

# UHF Reader

## RFLine

### Reader Library

---

## Contents

1	Introduction .....	3
1.1	Library structure .....	4
1.1.1	Level 1 – <i>Device Manager</i> (Reader Logic) .....	4
1.1.2	Level 2 – <i>Command Manager</i> (Default protocol) .....	4
1.1.3	Level 3 – <i>Sender</i> (physical transport) .....	4
1.2	Using the Library .....	5
1.2.1	Scope of use of the library .....	6
2	IUhfReaderManager <i>Interface Commands</i> .....	7
2.1	Events.....	7
2.1.1	SpontaneousMessage.....	7
2.1.2	TagReceived .....	7
2.2	Methods .....	8
2.2.1	GetFirmwareVersion .....	9
2.2.2	FirmwareUpdate.....	9
2.2.3	DeviceReset .....	9
2.2.4	SetDeviceActivationState .....	10
2.2.5	ReadReflectedPower .....	10
2.2.6	WriteROMParameters .....	11
2.2.7	ReadROMParameters.....	13
2.2.8	ResetROMParametersToDefault.....	13
2.2.9	GetInventory .....	14
2.2.10	ReadBufferData.....	15
2.2.11	ReadBufferDataCount.....	15
2.2.12	ResetBuffer .....	15
2.2.13	ProgramTagEpc.....	16
2.2.14	ReadTagData .....	17
2.2.15	WriteTagData .....	18
2.2.16	Kill Tag .....	19
2.2.17	Lock Tag.....	20
2.3	IDeviceManager Interface .....	21
2.4	Disclaimer and Limitations of Use .....	22
2.5	License to use .....	22

---

# 1 Introduction

The *RFLine Reader Library* is a .NET DLL designed to simplify the integration of the RFLine UHF reader into software projects.

It provides a high-level, fully device-oriented interface, hiding the complex details of low-level communication — for example, over a CAN bus (J1939 protocol) — and the implementation of the communication protocol itself.

Developers using this library can interact with the "abc" reader through semantic methods, such as:

- `GetFirmwareVersion`
- `GetInventory`
- `ProgramTagEpc`
- ...

without needing to know:

- The binary format of commands
- the low-level protocol, e.g. for CAN, TP.CM/TP.DT PDU management for long messages or data fragmentation
- the specifications of the *intermediate level* protocol.

The SDK is designed to:

- Simplify software development related to the "ABC" reader
- abstract communication details (e.g., CAN, parsing, events)
- provide a typical, robust and documented interface
- Facilitate integration into *WPF applications, Windows Forms, or .NET backend services*

---

## 1.1 Library structure

The library is structured according to an independent layer architecture, which allows you to completely decouple:

- high-level application logic (reader functions)
- the communication protocol with the device
- physical data transport (CAN, RS232, TCP)

This approach offers modularity, testability, and scalability, allowing the library to automatically adapt to the selected communication mode.

### 1.1.1 Level 1 – *Device Manager* (Reader Logic)

**Public interface:** *IDefaultUhfReaderManager*, *IDeviceManager*

**Class:** *DefaultUhfReaderManager*

This is the level exposed to the library user. It contains device-oriented semantic methods, for example, `GetFirmwareVersion`.

Each method completely hides the implementation details: it only deals with providing a clear, typed and consistent interface.

### 1.1.2 Level 2 – *Command Manager* (Default protocol)

**Internal Class:** *DefaultCommandManager*

This layer deals with the formatting, serialization and *parsing* of RFLine protocol-specific commands. Receives a high-level command (e.g. `GetFirmwareVersion`) and converts it into a binary sequence that conforms to the device's protocol. It also manages the receipt and validation of responses.

In addition:

- automatically subscribes to spontaneous events at the lower level
- forwards these events to the *DeviceManager* via its own internal event

### 1.1.3 Level 3 – *Sender* (physical transport)

**Internal class:** *SenderCanJ1939* (or *SenderTcp*, *SenderRs232*, etc.)

This level deals with physical communication with the reader. Transparently manages:

- opening and closing the connection (e.g. COM port, TCP socket, CAN)
- Sending and receiving raw packets
- fragmentation and recomposition of long messages (e.g. TP.CM/TP.DT on CAN)
- Handling spontaneous asynchronous messages

Each *sender* class implements a common *ISender* interface, which allows the upperlayer to remain independent of the transport medium.

---

## 1.2 Using the Library

The user of the library does not have to worry about the underlying protocol.

At initialization time, simply create an *ISenderConfig* specific to the type of connection you want and pass it to the reader manager. All communication will be automatically adapted to the selected channel.

The library supports several physical communication channels (CAN J1939, TCP/IP, RS232). To flexibly handle the different configurations, each transport mode has its own configuration class:

Modality	Sender class	Parameter class
CANJ1939	SenderCanJ1939	SenderConfigCanJ1939
TCP/IP	SenderTcp	SenderConfigTcp
Serial	SenderRs232	SenderConfigRs232

Ex. of use:

```
ISenderConfig senderConfig = new SenderConfigCanJ1939(  
    14, // masterId  
    235, // deviceId  
    SenderCanJ1939Channel.USBBUS2,  
    SenderCanJ1939Baudrate.BAUD_250K  
);  
  
IDefaultUhfReaderManager readerManager = new  
    DefaultUhfReaderManager(senderConfig);
```

The manufacturer of *DefaultUhfReaderManager* does not configure the channel. It Requires a ready-made *ISenderConfig* object , thereby giving full control over the connection type to the calling application..

From this point on, high-level controls work the same way, regardless of the type of channel you choose.

This structure makes the library extremely flexible and easily extendable to new modes of communication, maintaining consistency and ease of use for the end user.

---

### 1.2.1 Scope of use of the library

The *RFLine Reader Library* is designed for high-level software applications, such as:

- Configuration tools
- Desktop *applications* for diagnostics, testing, testing
- Real-time data viewers (monitoring tool)
- Utility for firmware update or TAG programming
- Experimental or validation applications

It is not intended for direct use in real-time industrial applications, such as:

- Automation systems on production lines
- Integrated PLC/RTU logics
- Machine interfaces with latency, synchronization or safety constraints

In case you want to integrate the reader directly into industrial systems, it is recommended to:

- Use the low-level protocol described in the official technical documentation
- directly manage communication over CAN, RS232 or TCP
- do not use the DLL, but integrate at a low level following the specifications (e.g. PGN, TP, CM, command codes)

**NOTE** - The *RFLine Reader Library* does not provide guarantees of real-time, determinism, or *hard real-time reliability*. ES:

Use case	DLL?	Notes
Reading firmware from WPF apps	YES	
Programming an EPC tag from a .NET tool	YES	
Real-time reading with on-screen logs	YES	
Configuration of the reader parameters	YES	
Integration into industrial production lines	NO	Use raw protocol
Control from PLC or embedded controller	NO	Use raw protocol

---

## 2 IUhfReaderManager *Interface Commands*

Main interface for the logical management of the RFLine UHF reader.

It provides high-level methods for device configuration, inventory management, interaction with TAGs, and diagnostics.

### 2.1 Events

The interface includes two events that describe spontaneous messages received from the layers below (and therefore, ultimately, from the *device*).

#### **Threading Notice:**

Events raised by this library may occur on background threads.

If the event handler performs any operations that interact with the UI (e.g., updating controls), it is the responsibility of the handler to dispatch those operations to the appropriate UI thread using mechanisms such as `Dispatcher.Invoke` (WPF) or `Control.Invoke` (WinForms).

#### 2.1.1 SpontaneousMessage

`EventHandler<SenderMessageReceivedEventArgs> SpontaneousMessage`

Notifies the user of a message received from the reader (e.g. standard diagnostic messages).

#### 2.1.2 TagReceived

`EventHandler<TagReceivedEventArgs> TagReceived`

Event generated upon asynchronous receipt of a tag (e.g., during continuous inventory). Contains detailed information about the detected tag.

---

## 2.2 Methods

The methods alternately return a common response or a specific response for the type of command.

The basic answer is defined as follows:

```
public class ManagerResultBase
{
    public bool Success { get; set; }
    public int MessageCode { get; set; }
    public string Message { get; set; }
}
```

Describes the outcome of the command with a possible return error code

Specific responses extend this base class with additional data based on the specific command.

Possible error codes (constants of the `UhfReaderManagerErrorCodes` class) are:

Name	Code	Notes
EmptyResponse	-201	The response received is blank
InvalidResponseFormat	-202	The response received is not in the expected format
InvalidInput	-203	Command not executed due to invalid parameters
DeviceCommandError	-204	
CorruptedResponse	-205	The answer, while valid in format, has one or more specific fields/sections/parameters that do not conform (e.g. truncated or missing value). Typical of commands that return data.
GenericException	-299	Generic exception

Each command can also return an error value expected by the low-level command (see specific protocol in correspondence with NAK responses).



---

### 2.2.1 GetFirmwareVersion

```
ManagerResult<FirmwareVersionData> GetFirmwareVersion()
```

Returns information about the currently installed firmware version.

FirmwareVersionData Specific Response:

```
string Version
```

Possible error codes:

- EmptyResponse
- InvalidResponseFormat
- CorruptedResponse
- GenericException

### 2.2.2 FirmwareUpdate

```
ManagerResultBase FirmwareUpdate(  
    byte[] binaryByteCode,  
    Action<int> progressCallback  
)
```

Update the player's firmware with a binary image.

Parameters:

- **binaryByteCode:** An array of bytes containing the new firmware.
- **progressCallback:** A *callback* function to notify progress (0–100%).

Possible error codes:

- EmptyResponse
- InvalidInput
- InvalidResponseFormat
- CorruptedResponse
- GenericException

### 2.2.3 DeviceReset

```
ManagerResultBase DeviceReset()
```

The command is used to send the reset to the device, which, once answered to the command, will perform a restart

Possible error codes:

- EmptyResponse
- InvalidResponseFormat
- GenericException

---

## 2.2.4 SetDeviceActivationState

```
ManagerResultBase SetDeviceActivationState(  
    bool isActive  
)
```

Turn the player on or off.

Parameters:

- `isActive`: *true* to activate the device, *false* to deactivate it.

Possible error codes:

- `EmptyResponse`
- `InvalidResponseFormat`
- `GenericException`

## 2.2.5 ReadReflectedPower

```
ManagerResult<ReflectedPowerData> ReadReflectedPower(  
    antennaNumber byte  
)
```

Reads the power level reflected on the specified antenna port.

NOTE: The return value is not the classical return loss; instead, the VSWR (Voltage Standing Wave Ratio) is returned.

Parameters:

- `antennaNumber`: number of the antenna port to be analyzed (e.g. 1, 2, ...).

**ReflectedPowerData** Specific Answer:  
decimal ReturnLoss

Possible error codes:

- `EmptyResponse`
- `InvalidResponseFormat`
- `CorruptedResponse`
- `GenericException`

---

## 2.2.6 WriteROMParameters

```
ManagerResultBase WriteROMParameters<T>(
    ROMSections section,
    T data
)
```

Writes the parameters to the specified ROM section.

Parameters:

- **section:** section of the ROMs to be updated; values taken from the ROMSections enumerative
- **data:** object containing the parameters to be written

The generic object `T` given has a structure that depends on the section of interest (different sections have different configuration parameters)

ROMSections and related parameter classes.

Name	Code	Parameter Specific Class
Generic	0	ROM_Generic
Antennas	2	ROM_AntennasSection
ChannelsFrequencies	3	ROM_FrequenciesChannelsSection
Reader	4	ROM_ReaderSection
Tags	5	ROM_TAGSection

Class ROM\_Generic:

```
byte DeviceAddress
bool OperationalMode (not used)
byte InventoryDuration (0.1 s units)
IPAddress Tcp_IPAddress
IPAddress Tcp_SubnetMask
IPAddress Tcp_Gateway
int Tcp_Port
int Serial_BaudRate
int Serial_DataBits
int Serial_StopBits
ParityType Serial_Parity
```

OperationalMode values:

- 0: Inventory on demand
- 1: Real-time reporting (spontaneous)
- 2: Internal buffer with querying

Class ROM\_AntennasSection:

```
List<AntennaParameters> Antennas
```

where AntennaParameters:

```
byte Number
ushort TxPower
ushort RxPower
bool IsActive
```

---

**Class** ROM\_FrequenciesChannelsSection:

```
List<uint> Frequencies
ushort DwellPerAntennaMs
ushort DwellPerFrequencyMs
```

**Class** ROM\_ReaderSection:

```
bool UseAntennaPortAsIdentifier
bool UseTagMemoryAsIdentifier
bool UseFastIDMode
RssiUsageMode RssiUsageMode
```

where RssiUsageMode can take the following (enumerated) values:

- UseLastRead = 0
- UseMaximumSeen = 1

UseTagMemoryAsIdentifier = true causes the reader to fetch the TID (via embedded command) together with the EPC, and to use the pair (EPC + TID) as the identifier.

UseFastIDMode is supported only by certain types of tags and only in “Real-time reporting (spontaneous)” mode

**Class** ROMParameters\_TAGSection:

```
SessionType Session
TargetType Target
RfMode RfMode
QValue bytes
```

where SessionType can take the following (enumerated) values:

- Session0 = 0
- Session1 = 1
- Session2 = 2
- Session3 = 3

where the TargetType can take the following (enumerated) values:

- A = 0
- B = 1

where RfMode can take the values provided in the reference doc (q.v.).

Possible error codes:

- EmptyResponse
- InvalidInput
- InvalidResponseFormat
- GenericException

---

### 2.2.7 ReadROMParameters

```
ManagerResult<T> ReadROMParameters<T>(
    ROMSections section
)
```

Reads the parameters from the specified ROM section.

Parameters:

- `section`: ROM section to read. Values taken from the `ROMSections` enumerative.

The return value is declined on the specific class of parameters for the required section.

For more info see `WriteROMParameters` command.

Possible error codes:

- `EmptyResponse`
- `InvalidResponseFormat`
- `CorruptedResponse`
- `GenericException`

### 2.2.8 ResetROMParametersToDefault

```
ManagerResultBase ResetROMParametersToDefault(
    ROMSections section
)
```

Resets the parameters of a ROM section to factory defaults.

Parameters:

- `section`: ROM section to be reset. Values taken from the `ROMSections` enumerative.

Possible error codes:

- `EmptyResponse`
- `InvalidResponseFormat`
- `GenericException`

---

### 2.2.9 GetInventory

```
ManagerResult<InventoryData> GetInventory(  
    bool include antenna,  
    bool includeRSSI,  
    int? timeoutMs = null  
)
```

Starts a synchronous inventory scan.

Parameters:

- `includeAntenna`: *true* to include the antenna port information.
- `includeRSSI`: *true* to include the received power (RSSI) values.
- `timeoutMs`: timeout (in milliseconds) for the command. Note: this value must account for the *inventory duration* (“observation period”). If not provided, a default of 8000 ms is used.

InventoryData Specific Response:

```
List<InventoryTagData> Tags
```

where InventoryTagData:

```
byte[] Epc  
byte[] Tid  
byte? AntennaNumber  
byte? Rssi
```

Tid is null if UseTagMemoryAsIdentifier = false.

Possible error codes:

- EmptyResponse
- InvalidResponseFormat
- CorruptedResponse
- GenericException

---

### 2.2.10 ReadBufferData

```
ManagerResult<InventoryData> ReadBufferData (
    byte maxRecords,
    bool deleteAfterReading
)
```

Reads tag data stored in the internal buffer.

Parameters:

- `maxRecords`: Maximum number of records to be read
- `deleteAfterReading`: *true* to clear records after reading

InventoryData specific response.

Possible error codes:

- `EmptyResponse`
- `InvalidResponseFormat`
- `CorruptedResponse`
- `GenericException`

### 2.2.11 ReadBufferDataCount

```
ManagerResult<long> ReadBufferDataCount ()
```

Returns the number of records currently in the buffer.

Possible error codes:

- `EmptyResponse`
- `InvalidResponseFormat`
- `CorruptedResponse`
- `GenericException`

### 2.2.12 ResetBuffer

```
ManagerResultBase ResetBuffer()
```

Completely clears the tag buffer.

Possible error codes:

- `EmptyResponse`
- `InvalidResponseFormat`
- `GenericException`

---

### 2.2.13 ProgramTagEpc

```
ManagerResultBase ProgramTagEpc(  
    byte[] currentEpc,  
    byte[] password,  
    byte[] newEpcValue  
)
```

Program a new EPC value on an existing tag.

**Note** – In `newEpcValue` pass only the value of the EPC, without PC word and CRC word.

Parameters:

- `currentEpc`: Current EPC of the tag to be programmed
- `Password`: Login password (if required)
- `newEpcValue`: New EPC value to write

Possible error codes:

- `EmptyResponse`
- `InvalidInput`
- `InvalidResponseFormat`
- `GenericException`



---

## 2.2.14 ReadTagData

```
ManagerResult<TagData> ReadTagData(  
    byte[] currentEpc,  
    byte[] password,  
    TagsMemoryBank memoryBank,  
    uint startAddress,  
    byte wordCount  
)
```

Reads data from a tag-specific memory.

Parameters:

- **currentEpc**: EPC of the tag to be queried
- **Password**: Login password
- **memoryBank**: memory to be read (EPC, TID, USER, RESERVED, taken from the **enumerative** TagMemoryBank)
- **startAddress**: Starting word address
- **wordCount**: number of words to read

Specific answer TagData:

```
List<byte[]> BlockData
```

Possible error codes:

- EmptyResponse
- InvalidInput
- InvalidResponseFormat
- CorruptedResponse
- GenericException

---

### 2.2.15 WriteTagData

```
ManagerResultBase WriteTagData(  
    byte[] currentEpc,  
    byte[] password,  
    TagsMemoryBank memoryBank,  
    uint startAddress,  
    List<byte[]> listBlocksToWrite  
)
```

Writes one or more blocks of data to the tag's memory.

#### Parameters:

- **currentEpc**: EPC of the tag to be written
- **Password**: Login password
- **memoryBank**: memory to be written (EPC, TID, USER, RESERVED, taken from the **enumerative** TagMemoryBank)
- **startAddress**: Start address in word
- **listBlocksToWrite**: List of blocks to write (2 bytes for each block).

#### Possible error codes:

- EmptyResponse
- InvalidInput
- InvalidResponseFormat
- GenericException

---

## 2.2.16 Kill Tag

```
ManagerResultBase KillTag(  
    byte[] currentEpc,  
    byte[] killPassword  
)
```

Permanently deactivates a tag (irreversible).

Parameters:

- `currentEpc`: EPC of the tag to be disabled
- `killPassword`: The kill password

Possible error codes:

- `EmptyResponse`
- `InvalidInput`
- `InvalidResponseFormat`
- `GenericException`

---

## 2.2.17 Lock Tag

```
ManagerResultBase SetLocks (  
    byte[] currentEpc,  
    byte[] password,  
    LockActions killPasswordPolicy,  
    LockActions accessPasswordPolicy,  
    LockActions EPCTagPolicy,  
    LockActions TIDBankPolicy,  
    LockActions UserBankPolicy  
)
```

Set the various locks on the TAG.

### Parameters:

- **currentEpc:** EPC of the tag to be disabled
- **password:** The access password
- **killPasswordPolicy:** Policy related to the Kill Password
- **accessPasswordPolicy:** Policy related to the Access Password
- **EPCTagPolicy:** Policy related to the EPC bank
- **TIDBankPolicy:** Policy related to the TID bank
- **UserBankPolicy:** Policy related to the User bank

### Possible error codes:

- EmptyResponse
- InvalidInput
- InvalidResponseFormat
- GenericException

---

## 2.3 IDeviceManager Interface

*DefaultUhfReaderManager* implements the *IDeviceManager* interface, which is designed to be common to multiple different types of device "managers," which exposes the following methods and properties:

- `bool Connect()`
- `bool Disconnect()`
- `bool IsConnected { get; }`

The `Connect()` and `Disconnect()` methods allow you to start and end the connection with the device in the expected mode (e.g. CAN). They return a `bool` with the outcome of the operation.

The *IsConnected* property exposes the state of the connection. You might want to query it before a command is executed.

---

## 2.4 Disclaimer and Limitations of Use

This library is provided for software development support purposes and is not intended for use in safety-critical or *industrial environments*.

The user declares to be aware that:

- The library does not guarantee deterministic or *real-time behavior*.
- Asynchronous operations (e.g. tag reception, spontaneous events) are not synchronized with external systems or machine cycles.
- There is no *watchdog*, redundancy, or *failover mechanism*.
- Use in environments with latency or synchronization constraints is not recommended.
- Any malfunction of the player or communication is not automatically intercepted by the library.

### **Caution:**

Integration in contexts such as:

- Machine control (PLC, CNC, robot)
- Automated systems
- systems subject to safety regulations (e.g. SIL, ISO 13849)

must be done exclusively through the use of the low-level protocol, which is available separately in the *RFLine technical documentation*.

## 2.5 License to use

The library is distributed without express or implied warranties.

Use is permitted only for the purposes of developing, testing, validating, and configuring the RFLine UHF Reader device.

iDTRONIC GmbH declines any responsibility for any direct or indirect damage resulting from the improper use of the library.