeIscListener interface

The **FeIscListener** interface enables event handling from the native library. This interface can be used to easily implement a log window for reader logs.

4.5.9. The FeTaskListener interface

The **FeTaskListener** interface enables event handling from the native library. With this interface, the transponder or reader data of an asynchronous read operation can be queried.

4.5.10. The FedmTaskOption property

The class **FedmTaskOption** contains settings for asynchronous tasks.

4.5.11. The Fedm interface

The **Fedm** interface gathers general constants for the class library.

4.5.12. The FedmIscReaderConst interface

The **FedmIscReaderConst** interface gathers general constants for the reader class **FedmIscReader**.

4.5.13. The FedmIscReaderID interface

The interface **FedmIscReaderID** gathers all access constants for temporary protocol data for the OBID *i-scan*® and OBID® *classic-pro* readers.

4.5.14. Das Interface FedmIscFunctionUnitID

The interface **FedmIscFunctionUnitID** gathers all access constants for the external OBID *i-scan*® Function Units.

4.6. Exception classes

4.6.1. FedmException

FedmException is a class which is triggered in exception situations in the area of the native C++ class library FEDM.

4.6.2. FePortDriverException

FePortDriverException is a class which is triggered in exception situations in the area of the native function libraries FECOM, FEUSB and FETCP.

4.6.3. FeReaderDriverException

FeReaderDriverException is a class which is triggered in exception situations in the area of the native function library FEISC.

5. Basic properties of the reader class

The reader class methods can be roughly divided into five categories:

a) Methods for initializing and finalizing

b) Methods for the communications channels

c) Methods for communication

d) Methods for data containers and serializing

e) Methods for tables

5.1. Initializing and finalizing

5.1.1. Initializing

Before using the reader class for the first time, several initializations must be performed:

1. Bus address — The bus address of the reader is preset in the class to 255. Any other address is set using the method *setBusaddress*.

2. Table size — The tables ISOTable and BRMTable contained in the reader class **FedmIscReader** do not have a preset size. Therefore you **must** (!) use the method *setTableSize* to dimension the required table before first communicating with a transponder.

   The reference for the size of a table is the maximum number of transponders that will be located in the reader's antenna field at one time.

   In general you size only one table, since the reader can not work simultaneously in Buffered Read Mode and ISO-Host Mode.

   The following memory capacity per table item is reserved for the tables:

   - BRMTable: 1104 bytes
   - ISOTable: 17496 bytes

3. Reader type

The reader type must be set in the reader class with one of three options:

1. Automatic (recommended): After a successful connection with one of the methods *connectCOMM(..., true)*, *connectUSB* or *connectTCP* the method *readReaderInfo* is executed internally and the reader type is set.
2. Manually 1: The call of the method *readReaderInfo* after a successful opening of a serial port with *connectCOMM(..., false)*.
3. Manually 2: Set of reader type with the method *setReaderType*. The constants of all reader types are listed in the interface *FedmIscReaderConst*.

5.1.2. Finalizing

In Java the garbage collector assumes the task of removing no longer needed objects. This works wonderfully in pure Java applications. But objects that were created in native code are not subject to the scrutiny of the garbage collector. Therefore the programmer must take over this work. In the class of this class library, this work is taken care of in one line: you invoke the reader class method *destroy* when you no longer need the reader object. If you omit this finalizing, you will get an exception no later than when the application is closed.

5.2. Administering the communications channels

Within the class library there are, with one exception, no classes for the communications channels. Instead methods are integrated in the reader class FedmiscReader: *connectCOM*, *connectUSB*, *connectTCP* open one channel respectively to the reader. *disconnect* is used to close this channel. For the serial port there is also the method *findBaudrate*, which detects a reader and correctly configures the port for the communications parameters (baud rate, frame).

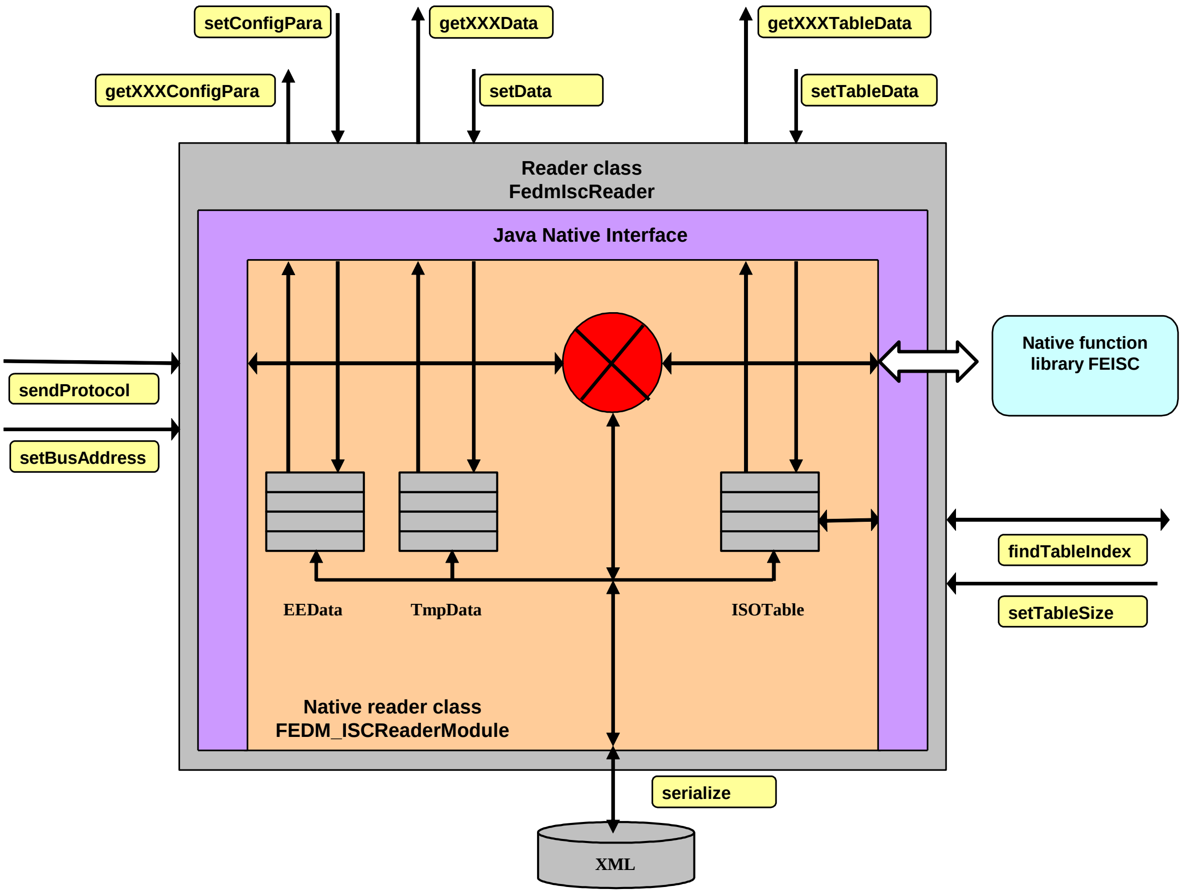
In the exceptional case that multiple USB readers have to be supported at the same time in an application, there is the class FeUsb, which provides special methods for this case.

Bluetooth Readers under Linux are administrated as /dev/rfcomm<x>. This makes the necessity to have a special connect method called *connectBT*. With Windows the call of *connectBT* will be redirected to *connectCOMM*.

5.3. Communication with the reader

5.3.1. Synchronous communication

The synchronous communication sequence in the reader class **FedmIscReader**, which is initiated by a host application, can be explained nicely in the following illustration: In the vertical dimension are the data flows that are moved using the (overloaded) methods *getXXXData*[5] and *setData*, as well as *getXXXTableData* and *setTableData*. In addition, the method *serialize* is sued to enable data flow between a reader object and a file.



In the horizontal axis is the control flow triggered by the method *sendProtocol*, the only communication method. This autonomously and internally gets all the necessary data from the integrated containers before outputting the send protocol and saves the received protocol data there. This means that the application program must write all the data needed for this protocol to the corresponding data containers and in the right locations before invoking *sendProtocol*. Likewise the receive data are stored at particular locations in corresponding data containers.

[5] XXX stands for Boolean, Byte, Integer, Long, String and represents the data types of the return value.

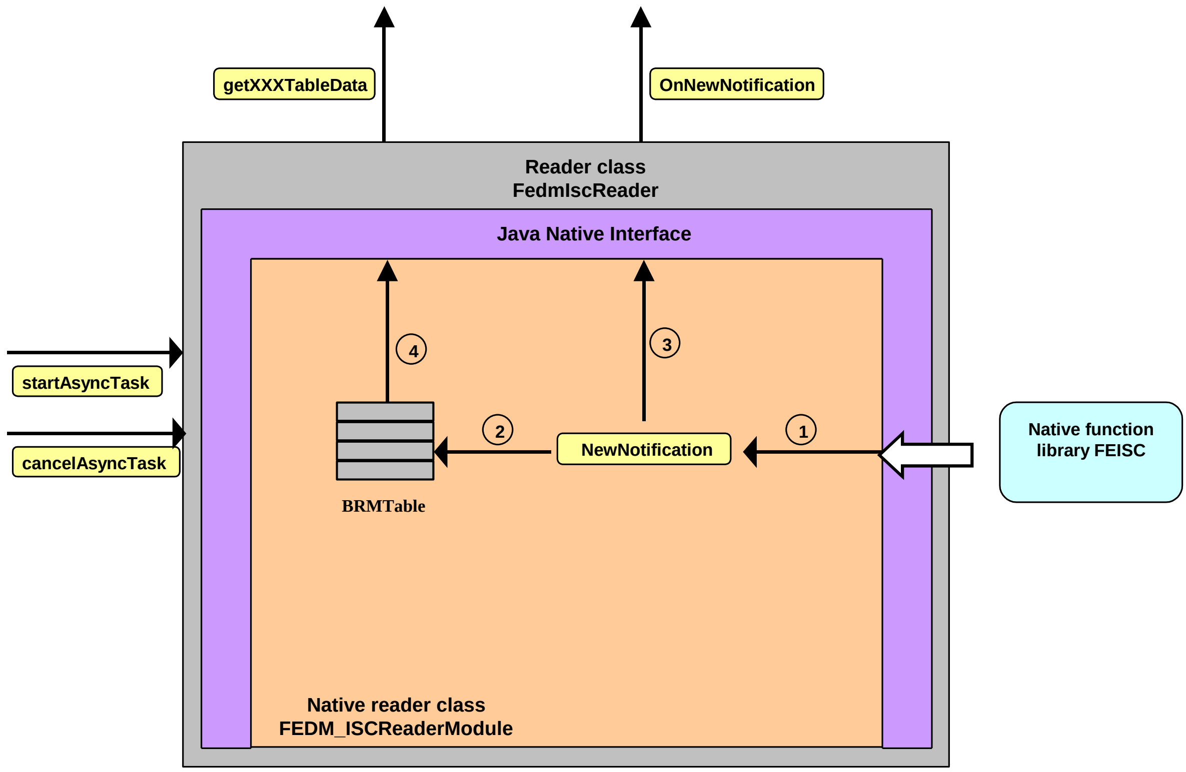
The key to the protocol data are so-called access constants for temporary protocol data (e.g. `FEDM_ISC_TMP_READER_INFO_MODE`) and the namespace OBID.ReaderConfig for reader configuration parameters (e.g. `de.feig.ReaderConfig.OperatingMode.Mode`). Anywhere from a few dozen to a hundred constants and names in the namespace OBID.ReaderConfig can be defined for each reader class. The structure is the same for all reader classes and is especially significant. This is explained in detail in 5.5.2. Access constants for temporary protocol data and 5.5.3. Reader Configuration Parameters in the Package de.feig.ReaderConfig. Since the access constants are of key significance for the entire function of the reader class, they are described in detail together with their use in section 5.6. Examples for using the method sendProtocol. The definition of each reader configuration parameter in the package de.feig.ReaderConfig is documented in the system manual of the reader.

The OBID®-Readers on the serial port are bus-compatible and require the bus address in the protocol. This should be set using the method *setBusAddress*.

5.3.2. Asynchronous communication

The asynchronous communication is initiated amongst others by the method **startAsyncTask** of the reader class **FedmIscReader** and is triggered by notification events of the reader. An asynchronous task initialled by **startAsyncTask** can only be used if the reader supports the Notification Mode or the asynchronous option for the Inventory command in the Host Mode[6]. Asynchronous tasks are also launched by the methods **sendTclApdu**, **sendSAMCommand** and **sendCommandQueue**. For each instance of **FedmIscReader** only one asynchronous task can be started.

The information flow can be explained nicely in the following illustration:



In the first step the notification is sent to the native part of the library. In the second step the transponder data are written into the table and the event method of the application is invoked (3rd step). Inside the event method (4th step) the application can use the overloaded methods getXXXTableData to query the information.

Transponder data from a reader working in Notification Mode will be written into the BRMTable. If the reader works in Host Mode the data are written into the ISOTable.

[6] The latter is only realized in the OBID® *classic-pro* Reader family

The table below lists the assignments of each listener methods to a task:

Task	Task-ID (FedmTaskOption)	Start Method	Listener Method (FedmTaskListener)
Single Inventory	ID_FIRST_NEW_TAG	startAsyncTask	onNewTag
Repetitiv Inventory	ID_EVERY_NEW_TAG	startAsyncTask	onNewTag
Notification	ID_NOTIFICATION	startAsyncTask	onNewNotification or onNewReaderDiagnostic
SAM communication	-	sendSAMCommand	onNewSAMResponse
Queue command	-	sendQueueCommand	onNewQueueResponse
T=CL APDU	-	sendTclApdu	onNewApduResponse

5.3.3. Secured data transmission with encryption

5.3.3.1. Overview

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 organisation openSSL (http://www.openssl.org), which are part of the library file libeay32.dll (Windows) rsp. 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 with FedmIscReader.connectTCP, transporting the enciphered password (password contains only nulls by default), to open a new session. Every successive protocol will then enciphered automatically.

Note: After the first authentication a new password should be saved in the Reader and a new authentication with the new password should be executed. This procedure – to switch into the crypto mode first and to change the password secondly – ensures that the new password will be transmitted enciphered! Otherwise the new password will be transmitted plain.

5.3.3.2. Feedback of error cases

A Reader with activated crypto mode ignores all plain protocols and returns the status 0x19 (Crypto Processing Error).

A Reader in plain mode ignores all enciphered protocols and returns the status 0x82 (Command not available).

An authentication into the Reader with a false password will be returned with status 0x12 (Authent Error).

A Reader with activated crypto mode signals with status 0x19 (Crypto Processing Error) an error case in the enciphered transmission. The Host must execute an authentication into the Reader again.

The error code -4093 or -4094 returned by FedmIscReader.sendProtocol signals a Host-side error case in the enciphered transmission. The Host must execute an authentication into the Reader again

The error code -4090 signals an error while loading the openSSL library file. Probably the library file is not installed or an incompatible version is installed.

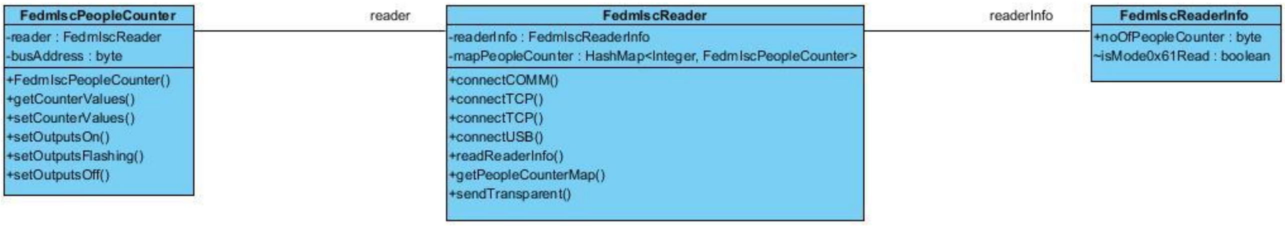
5.3.3.3. Notes for Programmers

Adding enciphered data transmission into a project needs only few aspects to be considered:

1. Every communication function of class FedmIscReader beginning with send… is prepared for plain and enciphered data transmission.
2. It is a requirement to link each OBID i-*scan*®- or OBID® *classic-pro* Reader with one Reader object exclusively, because every Reader object manages the individual session data.
3. Execution of a connection with FedmIscReader.connectTCP with authentication password is required.
4. If the Host application receives after a plain or enciphered data transmission the status 0x19 an authentication into the Reader is required.
5. If the error code -4093 or -4094 occurs in the Host application an authentication into the Reader is required.
6. In the Notification- and Access-Mode the data transmission is enciphered if the crypto mode is enabled in the Reader. Thus, the password must be added to the class FedmTaskOption.
7. If the crypto mode is disabled in the Reader configuration by a configuration protocol, the Reader object changes automatically back into the plain mode with the next plain protocol. This has the advantage that the existing Reader object can be maintained. A new connection is also not necessary.

5.4. Communication with a People Counter

A Reader detects all People Counters automatically after power-up. From host side, the information about the number of connected People Counters can be queried with the protocol [0x66] Get Reader Info with Mode-Byte 0x61 or with the method readReaderInfo() of the reader class FedmIscReader. Accordingly, all People Counter objects of type FedmIscPeopleCounter, collected in a HashMap, can be retrieved with the method getPeopleCounterMap(). The key of the HashMap is the bus address of the People Counter in the range 1…3.



Normally, the method ConnectUSB, ConnectTCP and (optional) ConnectCOM executes after a successful connection internally a ReadReaderInfo(). With it, all necessary information about the connected Reader and People Counter are stored in the reader class FedmIscReader. The HashMap with connected People Counter objects is already built and can be queried with getPeopleCounterMap() at once. The use of the class FedmIscPeopleCounter is explained in detail in the JavaDoc class reference as well as with an example in 5.11. Example for communicating with a People Counter.

5.5. Data containers

The task of the data containers is to administer all the reader parameters and temporary protocol data in a structured manner. Internally all data containers are organized as byte arrays in Motorola format (Big Endian). This format is compatible with any OBID®-Reader. Conversion into Intel format required for Intel-based PC's (Little Endian) is handled by the overloaded access methods.

The byte arrays are organized in 16-byte or 32-byte blocks. This organization also corresponds to that of the readers.

A total of 3 data containers are integrated

Data container	Description
EEData	for configuration parameters of the reader
RAMData	for temporary configuration parameters of the reader
TmpData	for general temporary protocol data

5.5.1. Data exchange

Access to the data is possible primarily using the overloaded methods *setData* and *getXXXData*[7]. Each method invocation can read or write exactly one parameter, which is identified by an access constant (see 5.5.2. Access constants for temporary protocol data).

Access to the configuration parameter is possible primarily using the overloaded methods *setConfigPara* and *getConfigParaAsXXX* .

The following section shows the use of *getXXXdata* and *setData* for various data types. The use of *getConfigParaAsXXX* and *setConfigPara* is analogous, but with the difference that instead of an access constant a string with the parameter name from the package de.feig.ReaderConfig must be passed.

5.5.1.1. Constant data

```
int iErr = setData(FedmIscReaderID.FEDM_ISC_TMP_READ_CFG_MODE, false);              // boolean
int iErr = setData(FedmIscReaderID.FEDM_ISC_TMP_READER_INFO_MODE, (byte)1);         // byte
int iErr = setData(FedmIscReaderID.FEDM_ISC_TMP_READER_INFO_MODE, (long)134);       // long
int iErr = setData(FedmIscReaderID.FEDM_ISC_TMP_READER_INFO_MODE, "0134");          // String
```

5.5.1.2. Data type boolean

```
boolean data = false;
data = getBooleanData(FedmIscReaderID.FEDM FEDM_ISC_TMP_INP_STATE_IN1);
```

[7] XXX stands for Boolean, Byte, Integer, Long, String and represents the data types of the return value.

```
int iErr = setData(FedmIscReaderID.FEDM_ISC_TMP_READ_CFG_MODE, data);
```

5.5.1.3. Data type byte

```
byte data = 1;
data = getByteData(FedmIscReaderID.FEDM_ISC_TMP_INP_STATE);
int iErr = setData(FedmIscReaderID.FEDM_ISC_TMP_READER_INFO_MODE, data);
```

5.5.1.4. Datentyp byte[]

```
byte[] data;
data = getByteArrayData(FedmIscReaderID.FEDM_ISC_TMP_SOFTVER_SW_REV);
iErr = setData(FedmIscReaderID.FEDM_ISCLR_TMP_READER_PW, data);
```

5.5.1.5. Data type int

```
int data = 0;
data = getData(FedmIscReaderID.FEDM_ISC_TMP_B0_RSP_DB_EXT_ADR_E);
int iErr = setData(FedmIscReaderID.FEDM_ISC_TMP_B0_REQ_DB_ADR_EXT, data);
```

5.5.1.6. Data type long

```
long data = 0;
data = getLongData(FedmIscReaderID.FEDM_ISC_TMP_INP_STATE);
int iErr = setData(FedmIscReaderID.FEDM_ISC_TMP_READER_INFO_MODE, data);
```

5.5.1.7. Data type String

ALL data that are read using a *getStringData* method are hex strings. This means for example that the numerical value 159 is not passed as "159" but rather as "9F". String values thus always consist of an even number of characters. The method collection in the class **FeHexConvert** (see 4.5.6. FeHexConvert)is provided for converting string values into other data types or the reverse.

To convert numerical values into string, which in the above example make up the number „159“, the Java library methods are recommended.

```
String data;
data = getStringData(FedmIscReaderID.FEDM_ISC_TMP_INP_STATE);
int iErr = setData(FedmIscReaderID.FEDM_ISC_TMP_READER_INFO_MODE, data);
```

5.5.2. Access constants for temporary protocol data

The access constants play a central role in data traffic between the application program and data containers of the reader class, as well as within the reader class between protocol method and data container. They identify the parameter and at the same time contain the coded storage location in one of the data containers.

An access constant is a string which generally has the following structure:

FC MM BN BY NB BI Nb

public static final String **FEDM_ISC_TMP_READER_INFO_MODE** “ **03 10 20 00 01 00 00**“

- Parameter name
- Memory type
- Reader type
- Component name

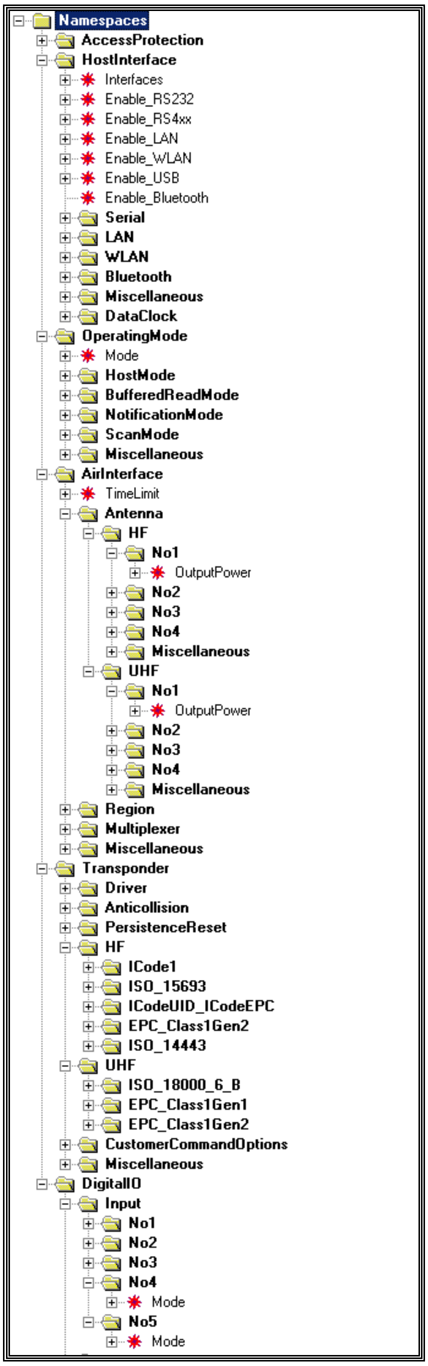
FC	Family Code (see Fedm)	**0..99**
MM	Memory Type (see Fedm)	**0..99**
BN	Block number	**0..99**
BY	1st byte in block	**0..99**
NB	No. of bytes	**0..99**
BI	1st bit in byte	**0..08**
Nb	No. of bits	**0..99**

These access constants are used exclusively with the methods *setData* and *getXXXData*. The access constant says nothing about the data type of a parameter. This is determined only by the data type of the access method. One can therefore read the bus address in the above example either as an integer or as a string or some other plausible data type (see 5.5.1. Data exchange).

All access constants are contained in the interface **FedmIscReaderID**.

5.5.3. Reader Configuration Parameters in the Package de.feig.ReaderConfig

The data exchange between an application and the data container for reader configuration parameters in the reader class is realized with overloaded methods which passes a string from the package de.feig.ReaderConfig representing the name of the configuration parameter. All names of reader configuration parameters of all OBID® readers are unified and divided in hierarchical order in groups and subgroups separated by a colon.



Detail of the tree order of the package de.feig.ReaderConfig

5.5.4. Management of the reader configuration

Each OBID i-*scan*® and OBID® *classic-pro* reader are 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 behaviour of the reader can be adapted to the application. Ideally, the program ISOStart 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 familiarise 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 bytes per configuration block. Each parameter cannot be addressed individually but must always be retrieved together with a configuration block using the protocol [0x80] Read Configuration, then modified and finally written back to the reader with the protocol [0x81] Write Configuration. This cycle must always be complied with and is also checked by the reader class FedmIscReader. 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 (public) byte array EEData for data from the EEPROM and RAMData 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.

Using the example of the configuration block CFG2 of the reader ID ISC.LR2000 which contains parameters for the configuration of the digital inputs and outputs, the following should explain how you specifically modify a parameter using the reader class FedmIscReader.

Byte	0	1	2	3	4	5	6
Contents	IDLE-MODE		FLASH-IDLE		IN-ACTIVE	0x00	REL1-TIME
Default	0x88A8		0xCC00		0x00		0x00

Byte	7	8	9	10	11	12	13
Contents	REL1-TIME	OUT1-TIME		REL2-TIME		REL3-TIME	REL4-TIME
Default	0x00	0x0000		0x0000		0x0000	

IDLE-MODE:

Defines the status of the signal emitters (OUT1 and RELx) during the idle mode.

Bit:	15	14	13	12	11	10	9	8
Function:	REL1 mode		0	0	OUT1 mode		0	0

7	6	5	4	3	2	1	0
REL2 mode		REL3 mode		REL4 mode		0	0

Mode	Function	
b 0 0	UNCHANGED	no effect on the status of the signal emitter
b 0 1	ON	signal emitter on
b 1 0	OFF	signal emitter off
b 1 1	FLASH	signal emitter alternating on

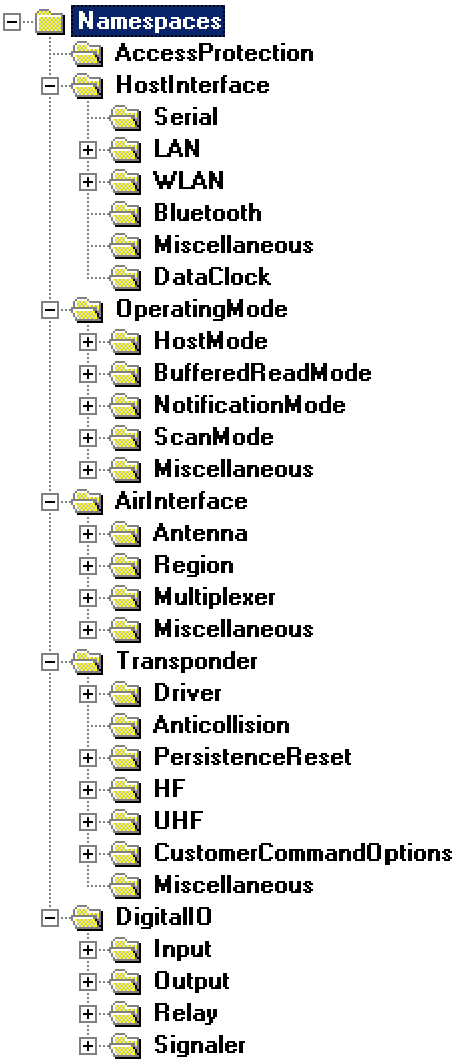
The assignment of the configuration block CFG2 is shown above. The parameter IDLE-MODE occupies two bytes and contains sub parameters for four relays and one digital output. Each output can be configured for one of four states according to the table. As the IDLE-MODE field is not greyed out, the modification can be made in the RAM of the reader.

The following steps are now necessary for the modification of REL1 mode inside IDLE-MODE:

```
// the example shows the reading, modification and rewriting of one block of the reader configuration
// reader is an object of the reader class FedmIscReader

byte CfgAdr = 2;            // Address of the configuration block
bool EEProm = false;        // Configuration data from/in RAM of the reader
int IdleModeRel1            // Parameter IDLE-MODE

// Defaults for the next sendProtocol
reader.setData(FEDM_ISC_TMP_READ_CFG, (byte)0x00);          // reset everything
reader.setData(FEDM_ISC_TMP_READ_CFG_ADR, CfgAdr);          // set address
reader.setData(FEDM_ISC_TMP_READ_CFG_LOC, EEProm);          // set memory location on RAM

// read configuration data
reader.sendProtocol(0x80);

IdleModeRel1 = 3;       // REL1 alternating on (Note: set frequency in Parameter IDLE-FLASH)

reader.setConfigPara(de.feig.ReaderConfig.DigitalIO.Relay.No1.IdleMode, IdleMode, false); // change value in RAM
```

```
// Defaults for the next sendProtocol
reader.setData(FEDM_ISC_TMP_WRITE_CFG, (byte)0x00);          // reset everything
reader.setData(FEDM_ISC_TMP_WRITE_CFG_ADR, CfgAdr);          // set address
reader.setData(FEDM_ISC_TMP_WRITE_CFG_LOC, EEProm);          // set memory location on EEPROM

// rewrite configuration data
reader.sendProtocol(0x81);
```

The methods *getConfigParaAsXXX* and *setConfigPara* receive a string with parameter name from the package de.feig.ReaderConfig. This package contains further interfaces in tree order and collects all parameter names of all OBID i-*scan*® and OBID® *classic-pro* reader in a unique manner. The picture below shows the main interface names.



The advantage of this schematic is the support by the intellisense functionality of modern IDEs which speeds-up the search for the proper parameter name.

5.5.5. Serializing

The integrated serializing method *serialize* allows saving of the reader configuration from the data containers to a file or loading the reader configuration from a file into data containers.

The standardizing of XML (Extensible Markup Language) has enabled an accepted description language for documents, which can be used independently of the computer language and operating systems. It therefore makes sense to use this language for defining the structure of a reader configuration file. Following is the content of an XML file that was created using the program ISOStart:


```xml
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<OBID>
   <file-header>
      <document-type>Reader Configuration File</document-type>
      <document-version>1.0</document-version>
      <reader-family>ISC</reader-family>
      <reader-name>ID ISC.MR100</reader-name>
      <reader-type>74</reader-type>
      <host-address>192.168.3.3</host-address>
      <port-number>10001</port-number>
      <communication-mode>TCP</communication-mode>
      <program-name>ID ISOStart</program-name>
      <program-version>05.03.03</program-version>
      <fedm-version>01.08</fedm-version>
      <date>07/18/03</date>
      <time>11:13:28</time>
   </file-header>
   <data-array name="Reader EEPROM-Parameter" blocks="16" size="16">
      <CFG0 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00" b10="00"
       b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG1 b0="00" b1="00" b2="08" b3="01" b4="00" b5="00" b6="00" b7="0A" b8="00" b9="00" b10="00"
      b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG2 b0="00" b1="20" b2="00" b3="25" b4="00" b5="04" b6="00" b7="2F" b8="0A" b9="64" b10="00"
      b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG3 b0="00" b1="39" b2="00" b3="07" b4="00" b5="00" b6="06" b7="00" b8="00" b9="00" b10="00"
      b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG4 b0="00" b1="00" b2="00" b3="00" b4="09" b5="00" b6="00" b7="00" b8="00" b9="00" b10="00"
      b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG5 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00" b10="00"
      b11="04" b12="00" b13="00" b14="00" b15="00"/>
      <CFG6 b0="00" b1="00" b2="00" b3="01" b4="00" b5="00" b6="00" b7="0A" b8="00" b9="00" b10="00"
      b11="05" b12="04" b13="00" b14="00" b15="00"/>
      <CFG7 b0="02" b1="20" b2="2C" b3="01" b4="0D" b5="00" b6="00" b7="00" b8="00" b9="00" b10="00"
      b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG8 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00" b10="00"
      b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG9 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00" b10="00"
      b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG10 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00"
      b10="00" b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG11 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00"
      b10="00" b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG12 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00"
      b10="00" b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG13 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00"
      b10="00" b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG14 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00"
      b10="00" b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG15 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00"
      b10="00" b11="00" b12="00" b13="00" b14="00" b15="00"/>
   </data-array>
   <data-array name="Reader RAM-Parameter" blocks="16" size="16">
      <CFG0 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00" b10="00"
      b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG1 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00" b10="00"
      b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG2 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00" b10="00"
      b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG3 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00" b10="00"
      b11="00" b12="00" b13="00" b14="00" b15="00"/>
```

```
      <CFG4 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00" b10="00"
      b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG5 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00" b10="00"
      b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG6 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00" b10="00"
      b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG7 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00" b10="00"
      b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG8 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00" b10="00"
      b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG9 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00" b10="00"
      b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG10 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00"
      b10="00" b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG11 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00"
      b10="00" b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG12 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00"
      b10="00" b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG13 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00"
      b10="00" b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG14 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00"
      b10="00" b11="00" b12="00" b13="00" b14="00" b15="00"/>
      <CFG15 b0="00" b1="00" b2="00" b3="00" b4="00" b5="00" b6="00" b7="00" b8="00" b9="00"
      b10="00" b11="00" b12="00" b13="00" b14="00" b15="00"/>
   </data-array>
</OBID>
```

Along with some header data, the tags `<data-array name="Reader EEPROM-Parameter" blocks="16" size="16">` and `<data-array name="Reader RAM-Parameter" blocks="16" size="16">` contain the reader parameters as hex values.

The *serialize* method can be used to create this file or read the reader configuration of such a file and place it in the internal memory EEData or RAMData. The prerequisite for generating the configuration file is that the entire reader configuration has first been read using *sendProtocol*.

To create a reader configuration file, use the call:

```
serialize(false, "c:\tmp\myreader.xml")
```

and to read the data from a reader configuration file, use the call:

```
serialize(true, "c:\tmp\myreader.xml")
```

5.6. Examples for using the method sendProtocol

The method *sendProtocol* of the reader class is vitally important for the protocol transfer. For this reason an example is shown for each control byte, which is intended to clarify which data are to be saved in data containers with which access constants before each protocol transfer, and which data are available after the protocol transfer. Some protocols allow various data to be transferred. In such a case only a typical example is shown.

All access constants are contained in the interface **FedmIscReaderID** and should be studied thoroughly together with the explanation of protocol data contained in the system manual for the Reader.

For reasons of clarity, the processes for evaluating return values and catching exceptions are omitted here. These processes should however always be performed in applications.

In the examples below it is assumed that the reader class **FedmIscReader** and the interfaces **FedmIscReaderID** and **FedmIscReaderConst** are incorporated:

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

The reader object shall be defined as:

```
FedmIscReader reader = new FedmIscReader;
```

[Control Byte] Protocol	Example
[0x18] Destroy EPC	byte mode = 0; // mode (always 0) byte len = 0; // number of bytes in EPC byte[] pw = new byte[3]; // password byte[] epc = new byte[32]; // buffer for EPC // take the data e.g. from input fields reader.**setData**(FEDM_ISC_TMP_EPC_DESTROY_MODE, (byte)0); reader.**setData**(FEDM_ISC_TMP_EPC_DESTROY_LEN, len); reader.**setData**(FEDM_ISC_TMP_EPC_DESTROY_PASSWORD, pw, 3); reader.**setData**(**FeMethod**.**getAdrOfID**(FEDM_ISC_TMP_DESTROY_EPC, 16), epc, len, TMP_DATA_MEM); reader.**sendProtocol**((byte)0x18);
[0x1A] Halt	reader.**sendProtocol**((byte)0x1A);
[0x1B] Reset QUIET Bit	reader.**sendProtocol**((byte)0x1B);
[0x1C] EAS	reader.**sendProtocol**((byte)0x1C);

[Control Byte] Protocol	Example
[0x21]Read Buffer	byte dataSets = 1; // Number requested Data sets byte trData = 0; // Data set structure byte recSets = 0; // Number of Data sets in protocol reader.**setData**(FEDM_ISCLR_TMP_BRM_SETS, dataSets); reader.**sendProtocol**((byte)0x21); // read data from transponder using Buffered Read Mode trData = reader.**getByteData**(FEDM_ISCLR_TMP_BRM_TRDATA); recSets = reader.**getByteData**(FEDM_ISCLR_BRM_RECSETS); // All other transponder datas are enclosed in the m_BRMTable. Example for data access in // 5.7.2. Examples for using the table for Buffered Read Mode
[0x22]Read Buffer	int dataSets = 1; // Number requested Data sets byte trData = 0; // Data set structure int recSets = 0; // Number of Data sets in protocol reader.**setData**(FEDM_ISC_TMP_ADV_BRM_SETS, dataSets); reader.**sendProtocol**((byte)0x22); // read data from transponder using Buffered Read Mode trData = reader.**getByteData**(FEDM_ISC_TMP_ADV_BRM_TRDATA1); recSets = reader.**getIntegerData**(FEDM_ISC_ADV_BRM_RECSETS); // All other transponder datas are enclosed in the m_BRMTable. Example for data access in // 5.7.2. Examples for using the table for Buffered Read Mode
[0x31] Read Data Buffer Info	int tabSize = 0; // Size of the data buffer int tabStart = 0; // Start address for the first data set int tabLen = 0; // Number of Data sets in the data buffer reader.**sendProtocol**((byte)0x31); tabSize = reader.**getIntegerData**(FEDM_ISCLR_TMP_TAB_SIZE); tabStart = reader.**getIntegerData**(FEDM_ISCLR_TMP_TAB_START); tabLen = reader.**getIntegerData**(FEDM_ISCLR_TMP_TAB_LEN);
[0x32] Clear Data Buffer	reader.**sendProtocol**((byte)0x32);
[0x33] Initialize Buffer	reader.**sendProtocol**((byte)0x33);
[0x34] Force Notify Trigger	reader.**sendProtocol**((byte)0x34);
[0x52] Baud Rate Detection	reader.**sendProtocol**((byte)0x52);
[0x55] Start Flash Loader	reader.**sendProtocol**((byte)0x55);
[0x63] CPU Reset	reader.**sendProtocol**((byte)0x63);
[0x64] System Reset	byte mode = 0; // LRU1000 RF-Controller (1 for LRU1000 AC-Controller) reader.**setData**(FEDM_ISC_TMP_SYSTEM_RESET_MODE, mode); reader.**sendProtocol**((byte)0x64);
[0x65] Get Software Version	String softVer = new String;// Software-version as string reader.**sendProtocol**((byte)0x65); softVer = reader.**getStringData**(FEDM_ISC_TMP_SOFTVER);
[0x66] Get Reader Info	String softVer = new String; // Software-version as string reader.**SetData**(FEDM_ISC_TMP_READER_INFO_MODE, (int)0); // identical with [0x65]

[Control Byte] Protocol	Example
	// reader.**setData**(FEDM_ISC_TMP_READER_INFO_MODE, (int1); // LRU1000: ACC reader.**sendProtocol**((byte)0x66); reader.**getData**(FEDM_ISC_TMP_SOFTVER, softVer); // identical with [0x65] **//** reader.**getData**(FEDM_ISC_TMP_FIRMWARE_VERSION, softVer); // LRU1000: ACC
[0x69] RF Reset	reader.**sendProtocol**((byte)0x69);
[0x6A] RF ON/OFF	byte RF = 1; // RF ON reader.**setData**(FEDM_ISC_TMP_RF_ONOFF, RF); reader.**sendProtocol**((byte)0x6A);
[0x6C] Set Noise Level	int NLMin = 500; // minimum Noise Level int NLAvg = 1000; // average Noise Level int NLMax = 1500; // maximum Noise Level reader.**setData**(FEDM_ISC_TMP_NOISE_LEVEL_MIN, NLMin); reader.**setData**(FEDM_ISC_TMP_NOISE_LEVEL_AVG, NLAvg); reader.**setData**(FEDM_ISC_TMP_NOISE_LEVEL_MAX, NLMax); reader.**sendProtocol**((byte)0x6C);
[0x6D] Get Noise Level	int NLMin = 0; // minimum Noise Level int NLAvg = 0; // average Noise Level int NLMax = 0; // maximum Noise Level reader.**sendProtocol**((byte)0x6D); NLMin = reader.**getIntegerData**(FEDM_ISC_TMP_NOISE_LEVEL_MIN); NLAvg = reader.**getIntegerData**(FEDM_ISC_TMP_NOISE_LEVEL_AVG); NLMax = reader.**getIntegerData**(FEDM_ISC_TMP_NOISE_LEVEL_MAX);
[0x6E] Reader Diagnostic	byte diagMode = 1; // Diagnostic-Mode reader.**setData**(FEDM_ISCLR_TMP_DIAG_MODE, diagMode); reader.**sendProtocol**((byte)0x6E);
[0x6F] Base Antenna Tuning	reader.**sendProtocol**((byte)0x6F); // the Long-Range-Reader changes into the special mode // the mode can be left only by performing a reset
[0x72] Set Output	// example from system manual ID ISC.LRU1000 reader.**setData**(FEDM_ISC_TMP_0x72_OUT_MODE, (byte)0x00); // set mode to 0 reader.**setData**(FEDM_ISC_TMP_0x72_OUT_N, (byte)0x03); // activate 3 outputs reader.**setData**(FEDM_ISC_TMP_0x72_OUT_NR_1, (byte)0x01); // output 1 reader.**setData**(FEDM_ISC_TMP_0x72_OUT_TYPE_1, (byte)0x00); // type: general output reader.**setData**(FEDM_ISC_TMP_0x72_OUT_MODE_1, (byte)0x03); // alternating reader.**setData**(FEDM_ISC_TMP_0x72_OUT_FREQ_1, (byte)0x01); // 4 Hz reader.**setData**(FEDM_ISC_TMP_0x72_OUT_TIME_1, (int)5); // 500 ms reader.**setData**(FEDM_ISC_TMP_0x72_OUT_NR_2, (byte)0x01); // relay 1 reader.**setData**(FEDM_ISC_TMP_0x72_OUT_TYPE_2, (byte)0x04); // type: relay reader.**setData**(FEDM_ISC_TMP_0x72_OUT_MODE_2, (byte)0x02); // switching off reader.**setData**(FEDM_ISC_TMP_0x72_OUT_FREQ_2, (byte)0x00); // unchanged reader.**setData**(FEDM_ISC_TMP_0x72_OUT_TIME_2, (int)2); // 200 ms reader.**setData**(FEDM_ISC_TMP_0x72_OUT_NR_3, (byte)0x02); // relay 2 reader.**setData**(FEDM_ISC_TMP_0x72_OUT_TYPE_3, (byte)0x04); // type: relay

[Control Byte] Protocol	Example
	reader.**setData**(FEDM_ISC_TMP_0x72_OUT_MODE_3, (byte)0x01); // switching on reader.**setData**(FEDM_ISC_TMP_0x72_OUT_FREQ_3, (byte)0x00); // unchanged reader.**setData**(FEDM_ISC_TMP_0x72_OUT_TIME_3, (int)10); // 1000 ms reader.**sendProtocol**((byte)0x72);
[0x71] Set Output	// Example 1 from the system manual ID ISC.M01 reader.**setData**(FEDM_ISCM_TMP_OUT_OS, (int)0); // OS-Bytes reset reader.**setData**(FEDM_ISCM_TMP_OUT_OS_OUT1, (byte)0x01); // Output 1 active reader.**setData**(FEDM_ISCM_TMP_OUT_OS_ LED_G, (byte)0x10); // LED green off reader.**setData**(FEDM_ISCM_TMP_OUT_OS_ LED_R, (byte)0x01); // LED red on reader.**setData**(FEDM_ISCM_TMP_OUT_OS_ BEEPER, (byte)0x11); // Beeper alternated on reader.**setData**(FEDM_ISCM_TMP_OUT_OSF, (int)0); // OSF-Bytes reset reader.**setData**(FEDM_ISCM_TMP_OUT_OSF_BEEPER, (byte)0x01); // Beeper with 4Hz reader.**setData**(FEDM_ISCM_TMP_OUT_OSTIME, (int)5); // 500ms active time Beeper and LED's reader.**setData**(FEDM_ISCM_TMP_OUT_OUTTIME, (int)3); // Output 1 300ms active reader.**sendProtocol**((byte)0x71);
[0x74] Get Input	// Example for ID ISC.LR bool in1 = false; // Input 1 bool in2 = false; // Input 2 bool dip1 = false; // Dip-Switch 1 bool dip2 = false; // Dip-Switch 2 bool dip3 = false; // Dip-Switch 3 bool dip4 = false; // Dip-Switch 4 reader.**sendProtocol**((byte)0x74); in1 = reader.**getBooleanData**(FEDM_ISC_TMP_ INP_STATE_IN1); in2 = reader.**getBooleanData** (FEDM_ISC_TMP_ INP_STATE_IN2); dip1 = reader.**getBooleanData** (FEDM_ISC_TMP_ INP_STATE_DIP1); dip2 = reader.**getBooleanData** (FEDM_ISC_TMP_ INP_STATE_DIP2); dip3 = reader.**getBooleanData** (FEDM_ISC_TMP_ INP_STATE_DIP3); dip4 = reader.**getBooleanData** (FEDM_ISC_TMP_ INP_STATE_DIP4);
[0x75] Adjust Antenna	int antValue = 0; // Antenna voltage reader.**sendProtocol**((byte)0x75); antValue = reader.**getIntegerData**(FEDM_ISCM_TMP_ ANTENNA_VALUE);
[0x80] Read Configuration und [0x81] Write Configuration	// The sample shows the read and write back function of one block in the reader configuration byte cfgAdr = 2; // Address of the configuration block bool eeProm = true; // Configuration datas from/for reader's EEPROM byte busAddress; // Bus address of the ISC.LR-reader from block 2 reader.**setData**(FEDM_ISC_TMP_READ_CFG, (byte)0x00); // reset all reader.**setData**(FEDM_ISC_TMP_READ_CFG_ADR, cfgAdr); // set address reader.**setData**(FEDM_ISC_TMP_READ_CFG_LOC, eeProm); // set Memory location to EEPROM reader.**sendProtocol**((byte)0x80); // read configuration data busAddress = reader.**getConfigParaAsByte**(ReaderConfig.HostInterface.Serial.BusAddress); reader.**setData**(FEDM_ISC_TMP_WRITE_CFG, (byte)0x00); // reset all reader.**setData**(FEDM_ISC_TMP_WRITE_CFG_ADR, cfgAdr); // set address

[Control Byte] Protocol	Example
	reader.**setData**(FEDM_ISC_TMP_WRITE_CFG_LOC, eeProm);// set Memory location to EEPROM reader.**sendProtocol**((byte)0x81); // write back configuration data
[0x82] Save Configuration	reader.**setData**(FEDM_ISC_TMP_SAVE_CFG, (byte)0x00); // reset all reader.**setData**(FEDM_ISC_TMP_SAVE_CFG_ADR, byte)0x00); // set address reader.**setData**(FEDM_ISC_TMP_SAVE_CFG_MODE, true); // save all blocks reader.**sendProtocol**((byte)0x82); // Save configuration data from RAM into EEPROM
[0x83] Set Default Configuration	reader.**setData**(FEDM_ISC_TMP_RESET_CFG, (byte)0x00); // reset all reader.**setData**(FEDM_ISC_TMP_RESET_CFG_ADR, (byte)0x02);// set address reader.**setData**(FEDM_ISC_TMP_RESET_CFG_LOC, false); // choose RAM reader.**setData**(FEDM_ISC_TMP_RESET_CFG_MODE, false); // set default only block 2 reader.**sendProtocol**((byte)0x83); // Set configuration data from block 2 in RAM to default
[0x85] Set System Timer	reader.**setData**(FEDM_ISCLR_TMP_TIME_H, (int)16); // 16 hours reader.**setData**(FEDM_ISCLR_TMP_TIME_M, (int)20); // 20 minutes reader.**setData**(FEDM_ISCLR_TMP_TIME_MS, (int)2000); // 2000 milliseconds reader.**sendProtocol**((byte)0x85); // set Timer
[0x86] Get System Timer	int hour = 0; // hours int minute = 0; // minutes int milliSec = 0; // milliseconds reader.**sendProtocol**((byte)0x86); // read timer hour = reader.**getIntegerData**(FEDM_ISCLR_TMP_TIME_H); // take over hours minute = reader.**getIntegerData**(FEDM_ISCLR_TMP_TIME_M); // take over minutes milliSec = reader.**getIntegerData**(FEDM_ISCLR_TMP_TIME_MS); // take over milliseconds
[0x87] Set System Date	reader.**setData**(FEDM_ISC_TMP_DATE_CENTURY, (byte)20); // 20th century reader.**setData**(FEDM_ISC_TMP_DATE_YEAR, (byte)4); // year 04 reader.**setData**(FEDM_ISC_TMP_DATE_MONTH, (byte)9); // September reader.**setData**(FEDM_ISC_TMP_DATE_DAY, (byte)15); // 15th September reader.**setData**(FEDM_ISC_TMP_DATE_TIMEZONE, (byte)0); // actually unused reader.**setData**(FEDM_ISC_TMP_DATE_HOUR, (byte)12); // hour reader.**setData**(FEDM_ISC_TMP_DATE_MINUTE, (byte)00); // minute reader.**setData**(FEDM_ISC_TMP_DATE_MILLISECOND, (int)0); // milliseconds (with seconds) reader.**sendProtocol**((byte)0x87); // set date and time
[0x88] Get System Date	byte century = 0; byte year = 0; byte month = 0; byte day = 0; byte timezone = 0; byte hour = 0; byte minute = 0; int milliSec = 0; reader.**sendProtocol**((byte)0x88); // read date and time century = reader.**getByteData**(FEDM_ISC_TMP_DATE_CENTURY); year = reader.**getByteData**(FEDM_ISC_TMP_DATE_YEAR); month = reader.**getByteData**(FEDM_ISC_TMP_DATE_MONTH); day = reader.**getByteData**(FEDM_ISC_TMP_DATE_DAY); timezone = reader.**getByteData**(FEDM_ISC_TMP_DATE_TIMEZONE);

[Control Byte] Protocol	Example
	hour = reader.**getByteData**(FEDM_ISC_TMP_DATE_HOUR); minute = reader.**getByteData**(FEDM_ISC_TMP_DATE_MINUTE); milliSec = reader.**getIntData**(FEDM_ISC_TMP_DATE_MILLISECOND);
[0xA0] Reader Login	byte[] passWord = new byte[4]; passWord[0] = 0x00; passWord[1] = 0x00; passWord[2] = 0x00; passWord[3] = 0x00; // Password reader.**setData**(FEDM_ISCLR_TMP_READER_PW, passWord); // set Password reader.**sendProtocol**((byte)0xA0); // send Password to the reader
[0xA2] Write Mifare Keys	Byte key[] = new byte[6]; // take the Mifare-Key e.g. from an input field reader.**setData**(FEDM_ISC_TMP_ISO14443A_KEY_TYPE, (byte)0); reader.**setData**(FEDM_ISC_TMP_ISO14443A_KEY_ADR, (byte)0); reader.**setData**(FEDM_ISC_TMP_ISO14443A_KEY, key); reader.**sendProtocol**((byte)0xB0); // send Mifare-Key to the reader
[0xAD] Write Reader Authent Key	Byte key[] = new byte[32]; // take the Authent-Key e.g. from an input field (32 chars containing hex-characters 0..9, A..F) reader.**setData**(FEDM_ISC_TMP_0xAD_KEY_TYPE, (byte)2); // AES256 reader.**setData**(FEDM_ISC_TMP_0xAD_KEY_LEN, (byte)32); reader.**setData**(FEDM_ISC_TMP_0xAD_KEY, key); reader.**sendProtocol**((byte)0xAD); // write Authent-Key into the reader
[0xB0] ISO Mandatory and Optional Commands	// the sample shows the [0x01] Inventory reader.**setData**(FEDM_ISC_TMP_B0_CMD, (byte)0x01); // Inventory reader.**setData**(FEDM_ISC_TMP_B0_MODE, (byte)0x00); // no More-Flag reader.**sendProtocol**((byte)0xB0); // the Inventory-data are in the m_ISOTable. Samples for data access in 5.7.1. Examples for using the table for ISO-Host Mode
[0xB1] ISO15693 Custumer and Proprietary Commands (only for Transponder from Philips Electronics N.V.)	// the sample shows the [0xA2] Set EAS // all others correspond to the 0xB1 commands String snr = new String; // for Serial number byte isoError = 0; // for ISO-Error code reader.**setData**(FEDM_ISC_TMP_B1_CMD, (byte)0xA2); // Set EAS reader.**setData**(FEDM_ISC_TMP_B1_MFR, (byte) ISO_MFR_PHILIPS); // Manufacturer reader.**setData**(FEDM_ISC_TMP_B1_MODE, (byte) ISO_MODE_ADR); // addressed // ... Serial number e.g. take from text field and store in sSnr reader.**setData**(FEDM_ISC_TMP_B1_REQ_UID, snr); int status = reader.s**endProtocol**((byte)0xB1); if(status == 0x95) { // take ISO-Error code

[Control Byte] Protocol	Example
	isoError = reader.**getIntegerData**(FEDM_ISC_TMP_B1_ISO_ERROR); }
[0xB2] ISO14443 Special Commands [0x2B] ISO14443-4 Transponder Info	byte FSCI = 0; byte FWI = 0; byte DSI = 0; byte DRI = 0; byte Nad = 0; byte Cid = 0; reader.**setData**(FEDM_ISC_TMP_B2_CMD, (byte)0x2B); // ISO14443-4 Transponder Info int status = reader.**sendProtocol**(0xB2); // transponder must previously selected with // [0x25] Select if(status == 0x00) { // get the table index of the selected transponder int idx = reader.**findTableIndex**(0, ISO_TABLE, DATA_IS_SELECTED, true); if(idx >= 0) { // get transponder data FSCI = reader.**getTableData**(idx, ISO_TABLE, DATA_FSCI) FWI = reader.**getTableData**(idx, ISO_TABLE, DATA_FWI) DSI = reader.**getTableData**(idx, ISO_TABLE, DATA_DSI) DRI = reader.**getTableData**(idx, ISO_TABLE, DATA_DRI) NAD = reader.**getTableData**(idx, ISO_TABLE, DATA_NAD) CID = reader.**getTableData**(idx, ISO_TABLE, DATA_CID) } }
[0xB2] ISO14443 Special Commands [0xB0] Authent Mifare	byte dbAddress = 0; // Address of the first data block byte keyType = 0; // Keytype for authentifikation byte keyAdr = 0; // EEPROM-Address of the Keys in the reader byte keyLocation = 0; // Location of the Authent-Key (0: Reader; 1: Protocol) String key = “000000000000”; // Authent-Key reader.**setData**(FEDM_ISC_TMP_B2_CMD, (byte)0xB0);// Authent Mifare reader.**setData**(FEDM_ISC_TMP_B2_MODE, (byte)0x00); // clear mode byte reader.**setData**(FEDM_ISC_TMP_B2_MODE, (byte)FEDM_ISC_ISO_MODE_SEL); //selected reader.**setData**(FEDM_ISC_TMP_B2_REQ_KEY_TYPE, keyType); reader.**setData**(FEDM_ISC_TMP_B2_REQ_DB_ADR, dbAddress); reader.**setData**(FEDM_ISC_TMP_B2_MODE_KL, keyLocation); if(keyLocation == 0) reader.**setData**(FEDM_ISC_TMP_B2_REQ_KEY_ADR, keyAdr); else reader.**setData**(FEDM_ISC_TMP_ISO14443A_KEY, key); reader.**sendProtocol**((byte)0xB2);
[0xB2] ISO14443 Special Commands [0xB1] Authent my-d	byte keyAdrTag = 5; // Address of the Keys on the Transponder byte keyAdrSam = 2; // Address of the Keys in the Authentifikation module byte cntAdr = 3; // Address of the Authtifikation counter byte authSeq = 0; // Authentifikation sequence reader.**setData**(FEDM_ISC_TMP_B2_CMD, (byte)0xB1); // Authent my-d reader.**setData**(FEDM_ISC_TMP_B2_MODE, (byte) FEDM_ISC_ISO_MODE_SEL); // selected

[Control Byte] Protocol	Example
	reader.**setData**(FEDM_ISC_TMP_B2_REQ_KEY_ADR_TAG, keyAdrTag); reader.**setData**(FEDM_ISC_TMP_B2_REQ_KEY_ADR_SAM, keyAdrSam); reader.**setData**(FEDM_ISC_TMP_B2_REQ_AUTH_COUNTER_ADR, cntAdr); reader.**setData**(FEDM_ISC_TMP_B2_REQ_KEY_AUTH_SEQUENCE, authSeq); reader.**sendProtocol**((byte)0xB2);
[0xB2] ISO14443 Special Commands [0x30] Mifare Value Commands	byte mfCmd = 0x01; // Mifare Command byte dbAdr = 0x05; // datablock address byte[] opValue = new byte[4]; // OP_VALUE byte destAdr = 0x05; // destination address opValue[0] = 0x00; opValue[1] = 0x00; opValue[2] = 0x00; opValue[3] = 0x03; reader.**setData**(FEDM_ISC_TMP_B2_CMD, (byte)0x30); // Mifare Value Commands reader.**setData**(FEDM_ISC_TMP_B2_MODE, (byte) FEDM_ISC_ISO_MODE_SEL); // selected reader.**setData**(FEDM_ISC_TMP_B2_REQ_MF_CMD, mfCmd); reader.**setData**(FEDM_ISC_TMP_B2_REQ_DB_ADR, dbAdr); reader.**setData**(FEDM_ISC_TMP_B2_REQ_OP_VALUE, opValue); reader.**setData**(FEDM_ISC_TMP_B2_REQ_DEST_ADR, destAdr); reader.**sendProtocol**(0xB2);

5.7. Tables

OBID *i-scan*® and OBID® *classic-pro* readers support protocols that can transport data for multiple transponders (ISO-Host Mode, Buffered Read Mode, Notification Mode) which make saving in the containers impossible. Ideally these data are structure in a table. The reader class **FedmIscReader** contains the tables ISOTable and BRMTable for these transponder data. Access to the table data is possible using the methods *getXXXTableData*[8], *setTableData* and *findTableIndex*. The methods *getTableSize*, *setTableSize*, *getTableLength* and *resetTable* are for table administration. In addition, the method *verifyTableData* can be used to perform a comparison of the sent with the received transponder data (ISO-Host Mode only).

Access to table data using the methods *setTableData* and *getXXXTableData* is also accomplished using the methods *getData* and *getData* for data containers. But they do not represent a string and therefore do not provide location coding. Instead, unambiguous identification of a table value is possible with the table index (idx) and the constants for the table type (tableID) and the table variable (dataID).

Example:

int **getIntegerTableData**(int index, int tableID, int dataID)

All constants for the table type and for the table variables are contained in the interface **FedmIscReaderConst**.

Alternately, tables can be output as table objects of type **FedmIsoTableItem**[] or **FedmBrmTableItem**[] using the method *getTable*. Use of the method interface of the table classes is analogous. Using the method *settable* you can also transfer a table created and filled in the application to the reader class and then write these data to the transponder.

The methods *getTableItem* and *setTableItem* permit the exchange of individual table elements.

Important note: A new reader object has unsized tables. You must therefore immediately set the size of your table using the method *setTableSize* (see 5.1.1. Initializing).

[8] XXX stands for Boolean, Byte, Integer, Long, String and represents the data types of the return value.

5.7.1. Examples for using the table for ISO-Host Mode

5.7.1.1. Anomaly of the addressed mode

Most of the Host Commands can be used in the addressed mode. In this case the serial number – or unified identifier (UID) – is part of the send protocol. In former versions the library has only supported UIDs with a length of 8 byte. With an extension flag in the mode byte (UID_LF) different UID length are now possible. If the UID_LF flag is set, the length of the UID must be added to the send protocol.

The following example demonstrates the use of a different UID length in a [0xB0][0xB23] Read Multiple Blocks:

```
// set UID for addressed mode (up to 32 byte)
reader.setData(FEDM_ISC_TMP_B0_REQ_UID, uid);
reader.setData(FEDM_ISC_TMP_B0_REQ_UID_LEN, uidLen);          // number of byte in UID

reader.setData(FEDM_ISC_TMP_B0_CMD, (byte)0x23);              // Command Read Multiple Blocks
reader.setData(FEDM_ISC_TMP_B0_MODE, (byte)0x00);             // clear mode byte
reader.setData(FEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01);         // addressed mode
reader.setData(FEDM_ISC_TMP_B0_MODE_UID_LF, true);            // UID_LF flag
reader.setData(FEDM_ISC_TMP_B0_REQ_DBN, (byte)0x01);          // request one data block
reader.setData(FEDM_ISC_TMP_B0_REQ_DB_ADR, dbAdr);            // set data block address

reader.sendProtocol(0xB0);     // communication wit reader/transponder
```

5.7.1.2. Examples for using the ISO table with [0xB0] Commands

[Control Byte] Protocol	Example
[0x01] Inventory for: HF-Transponder - Philips I-CODE1 - Texas Instruments Tag-it HF - ISO15693 - ISO14443A - ISO14443B - EPC (Electronic Product Code) - Philips I-CODE UID - Innovision Jewel - EPC Class1 Gen2 HF UHF-Transponder - ISO18000-6-A - ISO18000-6-B - EM4222 - EPC Class0/0+ - EPC Class1 Gen1 - EPC Class1 Gen2	byte trType = 0; // for Transponder type String snr = new String; // for Serial number (also EPC) String header = new String; // for EPC Header String domain = new String; // for EPC DomainManager-Field String object = new String; // for EPC ObjektClass-Field String epc = new String; // for EPC ("Header.DomainManager.ObjectClass.Serialnumber") reader.**setData**(FEDM_ISC_TMP_B0_CMD, (byte)0x01); // Command Inventory reader.**setData**(FEDM_ISC_TMP_B0_MODE, (byte)0x00); // no More-Flag // set table length to 0 and delete the content of the table completely reader.**deleteTable**(ISO_TABLE); reader.**sendProtocol**((byte)0xB0); // Communication with reader/transponder // All transponder data are in the table for(int cnt=0; cnt< reader.**getTableLength**(ISO_TABLE); ++cnt) { // get transponder typ trType = reader.**getByteTableData**(cnt, ISO_TABLE, DATA_TRTYPE); switch(trType) { case 0x00: // Philips I-CODE1 case 0x01: // Texas Instruments Tag-it HF

<table>
<tr><th>[Control Byte] Protocol</th><th>Example</th></tr>
<tr><td></td><td><pre>
        case 0x03: // ISO15693
        case 0x04: // ISO14443A
        case 0x05: // ISO14443B
        case 0x07: // I-Code UID
        case 0x08: // Innovision Jewel
        case 0x09: // EPC Class1 Gen2 HF
        case 0x81: // ISO18000-6-B
        case 0x83: // EM4222
        case 0x84: // EPC Class1 Gen2
        case 0x88: // EPC Class0/0+
        case 0x89: // EPC Class1 Gen1
            // get serial number as String
            snr = reader.getStringTableData(cnt, ISO_TABLE, DATA_SNR);
            break;
        case 0x06: // EPC (Electronic Product Code)
            // get EPC-Fields
            header = reader.getStringTableData(cnt, ISO_TABLE, DATA_EPC_HEADER);
            domain = reader.getStringTableData(cnt, ISO_TABLE, DATA_EPC_DOMAIN);
            object = reader.getStringTableData (cnt, ISO_TABLE, DATA_EPC_OBJECT);
            snr = reader.getStringTableData (cnt, ISO_TABLE, DATA_EPC_SNR);
            // or get EPC-Field as complete String
            epc = reader.getStringTableData (cnt, ISO_TABLE, DATA_EPC);
            break;
    }
}
</pre></td></tr>
<tr><td>[0x02] Stay Quiet</td><td><pre>
String snr = new String;   // for Serial number

// ... take serial number e.g. from text field and store it in snr

// set Serial number for Addressed Mode
reader.setData(FEDM_ISC_TMP_B0_REQ_UID, snr);

reader.setData(FEDM_ISC_TMP_B0_CMD, (byte)0x02);   // Command Stay Quiet
reader.setData(FEDM_ISC_TMP_B0_MODE, (byte)0x00);   // Mode-Byte reset
reader.setData(FEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01);   // Addressed Mode

reader.sendProtocol((byte)0xB0); // Communication with reader/transponder
</pre></td></tr>
<tr><td>[0x22] Lock Multiple Blocks</td><td><pre>
/* Attention: with this ISO Command all data blocks will be locked irreversible!!

String snr = new String;   // for Serial number

// ... take serial number e.g. from text field and store it in snr

// determine table index of the serial number
int idx = reader.findTableIndex(0, ISO_TABLE, DATA_SNR, snr);
if(idx < 0)
    return;

// set serial number for Addressed Mode
reader.setData(FEDM_ISC_TMP_B0_REQ_UID, snr);

reader.setData(FEDM_ISC_TMP_B0_CMD, (byte)0x22);   // Command Lock Multiple Blocks
reader.setData(FEDM_ISC_TMP_B0_MODE, (byte)0x00);   // Mode-Byte reset
reader.setData(FEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01);   // Addressed Mode
reader.setData(FEDM_ISC_TMP_B0_REQ_DBN, (byte)0x01);   // lock one Data block
reader.setData(FEDM_ISC_TMP_B0_REQ_DB_ADR, (byte)0x00); // set Data block-Address
</pre></td></tr>
</table>

[Control Byte] Protocol	Example
	reader.**sendProtocol**((byte)0xB0); // Communication with reader/transponder
[0x23] Read Multiple Blocks (standard address mode)	byte[] dataBlock; // Buffer for one data block byte dbAddress = 5; // Data block-address 5 String snr = new String; // for serial number // ... Serial number e.g. take from text field // set serial number for addressed mode reader.**setData**(FEDM_ISC_TMP_B0_REQ_UID, snr); reader.**setData**(FEDM_ISC_TMP_B0_CMD, (byte)0x23); // Command Read Multiple Blocks reader.**setData**(FEDM_ISC_TMP_B0_MODE, (byte)0x00); // Mode-Byte reset reader.**setData**(FEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01); // Addressed Mode reader.**setData**(FEDM_ISC_TMP_B0_REQ_DBN, (byte)0x01); // read one Data block reader.**setData**(FEDM_ISC_TMP_B0_REQ_DB_ADR, dbAddress); // set Data block-Address reader.**sendProtocol**((byte)0xB0); // Communication with reader/transponder // All Transponder data are content in the table // first determine the table index of the serial number int idx = reader.**findTableIndex**(0, ISO_TABLE, DATA_SNR, snr); if(idx < 0) return; // take the size of the data blocks (Block size) byte blockSize = reader.**getByteTableData**(idx, ISO_TABLE, DATA_BLOCKSIZE); // ... do something with the block size // take a data block (data block contents only the block size data byte) dataBlock = reader.**getByteArrayTableData**(idx, ISO_TABLE, DATA_RxDB, dbAddress); // ... do something with the data block
[0x23] Read Multiple Blocks (extended address mode)	byte[] dataBlock; // buffer for one data block byte dbAddress = 5; // data block address 5 String snr ; // for serial number String sPw; // for Access Passwort // ... take serial number e. g. from text field // ... take password e. g. from text field // // set serial number (> 8 Byte accpetable) for addressed mode reader.**setData**(FEDM_ISC_TMP_B0_REQ_UID, snr); reader.**setData**(FEDM_ISC_TMP_B0_REQ_UID_LEN, snr.length()/2); // length of UID in bytes reader.**setData**(FEDM_ISC_TMP_B0_CMD, (byte)0x23); // Command Read Multiple Blocks reader.**setData**(FEDM_ISC_TMP_B0_MODE, (byte)0x00); // clear mode byte reader.**setData**(FEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01); // Addr. Mode reader.**setData**(FEDM_ISC_TMP_B0_MODE_EXT_ADR, true);// extended addressed mode reader.**setData**(FEDM_ISC_TMP_B0_MODE_UID_LF, true); // length of UID != 8 reader.**setData**(FEDM_ISC_TMP_B0_BANK, (UCHAR)0x00); // clear bank nyte reader.**setData**(FEDM_ISC_TMP_B0_BANK_BANK_NR, (UCHAR)0x03); // bank User Memory reader.**setData**(FEDM_ISC_TMP_B0_BANK_ACCESS_FLAG, true); // with access password reader.**setData**(FEDM_ISC_TMP_B0_ACCESS_PW_LENGTH, (byte)sPw.length()/2);// len in bytes reader.**setData**(FEDM_ISC_TMP_B0_ACCESS_PW, sPw); // password reader.**setData**(FEDM_ISC_TMP_B0_REQ_DB_ADR_EXT, dbAddress); // datablock address reader.**setData**(FEDM_ISC_TMP_B0_REQ_DBN, (byte)0x01); // read one datablock

<table>
<tr><th>[Control Byte] Protocol</th><th>Example</th></tr>
<tr><td></td><td><pre>
reader.sendProtocol(0xB0);   // Communication with reader/transponder
// all transponder data are in the table
// first determine the table index of the serial number
int idx = reader.findTableIndex(0, ISO_TABLE, DATA_SNR, snr);
if(idx < 0)
      return;
// take the size of the data blocks (Block size)
byte blockSize;
reader.getTableData(idx, ISO_TABLE, DATA_BLOCKSIZE, out blockSize);
// ... do something with the block size
// take a data block (data block contents only the block size data byte)
reader.getTableData(idx, ISO_TABLE,DATA_RxDB, dbAddress, out dataBlock);
// ... do something with the data block
</pre></td></tr>
<tr><td>[0x24] Write Multiple Blocks

(standard address mode)</td><td><pre>
/* the example shows the [0x24] Write Multiple Blocks. In Addressed Mode an [0x01] Inventory must first be performed.

Note: If [0x23] Read Multiple Blocks was not yet carried out, then the block size is preset to 4. But if the transponder in the read field supports another block size, this must first be set in the table for this transponder!! You can use getBooleanTableData(.., DATA_IS_BLOCK_SIZE_SET) to check whether the block size was already read with [0x23] Read Multiple Blocks.
*/

byte[] dataBlock;   // Buffer for the data block
byte ucDBAdr = 5;   // Data block-address 5
String snr = new String;   // for serial number

// ... Serial number e.g. take from Text field and store it in snr

// ... Daten block e.g. take from Text field and store it in dataBlock

// determine table index of the serial-number
int idx = reader.findTableIndex(0, ISO_TABLE, DATA_SNR, snr);
if(idx < 0)
      return;

// set serial-number for Addressed Mode
reader.setData(FEDM_ISC_TMP_B0_REQ_UID, snr);

reader.setData(FEDM_ISC_TMP_B0_CMD, (byte)0x24);   // Command Read Multiple Blocks
reader.setData(FEDM_ISC_TMP_B0_MODE, (byte)0x00);   // Mode-Byte reset
reader.setData(FEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01);   // Addressed Mode
reader.setData(FEDM_ISC_TMP_B0_REQ_DBN, (byte)0x01);   // write one data block
reader.setData(FEDM_ISC_TMP_B0_REQ_DB_ADR, dbAddress); // set data block-address

reader.setTableData(idx, ISO_TABLE, DATA_BLOCK_SIZE, (byte)0x08); // set block size to 8

// write one data block (with blocksize of 8 bytes!) in the table
reader.setTableData(idx, ISO_TABLE, DATA_TxDB, ucDBAdr, dataBlock);

reader.sendProtocol((byte)0xB0); // Communication with reader/transponder
</pre></td></tr>
</table>

[Control Byte] Protocol	Example
[0x24] Write Multiple Blocks (extended address mode)	/* The example shows the [0x24] Write Multiple Block. In Addressed Mode an [0x01] Inventory must first be performed. Note: If [0x23] Read Multiple Blocks was not yet carried out, then the block size is preset to 4. But if the transponder in the read field supports another block size, this must first be set in the table for this transponder!! You can use GetTableData(.., DATA_IS_BLOCK_SIZE_SET) to check whether the block size was already read with [0x23] Read Multiple Blocks. */ byte[] dataBlock; // Buffer for the data block byte dbAddress = 5; // Data block-address 5 String snr; // for serial number String sPw; // for access password // ... Serial number e.g. take from Text field and store it in snr // ... data block e.g. take from Text field and store it in dataBlock // ... take password e. g. from text field // determine table index of the serial-number int idx = reader.**findTableIndex**(0, ISO_TABLE, DATA_SNR, snr); if(idx < 0) return; // set serial-number for Addressed Mode reader.**setData**(FEDM_ISC_TMP_B0_REQ_UID, snr); reader.**setData**(FEDM_ISC_TMP_B0_REQ_UID_LEN, snr.length()/2); // length of UID in byte reader.**setData**(FEDM_ISC_TMP_B0_CMD, (byte)0x24); // Command Read Multiple Blocks reader.**setData**(FEDM_ISC_TMP_B0_MODE, (byte)0x00); // clear mode byte reader.**setData**(FEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01);// addressed mode reader.**setData**(FEDM_ISC_TMP_B0_MODE_EXT_ADR, true);// extended addressed mode reader.**setData**(FEDM_ISC_TMP_B0_MODE_UID_LF, true); // length of UID != 8 reader.**setData**(FEDM_ISC_TMP_B0_BANK, (UCHAR)0x00); // clear bank nyte reader.**setData**(FEDM_ISC_TMP_B0_BANK_BANK_NR, (UCHAR)0x03); // bank User Memory reader.**setData**(FEDM_ISC_TMP_B0_BANK_ACCESS_FLAG, true); // with access password reader.**setData**(FEDM_ISC_TMP_B0_ACCESS_PW_LENGTH, (byte)sPw.length()/2);//Len in bytes reader.**setData**(FEDM_ISC_TMP_B0_ACCESS_PW, sPw); // password reader.**setData**(FEDM_ISC_TMP_B0_REQ_DB_ADR_EXT, dbAddress); // datablock address reader.**setData**(FEDM_ISC_TMP_B0_REQ_DBN, (byte)0x01); // write one data block reader.**setTableData**(idx, ISO_TABLE, DATA_BLOCK_SIZE, (byte)0x08); // set blocksize to e.g. 8 // write one data block (with blocksize of 8 bytes!) in the table reader.**setTableData**(idx, ISO_TABLE, DATA_TxDB, ucDBAdr, dataBlock); reader.**sendProtocol**(0xB0); // Communication with reader/transponder
[0x25] Select	String snr = new String; // for Serial-number // ... Serial number e.g. take from Text field and store it in snr // set Serial-number for Addressed Mode reader.**setData**(FEDM_ISC_TMP_B0_REQ_UID, snr); reader.**setData**(FEDM_ISC_TMP_B0_CMD, (byte)0x25); // Command Select reader.**setData**(FEDM_ISC_TMP_B0_MODE, (byte)0x00); // Mode-Byte reset reader.**setData**(FEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01); // Addressed Mode reader.**sendProtocol**((byte)0xB0); // Communication with reader/transponder

[Control Byte] Protocol	Example
[0x25] Select mit Option Card Information für ISO14443 Transponder	String snr = new String; // for Serial-number byte format = 0; // Format byte from response protocol // ... Serial number e.g. take from Text field and store it in snr // set Serial-number for Addressed Mode reader.**setData**(FEDM_ISC_TMP_B0_REQ_UID, snr); reader.**setData**(FEDM_ISC_TMP_B0_CMD, (byte)0x25); // Command Select reader.**setData**(FEDM_ISC_TMP_B0_MODE, (byte)0x00); // Mode-Byte reset reader.**setData**(FEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01); // Addressed Mode reader.**setData**(FEDM_ISC_TMP_B0_MODE_CINF, true); // CINF-Flag reader.**sendProtocol**((byte)0xB0); // Communication with reader/transponder // the Format byte is stored in TMPDATA_MEM format = reader.**getData**(FEDM_ISC_TMP_B0_RSP_FORMAT); // Format // the Card Information is stored in TMPDATA_MEM beginning at Index 2048 // the structur and length of the Card Information according to the system manual // the principle access looks like this: // byte[] cardInfo; // int length = s. system manual // cardInfo = reader. getByteArrayData(2048, length, TMPDATA_MEM);
[0x26] Reset to Ready	String snr = new String; // for serial-number // ... Serial number e.g. take from Text field and store it in snr // set serial-number for Addressed Mode reader.**setData**(FEDM_ISC_TMP_B0_REQ_UID, snr); reader.**setData**(FEDM_ISC_TMP_B0_CMD, (byte)0x26); // Command Reset to Ready reader.**setData**(FEDM_ISC_TMP_B0_MODE, (byte)0x00); // Mode-Byte reset reader.**setData**(FEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01); // Addressed Mode reader.**sendProtocol**((byte)0xB0); // Communication with reader/transponder
[0x27] Write AFI	String snr; // for Serial-number byte afi = 0; // for AFI // ... Serial number e.g. take from text field and store it in snr // ... AFI e.g. take from entry field and store it in afi // set serial-number for Addressed Mode reader.**setData**(FEDM_ISC_TMP_B0_REQ_UID, snr); // determine table index of the serial-number int idx = reader.**findTableIndex**(0, ISO_TABLE, DATA_SNR, snr); if(idx < 0) return; // write AFI in table reader.**setTableData**(idx, ISO_TABLE, DATA_AFI, afi); reader.**setData**(FEDM_ISC_TMP_B0_CMD, (byte)0x27); // Command Write AFI reader.**setData**(FEDM_ISC_TMP_B0_MODE, (byte)0x00); // Mode-Byte reset reader.**setData**(FEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01); // Addressed Mode reader.**sendProtocol**((byte)0xB0); // Communication with reader/transponder
[0x28] Lock AFI	String snr = new String; // for Serial number

[Control Byte] Protocol	Example
	// ... Serial number e.g. take from text field and store it in snr // set serial number for Addressed Mode reader.**setData**(FEDM_ISC_TMP_B0_REQ_UID, snr); reader.**setData**(FEDM_ISC_TMP_B0_CMD, (byte)0x28); // Command Lock AFI reader.**setData**(FEDM_ISC_TMP_B0_MODE, (byte)0x00); // Mode-Byte reset reader.**setData**(FEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01); // Addressed Mode reader.**sendProtocol**((byte)0xB0); // Communication with reader/transponder
[0x29] Write DSFID	String snr = new String; // for serial number Byte dsfid = 0; // for DSFID // ... Serial number e.g. take from text field and store it in snr // ... DSFID e.g. take from text field and store it in dsfid // set serial number for Addressed Mode reader.**setData**(FEDM_ISC_TMP_B0_REQ_UID, snr); // determine table index of the serial number int idx = reader.**findTableIndex**(0, ISO_TABLE, DATA_SNR, snr); if(idx < 0) return; // write DSFID in table reader.**setTableData**(idx, ISO_TABLE, DATA_DSFID, dsfid); reader.**setData**(FEDM_ISC_TMP_B0_CMD, (byte)0x29); // Command Write DSFID reader.**setData**(FEDM_ISC_TMP_B0_MODE, (byte)0x00); // Mode-Byte reset reader.**setData**(FEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01); // Addressed Mode reader.**sendProtocol**((byte)0xB0); // Communication with reader/transponder
[0x2A] Lock DSFID	String snr = new String; // for Serial number // ... Serial number e.g. take from text field and store it in snr // set serial number for Addressed Mode reader.**setData**(FEDM_ISC_TMP_B0_REQ_UID, snr); reader.**setData**(FEDM_ISC_TMP_B0_CMD, (byte)0x2A); // Command Lock DSFID reader.**setData**(FEDM_ISC_TMP_B0_MODE, (byte)0x00); // Mode-Byte reset reader.**setData**(FEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01); // Addressed Mode reader.**sendProtocol**((byte)0xB0); // Communication with reader/transponder
[0x2B] Get System Information	byte dsfid = 0; // for DSFID byte afi = 0; // for AFI byte[] ucMemSize = {0, 0}; // for Memory-Size byte icRef = 0; // for IC-Reference String snr; // for serial number // ... Serial number e.g. take from text field and store it in snr // set serial number for Addressed Mode reader.**setData**(FEDM_ISC_TMP_B0_REQ_UID, snr); reader.**setData**(FEDM_ISC_TMP_B0_CMD, (byte)0x2B); // Command Get System Information reader.**setData**(FEDM_ISC_TMP_B0_MODE, (byte)0x00); // Mode-Byte reset reader.**setData**(FEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01); // Addressed Mode

[Control Byte] Protocol	Example
	reader.**sendProtocol**((byte)0xB0); // Communication with reader/transponder // All transponder data content in the table // first determine table index of the serial number int idx = reader.**findTableIndex**(0, ISO_TABLE, DATA_SNR, snr); if(idx < 0) return; // get AFI afi = reader.**getByteTableData**(idx, ISO_TABLE, DATA_AFI); // ... do something with AFI // ... get all other data with the same procedure
[0x2C] Get Multiple Block Security Status	byte secStatus; // for Security Status String snr; // for Serial number // ... Serial number e.g. take from text field and store it in snr // set serial number for Addressed Mode reader.**setData**(FEDM_ISC_TMP_B0_REQ_UID, snr); reader.**setData**(FEDM_ISC_TMP_B0_REQ_DBN, (byte)0x05); // 5 Data blocks reader.**setData**(FEDM_ISC_TMP_B0_REQ_DB_ADR, (byte)0x00); // set 1. Data block-address reader.**setData**(FEDM_ISC_TMP_B0_CMD, (byte)0x2C); // Command Get Multiple Block // Security Status reader.**setData**(FEDM_ISC_TMP_B0_MODE, (byte)0x00); // Mode-Byte reset reader.**setData**(FEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01); // Addressed Mode reader.**sendProtocol**((byte)0xB0); // Communication with reader/transponder // All transponder data content in the table // first determine table index of the serial number int idx = reader.**findTableIndex**(0, ISO_TABLE, DATA_SNR, snr); if(idx < 0) return; // get the security status from block 0..4 for(int cnt=0; cnt<5; ++cnt) { secStatus = reader.**getByteTableData**(idx, ISO_TABLE, DATA_SEC_STATUS, cnt); // ... do something with secStatus }
[0xA0] Read Config Block	byte[] configBlock; // buffer for one Data block (Block size is always 4) byte cbAddress = 0; // Data block-Address 0 String snr; // for Serial number // ... Serial number e.g. take from text field and store it in snr // set serial number for Addressed Mode reader.**setData**(FEDM_ISC_TMP_B0_REQ_UID, snr); reader.**setData**(FEDM_ISC_TMP_B0_CMD, (byte)0xA0); // Command Read Configuration Block reader.**setData**(FEDM_ISC_TMP_B0_MODE, (byte)0x00); // Mode-Byte reset reader.**setData**(FEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01); // Addressed Mode reader.**setData**(FEDM_ISC_TMP_B0_REQ_CB_ADR, cbAddress); // set Data block-Address

[Control Byte] Protocol	Example
	reader.**sendProtocol**((byte)0xB0); // Communication with reader/transponder // All transponder data content in the table // first determine table index of the serial number int idx = reader.**findTableIndex**(0, ISO_TABLE, DATA_SNR, snr); if(idx < 0) return; // get the data block configBlock = reader.**getByteArrayTableData**(idx, ISO_TABLE, DATA_RxCB, cdAddress); // ... do something with the data block
[0xA1] Write Config Block	/* <u>Attention</u>: With this ISO Command you can change the configuration of the transponders and this can change the function of the transponder and so the transponder can be useless!! */ byte[] configBlock; // Buffer for a data block (Block size is always 4) byte cbAddress = 0; // Data block-Address 0 String snr; // for serial number // ... Serial number e.g. take from text field and store it in snr // ... take data block e.g. take it from a text field and store it in the configBlock // determine the table index of the serial number int idx = reader.**findTableIndex**(0, ISO_TABLE, DATA_SNR, snr); if(idx < 0) return; // set serial number for Addressed Mode reader.**setData**(FEDM_ISC_TMP_B0_REQ_UID, snr); reader.**setData**(FEDM_ISC_TMP_B0_CMD, (byte)0xA1); // Command Write Multiple Block reader.**setData**(FEDM_ISC_TMP_B0_MODE, (byte)0x00); // Mode-Byte reset reader.**setData**(FEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01); // Addressed Mode reader.**setData**(FEDM_ISC_TMP_B0_REQ_CB_ADR, cbAddress); // set data block-address // write a data block in the table reader.**setTableData**(idx, ISO_TABLE, DATA_TxCB, cbAddress, configBlock); reader.**sendProtocol**((byte)0xB0); // Communication with reader/transponder

5.7.1.3. Examples for using the ISO table with [0xB3] Commands

[Control byte] protocol	Example[9]
[0x18] Kill for UHF-Transponder: - EPC Class1 Gen1 - EPC Class1 Gen2	// Attention: with this command transponders are destroyed irretrievably! String epc; // for EPC String pw; // for Kill Password byte epcLen = 0; // length of EPC in byte byte pwLen = 0; // length of Kill Password // ... EPC e.g. take from text field and store it in epc, dito with the length // ... Kill Password e.g. take from text field and store it in pw, dito with the length // determine table index of the EPC int idx = reader.**findTableIndex**(0, ISO_TABLE, DATA_SNR, epc); // set EPC for addressed mode reader.**setData**(FEDM_ISC_TMP_B3_REQ_EPC, epc); reader.**setData**(FEDM_ISC_TMP_B3_REQ_EPC_LEN, epcLen); // length of EPC reader.**setData**(FEDM_ISC_TMP_B3_CMD, (byte)0x18); // Command Kill reader.**setData**(FEDM_ISC_TMP_B3_MODE, (byte)0x00); // reset mode byte reader.**setData**(FEDM_ISC_TMP_B3_MODE_ADR, (byte)0x01); // addressed mode reader.**setData**(FEDM_ISC_TMP_B3_MODE_EPC_LF, true); // EPC length flag reader.**setData**(FEDM_ISC_TMP_B3_KILL_PW_LENGTH, pwLen); // length of Kill Password reader.**setData**(FEDM_ISC_TMP_B3_KILL_PW, pw); // Kill Password reader.**sendProtocol**(0xB3); // communication with Reader/Transponder
[0x22] Lock Multiple Blocks for UHF-Transponder: - EPC Class1 Gen1 - EPC Class1 Gen2	// Attention: with this ISO Command all data blocks will be locked irretrievably! string epc; // for EPC string lockData; // for Lock data string pw; // for Access Password byte epcLen = 0; // length of EPC in byte byte trType = 0; // transponder type byte lockDataLen = 0; // length of Lock Data in byte byte pwLen = 0; // length of Access Password in byte // ... EPC e.g. take from text field and store it in epc, dito with the length // ... Lock Data e.g. take from text field and store it in lockData, dito with the length // ... Access Password e.g. take from text field and store it in pw, dito with the length // determine table index of the EPC int idx = reader.**findTableIndex**(0, ISO_TABLE, DATA_SNR, sEpc); // determine the transponder type trType = reader.**getByteTableData**(idx, ISO_TABLE, DATA_TRTYPE); // set EPC for addressed mode reader.**setData**(FEDM_ISC_TMP_B3_REQ_EPC, epc); reader.**setData**(FEDM_ISC_TMP_B3_REQ_EPC_LEN, epcLen); // length of EPC reader.**setData**(FEDM_ISC_TMP_B3_CMD, (byte)0x22); // Command Lock reader.**setData**(FEDM_ISC_TMP_B3_MODE, (byte)0x00); // reset mode byte reader.**setData**(FEDM_ISC_TMP_B3_MODE_ADR, (byte)0x01); // addressed mode

[9] all examples in C#

[Control byte] protocol	Example[9]
	reader.**setData**(FEDM_ISC_TMP_B3_MODE_EPC_LF, true); // EPC length flag reader.**setData**(FEDM_ISC_TMP_B3_REQ_TR_TYPE, trType); // transponder type reader.**setData**(FEDM_ISC_TMP_B3_LOCK_DATA_LENGTH, lockDataLen); // length of Lock // data reader.**setData**(FEDM_ISC_TMP_B3_LOCK_DATA, sLockData); // Lock data reader.**setData**(FEDM_ISC_TMP_B3_ACCESS_PW_LENGTH, pwLen); // length of Access // Password if(pwLen > 0) reader.**setData**(FEDM_ISC_TMP_B3_ACCESS_PW, sw); // Access Password reader.**sendProtocol**(0xB3); // communikation with Reader/Transponder
[0x24] Write Multiple Blocks for UHF-Transponder: - EPC Class1 Gen2	/* The example shows the [0x24] Write Multiple Block. In Addressed Mode an [0x01] Inventory must first be performed. Note: If [0x23] Read Multiple Blocks was not yet carried out, then the block size is preset to 4. But if the transponder in the read field supports another block size, this must first be set in the table for this transponder!! You can use GetTableData(.., DATA_IS_BLOCK_SIZE_SET) to check whether the block size was already read with [0x23] Read Multiple Blocks. */ byte[][] db; // buffer for Data (1. dimension for block number, 2. dimension für data) string epc; // for EPC string pw; // for optional Access Password byte epcLen = 0; // length of EPC in byte byte pwLen = 0; // length of optional Access Password // ... EPC e.g. take from Text field and store it in epc // ... Access Password e.g. take from text field and store it in pw, dito with the length // ... data block e.g. take from Text field and store it in db // determine table index of the EPC int iIdx = reader.**findTableIndex**(0, ISO_TABLE, DATA_SNR, epc); // set EPC for addressed mode reader.**setData**(FEDM_ISC_TMP_B3_REQ_UID, epc); reader.**setData**(FEDM_ISC_TMP_B3_REQ_EPC_LEN, epcLen); // length of EPC reader.**setData**(FEDM_ISC_TMP_B3_CMD, (byte)0x24); // Command Read Multiple // Blocks reader.**setData**(FEDM_ISC_TMP_B3_MODE, (byte)0x00); // reset mode byte reader.**setData**(FEDM_ISC_TMP_B3_MODE_ADR, (byte)0x01); // addressed mode reader.**setData**(FEDM_ISC_TMP_B3_MODE_EPC_LF, true); // EPC length flag reader.**setData**(FEDM_ISC_TMP_B3_MODE_EXT_ADR, true); // extended address mode reader.**setData**(FEDM_ISC_TMP_B3_BANK_ACCESS_FLAG, true); // Access Password flag reader.**setData**(FEDM_ISC_TMP_B3_BANK_BANK_NR, (byte)0x01); // EPC bank number reader.**setData**(FEDM_ISC_TMP_B3_REQ_DBN, (byte)0x06); // six data blocks to write reader.**setData**(FEDM_ISC_TMP_B3_REQ_DB_ADR_EXT, (uint)0); // first data block address reader.**setData**(FEDM_ISC_TMP_B3_REQ_DB_SIZE, (byte)0x02); // block size for command // set block size in table reader.**setTableData**(iIdx, ISO_TABLE, DATA_BLOCK_SIZE, (byte)2); // write data blocks in table for(int adr=0; adr<6; ++adr) reader.**setTableData**(idx, ISO_TABLE, DATA_TxDB, adr, db[adr]); reader.**sendProtocol**(0xB3); // communication with Reader/Transponder

5.7.2. Examples for using the table for Buffered Read Mode

[Control Byte] Protocol	Example
[0x21] Read Buffer	// this sample shows the reading of Data sets with serial number, data block and Timer-value byte dataSets = 1; // Number requested Data sets byte recSets = 0; // Number Data sets in Protocol byte[] dataBlock; // Buffer for a data block FeIscReaderTime time = 0; // for Timer-value String snr; // for serial number boolean snrFlag = false; // Flag for serial number in dataset boolean dbFlag = false; // Flag for data block in Dataset boolean timerFlag = false; // Flag for Timer in Datenset FedmBrmTableItem item; // a table entry with data for a transponder reader.**setData**(FEDM_ISCLR_TMP_BRM_SETS, dataSets); reader.**sendProtocol**((byte)0x21); // read blocks from transponder with Buffered Read Mode snrFlag = reader.**getBooleanData**(FEDM_ISCLR_TMP_BRM_TRDATA_SNR); dbFlag = reader.**getBooleanData**(FEDM_ISCLR_TMP_BRM_TRDATA_DB); timerFlag = reader.**getBooleanData**(FEDM_ISCLR_TMP_BRM_TRDATA_TIME); recSets = reader.**getByteData**(FEDM_ISCLR_TMP_BRM_RECSETS); // All transponder data content in the table for(int cnt=0; cnt< reader.**getTableLength**(BRM_TABLE); cnt++) { item = (FedmBrmTableItem) reader.**getTableItem**(cnt, BRM_TABLE); if(snrFlag) // get serial number snr = item.**getStringData**(DATA_SNR); if(dbFlag) // get data block 1 dataBlock = item.**getByteArrayData**(DATA_RxDB); if(timerFlag) // get Timer-value time = item.**getReaderTime**(); }
[0x22] Read Buffer	// this sample shows the reading of Data sets with serial number, antenna number and Timer-value byte[] dataBlock; // Puffer für einen Datenblock int dataSets = 1; // Number requested data sets int recSets = 0; // Number of received data sets FeIscReaderTime time = 0; // for date and time value String snr; // for serial number String db; // for data blocks byte blockSize = 0; // for blocksize int dbn = 0; // for number of data blocks byte antennaNumber = 0; // for antenna number byte input = 0; // for input byte byte state = 0; // for status byte boolean snrFlag = false; // flag (in TR-DATA1) for serial number in dataset boolean dbFlag = false; // flag (in TR-DATA1) for data block in dataset boolean antFlag = false; // flag (in TR-DATA1) for antenna number in dataset boolean timeFlag = false; // flag (in TR-DATA1) for time in dataset boolean dateFlag = false; // flag (in TR-DATA1) for date in dataset

[Control Byte] Protocol	Example

```
boolean extFlag = false;      // EXTENSION flag (in TR-DATA1): signals, that a second TR-DATA
                              // byte is following, where additional flags continues the definition of a
                              // data set
boolean inputFlag = FALSE;    // flag (in TR-DATA2) for input and status byte in data set
FedmBrmTableItem item;        // a table entry with data for one transponder

reader.setData(FEDM_ISC_TMP_ADV_BRM_SETS, dataSets);

reader.sendProtocol((byte)0x22);  // read data from transponder with Buffered Read Mode

snrFlag = reader.getBooleanData(FEDM_ISC_TMP_ADV_BRM_TRDATA1_SNR);
antFlag = reader.getBooleanData(FEDM_ISC_TMP_ADV_BRM_TRDATA1_ANT);
timeFlag = reader.getBooleanData(FEDM_ISC_TMP_ADV_BRM_TRDATA1_TIME);
dateFlag = reader.getBooleanData(FEDM_ISC_TMP_ADV_BRM_TRDATA1_DATE);
extFlag = reader.getBooleanData(FEDM_ISC_TMP_ADV_BRM_TRDATA1_EXT);
inputFlag = reader.getBooleanData(FEDM_ISC_TMP_ADV_BRM_TRDATA2_INPUT);
recSets = reader.getIntegerData(FEDM_ISC_TMP_ADV_BRM_RECSETS);

// All transponder data content in the table
for(int cnt=0; cnt< reader.getTableLength(BRM_TABLE); cnt++)
{
    item = (FedmBrmTableItem) reader.getTableItem(cnt, BRM_TABLE);
    if(snrFlag)   // get serial number
        snr = item.getStringData(DATA_SNR);

    if(db)    // get all data blocks
    {
        // get number of data blocks
        dbn = item.getIntegerData(iCnt, DATA_DBN);
        // get the blocksize
        blockSize = item.getData(iCnt, DATA_BLOCK_SIZE);
        // get data blocks
        for(int i=0; i<dbn; ++i)
        {
            db = item.getByteArrayData(iCnt, DATA_RxDB, i);
            // do anything with the data blocks
        }
    }

    if(antFlag)       // get antenna number
        antennaNumber = item.getByteData(DATA_ANT_NR);

    if(timerFlag || dateFlag) // get date and/or time value
        time = item.getReaderTime();

    if(extFlag && inputFlag)     // get input and status byte
    {
        input = item.getByteData(DATA_INPUT);
        state = item.getByteData(DATA_STATE);
    }
}
```

5.8. Example for using the method sendSAMCommand

The method *sendSAMCommand* of the reader class **FedmIscReader** executes an asynchronous communication with the connected OBID® *classic-pro* Reader.

For reasons of clarity, the processes for evaluating return values and catching exceptions are omitted here. These processes should however always be performed in applications.

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

For demonstration a test objekt myClass is defined which implements the interface FedmTaskListener. The constructor gets a reader object reader:

```
MyClass myClass = new MyClass(reader);

public class MyClass implements FedmTaskListener
{
   FedmIscReader reader;

   MyClass(FedmIscReader reader)
   {
      this.reader = reader;
   }

   public void send()
   {
      byte[] data = new byte[2];
      // Activate T=0
      data[0] = 0x01;
      data[1] = 0x01;

      // SAM-Slot 1, Timeout = 1s (10*100ms)
      // execute asynchronous communication
      reader.sendSAMCommand(this, 1, data, 10);
   }

   public void onNewSAMResponse(int error, byte[] responseData)
   {
      //if error = 0, responseData can contain data
   }
}
```

5.9. Example for using the method sendTclApdu

The method *sendTclApdu* of the reader class **FedmIscReader** executes an asynchronous communication with the connected OBID® *classic-pro* Reader.

For reasons of clarity, the processes for evaluating return values and catching exceptions are omitted here. These processes should however always be performed in applications.

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

For demonstration a test objekt myClass is defined which implements the interface FedmTaskListener. The constructor gets a reader object reader:

```
MyClass myClass = new MyClass(reader);

public class MyClass implements FedmTaskListener
{
   FedmIscReader reader;
   FedmCprApdu apdu =new FedmCprApdu(this);  // APDU objekt

   MyClass(FedmIscReader reader)
   {
      this.reader = reader;
   }

   public void send()
   {
      // prepare APDU objekt
      apdu.setCID(0);
      apdu.setNAD(0);

      // create APDU and add it to APDU objekt
      apdu.setApdu(buildApdu());

      // no use of CID and NAD
      // Timeout is calculated internally
      // execute asynchronous communication
      reader.sendTclApdu(false, false, apdu);
   }

   public void onNewApduResponse(int error)
   {
      //if error = 0, the APDU objekt can contain response data
      if(error)
         return;

      // access to the response data with method:
      // getLastResponseData
   }

}
```

5.10. Example for using the method sendCommandQueue

The method *sendCommandQueue* of the reader class **FedmIscReader** executes an asynchronous communication with the connected OBID® *classic-pro* Reader.

For reasons of clarity, the processes for evaluating return values and catching exceptions are omitted here. These processes should however always be performed in applications.

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

For demonstration a test objekt myClass is defined which implements the interface FedmTaskListener. The constructor gets a reader object reader:

```
MyClass myClass = new MyClass(reader);

public class MyClass implements FedmTaskListener
{
   FedmIscReader reader;
   FedmCprCommandQueue queue =new FedmCprCommandQueue(this); // Queue objekt

   MyClass(FedmIscReader reader)
   {
      this.reader = reader;
   }

   public void send()
   {
      String snr;              // UID (serial number) of Transponder

      // prepare Queue
      queue.clear();
      queue.setMode(0);
      queue.setTimeout(10);   // 1s (10*100ms)

      // 1. Command: [0xB0][0x25] Select
      reader.setDataFEDM_ISC_TMP_B0_REQ_UID, snr);            // set UID
      reader.setDataFEDM_ISC_TMP_B0_CMD, (byte)0x25);         // Command Select
      reader.setDataFEDM_ISC_TMP_B0_MODE, (byte)0x00);        // clear mode byte
      reader.setDataFEDM_ISC_TMP_B0_MODE_ADR, (byte)0x01);    // set addressed mode
      reader.addCommand(queue, 0x25);                         // add command to Queue
      // 2. Command: [0xB0][0x23] Read Multiple Blocks
      reader.setDataFEDM_ISC_TMP_B0_CMD, (byte)0x23);         // Read Multiple Blocks
      reader.setDataFEDM_ISC_TMP_B0_MODE, (byte)0x00);        // clear mode byte
      reader.setDataFEDM_ISC_TMP_B0_MODE_ADR, (byte)0x02);    // set selected mode
      reader.setDataFEDM_ISC_TMP_B0_REQ_DBN, (byte)0x01);     // number of data block
      reader.setDataFEDM_ISC_TMP_B0_REQ_DB_ADR, (byte)0x02); // first data block address
      reader.addCommand(queue, 0x23);                         // add command to Queue

      // execute asynchronous communication
      reader.sendCommandQueue(queue);
   }

   public void onNewQueueResponse(int error)
```

```
  {
    //if error = 0, the Queue objekt can contain response data
    if(error)
      return;

    // access to the response data with methods:
    // getLastCommandStep
    // getLastResponseCommand
    // getLastResponseStatus
    // getLastResponseData
  }
}
```

5.11. Example for communicating with a People Counter

```
FedmIscReader reader = new FedmIscReader();

…

// connecting the reader and execute readReaderInfo() internally
reader.connectTCP("192.168.10.10", 10001);

…

long[] values = null;

// query the map with all People Counter
HashMap<Integer, FedmIscPeopleCounter> mapPC = reader.getPeopleCounterMap();

// get People Counter object with busaddress 1
FedmIscPeopleCounter pc = mapPC.get(1);

if(pc != null)

{

   try

   {

      // query all counter values
      values = pc.getCounterValues();

   }

   catch(java.lang.Exception e)

   {

      // error handling

   }

}
```

6. Basic properties of the class FedmIscFunctionUnit

The reader class methods can be roughly divided into five categories:

a) Methods for initializing and finalizing

b) Methods for data containers

c) Methods for communication

d) Methods for child list management

6.1. Initializing und Finalizing

6.1.1. Initializing

Before using the function unit class for the first time, several initializations must be performed:

Address of the Function Unit	The address of the function unit object must be set with the method setPara(FEDM_ISC_FU_TMP_DAT_ADR, adr).

6.1.2. Finalizing

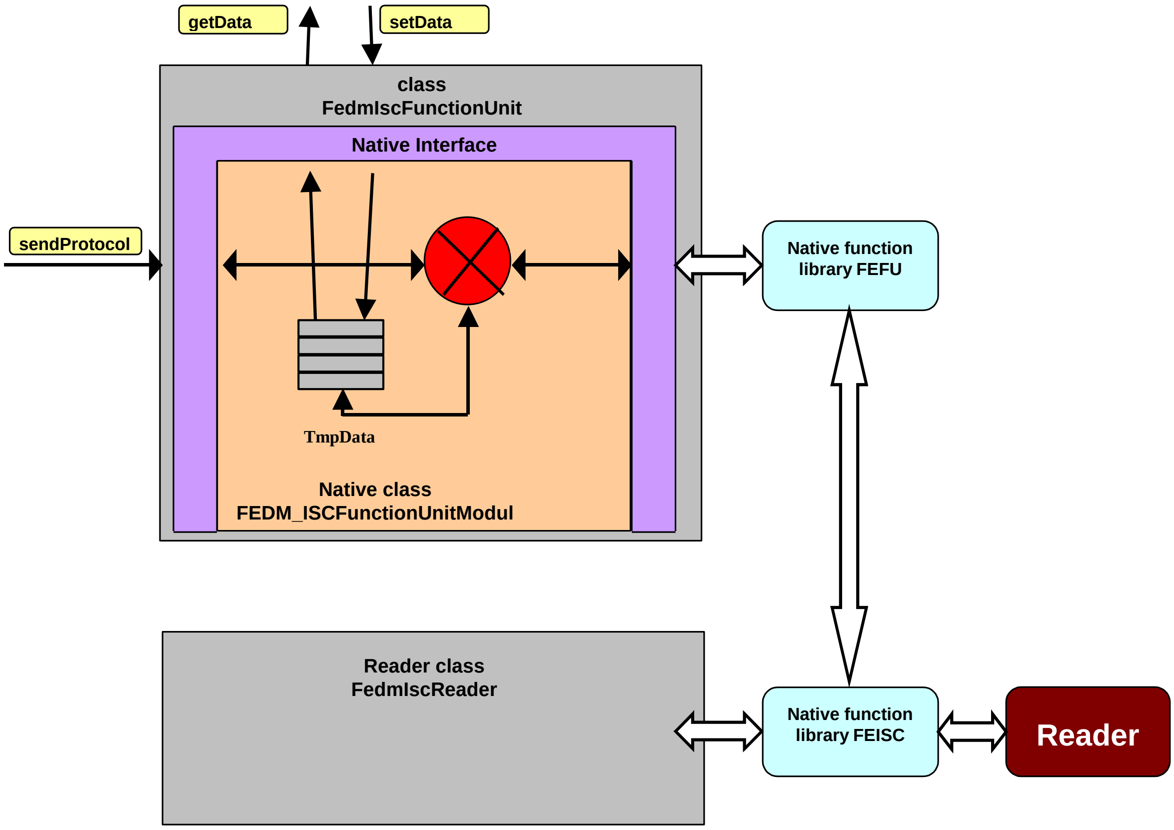
In Java the garbage collector assumes the task of removing no longer needed objects. This works wonderfully in pure Java applications. But objects that were created in native are not subject to the scrutiny of the garbage collector. Therefore the programmer must take over this work. In the class of this class library, this work is taken care of in one line: you invoke the reader class method *destroy* when you no longer need the reader object. If you omit this finalizing, you will get an exception no later than when the application is closed.

The *destroy* method destroys not the child function units objects, managed with the internal child table.

6.2. Communication with a function unit

The operation of the communication with a function unit is analog to the communication with a reader. This means that the application program must write **all** the data needed for this protocol to the data container TmpData and in the right locations **before** invoking *sendProtocol*. Likewise the receive data are stored at particular locations in data container TmpData.

The key to the protocol data are so-called access constants.



6.3. Examples for using the method sendProtocol

The method *sendProtocol* of the class is vitally important for the protocol transfer. For this reason an example is shown for each control byte, which is intended to clarify which data are to be saved in data containers with which access constants before each protocol transfer, and which data are available after the protocol transfer.

All access constants are contained in the interface **FedmIscFunctionUnitID** and should be studied thoroughly together with the explanation of protocol data contained in the system manual for the Function Unit.

For reasons of clarity, the processes for evaluating return values and catching exceptions are omitted here. These processes should however always be performed in applications.

In the examples below it is assumed that the Function Unit class **FedmIscFunctionUnit** and the interfaces **FedmIscFunctionUnitID** are incorporated:

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

The Function Unit object shall be defined as:

```
FedmIscFunctionUnit fu = new FedmIscFunctionUnit;
```

[Steuerbyte] Protokoll		Beispiel
ID ISC.DAT	[0xC0] Get Firmware Version	String firmware = new String; // buffer for firmware informations fu.**sendProtocol**((byte)0xC0); firmware = fu.**getStringData**(FEDM_ISC_FU_TMP_SOFTVER);
	[0xC1] CPU Reset	fu.**sendProtocol**((byte)0xC1);
	[0xC2] Set Capacities	fu.**setData**(FEDM_ISC_FU_TMP_DAT_ANT_VAL_C1, (byte)0xAB); // capacity 1 fu.**setData**(FEDM_ISC_FU_TMP_DAT_ANT_VAL_C2, (byte)0x9F); // capacity 2 fu.**sendProtocol**((byte)0xC2);
	[0xC3] Get Antenna Values	String antValues = new String; // buffer for tuning values fu.**sendProtocol**((byte)0xC3); antValues = fu.**getStringData**(FEDM_ISC_FU_TMP_DAT_ANT_VAL);
	[0xC4] Set Outputs	fu.**setData**(FEDM_ISC_FU_TMP_DAT_OUT, (byte)1); // switch output 1 fu.**sendProtocol**((byte)0xC4);
	[0xC5] Re-Tuning	fu.**sendProtocol**((byte)0xC5);
	[0xC6] Start Tuning	fu.**sendProtocol**((byte)0xC6);
	[0xC8] Store Settings	fu.**sendProtocol**((byte)0xC8);
	[0xC9] Detect	fu.**sendProtocol**((byte)0xC9);

[Steuerbyte] Protokoll		Beispiel
	[0xCA] Set Address	byte newAdr = 2; // new address fu.**setData**(FEDM_ISC_FU_TMP_DAT_NEW_ADR, newAdr); // new address for function unit fu.**sendProtocol**((byte)0xCA); // new address becomes valid fu.**setData**(FEDM_ISC_FU_TMP_DAT_ADR, newAdr); // set new address for communication
	[0xCB] Set Mode	fu.**setData**(FEDM_ISC_FU_TMP_DAT_MODE, (byte)1); // mode 1 fu.**sendProtocol**((byte)0xCB);
ID ISC.ANT.MUX	[0xDC] Detect	fu.**sendProtocol**(0xDC);
ID ISC.ANT.MUX	[0xDD] Select Channel	fu.**setData**(FEDM_ISC_FU_TMP_MUX_ OUT_CH 1, (byte)1); // set output 1 for input 1 fu.**setData**(FEDM_ISC_FU_TMP_MUX_ OUT_CH 2, (byte)8); // set output 8 for input 2 fu.**sendProtocol**((byte)0xDD);
ID ISC.ANT.MUX	[0xDE] CPU Reset	fu.**sendProtocol**((byte)0xDE);
ID ISC.ANT.MUX	[0xDF] Get Firmware Version	String firmware = new String; // buffer for firmware informations fu.**sendProtocol**((byte)0xDF); firmware = fu.**getStringData**(FEDM_ISC_FU_TMP_SOFTVER);
ID ISC.ANT.UMUX	[0xDC] Detect/Get Power	byte[] power; // buffer for Power Information byte UMuxState = 0; // statusbyte of response fu.**setData**(FEDM_ISC_FU_TMP_FLAGS, (byte)0); // set always to 0 fu.**sendProtocol** (0xDC); power = fu.**getByteArrayData**(FEDM_ISC_FU_TMP_UMUX_POWER); UMuxStatus = fu.**getByteData**(FEDM_ISC_FU_TMP_UMUX_LAST_STATE);
ID ISC.ANT.UMUX	[0xDD] Select Channel	byte UMuxState = 0; // statusbyte of response fu.**setData**(FEDM_ISC_FU_TMP_FLAGS, (byte)0); // set always to 0 fu.**setData**(FEDM_ISC_FU_TMP_MUX_OUT_CH1, (byte)1); // select output 1 fu.**sendProtocol**((byte)0xDD); UMuxStatus = fu.**getByteData**(FEDM_ISC_FU_TMP_UMUX_LAST_STATE);
ID ISC.ANT.UMUX	[0xDE] CPU Reset	byte UMuxState = 0; // statusbyte of response fu.**setData**(FEDM_ISC_FU_TMP_FLAGS, (byte)0); // set always to 0 fu.**sendProtocol**((byte)0xDE); UMuxStatus = fu.**getByteData**(FEDM_ISC_FU_TMP_UMUX_LAST_STATE);
ID ISC.ANT.UMUX	[0xDF] Get Firmware Version	byte[] Firmware; // buffer for Firmware Information byte UMuxState = 0; // statusbyte of response fu.**setData**(FEDM_ISC_FU_TMP_FLAGS, (byte)0); // set always to 0 fu.**sendProtocol**((byte)0xDF); Firmware = fu.**getByteArrayData**(FEDM_ISC_FU_TMP_SOFTVER); UMuxStatus = fu.**getByteData**(FEDM_ISC_FU_TMP_UMUX_LAST_STATE);

7. Error handling

7.1. Return value

Many methods in the class library perform internal error diagnostics and in case of an error return a negative value. The error codes for the Java class library ID OBIDISC4J have been directly taken from the native implementations. They are organized into ranges so that they do not overlap. The following ranges are reserved for the C++ class library ID FEDM and the native OBID®-function libraries:

Library	Value range for error codes
ID FEDM	-101 ... -999
ID FECOM	-1000…-1099
ID FEUSB	-1100…-1199
ID FETCP	-1200…-1299
ID FEISC	-4000…-4099
ID FEFU	-4199…-4100
ID FETCL	-4299…-4200

The method *getErrorText* of the reader class can be used to get an error text for the error code. The error code can also come from the area of a native OBID®-function library.

The last error code is saved internally and can be retrieved using the method *getLastError*.

7.2. Exceptions

Exceptions are generated in exceptional situations in the wrapper class for Java and during communication. The online documentation explains for each method whether and which exceptions are generated.

8. Appendix

8.1. Supported OBID® Readers

Reader	Notes
ID ISC.M02	
ID ISC.MR/PR100	all communication ports
ID ISC.PRH100/PRH101	all communication ports
ID ISC.MR/PR101	all communication ports
ID ISC.PRHD102	all communication ports
ID ISC.MR200	all communication ports
ID ISC.LR200	
ID ISC.LR2000	all communication ports
ID ISC.MRU200	all communication ports
ID ISC.LRU1000	all communication ports
ID ISC.LRU2000	all communication ports
ID ISC.LRU3000	all communication ports
ID CPR.02	
ID CPR.M02	all communication ports
ID CPR.04	all communication ports
ID CPR40.xx	all communication ports
ID CPR44.xx	all communication ports
ID CPR50.xx	all communication ports
ID MAX50.xx	all communication ports

8.2. Supported Transponders

The support of transponders depends on the implemented reader firmware. Please refer to the system manual of the reader.

The list below collects the transponder types, which are well-established during the development time of the library.

Transponder	Value	Notes
I-CODE 1	0x00	HF-Transponder
Tag-it	0x01	HF-Transponder
ISO15693	0x03	HF-Transponder
ISO14443-A	0x04	HF-Transponder
ISO14443-B	0x05	HF-Transponder
EPC	0x06	HF-Transponder (EPC-Types 1..4)
I-CODE UID	0x07	HF-Transponder
Innovision Jewel	0x08	HF-Transponder
EPC Class1 Generation 2 HF	0x09	HF-Transponder
STMicroelectronics SR176	0x0A	HF-Transponder
STMicroelectronics SRIxx	0x0B	HF-Transponder
Microchip MCRFxxx	0x0C	HF-Transponder
ISO18000-6-A	0x80	UHF-Transponder
ISO18000-6-B	0x81	UHF-Transponder
EM4222	0x83	UHF-Transponder
EPC Class1 Generation 2	0x84	UHF-Transponder
EPC Class0/0+	0x88	UHF-Transponder
EPC Class1 Generation 1	0x89	UHF-Transponder

8.3. Revision history

V3.02.03

- Support for Bluetooth Reader under Linux (see 5.2. Administering the communications channels)

V3.02.02

- Support for HF-Gates with People Counter **ID ISC.ANT1690/600-GPC** and **ID ISC.ANT1700/740-GPC**
- New reader configuration parameters in the package de.feig.ReaderConfig.
- Support for RSSI measurements in all Reader Modes for Reader **ID ISC.LRU3000**

V3.01.06

- Support for new Reader: **ID ISC.LRU3000**, **ID CPR44.xx**, **ID MAX50.xx**
- New reader configuration parameters in the package de.feig.ReaderConfig.
- New option for encrypted data transmission by use of openSSL library in the version 0.9.8l (s. 5.3.3. Secured data transmission with encryption).
- Extension of the class FedmIscReaderInfo to support new features with command [0x66] Reader Info.
- Sample Projects for Eclipse 3.5 and Netbeans 6.8

V3.00.11

- New reader configuration parameters in the package de.feig.ReaderConfig.
- Linux: Adaptions in native libraries to latest Kernel versions for serial communication to prevent timeouts.
- Linux: Class **FedmCprApdu** for asynchronous ISO14443-4 T=CL protocol exchange with OBID® *classic-pro* Reader supports now Linux too.

V3.00.07

- New methods in Reader class **FedmIscReader**: transferReaderCfgToXmlFile
transferXmlFileToReaderCfg
- New reader configuration parameters in the package de.feig.ReaderConfig.
- Modifications in the Reader class **FedmIscReader**:

- The methods connectUSB and connectTCP execute internally a readReaderInfo to collect important Reader properties.
- The method connectCOMM open a serial port and can optional, but recommended, execute internally a findBaudrate and after this, if the Reader is detected successfully, a readReaderInfo to collect important Reader properties.
- The method addCommand is renamed in addCommandToQueue
- The methode SendProtocol(0x72) use internally modified definitions of the constants `FEDM_ISC_TMP_0x72_OUT_TYPE_1...FEDM_ISC_TMP_0x72_OUT_TYPE_8`: Up to the previous release they adresses one bit. Now they addresses three bits. Thus, the OUT-TYPE 'Relay' must be set to 0x04 instead of 0x01 (5.6. Examples for using the method sendProtocol). This is applied to all reader types which supports the command [0x72] Set Output.

- Bugfix: Semaphore blocker in **FedmIscReader**.connectXXX methods solved.
- Bugfix: **FedmIscReader**.set/getTableItem methods evaluate different length of UID (serial number).

V3.00.04

- New reader configuration parameters in the package de.feig.ReaderConfig.
- Linux: the native libraries are linked against Libc V6 and Libstdc++ V6

V3.00.00

- Support for new reader: **ID ISC.MRU200**, **ID ISC.PRHD102** and **ID CPR40.xx.**
- The following older reader types are no longer supported: ID ISC.M01 and ID ISC.LR100.
- Support for UHF-Multiplexer **ID ISC.ANT.UMUX.**
- Support for transponder type EPC Class1 Gen2 HF.
- Automatic detection of version conflicts with dependent library files.
- New class **FedmIscReaderInfo** collecting important information from the connected reader.
- Collecting of all access constants for reader configuration in them package de.feig.ReaderConfig improves the clearness. Thus, the interfaces FedmIscReaderID_MR200, FedmIscReaderID_LR200, FedmIscReaderID_LR2000, FedmIscReaderID_LRU1000, FedmIscReaderID_LRU2000 as well as the access constants for OBID i-*scan*® Short- and Mid-Range reader and OBID® *classic-pro* reader in the interface FedmIscReaderID are removed.
- New overloaded methods getConfigParaAsXXX/setConfigPara in class **FedmIscReader** for modifying reader configuration parameters in the package de.feig.ReaderConfig.
- Writing of reader configuration is only possible for previous read configuration blocks except, if the reader configuration is load by a XML file.

- New high-level methods in **FedmIscReaderModule**
  - ApplyConfiguration
  - ReadCompleteConfiguration
  - WriteCompleteConfiguration
  - ResetCompleteConfiguration
  - ReadReaderInfo
- Removed methods in class **FedmIscReader**:
  - void **setByteContainer** (int arrayID, byte[] array)
  - byte[] **getByteContainer** (int arrayID)
  - int **setByteArrayData** (int address, byte[] data, int memID)
  - byte[] **getByteArrayData** (int address, int length, int memID)
- New class **FedmCprApdu** for asynchronous ISO14443-4 T=CL protocol exchange with OBID® *classic-pro* Reader (only for Windows).
- New class **FedmCprCommandQueue** for OBID® *classic-pro* Reader for asynchronous execution of [0xBC] Command Queue.

V2.05.07

- Support for USB reader
- Support for a new UHF reader command: [0x6B] Centralized RF Synchronization
- Linux: all native libraries are compiled with the GNU Compiler Collection 3.3.3.

V2.05.01

- Modified licence agreement
- Support for the UHF-Reader ID ISC.LRU2000.
- Extensions for the UHF-Reader ID ISC.LRU1000 concerning the configuration.
- New methods in the Reader class **FedmIscReader** for supporting asynchronous tasks.
- New interface **FedmTaskListener**
- New property class **FedmTaskOption**

V2.04.00

- New common constants for the UHF-Reader LRU1000:

Constant	Comment
FEDM_ISC_LRU1000_EE_SELMASK_EPC_CL1_GEN1_MASK_LGT	Constants for Selection Mask in the reader configuration for the transponder type EPC Class 1 Gen 1
FEDM_ISC_LRU1000_EE_SELMASK_EPC_CL1_GEN1_MASK_START_PTR	
FEDM_ISC_LRU1000_EE_SELMASK_EPC_CL1_GEN1_MASK	

Constant	Comment
FEDM_ISC_LRU1000_EE_SELMASK_EPC_CL1_GEN2_MASK_LGT	Constants for Selection Mask in the reader configuration for the transponder type EPC Class 1 Gen 2
FEDM_ISC_LRU1000_EE_SELMASK_EPC_CL1_GEN2_MASK_MODE	
FEDM_ISC_LRU1000_EE_SELMASK_EPC_CL1_GEN2_MASK_MODE_TRUNC	
FEDM_ISC_LRU1000_EE_SELMASK_EPC_CL1_GEN2_MASK_MODE_BANK	
FEDM_ISC_LRU1000_EE_SELMASK_EPC_CL1_GEN2_MASK_START_PTR	
FEDM_ISC_LRU1000_EE_SELMASK_EPC_CL1_GEN2_MASK_MSB	
FEDM_ISC_LRU1000_EE_SELMASK_ISO18000_6_B_MASK_LGT	Constants for Selection Mask in the reader configuration for the transponder type ISO18000-6-B
FEDM_ISC_LRU1000_EE_SELMASK_ISO18000_6_B_MASK_MODE	
FEDM_ISC_LRU1000_EE_SELMASK_ISO18000_6_B_MASK_START_PTR	
FEDM_ISC_LRU1000_EE_SELMASK_ISO18000_6_B_MASK	

V2.03.05

- The file **feusb.properties** is obsolete
- Support of the new HF-Reader ID ISC.LR2000
- Extensions for the UHF-Reader ID ISC.LRU1000 concerning the configuration
- Support for the new transponder types: HF-Transponder Innovision Jewel and UHF-Transponder EPC Class0/0+
- New communication methods in **FedmIscReader**:
  1. sendProtocol (byte cmdByte, String requestData)
  2. sendTransparent (String requestProtocol, boolean calcCrc)
- Support for the new protocol [0x72] Set Output
- New common constants:

Constant	Comment
FEDM_ISC_TMP_B0_MODE_CINF	Flag Card Information in Mode-Byte for [0xB0][0x25] Select
FEDM_ISC_TMP_B0_MODE_WR_NE	Flag Write-Erase in Mode-Byte for [0xB0][0x24] Write Multiple Blocks
FEDM_ISC_TMP_B0_RSP_FORMAT	Format Byte in response protocol of [0xB0][0x25] Select, if CINF-Flag is set
FEDM_ISC_TMP_B2_REQ_MF_CMD	Parameter for [0xB2][0x30] Mifare Value Commands
FEDM_ISC_TMP_B2_REQ_OP_VALUE	
FEDM_ISC_TMP_B2_REQ_DEST_ADR	
FEDM_ISC_TMP_ADV_BRM_TRDATA2	2. Byte of TR-DATA in response protocol of [0x22] Read Bufer
FEDM_ISC_TMP_ADV_BRM_TRDATA2_…	Flags in 2. Byte of TR-DATA in response protocol of [0x22] Read Bufer

Constant	Comment
FEDM_ISC_TMP_0x72_OUT...	Constants for [0x72] Set Output

- Modified common constants:

Old Constant	New Constant
FEDM_ISC_TMP_ADV_BRM_TRDATA1	FEDM_ISC_TMP_ADV_BRM_TRDATA
FEDM_ISC_TMP_ADV_BRM_TRDATA1_...	FEDM_ISC_TMP_ADV_BRM_TRDATA_...

V2.03.00

- Support for the new Reader ID ISC.MR200, ID ISC.MR/PR101 and ID CPR.M02-U
- Support for external Function Units with the class **FedmIscFunctionUnit**.
- The reader class **FedmIscReader** checks the version number of the native library OBIDISC4J.DLL/libobidisc4j.so. A version conflict throws an exception.
- The reader class **FedmIscReader** is no longer derived from the interfaces FedmIscReaderID, FedmIscReaderID_LR200 and FedmIscReaderID_LRU1000.
- New method in the reader class **FedmIscReader**: getNativeLibVersion
- Modifications in the table class **FedmBrmTableItem**: the datatype of the member blockCount is changed from byte into int. This causes adaptations in the get/set methods of this class. If you use up to now the method getByteData to query the value of the member blockCount (with constant DATA_DBN), then you must change the method call into getIntegerData.
- New member in the table class **FedmBrmTableItem**: member blockSize (access with constant DATA_BLOCK_SIZE) for supporting the Advanced Buffered Read Mode of the UHF-Reader ID ISC.LRU1000.
- New constants in the interface **FedmIscReaderConst** for UHF-Transponders.
- New constants for the UHF-Reader ID ISC.LRU1000 for the configuration CFG16 in interface **FedmIscReaderID_LRU1000**.
- New general constants:

Identifier	Comment
FEDM_ISC_TMP_DIAG_DATA	This identifier is valid for all reader diagnostic modes and substitutes the below listed removed constants for the modes 0x01, 0x02, 0x03.
FEDM_ISC_TMP_B3_…	Identifier for [0xB3] commands

- Modified general constants:

Old Identifier	New Identifier
FEDM_ISCLR_TMP_DIAG_MODE	FEDM_ISC_TMP_DIAG_MODE

- Removed general constants:

Identifier	Comment
FEDM_ISCLR_TMP_DIAG_0x01_DATA	Substituted throug FEDM_ISC_TMP_DIAG_DATA
FEDM_ISCLR_TMP_DIAG_0x02_DATA	Substituted throug FEDM_ISC_TMP_DIAG_DATA
FEDM_ISCLR_TMP_DIAG_0x03_DATA	Substituted throug FEDM_ISC_TMP_DIAG_DATA
FEDM_ISC_TMP_EPC_DESTROY_LEN	Removed, because of internal calculation of the length of EPC/UID based on the destroy mode and the header of the EPC.

V2.02.00

- Support for new reader ID ISC.MR200
- Protocols [0x18] Destroy supports new transponder type I-CODE UID.
- Remove of all access constants for data container RAMDATA_MEM. This reduces the number of constants dramatically. Alternatively, one can use the method toRAM of class FeMethods to modify the access constant for data container EEDATA_MEM.

V2.01.00

- Support for new reader ID ISC.LRU1000
- Support of new protocols [0x18] Destroy and [0x22] Read Buffer
- Support for advanced protocol frames (> 255 bytes)
- New helper class FeMethod with methods for access constants
- Rename of obid.dll (libobid.so) in obidisc4j.dll (libobidisc4j.so)
- Move of access constants concerning the configuration parameters in separate file for the readers ID ISC.LR200 and ID ISC.LRU1000

V2.00.05

Error correction in the class FeUsb

V2.00.04

Add of new item antennaNumber to the class FedmBrmTableItem

V2.00.03

This is the first release version.