IDENTIFICATION

FEIG
ELECTRONIC

TUTORIAL

ID ISC.SDK.xxx

The Art of Programming with C++ / C# / Java

Note:
This is not the current SDK documentation.
This document is for basic knowledge only and will not be updated.

preliminary – public (B)
2020-11-05 – Tutorial_H60810-e-ID-B.docx

Note

Content

Notes Concerning this Tutorial

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

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 This is necessary due to the constant updating of manuals, and it prevents confusion caused by information in out-of-date documents.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

1. Introduction

FEIG ELECTRONIC GmbH has developed different, hierarchical structured software libraries to simplify the integration of FEIG readers into customer's applications.

A common attribute of all components is the support of all FEIG reader families with a uniform Application Programming Interface (API).

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

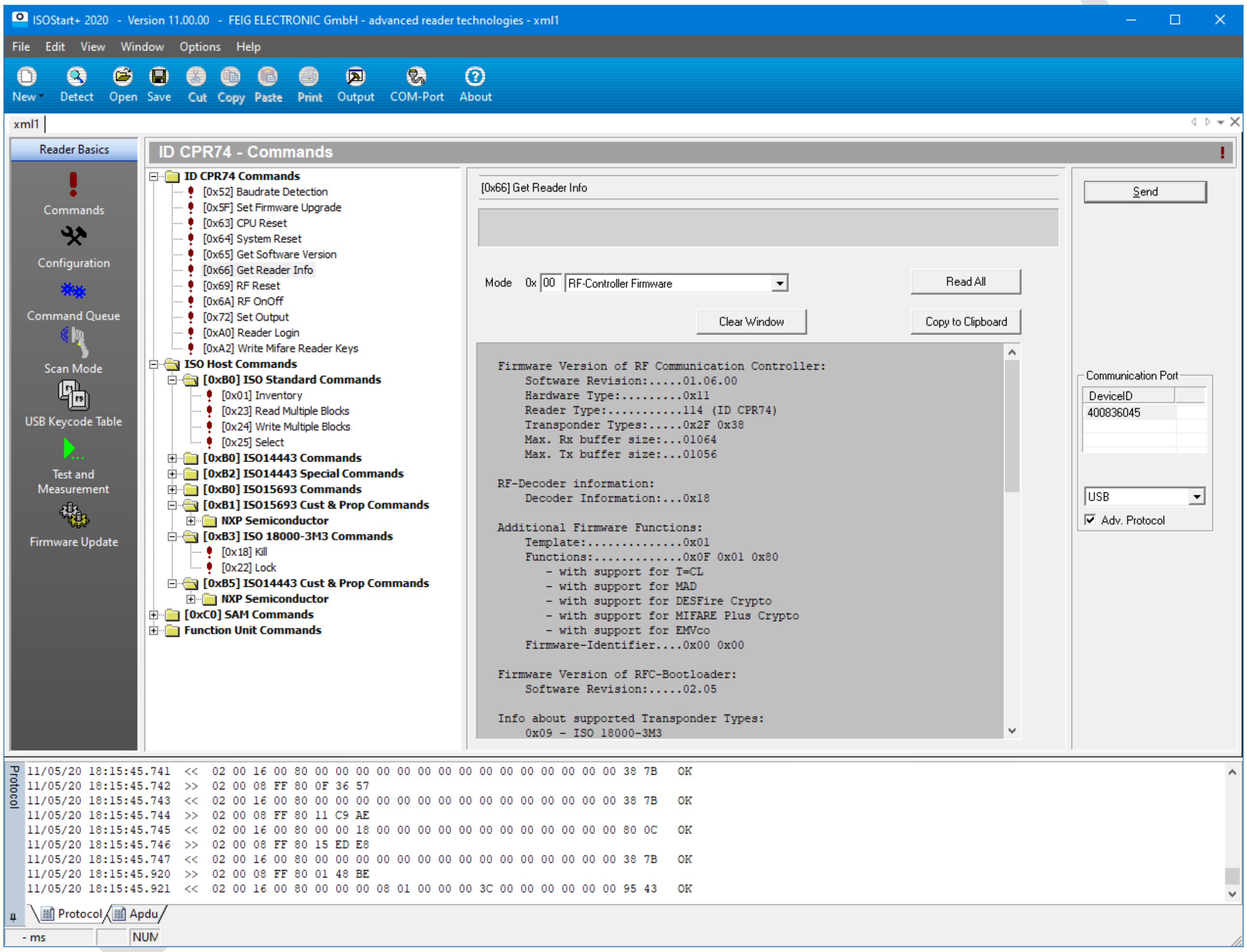
The class libraries are based upon function libraries, realizing the transport and protocol layers. These function libraries have a C interface and can be used with different programming languages.

The main intention of this tutorial is to give application programmers a structured introduction to the APIs for FEIG readers with the help of hands-on examples. The executable samples included in the SDKs offer advanced studies.

1.1. Standard Software Tools

1.1.1. The Application ISOStart+

The demo program **ISOStart+** is been developed to familiarize you with the functionality of the FEIG readers. Also, this tool can be used as reference applications to test intended interactions with FEIG readers and RFID-Transponders or to compare with the results of your application.



```
Firmware Version of RF Communication Controller:
    Software Revision:.....01.06.00
    Hardware Type:.........0x11
    Reader Type:...........114 (ID CPR74)
    Transponder Types:.....0x2F 0x38
    Max. Rx buffer size:...01064
    Max. Tx buffer size:...01056

RF-Decoder information:
    Decoder Information:...0x18

Additional Firmware Functions:
    Template:..............0x01
    Functions:.............0x0F 0x01 0x80
       - with support for T=CL
       - with support for MAD
       - with support for DESFire Crypto
       - with support for MIFARE Plus Crypto
       - with support for EMVco
    Firmware-Identifier....0x00 0x00

Firmware Version of RFC-Bootloader:
    Software Revision:.....02.05

Info about supported Transponder Types:
    0x09 - ISO 18000-3M3
```

```
11/05/20 18:15:45.741  <<  02 00 16 00 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 38 7B  OK
11/05/20 18:15:45.742  >>  02 00 08 FF 80 0F 36 57
11/05/20 18:15:45.743  <<  02 00 16 00 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 38 7B  OK
11/05/20 18:15:45.744  >>  02 00 08 FF 80 11 C9 AE
11/05/20 18:15:45.745  <<  02 00 16 00 80 00 00 18 00 00 00 00 00 00 00 00 00 00 00 00 80 0C  OK
11/05/20 18:15:45.746  >>  02 00 08 FF 80 15 ED E8
11/05/20 18:15:45.747  <<  02 00 16 00 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 38 7B  OK
11/05/20 18:15:45.920  >>  02 00 08 FF 80 01 48 BE
11/05/20 18:15:45.921  <<  02 00 16 00 80 00 00 00 08 01 00 00 00 3C 00 00 00 00 00 00 95 43  OK
```

Using this software you can:

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

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

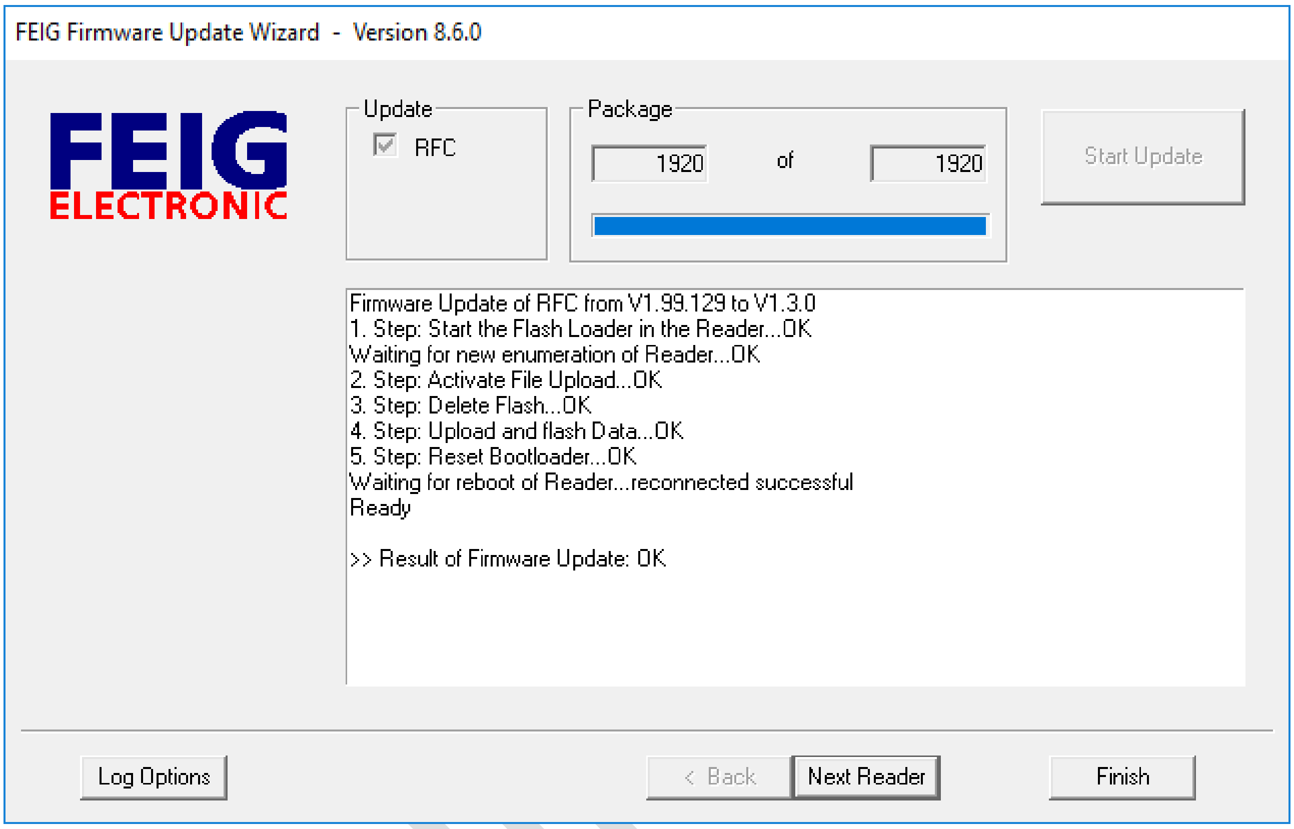
Unique features are:

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

1.1.2. Firmware-Update

The application Feig Firmware Update Wizard can be used to perform a firmware update with an FEIG reader. FEIG offers two versions:

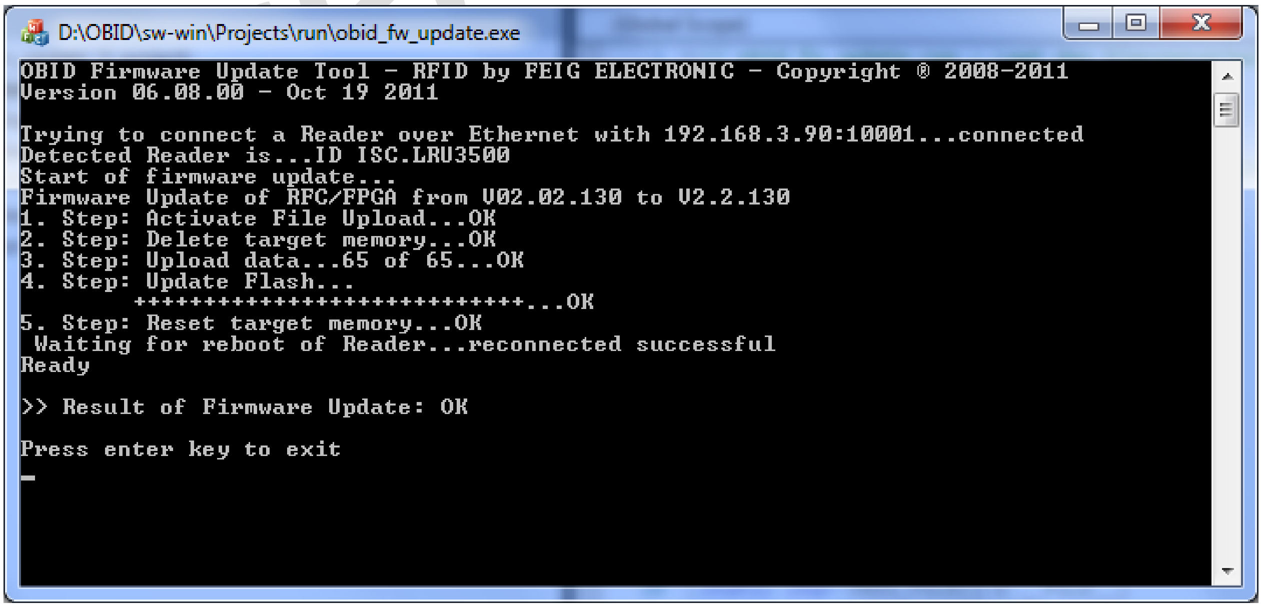
One tool with user interface (only for Windows).



```
Firmware Update of RFC from V1.99.129 to V1.3.0
1. Step: Start the Flash Loader in the Reader...OK
Waiting for new enumeration of Reader...OK
2. Step: Activate File Upload...OK
3. Step: Delete Flash...OK
4. Step: Upload and flash Data...OK
5. Step: Reset Bootloader...OK
Waiting for reboot of Reader...reconnected successful
Ready

>> Result of Firmware Update: OK
```

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



```
OBID Firmware Update Tool - RFID by FEIG ELECTRONIC - Copyright ® 2008-2011
Version 06.08.00 - Oct 19 2011

Trying to connect a Reader over Ethernet with 192.168.3.90:10001...connected
Detected Reader is...ID ISC.LRU3500
Start of firmware update...
Firmware Update of RFC/FPGA from V02.02.130 to V2.2.130
1. Step: Activate File Upload...OK
2. Step: Delete target memory...OK
3. Step: Upload data...65 of 65...OK
4. Step: Update Flash...
        ++++++++++++++++++++++++++++...OK
5. Step: Reset target memory...OK
 Waiting for reboot of Reader...reconnected successful
Ready

>> Result of Firmware Update: OK

Press enter key to exit
_
```

1.2. Standard Software Development Kits (SDK)

FEIG offers different SDKs for customers to support the fast integration of FEIG reader into their applications. An overview of all SDKs can be found in the document *ID_SDK_Presentation.pdf*.

2. Overview of all FEIG reader families

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

FEIG CPR	13,56 MHz reader for ISO 14443 compliant Transponders
FEIG HF RFID	13,56 MHz reader for ISO 15693 and ISO 18000-3M3 compliant Transponders
FEIG UHF RFID	860...960 MHz reader for EPC compliant Transponders

The similarities refer to:

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

2.1. Working Modes

Every FEIG reader supports at least the Host-Mode, which is a polling mode.

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

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

2.2. Communication Interfaces

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

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

2.3. Transmission Protocol

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

Reader ← Host

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

n-1	n
LSB CRC16	MSB CRC16

Host ← Reader

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

n-1	n
LSB CRC16	MSB CRC16

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

2.4. Reader-API

All FEIG readers support a subset of commands for controlling and configuring the reader.

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

2.5. Transponder-API

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

With Host-Mode, these are the obligatory ISO 15693, ISO 18000-3M3 and ISO 14443 commands. Some FEIG readers support additionally optional RF transparent commands, with which the manufacturer specific tag commands are forwarded by the reader without modifications. FEIG CPR readers support APDUs via the T=CL protocol with ISO 14443-4 compliant Transponders and have SAM-commands included.

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

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

2.6. Miscellaneous

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

3. Options for Integration

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

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

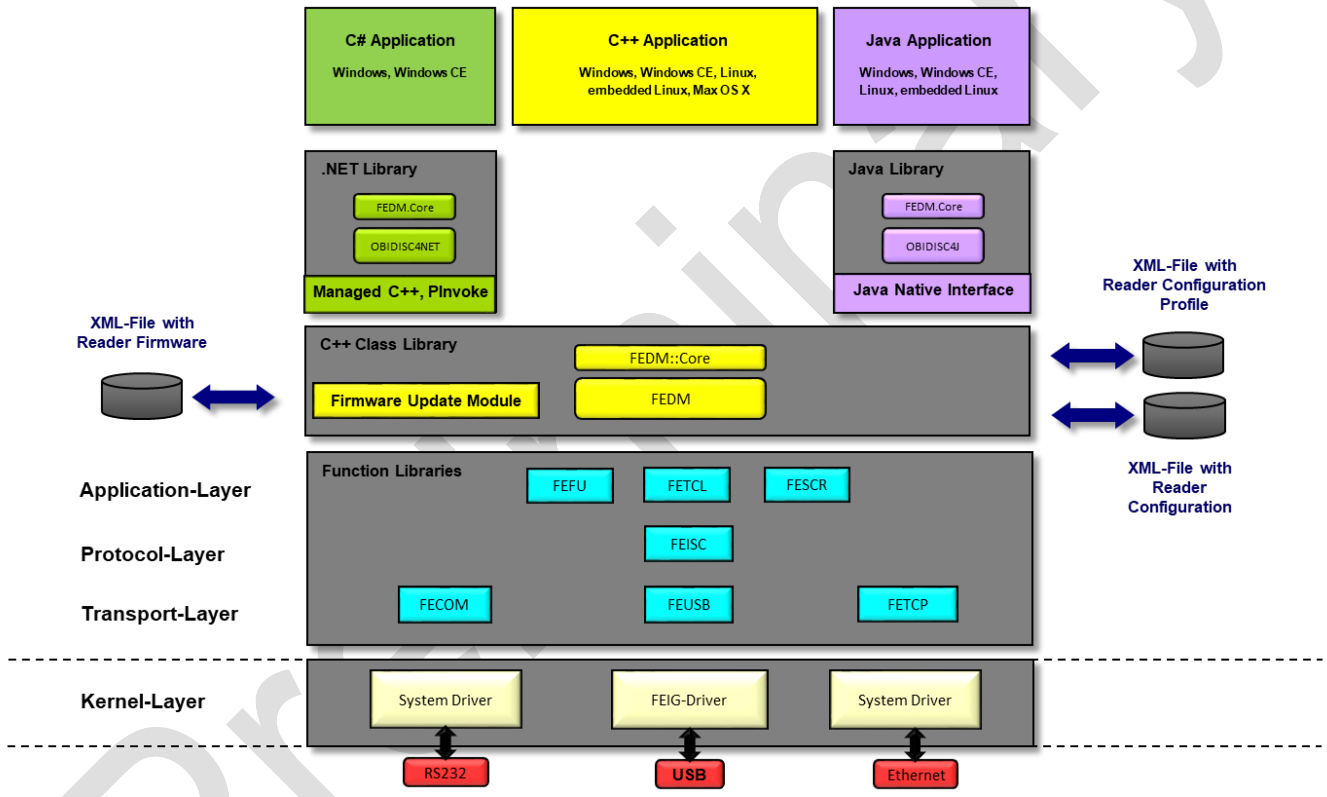


Fig. 1: Library stack from which the particular SDKs are built

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

3.1. Supported Operating Systems and programming languages

Standard SDKs are available for different operating systems. Actually the following OS are supported:

Operating System	Target		Programming language Support	Notes
	32-Bit	64-Bit		
Windows 10	X	X	C/C++, .NET, Java	
Linux	X	X	C/C++, Java	Intel CPUs only
Embedded Linux	X	X	C/C++, Java	On request
Raspberry PI 3 and 4	X	-	C/C++, Java	Raspbian OS (Stretch and Buster)
Android	X	X	C/C++, Java	Android 6.0 (API level 23) or higher

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

3.1.1. Standard-Driver for USB-reader

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

3.1.2. PC/SC-Driver for FEIG CPR reader

PC/SC is a standard for integrating smart cards and smart card readers. With Windows, PC/SC is realized as a Smart Card Library and part of the operating system. FEIG provides a PC/SC kernel driver for some FEIG CPR readers for 7, 8 and 10.

Drivers for Windows CE, Linux and Apple's Mac OS X are not provided.

3.2. FEIG Libraries

3.2.1. Function Libraries for the Transport Layer

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

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

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

Examples based on these libraries are not provided in the context of this tutorial.

3.2.2. Function Library for the Protocol Layer

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

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

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

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

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

[1] Connected with a virtual, serial port based on the Bluetooth stack with the Serial Port Profile (SPP)

3.2.3. Function Library for External Function Units (Multiplexer, Antenna Tuner)

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

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

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

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

3.2.4. Function Library for T=CL based APDU Handling

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

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

.NET and Java applications have no direct access to this layer. Instead of that, other interfaces are provided like specialized TagHandler classes in the respective class library.

3.2.5. Function Library for APDU Handling for ISO 7816 Compliant Smartcards

APDU (Application Protocol Data Unit) centric applications find support for ISO 7816 compliant Smartcards with the library **FESRC** which depends on FEISC. APDUs are transmitted to the reader internal SAM slot. The C interface makes this library compatible with the most important programming languages and development systems.

C++ programmers have the free choice to use this library or the specialized SamHandler class from the C++ class library **FEDM**[1].

.NET and Java applications have no direct access to this layer. Instead of that, the provided SamHandler class in the respective class library can be used1.

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

The class libraries **FEDM** for C++ (**FedmIscCoreApi**), Java (**OBIDISC4J_API**) and .NET (**OBIDISC4NET_API**) represent the highest level in the software stack and support all FEIG reader families. Java and .NET libraries are realized as a small wrapper layer above the C++ library and have almost the same API.

[1] SamHandler class is in development

All three class libraries are split into an implementation layer and an API layer. The first one can also be used as an API layer, but since 2017 it is wrapped by a new and modern API layer above the implementation layer.

All examples and samples in this tutorial are based on the modern API layer.

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

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

High-level methods in the reader class simplify the reader communication (e. g. ReadReaderInfo, ReadReaderDiagnostic, ReadCompleteConfiguration, Inventory, Select, etc.) while the concept of *TagHandler* classes provides an efficient programming model for Transponder communication in the Host-Mode with a collection of proxy classes for a wide range of Transponder standards (ISO 14443, ISO 15693, ISO 18000-3M3, EPC Class 1 Gen 2) as well as specific Chip types. Each TagHandler class offers a specific API for the identified Transponder type.

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

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

3.3. Custom Applications in the Reader

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

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

The figure below (Fig. 2) pictures the principal internal software architecture for the UHF-reader ID ISC.LRU3000, which is identical for all other readers with an ACC onboard.

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

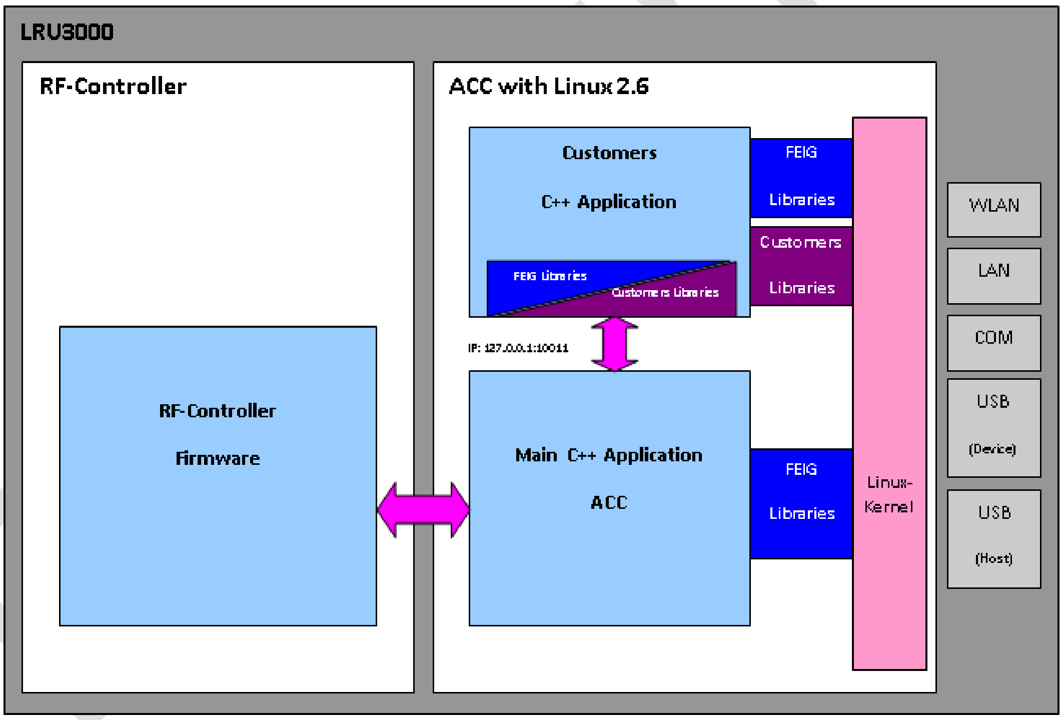


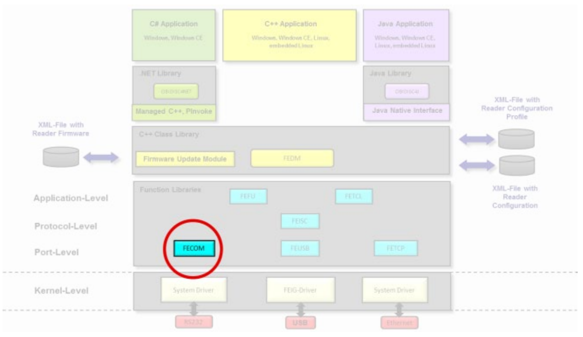
Fig 1: ACC Software Architecture

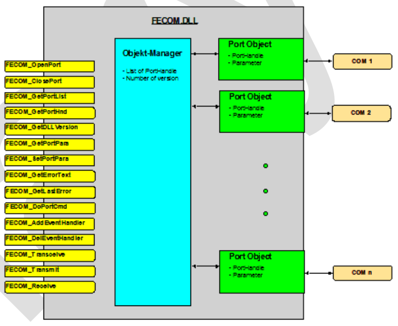
4. Overview of all Libraries

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

4.1. Function Libraries

4.1.1. FECOM





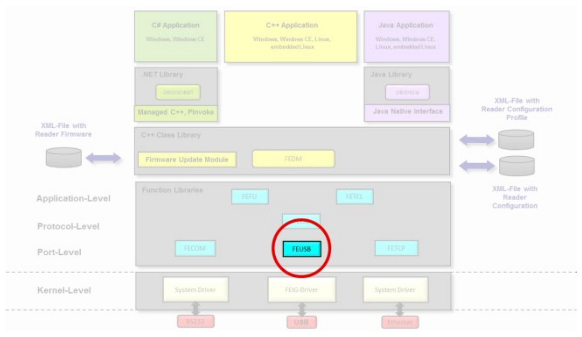
The transport layer library **FECOM** (= **FE**IG **COM**munication) encapsulates all the functions and parameters which the user needs in order to manage one or more serial ports open at the same time. These ports are independent from each other and can be used simultaneously. Event handling mechanisms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

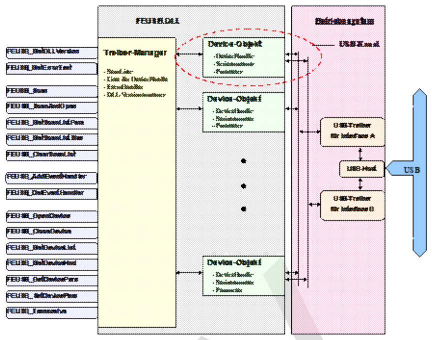
The API is identical for all Operating Systems and contains functions for managing of Serial Ports, communication functions and some more helper functions.

A selection of the important functions:

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

4.1.2. FEUSB





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

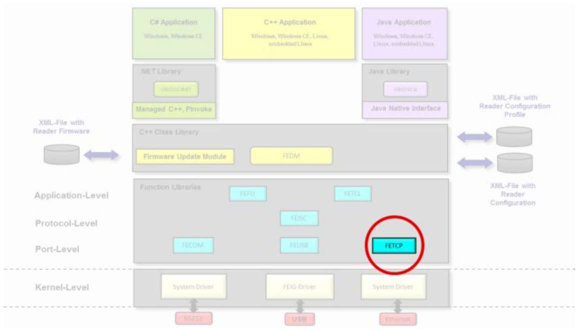
The API is identical for all Operating Systems and contains functions for managing of USB connections, communication functions and some more helper functions.

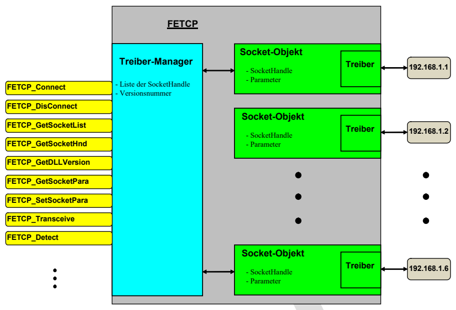
The first step in establishing a connection with an USB-reader is to detect (scan procedure) one or all USB-readers on the USB of the PC. Each found device is entered in the internal scan list but not opened. Before the first communication a USB-reader must be selected from the scan list and the function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

A selection of the important functions:

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

4.1.3. FETCP





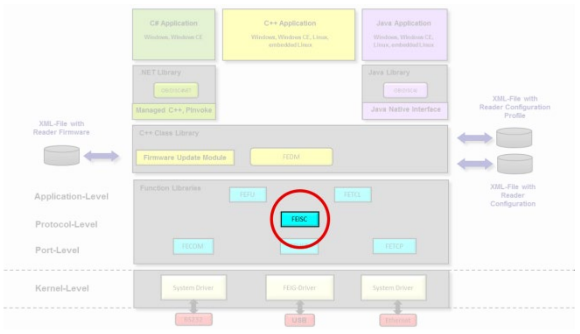
The transport layer library **FETCP** (= **FE**IG **TCP**) manages multiple TCP/IP (IPv4) connections over LAN or WLAN. These connections are independent from each other and can be used simultaneously.

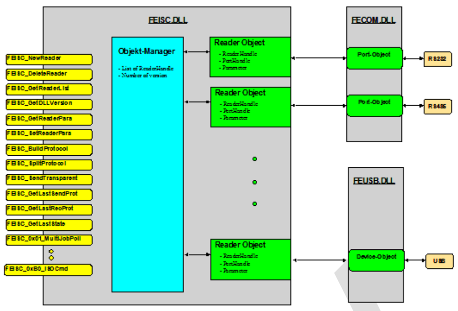
The API is identical for all Operating Systems and contains functions for managing of TCP/IP connections, communication functions and some more helper functions.

A selection of the important functions:

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

4.1.4. FEISC





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

A Plug-in mechanism is provided to support vendor specific port types.

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

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

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

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

A selection of the important functions:

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

int FEISC_0x81_WriteConfBlock(int iReaderHnd, unsigned char cBusAdr, unsigned char cConfAdr, unsigned char* cConfBlock, int iDataFormat)

int FEISC_0x83_ResetConfBlock(int iReaderHnd, unsigned char cBusAdr, unsigned char cConfAdr)

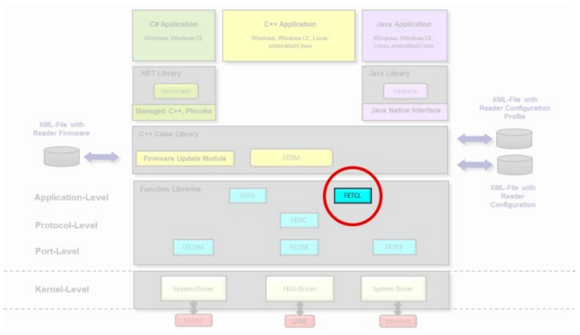
int FEISC_0xA0_RdLogin(int iReaderHnd, unsigned char cBusAdr, unsigned char* cRd_PW, int iDataFormat)

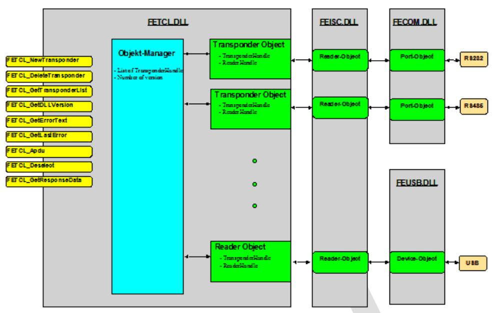
int FEISC_0xAE_ReaderAuthent(int iReaderHnd, unsigned char cBusAdr, unsigned char cMode, unsigned char cKeyType, unsigned char cKeyLen, unsigned char * cKey, int iDataFormat)

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

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

4.1.5. FETCL





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

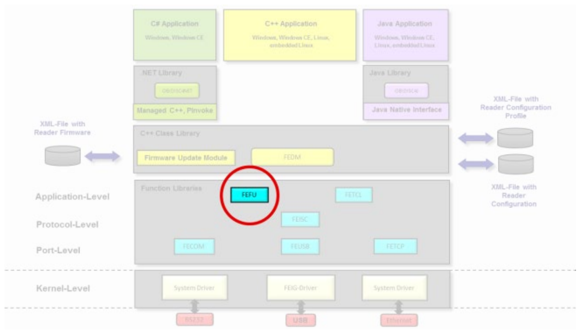
The functions in FETCL are responsible only for internal administration, T=CL protocol cycle handling, like split of uplink data and response data collection and any necessary error outputs. Every other ISO14443 commands, like Inventory, Select or Halt must be executed with FEISC.

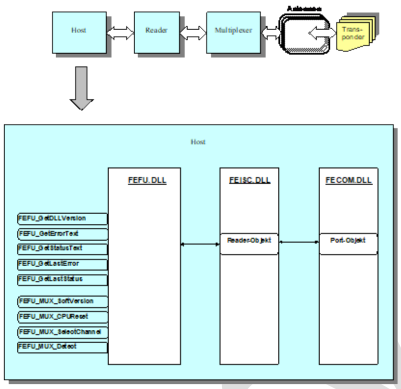
The API is identical for all Operating Systems and contains functions for managing of Transponder objects, communication functions and some more helper functions.

A selection of the important functions:

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

4.1.6. FEFU





The library FEFU (= **FE**IG **F**unction **U**nit) incorporates for the programmer all the necessary functions for easy communication with external Function Units that are accessed by readers. The picture on the right side shows the chain of communication within and outside the host.

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

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

Function list:

int FEFU_GetErrorText(int iErrorCode, char* cErrorText)

int FEFU_GetStatusText(unsigned char ucStatus, char* cStatusText)

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

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

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

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

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

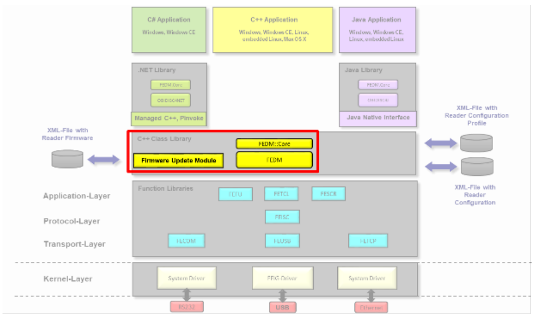
int FEFU_UMUX_Detect_GetPower(int iReaderHnd, unsigned char ucReaderBusAdr, unsigned char ucMuxAdr, unsigned char ucFlags, unsigned char* ucData)

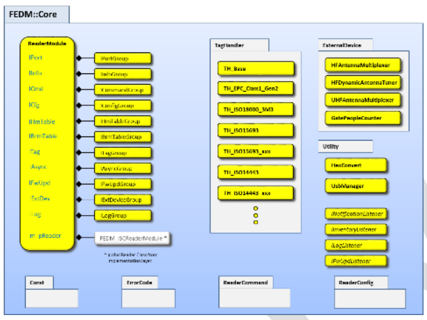
int FEFU_UMUX_SelectChannel(int iReaderHnd, unsigned char ucReaderBusAdr, unsigned char ucMuxAdr, unsigned char ucFlags, unsigned char ucChannelNo)

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

4.2. Class Libraries

4.2.1. FEDM





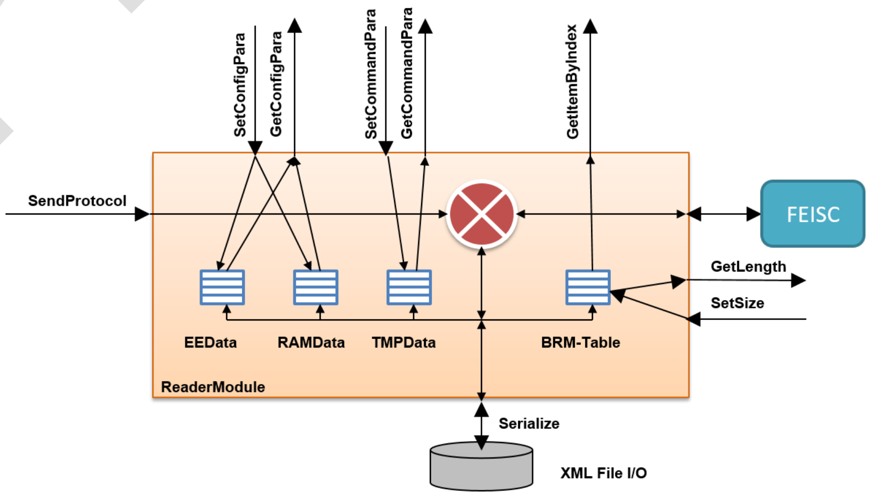
The C++ class library **FEDM** (**FE**IG **D**ata **M**odul) offers a comfortable API for FEIG readers and –Transponders and should be the best choice for C++ programmers.

Primarily in the year 2000, the library was designed for five, completely different RFID product families with different binary communication protocols to unify the high-level handling with FEIG ELECTRONIC GmbH readers. Since then, the library is permanently developed and specialized for the FEIG reader families. A lot of operations can be realized with few code lines by using the high-level reader class and since 2009/2010 by using Transponder classes.

With the beginning of 2017, the previous library is downgraded to an implementation layer and wrapped by a new and modern API layer, called **FedmIscCoreApi**. This new API layer will be extended with new features, like firmware update, and is recommended to be used for new projects.

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

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



The horizontal axis shows the control flow that is generated by the SendProtocol method, the only communication method. It independently retrieves all the necessary data from the integrated data containers before transmitting the send protocol and stores the received protocol data there as well. This means the application must write all the data necessary for this protocol to the corresponding data containers in the correct locations before invoking the SendProtocol. Likewise, the receive data are stored at particular locations in corresponding data containers. The manual for FEDM (Part B.ISC) contains examples for each reader command and the following sections demonstrate also a lot of more complex examples.

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

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

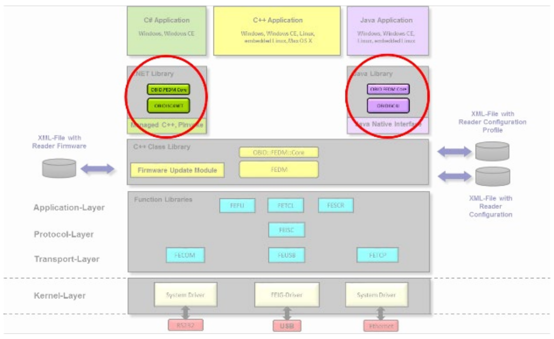
The concept is based on the automatic identification of the type of the transponder after a successful inventory. With ISO 15693 compliant transponders the manufacturer ID and the chipID, which are part of the serial number, are evaluated. With ISO 14443 compliant transponders the type of the TagHandler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

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

Due to the complexity of the reader API and the different reader modes, the API library is organized in namespaces:

Namespace	Content
FEDM::Core	Main namespace Contains the main reader class, all nested group classes of the reader class, helper classes and listener interfaces.
FEDM::Core::Const	Namespace for constants, organized in sub namespaces E.g. for reader types and reader Names
FEDM::Core::Error	Namespace with error codes
FEDM::Core::ReaderCommand	Namespace for constants, organized in sub namespaces with command parameters
FEDM::Core::ReaderConfig	Namespace for constants, organized in sub namespaces with configuration parameters
FEDM::Core::TagHandler	Namespace for TagHandler classes
FEDM::Core::ExternalDevice	Namespace for Function Unit and People Counter classes

4.2.2. OBIDISC4J_API and OBIDISC4NET_API



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

With the beginning of 2017, the previous library is downgraded to an implementation layer and wrapped by a new and modern API layer, called **OBIDISC4J_API** rsp. **OBIDISC4NET_API**. These new API layer will be extended with new features, like firmware update, and are recommended to be used for new projects.

4.3. Thread Security

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

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

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

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

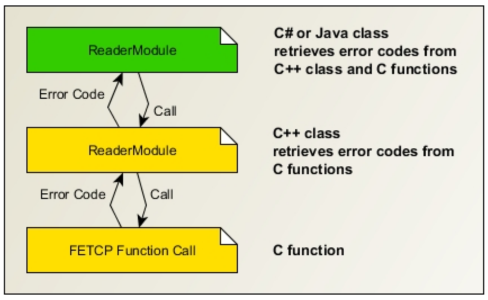
The library FETCL for ISO 14443-4 compliant Transponders is thread-safe, only when each Transponder object is connected with a different reader object and only one APDU is exchanged with each reader at the same time. Even if the function FETCL_Apdu can be called asynchronously, this means not, that multiple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

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

5. Error Handling

One of the most important, but likely unattended theme is the error handling. With each received protocol, FEIG readers signals with the Status byte the result of the last operation (see 2.3. Transmission Protocol). Every Status byte is listed in the system manual of each reader.

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


ReaderModule
C# or Java class retrieves error codes from C++ class and C functions
Error Code
Call
ReaderModule
C++ class retrieves error codes from C functions
Error Code
Call
FETCP Function Call
C function

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

Library	Value range for error codes	Reference
FedmCoreApi	-101 ... -999	HTML API documentation
FECOM	-1000…-1099	H80592-xx-ID-B
FEUSB	-1100…-1199	H00501-xx-ID-B
FETCP	-1200…-1299	H30802-xx-ID-B
FEISC	-4000…-4099	H9391-xx-ID-B
FEFU	-4100…-4199	H30801-xx-ID-B
FETCL	-4200…-4299	H50401-xx-ID-B
FESCR	-4300…-4399	H11010-xx-ID-B

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

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

6. Section 1: Basic Initializations

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

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

1. Bus address	The bus address for the reader is preset in the class for 255. To set a different address, use the SetBusAddress method. Note: The bus address is only relevant for the communication over the serial port.
2. Table size	The integrated tables for Buffered Read Mode (BRM) and Host Mode are not initialized. Before the initial communication, you must set the table size using the method IHmTableGroup::SetSize resp. iBrmTableGroup::SetSize. The size is selected equal to the maximum number of transponders located in the antenna field at the same time. Please consider, that a reader can process only one working mode. Thus, only the size of the table being used needs to be set.
3. Reader type	The reader type must be set in the reader class with one of two options: 1. The call of one of the IPortGroup::ConnectXXX methods, which calls internally ReadReaderInfo after a successful connection (recommended). 2. Set of reader type with the method SetReaderType. The constants of all reader types are listed in the namespace FEDM::Core::Const::ReaderType.

7. Section 2: Establish a Connection to the Reader

This section introduces the operations for establishing a connection to the FEIG reader.

7.1. Serial Port (RS232 / RS485 / RS422)

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

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

7.2. Bluetooth

For Windows and Windows CE, FEIG readers with Bluetooth interface are connected at a virtual serial port (e.g. COM15) and with Linux as a /dev/rfcomm device.

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

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

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

7.3. USB

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

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

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

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

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

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

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

7.4. TCP/IP (LAN and WLAN)

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

In the following example a connection to a reader with the IP-Address 192.168.1.100 and Port 10001 is established. 7.4.1. Excursion: Secured Data Transmission with Encryption

Some FEIG readers can secure the data transmission over Ethernet (TCP/IP) with a 256 bit AES algorithm. The Authentication Key (Password) is stored in the reader and cannot read back. The crypto mode is disabled by default.

The encrypted data transmission is realized with functions of the Open-Source organization openSSL (http://www.openssl.org), which are part of the library file libeay32.dll (Windows) 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, transporting the enciphered password (password contains only nulls by default), to open a new session. Every successive protocol will then enciphered automatically.

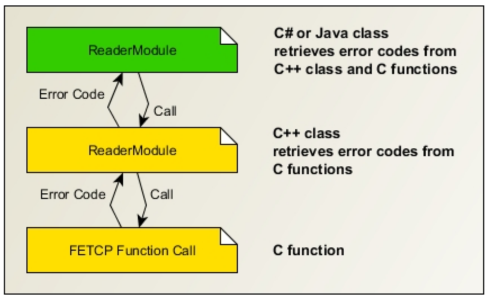
7.5. Excursion: Error Handling for TCP/IP Communication

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

In the following, we discuss error handling for:

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

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


ReaderModule
C# or Java class retrieves error codes from C++ class and C functions
Error Code
Call
ReaderModule
C++ class retrieves error codes from C functions
Error Code
Call
FETCP Function Call
C function

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

7.5.1. Communication Errors

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

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

The preset timeout is 3000ms and normally large enough for the most communication tasks. In rare cases it must be enlarged with the method `IPortGroup::SetPortPara` or `FETCP_SetPortPara`.

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

7.5.2. Errors while Establishing a Connection

When an error occurs with the method `IPortGroup::ConnectTCP` (C++, Java, .NET) or `FETCP_Connect` (C/C++, VB6, …), the error code must be analyzed in detail while error handling.

Function / Method	Error code	Error handling
Function in transport layer **FETCP_Connect**	-1211	Timeout for establishing a connection to the TCP/IP server. Cause may be that another client is blocking the connection. This is a normal runtime problem. The repetitive call can be applied until the Server (FEIG reader) can be connected. Other reasons: the FEIG reader is not powered on or not switched into the subnet or not configured properly concerning the TCP parameters. This must be analyzed by the installation team
	-1212	The parameter cHostAdr in the function is structurally defective. This is a critical error and must be analyzed by the development team.
	-1251	Pass parameter too large or too small, here: the transferred port number is out of range. This is a critical error and must be analyzed by the development team.
Method in C++ class library **IPortGroup::ConnectTCP** Method in Java/.NET class library **IPortGroup.ConnectTCP**	-106	Unknown transfer parameter, here: the transferred port number is out of range. This is a critical error and must be analyzed by the development team.
	-137	Reader object is already connected with a communication port. This is a runtime problem with multiple reasons: a) Another application is connected with this FEIG reader. This is a normal runtime problem. The repetitive call can be applied until the Server (FEIG reader) can be connected. b) Multiple call of ConnectTCP with the same reader objects from the same application. This is a critical error in the application structure and must be analyzed by the development team.
	-157	Reader object is already connected with a communication port. The reason is a multiple call of ConnectTCP with different reader objects from the same application. This is a critical error in the application structure which is not supported by the class libraries and must be analyzed by the development team.
	-1230 -1232 -1237	Communication error while reading the reader-Info and the connection is closed internally. This is an abnormal error and must be analyzed by the

Function / Method	Error code	Error handling
		development or installation team.

7.5.3. Errors while Closing the Connection

The closing of a connection is realized internally with a call of closesocket (Windows) or close (Linux) and returns while the process of closing is not finished. Thus, although the disconnection from the application-side is finished, the final TCP status TIME_WAIT is probably not yet reached. To indicate this situation, the last TCP status is reflected to inform the application. With successive calls of the method `GetTcpConnectionState()` (C++, Java, .NET) or the function `FETCP_GetSocketState(ip, port)` (C/C++, VB6, …) the closing process can be observed.

Only two important errors can occur with the method `DisConnect` (C++, Java, .NET) or `FETCP_DisConnect` (C/C++, VB6, …):

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

7.5.4. Problem with Broken Communication Link – the Keep-Alive Option

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

The solution for this very realistic scenario is the activating of the Keep-Alive option on the server-side. Every FEIG reader with Ethernet interface has parameters for Keep-Alive and it is recommended to enable this option.

The host application has to consider the Keep-Alive time span after closing the broken connection before the reopen is tried and, if successful, the communication is continued.

7.6. Excursion: Detecting Readers with Different Protocol Frames in One App

In constellations when FEIG readers supporting only Standard Protocol Frame or only Advanced Protocol Frame must be detected together in one application, the detection algorithm must be prepared with more logic to prevent long-term timeouts.

The following table illustrates the protocol frame support situation for the FEIG reader families:

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

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

High-level APIs like C++, Java or .NET class libraries have integrated a flexible algorithm with the methods `FindBaudRate()`[1] and `ReadReaderInfo()`, which are called by one of the methods `ConnectXXX` internally. Programmers using these Methods have nothing to do.

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

[1] only for serial or Bluetooth connection

7.6.1. Detecting at Serial Port

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

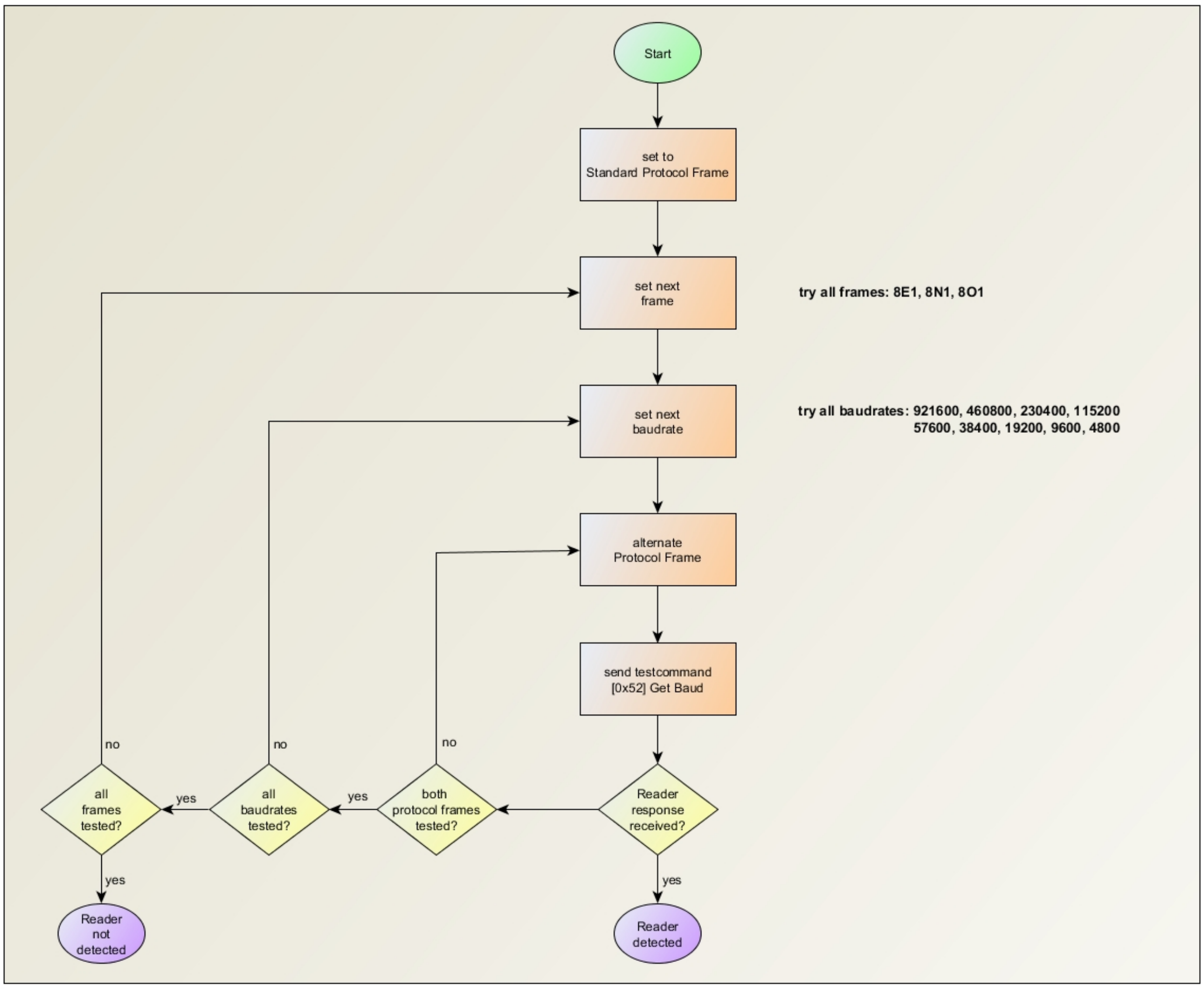

Start
set to Standard Protocol Frame
set next frame
try all frames: 8E1, 8N1, 8O1
set next baudrate
try all baudrates: 921600, 460800, 230400, 115200 57600, 38400, 19200, 9600, 4800
alternate Protocol Frame
send testcommand [0x52] Get Baud
Reader response received?
yes
Reader detected
both protocol frames tested?
no
yes
all baudrates tested?
no
yes
all frames tested?
no
yes
Reader not detected

Figure 2: Detection algorithm

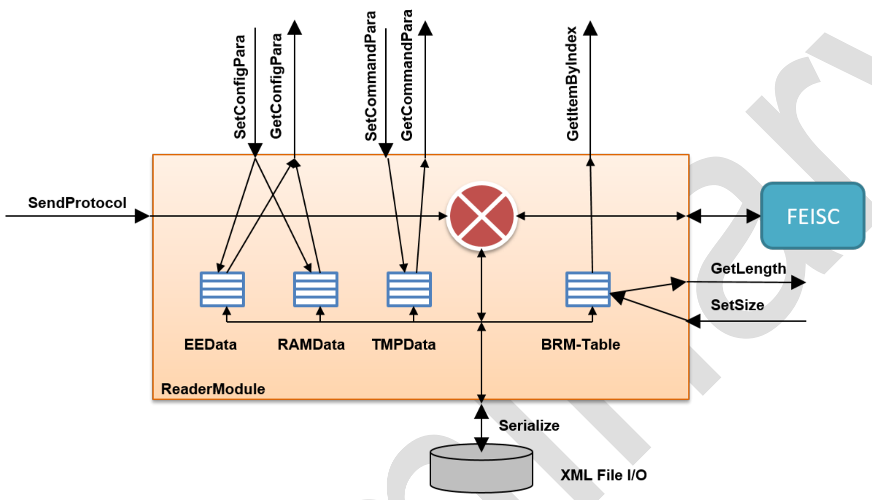
7.6.2. Detecting at USB

Detecting a reader at USB needs no protocol transmissions and it makes the setting of the protocol frame much easier. When a reader is detected by the kernel driver, the reader's name is requested and evaluated. If the reader name is found in a positive list (CPR.04, ISC.MR100, ISC.PRH100), the protocol frame is set to Standard, otherwise to Advanced.

8. Section 3: Basic Knowledge About How to Use SendProtocol()

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

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



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

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

GetCommandPara SetCommandPara	transfer methods for all common communication protocols. The key to the protocol data are parameter identifier and can be found in the namespace (C++), interface (Java) or structure (.NET) ReaderCommand. An example for a parameter identifier in the request command [0x66] Reader Info: SetCommandPara(ReaderCommand::_0x66::Req::MODE, (unsigned char)0x01);
GetConfigPara SetConfigPara	transfer methods for modifying the reader's configuration in the reader object. Each configuration parameter is identified by an identifier string from the namespace (C++), interface (Java) or structure (.NET) ReaderConfig. An example for an identifier is: ReaderConfig::DigitalIO::Relay::No1::IdleMode

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

9. Section 4: Read of Important Information from Reader

This section presents same valuable methods to simplify application programming.

9.1. Reader Information: The Method ReadReaderInfo()

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

The new and modern API layer, called **FedmIscCoreApi** has ConnectXXX methods, which call `ReadReaderInfo()` internally after a successful connection.

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

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

Nice to know, that the class ReaderInfo contains a method `GetReport()` to return a report string like it is used in ISOStart/CPRStart.

9.2. Reader Diagnostic: The Method ReadReaderDiagnostic()

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

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

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

9.3. Reader Configuration: The Method ReadCompleteConfiguration()

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

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

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

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

10. Section 5: Programming for the Host-Mode

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

Note: Programmers have two alternatives in the reader class ReaderModule for the communication with Transponders:

1. Table oriented API (shortly explained in this section)
2. TagHandler API, based on specialized Transponder classes (see 12. Section 6: Using TagHandler classes with Host-Mode)

It is recommended to use the 2nd API for new projects.

10.1. Inventory

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

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

Bit Number	Flag	Notes
7	MORE	query of more data from last Inventory cycle
6	NTFC	Notification (only FEIG CPR reader)
5	PRESC	no Inventory, only presence check (only FEIG CPR reader)
4	ANT	request of additional antenna information
3	-	
2	-	
1	-	
0	-	

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

A table item has data members for all ISO 15693, ISO 18000-3M3, ISO 14443 and EPC Class1 Gen2 compliant Transponders. Thus, not every table element is applicable with every Transponder type.

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

Element	Reader Support	Notes
IDD	all	Serial Number or UID or IDD with variable length of up to 96 bytes
IDD-Length	all	length of SNR/UID/IDD in number of bytes
TrType	all	Transponder type according the system manuals of the readers

TrType	Transponder Type
0x00	NXP I-Code1
0x01	National Instruments Tag-it
0x03	Transponder according ISO15693
0x04	Transponder according ISO14443A
0x05	Transponder according ISO14443B
0x06	NXP I-Code EPC
0x07	NXP I-Code UID
0x08	Jewel
0x09	ISO 18000-3M3
0x0A	STMicroelectronics SR176
0x0B	STMicroelectronics SRIxx (SRI512, SRIX512, SRI4K, SRIX4K)
0x0C	Microchip MCRFxxx
0x10	Innovatron (ISO 14443B')
0x11	ASK CTx
0x80	ISO18000-6 A
0x81	ISO18000-6 B (UCODE; UCODE EPC 1.19)
0x83	EM4222, EM4444
0x84	EPC class 1 Gen 2

Rx Data	ISO 15693 / ISO 14443	Receive data block with variable block size
Tx Data	ISO 15693 / ISO 14443	Transmit data block with variable block size
Rx EPC-Bank	EPC Class1 Gen2	Receive data block for EPC memory with block size 2
Tx EPC-Bank	EPC Class1 Gen2	Transmit data block for EPC memory with block size 2
Rx TID-Bank	EPC Class1 Gen2	Receive data block for TID memory with block size 2
Tx TID-Bank	EPC Class1 Gen2	Transmit data block for TID memory with block size 2
Rx Res-Bank	EPC Class1 Gen2	Receive data block for Reserved memory with block size 2
Tx Res-Bank	EPC Class1 Gen2	Transmit data block for Reserved memory with block size 2
AFI	ISO 15693	Application Family Identifier
DSFID	ISO 15693	Data Storage Family Identifier
IDDT	EPC Class1 Gen2	Identifier Data Type
TrInfo	ISO 14443 A	Transponder Info
OptInfo	ISO 14443 A	Optional Information
ProtoInfo	ISO 14443 B	Protocol Information
the following elements are antenna specific (ANT flag must be set in Mode)		
Flags	ISO 15693/EPC Class1 Gen2	Record information
RSSI-Map	ISO 15693/EPC Class1 Gen2	Map with RSSI and status value of each Antenna

10.2. Read / Write Transponder Data

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

Some readers have additionally the extended addressed mode implemented with which the block address is two bytes wide and the IDD can have a variable length. For UHF readers, the extended address mode supports also different memory banks and an access password.

10.2.1. Normal Addressed Mode

Examples can be found in 12.2. Inventory and Select of Section 6: Using TagHandler classes.

10.2.2. Extended Addressed Mode

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

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

```
SetSize( unsigned int uiSize,
         unsigned int uiRxDB_BlockCount, unsigned int uiRxDB_BlockSize,
         unsigned int uiTxDB_BlockCount, unsigned int uiTxDB_BlockSize).
```

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

10.3. Excursion: Inventory with Multiple Antennas

The standard Inventory command operates with one antenna. This covers the capability of the most FEIG readers. For readers with multiple antenna connectors and an internal multiplexer, the Inventory command is extended to operate with specified antennas. This option can be switched on with a flag in the mode byte.
The antenna number, where the Transponder is detected, is reflected in the returned data records.

11. Section 6: Using TagHandler Classes with Host-Mode

Programmers have two alternatives in the reader class ReaderModule for the communication with Transponders:

3. Table oriented API (s. 11. Section 5: Programming for the Host-Mode)
4. TagHandler API, based on specialized Transponder classes

It is recommended to use the 2nd API for new projects. TagHandler classes are specific to Transponder standards like ISO 14443, ISO 15693, ISO 18000-3M3 and EPC Class 1 Gen 2 or customized for manufacturer specific extended API. Each standard and chip type is implemented as a class and all classes together build a hierarchical system of derived classes. Base class is TagHandler::TH_Base.

Precondition for the use of TagHandler classes is 1st the use of the methods ITagGroup::Inventory and ITagGroup::Select from the reader class and 2nd the identifiability of the Transponder standard and/or chip type for the accurate creation of TagHandler classes. Unsupported chip types are assigned automatically to the base class TagHandler::TH_Base.

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

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

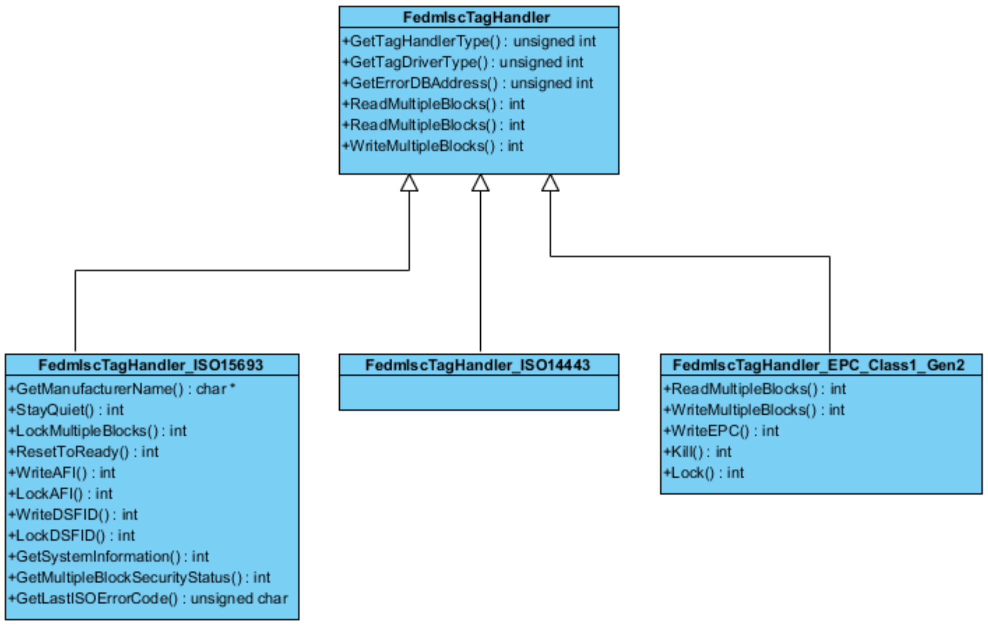
11.1. Benefit

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

11.2. TagHandler Classes

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

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


FedmIscTagHandler
+GetTagHandlerType() : unsigned int
+GetTagDriverType() : unsigned int
+GetErrorDBAddress() : unsigned int
+ReadMultipleBlocks() : int
+ReadMultipleBlocks() : int
+WriteMultipleBlocks() : int
FedmIscTagHandler_ISO15693
+GetManufacturerName() : char *
+StayQuiet() : int
+LockMultipleBlocks() : int
+ResetToReady() : int
+WriteAFI() : int
+LockAFI() : int
+WriteDSFID() : int
+LockDSFID() : int
+GetSystemInformation() : int
+GetMultipleBlockSecurityStatus() : int
+GetLastISOErrorCode() : unsigned char
FedmIscTagHandler_ISO14443
FedmIscTagHandler_EPC_Class1_Gen2
+ReadMultipleBlocks() : int
+WriteMultipleBlocks() : int
+WriteEPC() : int
+Kill() : int
+Lock() : int

11.2.1. Life Cycle of TagHandler Objects

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

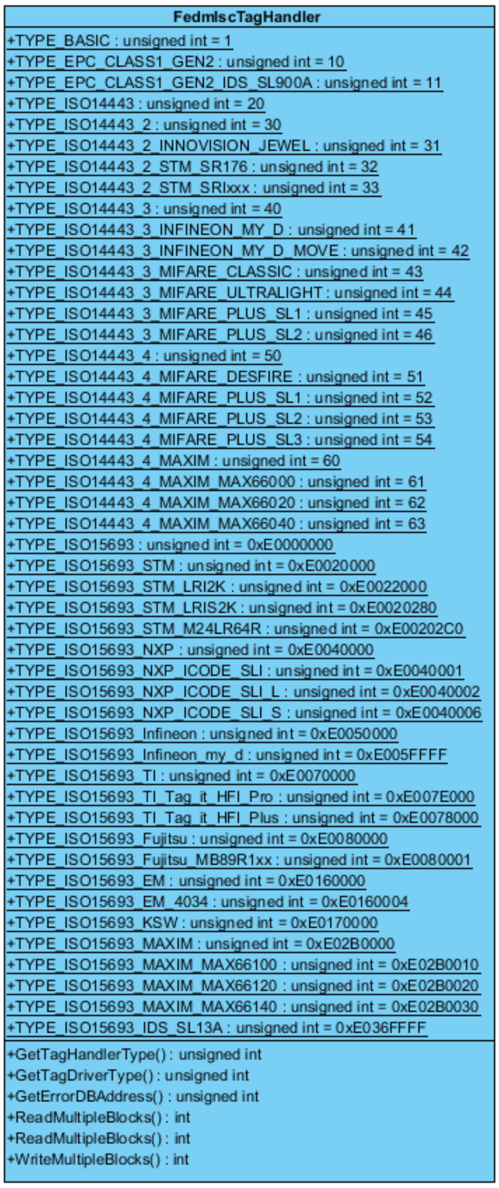
11.2.2. Naming Conventions

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

11.2.3. Base Class TH_Base

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

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



```
FedmIscTagHandler
+TYPE_BASIC : unsigned int = 1
+TYPE_EPC_CLASS1_GEN2 : unsigned int = 10
+TYPE_EPC_CLASS1_GEN2_IDS_SL900A : unsigned int = 11
+TYPE_ISO14443 : unsigned int = 20
+TYPE_ISO14443_2 : unsigned int = 30
+TYPE_ISO14443_2_INNOVISION_JEWEL : unsigned int = 31
+TYPE_ISO14443_2_STM_SR176 : unsigned int = 32
+TYPE_ISO14443_2_STM_SRIxxx : unsigned int = 33
+TYPE_ISO14443_3 : unsigned int = 40
+TYPE_ISO14443_3_INFINEON_MY_D : unsigned int = 41
+TYPE_ISO14443_3_INFINEON_MY_D_MOVE : unsigned int = 42
+TYPE_ISO14443_3_MIFARE_CLASSIC : unsigned int = 43
+TYPE_ISO14443_3_MIFARE_ULTRALIGHT : unsigned int = 44
+TYPE_ISO14443_3_MIFARE_PLUS_SL1 : unsigned int = 45
+TYPE_ISO14443_3_MIFARE_PLUS_SL2 : unsigned int = 46
+TYPE_ISO14443_4 : unsigned int = 50
+TYPE_ISO14443_4_MIFARE_DESFIRE : unsigned int = 51
+TYPE_ISO14443_4_MIFARE_PLUS_SL1 : unsigned int = 52
+TYPE_ISO14443_4_MIFARE_PLUS_SL2 : unsigned int = 53
+TYPE_ISO14443_4_MIFARE_PLUS_SL3 : unsigned int = 54
+TYPE_ISO14443_4_MAXIM : unsigned int = 60
+TYPE_ISO14443_4_MAXIM_MAX66000 : unsigned int = 61
+TYPE_ISO14443_4_MAXIM_MAX66020 : unsigned int = 62
+TYPE_ISO14443_4_MAXIM_MAX66040 : unsigned int = 63
+TYPE_ISO15693 : unsigned int = 0xE0000000
+TYPE_ISO15693_STM : unsigned int = 0xE0020000
+TYPE_ISO15693_STM_LRI2K : unsigned int = 0xE0022000
+TYPE_ISO15693_STM_LRIS2K : unsigned int = 0xE0020280
+TYPE_ISO15693_STM_M24LR64R : unsigned int = 0xE00202C0
+TYPE_ISO15693_NXP : unsigned int = 0xE0040000
+TYPE_ISO15693_NXP_ICODE_SLI : unsigned int = 0xE0040001
+TYPE_ISO15693_NXP_ICODE_SLI_L : unsigned int = 0xE0040002
+TYPE_ISO15693_NXP_ICODE_SLI_S : unsigned int = 0xE0040006
+TYPE_ISO15693_Infineon : unsigned int = 0xE0050000
+TYPE_ISO15693_Infineon_my_d : unsigned int = 0xE005FFFF
+TYPE_ISO15693_TI : unsigned int = 0xE0070000
+TYPE_ISO15693_TI_Tag_it_HFI_Pro : unsigned int = 0xE007E000
+TYPE_ISO15693_TI_Tag_it_HFI_Plus : unsigned int = 0xE0078000
+TYPE_ISO15693_Fujitsu : unsigned int = 0xE0080000
+TYPE_ISO15693_Fujitsu_MB89R1xx : unsigned int = 0xE0080001
+TYPE_ISO15693_EM : unsigned int = 0xE0160000
+TYPE_ISO15693_EM_4034 : unsigned int = 0xE0160004
+TYPE_ISO15693_KSW : unsigned int = 0xE0170000
+TYPE_ISO15693_MAXIM : unsigned int = 0xE02B0000
+TYPE_ISO15693_MAXIM_MAX66100 : unsigned int = 0xE02B0010
+TYPE_ISO15693_MAXIM_MAX66120 : unsigned int = 0xE02B0020
+TYPE_ISO15693_MAXIM_MAX66140 : unsigned int = 0xE02B0030
+TYPE_ISO15693_IDS_SL13A : unsigned int = 0xE036FFFF
+GetTagHandlerType() : unsigned int
+GetTagDriverType() : unsigned int
+GetErrorDBAddress() : unsigned int
+ReadMultipleBlocks() : int
+ReadMultipleBlocks() : int
+WriteMultipleBlocks() : int
```

11.2.4. Excursion: Class TH_ISO15693

The ISO 15693 standard is represented by the TagHandler class TH_ISO15693. Many Transponder manufacturers have extended the ISO 15693 standard with its own API and are supported with derived classes.

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

MSB LSB

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

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

The following manufacturers and Chip-Types are supported:

Manufacturer	Chip-Types
STMicroelectronics SA	LRI2K LRIS2K LRIS64K M24LR64R M24LRxxER ST25DVxxK
NXP Semiconductors	I-Code SLI I-Code SLI-L I-Code SLI-S I-Code SLIX I-Code SLIX-L I-Code SLIX-S I-Code SLIX2 I-CODE DNA
Infineon Technologies AG	my-d Light
Texas Instruments	Tag-it HF Tag-it HF-I Pro Tag-it HF-I Plus
Fujitsu Limited	MB89R116 MB89R118 MB89R119
EM Microelectronic-Marin SA	EM4034
KSW Microtec GmbH	TempSens
Maxim	MAX66100 MAX66120 MAX66140
IDS Microchip AG	SL13A

11.2.5. Excursion: Class TH_EPC_Class1_Gen2

The following manufacturers and Chip-Types are supported:

Manufacturer	Chip-Types
IDS	SL900A

11.2.6. Excursion: Classes TH_ISO14443

The following manufacturers and Chip-Types are supported:

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

12. Section 8: Programming for the Buffered-Read-Mode

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

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

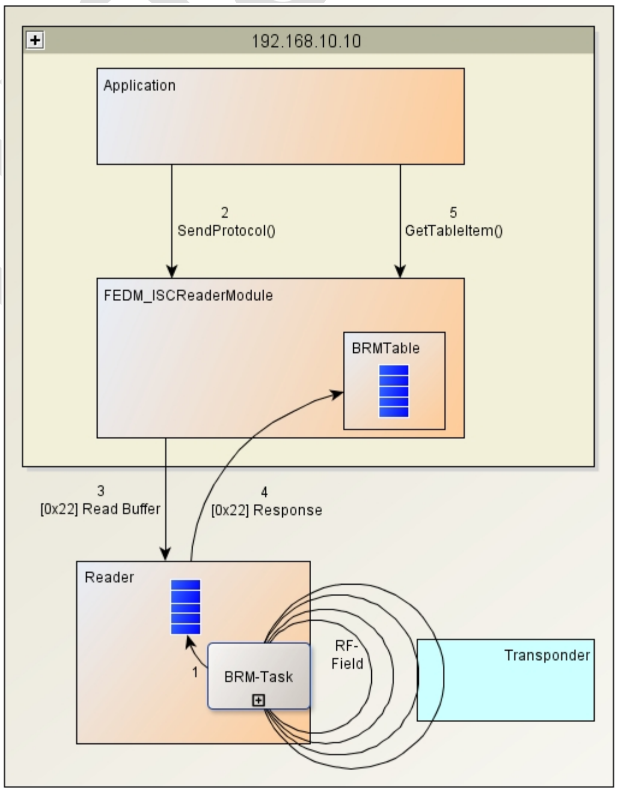
With settings in the parameter group `OperatingMode.BufferedReadMode.DataSelector`, the reader can be configured to read specific data like UID, data blocks or Date/Time.

12.1. Method of Operation

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

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

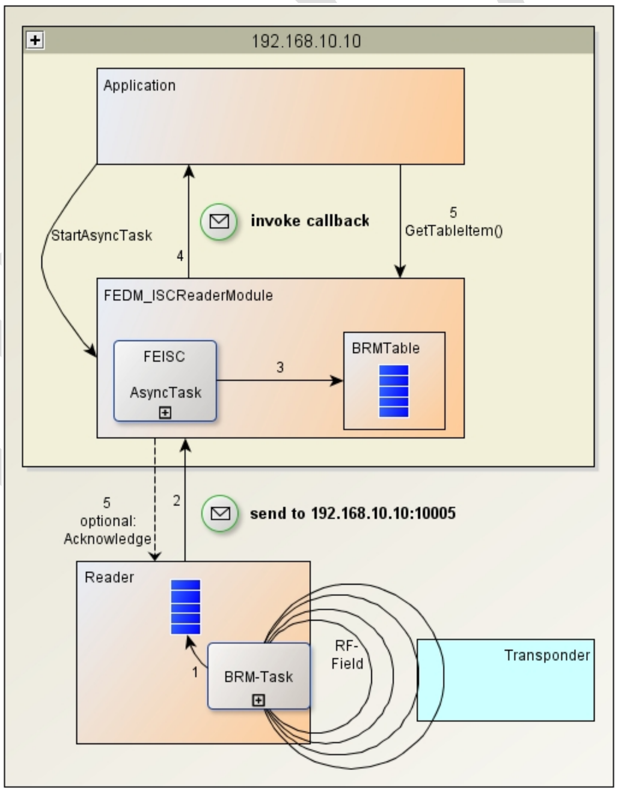
The final step from application-side is to query and process the new table data (5).



13. Section 9: Programming for the Notification-Mode

The Notification-Mode is an extension of the Buffered-Read-Mode and is supported by many FEIG readers. While the Buffered-Read-Mode needs polling from the host-side application for receiving the automatic scanned Transponder data, the Notification-Mode transmits these data automatically to a configurable destination (see parameter group: `OperatingMode.NotificationMode.Transmission.Destination.IPv4`).
When the Notification-Mode is enabled in the reader (parameter: `OperatingMode.Mode`), the Buffered-Read-Mode Task in the reader is started and scans continuously for Transponders.

13.1. Method of Operation



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

The initial step has to be the call of the method `IAsyncGroup::StartNotificationTask()` to start an asynchronous task inside the library FEISC (protocol layer) and to install a notification function[1] as an event handler. The asynchronous task is realized as a Thread.

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

Receiver is the module ReaderModule. It stores the new data elements in the internal BrmTable (3)[2] and calls the installed event handler (4) to inform the application. The final step from application-side is to query and process the new table data (5).

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

[1] In C++: callback function; in Java: listener method; in C#: delegate
[2] Step (3) overwrites older data records.
[3] When the secured data transmission is configured, handshake is enabled by default and cannot be disabled.

13.2. Cancel Asynchronous Task

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

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

13.3. Adjust the Method of Operation

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

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

1. A handshake mode can be enabled to let the reader wait for an acknowledge from the receiver with the parameter (`OperatingMode.NotificationMode.Transmission.Enable_Acknowledge`). When secured data transmission is configured, the handshake is enabled by default and cannot be disabled.
2. A limitation of the number of data records to be sent with one transmission can be set with `OperatingMode.NotificationMode.Transmission.DataSetsLimit`. Limitation is recommended for slow or bad conditioned links like WLAN or GPRS. In some constellations, the limit can be set to 1 data record to serialize the notifications for the following information processing.
3. A connection can be kept open after a notification transmission. This hold time can be adjusted with `OperatingMode.NotificationMode.Transmission.Destination.ConnectionHoldTime`. With every connect to a server, a new transmit port is selected in the reader's TCP stack and is blocked for up to 2 minutes before it gets into the closed state. In high frequented notification situations, this can cause a problem in the reader, because the number of ports is limited and when all ports are used, a notification cannot be sent until the first port reaches the closed state.

13.3.1. Excursion: Writing Data to the Transponder

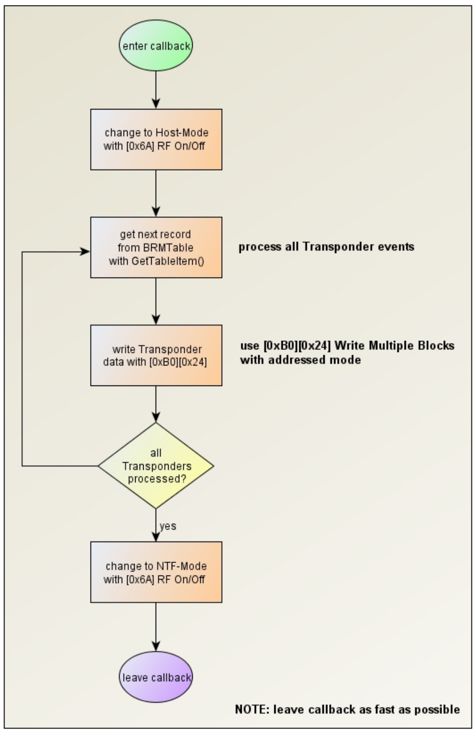
The Buffered-Read-Mode (BRM) and the extension, the Notification-Mode (NTFM), is an interruptible process. The switching from NTFM to Host-Mode (HM) and back to NTFM is controlled by the reader protocol [0x6A] RF On/Off.

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

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

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

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


enter callback
change to Host-Mode with [0x6A] RF On/Off
get next record from BRMTable with GetTableItem()
process all Transponder events
write Transponder data with [0xB0][0x24]
use [0xB0][0x24] Write Multiple Blocks with addressed mode
all Transponders processed?
yes
change to NTF-Mode with [0x6A] RF On/Off
leave callback
NOTE: leave callback as fast as possible

13.4. Considerations for Fail-Safe Operation

13.4.1. Keep-Alive Option for Detecting Broken Network Link

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

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

13.4.2. Avoiding Deadlock Situations

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

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

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

14. Section 11: Management of the Reader Configuration

14.1. Persistence of the Reader Configuration

Each FEIG reader is controlled by parameters which are stored grouped in blocks in an EEPROM and are described in detail in the system manual for the respective reader. After switching on or resetting the reader, all parameters are loaded into RAM, evaluated and incorporated in the controller.
All parameters can be modified using a protocol so that the behavior of the reader can be adapted to the application. Ideally, the program ISOStart/CPRStart is used for this adaptation and normally no parameters have to be changed in the application. Despite this, it can happen that one or more parameters from a program have to be changed. This chapter should familiarize you with the procedure using the reader class as an example.
A common characteristic of all readers is the grouping in blocks of thematically related parameters to 14 or 30 bytes per configuration block. Each parameter cannot be addressed individually but must always be retrieved together with a configuration block using the protocol [0x80] or [0x8A] Read Configuration, then modified and finally written back to the reader with the protocol [0x81] or [0x8B] Write Configuration. This cycle must always be complied with and is also checked by the reader class. This means that writing a configuration block without previously reading the same block is not possible.
The reader class manages the configuration data in a byte array for data from the EEPROM and a second byte array for data from the RAM of the reader. The differentiation is important as changes in RAM are used immediately while changes in the EEPROM of the reader do not become active until after a reset. Therefore the reader class has its own byte arrays for both configuration sets.