
ATR SE READER CORE

USER GUIDE

REVISION: 01 (DRAFT)

January 2026

- Key features 7
 - Summary 7
- Scope 8
- Intended audience 9
- Terms and abbreviations 10
- Modes transition 12
 - Application modes 12
 - Transient modes 13
 - Credentials 15
- External reader API 17
 - UART interface 17
 - SLIPMUX protocol 17
 - CoAP application layer 19
 - CoAP request/response codes 19
 - CBOR payload encoding 21
- OSDP 22
 - Interface 22
 - Supported commands and responses 22
 - OSDP_MFG 22
 - OSDP_MFG data frame 23
 - SCBK key loading 25
 - Card readout 25
- Wiegand 27
 - Interface 27
 - Configuration 27
 - Data format 27
- CoAP 28
 - Block-wise transfers 28
 - CoAP POST example 28
 - CoAP GET example 29
- Virtual config cards 30
 - VCC detection 30
 - VCC processing 30
 - Key rolling procedure via ORM Mobile 30
- CPE configuration 32
- Autonomous mode 33
- Simple UI setting 35
- Audio-visual sequences 36
 - Idle LED configuration 36
 - AV templates 36
 - Default AV templates and bonding 37
- System events 38
 - Reader Friendly Name 40
- Introduction 42
- Autonomous mode workflow 43

Configuration	45
UART output interface	46
Wiegand output interface	50
Data formatter	50
Personal Identity Verification (PIV) card data formats	51
PIV CHUID FASC-N data structure	52
EM Prox formatters	54
Example configuration for card data formatter	57
Example Outputs for different formats	58
CSN formatters	64
UART CSN formats	64
CSN formats for ISO14443 cards	65
Supported commands	70
Leaving Autonomous mode	71
Card processing engine	72
Introduction	72
Autonomous readout flow	72
Interface polling	72
Protocols configuration	73
Card Data Readout Engine	74
Card Data Readout Engine interface	74
Card Data Formatting Engine	74
Features	75
Card data procedures	76
Card Data Generic Procedure structure	77
Card Action structure	78
Card Data iCLASS/PicoPass specific structures	79
Card Data MIFARE specific structures	81
Card Data MIFARE DESFire specific structures	82
Card Data Seos specific structures	85
Seos chaining procedure flow	87
Card types	92
Anti-passback	93
Configuration	93
CSN based recognition	93
Modes	94
Card tracking	95
Constant card report	95
Heartbeat	96
Uart output	96
Uart output synchronization with other events	97
Wiegand output	98
Wiegand output synchronization with other events	98
Reset event	99
CDRE actions compatibility charts	99
CDRE actions compatibility automatic readout (Autonomous mode)	99
CDRE actions compatibility (Autonomous formatter support)	100
CDRE Actions compatibility Host driven mode (ERA)	100
Introduction	102
Data readout flow	103
Introduction	105
General reader commands	106
genChangeState	107
genGetSerialNumber	109
genGetUUID	110
genGetDeviceStatus	111
genWhoAreYou	113

genGetHwInfo	115
genGetIdentity	117
genGetReleaseVersion	118
I/O commands.....	119
ioGetLedState	120
ioSetLed	122
ioGetBeeperState	124
ioSetBeeper	125
ioGetFreeAudioVisualSlot (deprecated)	127
ioSetAudioVisualSequence (deprecated)	128
ioDeleteAudioVisualSequences	134
ioBondSequenceToEvent (deprecated)	135
ioAudioVisualSequenceExecute	137
ioGetIO	139
ioSetIO	141
Card-specific commands	143
rfidScan	144
rfidSelectAndAuthenticate	146
rfidReadData	148
rfidWriteData	150
rfidValueOperation	152
rfidTerminate	154
rfidReadPacs	156
rfidTransceive.....	158
rfidCardProcessingEngineGetFreeSlot (deprecated)	160
rfidCardProcessingEngineClearProcedures	162
Key management commands	164
rollDefaultWrappingKey	165
setUserKey.....	167
getUserKeyInfo.....	169
Transparent session commands	171
rfidTransparentSessionStart.....	172
rfidTransparentSessionStop.....	174
rfidTransparentSessionConfigure.....	176
rfidTransparentSessionTransceive	178
rfidTransparentSessionSetField	180
Config commands	182
configGetParam	183
configSetParam	185
configGetDataModelVersion	187
configGetCounter.....	188
configGetUUID	189
configGetName.....	190
configSetName	191
configReset.....	193
File update commands	195
fileStore	196
fileGetLength	198
fileClear	199
fileGetUpdateStatus	200
Firmware commands	202
fwGetVersion	203
fwGetCounter.....	205
fwGetUUID.....	206
Introduction	208
Shell mode activation	208
Shell interface	208

Introduction	210
General shell commands.....	211
General shell commands description	211
General shell commands description	214
Flashlog commands	214
Get runmode command	214
Get version command	214
Tracelog commands	215
mpm command	216
help command	216
hwtest command	216
stop command	217
Hardware (HW) test commands	218
GPIO test mode	218
USART test mode.....	220
I2C test mode	221
HF test mode	222
HW test start and stop commands	225
Detailed frames description	226
GPIO test mode commands	226
USART test mode commands	228
USB test mode commands.....	230
I2C test mode commands.....	231
HF test mode commands	233
HW test start and stop commands	241
Introduction	243
Introduction	245
Configuration file.....	245
Configuration using External Reader API.....	245
Configuration description	246
Configuration parameters	247
Configuration structures	247
OID access	247
General configuration structure	248
Autonomous mode configuration structure.....	249
Pinger configuration structure.....	256
Communication configuration structure	257
Security configuration structure	258
I/O configuration structure	261
Wiegand configuration structure.....	266
OSDP configuration structure.....	267
Introduction	271
AES KEK usage.....	272
Default AES KEK rolling	272
Custom AES KEKs injection	273
AES KEKs OIDs	274
User Key encoding.....	275
Key ID	276
Introduction	278
Update over ERA.....	279
Update requirements	279
Reader update flow	279
Troubleshooting	282
Introduction	284
Physical hardening	284
Cable security	284
Regular inspection	284

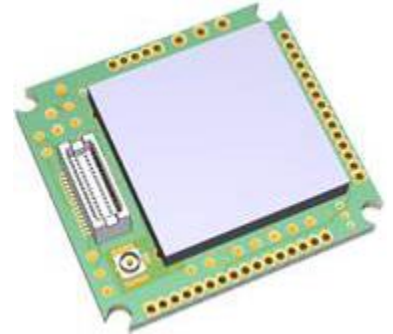
Additional security measures.....	284
Handling custom keys	284

Introduction

The ATR SE Reader Core is a derivative of the OMNIKEY® SE Reader Core™ for integration with the ATR215 range of ticket reader products.

The original OMNIKEY® SE Reader Core™ supports BLE and LF (125 kHz) communication as well as HF (13.56MHz) RFID communication. **ATR SE Reader Core only supports 13.56MHz RFID.** This manual may have occasional references to BLE & LF communication in the command structures etc. These options will have no effect in the ATR SE Reader Core.

The OMNIKEY® SE Reader Core™ is the next generation of embedded reader from HID Global® and is built on a new core platform. The OMNIKEY SE Reader Core replaces the iCLASS® SE™ Reader Module family. It includes more memory, faster performance and expanded connectivity, while still offering a secure and standards-based independent technology and flexible identity data structure based on the Secure Identity Object™ (SIO™). SIO is a portable, open, credential methodology. The OMNIKEY SE Reader Core has been designed to allow existing systems to be easily upgraded, since it uses the same format as the iCLASS SE Reader Module.



Key features

The OMNIKEY SE Reader Core allows integrators to design third-party solutions supporting a full range of contactless card technologies, including HID Prox™, iCLASS, Seos®, MIFARE Classic, MIFARE DESFire, and NFC applications. Another key feature of the OMNIKEY SE Reader Core is the straightforward field upgrade support, which allows firmware changes and feature upgrades for installed readers to extend the lifecycle of customer installations.

Connections to the OMNIKEY SE Reader Core are made via a 30-way board-to-board connector, wire-to-board PicoBlade system connectors, or pin headers to solder holes. Antenna connection is via PicoBlade or solder hole.

For quick and easy integration, HID Global also provides an OMNIKEY SE Developer Tool Kit (DTK). The DTK provides the necessary tools, documentation and developer resources to shorten integration cycles and speed time to market with finished products.

Summary

- Secure Identity Object. Multi-layer security that extends beyond the card technology, providing additional protection to identity data.
- Supports credentials including iCLASS, MIFARE Classic, MIFARE DESFire EV1/EV2/EV3, and NFC-enabled smart phones.
- Field upgradeable. Gives you the flexibility to implement new features in the field and extend the product lifecycle. A complete flexible product line of reader boards, antennas and accessories.
- Developer tools and support. OMNIKEY SE Reader Core provides a fully featured DTK providing web/desktop-based tools and documentation.
- Development support from HID Global Support Services.

Scope

This document describes the following parts of the OMNIKEY SE Reader Core:

- Reader modes of operation.
- Firmware features of the OMNIKEY SE Reader Core, including communication interfaces used for communication and control.
- Reader API description.

Intended audience

Developers should have the following knowledge:

- ISO smart card standards.
- Knowledge of RFID credentials.
- Software and firmware programming.
- RFID hardware design.
- Embedded environments.
- Data manipulation techniques.
- Good engineering practice and procedures for devices with RF interfaces.

Terms and abbreviations

Abbreviation/term	Description
ANT	Antenna
Autonomous mode	OMNIKEY SE Reader Core mode in which the reader automatically reads data out of credentials according to the reader configuration
CBOR	Concise Binary Object Representation. The data format used for encoding various data structures used by the reader.
CoAP	Constrained Application Protocol
CSN	Card Serial Number
ERA	External Reader API
External Reader API	Main communication protocol supported by the reader
External Reader API mode	Reader mode in which the reader supports the External Reader API protocol
FCC	Federal Communication Commission
FeliCa	FeliCa Card
HCE	Host Card Emulation
HF	High Frequency (13.56 MHz)
I/O	Input/Output
KEK	Key Encryption Key
LED	Light Emitting Diode
LF	Low Frequency (125 kHz)
MCU	Microcontroller
OSDP	Open Supervised Device Protocol
PHY	Physical Layer
PICC	Proximity™ IC Card
PSK	Phase Shift Keying
R&TTE	Radio and Telecommunications Terminal Equipment
RF	Radio Frequency
RFID	Radio Frequency Identification
RFU	Reserved for Future Use
SCBK	Secure Channel Base Key
SIO	Secure Identity Object
Shell	Command line interpreter supported by the reader
Shell mode	Reader mode in which the reader supports shell communication
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver Transmitter
UID	Unique Identifier
ULPM	Ultra Low Power Mode
USART	Universal Synchronous Asynchronous Receiver Transmitter
USB	Universal Serial Bus

Modes of operation

Introduction

The OMNIKEY SE Reader Core operates in several modes, which can be divided into two main types: application modes and transient modes.

Application modes:

- Autonomous mode
- External Reader API mode
- OSDP mode

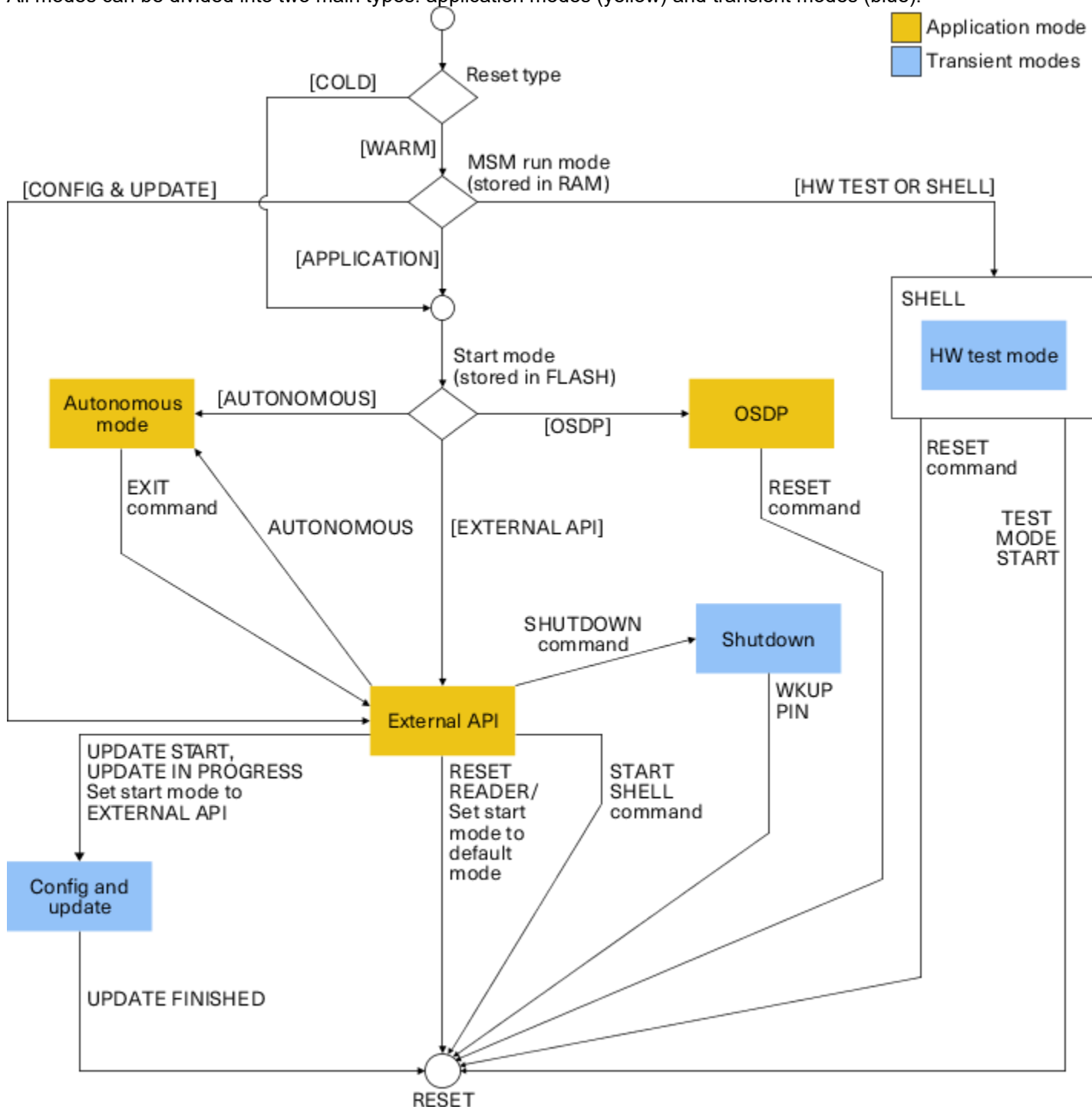
Transient modes:

- Shell mode
- HW test mode
- Shutdown
- Config and update

Application modes are modes that are permanently configured in the reader. When a specific application mode is configured, it is used by the reader until the configuration is changed to a different application mode. Transient modes are modes that you can temporarily enable in the reader. The activated transient mode is terminated by a cold or warm reset of the reader. After a reset, the reader starts in the currently configured default application mode. It is also possible to send a specific command to exit a transient mode, for example, a `shell_stop` command for Shell mode. A more detailed description of the modes and transitions is provided in the following paragraphs.

Modes transition

The following diagram shows the OMNIKEY SE Reader Core modes of operation together with the transition conditions. All modes can be divided into two main types: application modes (yellow) and transient modes (blue).



Application modes

After a cold MCU reset, the OMNIKEY SE Reader Core starts in the default mode of operation according to its internal configuration; see "General configuration structure" on page 248. The "Autonomous mode" on page 41 is set as the

default mode of operation after manufacturing. The "External Reader API mode" on page 101 can be by activated from the Autonomous mode or can be set as the default mode of operation.

Autonomous mode

In this mode, the OMNIKEY SE Reader Core waits in the IDLE state for a card presentation or mobile credential connection. When a card is detected, the credential detection algorithm is triggered. If the card is recognized and activated correctly, the OMNIKEY SE Reader Core automatically reads the credential content based on the configuration and transmits the data to the HOST over the configured channel.

From this mode, you can switch the OMNIKEY SE Reader Core into external API mode using the QUIT_AUTONOMOUS command; see "Supported commands" on page 70.

External Reader API mode

In this mode, the OMNIKEY SE Reader Core can receive and handle External Reader API commands. Of all the available commands, the general commands (see "General reader commands" on page 106) allow the External Reader API user to switch between different reader modes. From this mode, it is possible to switch the reader into the following modes by using the "genChangeState " on page 107 command:

- Transient modes:
 - Shutdown mode
 - Shell mode
- Application modes:
 - Autonomous mode

The genChangeState command is used to switch the reader into transient mode. To put the reader into application mode, the default operation mode of the reader must be changed using the "configSetParam " on page 185 command, followed by the "genChangeState " on page 107 command with the argument set to APP_MODE_RESET.

Additionally, the reader can enter into config and update mode when firmware update or configuration commands are received. However, this mode is used only during the update process and is not exposed to the user.

OSDP mode

Switching the reader into OSDP mode is necessary to enable communication over the OSDP interface. To switch to OSDP mode, you should change the default mode (see "General configuration structure" on page 248) using "configSetParam " on page 185 to set OSDP mode and reset the reader. Once the mode is enabled, any communication with an OSDP panel or OSDP enabled PC is allowed. Within the OSDP mode, the following major features are provided:

- A subset of OSDP commands from OSDP protocol version 2.1.7.
- A subset of External Reader API commands, encapsulated into an OSDP_MFG command.

For details about the OSDP interface and supported commands, see "OSDP" on page 22.

Transient modes

Transient modes cannot be set as default modes of operations. In general, to stop the reader operating in one of the transient modes, perform a COLD reset of the reader (power off/on sequence). The config and update or shutdown modes are "virtual" modes, so you cannot operate in these modes.

Shell mode

In this mode, the OMNIKEY SE Reader Core operates as a terminal and exposes several services' commands through a command line interface (see "Shell mode" on page 207). To stop operating in this mode and start default Autonomous mode, execute the SHELL STOP command or perform a COLD reset of the reader (power off/on sequence).

HW Test mode

This is a type of shell mode where additional shell commands are exposed to perform several hardware test functions such as:

- GPIO configuration.
- HF Test mode.

Shutdown mode

This device mode provides lower power consumption. The reader does not respond to any commands or detect any credentials. A wakeup event (wakeup pin or tamper) must be triggered to wake the reader. After waking, a reset is executed and the reader boots up in the default mode.

Contactless credential support

Introduction

The OMNIKEY SE Reader Core supports multiple contactless credential technologies.

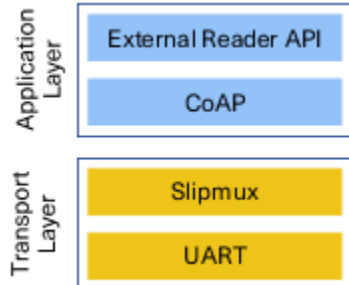
Credentials

Protocol	Supported card types
ISO/IEC 14443 Type A	MIFARE Classic MIFARE DESFire EV1 MIFARE DESFire EV2 MIFARE DESFire EV3 HID Seos
ISO/IEC 14443 Type B	-
ISO/IEC 15693	HID iCLASS

Communication

External reader API

This section outlines a communication protocol designed for interactions with the External Reader API. Data exchange is performed over a UART (Universal Asynchronous Receiver/Transmitter). The solution uses SLIPMUX for the transport layer encapsulation and CoAP (Constrained Application Protocol) for the application layer. This combination ensures reliable and efficient communication.



UART interface

The UART interface serves as the physical communication channel to the reader, allowing the exchange of data in a serial manner. The communication parameters for this UART interface are as follows:

- Baud rate: The UART operates at a configurable baud rate. The default is 115200 bits per second, but this can be changed by setting the value of OID 1.1.1.5.1.1 (see "Serial baud rate " on page 258).
- Parity: The UART employs standard parity, specifically 8N1. This means that each UART frame consists of 10 bits, including 8 data bits (to carry the payload), no parity bit (N), and 1 stop bit (1).
- The UART interface is not used exclusively for interactions with the External Reader API. Instead, it plays a central role in multiple modes of operation, including Autonomous mode and test mode. If no pull-up resistor is provided on UART-TX, the reset state may result in a logic state low appearing on the signal. This could potentially be interpreted as a null byte message.
- UART configuration parameters are not applied in SHELL mode. For UART interface parameters in SHELL mode see: "Shell interface" on page 208.

SLIPMUX protocol

The SLIPMUX protocol is used to encapsulate CoAP messages and enable their transmission over the UART interface. SLIPMUX provides a framing mechanism that allows multiple CoAP packets to be multiplexed into a single UART frame. All the CoAP frames, including External Reader API protocol frames, are encapsulated in a SLIP frame (Serial Line Internet Protocol) [RFC-1055] with the SLIPMUX extension enabling the transport of the CoAP frame.

SLIPMUX is a data link layer which provides the following services:

- Data synchronization (start and stop of the frame are easy to detect).
- Transmission error control (checksum FCS16).
- Data integrity.

The SLIP standard specifies special characters (bytes):

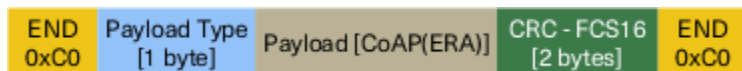
- END = 0xC0, 0300₍₈₎, indicates start or end of packet.
- ESC = 0xDB, 0333₍₈₎, indicates byte stuffing.
- ESC_END = 0xDC, 0334₍₈₎, ESC ESC_END means END data byte.
- ESC_ESC = 0xDD, 0335₍₈₎, ESC ESC_ESC means ESC data byte.

SLIPMUX specifies initial bytes which provide payload type information, for example:

- 0xA9 : CoAP message with 16-bit FCS.

The SLIPMUX frame containing the CoAP ERA request/response is structured as follows:

- Start byte (END).
- Initial byte (0xA9).
- CoAP frame with ERA request/response.
- Checksum FCS-16 (2 bytes).
- Stop byte (END).

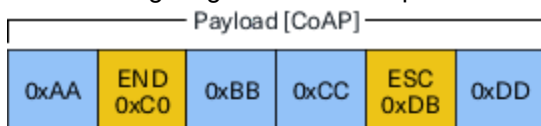


If the payload and checksum contain END or ESC bytes, they must be encoded (special bytes must be escaped):

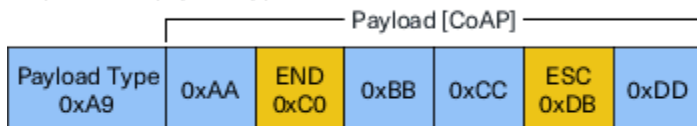
- END byte is replaced by 2 bytes ESC and ESC_END.
- ESC byte is replaced by 2 bytes ESC and ESC_ESC.

To decode a SLIPMUX frame, the reverse escape operation must be performed.

The following diagram shows the process of encapsulating a CoAP packet into a SLIPMUX frame:



Step 1: Insert payload type



Step 2: Calculate checksum



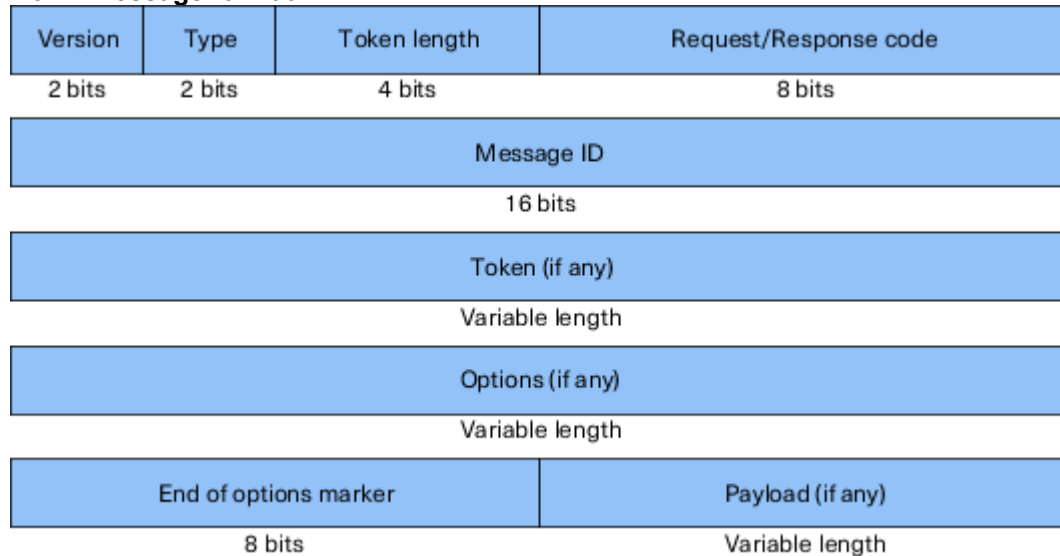
Step 3: Escape bytes and start/stop bytes



CoAP application layer

The Constrained Application Protocol (CoAP) operates as the application layer of the communication protocol. CoAP is specifically designed for constrained devices, allowing simple request-response interactions to be performed with low overhead. Refer to RFC 7252 for details.

CoAP message format



- Version: 2 bits. CoAP version (set to 1).
- Type: 2 bits. Type of CoAP message (Confirmable, Non-Confirmable, Acknowledgment, Reset).
- Token Length: 4 bits. Length of the token field.
- Request/Response code: 8 bits. CoAP request or response code.
- Message ID: 16 bits.
- Token: variable length.
- Options: variable length.
- End of options marker: 8 bits (value 0xFF). The marker is present only if there is a payload to transmit.
- Payload: variable length. CBOR encoded data.

CoAP request/response codes

CoAP uses a Class.Detail format for encoding request and response codes. Each code is encoded on 1 byte according to following table:

0	1	2	3	4	5	6	7
Class				Code			

The following descriptions are generic. See: "External Reader API commands" on page 104 for more detailed descriptions on response codes for particular commands.

Success codes (2.xx)

Code	Class.Detail	Name	Description	Byte (Hex]
65	2.01	Created	Resource created	0x41
66	2.02	Deleted	Resource deleted	0x42
67	2.03	Valid	Information is still valid	0x43
68	2.04	Changed	Resource updated	0x44
69	2.05	Content	Response contains content	0x45
95	2.31	Continue	Request continuation expected	0x5f

Client error codes (4.xx)

Code	Class.De tail	Name	Description	Byte (Hex]
128	4.00	Bad Request	Malformed syntax or invalid	0x80
129	4.01	Unauthorized	message	0x81
130	4.02	Bad Option	Authentication required or	0x82
131	4.03	Forbidden	failed	0x83
132	4.04	Not Found	Invalid option in request	0x84
133	4.05	Method Not Allowed	Server refuses to fulfill the	0x85
134	4.06	Not Acceptable	request	0x86
136	4.08	Request Entity	Resource not found	0x88
137	4.09	Incomplete	Method not supported	0x89
140	4.12	Conflict	Not acceptable by client	0x8C
141	4.13	Precondition Failed	preferences	0x8D
143	4.15	Request Entity Too Large	Block-wise transfer incomplete	0x8F
		Unsupported Content-Format	Resource conflict	
			Conditional request failed	
			Payload too large	
			Format not supported by server	

Server error codes (5.xx)

Code	Class.De tail	Name	Description	Byte (Hex]
160	5.00	Internal Server Error	Generic server error	0xA0
161	5.01	Not Implemented	Feature or method not	0xA1
162	5.02	Bad Gateway	supported	0xA2
163	5.03	Service Unavailable	Error from upstream server	0xA3
164	5.04	Gateway Timeout	Server temporarily	0xA4
165	5.05	Proxying Not Supported	unavailable	0xA5
			Timeout from upstream server	
			Proxy not supported	

CBOR payload encoding

When any request or response contains additional arguments or response data, this data is encoded in CBOR (Concise Binary Object Representation) format within the CoAP payload field. CBOR is a compact and efficient binary data serialization format. It allows the representation of complex data structures, including arrays and nested objects.

Example CoAP GET request with no payload

Hex	4001ED77B2736E	
CoAP Frame Type: Confirmable		40
CoAP Header	Code: GET	01
	Message ID: 60791	ED77
	Uri-Path: /sn	B2736E
CoAP Payload	None	

Example CoAP response with CBOR payload

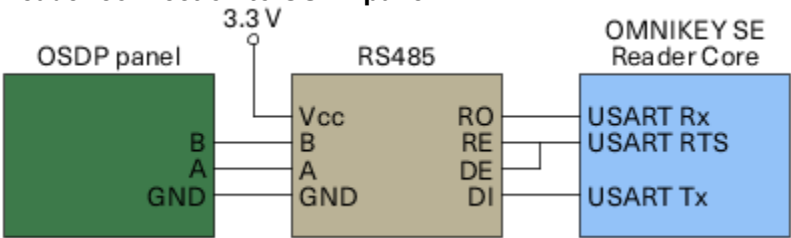
Hex	6045ED77FFA1613044AABBCCDD	
CoAP Frame Type: Acknowledgment		60
CoAP Header	Code: 2.05 Content	45
	Message ID: 60791	ED77
	End of options marker	FF
CoAP Payload: CBOR encoded data		A1613044AABBCCDD

OSDP

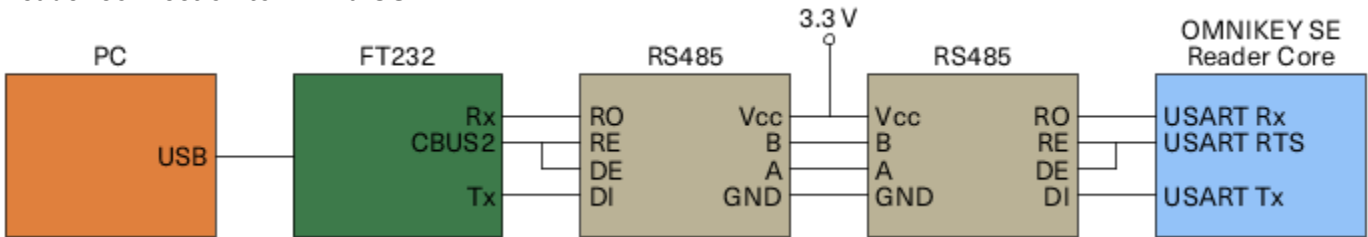
Interface

The OSDP interface is enabled in OSDP mode. It can be accessed over the USART interface, where the Rx, Tx, and RTS lines are used. The reader can be connected either to a dedicated OSDP panel via an RS485 converter, or to a PC running the OSDP DTK application. The following wiring diagrams show how to connect the reader to OSDP and PC devices.

Reader connection to OSDP panel



Reader connection to PC via OSDP



Supported commands and responses

The following OSDP commands and responses are supported by the OMNIKEY SE Reader Core.

Commands	Responses
osdp_POLL	osdp_ACK
osdp_ID	osdp_NAK
osdp_CAP	osdp_PDID
osdp_LSTAT	osdp_PDCAP
osdp_OSTAT	osdp_LSTATR
osdp_OUT	osdp_ISTATR
osdp_LED	osdp_OSTATR
osdp_BUZ	osdp_RSTATR
osdp_COMSET	osdp_RAW
osdp_KEYSET	osdp_COM
osdp_CHLNG	osdp_CCrypt
osdp_SCRIPT	osdp_RMAC_I
osdp_ABORT	osdp_MFGREP
osdp_MAXREPLY	osdp_BUSY
osdp_MFG	

For details about the commands, refer to Open Supervised Device Protocol (OSDP) version 2.1.7.

OSDP_MFG

OSDP_MFG provides company-specific, non-OSDP standard commands. Selected External Reader API commands have been encapsulated into OSDP_MFG frames. The following ERA commands have been encapsulated under OSDP_MFG:

General commands

- genStateChange
- genGetSerialNumber
- fwGetVersion

Configuration commands

- configSetParam
- configGetParam

I/O, LED, BEEPER commands

- ioSetLed
- ioGetLedState
- ioSetBeeper
- ioGetBeeperState
- ioSetIO
- ioGetIO

For more details about these OSDP commands, see "External Reader API commands" on page 104.

OSDP_MFG data frame

Encapsulating the ERA data frame into an OSDP_MFG frame is not trivial. The maximum size of an OSDP_MFG frame is limited to 64 bytes, whereas an ERA command is limited to 512 bytes (an ERA command in OSDP is limited to 512 bytes, but in ERA commands up to 1000 bytes are supported). This means the ERA frame has to be split into multiple sub-frames and sent as multiple consecutive OSDP_MFG frames.

Overview of the OSDP_MFG data frame, containing part of an ERA message

osdp_mfg header		osdp_mfg payload (64 bytes max)			
Vendor code	Command ID	Total length (2 bytes LSB)	Data offset (2 bytes LSB)	Data length (2 bytes LSB)	ERA command part (CBOR)

The values in the header are fixed:

- Vendor code: 00068E
- Command ID: 00

Example

In the following example, a single long ERA command is split into three concurrent osdp_MFG messages:

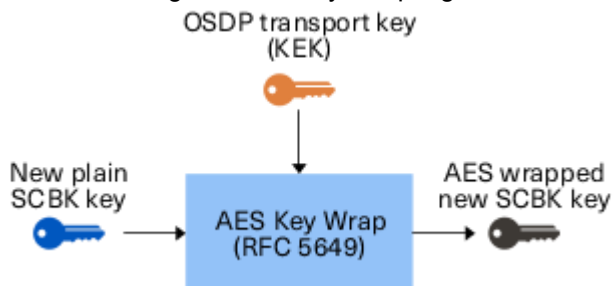
Input CoAP request of 106 bytes (POST method on /io-seq/slots resource):

```
4202A10DD97DB6696F2D73657105736C6F7473FFA26130016131584EA2613082A461301864613100
61324D8285001400140485000A000A03613300A46130006131016132498185001832001832056133
006131A361300061314D8285001400140485000A000A03613200
```

- osdp_MFG command to the reader (40 bytes, 22 bytes of ERA payload):
53002800068000068E006A00000016004202A10DD97DB669
6F2D73657105736C6F7473FFA2610A65
- osdp_ACK reply from the reader: 5380080006403BCA
- osdp_MFG command to the reader (40 bytes, 22 bytes of ERA payload):
53002800078000068E006A001600160030016131584EA261
3082A46130186461310061324D828522
- osdp_ACK reply from the reader:
5380080007400AF9
- osdp_MFG command to the reader (40 bytes, 22 bytes of ERA payload):
53002800058000068E006A002C0016008500140014048500
0A000A03613300A46130006131016528
- osdp_ACK reply from the reader:
538008000540689F
- osdp_MFG command to the reader (40 bytes, 22 bytes of ERA payload):
53002800068000068E006A00420016006132498185001832
001832056133006131A361300061F159
- osdp_ACK reply from the reader: 5380080006403BCA
- osdp_MFG command to the reader (40 bytes, 18 bytes of ERA payload):
53002400078000068E006A0058001200314D828500140014
0485000A000A03613200157B
- osdp_ACK reply from the reader:
5380080007400AF9
- osdp_POLL command to the reader:
530008000560DA99

SCBK key loading

The Secure Channel Base Key (SCBK) is used to set up a secure session with the `osdp_CHLNG` and `osdp_SCRIPT` commands. The key is always present in the reader, but you must change its default value. This is achieved with the `osdp_KEYSET` command, where an AES wrapped SCBK is expected as the command payload. SCBK key wrapping is achieved using the AES Key Wrap algorithm defined by RFC 5649.



Transport key

To wrap the new SCBK, the transport key is required. The transport key is stored under OID 03D001 and must be loaded into the reader before the new SCBK is loaded. Once the transport key is present, it can be used as the Key Encryption Key (KEK).

Example

The data listed below shows the KEK, input, and output for the AES Key Wrap algorithm:

- 16-byte transport key (KEK): 000102030405060708090A0B0C0D0E0F.
- 16-byte new plain SCBK to be stored (input): 313132333435363738393A3B3C3D3E3E
- 24-byte wrapped SCBK as payload for `osdp_KEYSET` (output):
2A265C9AFD0127AEC09AFEE30EC505C29D842E138803B0AB
- `osdp_KEYSET` command:
010A000000A01DBF2A1A8218 2A265C9AFD0127AEC09AFEE30EC505C29D842E138803B0AB

Card readout

The data read from the card is reported as an `OSDP_RAW` response. The response is sent to the `OSDP_POLL` command after the card is presented to the reader. For `OSDP_RAW` responses, only the Wiegand format is supported. The data read from the card is deleted from the reader:

- After sending data in response to the `OSDP_POLL` command.

OSDP uses the same CPE and CDRE settings as Autonomous mode, so any changes regarding supported RF technologies and the order and method of card reading can be made by changing these settings.

The following table shows an example of an `OSDP_RAW` response with card data in Wiegand format. Response data length is limited to 25 bytes.

CP command	PD Reply	PD Description
<code>OSDP_POLL</code>	<code>OSDP_RAW</code>	Card Data Report (<code>osdp_RAW</code>) (0x50)
Hex:	Hex:	b0-b1: 5380: SOM + header
FF53000700016045	FF5380240001500001C800D650185828	b2: 24 (36): length LSB (counting header and checksum)
	9D6DCACC9325A16859A46927C9D45C	b3: 00 : 00 : length MSB
	8	b4: 01 : CTL method = sequence number
	6501843E237	b5: 50 : <code>osdp_RAW</code>
		b6: 00 = reader number

CP command	PD Reply	PD Description
		b7: 01 = format code (0 = not specified, 1 = P/data/P (Wiegand) b8: C8 (200) = bit count LSB b9: 00 = bit count MSB b10-34: D6501858289D6DCACC9325A16859A46 927C9D45C86501843E2 Data, 25 bytes (200-bits) b35: 37 (checksum)

Wiegand

The Wiegand interface is a standard for connecting readers with access control systems. It defines how the devices will communicate to offer the best efficiency and performance. The OMNIKEY SE Reader Core can be configured to transmit card data over the Wiegand interface. Communication towards the reader is not possible over the Wiegand interface.

Interface

The Wiegand interface consists of three wires:

- DATA0 (nWD0-DATA-OP pin)
- DATA1 (nWD1-CLK-OP pin)
- Ground

Configuration

To enable card data output over the Wiegand interface, the Autonomous mode format under OID 1.1.1.2.1.1 must be configured to Wiegand. You can also configure:

- Wiegand pulse width: 20 to 200 microseconds in increments of 10. Default value: 40 microseconds.
- Wiegand space length: 200 to 20000 microseconds in increments of 50. Default value: 2000 microseconds.
- Wiegand guard time: 10 to 200 milliseconds in increments of 10. Default value: 10 milliseconds.

See "Wiegand configuration structure " on page 266 for details.

Data format

Wiegand supports CSN, PACS, PIV, and Custom data types. For details, see "Wiegand output interface" on page 50, "Data formatter" on page 50 and "CSN formatters" on page 64.

CoAP

Block-wise transfers

The current implementation of the CoAP protocol supports options for block-wise transfer (Block1 and Block2). The implementation complies with the RFC7959 specification. In the current implementation, the following assumptions must be met for block-wise communication:

- Total length of the command to be processed cannot be longer than 1024B.
- Subsequent command blocks must be sent sequentially.
- Preferred block length by the server is 512B.

The following examples demonstrate communication with the reader.

CoAP POST example

In the following example, a single long ERA command is split into concurrent CoAP POST block-wise messages. Input CoAP payload of POST method on /io-seq/slots resource (121 bytes):

```
A261300161315871A2613083A461300A 6131006132558385001400140585000A
000A058500181E00181E05613300A461
301461310161324D8285001400140585
000A000A05613300A461300A61310261
32518285001832001832028500183200
1832016133006131A361300A61314781
850014001405613200
```

CoAP block-wise communication (URI: /io-seq/slots, Block1 with block size: 32B):

- CoAP Tx:
42022813BB20B6696F2D73657105736C
6F7473D10309FFA261300161315871A2
613083A461300A613100613255838500
1400140585000A
- CoAP Rx:
625F2813BB20D10E09
- CoAP Tx:
42022814BB21B6696F2D73657105736C
6F7473D10319FF000A058500181E0018
1E05613300A461301461310161324D82
85001400140585
- CoAP Rx:
625F2814BB21D10E19
- CoAP Tx:
42022815BB22B6696F2D73657105736C
6F7473D10329FF000A000A05613300A4
61300A61310261325182850018320018
32028500183200
- CoAP Rx:
625F2815BB22D10E29
- CoAP Tx:
42022816BB23B6696F2D73657105736C
6F7473D10331FF1832016133006131A3 61300A61314781850014001405613200
- CoAP Rx:
62442816BB23

CoAP GET example

In the following example, a long configuration item is retrieved using CoAP GET block-wise requests.

ERA message for retrieving configuration item: A1613086010101070101

CoAP block-wise communication (URI: /cfg/param, Block1 with block size: 32B):

- CoAP Tx:
4201EC8B50A8B363666705706172616D C101FFA1613086010101070101
- CoAP Rx:
6245EC8B50A8D10A09FFA161305871A26
13083A461300A6131006132558385001
400140585000A000A05
- CoAP Tx:
4201EC8C50A9B363666705706172616D C111
- CoAP Rx:
6245EC8C50A9D10A19FF8500181E0018
1E05613300A461301461310161324D82
85001400140585000A00
- CoAP Tx:
4201EC8D50AAB363666705706172616D C121
- CoAP Rx:
6245EC8D50AAD10A29FF0A05613300A4
61300A61310261325182850018320018
32028500183200183201
- CoAP Tx:
4201EC8E50ABB363666705706172616D C131
- CoAP Rx:
6245EC8E50ABD10A31FF6133006131A3
61300A61314781850014001405613200

Received payload:

A161305871A2613083A461300A613100 6132558385001400140585000A000A05
8500181E00181E05613300A461301461
310161324D8285001400140585000A00
0A05613300A461300A61310261325182
85001832001832028500183200183201
6133006131A361300A61314781850014
001405613200

Virtual config cards

The OMNIKEY SE Reader Core supports Virtual Config Cards (VCC) v1.

VCC detection

Detection of VCC v1 credentials is performed by selecting the following AID: A000000382003D000101.

VCC detection is only possible within a detection window.

After a device reset (warm or cold), a configuration window opens. The duration of this window is defined by OID 1.1.1.6.3.2 (Wireless Configuration Time Window), with a default timeout of 20 seconds.

Upon successful VCC detection (indicated by a 9000 response to AID selection), the detection window timeout is extended to 90 seconds (the window can be extended a maximum of 5 times).

On the first successful VCC detection, the reader blocks detection of other Layer 4 credentials. This mechanism is intended to prevent the detection of mobile credentials (for example, SEOS) during VCC processing and to ensure a correct VCC processing.

If the device undergoes a state transition that includes reset, the detection window will reopen and the extension counter will reset.

In Autonomous mode VCC detection is done automatically during credential detection. In polling mode it is done periodically in pinger mode is done when pinger detects field change.

In ERA mode VCC detection is done on rfidScan call.

The 14A protocol must be enabled for VCC detection to work.

VCC processing

Once a VCC credential is detected (via a 9000 response on AID selection), processing is handed off to the Secure Element. During processing, the phone's NFC antenna must remain coupled with the reader antenna.

Each VCC session can serve various purposes, such as:

- Reading the Secure Element engine ID and parameters.
- Rolling the Secure Element keyset.
- Confirming the Secure Element keyset.

Every VCC operation must be confirmed by the reader core through the ERA_EVENT_CFG_SUCCESS event.

All VCC processing must occur within the active configuration window.

Between VCC operations, the device may temporarily detect the phone as a generic credential, potentially triggering the ERA_EVENT_CARD_REJECTED event.

Key rolling procedure via ORM Mobile

The key rolling procedure, typically consisting of three VCC processing steps, is orchestrated by the OMNIKEY Reader Manager mobile application.

During each step, the user is prompted to present the phone to the reader.

The OMNIKEY Reader Manager application manages the key rolling sequence. The reader device is unaware of the current stage of the procedure.

Low power support

Introduction

The OMNIKEY SE Reader Core can operate in various modes:

- Autonomous mode
- External Reader API mode
- OSDP
- Shutdown

There is no additional low-power mode (formerly ULPM, ultra low power mode) as power saving is an integral part of the design of the operation modes.

The reader is designed to automatically switch between various operations states (like pinging / polling / advertising etc.) according to configured schema and the current operation mode.

CPE configuration

CPE configuration has a significant impact on the power consumption of the reader since CPE is used as a main credential readout engine. CPE configuration includes, but is not limited to, the following parameters:

- Pingerless mode
- Mode
- Pinging / polling interval

Based on the configuration, you can eliminate redundant operations to save power and order protocols to optimize for the expected credential population. Moreover, you can adjust the pinging/polling configuration to obtain a preferable balance between power consumption and card readout time (faster pinging causes faster card readout but higher power consumption).

Autonomous mode

In this mode, by default, the reader wakes automatically with a defined interval (pinging interval). If a card is detected by the pinger, the reader polls the configured interfaces and if data is present, sends it to the host. Whenever no action is ongoing (like pinging) it automatically enters the power saving state.

If a credential exists, the reader reads the data from it, sends it to the host, and enters the power saving state without waiting for any action by the host.

As there are various interfaces and features which can be configured (see "CPE configuration" on page 32) the final power consumption will vary.

The reader can be woken up from power saving state by:

- Internal reader event:
 - Pinger or polling interval event.
- Host:
 - Wake-up byte sent over UART.

If the host triggers a wake-up, it must wait 2 ms for the reader to wake before any command can be sent. The reader core is then in the wake-up state and returns to the low-power mode after 2 s with no events.

Autonomous mode low power

User interface

Introduction

The user interface of the OMNIKEY SE Reader Core consists of multiple elements:

- Three LEDs, allowing color mixing.
- Buzzer.

The user interface is designed to indicate:

- Static reader state.
- Result of last executed operation (for example, card read).
- User defined audio-visual sequences.

Simple UI setting

The user interface can be set on demand using dedicated ERA commands. The UI state that you set overwrites any previously existing state. To define a new sequence, clear sequence to event bonding. For details, see "I/O commands" on page 119.

Audio-visual sequences

The user interface can be set automatically in response to a generated system event.

Once an audio-visual sequence is defined and bonded to a specific system event, the audio-visual sequence is triggered once the event is generated. Each event also has a specified priority. For more details, see "ioBondSequenceToEvent (deprecated) " on page 135.

Idle LED configuration

The idle LED indicates that the reader is in the idle state. It is enabled automatically after reader startup in Autonomous mode and disabled when the reader quits Autonomous mode.

AV sequences bonded with system events can temporarily disable the idle LED, but it will be turned ON again after sequence execution. An exception is an infinite sequence, where the idle LED will not be turned on again.

The color of the idle LED can be selected by configuration OID 1.1.1.7.1.0.

You can change the configuration by writing suitable values to the OID using a "configSetParam " on page 185 ERA command. The current configuration can be read using a "configGetParam " on page 183 ERA command.

AV templates

The reader has 14 sequence slots stored in OIDs 1.1.1.7.1.1 to 1.1.1.7.1.14. (these OIDs are empty and can be used to define new sequences). These sequences can be bonded with system events and triggered when a specific event occurs. You may overwrite any of these sequences. The idle LED indicates that the reader is in the idle state. It is enabled automatically after reader startup in autonomous mode and disabled when the reader quits autonomous mode.

AV sequences bonded with system events can temporarily disable the idle LED but it will be turned ON again after sequence execution. An exception is when user has defined infinite sequence, where the idle LED will not be turned ON again.

Color of idle LED can be selected by configuration OID 1.1.1.7.1.0. You can change the configuration by writing suitable values to the OID using a "configSetParam " on page 185ERA command. The current configuration can be read using "configGetParam " on page 183ERA command.

Default AV templates and bonding

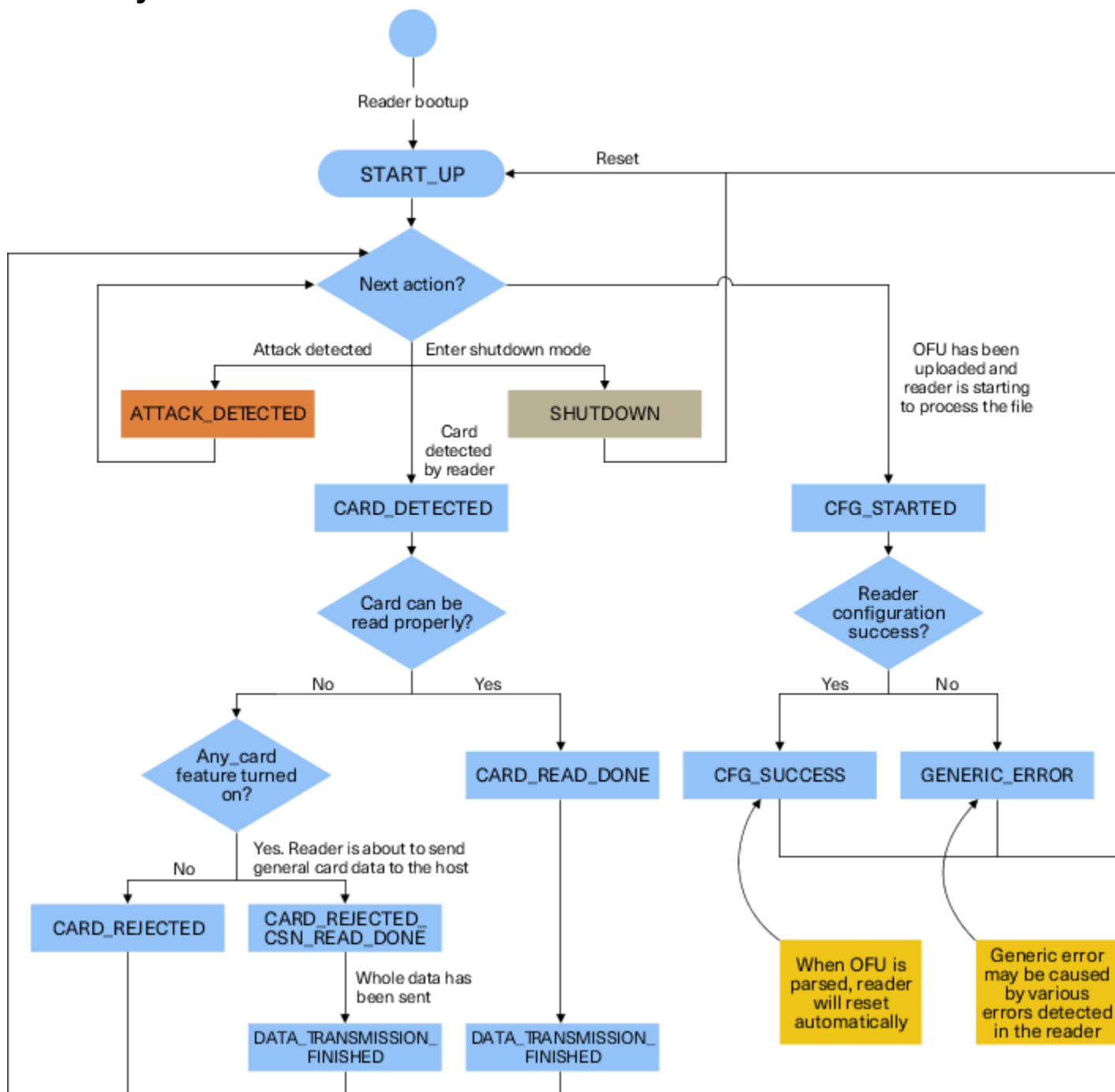
By default, events are bonded to predefined system events with an event priority. The following table describes the predefined AV events.

AV slot number	Event	Event priority	AV template description
1	ERA_EVENT_ATTACK_DETECTED	0	Infinite loop: 3 × (500 ms LED RED ON, 500 ms LED RED OFF) then 3 seconds off). Reader reset required to turn indication off.
2	ERA_EVENT_GENERIC_ERROR	1	3 × (500 ms LED RED ON, 500 ms LED RED OFF) then 3 seconds off. 1 × buzzer beep for 1.5 ms seconds, then 100 ms off.
3	ERA_EVENT_CARD_DETECTED	3	1 × (250 ms of LED GREEN ON + 100 ms LED GREEN OFF).
4	ERA_EVENT_CARD_READ_DONE	2	1 × (300 ms of buzzer beep, then buzzer off for 100 ms) + 3 × (300 ms LED GREEN ON + 300 ms LED GREEN OFF).
5	ERA_EVENT_CARD_REJECTED	2	3 × (300 ms LED RED ON + 300 ms LED RED OFF) then 3 seconds off.
6	ERA_EVENT_CFG_STARTED	3	3 × (300 ms of LED YELLOW on, 300 ms LED YELLOW off).
7	ERA_EVENT_CFG_SUCCESS	2	3 × (500 ms of LED GREEN, 500 ms LED GREEN off + 500 ms of buzzer beep, then 500 ms of buzzer off).

EVENT_CARD_REJECTED_CSN_READ_DONE	Indicates that an unknown card or card not configured to be read has been found. The reader will still send CSN and card type to the host (any_card feature is turned on).	CPE could not read data from the card (CPE was not configured to read presented card) and any_card feature is turned on.
EVENT_DATA_TRANSMISSION_FINISHED	Indicates that the reader has already sent whole data to the host.	Whole data has been sent to the host.
EVENT_CFG_STARTED	Indicates that a config/fw file (OFU) has been uploaded and parsing has started.	OFU file has been properly uploaded and parsing was started (parse has been sent).
EVENT_CFG_SUCCESS	Indicates that a config/fw file (OFU) has been properly applied.	OFU file processing is finished.
EVENT_GENERIC_ERROR	Indicates that some unexpected behavior/error has been found.	Example: Failed to update the reader using configuration file.

If an event is generated before shutdown or reset of the reader, it will be shortened to the predefined timeout. The timeout is currently set to 10 s.

Reader system events



Reader Friendly Name

Reader Friendly Name is a feature that provides the user a simple mechanism to identify a specific OMNIKEY SE Reader Core device. The user can set unique name for each Reader in use. To set or get a Reader Friendly Name configuration, OID should be used, see "I/O configuration structure " on page 261. Reader Friendly Name is also returned as part of "genWhoAreYou " on page 113 command.

Autonomous mode

Introduction

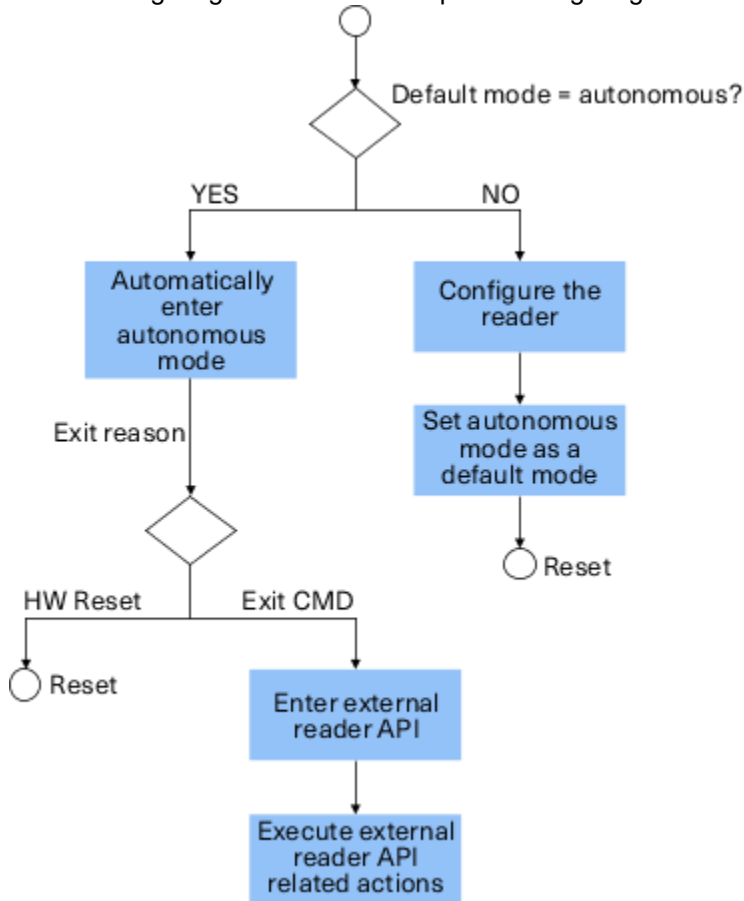
The OMNIKEY SE Reader Core supports Autonomous mode, which allows the host device to seamlessly receive credential data read from the credential.

Autonomous mode workflow

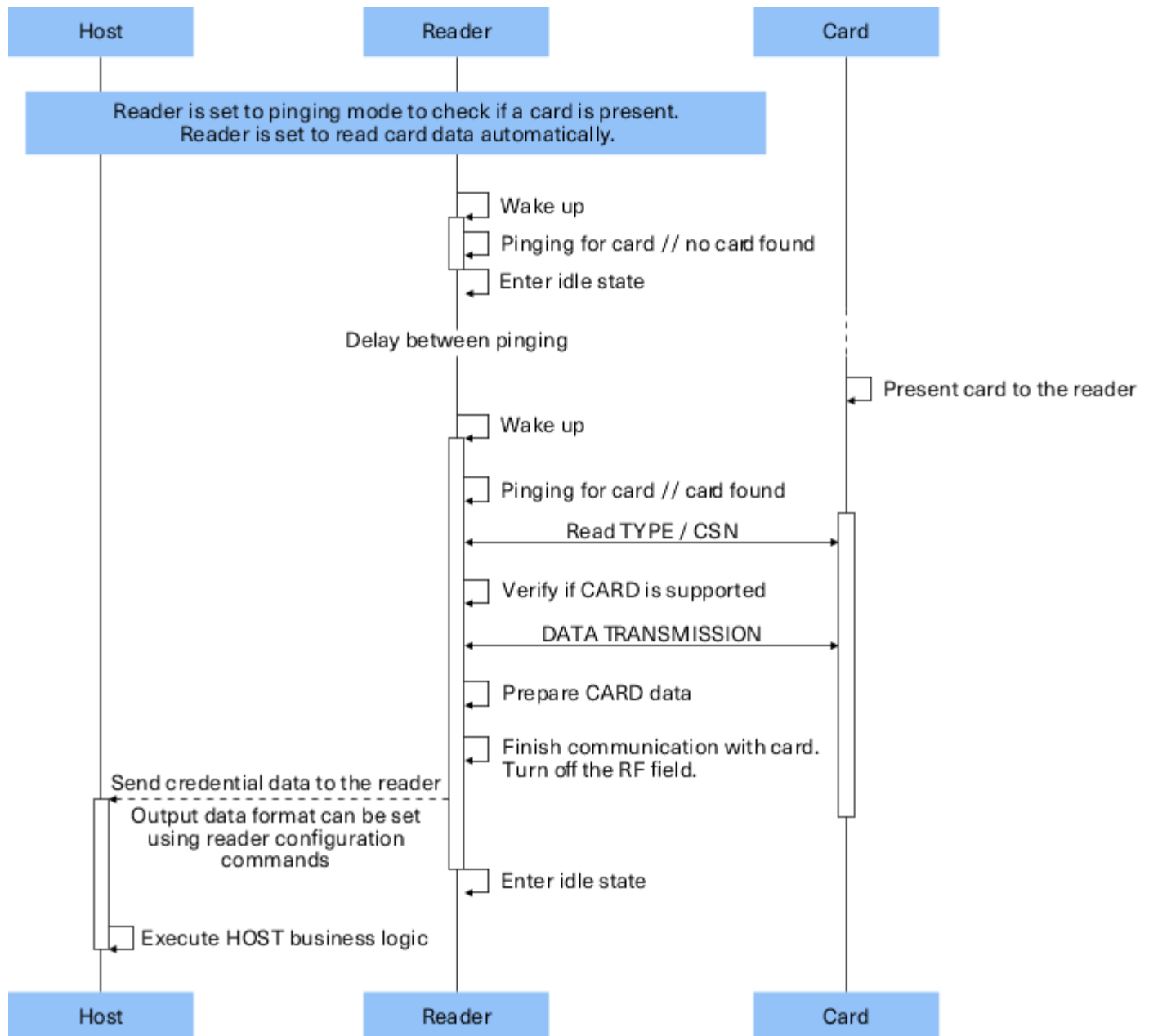
Autonomous mode requires the following steps to operate correctly:

- Credential keys preparation (the reader comes already provisioned with proper credential keys).
- Configure Autonomous mode.
- Enter Autonomous mode.

The following diagram shows the steps for configuring and entering Autonomous mode.



After entering Autonomous mode, the reader automatically sends the data to the host over UART whenever there is a card presented to the reader. The output format of the data is configurable.



Configuration

To configure Autonomous mode, use the External Reader API's `configSetConfigParam` command or configuration via file. There are several parameters which can be used to configure the reader:

Autonomous as default mode

This parameter sets the reader's default mode. For the configuration parameter details, see "General configuration structure" on page 248.

Autonomous mode output interface

This parameter configures the output interface of the Autonomous mode. It defines if credential data will be sent over UART or Wiegand. For the configuration parameter details, see "Autonomous mode configuration structure" on page 249.

Autonomous mode card-out message output enable

This parameter enables the CARD OUT output message/event in Autonomous mode. If enabled, a CARD OUT message (in text mode) or CARD OUT event (in binary mode) is sent when the card is removed from the reader. For the configuration details, see "Autonomous mode configuration structure" on page 249.

Autonomous mode - Card Processing Engine configuration

A group of parameters are available to configure the Card Processing Engine (see "Card processing engine" on page 72 and "Features" on page 75).

- RF priority.
Configures the RF priority for the Card Processor Engine.
- Enhanced polling.
Configures enhanced polling for the Card Processor Engine.
- HF protocols order configuration.
Contains a CBOR encoded HF protocols priority configuration. Each byte represents a single protocol. The protocol scanning order will be from the first to last item.
- Mode configuration.
Configures mode for the Card Processor Engine.
- CPE slots configuration.
There are 10 slots, where you can define HF credential readout procedures. For a detailed description, see "Card processing engine" on page 72.

Autonomous mode delay before sending credential data

This parameter configures the time that the reader waits between detecting a credential and sending its data. For the configuration parameter details, see "Autonomous mode configuration structure" on page 249.

Autonomous mode any card

This parameter configures the enabling/disabling of autonomous output when card data reading fails (the card does not contain the requested data). If enabled, the reader transmits the card CSN and type even if no data could be read. For the configuration parameter details, see "Autonomous mode configuration structure" on page 249. It is usually enabled for credential verification purposes.

Autonomous mode UART data encoding

This parameter configures the UART data output encoding. It is possible to choose between binary (CBOR) and text (human readable) encoding. For a more detailed description, see "UART output interface" on page 46. For the configuration parameter details, see "Autonomous mode configuration structure" on page 249.

Autonomous mode UART output text format

This parameter configures the data output format in UART text mode. You can choose between ASCII and RAW. For a more detailed description, see "Encoding: Binary" on page 47. For the configuration parameter details, see "Autonomous mode configuration structure" on page 249.

Autonomous mode card data formatting

There are 10 slots where you can define formatting rules for a card or group of cards. For a more detailed description, see "Data formatter" on page 50. For the configuration parameter details, see "Card output data formatter slot 0-9 " on page 255.

Autonomous mode facility code

This parameter configures the facility code required in the Wiegand CSN output format (see "CSN formatters" on page 64). For the configuration parameter, see "Autonomous mode configuration structure" on page 249.

UART output interface

This section describes one of the two Autonomous mode output interfaces. In this mode, data is sent over the UART interface according to the user configuration. There are two encoding variants available.

PACS and custom leading byte

PACs and custom data send over UART contains leading byte which indicates the number of unused bits in last byte used for the bitwise readout. This byte is always added, even if no padding bits are added (no unused bits are to be skipped). In such case, first byte is set to 0x00.

The leading byte is not included if TEXT+ASCII data format is selected.

Data		Description	Configurable	Encodings
Card type	Type of card read.		No	Binary/Text
CSN	Card serial number.		Yes	Binary/Text
PACS	PACs data read from card (if not TEXT + ASCII mode is selected, contains leading byte which indicates the number of unused bits in last byte).		Yes	Binary/Text
Custom	Custom data read from card (if not TEXT + ASCII mode is selected, contains leading byte which indicates the number of unused bits in last byte).		Yes	Binary/Text
CARD IN	Message instructing the user to place a card into reader range.		No	Binary/Text
CARD OUT	Message instructing the user to remove a card from reader range.		Yes	Binary/Text

Encoding: Binary

In binary mode, the data is encoded into CBOR format according to the following structure:

```
autonomous_mode_data = {
  "0": event_type_enum,
  ?"1": credential_data_structure
}

event_type_enum = (
  CARD_IN = 0x02,
  CARD_OUT = 0x03
)

credential_data_structure = (
  card: card_type,           ; type of detected card, see: Card Types
  cs_n: bstr,                ; card serial number
  bits: bstr,                ; PACS (custom data) bits, can be empty if PACS not present
  custom_data_array: [0*10 bstr] ; array of custom data byte strings (present if appropriate
                                ; actions are configured, otherwise empty array
)

card_type = &card_type_enum

card_type_enum = (
  CARD_TYPE_ICLASS_PICOPASS: 0,
  CARD_TYPE_MIFARE_CLASSIC_IK: 1,
  CARD_TYPE_MIFARE_CLASSIC_4K: 2,
  CARD_TYPE_MIFARE_PLUS: 3,
  CARD_TYPE_MIFARE_DESFIRE: 4,
  CARD_TYPE_MIFARE_ULTRALIGHT: 5,
  CARD_TYPE_SEOS: 6,
  CARD_TYPE_ISO15693: 7,
  CARD_TYPE_HID_PROX: 8,
  CARD_TYPE_PIV: 10,
  CARD_TYPE_INDALA: 21,
  CARD_TYPE_EM_PROX: 23,
  CARD_TYPE_TYPE_A_LAYER3: 32,
  CARD_TYPE_TYPE_A_LAYER4: 33,
  CARD_TYPE_TYPE_B_LAYER3: 34,
  CARD_TYPE_TYPE_B_LAYER4: 35,
  CARD_TYPE_UNKNOWN: 36,
)
```

CBOR output data example with leading bytes equal to 00 (no padding bytes)

```
CBOR: A2613002613184014C001122334455667788AABBCC4801020304050507088247000A0B0B
0D0E0F46001112131415
```

Diagnostic form: {"0": 2, "1": [1, h'001122334455667788AABBCC', h'0102030405050708', [h'000A0B0B0D0E0F', h'001112131415']]}

Decoded data:

```
event_type: 2 (CARD IN)
card_type: 1
```

csn: 001122334455667788AABBCC
bits: 0102030405050708
custom_data_array: [000A0B0B0D0E0F, 001112131415]

The same data read from card with following bitwise configuration for custom data: bit_offset = 4, bit_length = 36
CBOR:

A2613002613184014C001122334455667788AABBCC480102030405050708824604A0B0B0D0E046041121314150
Diagnostic form: {"0": 2, "1": [1, h'001122334455667788AABBCC', h'0102030405050708', [h'04A0B0B0D0E0',
h'041121314150']}]}

Decoded data:

event_type: 2 (CARD IN)

card_type: 1

csn: 001122334455667788AABBCC

bits: 0102030405050708

custom_data_array: [04A0B0B0D0E0, 041121314150] // bit_offset is set to 4 - data is shifted by 4 bits to the left,
bit_length is set to 36 - it is 8 bytes and 4 bits of data, so on last byte 4 bits are unused (padding byte 04)

The same data read from card with following bitwise configuration for custom data: bit_offset = 0, bit_length = 9

CBOR: A2613002613184014C001122334455667788AABBCC480102030405050708824307A08043071100

Diagnostic form: {"0": 2, "1": [1, h'001122334455667788AABBCC', h'0102030405050708', [h'07A080', h'071100']}]}

Decoded data:

event_type: 2 (CARD IN)

card_type: 1

csn: 001122334455667788AABBCC

bits: 0102030405050708

custom_data_array: [07A080, 071100] // bit_offset is set to 0 - there is no data shift, bit_length is set to 9 - it is 1 byte
and 1 bit of data, so on last byte 7 bits are unused (padding byte 07)

Encoding: Text

This mode encodes data into human readable format; see the following example. Text mode can present data in two formats: RAW and ASCII, which applies only to PACS and Custom data. In RAW format, each byte of data is converted to hexadecimal numbers, while ASCII treats every byte as an ASCII character. Some card data fields can be configured to be included or skipped in the output data.

End-Of-Line characters in card data output are CRLF [0x0D, 0x0A]

RAW and ASCII formats

Format	Example
RAW	CARD IN Type: MIFARE Classic 1K CSN: 0453636A533F80 PACs: 0034353637383930313233 Custom: 0048656C6C6F576F726C64 CARD OUT
ASCII	CARD IN Type: MIFARE Classic 1K CSN: 0453636A533F80 PACs: 4567890123 Custom: HelloWorld CARD OUT

RAW and ASCII formats with bitwise configuration for custom data: bit_offset = 8, bit_length = 72

Format	Example
RAW	CARD IN Type: MIFARE Classic 1K CSN: 0453636A533F80 PACs: 0034353637383930313233 Custom: 00656C6C6F576F726C64 CARD OUT
ASCII	CARD IN Type: MIFARE Classic 1K CSN: 0453636A533F80 PACs: 4567890123 // note that leading byte is not send Custom: elloWorld // note that leading byte is not send CARD OUT

Wiegand output interface

Data can be transmitted over the Wiegand protocol according to the user configuration. Observe the following points before using the Wiegand interface:

- The Wiegand interface must be configured if required, otherwise the default configuration is applied. See "Wiegand" on page 27.
- The data size is limited to 25 bytes maximum.
- Data sent via the interface is a concatenation of all data read according to actions defined via the CPE procedure, in the order defined in CPE. For example, if the CPE procedure is defined to read PACS and CUSTOM data, the output will contain PACS bits and custom bits concatenated into a single data output. There is no delimiter between different data types. For example, if PACS data consists of 26 bits, custom data will be stored starting from bit 27. As no delimiter exists, the data sizes must already be known to be able to parse the output or to limit actions in the CPE procedure to a single read action when Wiegand is selected.
- PACS and custom data over Wiegand is sent in raw form; there are no additional bytes/bits like in the UART output.
- When bitwise configuration is not empty, only requested bits are sent over Wiegand interface. For example, if user set bitwise_offset to 8 and bitwise_length to 9, second byte of data readout and 1 bit (MSB) of third data readout will be sent.

Data formatter

There are 10 configuration slots which allow you to configure the data output formatting. Each slot can store the configuration for one card type. You can configure a slot to format card output. It is possible to format particular data type (CSN, PACs, Custom) and optionally discard any of them if needed. CSN formatting options are divided into Wiegand and UART formats (with separate configurations). Content consists of CBOR encoded card formatter configuration. The configuration will also be rejected if there are duplicate card settings.

Card formatter configuration structure

Structure	Description
card_data_formatter_slot = ("0" : card_edge_type, "1" : formatter_config)	Definition of single Card Data Formatter slot.
card_edge_type = (CARD_EDGE_HF : 0, CARD_EDGE_LF : 1)	Flag defining which config is set.
formatter_config = bstr	CBOR structure which defines a certain configuration. Must be hf_formatter_config.
hf_formatter_config = ("0" : hf_card_types, ? "1" : csn_format, ? "2" : pacs_format, ? "3" : custom_data format)	Definition of formatting used for HF credentials.
hf_card_types = [1*1 card_type] card_type = see "Card types" on page 92	Array of one element containing ID of cards defined by this slot. Unsupported card types will be ignored during processing. Only HF card types can be configured here.
csn_format = (? "0" : discard_flag, ? "1" : wiegand_csn, ? "2" : uart_csn)	Defines the CSN format. discard_flag is optional (used if CSN is to be discarded from data sent to the HOST). Use cases: If you want to configure csn format, a specific format (for Wiegand or UART) must be set, but the discard flag does not need to be set (discard_flag set to false). If you want to discard csn, wiegand_csn and uart_csn do not need to be set (will be skipped anyway) but discard_flag must be set to true.

Structure	Description
	At least one of the optional parameters must be set, otherwise the configuration will be rejected. This is true for all substructures: csn_format, pacs_format, custom_data_format. Since all of them contain only optional parameters, if used, at least one must be set.
pacs_format = (? "0" : discard_flag, ? "1" : custom_format)	Defines the PACS format. discard_flag is optional (used if PACS data is to be discarded from data sent to the HOST). Currently, custom_format is not supported so the only valid configuration here is to set discard_flag.
custom_data_format = (? "0" : discard_flag, ? "1" : custom_format)	Defines the CUSTOM format. discard_flag is optional (used if PACS data is to be discarded from data sent to HOST). Currently, custom_format is not supported so the only valid configuration here is to set discard_flag.
wiegand_csn = ("0" : &uart_csn_enum, "1" : reversed_flag)	Configuration defining the wiegand CSN format, see "CSN formatters" on page 64.
uart_csn = ("0" : &uart_csn_enum, "1" : reversed_flag)	Configuration defining the UART CSN format, see "CSN formatters" on page 64.
discard_flag = bool	Defines if a particular data type will be filtered out from the output. If set to true, data will be filtered out.
custom_format = bstr	RFU. Custom formatting; will be different per credential type and data type. Currently, this field is not supported, so it should not be set during configuration of the reader as it will be rejected.

Personal Identity Verification (PIV) card data formats

The OMNIKEY SE Reader Core is capable of reading the CHUID (Card Holder Unique Identifier) data object from a PIV card.

The content of the CHUID (more detailed information can be found in NIST Special Publication 800-73-4).

Card Holder Unique Identifier (CHUID)

Data Element (TLV)	Tag	Type	Max. Bytes*
Buffer Length (Optional)	0xEE	Fixed	2
FASC-N	0x30	Fixed	25
Organizational Identifier (Optional)	0x32	Fixed	4
DUNS (Optional)	0x32	Fixed	4
GUID	0x33	Fixed	9
Expiration Date	0x34	Fixed	16
Cardholder UUID (Optional)	0x35	Fixed	8
Issuer Asymmetric Signature	0x36	Fixed	16
Error Detection Code	0x3E	Variable	2816**
Error Detection Code (LRC)	0xFE	LRC	0

The following data is read by the OMNIKEY SE Reader Core and formatted as an output:

- Federal Agency Smart Credential Number (FASC-N)
- Global Unique Identification (GUID) number
- Expiration date

PIV CHUID FASC-N data structure

The FASC-N is a BCD (Binary Coded Data) credential number for which data structure is presented below:

FASC-N

SS	Agency Code	FS	System Code	FS	Credential Number	FS	CS	FS	ICI	FS	PI + OC + OI + POA	ES	LRC
----	-------------	----	-------------	----	-------------------	----	----	----	-----	----	--------------------	----	-----

FASC-N (200b)																								
D4	32	48	58	21	0C	2D	31	71	B5	25	A1	68	5A	08	C9	2A	DE	0A	61	84	32	48	43	E2

B	1	3	4	1	D	0	0	0	1	0	0	0	1	0	1	0			
1101	0	1000	0	1100	1	0010	0	1000	0	1011	0	0000	1	0000	1	0000	1	1000	0
D	9	8	7	6	5	4	D	1	D	1	0010	0	1011	0	1000	0	1011	0	
1011	0	1001	1	0001	0	1110	0	0110	1	1010	1	0010	0	1011	0	1000	0	1011	0
1	D	1	2	3	4	5	6	7	8										
1000	0	1011	0	1000	0	0100	0	1100	1	0010	0	1010	1	0110	1	1110	0	0001	0
9	0	1	1	3	4	1	1	F	8										
1001	1	0000	1	1000	0	1000	0	1100	1	0010	0	1000	0	1000	0	1111	1	0001	0

FASC-N		FS	Field Separator
- Agency Code (AC): 4 BCD digits = 1341 - System Code (SC): 4 BCD digits = 0001 - Credential Number (CN): 6 BCD digits = 987654 - Credential Series (CS): 1 BCD digit = 1 - Individual Credential Issue (ICI): 1 BCD digit = 1 - Person Identifier (PI): 10 BCD digits = 1234567890 - Organizational Category (OC): 1 BCD digit = 1 - Organizational Identifier (OI): 4 BCD digits = 1341 - Organizational Category (POA): 1 BCD digit = 1		SS	Start Sentinel
		ES	End Sentinel
		X	Parity bit
		LRC	LRC

PIV CHUID format summary

The summary of supported PIV data formats.

Format	SS,ES, FS, LRC	Parity	Reversed BCD Bits	FASC-N				Data					GUID	Exp.Date
				AC	SC	CN	CS	ICI	PI	OC	OI	POA		
BITS_328*	x	x		x	x	x	x	x	x	x	x	x	x	
BITS_264**	x	x		x	x	x	x	x	x	x	x	x		x
BITS_200*	x	x		x	x	x	x	x	x	x	x	x		
BITS_160_LSB*	x			x	x	x	x	x	x	x	x	x		
BITS_160_MSB*	x			x	x	x	x	x	x	x	x	x		
BITS_128_LSB*				x	x	x	x	x	x	x	x	x		
BITS_128_MSB*				x	x	x	x	x	x	x	x	x		
BITS_75*				x	x	x								x
BITS_64_LSB*				x	x	x	x	x						
BITS_64_MSB*				x	x	x	x	x						
BITS_58***				x	x	x	x	x						
BITS_40_LSB*					x	x								
BITS_40_MSB*					x	x								

* for PIV-I card and enabled PIV GUID fallback flag the output is GUID (128b).

** for PIV-I card and enabled PIV GUID fallback flag the output is concatenation of GUID(128b) and Expiration Date (64b).

*** PIV GUID fallback flag is ignored.

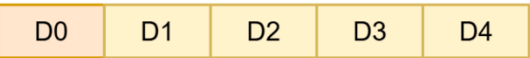
PIV CHUID formatter configuration structure

The PIV CHUID format configuration is provided by defining custom format for PIV card. The custom format data structure is defined below:

Structure	Description
custom_data_format = "0" : piv_chuid_format "1" : piv_guid_fallback	Defines format type of PIV CHUID data.
piv_chuid_format = (BITS_40_MSB: 0x00, BITS_40_LSB: 0x01, BITS_58: 0x02, BITS_64_MSB: 0x03, BITS_64_LSB: 0x04, BITS_75: 0x05, BITS_128_MSB: 0x06, BITS_128_LSB: 0x07, BITS_160_MSB: 0x08, BITS_160_LSB: 0x09, BITS_200: 0x0A, BITS_264: 0x0B, BITS_328: 0x0C,)	Format type of CHUID data.
piv_guid_fallback = bool	When set to TRUE and PIVI-I card is detected the FASC-N is replaced with GUID For exemplary card formatter configuration see "Card formatter configuration structure" on page 50.

EM Prox formatters

EM Prox card data is 40bit long and contains version/customer code (8 bits) and card data (32 bits).

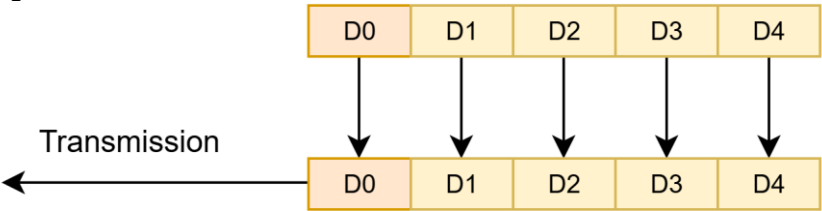


-  version / customer ID
-  card data

The following data formats are supported for EM Prox card data:

ASK_RAW_40BIT (default)

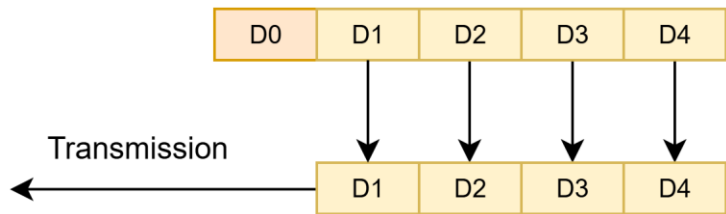
All 5 bytes of the card data are reported, starting with the least significant byte (LSByte), and within each byte, the most significant bit is transmitted first.



ASK_WIEGAND_26BIT

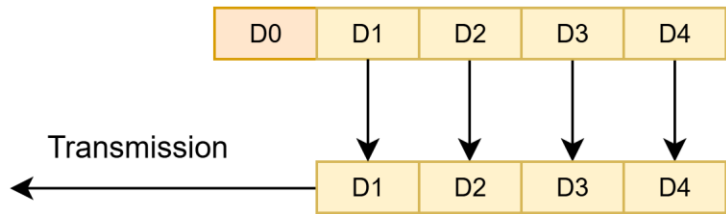
The data is constructed from the three most significant bytes of the card data. An even parity bit is calculated from the 12 least significant bits of the data and inserted before the least significant bit. An odd parity bit is calculated from the 12

most significant bits of the data and inserted after the most significant bit. The data is reported starting with the least significant byte (LSByte), with the most significant bit transmitted first within each byte.



ASK_WIEGAND_32BIT

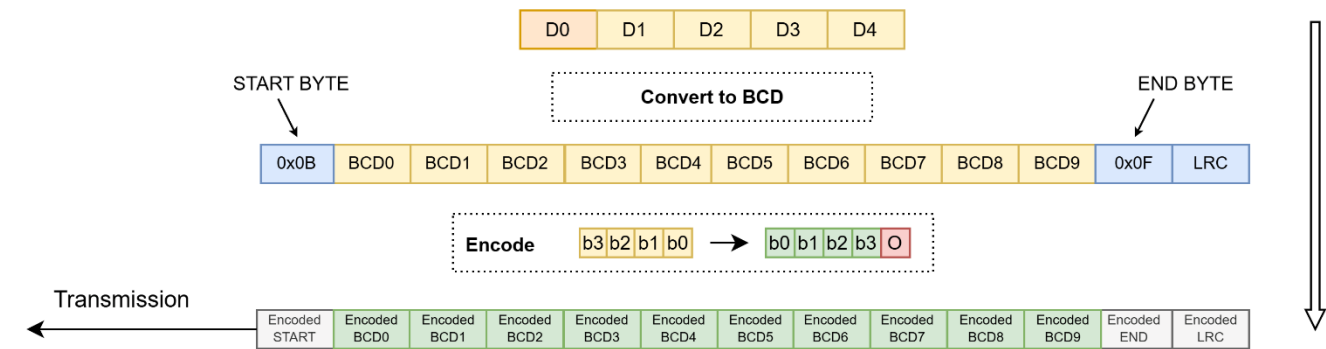
The four most significant bytes of the card data are reported in least significant byte (LSByte) first order, with the most significant bit (MSB) transmitted first within each byte.



ASK_ABA_10_DIGIT_BCD

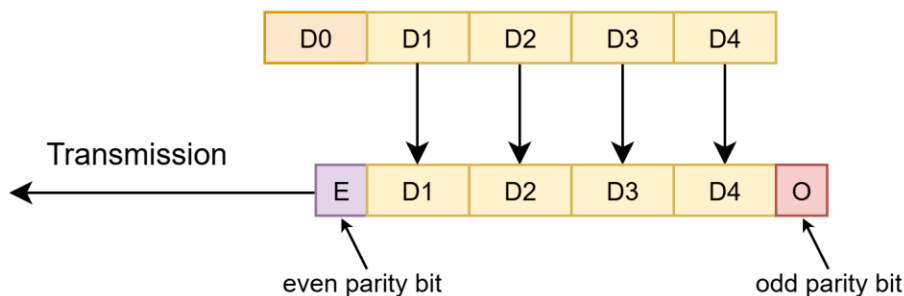
The data is constructed from the four least significant bytes of the card data. These bytes are first serialized into BCD format (Binary Coded Decimal), converting each nibble (4 bits) into a separate BCD digit. A START sentinel (BCD 0x0B) is inserted at the beginning, and an END sentinel (BCD 0x0F) is appended at the end. An LRC (Longitudinal Redundancy Check) byte is then computed as the bitwise XOR of all preceding BCD bytes, including START and END sentinels. The LRC byte is also serialized into BCD and appended to the end. Each BCD digit is then encoded into a 5-bit symbol as follows:

- The 4-bit BCD value is first bit-reversed (from MSb to LSb).
 - An odd parity bit is computed over the 4 reversed bits and added as the fifth bit, ensuring that the total number of 1s in the 5-bit symbol is odd.
- The resulting sequence of 5-bit symbols (including START, data, END, and LRC) forms the final encoded ABA data. The symbols are transmitted in order, most significant bit first within each 5-bit symbol.



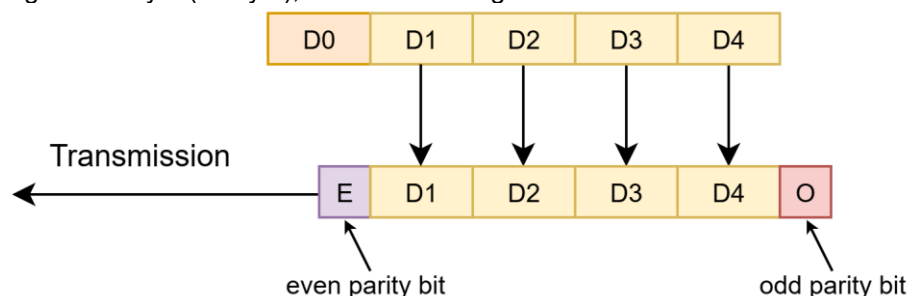
ASK_WIEGAND_42BIT

The data is constructed from all five bytes of the card data. An even parity bit is calculated from the 20 least significant bits of the data and inserted before the least significant bit. An odd parity bit is calculated from the 20 most significant bits of the data and inserted after the most significant bit. The data is reported starting with the least significant byte (LSByte), with the most significant bit transmitted first within each byte.



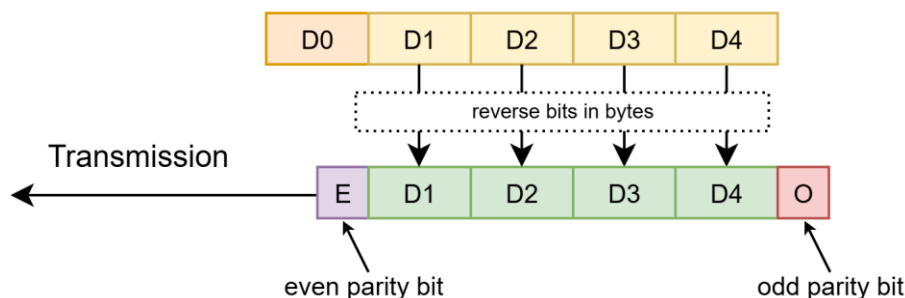
ASK_WIEGAND_34BIT

The data is constructed from the four most significant bytes of the card data. An even parity bit is calculated from the 16 least significant bits of the data and inserted before the least significant bit. An odd parity bit is calculated from the 16 most significant bits of the data and inserted after the most significant bit. The data is reported starting with the least significant byte (LSByte), with the most significant bit transmitted first within each byte.



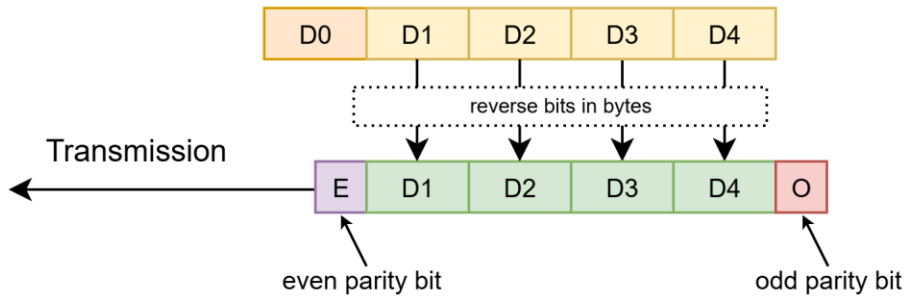
ASK_WIEGAND_42BIT_MLF

The data is constructed from all five bytes of the card data. Each byte has its bit order reversed before further processing. An even parity bit is calculated from the 20 least significant bits of the resulting 40-bit data and inserted before the least significant bit. An odd parity bit is calculated from the 20 most significant bits of the resulting data and inserted after the most significant bit. The data is reported starting with the least significant byte (LSByte), with the most significant bit transmitted first within each byte.



ASK_WIEGAND_34BIT_MLF

The data is constructed from all five bytes of the card data. Each byte has its bit order reversed before further processing. An even parity bit is calculated from the 16 least significant bits of the resulting 40-bit data and inserted before the least significant bit. An odd parity bit is calculated from the 16 most significant bits of the resulting data and inserted after the most significant bit. The data is reported starting with the least significant byte (LSByte), with the most significant bit transmitted first within each byte.



Example configuration for card data formatter

Presented example assumes:

- format type: BITS_160_LSB
- piv guid fallback flag: true

For this configuration the card formatter CBOR is defined as follows:

CBOR: A2613000613155A26130810A6133A26130F4613147A26130096131F5

Decoded:

- card_edge_type: 0x00 - CARD_EDGE_HF
- formatter_config: A26130810A6133A26130F4613147A26130096131F5
 - card_types: [0x08] - PIV (0x08)
 - custom_data_format:
 - discard_flag: false
 - custom_format: A26130096131F5
 - piv_chuid_format: BITS_160_LSB (0x09)
 - piv_guid_fallback: true

Example Outputs for different formats

Assuming PIV CHUID as presented in CHUID data below, the output data will take a form:

CHUID data

FASC-N (200b) D4 32 48 58 21 0C 2D 31 71 B5 25 A1 68 5A 08 C9 2A DE 0A 61 84 32 48 43 E2
Expiration Date (64b): 20110411 32 30 31 31 30 34 31 31
GUID (128b): b8c8bd84-20bb-4026-8133-613153c02ec1 B8 C8 BD 84 20 BB 40 26 81 33 61 31 53 C0 2E C1

BITS_328

328 Bit Output

FASC-N (200bit): D4324858210C2D3171B525A1685A08C92ADE0A6184324843E2	GUID (128bit): B8C8BD8420BB40268133613153C02EC1
Output (HEX) : D4 32 48 58 21 0C 2D 31 71 B5 25 A1 68 5A 08 C9 2A DE 0A 61 84 32 48 43 E2 B8 C8 BD 84 20 BB 40 26 81 33 61 31 53 C0 2E C1	

BITS_264

264 Bit Output

FASC-N (200bit): D4324858210C2D3171B525A1685A08C92ADE0A6184324843E2	Expiration Date (64bit): 3230313130343131
Output (HEX) : D4 32 48 58 21 0C 2D 31 71 B5 25 A1 68 5A 08 C9 2A DE 0A 61 84 32 48 43 E2 32 30 31 31 30 34 31 31	

BITS_200

200 Bit Output

SS		Agency Code								FS		System Code																
1101	0	1	0000	0	3	1100	1	0010	0	1000	0	1011	0	0000	1	0000	1	0000	1	1000	0							
FS		Credential Code												FS		CS		FS										
1011	0	9	1001	1	8	0001	0	7	1110	0	6	0110	1	5	1010	1	4	0010	0	1	1011	0						
ICI		FS		PI																								
1	1000	0	1011	0	1	1000	0	2	0100	0	3	1100	1	4	0010	0	5	1010	1	6	0110	1	7	1110	0	8	0001	0
				OC		OI								POA		ES		LRC										
9	1001	1	0	0000	1	1	1000	0	1	1000	0	3	1100	1	4	0010	0	1	1000	0	1	1000	0	1111	1	8	0001	0

Output (HEX) :

D4 32 48 58 21 0C 2D 31 71 B5 25 A1 68 5A 08 C9 2A DE 0A 61 84 32 48 43 E2

BITS_160_LSB

160 Bit Output LSB

SS	Agency Code				FS	System Code			
1101	1000	1100	0010	1000	1011	0000	0000	0000	1000
FS	Credential Code						FS	CS	FS
1011	9001	8001	7110	6010	51010	40010	1011	1000	1011
ICI	FS	PI							
1000	1011	1000	0100	3100	40010	51010	60110	71110	80001
		OC	OI				POA	ES	LRC
9001	0000	1000	1000	3100	40010	1000	1000	1111	8001

Output (HEX) :

D8 C2 8B 00 08 B9 1E 6A 2B 8B 8B 84 C2 A6 E1 90 88 C2 88 F1

BITS_160_MSB

160 Bit Output MSB

SS	Agency Code				FS	System Code			
1011	1 0001	3 0011	4 0100	1 0001	1101	0 0000	0 0000	0 0000	1 0001
FS	Credential Code						FS	CS	FS
1101	9 1001	8 1000	7 0111	6 0110	5 0101	4 0100	1101	1 0001	1101
ICI	FS	PI							
1 0001	1101	1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111	8 1000
		OC	OI				POA	ES	LRC
9 1001	0 0000	1 0001	1 0001	3 0011	4 0100	1 0001	1 0001	1111	8 1000

Output (HEX) :

B1 34 1D 00 01 D9 87 65 4D 1D 1D 12 34 56 78 90 11 34 11 F8

BITS_128_LSB

128 Bit Output LSB

Agency Code				System Code				Credential Number						CS	ICI
1 1000	3 1100	4 0010	1 1000	0 0000	0 0000	0 0000	1 1000	9 1001	8 0001	7 1110	6 0110	5 1010	4 0010	1 1000	1 1000
PI										OC	OI			POA	
1 1000	2 0100	3 1100	4 0010	5 1010	6 0110	7 1110	8 0001	9 1001	0 0000	1 1000	1 1000	3 1100	4 0010	1 1000	1 1000

HEX:

8C 28 00 08 91 E6 A2 88 84 C2 A6 E1 90 88 C2 88

BITS_128_MSB

128 Bit Output MSB

Agency Code				System Code				Credential Number						CS	ICI
1 0001	3 0011	4 0100	1 0001	0 0000	0 0000	0 0000	1 0001	9 1001	8 1000	7 0111	6 0110	5 0101	4 0100	1 0001	1 0001
PI										OC	OI			POA	
1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111	8 1000	9 1001	0 0000	1 0001	1 0001	3 0011	4 0100	1 0001	1 0001

Output (HEX) :

13 41 00 01 98 76 54 11 12 34 56 78 90 11 34 11

BITS_75

75 Bit Output

EP 1	AC: 0x053D (LSB 14) 00010100111101	SC: 0x01 (LSB 14) 000000000000001	CN: 0x0F1206 (LSB 20) 11110001001000000110	Exp. Date: 0x0132DCB4 (LSB 25) 1001100101101110001001011	OP 1
---------	--	---	--	--	---------

padding (5 bits)

0	C	5	3	D	0	0	0	7	C	4	8	1	A	6	5
0000	1100	0101	0011	1101	0000	0000	0000	0111	1100	0100	1000	0001	1010	0110	0101
B	8	9	7												
1011	1000	1001	0111												

HEX:

0C 53 D0 00 7C 48 1A 65 B8 97

BITS_64_LSB

64 Bit Output LSB

Agency Code				System Code				Credential Number						CS	ICI
1	3	4	1	0	0	0	1	9	8	7	6	5	4	1	1
1000	1100	0010	1000	0000	0000	0000	1000	1001	0001	1110	0110	1010	0010	1000	1000

Output (HEX) :

8C 28 00 08 91 E6 A2 88

BITS_64_MSB

64 Bit Output MSB

Agency Code				System Code				Credential Number						CS	ICI
1	3	4	1	0	0	0	1	9	8	7	6	5	4	1	1
0001	0011	0100	0001	0000	0000	0000	0001	1001	1000	0111	0110	0101	0100	0001	0001

Output (HEX) :

13 41 00 01 98 76 54 11

BITS_58

58 Bit Output

Agency Code				CS	SC[3]	Credential Number						SC[0:2]			ICI
1	3	4	1	1	1	9	8	7	6	5	4	0	0	0	1
0001	0011	0100	0001	1000	0001	1001	1000	0111	0110	0101	0100	0000	0000	0000	0001

OP
0

Value (20b):
dec: 134111
bin: 0010 0000 1011 1101 1111

Value (32b):
dec: 987654000
bin: 0011 1010 1101 1110 0110 0111 0111 0000

Value (4b):
dec: 1
bin: 0001

EP
1

padding (6 bits)

0	0	4	1	7	B	E	7	5	B	C	C	E	E	0	3
0000	0000	0100	0001	0111	1011	1110	0111	0101	1011	1100	1100	1110	1110	0000	0011

HEX:

00 41 7B E7 5B CC EE 03

BITS_40_LSB

40 Bit Output LSB

System Code				Credential Number					
0	0	0	1	9	8	7	6	4	5
0000	0000	0000	1000	1001	0001	1110	0110	0010	1010

Output (HEX) :

00 08 91 E6 A2

BITS_40_MSB

40 Bit Output MSB

System Code				Credential Number					
0	0	0	1	9	8	7	6	5	4
0000	0000	0000	0001	1001	1000	0111	0110	0101	0100

Output (HEX) :

00 01 98 76 54

PIV-I Guid fallback (only GUID)

128 Bit PIV-I GUID Fallback

GUID (128bit):
B8C8BD8420BB40268133613153C02EC1

Output (HEX) :
B8 C8 BD 84 20 BB 40 26 81 33 61 31 53 C0 2E C1

PIV-I Guid fallback with expiration date

264 Bit Output (PIV-I GUID Fallback)

GUID (128bit): B8C8BD8420BB40268133613153C02EC1	Expiration Date (64bit): 3230313130343131
--	--

Output (HEX) :
B8 C8 BD 84 20 BB 40 26 81 33 61 31 53 C0 2E C1 32 30 31 31 30 34 31 31

CSN formatters

The reader allows you to format the card CSN. The different formats available for UART and Wiegand interfaces are described in the following sections.

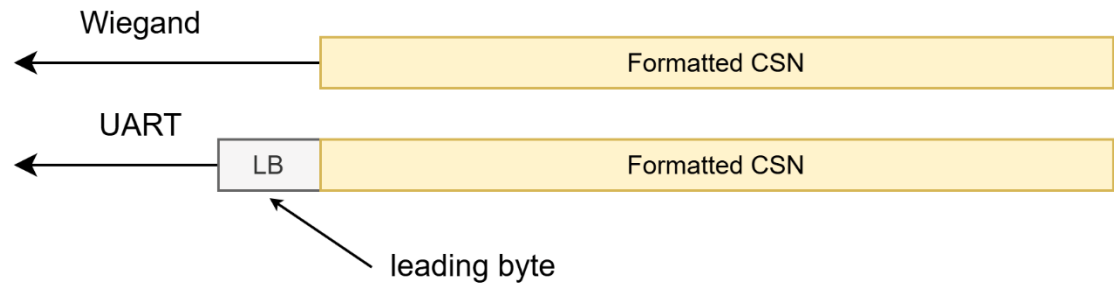
UART	Wiegand
<pre>uart_csn_enum = (UNFORMATTED: 0x00, BITS_32_MSB: 0x01, BITS_32_LSB: 0x02, BITS_56_MSB: 0x03, BITS_56_LSB: 0x04, BITS_64_MSB: 0x05, BITS_64_LSB: 0x06)</pre>	<pre>wiegand_csn_enum = (UNFORMATTED: 0x00, BITS_32_MSB: 0x01, BITS_32_LSB: 0x02, PARITY_FACILITY_16_BITS_MSB: 0x03, PARITY_32_BIT_MSB: 0x04, BITS_32_CRC: 0x05, BITS_37_LSB: 0x06, BITS_56_PHILLIPS: 0x07, BITS_64_LSB: 0x08, BITS_26_MSB: 0x09, BITS_56_HIRSH: 0x0A)</pre>

UART CSN formats

Depending on card technology or protocol the following data is used as a CSN source:

- ISO14443-3 A card: UID (Unique Identifier) received from card during Type A anti-collision process.
- ISO14443-3 B card: PUPI (Pseudo-Unique PICC Identifier) received from card during Type B anticollision process.
- ISO15693-3 card: UID received from card during anti-collision process (Inventory command).
- Picopass card: Anti-collision Serial Number (used by Identify and Select commands).

Formatting applies only to ISO14443 cards, any other credential type is output as unformatted CSN (reversed or not, depending on configuration). With the default settings, an unformatted CSN value is returned. It is possible to set formatting independently for Wiegand and UART interfaces. Due to the fact that the number of bits for formatted CSN may not be byte-aligned, for UART output, the CSN bytes are preceded by a padding byte, which specifies the number of zero bits added to the end of the formatted CSN.



The following CSN formats are supported for ISO14443 cards:

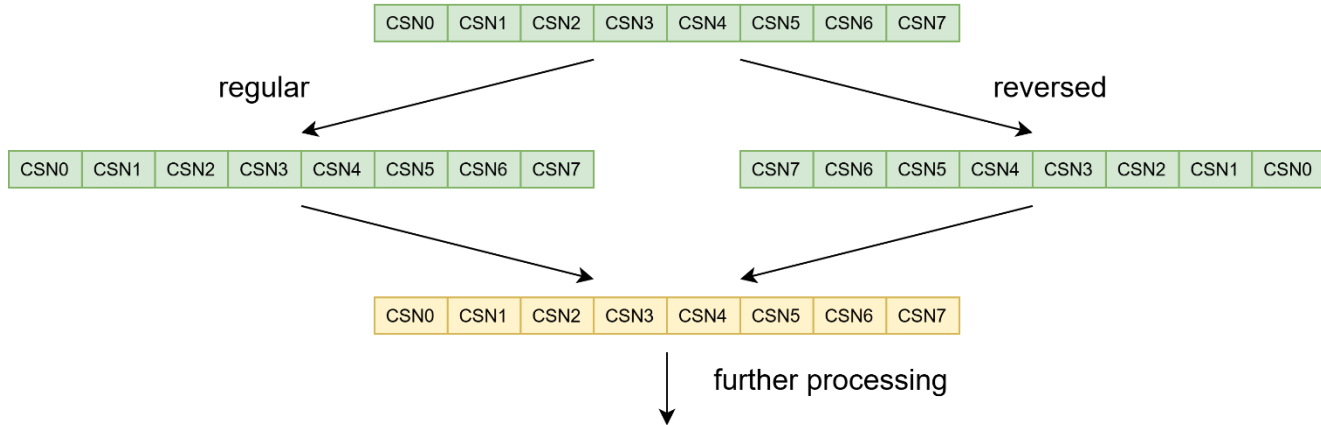
- UNFORMATTED: 0x00,
- BITS_32_MSB: 0x01,
- BITS_32_LSB: 0x02,
- PARITY_FACILITY_16_BITS_MSB: 0x03,
- PARITY_32_BIT_MSB: 0x04,
- BITS_32_CRC: 0x05,
- BITS_37_LSB: 0x06,
- BITS_56_PHILLIPS: 0x07,
- BITS_64_LSB: 0x08,
- BITS_26_MSB: 0x09,

- BITS_56_HIRSH: 0x0A

Format details are described in next sections.

CSN formats for ISO14443 cards

CSN is represented as a byte array. The byte order corresponds to the byte order received during card reading. The reverse flag available in the formatter settings allows changing the CSN byte order before further processing.

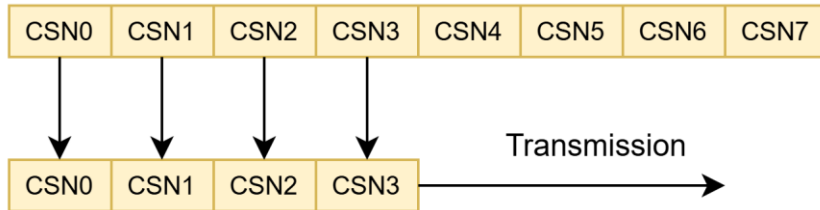


UNFORMATTED

The entire CSN is reported with no additional formatting.

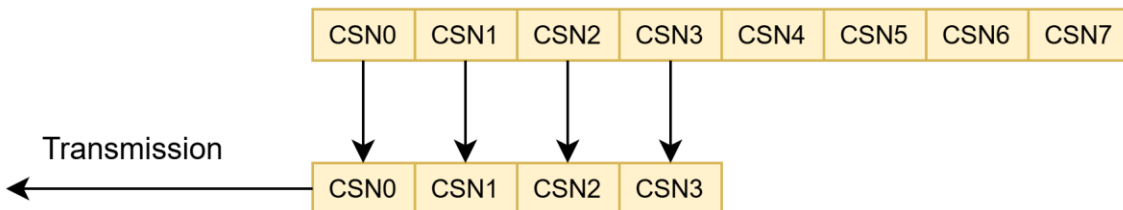
32_BITS_MSB

The 4 least significant bytes of the CSN are reported, MSByte first, with no additional formatting. Each byte is transmitted MSBit first.



32_BITS_LSB

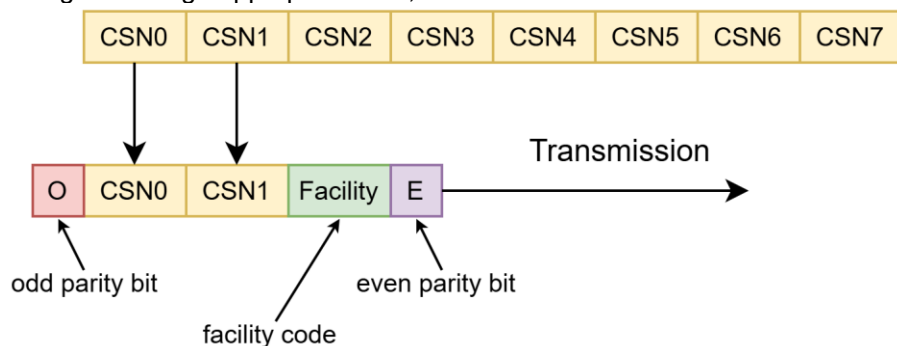
The 4 least significant bytes of the CSN are reported, LSByte first, with no additional formatting. Each byte is transmitted MSBit first.



PARITY_FACILITY_16_BITS_MSB

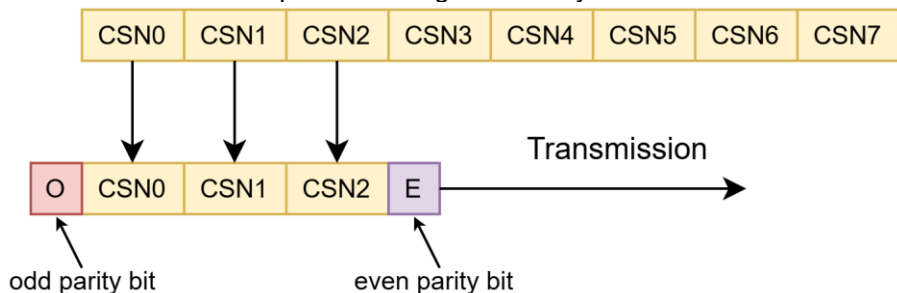
Data is constructed from 2 least significant bytes of the CSN and 1-byte facility code. An even parity bit is calculated from 12 most significant bits of data and inserted after the MSBit. An odd parity bit is calculated from 12 least significant bits of

data and inserted before LSBit. Data is reported starting from MSByte and MSBit first. Facility code should be previously configured through appropriate OID, otherwise the default value will be taken (see: Card output formatter facility code).



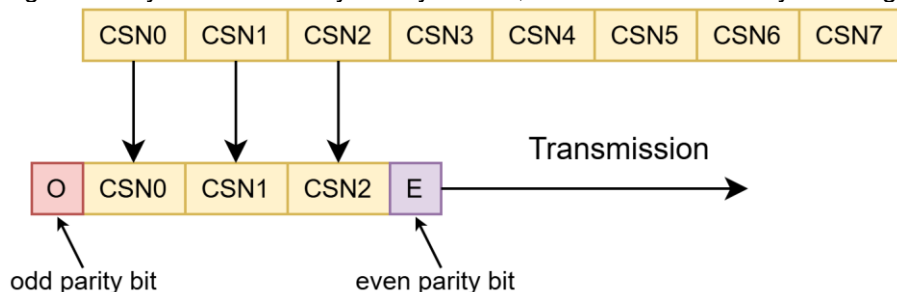
PARITY_32_BIT_MSB

Data is constructed from 4 least significant bytes of the CSN. An even parity bit is calculated from 16 most significant bits of data and inserted after the MSBit. An odd parity bit is calculated from 16 least significant bits of data and inserted before LSBit. Data is reported starting from MSByte and MSBit first.



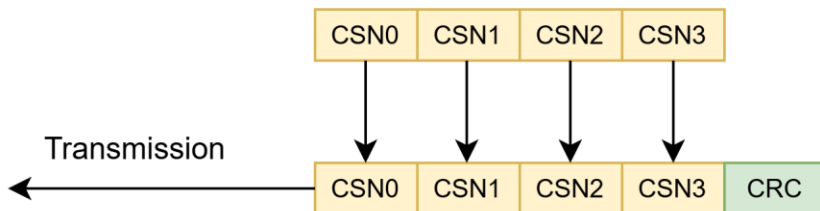
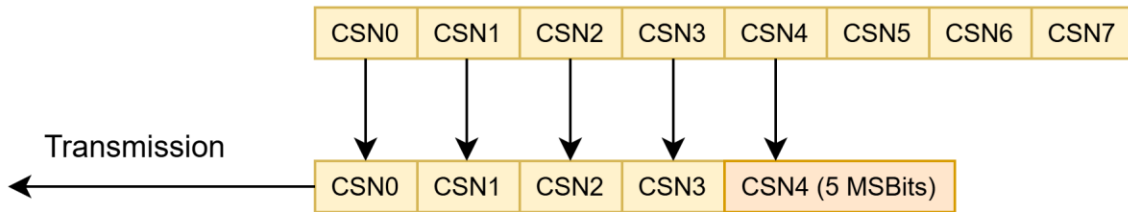
BITS_32_CRC

Data is constructed from 4 least significant bytes of the CSN, MSByte first, with no additional formatting. The least significant byte is followed by a 1-byte CRC, which is calculated by XOR'ing 4 bytes taken from the CSN.



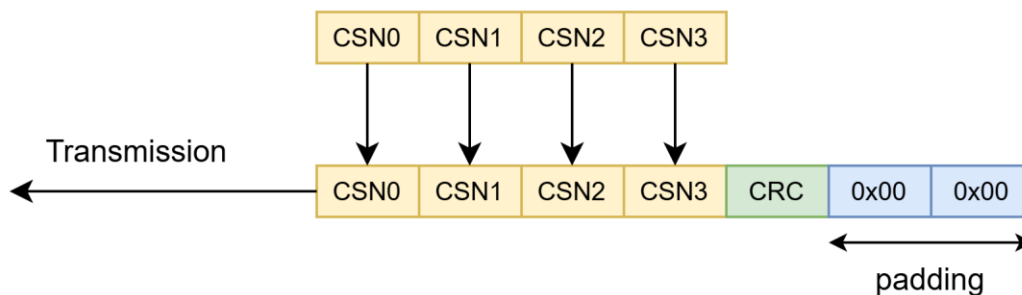
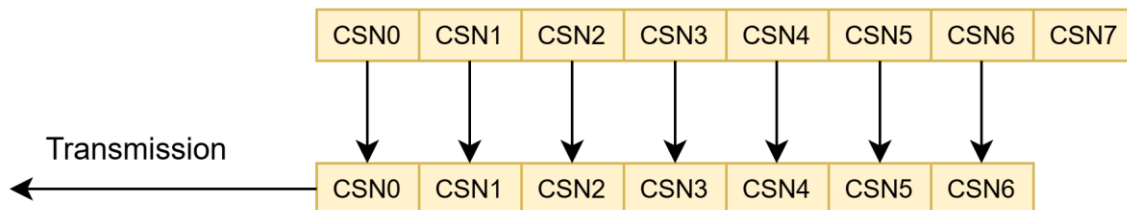
BITS_37_LSB

Data is constructed from 4 least significant bytes of the CSN, LSByte first, with no additional formatting. The fourth byte of CSN is followed by 5 additional bits to bring the message length up to 37 bits. In case of CSN's longer than 4 bytes, the final 5 bits of message are the 5 MSBits of the 5th CSN byte. Otherwise the final 5 bits of the message are 5 MSBits of a CRC calculated by XOR'ing the 4 bytes taken from CSN.



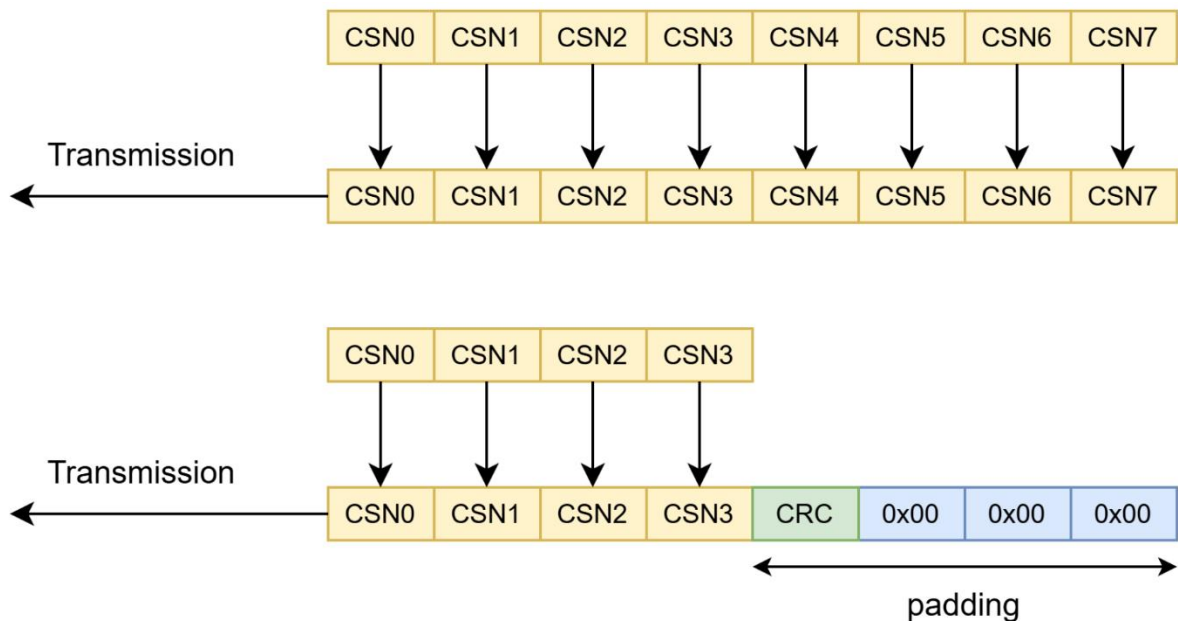
BITS_56_PHILLIPS

Data is constructed from 7 least significant bytes of the CSN. For credentials with a CSN shorter than 7 bytes, CSN data is padded with a CRC and 0's to bring the total number of bits up to 56. CSN is reported LSByte first, followed by CRC, then padding. CRC is calculated by XOR'ing bytes taken from CSN.



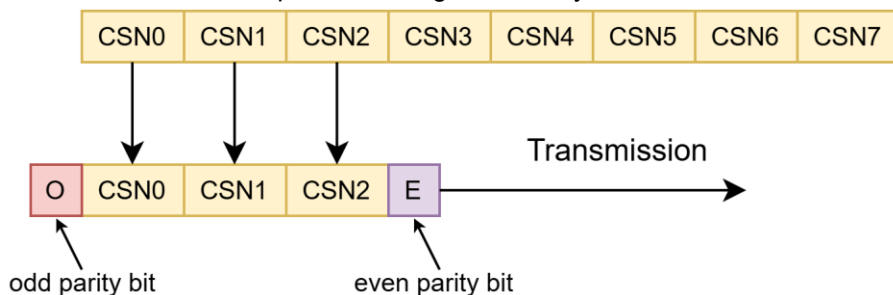
BITS_64_LSB

The 8 least significant bytes of the CSN are reported (LSByte first). For credentials with a CSN shorter than 8 bytes, data is padded with 0's to bring the total number of bits to 64. For credentials with 4 bytes CSN, the reported CSN is followed by CRC and padding bytes. CRC is calculated by XOR'ing bytes taken from CSN.



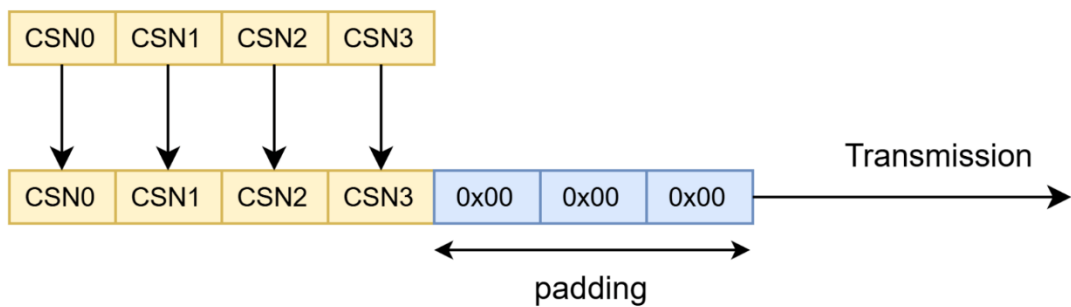
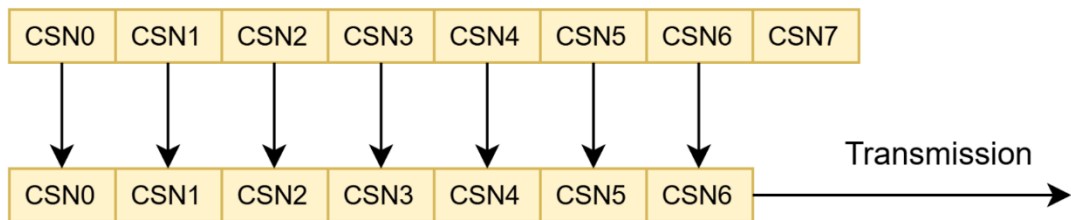
BITS_26_MSB

Data is constructed from 3 least significant bytes of the CSN. An even parity bit is calculated from 12 most significant bits of data and inserted after the MSBit. An odd parity bit is calculated from 12 least significant bits of data and inserted before LSBit. Data is reported starting from MSByte and MSBit first.



56_BITS_HIRSH

The 7 least significant bytes of the CSN are reported (MSByte first). For credentials with a CSN shorter than 7 bytes, the reported data is padded with zeros in the most significant bytes, aligning the UID to the least significant bytes.



Supported commands

When the reader is in Autonomous mode, it supports only two shell commands which can be sent over UART:

- `QUIT_AUTONOMOUS` is a request to leave Autonomous mode. When sent, the reader responds with the string "ACKNOWLEDGED" and switches to External Reader API mode.
- `WHO_ARE_YOU` verifies if the reader is in Autonomous mode. When sent, the reader will respond with the following response format: `OMNIKEY_SE_READER KEY:VAL`.

Currently supported `WHO_ARE_YOU` response keys:

- `TYPE`: returns the reader type; for OMNIKEY READER CORE the type is CORE.
- `SN`: returns the reader serial number, sent in format `A0...An`; currently, the serial number is defined as a 10 character string.
- `UUID`: returns the related device UUID, 36 bytes total sent in format `B0...Bn`.
- `NAME` - returns "Reader Friendly Name" (for details please refer to section: 14.3.3.2). Note: value is returned only if this property was set.

WHO_ARE_YOU response examples

- `OMNIKEY_SE_READER TYPE:CORE SN:12345ZFGCG UUID:5d0b30a9-f638-5c9b-8114- 226f320b901e`
- with Reader Friendly Name: `OMNIKEY_SE_READER TYPE:CORE SN:12345ZFGCG UUID:5d0b30a9- f638-5c9b-8114-226f320b901e NAME:FRIENDLY_NAME`

The reader must first be woken up from power saving mode by the UART peripheral by sending any character End-Of-Line characters in commands responses are CRLF [0x0D, 0x0A].

Leaving Autonomous mode

To leave Autonomous mode, the `QUIT_AUTONOMOUS` command must be sent over UART. This switches the reader to External Reader API mode. After switching, it is possible to use External Reader API commands to switch immediately to another mode or permanently change (until the next change) the reader's default mode.

Card processing engine

Introduction

The Card Processing Engine is used to automatically read PACS and/or custom data and/or CSN from the credential in Autonomous mode. When configured, if a card is presented to the reader and the reader is in Autonomous mode, it will behave according to the configuration:

- Read the card type.
- Read out selected data if possible.
- Report data.
- Report invalid card error.

Autonomous readout flow

The entire card readout flow can be divided into the following stages:

- Interface polling
- Protocol polling
- Card data readout engine
- Card data formatting engine

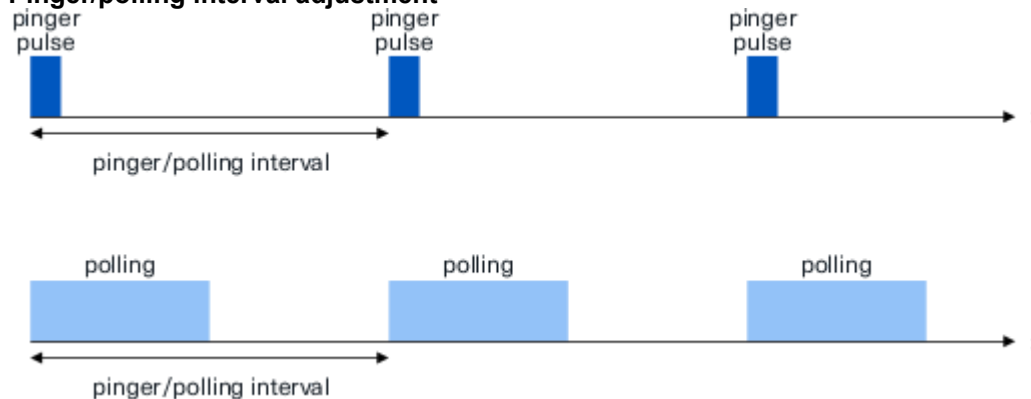
Interface polling

Depending on the configuration, the reader will poll at configured intervals.

Pinger / polling interval

It is possible to define the pinger/polling interval. The following diagrams show this approach. The reader will either ping or skip the pinger and poll based on the reader configuration.

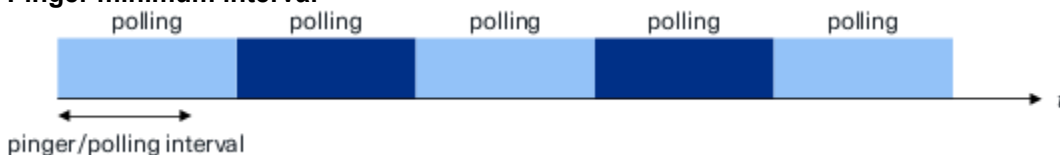
Pinger/polling interval adjustment



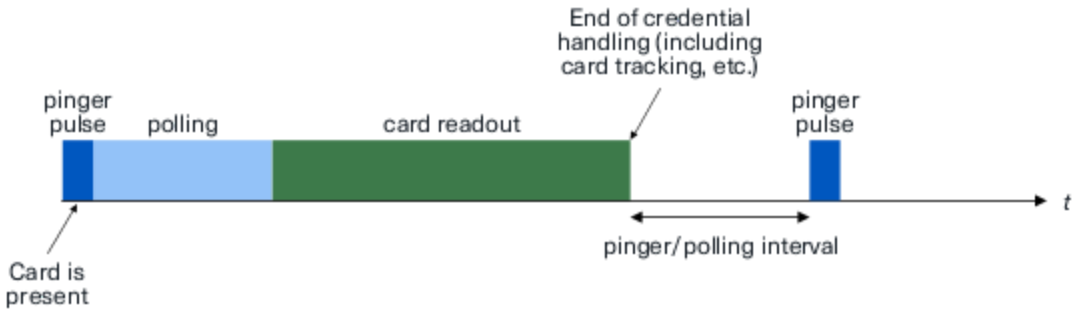
An interval timer counts from the start of n th pinger/polling event to the start of the $n+1$ event (it is *not* a delay between two consecutive events).

It is possible that polling can take more time than the configured interval. In this case, polls are triggered one after another without any intermediate delay.

Pinger minimum interval



If a card is detected and readout starts, the timer is reset after the readout is finished, as shown below. Pinger mode is shown, but the same behavior applies to polling mode.
Pinger interval after card processing



Protocols configuration

The enabling/disabling and order of HF protocols polling is set using the reader configuration. The following CBOR encoded data structure is used:

Structure	Description
hf_protocol_order = [1*5 int]	Definition of HF protocol polling order configuration. An array of integers contains the protocols to be used in polling, in the correct order. If a particular protocol is not inserted, it will not be used during polling. A protocol should only be entered once, otherwise behavior will not be predictable. Example (CBOR encoded bstr): h'8400020301' - order: ECP, ISO14443B, 15694, ISO14443A, iCLASS turned off
hf_protocol_sel = (HF_PROTO_ECP: 0x00, HF_PROTO_ISO14443A: 0x01, HF_PROTO_ISO14443B: 0x02, HF_PROTO_ISO15693: 0x03, HF_PROTO_iClass: 0x04)	Protocol values to be used for configuration of the HF polling order.

Card Data Readout Engine

The Card Data Readout Engine is the main engine which reads the data from credentials based on the provided configuration. It uses a list of OIDs which define the behaviour of the reader when a particular card is detected. Each OID stores one procedure in the Card Data Readout Engine and is used to process a single card in a specific way. It is possible to process multiple data containers within one card, therefore each of the entries includes a set of actions (described in detail in the following sections).

- Where to read from? A credential type, e.g. MIFARE Classic.
- How to authenticate? Which keys shall be used, where will the engine authenticate to.
- What to read? PACS, custom data or CSN. If custom data is selected, it contains the definition of the memory space to be read (with respect to a particular card type). It also contains the size of the data to be read.

Card Data Readout Engine interface

Card Data Readout for the interface is handled as follows:

1. Identify the card type.
2. Read the procedure (OID) at the lowest position (highest priority).
3. Verify if the card type matches the detected card type:
 - a. If yes, go to the next step.
 - b. If not, and this is not the last procedure, read the next procedure and repeat step 3.
 - c. If not, and this is last procedure, send the card type + CSN to the data formatter engine and finish the readout flow.
4. Try to authenticate to the card (if applicable) and read the first data container set in the procedure:
 - a. If successful, go to the next step.
 - b. If failed, and this is last procedure, send the card type + CSN to the data formatter engine and finish the readout flow.
 - c. Otherwise, read the next procedure and repeat step 3.
5. Store data, and verify if there is any other action within the action:
 - a. If yes, go to step 6.
 - b. If not, pass the read data, card type, and CSN to the data formatter engine and finish the readout flow.
6. Try to authenticate to the card (if applicable) and read the next data container set in the OID:
 - a. If successful, go to step 5.
 - b. If failed, send the card type + CSN to the data formatter engine and finish the readout flow.

Card Data Formatting Engine

When data is read from a card, it is immediately pushed to the Card Data Formatting Engine. The Card Data Formatting Engine is responsible for preparing the data in a preconfigured format to send to the host.

If a card is detected but the Card Data Readout Engine is not configured to read this type of card, or could not authenticate it due to incorrect keys, the basic card data is still pushed to the Card Data Formatting Engine. In this case, it can be configured to either send basic data (like CSN + card type) or to send the data only after successful authentication/authorization. If so, only known card data is sent. If an unknown card is used, CPE will remain silent.

Features

The following features are included in the Card Processing Engine:

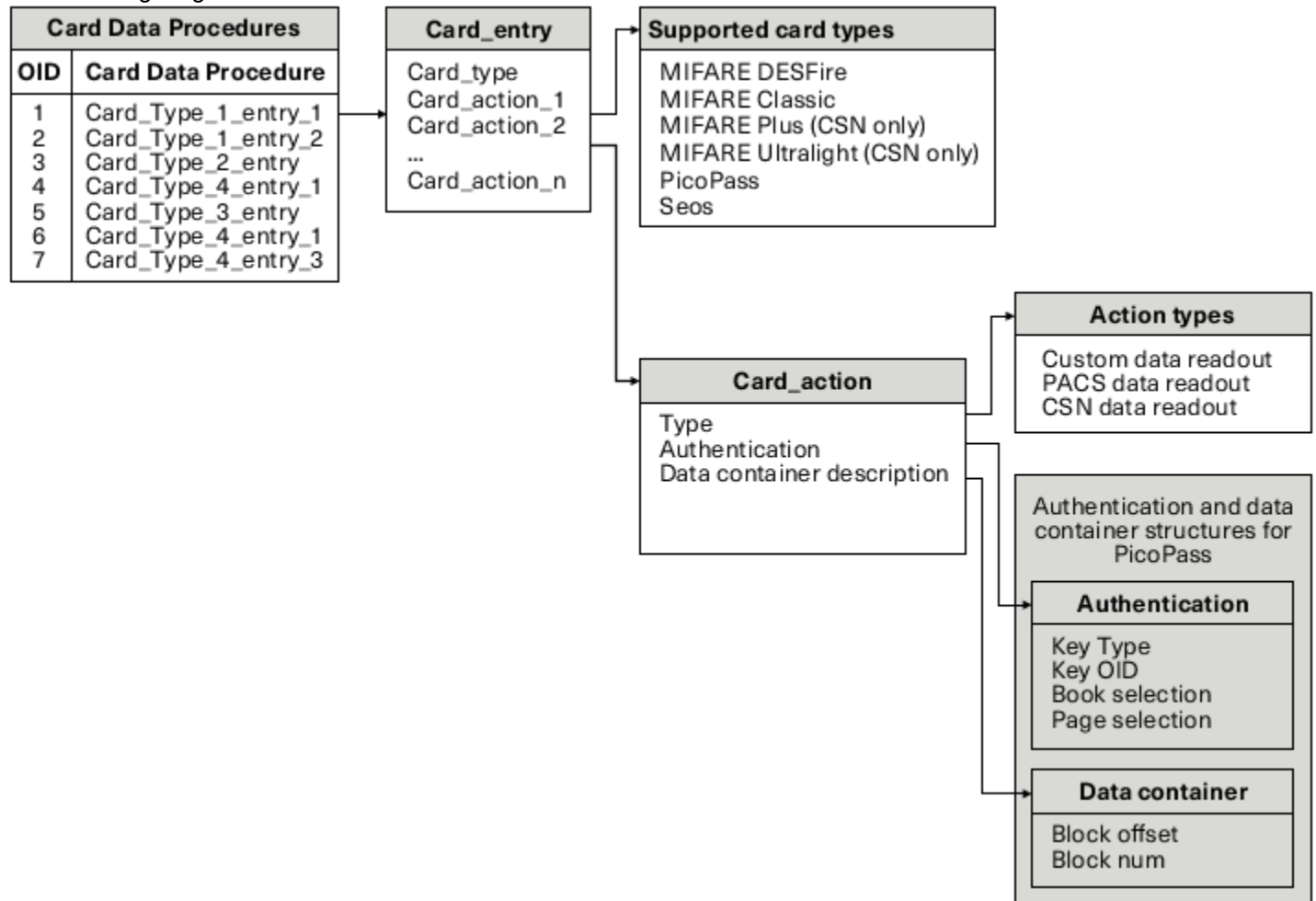
- Enhanced-Polling
 - When enabled, the reader will try to poll over other protocols/modulations, even if the card has already responded to one of the protocols but could not be properly read. This allows operation with cards containing many different technologies (like 2xHF).
 - When disabled, the reader will try to poll the protocols in the specified order. As soon as the card responds, the reader stops polling and proceeds to the Card Data Readout Engine. If the card could not be read (or the Card Data Readout Engine does not include any related procedure) the entire process is finished and the CSN + card type is passed to the Card Data Formatting Engine.
- Any card report
When enabled, basic card data (card type, CSN) are reported even when the card could not be read correctly (PACS, custom). Works best with the Enhanced-Polling feature enabled.
- Polling mode
When enabled, the reader will skip pingers and poll the without checking if any card is present. Delay before sending data
When a delay is set, the reader will wait between detecting a credential and sending its data.

Card data procedures

A card data procedure is a set of data components which are used to properly process a card. A procedure contains:

- Card type (Seos, MIFARE Classic, PicoPass, DESFire, etc.).
- A set of actions. Each action contains:
 - Action type (Custom, PACS, CSN, Transact, Allegion).
 - Optional: For custom only, authentication info: algorithm, key OID, etc.
 - Optional: For custom only, data container.

The following diagram shows the Card Data Procedure structure.



Card Data Generic Procedure structure

A Card Data Procedure (single OID) is encoded as a CBOR structure, as shown in the following table:

Structure	Description
<pre>card_data_procedure = { "0" : card_type, "1" : [bstr] }</pre>	Definition of a single Card Data Procedure "1" : list of byte strings representing actions
card_type = see "Card types" on page 92.	Supported card types

Card Action structure

A Card Action structure is encoded as a CBOR structure and placed inside a `card_data` procedure as a byte string.

Structure	Description
<pre> action = { "0" : action_type, ? "1" : bstr ; authentication action data, ? "2" : bstr ; data read action data "3" : ? int ; action user id "4" : ? bitwise_config ; bitwise config } </pre>	Definition of a single data action. "1" optional byte string of authentication action "2" optional byte string of data read action "1" and "2" can be set only if "action_type" == CUSTOM "3" optional - RFU "4" bitwise option is only meaningful if custom action includes data readout (otherwise it is ignored)
<pre> action_type= &action_type_enum action_type_enum= (ACTION_TYPE_PACS = 0x00, ACTION_TYPE_CUSTOM = 0x01, ACTION_TYPE_CSN = 0x02 ACTION_TYPE_TRANSACT = 0x0A, ACTION_TYPE_ALLEGION = 0x0B) </pre>	Supported action types
<pre> bitwise_config = { action_type_enum= ("0" : bit_offset, "1" : bit_length) } </pre>	Defines how the data shall be read bitwise. If not set, Reader will push data as it is without any bitwise trimming / offset.
<pre> bit_offset = 0..7991 </pre>	bit_offset defines the bit offset from which the data shall be printed (from the beginning of the data read within specific action). Max offset is defined as 999 (data limit) *8 bits, still reading very last bits is quite unlikely. The total size read from the card depends on the CPE configuration and card type. If configured offset exceeds total data size an error (CARD_READ_REJECTED) will be raised.
<pre> bit_length= 1..7992 </pre>	bit_length - defines how many bits shall be printed to host (starting from bit_offset) Bit length is defined as 999 (data limit) *8 bits The total size read from the card depends on the CPE configuration and card type. If configured offset exceeds total data size an error (CARD_READ_REJECTED) will be raised.

Authentication action data and data read action data need to be encoded using appropriate structures for the selected card type. Additionally, each action type can be used only with a supported card type, as shown in the following table.

Action type	Supported card types
PACS	iCLASS/PicoPass, MIFARE Classic 1K, MIFARE Classic 4K, MIFARE DESFire, Seos, HID Prox
CUSTOM	iCLASS/PicoPass, MIFARE Classic 1K, MIFARE Classic 4K, MIFARE DESFire, Seos
CSN	iCLASS/PicoPass, MIFARE Classic 1K, MIFARE Classic 4K, MIFARE Plus, MIFARE DESFire, MIFARE Ultralight, Seos
TRANSACT	Mifare DESFire
ALLEGION	Mifare DESFire

Card Data iCLASS/PicoPass specific structures

PicoPass authenticate structure

The PicoPass authentication operation contains two steps: PAGESELECT (book_sel, page_sel) and AUTHENTICATION (key_type, key_oid). You can skip both steps, as all parameters of the authentication structure are optional. If you want to perform PAGESELECT, both PAGESELECT parameters should be set. If you want to perform the AUTHENTICATE operation, both AUTHENTICATION parameters should be set. If you provide only one parameter for an operation (PAGESELECT or AUTHENTICATE) the operation will fail.

For cards where PAGESELECT is not supported (for example, PicoPass 2K) you should leave the PAGESELECT parameters empty.

Structure	Description
<pre>auth_picopass = { ? "0" = key_type, ? "1" = key_oid, ? "2" = book_sel, ? "3" = page_sel }</pre>	Structure defining data required for authentication to iCLASS or PicoPass. Possible cases: • book_sel and page_sel are set if you want to access unauthorized memory. In this case, the credential is set to allow unauthorized access to memory. Only the select command is triggered below. • key_type and key_oid are set when you want to authenticate to default memory (book 0 and page 0 for most of the cards). In this scenario, only the mutual auth command is triggered below; the select command is skipped. • All items must be set if you want to access authorized memory space. It can not be used for PicoPass 2K, as stated in the note above.
<pre>key_type = &key_type_enum key_type_enum = (KEY_TYPE_CREDIT = 0x00, KEY_TYPE_DEBIT = 0x01)</pre>	Definition of the key type
<pre>key_oid = bstr</pre>	Authentication key oid
<pre>book_sel = 0...255</pre>	Book selection
<pre>page_sel = 0...7</pre>	Page selection

PicoPass read/write data structure

This is an example of an iCLASS or PicoPass data read/write structure. It is encoded as a CBOR structure and placed inside an action as a byte string. The write operation is available only via ERA commands. Autonomous CPE actions accept only read operations. During a write operation (using an ERA RFID command) the block_num param is not used. The number of blocks is calculated based on the provided write operation payload. The payload must be divisible by 8, otherwise the operation will fail.

Structure	Description
data_picopass = { "0" = block_sel, ? "1" = block_num }	Structure defining what data will be read from a PicoPass card
block_sel = 0...255	Block selection
block_num = 1...64	Book selection

The reader does not validate card memory settings. You should carefully manage the block selection and block number to read/write, otherwise the operation result can not be defined (depends on the card type and configuration).

PicoPass preloaded default keys

For testing purposes, you can use preloaded default PicoPass keys.

Kc represents the Credit Key and Kd represents the Debit Key. For more details, please refer to the PicoPass documentation.

Key OID	Description
030023	Default PicoPass Book0 Page0 App2 (Kc)
030024	Default PicoPass Book0 Page1 App1 (Kd)
030025	Default PicoPass Book0 Page1 App2 (Kc)
030026	Default PicoPass Book0 Page2 App1 (Kd)
030027	Default PicoPass Book0 Page2 App2 (Kc)
030028	Default PicoPass Book0 Page3 App1 (Kd)
030029	Default PicoPass Book0 Page3 App2 (Kc)
03002A	Default PicoPass Book0 Page4 App1 (Kd)
03002B	Default PicoPass Book0 Page4 App2 (Kc)
03002C	Default PicoPass Book0 Page5 App1 (Kd)
03002D	Default PicoPass Book0 Page5 App2 (Kc)
03002E	Default PicoPass Book0 Page6 App1 (Kd)
03002F	Default PicoPass Book0 Page6 App2 (Kc)
030030	Default PicoPass Book0 Page7 App1 (Kd)
030031	Default PicoPass Book0 Page7 App2 (Kc)
030032	Transport PicoPass App1 (Kd)
030033	Transport PicoPass App2 (Kc)
030034	Default PicoPass Book1 Page0 App1 (Kd)

Card Data MIFARE specific structures

MIFARE authenticate structure

This is an example of a MIFARE authenticate structure. It is encoded as a CBOR structure and placed inside an action as a byte string.

Structure	Description
<pre>auth_mifare = { "0" = key_type, "1" = key_oid, "2" = block_index }</pre>	Structure defining data required for authentication to a MIFARE card
<pre>key_type = &key_type_enum key_type_enum = (MIFARE_KEY_A = 0x60, MIFARE_KEY_B = 0x61)</pre>	Definition of the key type
<pre>key_oid = bstr</pre>	Authentication key oid
<pre>block_index</pre>	Index of block to be authenticated

MIFARE read data structure

This is an example of a MIFARE data read structure. It is encoded as a CBOR structure and placed inside an action as a byte string.

Structure	Description
<pre>data_mifare = { "0" = block_index, "1" = ? : int ; number_of_blocks }</pre>	Structure defining what data will be read from a MIFARE card
<pre>block_index</pre>	Index of block to be read
<pre>number_of_blocks = 0...15</pre>	Number of blocks to be read. Unused for MIFARE Classic.

Card Data MIFARE DESFire specific structures

MIFARE DESFire authenticate structure

This is an example of a MIFARE DESFire authenticate structure. It is encoded as a CBOR structure and placed inside an action as a byte string.

Structure	Description
<pre>auth_desfire = { "0" : aid, "1" : key_no, "2" : key_oid, "3" : auth_conf }</pre>	Structure defining data required for authentication to DESFire card
<pre>aid = bstr</pre>	Application ID to be selected (bstr, len = 3)
<pre>key_no = 0...127</pre>	Number of the key to be used in authentication
<pre>key_oid = bstr</pre>	Authentication key oid
<pre>EV2 = 0x00, ISO = 0x01, NATIVE = 0x02</pre>	Authentication type constants
<pre>auth_conf= ev2_conf / iso_conf / native_conf</pre>	Defines the authentication config of particular authentication type
<pre>ev2_conf = { "0" : EV2, ? "1" : ev2_div_algo, ? "2" : div_data }</pre>	Defines the ev2 authentication config If div_data is not set, default data is used (based on the CSN and with respect to selected algo).
<pre>iso_conf = { "0" : ISO, "1" : auth_algo, ? "2" : iso_div_algo, ? "3" : div_data, ? "4" : iso_div_padding }</pre>	Defines the iso authentication config • If iso_div_algo is not set, then the reader sets it automatically to NONE. • If div_data is not set, the reader uses default div_data which is created based on the key number and card UID. • If iso_div_padding is not set, then CMAC_32B_PADDING is set. If iso_div_algo is not set, div_data is not required / will be skipped. Depending on the selected iso_div_algo, different div_data is required. For 3K3DES, 24 bytes of data is required. For other, 16 bytes are required.
<pre>native_conf = { "0" : NATIVE, ? "1" : native_div_algo, ? "2" : div_data }</pre>	Defines the native authentication config • If native_div_algo is not set, then the reader sets it automatically to NONE. • If div_data is not set, the reader uses default div_data which is created based on the key number and card UID. If native_div_algo is not set, div_data is not required / will be skipped. Depending on the selected native_div_algo, different div_data is required. For 3K3DES, 24 bytes of data is required. For other, 16 bytes are required.
<pre>div_data = bstr</pre>	IV for diversification algo
<pre>ev2_div_algo= &ev2_div_algo_enum ev2_div_algo_enum = (NXP_AV1_1KTDES : 0x01, NXP_AV1_2KTDES : 0x02, NXP_AV1_3KTDES : 0x03, NXP_AV1_AES128 : 0x04, CMAC_AES128 : 0x05,</pre>	EV2 diversification algorithms

SmartMAX : 0x06)	
native_div_algo= &native_div_algo_enum native_div_algo_enum = (NONE : 0x00, 1K3DES : 0x01, 2K3DES : 0x02, 3K3DES : 0x03, CMAC_AES128 : 0x04)	Native diversification algorithms. If NONE is selected, master key is used for authentication (without any diversification).
auth_algo= &auth_algo_enum auth_algo_enum = (2K3DES : 0x02, 3K3DES : 0x04, AES : 0x09)	Algorithm to be used in authentication
iso_div_algo= &iso_div_algo_enum iso_div_algo_enum = (NONE : 0x00, 1K3DES : 0x01, 2K3DES : 0x02, 3K3DES : 0x03, AES128 : 0x04, CMAC_AES128 : 0x05)	ISO diversification algorithms. If NONE is selected, master key is used for authentication (without any diversification).
iso_div_padding= &iso_div_padding_enum iso_div_padding_enum = (NONE : 0x00, CMAC_16B : 0x01, CMAC_32B : 0x02)	Algorithm to be used in authentication.

MIFARE DESFire read data structure

This is an example of a MIFARE DESFire data read structure. It is encoded as a CBOR structure and placed inside an action as a byte string.

Structure	Description
<pre>data_desfire = { "0" = file_no, "1" = offset, ? "2" = data_len, "3" = mode, ? "4" : bstr ; aid }</pre>	Structure defining what data will be read from a MIFARE DESFire card
file_no	Number of the file to be read
offset	Offset into file to start reading
data_len	Length of data to be read. Unused for write operation.
<pre>mode = &mode_enum mode_enum = (PLAIN = 0x00, MAC = 0x01, ENCRYPT = 0x03)</pre>	Communication mode of read operation
aid = bstr	Application ID to be selected (bstr, len = 3). If present, then DesfireSelectApp function is performed. If not, previously selected application is used.

Card Data Seos specific structures

Seos authenticate structure

This is an example of Seos authenticate structure. It is encoded as a CBOR structure and placed inside an action as a byte string.

Structure	Description
<pre>auth_seos = { "0" : adf_oid, "1" : priv_enc_key_oid, "2" : priv_mac_key_oid, "3" : auth_key_oid, "4" : ref_data_qualifier }</pre>	Structure defining data required for authentication to a Seos card
<pre>adf_oid = bstr</pre>	ADF OID
<pre>priv_enc_key_oid = bstr</pre>	Privacy encryption key reference OID
<pre>priv_mac_key_oid = bstr</pre>	Privacy MAC key reference OID
<pre>auth_key_oid = bstr</pre>	Authentication key reference OID
<pre>ref_data_qualifier = bstr</pre>	Reference data qualifier

Seos read/write data structure

The write operation is available only via ERA commands. CPE accepts only read operations.

Reading and writing to Seos involves the Seos chaining mechanism for larger data strings. Chaining must be used when the data size (read/write) is greater than the core buffer (1 KB). The type of credential buffer may also influence this, but it is usually an extremely large number, for example, greater than 1 KB. During a write operation, each data chunk in the chaining sequence between the Core MCU and secure element, excluding the last data portion, must be 224 bytes.

Therefore, each subsequent chunk (excluding the last chunk) from the host application point of view is 896 bytes. Seos chaining requires a specialized assembly present in the secure element.

Example of Seos read/write data structure

Structure	Description
<pre>data_seos = { ? "0" : object_tag, ? "1" : data_len, ? "2" : write_chunk , ? "3" : read_continue }</pre>	Structure defining read/write operations for a Seos credential
<pre>object_tag = bstr</pre>	Object tag: 1 or 2 bytes. This value shall be set only if no chaining is to be used or if this is the first chunk of the chain. Used for read and write operations.
<pre>data_len = int</pre>	Total data length. This value shall be set only if no chaining is to be used or if this is the first chunk of the chain. Used for read and write operations.
<pre>write_chunk_type = &write_chunk_type_enum write_chunk_type_enum = FIRST_CHUNK : 0x01, NEXT_CHUNK : 0x02, LAST_CHUNK : 0x03</pre>	Used for write with chaining operation only. Shall not be set if a single write operation shall be executed.

read_continue = bool	Used for read with chaining operation. If there is more data to be read (indicated by the previous response) this flag shall be set to true and all other items in the structure shall be excluded. This flag shall be set only for subsequent chunks. It shall not be set for the first chunk.
----------------------	--

Seos chaining parameters usage

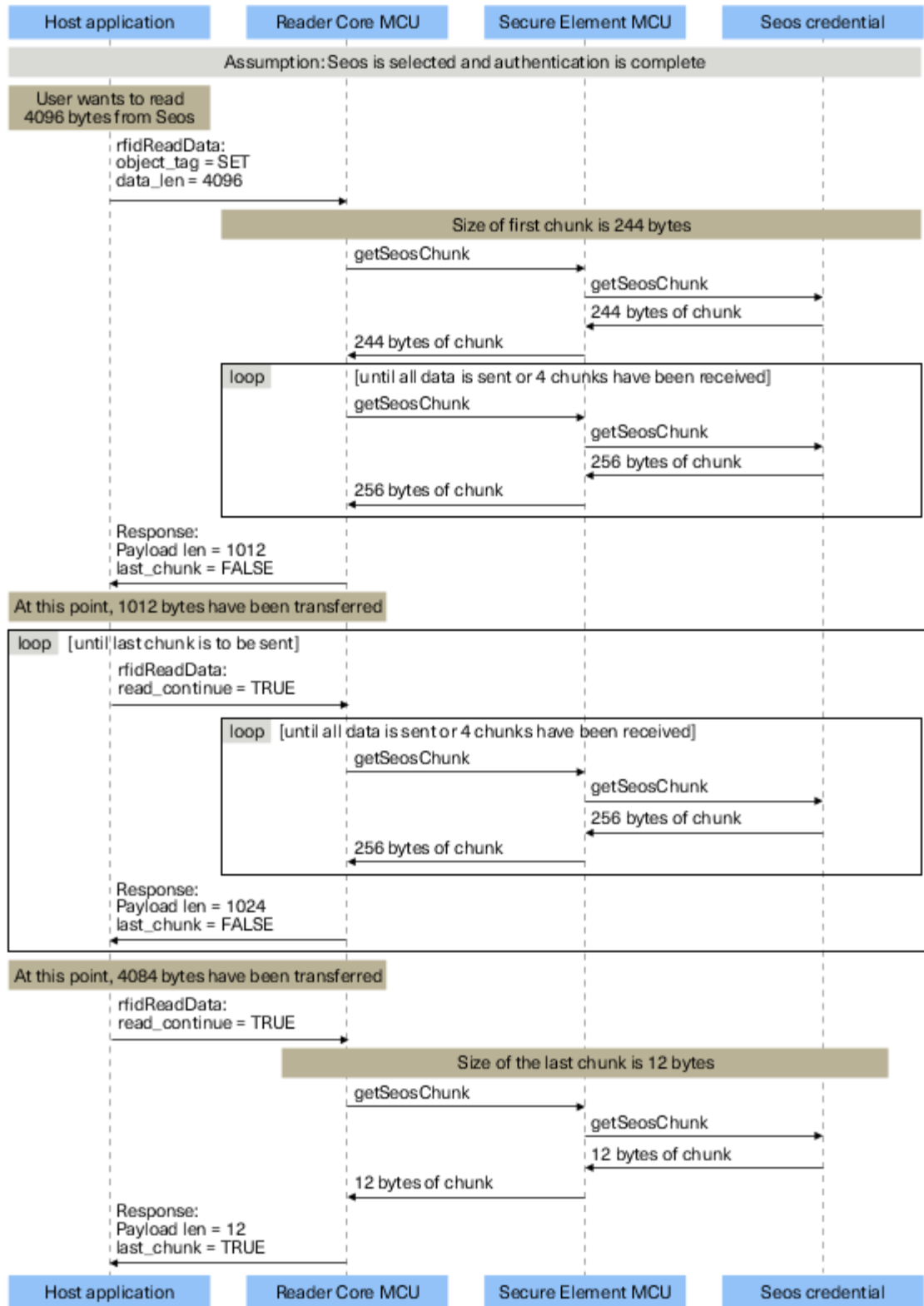
The following table describes typical usages with/without chaining and the required parameters.

Use-case / required (allowed) parameters	Single write operation	Write with chaining (first chunk)	Write with chaining (subsequent/last chunk)	Single read operation	Read with chaining (first chunk)	Read with chaining (subsequent/last chunk)
object_tag	MANDATORY Y	MANDATORY	N/A	MANDATORY Y	MANDATORY	N/A
data_len	MANDATORY Y	MANDATORY	N/A	MANDATORY Y	MANDATORY	N/A
write_chunk	N/A	MANDATORY	MANDATORY	N/A	N/A	N/A
read_continue	N/A	N/A	N/A	N/A	N/A	MANDATORY

Seos chaining procedure flow

Seos chaining read

The following diagram shows the Seos chaining procedure (read).

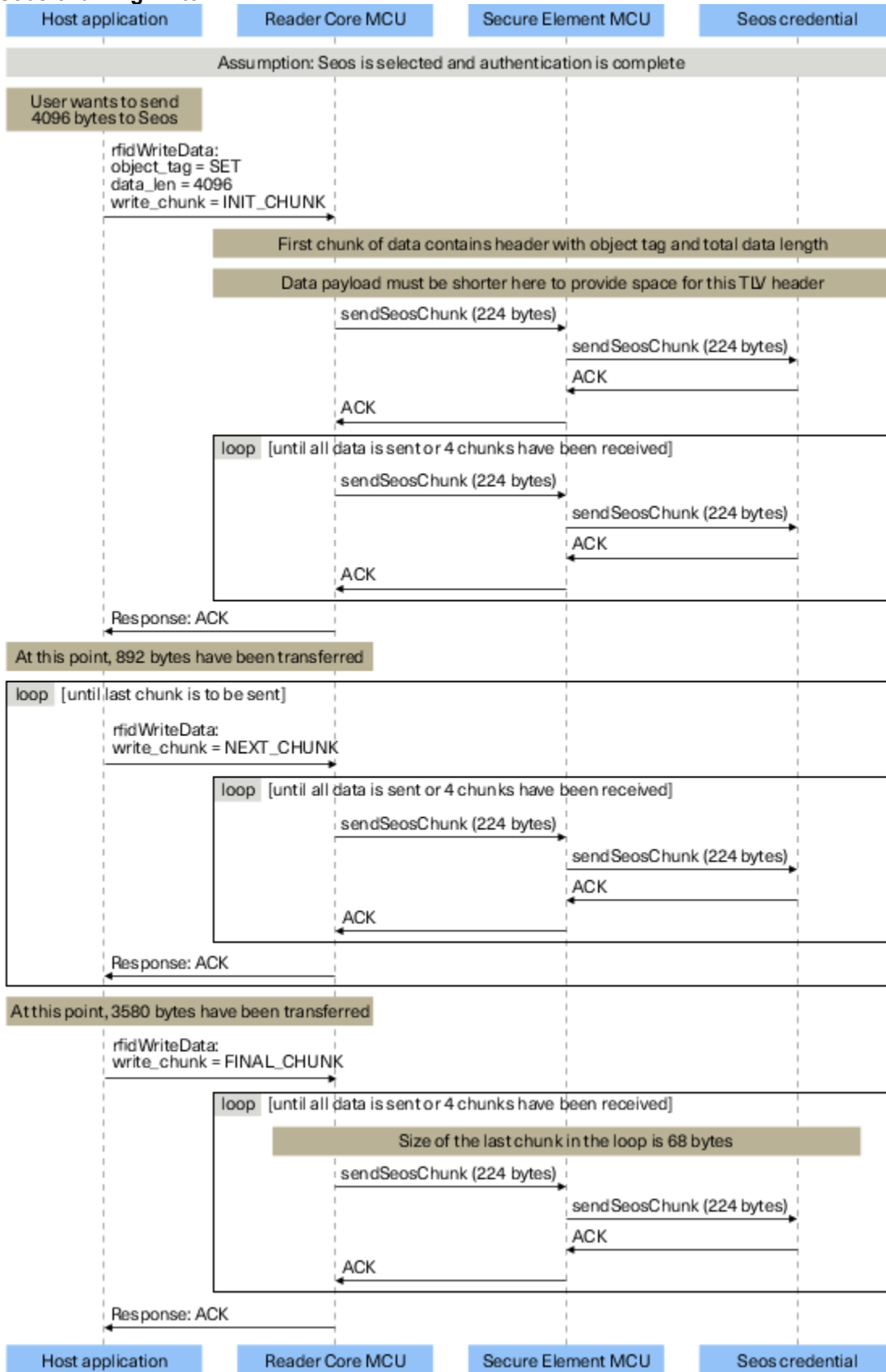


Seos chaining write

Seos chaining write requires that:

- All data chunks need to be divisible by 224, except the last.
 - The data sent by the user is used to calculate a TLV structure that is sent to the card before the data. The size of this TLV data needs to be subtracted from the size of the first chunk of data sent to the reader.
 - The maximum chunk size is 896 bytes.
-
- Data with a size less than or equal to 896 can be sent in one chunk. There is no need to subtract the TLV size in this case.
 - Data with a size less than or equal to 896 can be sent in multiple chunks. For example, 448 bytes need to have space for a TLV structure in the first chunk, for 448 chunks the first needs to be 444 bytes long.
 - Data with size greater than 896 needs to be sent in multiple chunks, where the first is of size:
 $(n * 224) - \text{TLV_size}$
where n is a value 1-4.

Seos chaining write



Seos preloaded default keys

For testing purposes, you can use preloaded Seos Open ADF keys:

Key OID	Description
30501	Seos Open ADF Key ENC
30502	Seos Open ADF Key MAC
30503	Seos Open ADF Key AUTH

Card types

Structure	Description
<pre>card_type = &card_type_enum card_typ_enum = (ICLASS_PICOPASS = 0x00, MIFARE_CLASSIC_1K = 0x01, MIFARE_CLASSIC_4K = 0x02, MIFARE_PLUS = 0x03, MIFARE_DESFIRE = 0x04, MIFARE_ULTRALIGHT = 0x05, SEOS = 0x06, ISO15693 = 0x07, HID_PROX = 0x08, PIV = 0x0A, INDALA = 0x15, EM_PROX = 0x17, TYPE_A_LAYER3 = 0x20, TYPE_A_LAYER4 = 0x21, TYPE_B_LAYER3 = 0x22, TYPE_B_LAYER4 = 0x23, UNKNOWN = 0x24)</pre>	Card types

Anti-passback

Anti-passback prevents reporting any card data more than once within a predefined time window. Anti-passback has two modes of operation for timer handling: fixed or sliding window.

Repeating reads of the credential might have a different effect on the related window timer.

Each credential has its own timer. There might be up to three different credentials recognized by the anti-passback mechanism.

If a new credential is presented and the anti-passback mechanism already keeps track of three, the new one overwrites the oldest in the list.

The width of this time window and timer resolution is measured in milliseconds. If the value of the delay time is set to zero then the anti-passback feature is disabled.

Enabling anti-passback functionality (delay time other than zero) disables all low power modes for the time any credential is blocked (any time window timer running).

Anti-passback implements two card identification mechanisms: CSN (Card Serial Number) or credential data like PACS (for randomized CSN cards). The identification mechanisms work differently:

- CSN identification mechanism: This works only if a static CSN is used. Moreover, a CSN action is not required to be added in a CPE slot to be able to use CSN filtering.
- Credential data mechanism: Filtering works only if CPE is configured to read requested data (PACS).

Configuration

To configure the anti-passback, you must use the External Reader API `configSetParam` command or a configuration file. There are several parameters which can be used to configure the reader:

- Anti-passback enable
Enables/disables the anti-passback feature. For the configuration parameter details, see "Security configuration structure " on page 258.
- Anti-passback delay
Sets the anti-passback delay. The same card cannot be used for the defined duration. For the configuration parameter detail, see "Security configuration structure " on page 258.
- Anti-passback sliding window
Enables/disables the anti-passback sliding window. If enabled, the delay timer is restarted every time the same credential is detected. For the configuration parameter details, see "Security configuration structure " on page 258.

CSN based recognition

The CSN identification mechanism records the Card Serial Number (CSN) of the presented card after it has been successfully processed.

When this CSN is found on the anti-passback blocked credentials list, further processing of this card is stopped. This mechanism is ineffective for cards with a random CSN (such as Seos).

Modes

Credential data based recognition

This mode takes credential data as the card recognition credential and stops its processing if the number is found on the anti-passback blocked credentials list, until the timer runs out. CPE action is required for filtering.

PACS based recognition

This mode takes PACS as the card recognition credential and stops its processing if this number is found on the anti-passback blocked credentials list (until the timer has ended).

Fixed window mode

Each time a new credential is presented, a timer is started. If the same card is detected again, it does not affect the currently running timer. When the timer ends, printing of the data related to this credential is allowed.

Sliding window mode

When a new credential is read, a timer starts. If a card with the same credential is detected again, it restarts the running timer. When the timer finally ends, printing of the data related to this credential is allowed.

Card tracking

Card tracking mechanism when turned on is automatically tracking the Card after readout to avoid repetitive read of the same card. Card tracking can be used together with Constant card report feature.

Configuration

To configure the Card tracking, user needs to use the External Reader API (configSetParam command or configuration via file). Following parameters shall be used to configure the card tracking feature:

- Card tracking enable Enables/disables card tracking feature. For the configuration parameter details please see "Security configuration structure " on page 258

Constant card report

If enabled, the reader will send the same data repeatedly over selected interface as long as the card is kept presented to the reader.

This only works if the card tracking feature is enabled. This feature does not affect the anti-passback feature (meaning if Anti-passback is turned on, the reader will not send reports when the same card is presented again within the antipassback window).

The transmission of data on Wiegand takes significantly more time than using UART interface. This means that if device is configured to read and report long data stream, period of the constant card report shall be long enough so that reader can send whole data.

there is minor risk that when feature is enabled, it could be possible to switch credential to another one and keep it presented. Reader will still report the same data read from first legitimate credential.

Configuration

To configure the Constant card report, you need to use the External Reader API (configSetParam command or configuration via file).

The following parameters shall be used to configure the card tracking feature:

- Constant card report Enables/disables constant card report feature. For the configuration parameter details please see "Anti-passback sliding window " on page 259.
- Constant card report interval Defines the interval of constant card report. For the configuration parameter details please see "Constant card report interval" on page 260.

Heartbeat

To monitor the devices operational status, a heartbeat message can be configured to be sent at a specified interval. Additionally, an event notification will be sent whenever the device undergoes a reset, indicating that a reset has occurred. Following configuration items affect the heartbeat functionality:

- Heartbeat_en - enables / disables the heartbeat functionality.
- Heartbeat_interval - defines the interval of heartbeat message.
- Heartbeat_wiegand_msg - defines the data which shall be sent over Wiegand to indicate the heartbeat (different data for regular heartbeat and reset).

For Autonomous mode, heartbeat is sent on the interface which is selected via Autonomous mode output interface configuration (if data is sent over UART then heartbeat is also sent over UART, similarly for Wiegand). Heartbeat affects Low Power mode of OMNIKEY SE Reader Core effectively disabling it.

Uart output

Uart text format

In text mode the reader will send following messages:
"OMNIKEY_SE_RC_HEARTBEAT_RESET" - just after the reader reset.
"OMNIKEY_SE_RC_HEARTBEAT" - with configured interval.
End-Of-Line characters in heartbeat messages are CRLF [0x0D, 0x0A].

Uart binary format

In binary mode data is encoded into CBOR format according to the following structure:

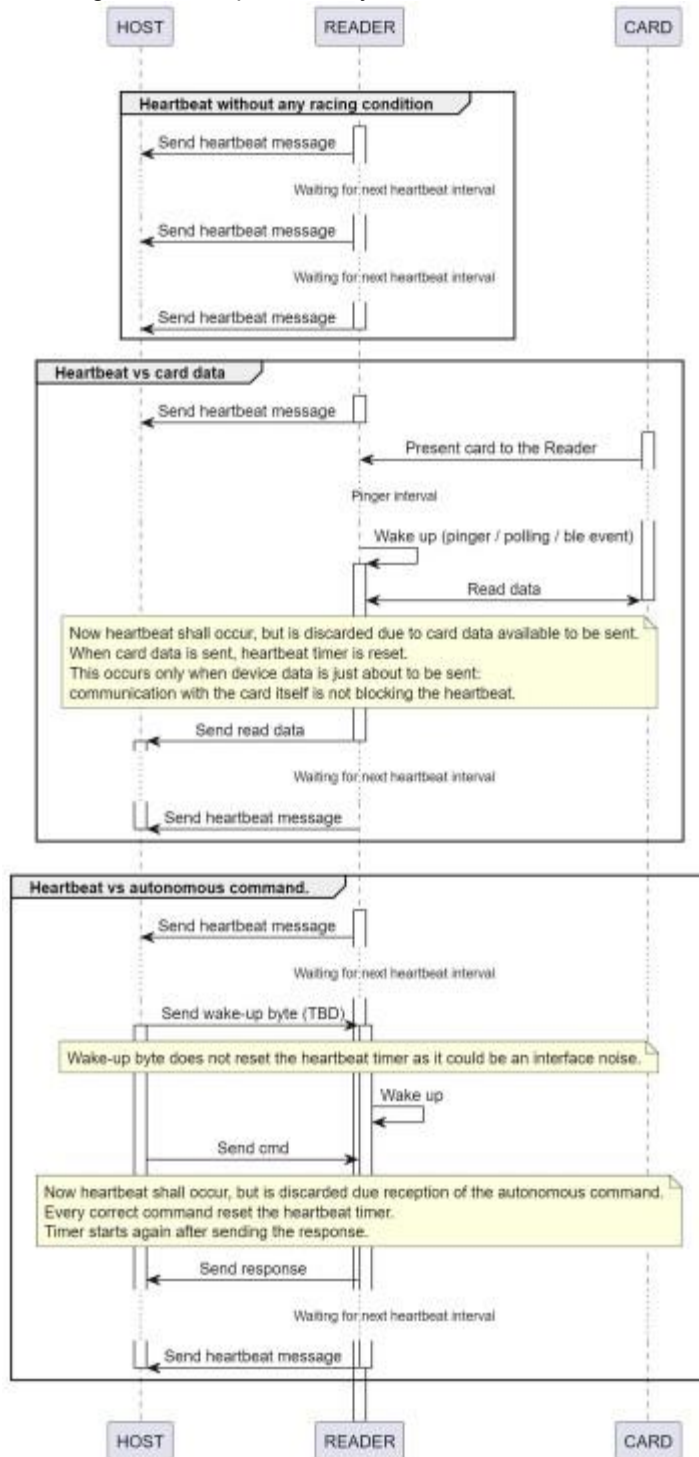
Structure	Description
autonomous_mode_data = { "0" : event_type }	Type of event
event_type = (HEARTBEAT = 0x00, HEARTBEAT_RESET = 0x01)	Supported event types

Example

CBOR heartbeat message: A1613000
Decoded:
autonomous_mode_data = {
 "0": 0
}
CBOR heartbeat reset message: A1613001
Decoded:
autonomous_mode_data = {
 "0": 1
}

Uart output synchronization with other events

The diagram below presents synchronization of Heartbeat with other events.



Wiegand output

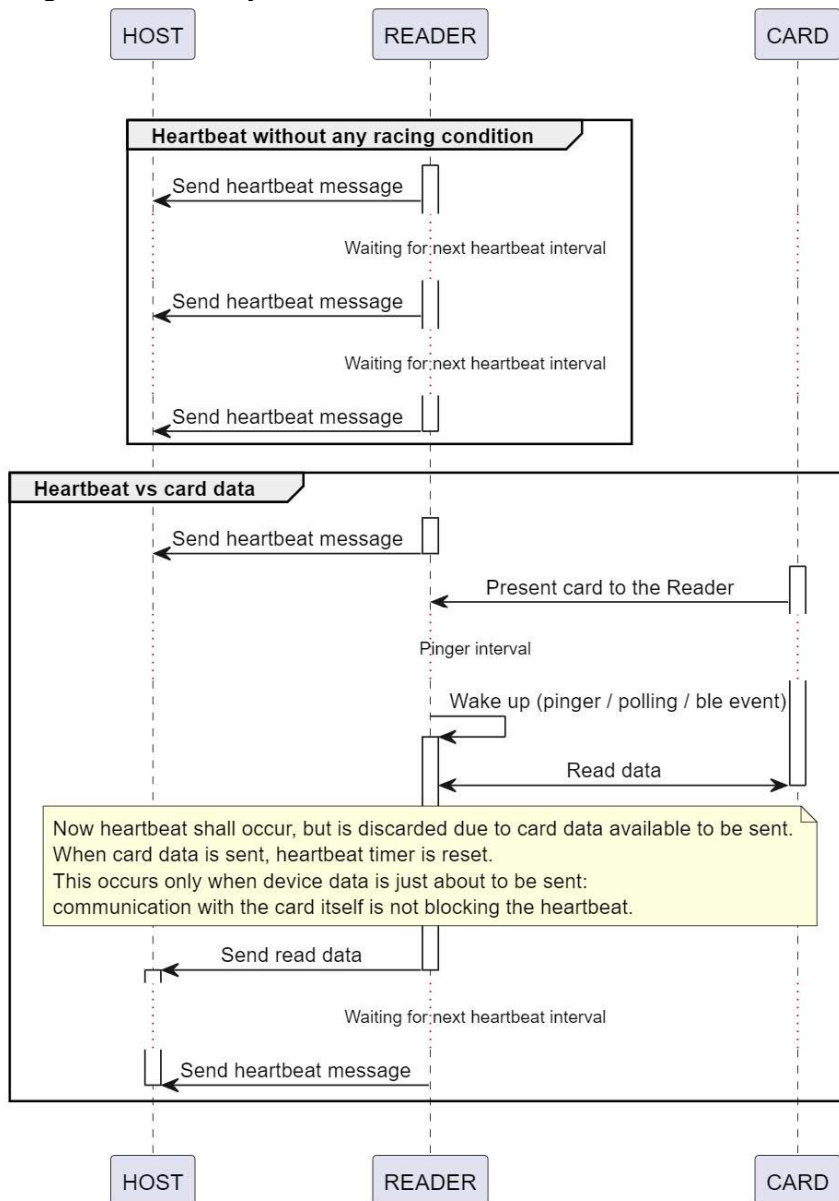
Reader is putting configured bytes defined in Heartbeat_wiegand_msg and Heartbeat_wiegand_reset_msg on the Wiegand interface.

Wiegand is not incorporating any protocol, therefore time guard and data payload (even size) is used to distinguish between card data and heartbeat.

Wiegand output synchronization with other events

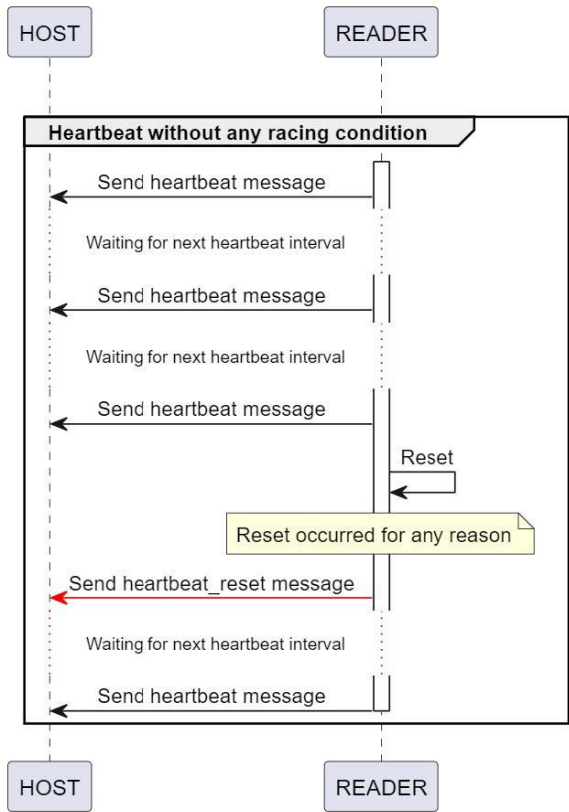
The diagram below presents synchronization of Heartbeat with other events.

Wiegand Heartbeat synchronization.



Reset event

The diagram below describes Heartbeat reset event occurrence.
Heartbeat reset event.



CDRE actions compatibility charts

CDRE actions compatibility automatic readout (Autonomous mode)

	CSN	PACS	CUSTOM	TRANSACTION	ALLEGION (AA ONLY)
ICLASS_PICOPASS	X	X	X		
MIFARE_CLASSIC_1K	X	X	X		
MIFARE_CLASSIC_4K	X	X	X		
MIFARE_PLUSX	X				
MIFARE_DESFIRE	X	X	X	X	X
MIFARE_ULTRALIGHT	X				
SEOS	X	X	X		
HID_PROX		X			
PIV	X		X		
INDALA		X			
EM_PROX		X			
ISO15693					
TYPE_A_LAYER3					
TYPE_A_LAYER4					
TYPE_B_LAYER3					
TYPE_B_LAYER4					

CDRE actions compatibility (Autonomous formatter support)

	CSN	PACS (see note)	CUSTOM (see note)
ICLASS_PICOPASS	X	X	X
MIFARE_CLASSIC_1K	X	X	X
MIFARE_CLASSIC_4K	X	X	X
MIFARE_PLUSX	X		
MIFARE_DESFIRE	X	X	X
MIFARE_ULTRALIGHT	X		
SEOS	X	X	X
HID_PROX		X	
PIV	X		X
ISO15693			
TYPE_A_LAYER3			
TYPE_A_LAYER4			
TYPE_B_LAYER3			
TYPE_B_LAYER4			

For now the only functionality implemented for PACS and CUSTOM formatters is discard flag.

CDRE Actions compatibility Host driven mode (ERA)

	CSN	PACS	CUSTOM	TRANSACT	ALLEGION (AA ONLY)
ICLASS_PICOPASS	X	X	X		
MIFARE_CLASSIC_1K	X	X	X		
MIFARE_CLASSIC_4K	X	X	X		
MIFARE_PLUSX	X				
MIFARE_DESFIRE	X	X	X	X	X
MIFARE_ULTRALIGHT	X				
SEOS	X	X	X		
HID_PROX		X			
PIV	X		X		
ISO15693					
TYPE_A_LAYER3					
TYPE_A_LAYER4					
TYPE_B_LAYER3					
TYPE_B_LAYER4					

External Reader API mode

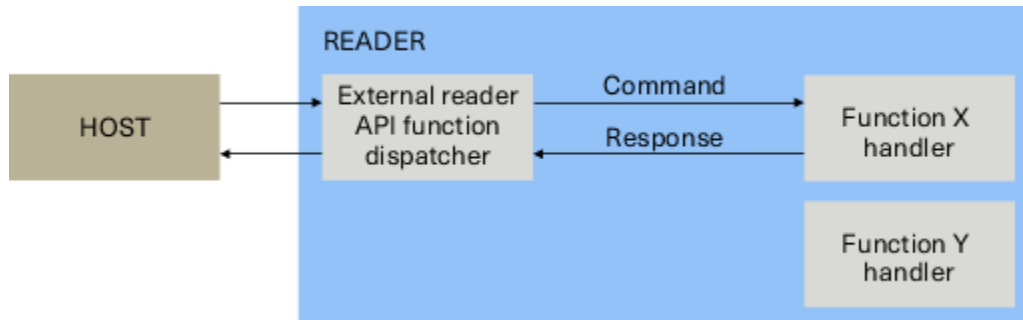
Introduction

In the External Reader API application mode, the reader accepts External Reader API commands. The commands are used for reader configuration, FW upgrade, and advanced operations on cards and I/O. This mode can be entered by:

- Direct entry after startup when set as default mode.
- Leaving any Autonomous mode.

Data readout flow

The data flow inside the External Reader API is organized around a single command dispatcher where the received External Reader API command is analyzed, validated for completeness, and then directed towards a specific command handler.



External Reader API commands

Introduction

The External Reader API exposes resources that you can interact with by invoking particular CoAP methods. Each resource represents a particular feature of the reader and may expose a different set of available methods, like GET, POST or DELETE. Each method for every resource allows a specific command to be executed, ranging from simple actions like changing the reader state, to file uploads and specific tasks like reading and handling credentials.

General reader commands

General commands are used to switch between different reader modes, get the latest reader status, and trigger a reader reset or shutdown.

genChangeState

Changes the mode/state of the reader. Before execution, the reader will send the response CHANGED (204) to indicate that the command has been accepted. In case of any issues, an error code will be sent and the mode/state will not be changed. This command enters the requested mode only temporarily; the default mode is not modified. For most of the modes, which are transient, it is not possible to set the same mode as the default. If reset is selected, this command triggers a device reset and reboots to default mode (see "Modes transition" on page 12) so it is the only way to finish a started update. If a delay is specified, the reader will not send any additional indication before executing the mode change (after the delay).

You could define an audio-visual sequence for entering APP_MODE_SHUTDOWN, triggered before a mode switch. If a user-defined sequence is longer than 10 seconds, the reader automatically cuts the sequence to 10 seconds.

If an AV sequence of higher priority is triggered (for example ATTACK DETECTED sequence) it will be cut off after 10 seconds and the reader will switch to SHUTDOWN mode.

Command format

URI	/state
TYPE	CON
REQUEST CODE	POST
args = { "0" : mode_sel, ? "1" : stim }	Command arguments.
mode_sel= &mode_sel_enum mode_sel_enum= (APP_MODE_SHELL : 0x00, APP_MODE_SHUTDOWN : 0x02, APP_MODE_AUTONOMOUS : 0x03, APP_MODE_RESET : 0x04)	Mode selection.
stim = 0...10000	Delay before the reset.

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.04 CHANGED
RESPONSE CODE POSSIBLE ERRORS	5.03 SERVICE UNAVAILABLE Device is busy and cannot process the client's request.
	4.02 BAD REQUEST Incorrect param has been provided or CBOR structure is corrupted.
	4.29 TOO MANY REQUESTS Previous command processing has not finished yet.
data = none	No response data is returned,

Examples

Example description	Full CoAP command	Arg / Res data
Command: Reset the reader after 1 sec	Hex: 4002295BB57374617465FF A261300461311903E8	Hex: A261300461311903E8 Decoded:

Example description	Full CoAP command		Arg / Res data		
	Decoded:		A2	# map	
	40	# CON, token len = 0	61 30	# ARG1: mode sel	
	02	# POST	04	# RESET	
	295B	# MSG ID	61 31	# ARG2: delay	
	B57374617465	# Option URI = /state	19 03E8	# 1000ms	
	FF	# End of options marker			
	A261300461311903E8	# Args			
Response: OK	Hex: 6044295B	NONE			
	Decoded:				
60	# ACK, token len = 0				
44	# CHANGED				
	295B	# MSG ID			

genGetSerialNumber

Gets the serial number of the reader.

Command format

URI	/sn
TYPE	CON
REQUEST CODE	GET
ARG data = none	No arguments are required.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.05 CONTENT	
RESPONSE CODE POSSIBLE ERRORS	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS	Previous command processing has not been finished yet.
data = { "0" : serial_number }	Command arguments.	
serial_number = text	Serial number of the reader.	

Examples

Example description	Full CoAP command	Arg / Res data
Command: Get serial number of the reader.	Hex: 4001ED77B2736E Decoded: 40 #CON, token len = 0 01 #GET ED77 #MSG ID B2736E #Option URI = /sn	NONE
Response: OK	Hex: 6045ED77FFA1613064AABBCCDD Decoded: 60 #ACK, token len = 0 45 #CONTENT ED77 #MSG ID FF #End of options marker A1613044AABBCCDD #Args	Hex: A1613044AABBCCDD Decoded: A1 #map 61 30 #ARG: serial number 64 #text(4) AABBCCDD #Serial number

genGetUUID

Gets the UUID of the reader.

Command format

URI	/uuid
TYPE	CON
REQUEST CODE	GET
ARG data = none	No arguments are required.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.05 CONTENT	
RESPONSE CODE POSSIBLE ERRORS	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS	Previous command processing has not been finished yet.
data = { "0" : uuid }	Command arguments.	
uuid = text	UUID of the reader.	

Examples

Example description	Full CoAP command	Arg / Res data
Command: Get UUID of the reader.	Hex: 4001ED77B475756964 Decoded: 40 #CON, token len = 0 01 #GET ED77 #MSG ID B475756964 #Option URI = /uuid	NONE
Response: OK	Hex: 6045ED77FFA161306441424344 Decoded: 60 #ACK, token len = 0 45 #CONTENT ED77 #MSG ID FF #End of options marker A161306441424344 #Payload (response)	Hex: A161306441424344 Decoded: A1 #map 61 30 #ARG: UUID 64 #text ABCD #UUID

genGetDeviceStatus

This command is used to get the current status of the device.

Command format

TYPE	CON
REQUEST CODE	GET
args = none	No arguments are required.

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.05 CONTENT
RESPONSE CODE POSSIBLE ERRORS	5.03 SERVICE UNAVAILABLE Device is busy and cannot process the client's request. 4.29 TOO MANY REQUESTS Previous command processing has not been finished yet.
data = { "0" : status_code, }	Response data
device_state = &device_state_enum device_state_enum = (DEVICE_STATE_OK: 0, DEVICE_STATE_ERROR_GENERIC: 1, DEVICE_STATE_ERROR_DEVICE_LOCKED: 2, DEVICE_STATE_ERROR_DEVICE_MALFUNCTION: 3,)	Device status codes. * DEVICE_STATE_OK - No error detected * DEVICE_STATE_ERROR_GENERIC - Core error detected - RFU * DEVICE_STATE_ERROR_DEVICE_LOCKED - Device is locked due to attack detection * DEVICE_STATE_ERROR_DEVICE_MALFUNCTION - Device detected internal malfunction In case of status codes: * DEVICE_STATE_ERROR_GENERIC * DEVICE_STATE_ERROR_DEVICE_LOCKED * DEVICE_STATE_ERROR_DEVICE_MALFUNCTION contact with HID field support is required.

Examples

Full CoAP command	Arg / Res data
Hex: 40013f80b6737461747573 Decoded: 40 # CON, token len = 0 01 # GET 3F80 # MSG ID B6737461747573 # Option URI = /status Hex: 60453f80ffa1613000	NONE Hex: A1613000 Decoded: A1 # map

Full CoAP command	Arg / Res data
Decoded: 60 # ACK, token len = 0 45 # CONTENT 3F80 # MSG ID FF # End of options marker A1613000 # Data	61 30 # ARG1: status code 00 # DEVICE_STATE_OK

genWhoAreYou

Gets basic information about the reader.

Command format

URI	/who-are-you
TYPE	CON
REQUEST CODE	GET
ARG data = none	No arguments are required.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.05 CONTENT	
RESPONSE CODE POSSIBLE ERRORS	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS	Previous command processing has not been finished yet.
data = { "0" : platform, "1" : device_type, "2" : api_version ?"3" : reader_friendly_name }	Command arguments.	
platform = text	Platform name of the device. Current implementation returns "OMNIKEY_SE_READER"	
device_type = &device_type_enum device_type_enum = CORE : 0x00	Command arguments.	
api_version = uint	Version of the implemented External Reader API. Current implementation returns 2.	
reader_friendly_name = text	Reader friendly name text string, by default value is empty.	

Examples

Example description	Full CoAP command	Arg / Res data
Command: Get who are you response.	Hex: 400106AFBB77686F2D6172652D796F75 Decoded: 40 01 06AF BB77686F2D6172652D796F75	NONE #CON, token len = 0 #GET #MSG ID #Option URI = /who-are-you
	Hex: 604506AFFFA46130714F4D4E494B 45595F53455F5245414445526131 006132026133644E616D65	Hex: A46130714F4D4E494B45595F53455 F5245414445526131006132026133 644E616D65

Example description	Full CoAP command	Arg / Res data
Decoded:		Decoded:
60	#ACK, token len = 0	A3
45	#CONTENT	#map
06AF	#MSG ID	61 30
FF	#End of options marker	71
A46130714F4D4E	#Args	#text(17)
494B45595F53455		4F4D4E494B45595F5
F5245414445526		3455F524541444552
13100613203613		#'OMNIKEY_SE_R
3644E616D65		EADER'
		AABBCCDD
		#Serial number
		61 31
		#ARG: device_type
		0
		#unsigned (0)
		CORE
		61 32
		#ARG: api_version
		02
		#unsigned (2) value
		61 33
		#ARG
		reader_friendly_na
		me
		64
		#text(4)
		4E616D65
		#'Name'

genGetHwInfo

Gets the reader HW info.

Command format

URI	/hw-info
TYPE	CON
REQUEST CODE	GET
args = none	No arguments are required.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.05 CONTENT	
RESPONSE CODE POSSIBLE ERRORS	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS	Previous command processing has not been finished yet.
data = { "0" : device_class_id, "1" : hw_revision, "2" : product_info, "3" : date }	Response data.	
device_class_id = bstr	Device class id	
hw_revision = bstr	HW (PCA) revision as three ascii chars e.g. " A, "A.1", "A.2"	
product_info = [reader_type : uint, color : uint, rfu : uint]	Product information containing type and color, encoded.	
rfu = 0xFFFF		
date = [day : uint, month : uint, year : uint]	Manufacturing date as uint array.	

Examples

Example description	Full CoAP command	Arg / Res data
Command: Get HW info	Hex: 4001C79CB768772D696E666F Decoded: 40 01 C79C	NONE #CON, token len = 0 #GET #MSG ID

Example description	Full CoAP command		Arg / Res data	
	B768772D696E666F	#Option URI = hw-info		
Response: HW info	Hex:		Hex:	
	6045C79CFFA46130503824665		A46130503824665E342052558759C67	
	E342052558759C67D510416FA		D510416FA613143422E316132831841	
	613143422E31613283184118		184219FFFF6133831819051907E7	
	4219FFFF6133831819051907E7		Decoded:	
	Decoded:		A4	#map(4)
	60	#ACK, token len = 0	61	#text(1)
	45	#CONTENT	30	#"0"
	C79C	#MSG ID	50	#bytes(16)
	FF	#End of options marker	3824665E34205255	#uuid
			8759C67D510416FA	
	A46130503824665	#Data	61	#text(1)
	E342052558759C67		31	#"1"
	D510416FA6131434		43	#bytes(3)
	22E3161328318411		422E31	#hw_rev: "B.1"
	84219FFFF61338318		61	#text(1)
	19051907E7		32	#"2"
			83	#product info:
			18 41	#type: "A"
			18 42	#color: "B"
			19 FFFF	#rfu: 0xFFFF
			61	#text(1)
			33	#"3"
			83	#date
			18 19	#day: 25
			05	#month: 5
			19 07E7	#year: 2023

genGetIdentity

Gets the identity of the device.

Command format

URI	/identity
TYPE	CON
REQUEST CODE	GET
ARG data = none	No arguments are required.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.05 CONTENT	
RESPONSE CODE POSSIBLE ERRORS	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS	SE is busy with another operation requested by the same client. Command shall be retried.
data = { "0" : identity }	Response data.	
identity = bstr	Response from SE containing engine_id and username.	

Examples

Example description	Full CoAP command		Arg / Res data	
Command: Get reader's SE identity.	Hex:		NONE	
	4001ED77B86964656E74697479			
	Decoded:			
	40	#CON, token len = 0		
	01	#GET		
Response: Reader Engine_ID and username	ED77	#MSG ID	Hex: A26130440011223361314400112233 Decoded: A2#map 61 30#ARG1: identity 49#bstr 001122334455667788#dummy identity	
	B86964656E74	#Option URI = /identity		
	Hex:			
	6045ED77FFA261304400112233			
	61314400112233			
	Decoded:			
	60	#ACK, token len = 0		
	45	#CONTENT		
	ED77	#MSG ID		
	FF	#End of options marker		
	A2613044001122336	#Args		
	1314400112233			

genGetReleaseVersion

Gets the release version of the reader.

Command format

URI	/release
TYPE	CON
REQUEST CODE	GET
ARG data = none	No arguments are required.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.05 CONTENT	
RESPONSE CODE POSSIBLE ERRORS	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS	Previous command processing has not been finished yet.
data = { "0" : release_version }	Command arguments.	
release_version = text	Release version of the firmware.	

Examples

Example description	Full CoAP command	Arg / Res data
Command: Get release version of the reader	Hex: 4001ED77B772656C65617365 Decoded: 40 #CON, token len = 0 01 #GET ED77 #MSG ID B772656C65617365 #Option URI = /release	NONE
Response: OK	Hex: 6045ED77FFA161306432342E42 Decoded: 60 #ACK, token len = 0 45 #CONTENT ED77 #MSG ID FF #End of options marker A161306432342E42 #Payload (response)	Hex: A161306432342E42 Decoded: A1 #map 61 30 #ARG1: Release version 64 #text 24.B #Release version

I/O commands

I/O control commands are used to control the reader's LEDs and buzzer. In addition to simple set and get operations, sequences can be generated on both the LEDs and buzzer.

ioGetLedState

Gets the current state of an LED.

Command format

URI	/led
TYPE	CON
REQUEST CODE	GET
args = { "0" : led_sel }	Command arguments.
led_sel = &led_sel_enum led_sel_enum = (ERA_IO_LED_RED : 0x00, ERA_IO_LED_GREEN : 0x01, ERA_IO_LED_BLUE : 0x02)	Selection of the LED.

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.05 CONTENT
RESPONSE CODE POSSIBLE ERRORS	4.00 BAD REQUEST Incorrect param has been provided or CBOR structure is corrupted.
	5.03 SERVICE UNAVAILABLE Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS Previous command processing has not finished yet.
data = { "0" : val_sel }	Response data.
val_sel = &val_sel_enum val_sel_enum = (ERA_IO_OFF : 0x00, ERA_IO_ON : 0x01)	Current value of selected LED.

Examples

Example description	Full CoAP command	Arg / Res data
Command: Get state of a LED1.	Hex: 4001574EB36C6564FFA1613001 Decoded: 40 01 574E B36C6564 FF	Hex: A1613001 Decoded: A1 61 30 01
	#CON, token len = 0	#map
	#GET	#ARG1: led_sel
	#MSG ID	#LED1
	#Option URI = led	
	# End of options marker	

Example description	Full CoAP command	Arg / Res data
Response: LED state = ON	A1613001 #Args	
	Hex: 6045574E	Hex: A1613001
	Decoded: 60	Decoded: A1
	45 #ACK, token len = 0	61 30 #map
	574E #CONTENT	01 #ARG1: val_sel
	FF #MSG ID	01 #ON
	A1613001 #End of options marker	
Response: Wrong param value	A1613001 #Data	
	Hex: 6080574E	NONE
	Decoded: 60	
	80 #ACK, token len = 0	
	574E #BAD REQUEST	
	574E #MSG ID	

ioSetLed

Controls the LEDs. It is possible to change the state of the LED:

- Immediately.
- With a fading effect.

If the fading effect is to be used, an additional `ftim` argument must be provided. If the immediate operation is selected, `ftim` is ignored.

The fading effect is not implemented. A request for fading (a value other than 0) will cause the 5.01 NOT IMPLEMENTED status to be returned.

Command format

URI	/led
TYPE	CON
REQUEST CODE	POST
args = { "0" : led_sel, "1" : val_sel, "2" : modulation_time }	Command arguments.
led_sel = &led_sel_enum led_sel_enum = (ERA_IO_LED_RED : 0x00, ERA_IO_LED_GREEN : 0x01, ERA_IO_LED_BLUE : 0x02)	Selection of the LED.
val_sel = &val_sel_enum val_sel_enum = (ERA_IO_OFF : 0x00, ERA_IO_ON : 0x00)	Specifies if the LED will be turned on.
modulation_time = 0...1000	Used for the fading effect. Defines how long the fading lasts. Possible values: 0 (instant) up to 1000. Unit is 10 ms, so modulation_time = 10 means 100 ms. NOT IMPLEMENTED. Request for fading (a value other than 0) will cause the ERA_STATUS_ERROR_NOT_IMPLEMENTED status to be returned.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.04 CHANGED	
RESPONSE CODE POSSIBLE ERRORS	5.01 NOT IMPLEMENTED	Feature is not yet implemented.
	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.00 BAD REQUEST	Incorrect param has been provided or CBOR structure is corrupted.
	4.29 TOO MANY REQUESTS	Previous command processing has not been finished yet.
data = none	No response data is returned.	

Examples

Example description	Full CoAP command		Arg / Res data	
Command: Turn on LED1 instantly.	Hex:		Hex:	
	40025D18B36C6564FF		A3613001613101613200	
	A3613001613101613200		Decoded:	
	Decoded:		A3	#map
	40	#CON, token len = 0	61 30	#ARG1: led_sel
	02	#POST	01	#LED1
	5D18	#MSG ID	61 31	#ARG2: val_sel
	B36C6564	#Option URI = led	01	#Val to be set = ON
	FF	#End of options marker	61 32	#ARG3: modulation_time
		00	#time to be set = 0 (instant)	
A3613001613101613200	#Args			
Command: Turn off LED1 with fading effect lasting for 2 s.	Hex:		Hex:	
	40025DDFB36C6564		A3613001613100613218C8	
	FFA3613001613100613218C8		Decoded:	
	Decoded:		A3	#map
	40	#CON, token len = 0	61 30	#ARG1: led_sel
			01	#LED1
	02	#POST	61 31	#ARG2: val_sel
	5DDF	#MSG ID	00	#Val to be set = OFF
	B36C6564	#Option URI = led	61 32	#ARG3: modulation_time
FF	#End of options marker	18 C8	#Duration = 200 = 2000 ms == 2 s	
A3613001613100613218C8	#Args			
Response: OK	Hex:		NONE	
	60445D18			
	Decoded:			
	60	#ACK, token len = 0		
44	#CHANGED			
5D18	#MSG ID			
Response: Wrong param value	Hex:		NONE	
	60805D18			
	Decoded:			
	60	#ACK, token len = 0		
80	#BAD REQUEST			
5D18	#MSG ID			
Response: Not implemented	Hex:		NONE	
	60A15DDF	Decoded:		
	60	#ACK, token len = 0		
	A1	#NOT IMPLEMENTED		
5DDF	#MSG ID			

ioGetBeeperState

Gets the current state of the beeper.

Command format

URI	/beeper
TYPE	CON
REQUEST CODE	GET
args = none	No arguments are returned.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.05 CONTENT	
RESPONSE CODE POSSIBLE ERRORS	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS	Previous command processing has not been finished yet.
data = { "0" : val_sel, }	Response data.	
val_sel = &val_sel_enum val_sel_enum = (ERA_IO_OFF : 0x00, ERA_IO_ON : 0x01)	Current value of beeper.	

Examples

Example description	Full CoAP command	Arg / Res data
Command: Get state of the beeper.	Hex: 40013909B6626565706572 Decoded: 40 01 3909 B6626565706572 #CON, token len = 0 #GET # MSG ID #Option URI = beeper	NONE
Response: Beeper state = ON.	Hex: 60453909FFA1613001 Decoded: 60 45 3909 FF A1613001 #ACK, token len = 0 #CONTENT #MSG ID #End of options marker #Data	Hex: A1613001 Decoded: A1 61 30 01 #map #ARG1: val_sel #ON

ioSetBeeper

Controls the beeper.

Command format

URI	/beeper
TYPE	CON
REQUEST CODE	POST
args = { "0" : val_sel, }	Command arguments.
val_sel = &val_sel_enum val_sel_enum = (ERA_IO_OFF : 0x00, ERA_IO_ON : 0x01)	Specifies if the beeper will be turned on or off.

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.04 CHANGED
RESPONSE CODE POSSIBLE ERRORS	4.00 BAD REQUEST Incorrect param has been provided or CBOR structure is corrupted.
	5.03 SERVICE UNAVAILABLE Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS Previous command processing has not finished yet.
data = none	No response data is returned.

Examples

Example description	Full CoAP command	Arg / Res data
Command: Turn the beeper on.	Hex: 4002FE0AB6626565706572FFA1613001 Decoded: 40 02 FE0A B6626565706572 FF A1613001 #CON, token len = 0 #POST #MSG ID #Option URI = beeper #End of options marker #Args	Hex: A1613001 Decoded: A1 61 30 01 #map #ARG1: val_sel #Val to be set = ON
Response: OK.	Hex: 6044FE0A Decoded: 60 44 FE0A #ACK, token len = 0 #CHANGED #MSG ID	NONE
Response:	Hex:	NONE

Example description	Full CoAP command		Arg / Res data
Wrong param value.	6080FE0A		
	Decoded:		
	60	#ACK, token len = 0	
	80	#BAD REQUEST	
	FE0A	#MSG ID	

ioGetFreeAudioVisualSlot (deprecated)

Command is deprecated and will be removed in the next release.
Gets the first audio-visual sequence free slot.

Command format

URI	/io-seq/slots
TYPE	CON
REQUEST CODE	GET
args = none	No arguments are returned.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.05 CONTENT	
RESPONSE CODE POSSIBLE ERRORS	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	5.00 INTERNAL SERVER ERROR	No free slots.
	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
data = { "0" : slot_number }	Response data.	
slot_number = 1...14	Free slot number.	

Examples

Example description	Full CoAP command		Arg / Res data	
Command: Get free slot.	Hex:	40013909B6696F2D73657105736C6F7473	NONE	
	Decoded:			
	40			
	01			
	3909			
	B6696F2D736571			
Response: Free slot number = 1.	05736C6F74	#CON, token len = 0 #GET #MSG ID #Option URI = io-seq #Option URI = slots	Hex: A1613001 Decoded: A1 61 30 01 #map #ARG1: slot number #1	
	Hex:			
	60453909FFA1613001			
	Decoded:			
	60			
	45			
	3909	#ACK, token len = 0 #CONTENT #MSG ID #End of options marker #Data		
	FF			
	A1613001			

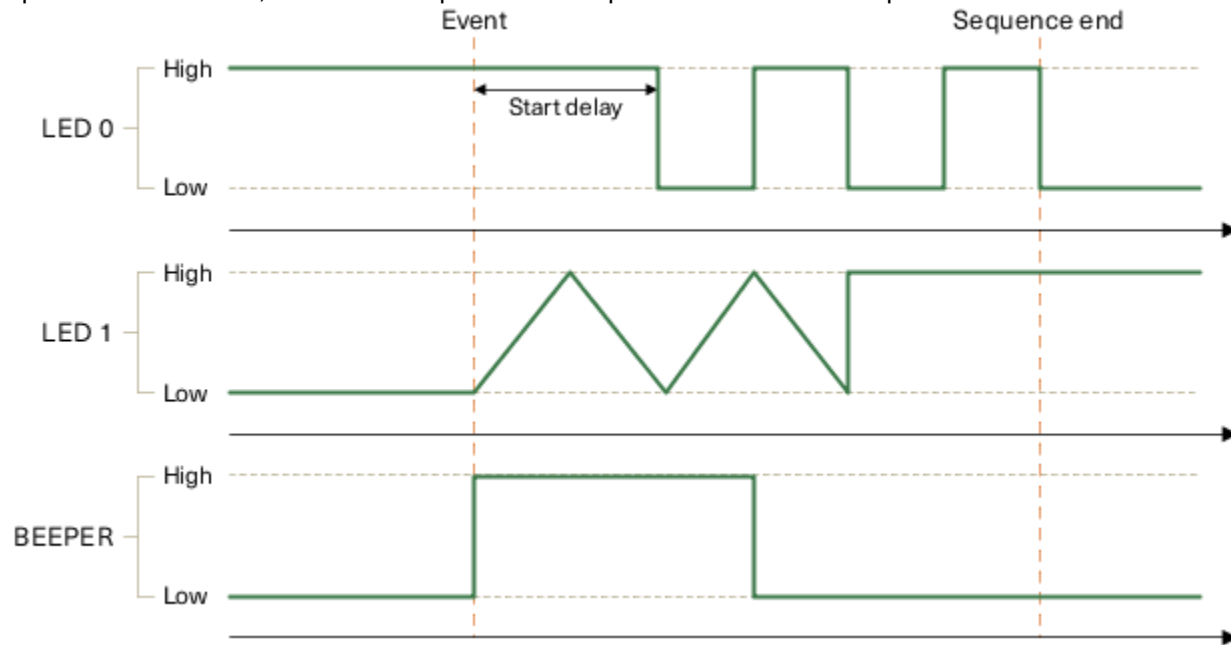
ioSetAudioVisualSequence (deprecated)

Command is deprecated and will be removed in the next release.
Configures an audio-visual sequence. Up to 15 sequences are available.

- Fixed, predefined, constant sequence (index 0) that turns off all I/O.
- 14 configurable sequences (indexes from 1 to 14).

In the default reader configuration, sequences 1 to 7 are predefined and associated with appropriate system events. However, you can customize the sequences and their connection with system events at any time. One sequence consists of independent configurations for each LED and the beeper. You can configure a single I/O or any combination of all I/O. The following diagram shows an example of a configuration. The exact data is described in the following table.

The functionality related to `up_time` and `down_time`, which involves the brightening and fading of LEDs, is currently not operational. However, this feature is planned for implementation in future updates.



I/O sequence triggered by an event

I/O	Start delay	Stage 0	Stage 1	Stage 2
LED0	1000 ms	up_time: 0 ms up_sustain_time: 0 ms down_time: 0 ms down_sustain_time: 500 ms repeat_times: 1	up_time: 0 ms up_sustain_time: 500 ms down_time: 0 ms down_sustain_time: 500 ms repeat_times: 1	up_time: 0 ms up_sustain_time: 500 ms down_time: 0 ms down_sustain_time: 0xFFFF repeat_times: 1
LED1	0 ms	up_time: 500 ms up_sustain_time: 0 ms down_time: 500 ms down_sustain_time: 0 ms repeat_times: 2	up_time: 0 ms up_sustain_time: 0xFFFF down_time: 0 ms down_sustain_time: 0 ms repeat_times: 1	
BEEPER	0 ms	up_time: 0 ms up_sustain_time: 1500 ms down_time: 0 ms down_sustain_time: 0xFFFF repeat_times: 1		

Command format

URI	/io-seq/slots
TYPE	CON
REQUEST CODE	POST
args = { "0" : sequence_slot, "1" : bstr ; sequences }	Command arguments. The field with key "1" contains a byte string with CBOR encoded sequences.
sequence_slot = 1 .. 14	Number of slot for AV sequence.
sequences = { ? "0" : [+ led_sequence], ? "1" : beeper_sequence }	Definition of AV sequences.
led_sequence = { "0" : start_delay, "1" : led_sel, "2" : bstr ; sequence_stages, ? "3" : inf_flag }	Definition of a sequence for single LED. The field with key "2" contains a byte string with CBOR encoded sequence stages.
beeper_sequence = { "0" : start_delay, "1" : bstr ; sequence_stages, ? "2" : inf_flag }	Definition of a sequence for beeper. The field with key "1" contains a byte string with CBOR encoded sequence stages.
sequence_stages = [+ sequence_stage]	Definition of a single sequence stage.
sequence_stage = [up_time: UpTime, up_sustain_time: UpSustainTime, down_time: DownTime, down_sustain_time: DownSustainTime, repeat_times: RepeatTimes]	Definition of a single sequence stage.
start_delay = 0 .. 500	Delay between the event start and the LED/beeper sequence start. Unit is 10 ms, so a start_delay of 100 means 1000 ms.
led_sel = &led_sel_enum led_sel_enum = (ERA_IO_LED_RED : 0x00, ERA_IO_LED_GREEN : 0x01, ERA_IO_LED_BLUE : 0x02)	Selection of the LED.
inf_flag= &inf_flag_enum inf_flag_enum = (INF_OFF : 0x00, INF_ON : 0x01)	Optional: Defines if the sequence will run infinitely. If set, when the last stage is finished, the reader will repeat the sequence from the beginning. Default value is off. If the parameter is not provided, the sequence is not repeated.
UpTime= 0 .. 500	For the beeper, the up_time parameter is ignored, as the change is always instant. For LEDs, an up_time of 0 means instantly turn on of the LED. If up_time is greater than 0, it causes LED brightening (from 0% to 100%) during this time.

	Values range: 0 to 500. Unit is 10 ms, so up_time of 50 means 0.5 s (500 ms).
UpSustainTime = 10 .. 655350, 0, 0xFFFF	The (high/on) state of I/O will be kept during this time and the next part of the sequence will start after this time. Values in range: 10 to 655350, with 0 and 0xFFFF as special values. Unit is 10 ms, so an up_sustain_time of 10 means 100 ms. Value of 0 (zero), when up_time also has value 0, means that the first part of the stage (switching to high/on state) will be skipped. In this case, the repeat_times parameter will also be ignored. Value of 0xFFFF: This value means that high/on state of I/O should be kept permanently, and all following parts of the sequence stage will be ignored. Configuration where both up_sustain_time and down_sustain_time have this value is considered invalid.
DownTime = 0 .. 500	For the beeper, the down_time parameter is ignored, as the change is always instant. For LEDs, a down_time of 0 means instantly turn off of the LED. When down_time is greater than 0, for LEDs it will cause LED fade-off (dimming) from 100% to 0% during this time. Values range: 0 to 1000. Unit is 10 ms, so a down_time of 100 means 1 s (1000 ms).
DownSustainTime= 10 .. 655350, 0, 0xFFFF	The (low/off) state of I/O will be kept during this time and the next part of the sequence (repetition or next stage) will start after this time. Values in range: 10 to 655350, with 0 and 0xFFFF as special values. Unit is 10 ms, so a down_sustain_time of 10 means 100 ms. Value of 0 (zero), when down_time is also 0, means that the second part of this stage (switching to high/on state) will be skipped. In this case, the repeat_times parameter is ignored. Value of 0xFFFF: This value means that the low/off state of I/O should be kept permanently, and all following parts of the sequence stage will be ignored. Configuration where both down_sustain_time and up_sustain_time have this value is considered invalid.
RepeatTimes = 1 .. 65535	The present (this) sequence stage will be repeated repeat_times times. Values range: 1 to 65535.

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.04 CHANGED
	5.03 SERVICE UNAVAILABLE
	Device is busy and cannot process the client's request.

data = none	4.00 BAD REQUEST	Incorrect param has been provided or CBOR structure is corrupted.
	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
	No response data is returned.	

Example

Example description	Full CoAP command	Arg / Res data
Command : Configure the I/O sequence described above	Hex: 4002F11EB6696F2D73657105736C6F7473FFA26130016131A2613082A361301864613100613283850000001832018500183200183201850018320019FFFF01A3613006131016132828518320018320002850019FFFF0000016131A2613000613181850018960019FFFF01 Decoded: 40 #CON, token len = 0 02 #POST F11E # MSG ID B6696F2D736571 #Option URI = io-seq 05736C6F7473 #Option URI = slots FF #End of options marker A26130016131A2613082A3613018646131613283850000001832018500183200003201850018320019FFF01A36130006131016132828518320010020002850019FFFF0000016131A2613000613181850018960019FFFF01	Hex: A26130016131A2613082A361301864613100613283850000001832018500183200183201850018320019FFFF01A36130006131016132828518320018320002850019FFFF0000016131A2613000613181850018960019FFFF01 Decoded: A2 #map 61 30 #ARG1: sequence_slot 01 #slot 1 61 31 #ARG2: sequences A2 #sequences map 61 30 #leds configuration 82 #leds array A3 #led 0 map 61 30 #start delay 18 64 #100 = 1000 ms 61 31 #led selection 00 #led 0 61 32 #sequence stages 83 #stages array 85 #stage 0 00 #up time 0 00 #up sustain time 0 00 #down time 0 18 32 #down sustain time 50 = 500 ms 01 #repeat times 1 85 #stage 1 00 #up time 0 18 32 #up sustain time 50 = 500 ms 00 #down time 0 18 32 #down sustain time 50 = 500 ms 01 #repeat times 1 85 #stage 2 00 #up time 0 18 32 #up sustain time 50 = 500 ms 00 #down time 0 19 FFFF #down sustain time forever 01 #repeat times 1 A3 #led 1 map 61 30 #start delay 00 #no delay 61 31 #led selection 01 #led 1 61 32 #sequence stage 82 #stages array 85 #stage 0 18 32 #up time 50 = 500 ms

		00	#up sustain time 0
		18 32	#down time 50 = 500 ms
		00	#down sustain time 0
		02	#repeat times 2
		85	#stage 1
		00	#up time 0
		19 FFFF	#up sustain time forever
		00	#down time 0
		00	#down sustain time 0
		01	#repeat times 1
		61 31	#beeper configuration
		A2	#beeper map
		61 30	#start delay
		00	#no delay
		61 31	#sequence stages
		81	#array(1)
		85	#stage 0
		00	#up time 0
		18 96	#up sustain time 150 = 1500 ms
		00	#down time 0
		19 FFFF	#down sustain time forever
		01	#repeat times 1
Response : OK	Hex: 6044F11E Decoded: 60	NONE	
	44		
	F11E		
Response : Wrong param value	Hex: 6080F11E Decoded: 60	NONE	
	80		
	F11E		

ioDeleteAudioVisualSequences

Deletes all audio-visual sequences (except the sequence in slot 0) and unbinds all sequences from events.

Command format

URI	/io-seq/slots
TYPE	CON
REQUEST CODE	DELETE
args = none	No arguments are returned.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.02 DELETED	
RESPONSE CODE POSSIBLE ERRORS	4.00 BAD REQUEST	Incorrect param has been provided or CBOR structure is corrupted.
	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
data = none	No response data is returned.	

Examples

Example description	Full CoAP command	Arg / Res data
Command: Deletes all sequences.	Hex: 40049E5BB6696F2D73657105736C6F7473	None
	Decoded: 40	
	04	
	9E5B	
	B6696F2D736571	
	05736C6F74	
Response: OK	Hex: 60429E5B	NONE
	Decoded: 60	
	42	
	9E5B	

ioBondSequenceToEvent (deprecated)

Command is deprecated and will be removed in the next release.

Bonds (pairs) a particular sequence to a system event, including event priority. When the particular event occurs, the paired sequence is executed. A meaningful error code is returned if:

- The selected event does not exist in the system.
- The slot number is incorrect.
- The selected priority is incorrect.

Command format

URI	/io-seq/events
TYPE	CON
REQUEST CODE	POST
args = { "0" : event_sel, ? "1" : sequence_and_priority }	Command arguments.
event_sel= &event_sel_enum event_sel_enum = (ERA_EVENT_START_UP : 0x00, ERA_EVENT_SHUTDOWN : 0x01, ERA_EVENT_CFG_STARTED : 0x02, ERA_EVENT_CFG_SUCCESS : 0x03, ERA_EVENT_CARD_DETECTED : 0x04, ERA_EVENT_CARD_READ_REJECTED : 0x05, ERA_EVENT_CARD_READ_REJECTED_CSN_READ_DONE : 0x06, ERA_EVENT_CARD_READ_DONE : 0x07, ERA_EVENT_GENERIC_ERROR : 0x08, ERA_EVENT_ATTACK_DETECTED : 0x09, ERA_EVENT_DATA_TRANSMISSION_FINISHED : 0x0A)	Event selection.
sequence_and_priority = bstr	AV sequence slot number (first element) and bonding priority (second element). If attribute is not present, the event binding is removed.
sequence = 0 .. 14	AV sequence slot number.
priority = 0 .. 4	Bonding priority (lower value means higher priority).

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.04 CHANGED
RESPONSE CODE POSSIBLE ERRORS	4.00 BAD REQUEST 5.03 SERVICE UNAVAILABLE 4.29 TOO MANY REQUESTS
	Incorrect param has been provided or CBOR structure is corrupted. Device is busy and cannot process the client's request. Previous command processing has not finished yet.
data = none	No response data is returned.

Examples

Example description	Full CoAP command		Arg / Res data	
Command: Bond event "SHUTDOWN" with sequence in slot 1. Priority is medium (level2).	Hex:		Hex:	
	4002F11EB6696F2D736571066576656E		A26130016131820102	
	7473FFA26130016131420102		Decoded:	
	Decoded:		A2	#map(2)
	40	#CON, token len = 0	61 30	#ARG1: Event
	02	#POST	01	#event (1)
	F11E	#MSG ID	61 31	#ARG2: sequence & priority
	B6696F2D736571	#Option URI = io-seq	42	#bytes(2)
	066576656E7473	#Option URI = events	01	#sequence slot nr 1
Response: OK.	FF	#End of options marker	02	#priority 2
	A26130016131420102	#Args		
	Hex:		NONE	
	6044F11E			
	Decoded:			
	60	#ACK, token len = 0		
	44	#CHANGED		
	F11E	#MSG ID		

ioAudioVisualSequenceExecute

Starts a previously defined (set) audio-visual sequence or stops the currently running sequence.

Command format

URI	/io-seq/state
TYPE	CON
REQUEST CODE	POST
args = { "0" : sequence_slot }	Command arguments.
sequence_slot = 0 .. 14	Number of slot for AV sequence to be started. Selection of slot 1 to 14 will start a particular sequence. If 0x00 is set, the predefined sequence is set, which turns off all I/O (equivalent to disabling the currently running sequence).

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.04 CHANGED	
RESPONSE CODE POSSIBLE ERRORS	4.00 BAD REQUEST	Incorrect param has been provided or CBOR structure is corrupted.
	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
data = none	No response data is returned.	

Examples

Example description	Full CoAP command		Arg / Res data	
Command: Starts the audio-visual sequence in slot number 5.	Hex:		Hex:	
	4002F11EB6696F2D7365710573746174		A1613005	
	65FFA1613005		Decoded:	
	Decoded:		A1	#map
	40	#CON, token len = 0	61 30	#ARG1:
	02	#POST		sequence_slot
	F11E	#MSG ID	05	#slot number 5
	B6696F2D736571	#Option URI = io-seq		
Command: Stop the currently running audio-visual sequence.	05736C6F74	#Option URI = state		
	FF	#End of options marker		
	A1613005	#Args		
	Hex:		Hex:	
	4002F11EB6696F2D7365710573746174		A1613000	
	65FFA1613000		Decoded:	
	Decoded:		A1	#map
	40	#CON, token len = 0		

Example description	Full CoAP command		Arg / Res data	
	02	#POST	61 30	#ARG1:
	F11E	#MSG ID		sequence_slot
	B6696F2D736571	#Option URI = io-seq	00	#slot number 0x00
	05736C6F74	#Option URI = state		
	FF	#End of options marker		
	A1613000	#Args		
Response: OK.	Hex:		NONE	
	6044F11E			
	Decoded:			
	60	#ACK, token len = 0		
	44	#CHANGED		
	F11E	#MSG ID		

ioGetIO

Gets the state of the selected I/O.

Command format

URI	/io
TYPE	CON
REQUEST CODE	GET
args = { "0" : io_sel }	Command arguments.
io_sel= &io_sel_enum io_sel_enum= (ERA_IO_SEL0 : 0x00, ERA_IO_SEL1 : 0x01, ERA_IO_SEL2 : 0x02)	Selection of the I/O.

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.05 CONTENT
RESPONSE CODE POSSIBLE ERRORS	4.00 BAD REQUEST Incorrect param has been provided or CBOR structure is corrupted.
	5.03 SERVICE UNAVAILABLE Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS Previous command processing has not finished yet.
data = { 0 : val_sel }	Response data.
val_sel = &val_sel_enum val_sel_enum = (ERA_IO_OFF : 0x00, ERA_IO_ON : 0x01)	Current value of selected I/O.

Examples

Example description	Full CoAP command	Arg / Res data
Command: Get the state of I/O1.	Hex: 4001F986B2696FFFA1613001 Decoded: 40 01 F986 B2696F	Hex: A1613001 Decoded: A1 61 30 01 #map #ARG1: io_sel #IO1
	#CON, token len = 0 #GET #MSG ID #Option URI = io	

Example description	Full CoAP command		Arg / Res data	
Response: OK.	FF	#End of options marker		
	A1613001	#Args		
	Hex:		Hex:	
	6045F986FFA1613001		A1613001	
	Decoded:		Decoded:	
	60	#ACK, token len = 0	A1	#map
	45	#CONTENT	61 30	#ARG1: val_sel
	F986	#MSG ID	01	#ON
Response: Wrong param value.	FF	#End of options marker		
	A1613001	#Data		
	Hex:		NONE	
	6080A12A			
	Decoded:			
	60	#ACK, token len = 0		
	80	#BAD REQUEST		
	F986	#MSG ID		

ioSetIO

Sets the state of the selected I/O.

Command format

URI	/io
TYPE	CON
REQUEST CODE	POST
args = { "0" : io_sel, "1" : val_sel, }	Command arguments.
io_sel= &io_sel_enum io_sel_enum= (ERA_IO_SEL0 : 0x00, ERA_IO_SEL1 : 0x01, ERA_IO_SEL2 : 0x02)	Selection of the I/O.
val_sel = &val_sel_enum val_sel_enum = (ERA_IO_OFF : 0x00, ERA_IO_ON : 0x01)	Argument specifies if I/O is turned on or off. ON == high.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.04 CHANGED	
RESPONSE CODE POSSIBLE ERRORS	4.00 BAD REQUEST	Incorrect param has been provided or CBOR structure is corrupted
	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request
	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
	4.12 PRECONDITION FAILED	I/O is not configured as output.
	5.00 INTERNAL SERVER ERROR	Failed to set I/O state.
data = none	No response data is returned.	

Examples

Example description	Full CoAP command		Arg / Res data	
Command: Set the state of I/O0 to ON (high).	Hex:		Hex:	
	4002A12BB2696FFFA2613000613101		A2613000613101	
	Decoded:		Decoded:	
	40	#CON, token len = 0	A2	#map
	02	#POST	61 30	#ARG1: io_sel

Example description	Full CoAP command		Arg / Res data	
	A12B	#MSG ID	00	#IO0
	B2696F	#Option URI = io	61 31	#ARG1: val_sel
	FF	#End of options marker	01	#Val to be set = ON
	A2613001613101	#Args		
Response: OK.	Hex: 6044A12A		NONE	
	Decoded: 60	#ACK, token len = 0		
	44	#CHANGED		
	A12B	#MSG ID		
Response: Wrong param value.	Hex: 6080A12A		NONE	
	Decoded: 60	#ACK, token len = 0		
	80	#BAD REQUEST		
	A12B	#MSG ID		

Card-specific commands

Card-specific commands allow you to use the external API to scan cards, read data from cards, write data to cards, authenticate, and configure the Card Processing Engine. The commands are grouped under one resource with the general command format shown below.

URI	/card
TYPE	CON
REQUEST CODE	POST
args = { "0" : action, ? "1" : scan_param / select_and_auth_param / read_data_param / write_data_param / value_op_param / terminate_param/ transceive_param }	Command arguments.
action = ((SCAN : 0x00) / (SELECT_AND_AUTH : 0x01) / (READ : 0x02) / (WRITE : 0x03) / (VALUE_OP : 0x04) / (TERMINATE : 0x05) / (READ_PACS : 0x06)/ (TRANSCIVE : 0x08))	RFID specific actions.

A particular RFID command is chosen by selecting the correct action parameter value and providing additional parameters, if required. See "rfidScan " on page 144 for details of the RFID commands.

rfidScan

Scans the field to detect a credential. The command scans the field according to the configuration. It is possible to scan credentials in any configuration.

The command reports the first credential detected to make the readout as fast as possible. This ensures the order of the protocols/modulations are selected correctly.

- HF credential found: the reader sends the type + CSN and keeps the RF field on for a predefined duration so that the host can send following commands, like select, authenticate, etc.

The timeout for sustaining the RF field can be set via configuration.

Command format

URI	/card
TYPE	CON
REQUEST CODE	POST
args = { "0" : action, "1" : scan_param }	Command arguments. If scan argument "0" is set to SCAN, argument "1" should contain data of type scan_param.
action = (SCAN : 0x00)	Scan action.
scan_param = { ? "0" : priority_sel, ? "1" : hf_proto_list, ? "2" : lf_mod_list }	Scan parameters.
hf_proto_list	Definition of HF protocol polling order configuration. It is the same string as used for CPE. For details, see "HF protocols configuration" on page 73.

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.04 CHANGED
RESPONSE CODE POSSIBLE ERRORS	4.12 PRECONDITION FAILED RFID Transparent Session has been started.
	5.03 SERVICE UNAVAILABLE Device is busy and cannot process the client's request.
	4.00 BAD REQUEST Incorrect param has been provided or CBOR structure is corrupted.
	5.00 INTERNAL SERVER ERROR No card has been found.
	4.29 TOO MANY REQUESTS Previous scan command has not finished yet.
data = { ? "0" : type_sel, ? "1" : rfid_csn }	Response data. Data is returned ONLY if a card has been discovered.
type_sel = See "Card types" on page 92.	Optional: Detected card type if status is ERA_STATUS_OK.
rfid_csn = bstr	Optional: Read card CSN.

Examples

Example description	Full CoAP command	Arg / Res data
Command: Request to scan RF field.	Hex: 40020271B463617264FFA26130 006131A161318400020301 Decoded: 40 #CON, token len = 0 02 #POST 0271 #MSG ID B463617264 #Option URI = card FF #End of options marker A26130006131A161 #Args 318400020301	Hex A26130006131A161318400020301 Decoded: A2 #map 61 30 #ARG1: action 00 #scan 61 31 #ARG2: scan_param A2 #map 61 31 #ARG1: hf_proto_list 8400020301 #bstr indicating following protocols: ECP, ISO14443B, ISO15694, ISO14443A. iCLASS is turned off
Response: MIFARE Classic card detected.	Hex: 60450271FFA2613001613144AABBCCDD Decoded: 60 #ACK, token len = 0 45 #CONTENT 0271 #MSG ID FF #End of options marker A2613001613144 #Data AABBCCDD	Hex: A2613001613144AABBCCDD Decoded: A2 #map 61 30 #ARG1: type_sel 01 #MIFARE Classic 1K card 61 31 #ARG2: csn 44 #bstr AABBCCDD #CSN
Response: No card found. Works in any mode.	Hex: 60A00271 Decoded: 60 #ACK, token len = 0 A0 #INTERNAL ERROR 0271 #MSG ID	NONE

rfidSelectAndAuthenticate

Selects a data container on a previously detected card (see "rfidScan " on page 144) and optionally authenticates it, all in a single request. The command parameter will differ based on card type, for example, MIFARE Classic or iCLASS.

Command format

URI	/card
TYPE	CON
REQUEST CODE	POST
args = { "0" : action, "1" : select_and_auth_param }	Command arguments. For select and authenticate, argument "0" shall be set to SELECT_AND_AUTH and argument "1" should contain data of type select_and_auth_param.
action = ((SELECT_AND_AUTH : 0x01))	Select and authenticate action.
select_and_auth_param = { ? "0" : authentication_action_data }	
authentication_action_data = bstr	Optional parameter. See "Card data procedures" on page 76 for the appropriate authentication structure based on a card type.

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.04 CHANGED
RESPONSE CODE POSSIBLE ERRORS	4.12 PRECONDITION FAILED 5.00 INTERNAL SERVER ERROR 4.00 BAD REQUEST 5.03 SERVICE UNAVAILABLE 4.29 TOO MANY REQUESTS
	Scan has not been performed (no card to be selected) or RFID Transparent Session has been started. Card could not be selected and/or authenticated and/or is no longer available. Incorrect param has been provided or CBOR structure is corrupted. Device is busy and cannot process the client's request. Previous command processing has not finished yet.
data = none	

Examples

Example description	Full CoAP command	Arg / Res data
Command: Select and Authenticate to PicoPass card Book 2 Page 0.	Hex: 40020271B463617264FFA26130016131A1613050A4613000613143FFFFFFF61322613300 Decoded: 40#CON, token len = 0	Hex: A26130016131A1613050A4613000613143FFFFFFF613202613300 Decoded: A2#map(2) 61 30#ARG1: action

Example description	Full CoAP command	Arg / Res data
	02 #POST	01 #SELECT_AND_AUTH
	0271 #MSG ID	H
	B463617264 #Option URI = card	61 31 #ARG2:
	FF #End of options marker	select_and_auth_param
	A26130A26130016 #Args	A1 #map(1)
	131A1613050A461	61 30 #ARG1:
	3000613143FFFF	authentication_action_data
	F613202613300	
		50
		A4613000613143FFF #authentication data
		FFF
		613202613300
		Decoded authentication data:
		A4 #map(4)
		61 30 #ARG1: key_type
		00 #unsigned(0)
		61 31 #ARG2: key_oid
		43 #bytes(3)
		FFFFFF
		61 32 #ARG3: book_sel
		02 #unsigned(2)
		61 33 #ARG4: page_sel
		00 #unsigned(0)

rfidReadData

Reads data from the card. Data is read according to the configuration stored in the `data_read_action` parameter. This parameter will differ based on card type. Before this command can be used, you must execute the "rfidScan " on page 144 and "rfidSelectAndAuthenticate " on page 146 commands successfully, otherwise error 4.12 PRECONDITION FAILED will be returned.

Command format

URI	/card
TYPE	CON
REQUEST CODE	POST
args = { "0" : action, "1" : read_data_param }	Command arguments. If read data argument "0" is set to READ, argument "1" should contain data of type read_data_param.
action = (READ : 0x02)	Read action
read_data_param = { "0" : data_read_action ?"1" : bitwise_config }	See "Card data procedures" on page 76 for the appropriate data read structure based on a card type. If bitwise is used for readout which needs sending multiple commands (chaining) operation failure error will be raised.

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.04 CHANGED
RESPONSE CODE POSSIBLE ERRORS	4.12 PRECONDITION FAILED Card has not yet been selected and/or authenticated or RFID Transparent Session has been started.
	5.00 INTERNAL SERVER ERROR Selected resource could not be read from the card.
	4.00 BAD REQUEST Incorrect param has been provided or CBOR structure is corrupted.
	5.03 SERVICE UNAVAILABLE Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS Previous command processing has not finished yet.
data = { ? "0" : bstr ; payload }	payload: Data read from the card, present only if status is 2.04 CHANGED .

Examples

Example description	Full CoAP command	Arg / Res data
Command: Read block data from a PicoPass card.	Hex: 40020271B463617264FFA2613002 6131A1613047A2613001613101 Decoded:	Hex: A26130026131A1613047A2613001613101 Decoded: A2 #map(2)

Example description	Full CoAP command	Arg / Res data
	40 #CON, token len = 0	61 30 #ARG1: action
	02 #POST	02 #READ
	0271 #MSG ID	61 31 #ARG2:
	B463617264 #Option URI = card	read_data_param
	FF #End of options marker	A1 #map(1)
	A26130026131A1613 #Args	61 30 #ARG1:
	047A2613001613101	data_read_action
		47 #bytes(7)
		A2613001613101 #read action data
		Decoded Data Read Action:
		A2 #map(2)
		61 30 #ARG1: block_sel
		01 #starting block number: 1
		61 31 #ARG2: block_num
		01 #number of subsequent blocks: 1

rfidWriteData

Writes data to the card. Data is written according to the configuration stored in data_action parameter. This parameter will differ based on card type. Before this command can be used, you must execute the "rfidScan " on page 144 and "rfidSelectAndAuthenticate " on page 146 commands successfully, otherwise error 4.12 PRECONDITION FAILED will be returned.

rfidWriteData can be used for writing data into existing files on a card. To create a new file, a separate command must be used first (not implemented yet).

Command format

URI	/card
TYPE	CON
REQUEST CODE	POST
args = { "0" : action, "1" : write_data_param }	Command arguments. If write data argument "0" is set to WRITE, argument "1" should contain data of type write_data_param.
action = ((WRITE : 0x03))	Write action.
write_data_param = { "0" : data_action "1" : payload }	
data_action	See "Card Action structure" on page 78 for the appropriate data write structure based on a card type.
payload = bstr	Data to be written on the card.

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.04 CHANGED
RESPONSE CODE POSSIBLE ERRORS	4.12 PRECONDITION FAILED Card has not yet been selected and/or authenticated or RFID Transparent Session has been started.
	5.00 INTERNAL SERVER ERROR Could not write to the selected resource.
	4.00 BAD REQUEST Incorrect param has been provided or CBOR structure is corrupted.
	5.03 SERVICE UNAVAILABLE Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS Previous command processing has not finished yet.
data = none	

Examples

Example description	Full CoAP command	Arg / Res data
Command:	Hex:	Hex:

Example description	Full CoAP command	Arg / Res data
Write 0102030405060708 to block 1 on a PicoPass card.	40020271B463617264FFA26130036131A261	A26130036131A2613044A16130016131
	1	480102030405060000
	3044A16130016131480102030405060708	Decoded:
	Decoded:	A2 #map(2)
	40 #CON, token len = 0	61 30 #ARG1: action
	02 #POST	03 #WRITE
	0271 #MSG ID	61 31 #ARG2:
	B463617264 #Option URI = card	write_data_param
	FF #End of options marker	A2 #map(2)
	A26130036131A261 #Args	61 30 #ARG1: data_action
	3044A16130016131	44 #bytes(4)
	48010203040506070	A1613001
	8	61 31 #ARG2: payload
		48 #bytes(8)
		0102030405060708
		Decoded data_action:
		A1 #map(1)
		61 30 #ARG1: block_sel
		01 #1

rfidValueOperation

This command can be used with MIFARE Classic cards and works in combination with two MIFARE commands:

- MIFARE Increment/Decrement/Restore: Performs a read from the src block, increments, decrements, or restores the data, and stores it in the transfer buffer.
- MIFARE Transfer: Writes the data in the transfer buffer to the destination block.

Command format

URI	/card
TYPE	CON
REQUEST CODE	POST
args = { "0" : action, "1" : value_op_param }	Command arguments. If value operation argument "0" is set to VALUE_OP, argument "1" should contain data of type value_op_param.
action = (VALUE_OP : 0x04)	Value operation action
value_op_param= { "0" : type, "1" : src_block, "2" : dst_block, "3" : value }	Value operation parameters.
type = &type_enum type_enum = (OP_INCREASE = 0x00, OP_DECREASE = 0x01, OP_RESTORE = 0x02)	
src_block = int	Source block address, 00h to FFh.
dst_block = int	Destination block address, 00h to FFh.
value	4-bytes operand value.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.04 CHANGED	
RESPONSE CODE POSSIBLE ERRORS	4.12 PRECONDITION FAILED	Card has not yet been selected and/or authenticated or RFID Transparent Session has been started.
	5.00 INTERNAL SERVER ERROR	Could not update the selected resource on the card.
	4.00 BAD REQUEST	Incorrect param has been provided or CBOR structure is corrupted.
	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
data = none		

Examples

Example description	Full CoAP command		Arg / Res data	
Command: Decrement value from block 4 by 0000000Ah and write the result to block 5.	Hex:		Hex:	
	40020271B463617264FFA26130046131A4		A26130046131A4613001613104613205	
	6130016131046132056133440000000A		61334400000000A	
	Decoded:		Decoded:	
	40	#CON, token len = 0	A2	#map(2)
	02	#POST	61 30	#ARG1: action
	0271	#MSG ID	04	#VALUE_OP
	B463617264	#Option URI = card	61 31	#ARG2:
	FF	#End of options marker		value_op_param
			A4	#map(4)
	A261300401A46130	#Args	61 30	#ARG1: type
	0161310461320561		01	#OP_DECREASE
	33440000000A		61 31	#ARG2: src_block
			04	#source block: 4
			61 32	#ARG3: dst_block
			05	#destination block: 5
			61 33	#ARG4: value
			44	#bytes(4)
			0000000A	#value

rfidTerminate

Terminates communication with the card by:

- Turning off the RFID field.
- Forgetting all card information within the app (CSN, card type) and releasing resources.

Command format

URI	/card
TYPE	CON
REQUEST CODE	POST
args = { "0" : action, "1" : terminate_param }	Command arguments. If terminate argument "0" is set to TERMINATE, argument "1" should omitted.
action = ((TERMINATE : 0x05))	Terminate action

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.04 CHANGED	
RESPONSE CODE POSSIBLE ERRORS	4.12 PRECONDITION FAILED	There is no communication with a card which could be terminated, or RFID Transparent Session has been started.
	4.00 BAD REQUEST	Incorrect param has been provided or CBOR structure is corrupted.
	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
data = none		

Examples

Example description	Full CoAP command	Arg / Res data
Command: Terminate.	Hex: 40020271B463617264FFA1613005 Decoded: 40 #CON, token len = 0 02 #POST 0271 #MSG ID B463617264 #Option URI = card FF #End of options marker A1613005 #Args	Hex: A1613005 Decoded: A1 #map(1) 61 30 #ARG1: action 05 #TERMINATE
Response: OK.	Hex: 60440271 Decoded: 60 #ACK, token len = 0	NONE

Example description	Full CoAP command	Arg / Res data
44	#CHANGED	
0271	#MSG ID	

rfidReadPacs

Reads PACS bits on a previously detected credential. The command returns the appropriate status if PACS bits are not present.

Command format

URI	/card
TYPE	CON
REQUEST CODE	POST
args = { "0" : action, "1" : read_pacs_param }	Command arguments. If the read PACS argument "0" is set to READ_PACS, argument "1" should be omitted.
action = (READ_PACS : 0x06)	Read PACS.
read_pacs_param	Not used at this time. RFU.

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.04 CHANGED
RESPONSE CODE POSSIBLE ERRORS	4.12 PRECONDITION FAILED Card has not yet been selected and/or authenticated or RFID Transparent Session has been started
	5.00 INTERNAL SERVER ERROR Could not read PACS data from the card.
	4.00 BAD REQUEST Incorrect param has been provided or CBOR structure is corrupted
	5.03 SERVICE UNAVAILABLE Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS Previous command processing has not finished yet.
data = { ? "0" : bstr ; payload }	payload: PACS data, if present, and with 2.04 CHANGED status.

Examples

Example description	Full CoAP command	Arg / Res data
Command: Read PACS	Hex: 40020271B463617264FFA1613006 Decoded: 40 02 0271 B463617264 FF A1613006	Hex: A1613006 Decoded: A1 61 30 06 #map(1) #ARG1: action #READ_PACS
	#CON, token len = 0	
	#POST	
	#MSG ID	
	#Option URI = card	
	#End of options marker	
	#Args	

rfidTransceive

Performs a transceive operation with a smart card.

Command format

URI	/card
TYPE	CON
REQUEST CODE	POST
args = { "0" : action, "1" : command }	Command arguments. For read data argument "0" shall be set to TRANSCEIVE and argument "1" should contain command data to be sent to card.
action = (TRANSCEIVE : 0x08)	Read action.
command = { "0" : command data ? "1" : chaining }	Card command data to be sent to card (max 512B). Chaining is not supported.

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.04 CHANGED
RESPONSE CODE POSSIBLE ERRORS	4.12 PRECONDITION FAILED 4.00 BAD REQUEST 5.03 SERVICE UNAVAILABLE 4.29 TOO MANY REQUESTS
	There is no communication with card which could be terminated or RFID Transparent Session has been started. Incorrect param has been provided or CBOR structure is corrupted Device is busy and cannot process the client's request. Previous command processing has not finished yet.
data = none	

Examples

Example description	Full command	Arg / Res data
Command: Mifare DESFire Get Version	Hex: 40020271B463617264FFA2613008 6131a16130459060000000 Decoded: 40 # CON, token len = 0 02 # POST 0271 # MSG ID B463617264 # Option URI= card FF # End of options marker a26130086131a16130459060000000 # Args	Hex: A26130086131A16130459060000000 Decoded A2 # map(2) 6130 # ARG1: action 08 # TRANSCEIVE 61 31 # ARG2: read_data_param A1 # map(1) 61 30 # ARG1: data_read_action 45 # bytes(5) 9060000000 # Card command data

Example description	Full command	Arg / Res data
Response: First chunk of Mifare DESFire version data	Hex: 60440271ffa161304904010101001a0591af Decoded: 60 # ACK, token len = 0 44 # CONTENT 0271 # MSG ID FF # End of options marker a161304904010101001a0591af# Args	Hex: a161304904010101001a0591af Decoded A1 # map(2) 61 30 # ARG1: action 49 # bytes (9) 04010101001a0591af # Card response data

rfidCardProcessingEngineGetFreeSlot (deprecated)

Command is deprecated and will be removed in the next release.

Gets the first free slot (number) in the procedures list. The slot can be set later by:

- An "rfidCardProcessingEngineSetSlot (deprecated) " on page **Error! Bookmark not defined.** command using a slot number.
- A "configSetParam " on page 185 using the slot's assigned OID.

Command format

URI	/cpe/slot
TYPE	CON
REQUEST CODE	GET
args = { "0" : rfid_tech_sel }	Command arguments.
rfid_tech_sel = &rfid_tech_sel_enum rfid_tech_sel_enum = (LF: 0x00, HF: 0x01)	Selection of the RFID technology. Determines if a free slot in the LF or HF procedures list will be returned.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.05 CONTENT	
RESPONSE CODE POSSIBLE ERRORS	5.01 NOT IMPLEMENTED	Feature is not yet implemented.
	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	5.00 INTERNAL SERVER ERROR	No free slots.
	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
	4.00 BAD REQUEST	Incorrect param has been provided or CBOR structure is corrupted.
data = { "0" : slot_number }	Response data.	
slot_number = uint	Free slot number.	

Examples

Example description	Full CoAP command	Arg / Res data
Command:	Hex:	Hex:
Get the first free slot in the HF procedures list.	40013909B363706504736C6F74FFA1613001	A1613001
	Decoded:	Decoded:
	40	A1
	#CON, token len = 0	#map
	01	61 30
	#GET	#ARG1: RFID tech selection

Example description	Full CoAP command		Arg / Res data	
	3909	#MSG ID	01	#HF selection
	B3637065	#Option URI = cpe		
	04736C6F	#Option URI = slot		
	FF	#End of options marker		
	A2613001613101	#Args		
Response: Free slot number = 1.	Hex: 60453909FFA1613001		Hex: A1613001	
	Decoded: 60	#ACK, token len = 0	A1	#map
	45	#CONTENT	61 30	#ARG1: slot number
	3909	#MSG ID	01	#1
	FF	#End of options marker		
	A1613001	#Data		

Response: OK.	Hex:	NONE	
	60443909		
	Decoded:		
	60	#ACK, token len = 0	
	44	#CHANGED	
	3909	#MSG ID	

rfidCardProcessingEngineClearProcedures

Deletes all procedures from a selected technology list.

Command format

URI	/cpe/slots
TYPE	CON
REQUEST CODE	DELETE
args = { "0" : rfid_tech_sel }	Command arguments.
rfid_tech_sel = &rfid_tech_sel_enum rfid_tech_sel_enum = (LF: 0x00, HF: 0x01)	Selection of the RFID technology. Determines if the LF or HF list will be cleared. Currently, only the HF procedure list is supported.

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.02 DELETED
RESPONSE CODE POSSIBLE ERRORS	4.00 BAD REQUEST Incorrect param has been provided or CBOR structure is corrupted.
	5.03 SERVICE UNAVAILABLE Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS Previous command processing has not finished yet.
data = none	No response data is returned.

Examples

Example description	Full CoAP command	Arg / Res data
Command: Clear all procedures in the HF procedures list.	Hex: 40043909B363706505736 C6F7473FFA1613001 Decoded: 40 04 3909 B3637065 05736C6F7473 FF A1613001 #CON, token len = 0 #DELETE #MSG ID #Option URI = cpe #Option URI = slots #End of options marker #Args	Hex: A1613001 Decoded: A1 61 30 01 #map #ARG1: RFID tech selection #HF sel
Response: OK.	Hex: 60429E5B Decoded: 60 #ACK, token len = 0	None

Example description	Full CoAP command	Arg / Res data
42	#DELETED	
9E5B	#MSG ID	

Key management commands

Key management commands are provided for managing user keys. They include commands for rolling wrapping keys to ensure enhanced security across the system.

rollDefaultWrappingKey

Re-rolls the default AES KEK used for encryption of custom keys (including other AES KEKs). The new key provided in the argument must be wrapped using the default AES KEK. Command wrapping is performed according to [NIST.SP.800-38F](#).

This command must be used if you want to inject any custom key. Until this command is called, the reader will not accept any other custom key. Due to limited authenticity protection, a failure counter is implemented (feature configurable in SE). After 255 unsuccessful injections of a new key, the reader will block any further attempts. In this situation, it will be necessary to contact HID. The failure counter is cleared after injection of new key material. Injection of an old key will not clear the counter. The OID of the default AES KEK is stated in "AES KEKs OIDs " on page 274.

Command format

URI	/wrap-key
TYPE	CON
REQUEST CODE	POST
args = { "0" : key_material, ? "1" : key_id }	Command arguments.
key_material = bstr	New AES KEK wrapped using the default AES KEK.
key_id = bstr .size 8	8 bytes of key ID

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.04 CHANGED
RESPONSE CODE POSSIBLE ERRORS	5.03 SERVICE UNAVAILABLE Device is busy and cannot process the client's request.
	4.00 BAD REQUEST Incorrect param has been provided or CBOR structure is corrupted.
	4.29 TOO MANY REQUESTS Previous command processing has not finished yet.
data = none	No response data is returned.

Examples

Example description	Full CoAP command	Arg / Res data
Command: Change default AES KEK to a new value. Encrypted key = 0x0102030405060708.	Hex: 4002F88DB8777261702D6B6579 FFA16130480102030405060708 Decoded: 40 02 F88D B8777261702D6B6579 9 FF #CON, token len = 0 #POST #MSG ID #Option URI = wrap-key #End of options marker	Hex: A16130480102030405060708 Decoded: A1 61 30 48 0102030405060708 #map #ARG1: Wrapped new AES KEK key #bstr #Wrapped key

Example description	Full CoAP command		Arg / Res data
Response: OK.	A16130480102		NONE
	030405060708	#Args	
	Hex:		
	6044F88D		
	Decoded:		
	60	#ACK, token len = 0	
	44	#CHANGED	
	F88D	#MSG ID	

setUserKey

Inserts a new RFID user key. The new key must be encrypted using an existing key. There is no possibility to insert a new key in plain text. Command wrapping is performed done according to [NIST.SP.800-38F](#).

Due to limited authenticity protection, a failure counter is implemented. After two hundred and fifty five (255) unsuccessful injections of a new key, the reader will block any further attempts. In this situation, it will be necessary to contact HID. The failure counter is cleared after injection of new key material. Injection of an old key will not clear the counter.

Before this command can be used, the default AES KEK key must be rolled using "rollDefaultWrappingKey " on page 165. OIDs for new keys/wrapping keys are specified in "AES KEKs OIDs " on page 274.

Command format

URI	/key
TYPE	CON
REQUEST CODE	POST
args = user_key_package	Command arguments. See "User Key encoding " on page 275.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.04 CHANGED	
RESPONSE CODE POSSIBLE ERRORS	5.00 INTERNAL SERVER ERROR	Key to be used for key injection has not yet been uploaded.
	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.12 PRECONDITION FAILED	Default key (KEK) has not been changed yet.
	4.00 BAD REQUEST	Incorrect param has been provided or CBOR structure is corrupted.
	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
data = none	No response data is returned.	

Examples

Example description	Full CoAP command		Arg / Res data	
Command: Insert a new key and set its key reference to 030101. Use an already existing key (key ref: 030102) to decrypt the key. Encrypted key = 0x0102030405060708.	Hex:		Hex:	
	4002F88DB36B6579FFA361304303010161		A3613043030101613143030102613248	
	31430301026132480102030405060708		0102030405060708	
	Decoded:		Decoded:	
	40	#CON, token len = 0	A3	#map
	02	#POST	61 30	#ARG1: New key ref
	F88D	#MSG ID	43	#bstr
	B36B6579	#Option URI = key	030101	#New key ref
	FF	#End of options marker	61 31	#ARG2: Decryption key OID
	A361304303010161	#Args	43	#bytes

Example description	Full CoAP command		Arg / Res data	
Response: OK.	3143030102613248		030102	#Decryption key ref
	0102030405060708		61 32	#ARG3: Encrypted key
			48	#bytes
			0102030405060708	#Encrypted key
	Hex: 6044F88D Decoded: 60 44 F88D	#ACK, token len = 0 #CHANGED #MSG ID	NONE	

getUserKeyInfo

Retrieves information about the user's key. The function returns the type of key (AES KEK/media key) and the ID of the key. The key ID is owned by the customer and can be optionally included during the key injection. If the key has not been inserted yet, 5.00 INTERNAL SERVER ERROR is returned. If you have not set the key ID, then no key ID is returned.

Command format

URI	/key
TYPE	CON
REQUEST CODE	GET
args = { "0" : key_ref }	Command arguments.
key_ref = bstr .size 3	Key ref of the key to be verified.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.05 CONTENT	
RESPONSE CODE POSSIBLE ERRORS	5.00 INTERNAL SERVER ERROR	Key has not yet been uploaded.
	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.12 PRECONDITION FAILED	Default key (KEK) has not been changed yet.
	4.00 BAD REQUEST	Incorrect param has been provided or CBOR structure is corrupted.
	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
data = { "0" : key_type, ? "1" : key_id }	Response data.	
key_type = &key_type_enum key_type_enum= (ERA_KEY_KEK : 0x00, ERA_KEY_MEDIA : 0x01)	Key type.	
key_id = bstr .size 8	8 bytes of key ID.	

Examples

Example description	Full CoAP command		Arg / Res data	
Command: Get info on key with ref = 030101.	Hex:		Hex:	
	4001F88DB36B6579FFA161304330101		A1613043030101	
	Decoded:		Decoded:	
	40	#CON, token len = 0	A1	#map
	01	#GET	61 30	#ARG1: Key reference

Example description	Full CoAP command		Arg / Res data	
Response: OK.	F88D	#MSG ID	43	#bytes
	B36B6579	#Option URI = key	030101	#Key ref
	FF	#End of options marker		
	A1613043030101	#Args		
	Hex:		Hex:	
	6045F88DFFA261300061314801		A26130006131480102030405060708	
	02030405060708		Decoded:	
	Decoded:		A2	#map
	60	#ACK, token len = 0	61 30	#ARG1: Key type
	45	#CONTENT	00	#AES KEK
	F88D	#MSG ID	61 31	#ARG1: Key ID
	FF	#End of options marker	48	#bytes
	A2613000613148	#Data	0102030405060708	#Key ID
	0102030405060708			

Transparent session commands

A transparent session allows data to be passed transparently from an application to a contactless ICC, and return received data transparently back to the application. Using exposed commands, it is possible to configure protocol settings and transceive parameters. The following commands are available for transparent session mode:

- rfidTransparentSessionStart
- rfidTransparentSessionStop
- rfidTransparentSessionConfigure
- rfidTransparentSessionSetField
- rfidTransparentSessionTransceive

The commands are grouped under one command with a standard format.

General command format

URI	/card/ts
TYPE	CON
REQUEST CODE	POST
args = { "0" : action ? "1" : configuration / field_command / transceive_command }	Command arguments.
action = (START: 0x00, STOP: 0x01, CONFIGURE: 0x02, SET_FIELD: 0x03, TRANSCIVE: 0x04)	Transparent session-specific actions.

The required transparent session command is chosen by selecting the correct action parameter value and providing additional parameters, if required. See sections "rfidTransparentSessionStart " on page 172 to "rfidTransparentSessionTransceive " on page 178 for details.

rfidTransparentSessionStart

Starts a transparent session. Once the transparent session is activated, all other card commands, except for transparent session related commands, are blocked.

Command format

URI	/card/ts
TYPE	CON
REQUEST CODE	POST
args = { "0" : action }	Command arguments.
Command arguments. action = (START : 0x00))	Transparent session specific actions.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.04 CHANGED	
RESPONSE CODE POSSIBLE ERRORS	4.12 PRECONDITION FAILED	Transparent session is already started.
	4.00 BAD REQUEST	Incorrect param has been provided or CBOR structure is corrupted.
	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
data = { "0" : uint ; reason }	The reason provides more detailed information on the communication status with the card. Possible values are: reason = { (SUCCESS : 0x00) / (INVALID_STATE : 0x01) / (INVALID_PARAM : 0x02) / (BUFFER_OVERFLOW : 0x03) / (INTEGRITY : 0x04) / (TIMEOUT : 0x05) / (NOT_IMPLEMENTED : 0x06) / (UNDEFINED : 0x07) }	

Examples

Example description	Full CoAP command	Arg / Res data
Command: Request to start a transparent session.	Hex: 40020271B4636172640274 73FFA2613000 Decoded:	Hex: A1613000 Decoded: A1 #map

Example description	Full CoAP command		Arg / Res data	
	40	#CON, token len = 0	61 30	#action
	02	#POST	00	#transparent_session_start
	0271	#MSG ID		
	B463617264	#Option URI = card		
	027473	#Option URI = ts		
	FF	#End of options marker		
	A1613000	#Args		
Response: OK.	Hex:		A1613000	
	60440271FFA1613000		Decoded:	
	Decoded:		A1	#map
	60	#ACK, token len = 0	61 30	#ARG1: reason
	44	#CHANGED	00	#SUCCESS
	0271	#MSG ID		
	FF	#End of options marker		
	A1613000	#Data		

rfidTransparentSessionStop

Stops a transparent session.

Command format

URI	/card/ts
TYPE	CON
REQUEST CODE	POST
args = { "0" : action }	Command arguments.
Command arguments. action = (STOP : 0x01)	Transparent session specific actions.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.04 CHANGED	
RESPONSE CODE POSSIBLE ERRORS	4.12 PRECONDITION FAILED	Transparent session has not yet been started.
	4.00 BAD REQUEST	Incorrect param has been provided or CBOR structure is corrupted.
	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
data = { "0" : uint ; reason }	Reason provides more detailed information on the communication status with the card. Possible values are: reason = { (SUCCESS : 0x00) / (INVALID_STATE : 0x01) / (INVALID_PARAM : 0x02) / (BUFFER_OVERFLOW : 0x03) / (INTEGRITY : 0x04) / (TIMEOUT : 0x05) / (NOT_IMPLEMENTED : 0x06) / (UNDEFINED : 0x07) }	

Examples

Example description	Full CoAP command		Arg / Res data	
Command: Request to stop the transparent session.	Hex:		Hex:	
	40020271B4636172640274		A1613001	
	73FFA2613001		Decoded:	
	Decoded: 40	#CON, token len = 0	A1 61 30	#map #action

Example description	Full CoAP command		Arg / Res data	
	02	#POST	01	#transparent_session_stop
	0271	#MSG ID		
	B463617264	#Option URI = card		
	027473	#Option URI = ts		
	FF	#End of options marker		
	A1613001	#Args		
Response: OK.	Hex:		A1613000	
	60440271FFA1613000		Decoded:	
	Decoded:		A1	#map
	60	#ACK, token len = 0	61 30	#ARG1: reason
	44	#CHANGED	00	#SUCCESS
	0271	#MSG ID		
	FF	#End of options marker		
	A1613000	#Data		

rfidTransparentSessionConfigure

Configures the previously started transparent session (started using a "rfidTransparentSessionStart " on page 172 command).

Command format

URI	/card/ts
TYPE	CON
REQUEST CODE	POST
args = { "0" : action, "1" : configuration }	Command arguments. For session configuration, argument "1" should contain data of type configuration.
action = (CONFIGURE : 0x02)	Transparent session configuration action.
configuration = { "0": protocol, "1": baudrate }	
protocol = &protocol_enum protocol_enum = (ISO14443A: 0x00, ISO14443B: 0x01, ISO15693: 0x02, iClass15693: 0x03, FeliCa: 0x04)	Protocol type.
baudrate= &baudrate_enum baudrate_enum = (BAUDRATE_106: 0x00, BAUDRATE_212: 0x01, BAUDRATE_424: 0x02, BAUDRATE_848: 0x03)	Communication speed (baud rate).

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.04 CHANGED	
RESPONSE CODE POSSIBLE ERRORS	4.00 BAD REQUEST	Incorrect param has been provided or CBOR structure is corrupted.
	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the clients request.
	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
data = { "0" : uint ; reason }	The reason provides more detailed information on the communication status with the card. Possible values are: reason = { (SUCCESS : 0x00) / (INVALID STATE : 0x01) /	

	(INVALID_PARAM : 0x02) / (BUFFER_OVERFLOW : 0x03) / (INTEGRITY : 0x04) / (TIMEOUT : 0x05) / (NOT_IMPLEMENTED : 0x06) / (UNDEFINED : 0x07) }
--	---

Examples

Example description	Full CoAP command	Arg / Res data
Command: Request to configure TS	Hex: 40020271B463617264027473FF A26130026131A2613000613100 Decoded: 40 #CON, token len = 0 02 #POST 0271 #MSG ID B463617264 #Option URI = card 027473 #Option URI = ts FF #End of options marker A26130026131 #Args A2613000613100	Hex: A26130026131A2613000613100 Decoded: A2 #map 61 30 #action 02 #transparent_session_configure 61 31 #configuration A2 #map 61 30 #protocol 00 #iso14443a 61 31 #baudrate 00 #106 kbit/s
Response: OK	Hex: 60440271FFA1613000 Decoded: 60 #ACK, token len = 0 44 #CHANGED 0271 #MSG ID FF #End of options marker A1613000 #Data	A1613000 Decoded: A1 #map 61 30 #ARG1: reason 00 #SUCCESS

rfidTransparentSessionTransceive

Transceives data in a transparent session.

Command format

URI	/card/ts
TYPE	CON
REQUEST CODE	POST
args = { "0" : action, "1" : transceive_command }	Command arguments. For the session transceive argument, "1" should contain data of type transceive_command.
action = (TRANSCIVE : 0x04)	Transparent session transceive action.
transceive_command = { "0": rx_settings, "1": tx_settings, "2": 1..100000 ; timeout microseconds "3": bstr ; data }	Command-specific arguments.
rx_settings = { "0" : bool, ; CRC enabled "1" : bool, ; parity enabled "2" : bool ; emd_suppression_en }	Rx settings.
tx_settings = { "0" : bool, ; CRC enabled "1" : bool, ; parity enabled "2" : 0..7 ; last byte valid bits num }	Tx settings.

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.04 CHANGED
RESPONSE CODE POSSIBLE ERRORS	4.00 BAD REQUEST Incorrect param has been provided or CBOR structure is corrupted.
	5.03 SERVICE UNAVAILABLE Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS Previous command processing has not finished yet.
	5.00 INTERNAL SERVER ERROR Card has not responded / there is no card in the field.
data = { "0" : uint, ; reason ? "1" : bstr ; rx_data }	Reason and response from card (if any). The reason provides more detailed information on the communication status with the card. Possible values are: reason = { (SUCCESS : 0x00) / (INVALID_STATE : 0x01) / (INVALID_PARAM : 0x02) /

```
(BUFFER_OVERFLOW : 0x03) /
(INTEGRITY : 0x04) /
(TIMEOUT : 0x05) /
(NOT_IMPLEMENTED : 0x06) /
(UNDEFINED : 0x07)
}
```

Examples

Example description	Full CoAP command	Arg / Res data
Command: Request to perform a transceive operation in TS.	Hex: 40020271B46361726402 7473FFA26130046131A4 6130A36130F46131F561 32F56131A36130F46131 F5613207613218466133 4126 Decoded: 40 02 0271 B463617264 027473 FF A26130046131A4 6130A36130F461 31F56132F56131 A36130 F46131F 56132076132184 661334126	Hex: A26130046131A46130A36130F46131F5 6132F56131A36130F46131F561320761 32184661334126 Decoded: A2 6130 04 6131 A4 6130 A3 6130 F4 6131 F5 6134 F5 6131 A3 6130 F4 6131 F5 6132 07 6132 18 46 6133 41 26
	#CON, token len = 0 #POST #MSG ID #Option URI = card #Option URI = ts #End of options marker #Args	#map #action #transparent_session_transceive #transceive_command #map #rx_settings #map #crc_en #false #parity_en #true #emd_suppression_en #true #tx_settings #map #crc_en #false #parity_en #true #last_byte_valid_bits_num #7 #timeout_us #70 (in µs) #data #bytes(1) #0x26
Response: OK.	Hex: 60450271FFA26130006131424403 Decoded: 60 44 0271 FF A26130006131424403	Hex: A26130006131424403 Decoded: A2 6130 00 6131 42 4403
	#ACK, token len = 0 #2.04 CHANGED #MSG ID #End of options marker #Data	#map #reason #SUCCESS #card response #bstr #card response data

rfidTransparentSessionSetField

Turns on/off the RF field in a transparent session.

Command format

URI	/card/ts
TYPE	CON
REQUEST CODE	POST
args = { "0" : action, "1" : field_command }	Command arguments. For the set field command argument, "1" should contain data of type field_command.
Command arguments. action = (SET_FIELD : 0x03)	Transparent session set field action.
field_command = bool	RF field state request (true for field on, false for field off).

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.04 CHANGED
RESPONSE CODE POSSIBLE ERRORS	4.00 BAD REQUEST Incorrect param has been provided or CBOR structure is corrupted.
	5.03 SERVICE UNAVAILABLE Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS Previous command processing has not finished yet.
data = { "0" : uint ; reason }	The reason provides more detailed information on the communication status with the card. Possible values are: reason = { (SUCCESS : 0x00) / (INVALID_STATE : 0x01) / (INVALID_PARAM : 0x02) / (BUFFER_OVERFLOW : 0x03) / (INTEGRITY : 0x04) / (TIMEOUT : 0x05) / (NOT_IMPLEMENTED : 0x06) / (UNDEFINED : 0x07) }

Examples

Example description	Full CoAP command	Arg / Res data
Command: Request to set field for TS.	Hex: 40020271B463617264027473 FFA26130036131F5 Decoded:	Hex: A26130036131F5 Decoded: A2 #map 61 30 #action

Example description	Full CoAP command		Arg / Res data	
	40	#CON, token len = 0	03	#transparent_session_set_field
			61 31	#action
	02	#POST	F5	#true - RF field on
	0271	#MSG ID		
	B463617264	#Option URI = card		
	027473	#Option URI = ts		
	FF	#End of options marker		
	A26130036131F5	#Args		
Response: OK.	Hex:		A1613000	
	60440271FFA1613000		Decoded:	
	Decoded:		A1	#map
	60	#ACK, token len = 0	61 30	#ARG1: reason
	44	#CHANGED	00	#SUCCESS
	0271	#MSG ID		
	FF	#End of options marker		
	A1613000	#Data		

Config commands

Configuration commands allow you to:

- Write a complete reader configuration using a manifest file.
- Manipulate single config items specified by OIDs.

configGetParam

Gets the value of the selected parameter. Parameter selection is by parameter OID.
Special OIDs (1.1.9.0.*) cannot be read using this command.

Command format

URI	/cfg/param
TYPE	CON
REQUEST CODE	GET
args = { "0" : oid_sel }	Command arguments.
oid_sel = [6*6 uint]	Object identifier for configuration item.

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.05 CONTENT
RESPONSE CODE POSSIBLE ERRORS	4.00 BAD REQUEST Incorrect param has been provided or CBOR structure is corrupted.
	5.03 SERVICE UNAVAILABLE Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS Previous command processing has not finished yet.
data = { "0" : value }	Response data.
value = bstr	Parameter value.

Examples

Example description	Full CoAP command	Arg / Res data
Command: Get the value of parameter [OID] 1.1.1.1.1.1	Hex: 4001C43BB363666705706172616DFF A16130860101010101 Decoded: 40 01 C43B B3636667 05706172616D FF A1613086010101 010101 #CON, token len = 0 #GET #MSG ID #Option URI = cfg #Option URI = param #End of options marker #Args	Hex: A1613086010101010101 Decoded: A1 61 30 86 010101010101 #map #ARG1: OID sel #bstr #OID
Response: OK.	Hex: 6045C43BFFA16130420102 Decoded: 60 #ACK, token len = 0	Hex: A16130420102 Decoded: A1 #map

Example description	Full CoAP command		Arg / Res data	
Response: Wrong param value.	45	#CONTENT	61 30	#ARG1: Param value
	C43B	#MSG ID	42	#bstr
	FF	#End of options marker	0102	#dummy response value
	A16130420102	#Data	NONE	
	Hex: 6080C43B			
	Decoded:			
	60	#ACK, token len = 0		
	80	#BAD REQUEST		
	C43B	#MSG ID		

configSetParam

Sets a single configuration option by OID parameter value (see "Configuration structures" on page 247). Changed configuration options are applied after a device reset. The OID is a CBOR-encoded six-element uint array.

- LONG INTEGER type is encoded on 8 bytes as a signed integer with two's complement encoding, little-endian.
- INTEGER type is encoded on 4 bytes as a signed integer with two's complement encoding, little-endian.
- ENUM type is encoded the same way as the INTEGER type.
- BOOL type uses one (unsigned) byte. Any non-zero value is treated as a TRUE/ON/ENABLED, and 0 is treated as FALSE/OFF/DISABLED.
- BYTES type is stored "as is" without decoding/encoding. This type can be used for storing a data structure, so a detailed encoding is described in the appropriate row in the configuration item OIDs table.

Command format

URI	/cfg/param
TYPE	CON
REQUEST CODE	POST
args = { "0" : oid_sel, "1" : val }	Command arguments.
oid_sel = [6*6 uint]	Object identifier for configuration item.
val = bstr	Parameter value.

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.04 CHANGED
RESPONSE CODE POSSIBLE ERRORS	4.00 BAD REQUEST Incorrect param has been provided or CBOR structure is corrupted.
	5.03 SERVICE UNAVAILABLE Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS Previous command processing has not finished yet.
data = none	No response data is returned.

Examples

Example description	Full CoAP command	Arg / Res data
Command: Set parameter 1.1.1.1.1 to value 0x0102.	Hex: 400295B6B363666705706172616D FFA26130860101010101016131420102 Decoded: 40 02 95B6 B3636667 05706172616D	Hex: A26130860101010101016131420102 Decoded: A2 61 30 86 01 01 01 01 01 01 61 31 42
	#CON, token len = 0	#map
	#POST	#ARG1: OID sel
	#MSG ID	#array
	#Option URI = cfg	#OID
	#Option URI = param	#ARG2: param value
		#bstr

Example description	Full CoAP command		Arg / Res data
	FF	#End of options marker	0102 #param value - dummy # not proper value
	A26130860101010101016131420102	#Args	
Response: OK.	Hex: 604495B6 Decoded: 60 44 95B6	 #ACK, token len = 0 #CHANGED #MSG ID	NONE
Response: Wrong param value.	Hex: 608095B6 Decoded: 60 80 95B6	 #ACK, token len = 0 #BAD REQUEST #MSG ID	NONE

configGetDataModelVersion

Gets the supported Config Data Model version from the reader.

Command format

URI	/cfg/model
TYPE	CON
REQUEST CODE	GET
ARG data = none	No arguments are returned.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.05 CONTENT	
RESPONSE CODE POSSIBLE ERRORS	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
data= { "0" : config_data_model }	Response data.	
config_data_model = [major : uint8, minor : uint8]	Config Data Model version definition.	

Examples

Example description	Full CoAP command		Arg / Res data	
Command: Get Config Data Model.	Hex:		NONE	
	40015FB2B3636667056D6F64656C			
	Decoded:			
	40	#CON, token len = 0		
	01	#GET		
	5FB2	#MSG ID		
Response: Supported Config Data Model == 1.0.	B3636667	#Option URI = cfg	Hex: A16130820100 Decoded: A1 61 30 82 01 00	
	056D6F6465	#Option URI = model		
	Hex:			
	60455FB2FFA16130820100			
	Decoded:			
	60	#ACK, token len = 0		
	45	#CONTENT		#map
	5FB2	#MSG ID		#ARG1: config_data_model
	FF	#End of options marker		#array
	A16130820100	#Data		#MAJOR = 1
				#MINOR = 0

configGetCounter

Gets the manifest counter which has been used to update the device configuration to the current state.

Command format

URI	/cfg/counter
TYPE	CON
REQUEST CODE	GET
args = none	No arguments are required.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.05 CONTENT	
RESPONSE CODE	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
POSSIBLE ERRORS	4.29 TOO MANY REQUESTS	
	Previous command processing has not finished yet.	
data= { "0" : counter_val }	Response data.	
counter_val = uint .size 8	Current counter stored in the device. For C, it is the equivalent of the uin64_t type.	

Examples

Example description	Full CoAP command	Arg / Res data
Command: Get the counter value stored in the device.	Hex: 4001D1EEB363666707636F756E746572 Decoded: 40 #CON, token len = 0 01 #GET D1EE #MSG ID B3636667 #Option URI = cfg 07636F756E7465 #Option URI = counter	NONE
Response: Counter value = 0x100.	Hex: 6045D1EEFFA16130190100 Decoded: 60 #ACK, token len = 0 45 #CONTENT D1EE #MSG ID FF #End of options marker A16130190100 #Data	Hex: A16130190100 Decoded: A1 #map 61 30 #ARG1: counter_val 19 0100 #Counter

configGetUUID

Gets the UUID of the manifest which was used to update the configuration of the device to its current state.

Command format

URI	/cfg/uuid
TYPE	CON
REQUEST CODE	GET
args = none	No arguments are required.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.05 CONTENT	
RESPONSE CODE POSSIBLE ERRORS	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
data= { "0" : uuid_val }	Response data.	
uuid_val = bstr	UUID of the manifest which was used to update the configuration of the device to its current state.	

Examples

Example description	Full CoAP command		Arg / Res data			
Command: Get the UUID of the manifest which was used to update the configuration of the device to its current state.	Hex:	4001d1eeb36366670475756964	NONE			
	Decoded:					
	40				#CON, token len = 0	
	01				#GET	
	D1EE				#MSG ID	
	B3636667				#Option URI = cfg	
	07636F756E7465	#Option URI = uuid				
Response: Dummy UUID.	Hex:	6045ED77FFA1613049001122334455667788	Hex:	A1613049001122334455667788		
	Decoded:		Decoded:			
	60		#ACK, token len = 0		A1	#map
	45		#CONTENT		61 30	#ARG1:
	ED77		#MSG ID			uuid_val
	FF		#End of options marker		49	#bstr
	A1613049001122334455667788		#Data		001122334455667788	#dummy uuid

configGetName

Gets the configuration name (description) value.

Command format

URI	/cfg/name
TYPE	CON
REQUEST CODE	GET
args = none	No arguments are required.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.05 CONTENT	
RESPONSE CODE POSSIBLE ERRORS	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.00 BAD REQUEST	Incorrect CBOR structure.
	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
data= { "0" : name }	Response data.	
name = bstr	UTF8 configuration name.	

Examples

Example description	Full CoAP command		Arg / Res data	
Command: Get config name.	Hex:	40017572B3636667046E616D65	NONE	
	Decoded:			
	40			
	01			
	7572			
	B3636667			
Response: Counter value == "abc".	046E616D65	#CON, token len = 0 #GET #MSG ID #Option URI = cfg #Option URI = name	Hex: A1613043616263 Decoded: A1 61 30 43 616263 #map #ARG1: name #bstr #"abc"	
	Hex:			
	60457572FFA1613043616263			
	Decoded:			
	60			
	45			
	7572	#ACK, token len = 0 #CONTENT #MSG ID #End of options marker #Data		
	FF			
	A1613043616263			

configSetName

Sets the configuration name (description) of a configuration set. The OID parameter is a UTF8 encoded byte string containing the new name for the configuration set.

Command format

URI	/cfg/name
TYPE	CON
REQUEST CODE	POST
args = { "0" : name }	Command arguments.
args = bstr	UTF8 configuration name.

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.05 CHANGED
RESPONSE CODE POSSIBLE ERRORS	5.03 SERVICE UNAVAILABLE Device is busy and cannot process the client's request.
	4.00 BAD REQUEST Incorrect CBOR structure.
	4.29 TOO MANY REQUESTS Previous command processing has not finished yet.
data= none	No response data is returned.

Examples

Example description	Full CoAP command	Arg / Res data
Command: Set name to "abc".	Hex: 40027572B3636667046E616D65 FFA1613043616263 Decoded: 40 #CON, token len = 0 02 #POST 7572 #MSG ID B3636667 #Option URI = cfg 046E616D65 #Option URI = name FF # End of options marker A1613043616263 #Args	Hex: A1613043616263 Decoded: A1 #map 61 30 #ARG1: name sel 43 #bstr 616263 #"abc"
Response: OK.	Hex: 60447572 Decoded: 60 #ACK, token len = 0 44 #CHANGED 7572 #MSG ID	NONE

configReset

Resets the reader configuration to default.

For current FW release, configuration name set using configSetName command is left unchanged after this command execution.

Command format

URI	/cfg/name
TYPE	CON
REQUEST CODE	POST
args = { "0" : reset_target }	Command arguments.
reset_target = &reset_target_enum = { reset_target_enum = (CONFIGURATION_TO_DEFAULT: 0x00)	Selection of the component to be reset to defaults.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.04 CHANGED	
RESPONSE CODE POSSIBLE ERRORS	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.00 BAD REQUEST	Incorrect CBOR structure.
	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
	5.00 INTERNAL SERVER ERROR	Clearing one or more configuration items data failed. Operation shall be retried.
data= none	No response data is returned.	

Examples

Example description	Full CoAP command	Arg / Res data
Command: Reset configuration to default.	Hex: 4202EEF3A044B3636667057265736574FFFA 1613000 Decoded: 42 #CON, token len = 2 02 #POST EEF3 #MSG ID A044 #Option URI = card B3636667057265736574 #Option URI = /cfg/reset FF #End of options marker A1613000 #Args	Hex: A1613000 Decoded: A1 #map 61 30 #ARG: reset_target 02 #unassigned (0) CONFIGURATION_TO_DEFAULT
Response: OK.	Hex: 6244EEF3A044 Decoded: 62 #ACK, token len = 2	NONE

Example description	Full CoAP command	Arg / Res data
	44 #CHANGED EEF3 #MSG ID A044 #TOKEN	

File update commands

File update commands provide upgrade features for all reader components (core and secure element) and configuration.

fileStore

Initializes firmware/config file storage and then sends chunks. Depending on the action type, different operations will be executed.

Command format

URI	/file
TYPE	CON
REQUEST CODE	POST
args = { "0" : action, ? "1" : file_chunk }	Command arguments. For the CHUNK action, argument "1" should be a chunk. For the INIT and PROCESS actions, argument "1" should not be present.
action = ((INIT : 0x00) / (CHUNK : 0x01) / (PROCESS : 0x02))	File transfer specific actions.
file_chunk = bstr	Next chunk of the file.

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.04 CHANGED
RESPONSE CODE POSSIBLE ERRORS	4.12 PRECONDITION FAILED
	Incorrect process flow has been used: CHUNK has been sent before INIT has been run, or PROCESS has been sent before all CHUNKs have been uploaded and verified.
	5.00 INTERNAL SERVER ERROR
	File upload and processing error. Check status using "fileGetUpdateStatus " on page 200.
	4.00 BAD REQUEST
	Incorrect param has been provided or CBOR structure is corrupted.
	5.03 SERVICE UNAVAILABLE
	Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS
	Previous command processing has not finished yet OR file chunk is being stored/processed. Command needs to be retried.
data = none	No response data is returned.

Examples

Example description	Full CoAP command	Arg / Res data
Command: Initialize file transfer.	Hex: 4002E381B466696C65FFA1613000 Decoded: 40 02 E381	Hex: A1613000 Decoded: A1 30 00
	#CON, token len = 0	#map 61
	#POST	#action
	#MSG ID	#Init

Example description	Full CoAP command		Arg / Res data	
	B466696C65	#Option URI = file		
	FF	#End of options marker		
	A1613000	#Args		
Command: Send next chunk.	Hex:		Hex:	
	4002E381B466696C65FF		A261300161314400112233	
	A261300161314400112233		Decoded:	
	Decoded:		A1	#map
	40	#CON, token len = 0	61 30	#action
	02	#POST	01	#Chunk
	E381	#MSG ID	6131	#chunk
	B466696C65	#Option URI = file	44	#bstr
	FF	#End of options marker	00112233	#file_chunk
	A26130006131	#Args		
Command: Finalize file transfer and request device to process it.	Hex:		Hex:	
	4002E381B466696C65FFA1613002		A1613002	
	Decoded:		Decoded:	
	40	#CON, token len = 0	A1	#map
	02	#POST	61 30	#action
	E381	#MSG ID	02	#process
	B466696C65	#Option URI = file		
	FF	#End of options marker		
Response: OK.	A1613002	#Args	NONE	
	Hex:			
	6044E381			
	Decoded:			
	60	#ACK, token len = 0		
	44	#CHANGED		
	E381	#MSG ID		

fileGetLength

Gets the length of the currently stored file (total length of all chunks stored so far).

Command format

URI	/file
TYPE	CON
REQUEST CODE	GET
args = none	No arguments are required.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.05 CONTENT	
RESPONSE CODE	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
POSSIBLE ERRORS	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
data = { "0" : file_size }	Response data.	
file_size = unsigned	File length	

Examples

Example description	Full CoAP command	Arg / Res data
Command: Get the current size of the file	Hex: 400181FEB466696C65 Decoded: 40 #CON, token len = 0 01 #GET 3909 #MSG ID B466696C65 #Option URI = file	None
Response: File len = 0x100	Hex: 6045B561FFA16130190100 Decoded: 60 #ACK, token len = 0 45 #CONTENT 3909 #MSG ID FF #End of options marker A16130190100 #Data	Hex: A16130190100 Decoded: A1 #map 61 30 #ARG1: file size 19 0100 #File size

fileClear

Clears all file data sent so far.

Command format

URI	/file
TYPE	CON
REQUEST CODE	DELETE
args = none	No arguments are returned.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.02 DELETED	
RESPONSE CODE POSSIBLE ERRORS	4.12 PRECONDITION FAILED	No file chunk has been uploaded yet.
	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
data = none	No response data is returned.	

Examples

Example description	Full CoAP command	Arg / Res data
Command: Delete the file data.	Hex: 40042437B466696C65	NONE
	Decoded: 40	
	04	
	2437	
	B466696C65	
Response: OK	#CON, token len = 0	NONE
	#DELETE	
	#MSG ID	
	#Option URI = file	
	Hex: 60422437	
	Decoded: 60	
	42	
	2437	
	#ACK, token len = 0	
	#DELETED	
	#MSG ID	

fileGetUpdateStatus

Gets the status of the update for firmware or configuration. The command can be used to verify file processing and to get information about any issue that occurred. The command also returns the status of the previously executed update if no update is ongoing. A file_id is returned if:

- A firmware update is ongoing, to report which file is currently being processed.
- A firmware update has finished with errors. The file_id describes the file which caused the problem.

For configuration, there is always one file.

Command format

URI	/file/status
TYPE	CON
REQUEST CODE	GET
ARG data = none	No arguments are returned.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.05 CONTENT	
RESPONSE CODE POSSIBLE ERRORS	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
data= { "0" : update_status, "1" : mft_uuid, "2" : mft_counter, ? "3" : file_id }	Response data.	
update_status = &update_status_enum update_status_enum = (ERA_FILE_STATUS_IN_PROGRESS : 0, ERA_FILE_STATUS_FILE_VALIDATION : 1, ERA_FILE_STATUS_FILE_VALID : 2, ERA_FILE_STATUS_OK : 3, ERA_FILE_STATUS_BACKUP_ERROR : 4, ERA_FILE_STATUS_PREPARE_ERROR : 5, ERA_FILE_STATUS_UPDATE_ERROR : 6, ERA_FILE_STATUS_VERIFY_ERROR : 7, ERA_FILE_STATUS_COMMIT_ERROR : 8, ERA_FILE_STATUS_GENERIC_ERROR : 9, ERA_FILE_STATUS_CONFIGURATION_UPDATE_ERRO R : 10)	Status of the FW update.	
mft_uuid = bstr	16 bytes of the manifest UUID used for last firmware update if the reader is in the idle state, or the currently uploaded manifest otherwise. If no update has been performed yet, it will contain 16 zeros.	
counter_val = uint .size 8	Counter of the UUID used for the update.	

	In no update has been performed yet, the value will be zero.
file_id = 1...4	ID of the file which is now being processed or caused an error during previous firmware update.

Examples

Example description	Full CoAP command	Arg / Res data
Command: Get the firmware update status	Hex: 4001E950B466696C6506737461747573 Decoded: 40 #CON, token len = 0 01 #GET E950 #MSG ID B466696C65 #Option URI = file 06737461747573 #Option URI = status	NONE
Response: Previous firmware update has finished with an error. There was an update issue during processing of file == 2.	Hex: 6045E950FFA4613006613150001122334455 66778899AABBCCDDEEFF6132190100613302 Decoded: 60 #ACK, token len = 0 45 #CONTENT E950 #MSG ID FF #End of options marker A461300661315000 #Data 1122334455667788 99AABBCCDDEEFF613 2190100613302	Hex: A461300661315000112233445566 778899AABBCCDDEEFF6132190100 613302 Decoded: A4 #map 61 30 #ARG1: update_status 06 #ERA_FILE_STATUS_VERIFY_ERROR 61 31 #ARG2: mft_uuid 50 #bstr 0011223344556677 #mft_uuid 8899AABBCCDDEEFF 61 32 #ARG3: mft_counter 19 0100 #Counter = 0x100 61 33 #ARG4: file_id 02 #file_id = 0x02

Firmware commands

Firmware commands allow you to use the External Reader API to inspect the firmware.

fwGetVersion

Gets the version of particular components of the reader. It is possible to define which component version will be returned.

Command format

URI	/fw/version
TYPE	CON
REQUEST CODE	GET
args = { "0" : fw_sel }	Command arguments.
fw_sel= &fw_sel_enum fw_sel_enum= (ERA_GEN_FW_CORE : 0x00, ERA_GEN_FW_BLE : 0x01, ERA_GEN_FW_SAM : 0x02, ERA_GEN_FW_ALL : 0x03)	Selection of the component version to be returned.

Response format

RESPONSE TYPE	ACK
RESPONSE CODE OK	2.05 CONTENT
RESPONSE CODE POSSIBLE ERRORS	5.03 SERVICE UNAVAILABLE 4.29 TOO MANY REQUESTS
	Device is busy and cannot process the client's request. Previous command processing has not finished yet OR the Secure Element is busy with another operation requested by the client.
data= { ? "0" : core_fw_ver, ? "1" : ble_fw_ver, ? "2" : sam_fw_ver }	Response data.
core_fw_ver =[major : uint8, minor : uint8, patch : uint16, short_sha : uint32]	Core firmware version definition.
sam_fw_ver = [major : uint8, minor : uint8]	SAM firmware version definition.

Examples

Example description	Full CoAP command	Arg / Res data
Command:	Hex:	Hex:

Example description	Full CoAP command	Arg / Res data
Get core firmware version.	4001E50DB266770776657273696F6E	A1613000
	FFA1613000	Decoded:
	Decoded:	A1
	40	#CON, token len = 0
	01	#GET
	E50D	#MSG ID
	B26677	#Option URI = fw
	0776657273696F6E	#Option URI = version
Response: OK.	FF	#End of options marker
	A1613000	#Args
	Hex:	Hex:
	6045E50DFFA161308401188019	A16130840118801901001AAABBCCDD
	01001AAABBCCDD	Decoded:
	Decoded:	A1
	60	#ACK, token len = 0
	45	#CONTENT
	E50D	#MSG ID
	FF	#End of options marker
	A161308401188019	#Data
	01001AAABBCCDD	
		1A AABBCCDD
		#SHA = 0xAABBCCDD
		#PATCH = 256
		#MINOR = 128
		#MAJOR = 1
		#array
		#ARG1: core_fw_ver
		#map

fwGetCounter

Gets the manifest counter which has been used to update the device to the current firmware version.

Command format

URI	/fw/counter
TYPE	CON
REQUEST CODE	GET
args = none	No arguments are required.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.05 CONTENT	
RESPONSE CODE POSSIBLE ERRORS	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
	4.00 BAD REQUEST	Incorrect param has been provided or CBOR structure is corrupted.
data= { "0" : counter_val }	Response data.	
counter_val = uint .size 8	Current counter stored in the device. For C, it is the equivalent of the uin64_t type.	

Examples

Example description	Full CoAP command	Arg / Res data		
Command: Get the counter value stored in the device.	Hex:	NONE		
	4001D1EEB2667707636F756E746572			
	Decoded:			
	40			#CON, token len = 0
	01			#GET
	D1EE			#MSG ID
Response: Counter value = 0x100.	B26677	Hex: A16130190100 Decoded: A1		
	07636F756E7465			#Option URI = fw
				#Option URI = counter
	6045D1EEFFA16130190100	61 30		
	Decoded:			
	60			#ACK, token len = 0
	45			#CONTENT
	D1EE			#MSG ID
	FF			#End of options marker
	A16130190100	19 0100		

fwGetUUID

Gets the manifest UUID of the manifest which was used to update the firmware of the device to its current state.

Command format

URI	/fw/uuid
TYPE	CON
REQUEST CODE	GET
args = none	No arguments are required.

Response format

RESPONSE TYPE	ACK	
RESPONSE CODE OK	2.05 CONTENT	
RESPONSE CODE	5.03 SERVICE UNAVAILABLE	Device is busy and cannot process the client's request.
POSSIBLE ERRORS	4.29 TOO MANY REQUESTS	Previous command processing has not finished yet.
data= { "0" : uuid_val }	Response data.	
uuid_val = bstr	UUID of the manifest which was used to update the configuration of the device to its current state.	

Examples

Example description	Full CoAP command		Arg / Res data	
Command: Get the UUID of the manifest which was used to update the firmware of the device to its current state.	Hex:		NONE	
	4001D1EEB266770475756964			
	Decoded:			
	40	#CON, token len = 0		
	01	#GET		
	D1EE	#MSG ID		
Response: Dummy UUID.	B26677	#Option URI = fw	Hex: A1613049001122334455667788 Decoded: A2 61 30 49 001122334455667788	
	07636F756E7465	#Option URI = uuid		
	Hex:			
	6045ED77FFA1613049001122334455667788			
	Decoded:			
	60	#ACK, token len = 0		
	45	#CONTENT	#map	
	ED77	#MSG ID	#ARG1: uuid_val	
	FF	#End of options marker	#bstr	
	A1613049001122334455667788	#Data	#dummy uuid	

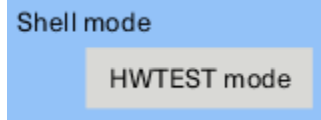
Shell mode

Introduction

Shell mode allows you to communicate with the reader using human readable commands. Shell mode is available on the LPUART interface.

Shell mode provides two sets of commands: general commands and hwtest commands. You can switch between shell modes using "External Reader API commands" on page 104. Hwtest commands are available only from hwtest shell mode.

General commands are available from both shell modes, but there is one exception: the `get_version` command (general commands) is not available from hwtest mode.



Shell mode activation

According to the transition diagram, SHELL MODE activation is possible only from the EXTERNAL API reader mode. To perform a switch to SHELL MODE, send a "genChangeState " on page 107 command with the correct parameter.

Shell interface

SHELL MODE is available on the LPUART interface with the following parameters:

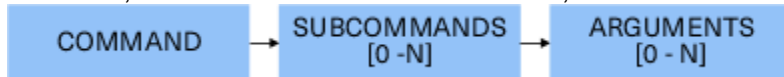
- Baud rate: 115200 (only)
- Parity: None
- Stop bits: 1

SHELL MODE provides a Unix-like shell interface.

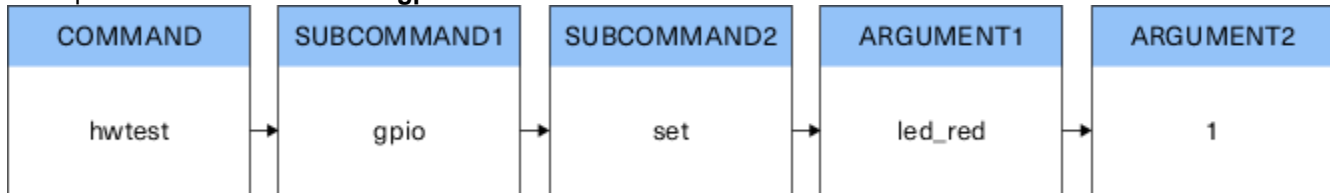
Shell commands

Introduction

This section describes all the available shell commands. The general shell frame construction is composed of the command, a variable number of subcommands, and a variable number of arguments.



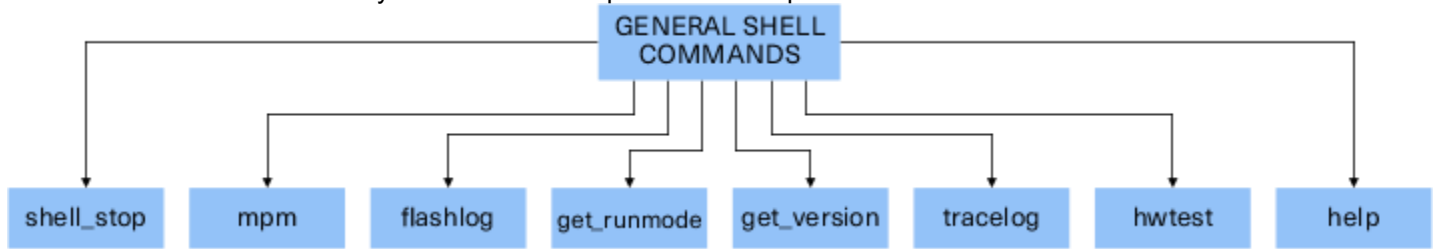
Example shell command **hwtest gpio set led red1**:



The general response is a human readable message, for example: led_blue, set to value: 1. If the command or arguments are incorrect, a "command not found" response will be printed, or a potential hint.

General shell commands

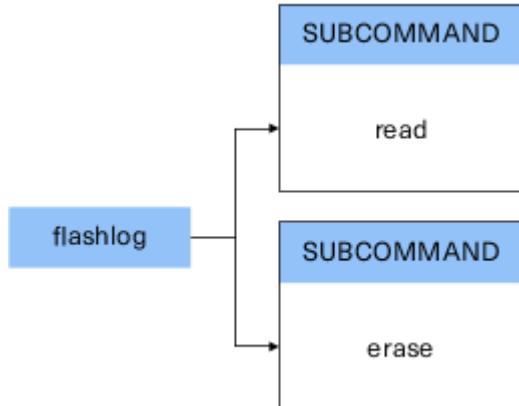
General shell commands allow you to check some parts of reader parameters.



General shell commands description and usage

Flashlog commands

These commands allow you to read warnings and error logs raised by external flash API.



Get runmode command

Provides information about the current reader run mode.

`get_runmode`

Get version command

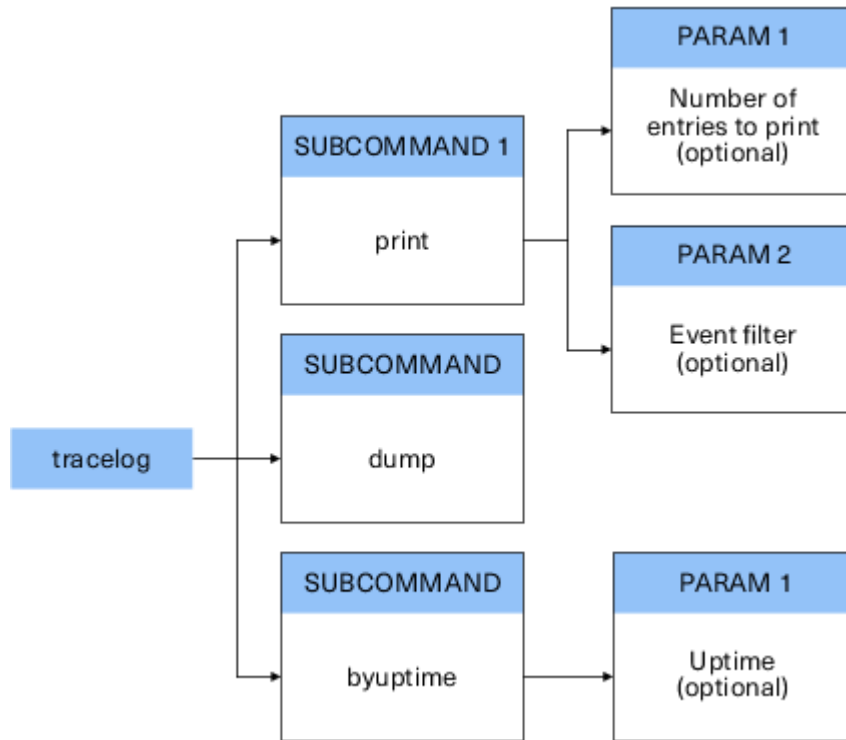
This command allows you to get information about the current versions of reader FW components.

- Product serial number
- SE serial number
- Core version
- SE version
- Revision
- Product info (reader type, color)
- Manufacturing date

`get_version`

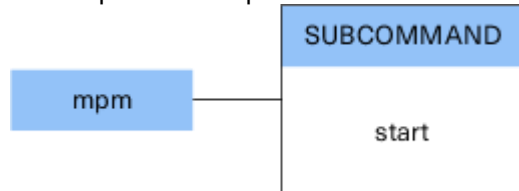
Trace log commands

The trace log module is part of the reader core firmware. The module stores the reader core's crucial logs and assertions in flash memory. You can read the stored data using a shell command.



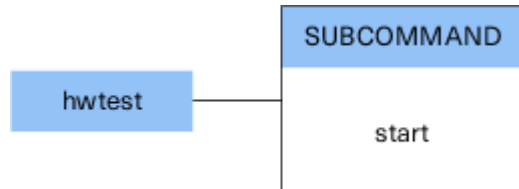
help command

The help command prints information about available shell commands.



hwtest command

The hwtest command starts hwtest mode from general shell mode.



Stop command

The stop command resets the reader and starts it in default configured mode.

stop

General shell commands description

Flashlog commands

Command description	Reads stored external flash logs
Command frame	flashlog read
Potential responses	Human readable table with stored external flash logs
Example usage	flashlog read
Example response	TIME FILE ID FILE LINE EVENT ID INFO(HEX) INFO(ASCII) 1024005859 8 375 48 303B4653207573616765203E205B255D3935 0;FS usage >[%]95
Command description	Clears external flash logs buffer
Command frame	flashlog erase
Potential responses	No response
Example usage	flashlog erase

Get runmode command

Command description	Returns current reader core run mode
Command frame	get_runmode
Potential responses	Human readable information about current run mode casted to the decimal form. Decimal numbers assigned to modes: MSM_MODE_APPLICATION = 0 MSM_MODE_UPDATE = 1 MSM_MODE_FSRECOVERY = 2 MSM_MODE_MPM_SPI = 3 MSM_MODE_MPM_UART = 4 MSM_MODE_BLE_TEST = 5 MSM_MODE_FACTORY = 6 MSM_MODE_HW_TEST = 7 MSM_MODE_SHELL = 8 MSM_MODE_UNKNOWN = 9
Example usage	get_runmode
Example response	Current run mode is: 6

Get version command

Command description	Prints information about current versions of FW components
Command frame	get_version
Potential responses	Human readable information
Example usage	get_version
Example usage	Product serial number: 01-ZG0K4O0006WO99920200616CM-01 SAM serial number: 2A8A204BDE0AC23701490300 Core version: 0.5.0 SE version: 7.55 Ble stack version: 1.17.0 Ble FUS version: 1.2.0 revision: A product info: reader type: B color: R date: 11 - 6 - 2023

Tracelog commands

print

Command description	Prints stored tracelog entries
Command frame	tracelog print PARAM1(optional) PARAM2(optional)
PARAM1	Entries num
PARAM2	Event filter
	Number of events to print. Allowed data 1-30 or 'all'. Filter for events to print. Possible values: TRACELOG_EVENT_ID_MAIN_START TRACELOG_EVENT_FS_FAILURE TRACELOG_EVENT_WDG_TRIGGERED TRACELOG_EVENT_APP_SETTINGS_FAILURE TRACELOG_EVENT_MSM_RUN_REQ_FAILURE TRACELOG_EVENT_U_CONTROLLER_TOO_MANY_RESETS TRACELOG_EVENT_NODE_FSM_STATES_UNKNOWN EVENT TRACELOG_EVENT_SAM_NODE_CONTROLLER_COMM FAILURE TRACELOG_EVENT_ERROR_MESSAGE TRACELOG_EVENT_WARNING_MESSAGE TRACELOG_EVENT_ASSERTION_OS TRACELOG_EVENT_ASSERTION_APP TRACELOG_EVENT_ASSERTION_ETL TRACELOG_EVENT_ID_TEST TRACELOG_EVENT_ID_TEST2
Potential responses	Human readable table with entries
Example usage	tracelog print all TRACELOG_EVENT_FS_FAILURE
Example response	UPTIME(sec) FILE_ID LINE EVENT [LOG] 11 8 375 48 [0;FS usage >[%]95]
Command description	Prints stored tracelog entries with tags

dump

Command description	Prints stored tracelog entries in uptime order with optional filter
Command frame	tracelog dump
Potential responses	Entries with tags
Example usage	tracelog dump
Example response	<tracelog> <entry>UPTIME(sec);FILE_ID;LINE;EVENT;[LOG]</entry> <entry>11;8;375;48;[0;FS usage >[%]95]</entry> </tracelog>

byuptime

Command frame	tracelog byuptime PARAM(optional)
PARAM	Filter entries printing only entries with uptime (in seconds) below this parameter value
Potential responses	Tracelog entries in human readable table
Example usage	tracelog byuptime 12
Example response	UPTIME(sec) FILE_ID LINE EVENT [LOG] 11 8 375 48 [0;FS usage >[%]95]

mpm command

Command description	Starts MPM mode of reader
Command frame	mpm start
Potential responses	No response
Example usage	mpm start

help command

Command description	Prints information about available commands
Command frame	help
Potential responses	flashlog: Flash logger shell commands get_runmode: Get current MSM run mode get_version: Get version of available modules help: Prints the help message hwtest: Test modes mpm :MPM shell commands shell_stop: Stop shell mode. tracelog :Tracelog commands
Example usage	help

hwtest command

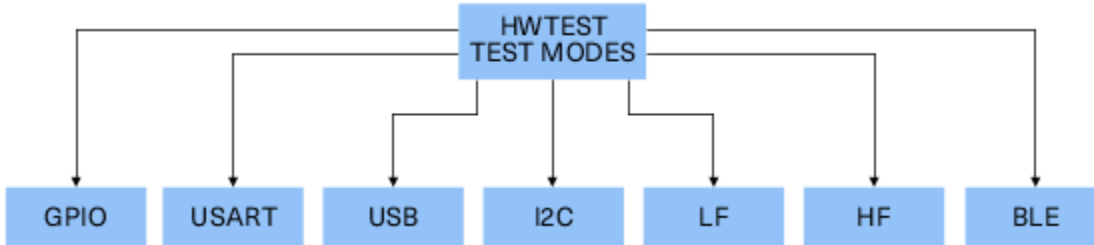
Command description	Starts hwtest mode
Command frame	hwtest start
Potential responses	Entering test mode
Example usage	hwtest start

stop command

Command description	Stops shell mode, resets reader. After reset reader starts in default mode.
Command frame	shell_stop
Potential responses	No response
Example usage	shell_stop

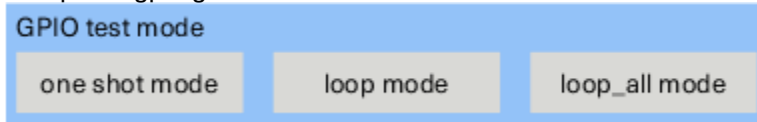
Hardware (HW) test commands

HW test commands are used to test and check reader features. HW test mode can be activated only from SHELL MODE. To switch to this mode, you must send the hwtest start command. Each test mode is designed to work on an initialized reader. If you want to switch between test modes, it is recommended to reset the reader manually. The following diagram shows the available test modes:



GPIO test mode

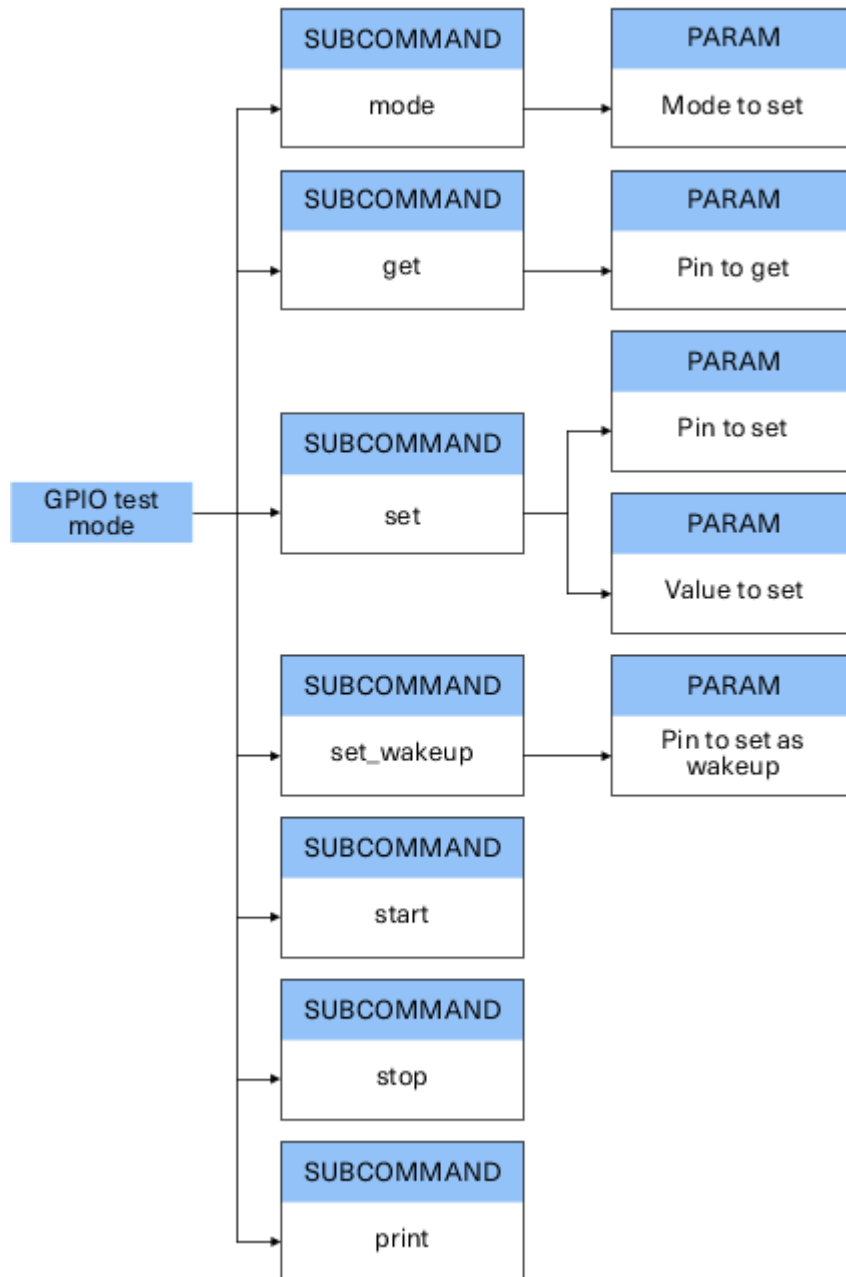
GPIO test mode provides several tests commands to allow you to control I/O pins. Most work as a single operation, except the gpio get command which has several modes:



Modes

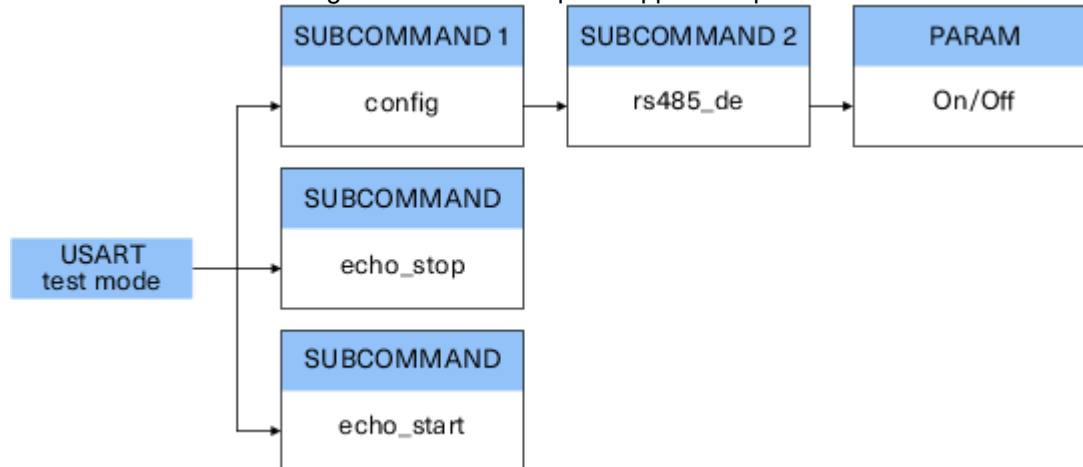
- One shot: The get operation is done once per command.
- Loop: Reads a single pin value every 10 ms. The pin first has to be configured using a get command. Its value is printed on a change event. Usage:
 - Set mode: `hwtest gpio mode loop`.
 - Set pin to read in a loop: `hwtest gpio get pin_label`.
 - Start gpio test mode: `hwtest gpio start`.
 - Selected pin value is printed to the console on every change event.
- loop_all: Reads all input pins in a loop. The value of a pin is printed on a change event. Usage:
 - Set mode: `hwtest gpio mode loop_all`.
 - Start gpio test mode: `hwtest gpio start`.
 - On each change of an input pin value, its name and new value is printed to the console .

Possible operations



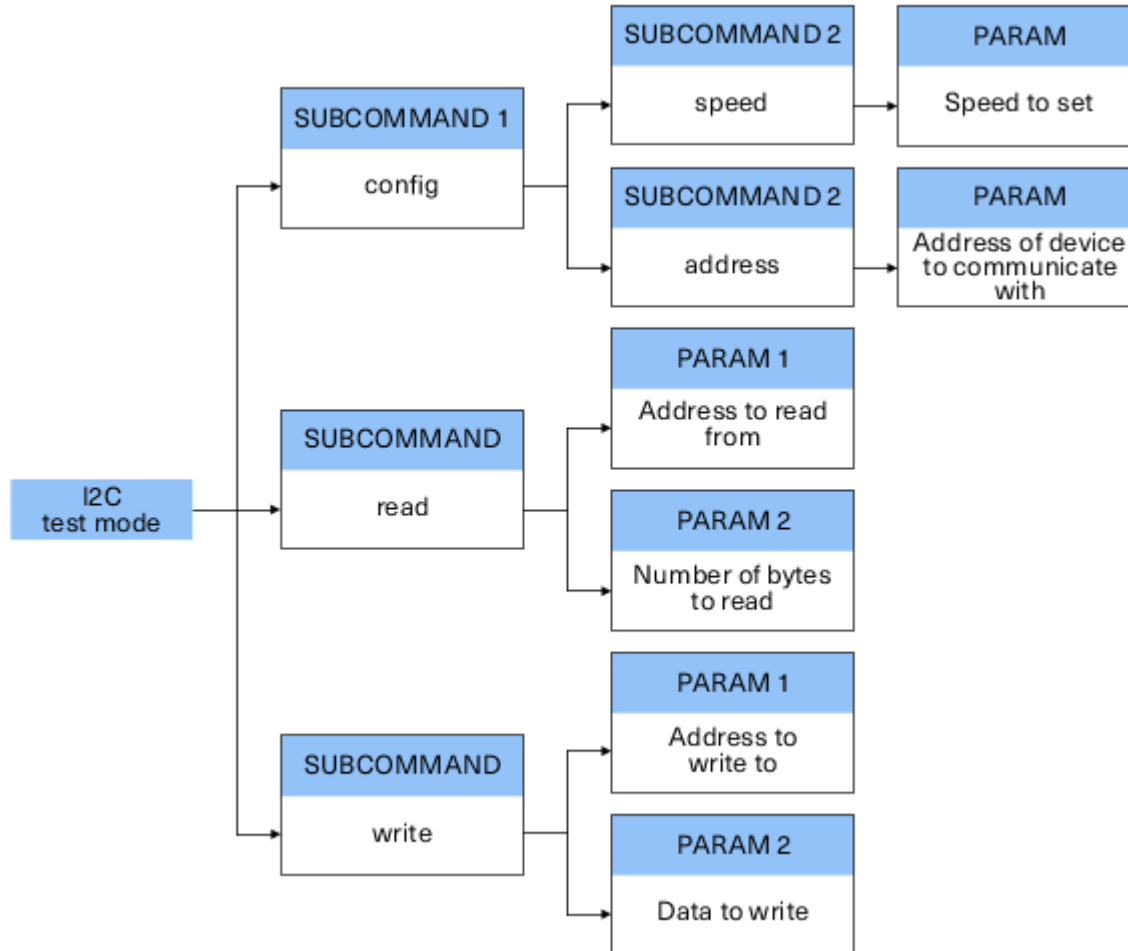
USART test mode

This mode provides a simple way to test USART communication. Using these commands, you can run USART in "echo" mode and can configure the RS485 de pin. Supported operations:



I2C test mode

This mode's commands provide an API to communicate with an external I2C device. Using these commands, you can read or write data using several supported I2C speeds. Supported operations:



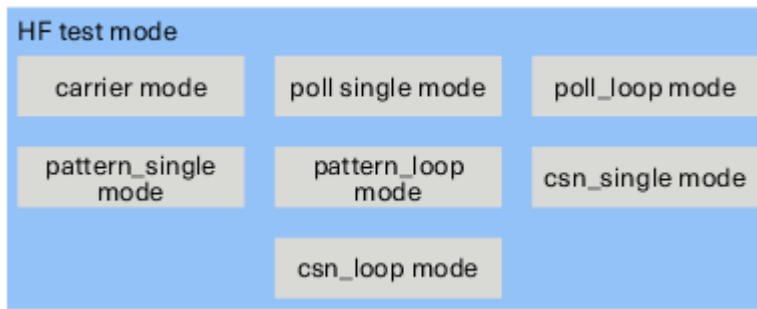
HF test mode

In this mode you can check HF reader operations. You can set the required mode, protocol and data rate (kbit/s):

- Supported protocols:
 - iso14443A, iso14443B
 - iso15693
 - iCLASS
- supported data rates (airspeed):
 - b106
 - b212
 - b424
 - b848

This option applies only to ISO14443 A/B protocols in pattern mode.

HF test mode operating modes:



Mode description

- carrier:** In this mode, the front end generates a constant carrier. Any antenna related settings are taken from the eeprom configuration for ISO14443A @106.
- poll_single:** In this mode, a single command is transmitted to initiate card communication. Depending on the selected protocol this is:
 - WUPA for ISO14443A.
 - WUPB for ISO 14443B.
 - Inventory request for ISO15693.
 - ACTALL for iCLASS.If any response is detected, it will be printed to the terminal (ATQA, ATQB or UID). After execution, this test mode will terminate automatically.
- poll_loop:** Same as poll_single but repeated with the period set by loop_delay (default is 500 ms). To exit this mode, use the "hf_test stop" command.
- pattern_single:** The provided pattern is transmitted using the selected protocol. If any response is detected, it will be printed to the terminal. The transmitted frame will be formatted according to protocol.
- pattern_loop:** Emits the given pattern in a loop until stopped. The loop is repeated with the period set by loop_delay (default is 500 ms).
- csn_single:** Performs CSN reading of the selected credential. In single mode, only one attempt is made. If the read is successful, the CSN is printed to the terminal.
- csn_loop:** Performs CSN reading of the selected credential. In loop mode, reads are performed continuously until stopped. If the read is successful, the CSN is printed to the terminal. The loop is repeated with the period set by loop_delay (default is 500 ms).

Proper usage of test mode requires a specific command sequence:

- Configuration of HF test mode, protocol, bit rate and all required settings (using the config subcommand).
- Start test mode (using the start subcommand).
- If it is a looping command, it should be stopped before other operations (using the stop subcommand).

When the first command for the HF module is executed, "Welcome to HF test mode. RC663 Version:0x1A" is printed to the console.

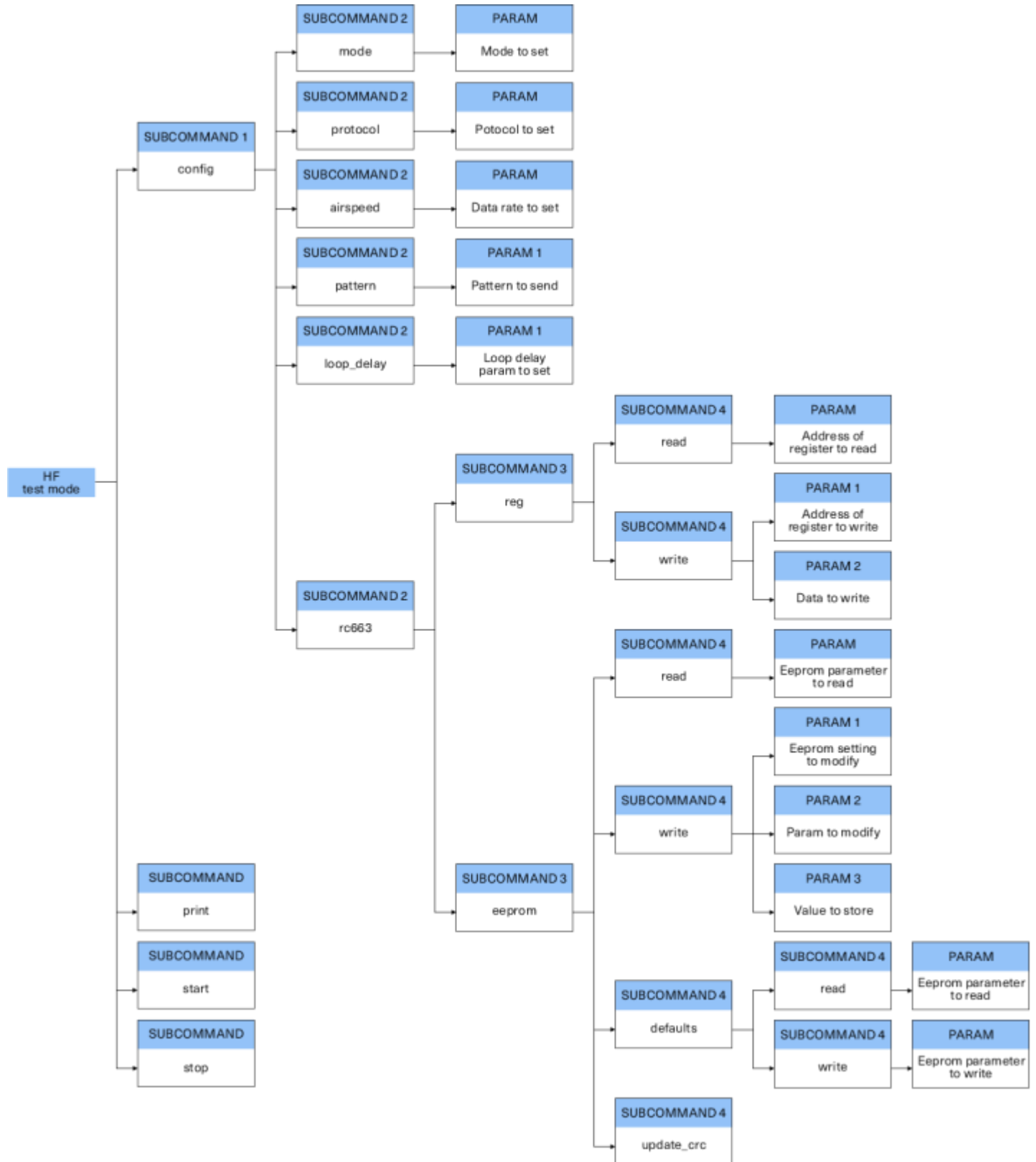
It is also possible to change parameters of the reader's HF chip (currently an RC663). This API provides two suboperations:

- reg: Possibility to set or get the value of a selected register.
- eeprom: Possibility to modify the RC663 eeprom.

There are two sets of RC663 values in the reader. The first copy of RC663 values is stored in the RC663 eeprom. The second copy is stored in the reader core's flash memory. The RC663 operates using the values stored in its eeprom memory. Values stored in the reader core memory are "default" values. Commands used to handle RC663 eeprom values are:

- read/write: Modifies values stored in the RC663 eeprom.
- defaults: Provides the possibility to read all default values from the reader core memory and write them to the RC663 eeprom.
- update_crc: Calculates the crc of the RC663 eeprom and stores it in the RC663. This command should be called every time you change a value in the RC663 eeprom and want to save this value permanently. Otherwise, during reset the RC663 eeprom will be restored to default values.

Supported operations



HW test start and stop commands

In shell mode, you must send an HW test start command to gain access to hw test commands. To stop hw test mode, send a stop command, which resets the reader. The reader restarts in default mode (see "Modes transition" on page 12).

Detailed frames description

GPIO test mode commands

mode

Command frame	hwtest gpio mode PARAM	
Command description	Configures the gpio hwtest to operate in the selected mode.	
PARAM	oneshot	Single read/write operation
	loop	Read operation in loop. Reads only single pin value. Pin first has to be configured using a get command. Pin is read every 10 ms. Value is printed on a change event.
Potential responses	loop_all	Reads all input pins in a loop
	Success	Mode set to mode
	Failure	Human readable information about the reason of failure
Example usage	hwtest gpio mode loop	
Example response	Mode set to LOOP	

get

Command frame	hwtest gpio get PARAM	
Command description	In the oneshot mode, reads the value of the selected pin once and prints it on the shell terminal. In the loop mode, selects this pin for continuous reading and prints its value on a change event.	
PARAM	led_red, led_blue, led_green, tamper, nhold, gpio0, gpio1, npph_rst_op, rs_485_de_usart, rs_485_de_lpuart, wiegand_data_0, wiegand_data_1, dc_dc_enable, digital_enable, rf_enable_dev, tamper_osdp, wakeup, buzzer, sync_mode, hf_pdown, nint_mcu, npph_irq_ip	Input pin
Potential responses	Success	Pin, value: 0/1
	Failure	Human readable information about the reason of failure
Example usage	hwtest gpio get led_green	
Example response	led_green, set to value: 1	

set

Command frame	hwtest gpio set PARAM1 PARAM2	
Command description	Sets the selected output pin to the selected electrical value.	
PARAM1	led_red, led_blue, led_green, gpio0, gpio1, npph_rst_op, rs_485_de_usart, rs_485_de_lpuart, wiegand_data_0, wiegand_data_1, dc_dc_enable, digital_enable, rf_enable_dev, sync_mode, hf_pdown, buzzer	Output pin
PARAM2	0	Electrical 0 value
	1	Electrical 1 value
Potential responses	Success	Pin, set to value: value
	Failure	Human readable information about the reason of failure
Example usage	hwtest gpio set gpio1 1	
Example response	gpio1, set to value: 1	

set_wakeup

Command frame	hwtest gpio set_wakeup PARAM	
Command description	Usable only with ULPM HW tests mode. Sets wakeup pin for ULPM STOP2 mode.	
PARAM	nhold, tamper, tamper_osdp, wakeup	Wakeup pin
Potential responses	Success	Wakeup set to GPIO PIN
	Failure	Human readable information about reason of the failure
Example usage	hwtest gpio set_wakeup nhold	
Example response	wakeup set to GPIOH 0	

start

Command frame	hwtest gpio start	
Command description	If loop or loop_all mode is selected, command starts the loop, otherwise there is no action	
Potential responses	There is no response for this command	
Example usage	hwtest gpio start	

stop

Command frame	hwtest gpio stop
Command description	If loop or loop_all mode is started, command stops the loop, otherwise there is no action
Potential responses	There is no response for this command
Example usage	hwtest gpio stop

print

Command frame	hwtest gpio print
Command description	Prints current configuration of hwtest gpio mode
SUBCOMMAND	print
Potential responses	Mode Started Yes/No
Example usage	hwtest gpio print
Example response	Mode ONE_SHOT Started No

For all GPIO commands, if the selected pin is not available on the hardware, "Pin not found" will be printed.

USART test mode commands

config rs485_de

Command frame	hwtest usart config rs485_de PARAM	
Command description	Enables or disables hardware control of RS485 driver line	
Param	on	New state of rs485_de
	off	
Potential response	There is no response for this command	
Example usage	hwtest usart config rs485_de on	

echo_stop

Command frame	hwtest usart echo_stop
Command description	Stops echo loop for USART interface
Potential response	Human readable response about echo status
Example usage	hwtest usart echo_stop
Example response	Echo ended

echo_start

Command frame	hwtest usart echo_start
Command description	Starts echo loop for USART interface
Potential response	Human readable response about echo status
Example usage	hwtest usart echo_start
Example response	USART echo started.

USB test mode commands

start

Command frame	hwtest usb start
Command description	Test start, triggers enumeration
Potential response	In case of error, human readable information about the error cause
Example usage	hwtest usb start
Example response	If command ok, there is no response

stop

Command frame	hwtest usb stop
Command description	Test stop, tries to stop and detach device, however at the time of writing it, it is not implemented on the ZephyrRTOS/ST side, so the only way to disable USB is to reset the reader. We recommend calling it in all test suites, as it will ultimately function as expected.
Potential response	There is no response for this command
Example usage	hwtest usb start

print

Command frame	hwtest usb print
Command description	Prints USB device options. Could be used only after usb start command.
Potential response	VID: vid PID: pid UIDw2: uidw2 UIDw1: uidw1 UIDwo: uidw0 Device ID: device id
Example usage	hwtest usb print
Example response	VID: 0x076B PID: 0x5500 UIDw2: 0x20383650 UIDw1: 0x564D5002 UIDw0: 0x004B0030 Device ID: USB\VID_076B&PID_5500\564D5002004B0030

I2C test mode commands

config speed

Command frame	hwtest i2c config speed PARAM	
Command description	Configures i2c interface speed	
Param	standard (100 kHz), fast (400 kHz), fast_plus (1 MHz)	Selected speed for i2c interface
Potential responses	Success	No response
	Failure	Human readable information about the reason of the failure
Example usage	hwtest i2c config speed fast	
Example response	There is no response on success	

config address

Command frame	hwtest i2c config address PARAM	
Command description	Configures i2c interface address of device to communicate	
Param	Address of device in hex format	
Potential responses	Success	No response
	Failure	Human readable information about the reason of failure
Example usage	hwtest i2c config address 0	
Example response	There is no response on success	

read

Command frame	hwtest i2c read PARAM1 PARAM2	
Command description	Reads data from device	
PARAM1	Address of register [1 byte size], if address 0 is set, i2c reads without address. Address should be hex string.	
PARAM2	Number of bytes to read [2 bytes size]	
Potential responses	Success	Data read from device
	Failure	Human readable information about the reason of failure
Example usage	hwtest i2c read 00 1	
Example response	00000000: ff .	

write

Command frame	hwtest i2c write PARAM1 PARAM2	
Command description	Writes data to device	
PARAM1	Address of register [1 byte size], if address 0 is set, i2c reads without address. Address should be hex string	
PARAM2	Hex string with data to write [max 64 bytes]	
Potential responses	Success	No message
	Failure	Human readable information about the reason of failure
Example usage	hwtest i2c write 00 F1	
Example response	There is no response on success	

HF test mode commands

config mode

Command frame	hwtest hf config mode PARAM	
Command description	Configures hf test mode	
PARAM	carrier, poll_single, poll_loop, pattern_single, pattern_loop, csn_single, csn_loop	Mode to set
Potential responses	No response	
Example usage	hwtest hf config mode csn_loop	

config protocol

Command frame	hwtest hf config protocol PARAM	
Command description	Configures hf test mode protocol	
PARAM	iso14443A, iso14443B, iso15693, iCLASS	Protocol to set
Potential responses	No response	
Example usage	hwtest hf config protocol iso15693	

config airspeed

Command frame	hwtest hf config airspeed PARAM	
Command description	Configures hf test mode data rate	
PARAM	106, 212, 424, 848	Data rate to set
Potential responses	No response	
Example usage	hwtest hf config airspeed 212	

config pattern

Command frame	hwtest hf config pattern PARAM
Command description	Configures pattern for pattern modes
PARAM	Pattern to send in HEX string format
Potential responses	Human readable message about current pattern
Example usage	hwtest hf config pattern ABCDEF
Example response	Current pattern: ABCDEF

config loop_delay

Command frame	hwtest hf config loop_delay PARAM1
Command description	Configures the delay for loop modes
PARAM	Loop delay value (default: 500)
Potential responses	Human readable response about new value for loop delay
Example usage	hwtest hf config loop_delay 200
Example response	Set loop delay to: 200

config rc663 reg read

Command frame	hwtest hf config rc663 reg read PARAM
Command description	Reads requested param from register
PARAM	Address of requested register in decimal or hex (with 0x) format
Potential responses	Register content or human readable information about error
Example usage	hwtest hf config rc663 reg read 10
Example response	0000000A: 00 .

config rc663 reg write

Command frame	hwtest hf config rc663 reg write PARAM1 PARAM2
Command description	Writes requested value to the requested register
PARAM1	Address of requested register in decimal or hex (with 0x) format
PARAM2	Data to write to the register in decimal or hex (with 0x) format
Potential responses	Human readable information about status of operation
Example usage	hwtest hf config rc663 reg write 0x00 0x01
Example response	Addr: 0x00, Data: 0x01 - Write OK

config rc663 eeprom read

Command frame	hwtest hf config rc663 eeprom read PARAM	
Command description	Reads requested value from RC663 eeprom	
PARAM	0_I14443A_106 1_I14443A_212 2_I14443A_424 3_I14443A_848 4_I14443B_106 5_I14443B_EMVCO_106 6_I14443B_212 7_I14443B_424 8_I14443B_848 9_FELICA_212 10_FELICA_424 11_I15693_HIGH 12_I15693_FAST_HIGH 13_EPC_UID 14_I18000P3M3_DSM2 15_I18000P3M3_DSM4 16_I18000P3M3_QSM2 17_I18000P3M3_QSM4	Eeprom parameter to read
Potential responses	Human readable information about status of operation and read data	
Example usage	hwtest hf config rc663 eeprom read 12_I15693_FAST_HIGH	
Example response	Address: 0x01A8 0x0F - DrvMode 0x14 - TxAmp 0x01 - DrvCon 0x0A - TxL 0x08 - TXCrcPreset 0x08 - RXCrcPreset 0x08 - TxDataNum 0x00 - TxModWidth 0x00 - TxSym10BurstLen 0xC0 - TxWaitCtrl 0xA9 - TxWaitLo 0x0F - TxFrameCon 0x00 - RxSofD 0x02 - RxCtrl 0x9C - RxWait 0x8F - RxThreshold 0x92 - Rcv 0x02 - RxAna	

config rc663 eeprom write

Command frame	hwtest hf config rc663 eeprom write PARAM1 PARAM2 PARAM3	
Command description	Writes requested value from RC663 eeprom	
PARAM1	0_I14443A_106 1_I14443A_212 2_I14443A_424 3_I14443A_848 4_I14443B_106 5_I14443B_EMVCO_106 6_I14443B_212 7_I14443B_424 8_I14443B_848 9_FELICA_212 10_FELICA_424 11_I15693_HIGH 12_I15693_FAST_HIGH 13_EPC_UID 14_I18000P3M3_DSM2 15_I18000P3M3_DSM4 16_I18000P3M3_QSM2 17_I18000P3M3_QSM4	Eeprom setting to modify
PARAM2	0_DrvMode 1_TxAmp 2_DrvCon 3_TxL 4_TXCrcPreset 5_RXCrcPreset 6_TxDataNum 7_TxModWidth 8_TxSym10BurstLen 9_TxWaitCtrl 10_TxWaitLo 11_TxFrameCon 12_RxSofD 13_RxCtrl 14_RxWait 15_RxThreshold 16_Rcv 17_RxAna	Param to modify
PARAM3	Value to store in selected param in hex (with 0x) or decimal format	
Potential responses	Human readable information about status of operation	
Example usage	hwtest hf config rc663 eeprom write 12_I15693_FAST_HIGH 0_DrvMode 0xA	
Example response	Write OK Changes will be reverted after reset unless you update checksum Verified 000001A8: 0A	

config rc663 eeprom defaults read

Command frame	hwtest hf config rc663 eeprom defaults read PARAM	
Command description	Reads default value of requested param	
PARAM	0_I14443A_106 1_I14443A_212 2_I14443A_424 3_I14443A_848 4_I14443B_106 5_I14443B_EMVCO_106 6_I14443B_212 7_I14443B_424 8_I14443B_848 9_FELICA_212 10_FELICA_424 11_I15693_HIGH 12_I15693_FAST_HIGH 13_EPC_UID 14_I18000P3M3_DSM2 15_I18000P3M3_DSM4 16_I18000P3M3_QSM2 17_I18000P3M3_QSM4	Eeprom parameter to read
Potential responses	Human readable information about status of operation and read data	
Example usage	hwtest hf config rc663 eeprom defaults read 12_I14443A_106	
Example response	Address: 0x01A8 0x0F - DrvMode 0x14 - TxAmp 0x01 - DrvCon 0x0A - TxL 0x08 - TXCrcPreset 0x08 - RXCrcPreset 0x08 - TxDataNum 0x00 - TxModWidth 0x00 - TxSym10BurstLen 0xC0 - TxWaitCtrl 0xA9 - TxWaitLo 0x0F - TxFrameCon 0x00 - RxSofD 0x02 - RxCtrl 0x9C - RxWait 0x8F - RxThreshold 0x92 - Rcv 0x02 - RxAna	

config rc663 eeprom defaults write

Command frame	hwtest hf config rc663 eeprom defaults write PARAM1	
Command description	Writes selected default value to RC663 eeprom	
PARAM1	0_I14443A_106 1_I14443A_212 2_I14443A_424 3_I14443A_848 4_I14443B_106 5_I14443B_EMVCO_106 6_I14443B_212 7_I14443B_424 8_I14443B_848 9_FELICA_212 10_FELICA_424 11_I15693_HIGH 12_I15693_FAST_HIGH 13_EPC_UID 14_I18000P3M3_DSM2 15_I18000P3M3_DSM4 16_I18000P3M3_QSM2 17_I18000P3M3_QSM4 all	Eeprom parameter to write
Potential responses	Human readable information about status of operation and read data	
Example usage	hwtest hf config rc663 eeprom defaults write 12_I15693_FAST_HIGH	
Example response	Write OK Verified 000001A8: 0F 14 01 0A 08 08 08 00 00 C0 0F 02 8F 92 02 Changes will be reverted after reset unless you update checksum	

config rc663 eeprom update_crc

Command frame	hwtest hf config rc663 eeprom update_crc
Command description	Recalculates RC663 eeprom crc and writes it into the RC663 chip memory
Potential responses	Human readable information about status of operation
Example usage	hwtest hf config rc663 eeprom update_crc
Example response	RC663 EEPROM checksum updating... Ok

print

Command frame	hwtest hf print
Command description	Prints current hf test mode configuration
Potential responses	Human readable information about current hf test mode configuration
Example usage	hwtest hf print
Example response	Mode: CARRIER Protocol: ISO14443A Airspeed: B106 Loop iteration delay [ms]: 200 Started: No Pattern: 0A0C0E

start

Command frame	hwtest hf start
Command description	Starts currently configured HF test mode
Potential responses	Human readable information about status of operation and results of test
Example usage	hwtest hf start
Example response	Hf test mode task started. HF carrier turned on.

stop

Command frame	hwtest hf stop
Command description	Stops currently configured HF test mode
Potential responses	Human readable information about status of operation
Example usage	hwtest hf stop
Example response	hwtest hf stop. HF carrier turned off.

HW test start and stop commands

start

Command frame	hwtest start
Command description	Starts hw test mode
Potential responses	Human readable information about test mode
Example usage	hwtest start
Example response	Entering test mode.

stop

Command frame	hwtest stop
Command description	Stops hw test mode and resets reader
Potential responses	Human readable information about test mode
Example usage	hwtest stop
Example response	Exiting test mode.

Tamper

Introduction

The reader has one tamper pin available (denoted on the schematic as nTamper). The tamper pin states are defined as HIGH (default state) and LOW (active state). Handling of the tamper detection event depends on the reader operation mode.

In OSDP mode, when the state of the tamper pin is changed, the local status report with the new tamper pin state is generated. This report is internally queued and sent as a response to the `osdp_POLL` command. In addition, the tamper state can be checked at any time by sending the `osdp_LSTAT` command.

In other reader operation modes, whenever the tamper pin state is changed to active, the system event `ATTACK DETECTED` is generated. This event can be bonded with a custom AV sequence to signal its occurrence.

Configuration

Introduction

The OMNIKEY SE Reader Core can be configured using the OMNIKEY Reader Manager desktop application or OMNIKEY SDK. There are two ways to access the OMNIKEY SE Reader Core OID items for configuring the reader. It is possible to configure OMNIKEY SE Reader Core by the configuration file generated with OMNIKEY Reader Manager or by external API commands.

All OMNIKEY SE Reader Core configurable items are identified by a unique object identifier (OID). A full description is given in the following sections, including default values and example get and set commands.

- When you change the configuration, the reader validates only the correctness of the data, that is the data type, length and value format.
- Setting configuration values that are logically incorrect may result in unpredictable behavior.

Configuration file

Once a configuration file is generated in OMNIKEY Reader Manager, it can be uploaded and processed by the reader in the same way as any other binary file, as described in "Reader update" on page 277.

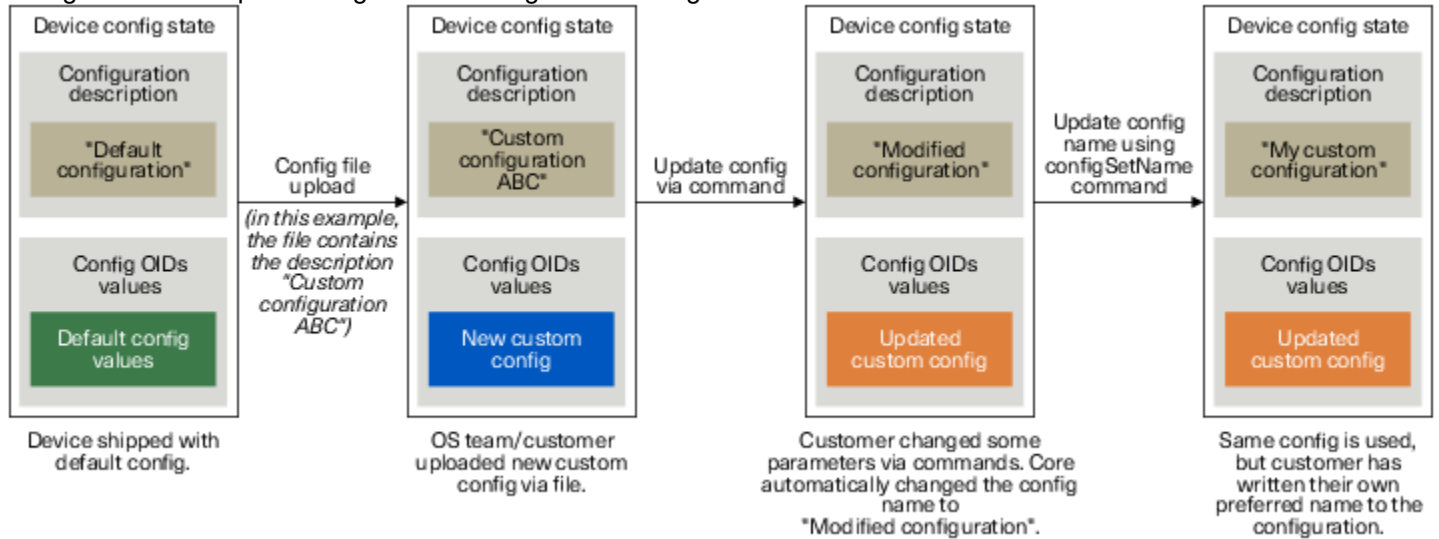
Configuration using External Reader API

To change any configuration parameter, the reader must operate in External Reader API mode. Retrieving or setting a configuration value can be performed by sending an appropriate request. See "Config commands" on page 182 for available requests and payload structures.

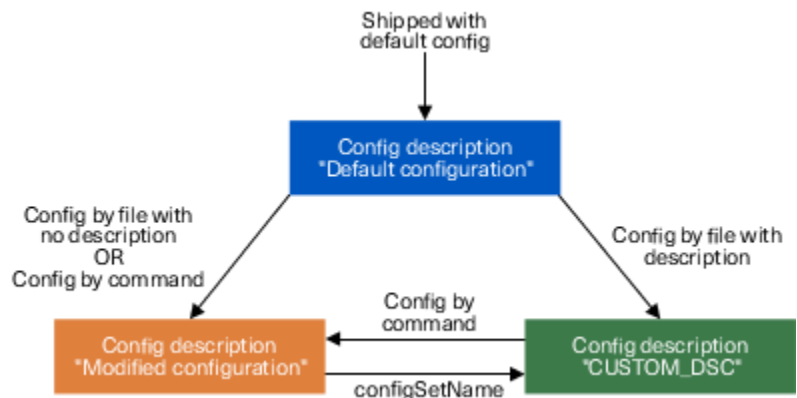
After changing any value, the reader must be reset for the changes to take effect.

Configuration description

The reader core configuration allows a description to be added for the user configuration set. This description, when shipped, is filled with the default configuration name used for device provisioning. Whenever you change the configuration (using a configuration file or a command) the description is updated. The following diagram presents the lifecycle of the configuration description alongside the configuration change.



The value of the configuration description also depends on the manual configuration parameters that have changed. As shown in the following diagram, if a configuration item value is manually updated (using a command, instead of a config file) the configuration is not equal to the one which has been uploaded previously, so its name is automatically changed to "Modified configuration". It is still possible to name a particular configuration using the "configSetName" on page 191 command. It is also possible that the configuration file will be shipped without the description. In this case, the reader will also automatically change the configuration to "Modified configuration". You can retrieve the current configuration description using the "configGetName" on page 190 command. The default configuration name is set to "Default configuration".



Configuration parameters

The following sections describe the OID list and possible values.

Configuration structures

OMNIKEY SE Reader Core structure

Root	Branch
config(1)	reader (1)
	autonomous (2)
	pinger (3)
	lf (4)
	communication (5)
	security (6)
	io (7)
	wiegand (8)
	osdp (9)
	ble (10)
	keypad (11)

The following sections describe every leaf in detail, with default values where appropriate. The following table shows an example OID number of the "HF pinger pulse length configuration": 1.1.1.3.1.4.

HID ROOT	ROOT	BRANCH			
hid (1)	omnikey (1)	configure (1)	pinger (3)	hf (1)	pulse_len (4)

OID access

Configuration items are set and retrieved using an OMNIKEY SE Reader Core specific external API command for OID access. There are two external API commands: "configSetParam " on page 185 and "configGetParam " on page 183.

General configuration structure

This section describes the OID items that contain information about the general reader configuration.

HID ROOT	root	branch			
hid (1)	omnikey (1)	configure (1)	reader (1)	generic (1)	default_mode (1) Reader_friendly_name(2)

Default mode

Relative OID	1.1.1
Access	Read-write
Type	ENUM
Value	0 = EXTERNAL API 1 = AUTONOMOUS 4 = OSDP
Description	OMNIKEY SE Reader Core default mode
Default value	1 = AUTONOMOUS

Reader friendly name

Relative OID	1.1.2
Access	Read-write
Type	BYTES
Length	0 ÷ 50
Value	Byte string consisting reader friendly name
Description	Reader friendly name
Default value	EMPTY

Autonomous mode configuration structure

This section describes the OID items that contain information about the Autonomous mode configuration.

HID ROOT	ROOT	Branch
		formatter (1)
		interface (1)
hid (1)	omnikey (1)	config (1)
		autonomous(2)
		cardout_en (2)
		cpe_rf_prio (3)
		cpe_enhanced_poll (4)
		cpe_hf_config (5)
		lf_modulations_config (6)
		enable_ble_on_pinger (8)
		cpe_polling_mode (9)
		output_delay (10)
		any_card (11)
		cpe_hf (3)
		slot0 (0)
		slot1 (1)
		slot2 (2)
		slot3 (3)
		slot4 (4)
		slot5 (5)
		slot6 (6)
		slot7 (7)
		slot8 (8)
		slot9 (9)
		cpe_lf (4)
		slot0 (0)
		slot1 (1)
		slot2 (2)
		slot3 (3)
		slot4 (4)
		slot5 (5)
		slot6 (6)

hid (1)	omnikey (1)	config (1)	autonomous(2)	formatter (6)	slot7 (7)
					slot8 (8)
					slot9 (9)
					uart_format (0)
					uart_encoding (1)
					mifare_facility_code (2)
					credential0 (0)
					credential1 (1)
					credential2 (2)
					credential3 (3)
					credential4 (4)
					credential5 (5)
					credential6 (6)
					credential7 (7)
					credential8 (8)
					credential9 (9)

Output interface

Relative OID	2.1.1
Access	Read-write
Type	ENUM
Value	0 = UART 1 = Wiegand
Description	Autonomous mode output interface
Default value	0 = UART

Card output enabled

Relative OID	2.1.2
Access	Read-write
Type	BOOL
Value	0 = False 1 = True
Description	Card output will be sent only if this flag is set (true). This output (if enabled) is sent when the reader cannot communicate anymore with the previously detected card.
Default value	1 = True

Card processing engine RF priority

Relative OID	2.1.3
Access	Read-write
Type	ENUM
Value	1 = HF priority 2 = LF priority
Description	Configures RF priority for the Card Processor Engine.
Default value	1 = HF priority

Card processing engine enhanced polling

Relative OID	2.1.4
Access	Read-write
Type	BOOL
Value	0 = Disable 1 = Enable
Description	Configures enhanced polling for the Card Processor Engine.
Default value	0 = Disabled

Card processing HF protocols order configuration

Relative OID	2.1.5
Access	Read-write
Type	BYTES
Length	2 to 6
Value	0 to 4
Description	Contains a CBOR encoded array of HF protocols priority configuration. Each value represents a single protocol. The order of protocol scanning will be from the first item to the last item. For details, see "HF protocols configuration" on page 73. Cannot be empty.
Default value	h'820104' ORDER: ISOA, iCLASS

Polling mode

Relative OID	2.1.9
Access	Read-write
Type	BOOL
Value	0 = Disable 1 = Enable
Description	Configures polling mode for the Card Processor Engine. See "Features" on page 75.
Default value	0 = Disable

Delay before sending data in Autonomous mode

Relative OID	2.1.10
Access	Read-write
Type	INTEGER
Value	0 to 50
Description	The time that the reader waits between detecting a credential and sending its data. Unit: milliseconds.
Default value	0

Autonomous output when data reading fails

Relative OID	2.1.11
Access	Read-write
Type	BOOL
Value	0 = Disable 1 = Enable
Description	Configures if any card output will be present if the card does not contain configured data.
Default value	0 = Disable

Period of polling mode

Relative OID	2.1.12
Access	Read-write
Type	INTEGER
Value	20 ÷ 1000
Description	Configures value of polling period for polling mode.
Default value	200 = 200ms

Indala formatter configuration

Relative OID	2.1.13
Access	Read-write
Type	BYTES
Value	165, 259
Description	Indala formatter configuration.
Default value	A261301903EA613158930003001AFF03EA0219110COF1015 071A0A0D0B08140403130E010516181B120906FFFFFFFFFFFF FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFBAE28C84C33B68 78F8B9CESC42F49D413A823C3474DF1F288C6AAEFOE6A84D C5D90C7DCODC032500

Number of ECP retries before determining that no mobile device is present

Relative OID	2.1.14
Access	Read-write
Type	INTEGER
Value	1 ÷ 6
Description	Configures how many times EPC polling is retried before determining that no mobile device is present (6 as required by the specification).
Default value	1

Minimum ECP polling duration

Relative OID	2.1.15
Access	Read-write
Type	INTEGER
Value	0 ÷ 600
Description	Configures how many times EPC polling is retried before determining that no mobile device is present (6 as required by the specification).
Default value	0

Card processing engine HF slot 0

Relative OID	2.3.0
Access	Read-write
Type	BYTES
Length	0, 12 to 200
Value	Content consists of a CBOR encoded card data readout procedure.
Description	For details see "Card data procedures" on page 76.
Default value	A261300661318144A1613000 Procedure to read PACS for Seos.

Card processing engine HF slot 1

Relative OID	2.3.1
Access	Read-write
Type	BYTES
Length	0, 12 to 200
Value	Content consists of a CBOR encoded card data readout procedure.
Description	For details see "Card data procedures" on page 76.
Default value	A261300461318144A1613000 Procedure to read PACS for MIFARE DESFire.

Card processing engine HF slot 2

Relative OID	2.3.2
Access	Read-write
Type	BYTES
Length	0, 12 to 200
Value	Content consists of a CBOR encoded card data readout procedure.
Description	For details see "Card data procedures" on page 76.
Default value	A261300161318144A1613000 Procedure to read PACS for MIFARE Classic 1k.

Card processing engine HF slot 3

Relative OID	2.3.3
Access	Read-write
Type	BYTES
Length	0, 12 to 200
Value	Content consists of a CBOR encoded card data readout procedure.
Description	For details see "Card data procedures" on page 76.
Default value	A261300261318144A1613000 Procedure to read PACS for MIFARE Classic 4k.

Card processing engine HF slot 4

Relative OID	2.3.4
Access	Read-write
Type	BYTES
Length	0, 12 to 200
Value	Content consists of a CBOR encoded card data readout procedure.
Description	For details see "Card data procedures" on page 76.
Default value	A261300061318144A1613000 Procedure to read PACS for PicoPass.

Card processing engine HF slot 5-9

Relative OID	2.3.5-9
Access	Read-write
Type	BYTES
Length	0, 12 to 200
Value	Content consists of a CBOR encoded card data readout procedure.
Description	For details see "Card data procedures" on page 76.
Default value	Empty

Card output UART format

Relative OID	2.6.0
Access	Read-write
Type	ENUM
Value	0 = RAW 1 = ASCII
Description	Autonomous mode output format when the UART interface is enabled as the output interface.
Default value	0

Card output UART encoding

Relative OID	2.6.1
Access	Read-write
Type	ENUM
Value	0 = Binary 1 = Text
Description	Autonomous mode data encoding format when the UART interface is enabled as the output interface.
Default value	0

Card output formatter facility code

Relative OID	2.6.2
Access	Read-write
Type	INTEGER
Value	0 to 255
Description	Facility code value to be used by (mostly MIFARE) data formatter.
Default value	0

Card output data formatter slot 0-9

Relative OID	2.7.0 - 2.7.9
Access	Read-write
Type	BYTES
Length	0, 12 to 200
Value	See "Card formatter configuration structure" on page 50.
Description	Card data formatter slots. For details, see "Data formatter" on page 50.
Default value	Empty.

Pinger configuration structure

This section describes the OID items that contain information about the pinger configuration.

HID ROOT		ROOT		BRANCH	
hid (1)	omnikey (1)	configure (1)	pinger (3)	general (0)	period (1)
				hf (1)	threshold (1)
					coefs (2)
					pulse_len (4)
					adc_delay (5)
				lf (2)	threshold (1)
					coefs (2)
					pulse_len (4)
					adc_delay (5)

Pinger period

Relative OID	3.0.1
Access	Read-write
Type	INTEGER
Value	10, 25, 50, 100, 125, 175, 250, 500, 1000, 2000, 3000, 4000, 5000
Description	Period of the pinger polling in milliseconds.
Default value	10

HF pinger threshold

Relative OID	3.1.1
Access	Read-write
Type	INTEGER
Value	5 to 255, step =1
Description	Card detection threshold for the DSP processed pinger signal. If larger than this value, the pinger informs the application about HF detection.
Default value	7

HF pinger algorithm coefficients

Relative OID	3.1.2
Access	Read-write
Type	BYTE
Length	3 to 5
Value	CBOR encoded 10 to 254
Description	Contains a CBOR array with two integer elements with coefficients for the low pass filters used by the card detection DSP algorithm. Unit: coefficient.
Default value	82185210 coefficient for the 1st low pass filter = 82 coefficient for the 2nd low pass filter = 16

HF pinger pulse length

Relative OID	3.1.4
Access	Read-write
Type	INTEGER
Value	100, 125, 188, 250, 500
Description	Length of HF pinger pulse in nanoseconds.
Default value	125

HF pinger ADC sampling delay

Relative OID	3.1.5
Access	Read-write
Type	INTEGER
Value	3 to 16, step = 1
Description	Delay used for delaying ADC sampling after pulse generation (in microseconds).
Default value	4

Communication configuration structure

This section describes the OID items that contain information about communication interface configuration.

HID ROOT	root		branch		
hid (1)	omnikey (1)	config (1)	communication (5)	general (1)	serial_baudrate (1)
					control_flow (4)
				front_end (2)	BLE_enable (1)

HID ROOT	root		branch	HF_enable (2)
				LF_enable (3)

Serial baud rate

Relative OID	5.1.1
Access	Read-write
Type	ENUM
Value	0 = 9600 1 = 19200 2 = 57600 3 = 115200 4 = 230400
Description	Serial port baud rate.
Default value	3

Serial baud rate configuration does not affect Shell mode interface. For more details see section "Shell interface" on page 208.

Control flow

Relative OID	5.1.4
Access	Read-write
Type	BOOL
Value	0 = Disable 1 = Enable
Description	Enables/disables RTS/DE HW lines for Autonomous and External Reader API modes.
Default value	0 = Disable

Serial baud rate configuration does not affect Shell mode interface. For more details see section "Shell interface" on page 208.

HF enable

Relative OID	5.2.2
Access	Read-write
Type	BOOL
Value	0 = Disable 1 = Enable
Description	Enables/disables HF polling.
Default value	1 = Enable

Security configuration structure

This section describes the OID items that contain information about the security features configuration.

HID ROOT	ROOT	BRANCH			
hid (1)	omnikey (1)	config (1)	security (6)	anti-passback (1)	anti-passback_enable (1)
					anti-passback_delay (2)
					anti-passback_sliding_window (3)
				card_tracking (2)	card_tracking_enable (1)
					constant_card_report (2)
					constant_card_report_interval (3)
				timeout_settings (3)	wireless_config_timeout (2)
				heartbeat (4)	heartbeat_en (0)
					heartbeat_interval (1)
					heartbeat_wiegand_msg (2)
					heartbeat_wiegand_reset_msg (3)

Anti-passback enable

Relative OID	6.1.1
Access	Read-write
Type	INTEGER
Value	0 = Disabled 1 = Enabled, CSN 2 = Enabled, CREDENTIAL_DATA
Description	If CSN is selected, anti-passback verifies the CSN value. Credential data (like PACS) is verified if CREDENTIAL_DATA was selected.
Default value	0 = Disabled

Anti-passback delay

Relative OID	6.1.2
Access	Read-write
Type	INTEGER
Value	100 to 20000
Description	Sets the anti-passback delay. The same card cannot be used for the defined duration. Unit: milliseconds.
Default value	500 = 500 ms

Anti-passback sliding window

Relative OID	6.1.3
Access	Read-write
Type	BOOL
Value	0 = OFF 1 = ON
Description	Turns ON / OFF the anti-passback sliding window. If enabled, a delay timer is restarted every time the same credential is detected.
Default value	0 = OFF

Card tracking enable

Relative OID	6.2.1
Access	Read-write
Type	BOOL
Value	0 = OFF 1 = ON
Description	Turn ON / OFF automatic tracking of the card after readout to avoid repetitive reads of the same card.
Default value	1 = ON

Constant card report

Relative OID	6.2.2
Access	Read-write
Type	BOOL
Value	0 = OFF 1 = ON
Description	Turn ON / OFF constant data report. If turned on, reader will repeat the same card data as long as the card is kept presented to the reader.
Default value	0 = OFF

Constant card report interval

Relative OID	6.2.3
Access	Read-write
Type	INTEGER
Value	100 - 1000
Description	Defines the interval of subsequent data reports (card tracking reports).
Default value	200 = 200ms

Heartbeat enable

Relative OID	6.4.0
Access	Read-write
Type	BOOL
Value	0 = OFF 1 = ON
Description	Turn ON / OFF heartbeat functionality.
Default value	0 = OFF

For Autonomous mode, heartbeat is sent on the interface which is selected via "Autonomous mode output interface" on page 45 configuration (if data is sent over UART then heartbeat is also sent over UART, similarly for Wiegand).

Heartbeat interval

Relative OID	6.4.1
Access	Read-write
Type	INTEGER
Value	1 ÷ 3600
Description	Defines the interval of heartbeat message. Unit - seconds.
Default value	5

Heartbeat Wiegand message

Relative OID	6.4.2
Access	Read-write
Type	BYTES
Length	1 ÷ 2
Value	Byte string containing Wiegand message.
Description	Defines the Wiegand message which shall be sent for every heartbeat event.
Default value	AA

Heartbeat Wiegand reset message

Relative OID	6.4.3
Access	Read-write
Type	BYTES
Length	1 ÷ 2
Value	Byte string containing Wiegand message.
Description	Defines the Wiegand message for first heartbeat after reader reset.
Default value	CC

I/O configuration structure

This section describes the OID items that contain information about the I/O configuration.

HID ROOT	ROOT			BRANCH	
hid (1)	omnikey (1)	config (1)	io (7)	general (1)	idle_led (0)
					sequence_1 (1)
					sequence_2 (2)
					sequence_3 (3)
					sequence_4 (4)
					sequence_5 (5)
					sequence_6 (6)
					sequence_7 (7)
					sequence_8 (8)
					sequence_9 (9)
					sequence_10 (10)
					sequence_11 (11)
					sequence_12 (12)
					sequence_13 (13)
					sequence_14 (14)
				bonding (2)	event_bonding (0)
				gpio (3)	gpio0 (0)
					gpio1 (1)
					osdp (5)

Idle LED color

Relative OID	7.1.0
Access	Read-write
Type	ENUM
Value	0 = OFF 1 = RED 2 = GREEN
Description	Idle LED color
Default value	1 (RED)

Definition of I/O AV sequence

Relative OID	7.1.1
Access	Read-write
Type	BYTES
Length	0, 17 to 209
Value	Byte string comprising a CBOR encoded audiovisual sequence.
Description	CBOR encoded audiovisual sequence. For details, see the sequences type definition in "ioSetAudioVisualSequence (deprecated)" on page 128. If you configure this parameter with an empty value, the slot is cleared.
Default value	A1613081A46130006131006132518285 001832001832038500000019012C0161 3301

Definition of I/O AV sequence

Relative OID	7.1.2
Access	Read-write
Type	BYTES
Length	0, 17 to 209
Value	Byte string comprising a CBOR encoded audiovisual sequence.
Description	See "I/O configuration structure " on page 261.
Default value	A2613081A46130006131006132518285 001832001832038500000019012C0161 33006131A36130006131488185001896 000A01613200

Definition of I/O AV sequence

Relative OID	7.1.3
Access	Read-write
Type	BYTES
Length	0, 17 to 209
Value	Byte string comprising a CBOR encoded audiovisual sequence.
Description	See "I/O configuration structure " on page 261.
Default value	A1613081A46130006131016132488185 001819000A01613300

Definition of I/O AV sequence

Relative OID	7.1.4
Access	Read-write
Type	BYTES
Length	0, 17 to 209
Value	Byte string comprising a CBOR encoded audiovisual sequence.
Description	See "I/O configuration structure " on page 261.
Default value	A2613081A46130006131016132498185 00181E00181E036133006131A3613000 613148818500181E000A01613200

Definition of I/O AV sequence

Relative OID	7.1.5
Access	Read-write
Type	BYTES
Length	0, 17 to 209
Value	Byte string comprising a CBOR encoded audiovisual sequence.
Description	See "I/O configuration structure " on page 261.
Default value	A1613081A46130006131006132518285 00181E00181E038500000019012C0161 3300

Definition of I/O AV sequence

Relative OID	7.1.6
Access	Read-write
Type	BYTES
Length	0, 17 to 209
Value	Byte string comprising a CBOR encoded audiovisual sequence.
Description	See "I/O configuration structure " on page 261.
Default value	A1613082A46130006131006132498185 00181E00181E03613300A46130006131 01613249818500181E00181E03613300

Definition of I/O AV sequence

Relative OID	7.1.7
Access	Read-write
Type	BYTES
Length	0, 17 to 209
Value	Byte string comprising a CBOR encoded audiovisual sequence.
Description	See "I/O configuration structure " on page 261.
Default value	A2613081A46130006131016132498185 001832001832036133006131A3613000 613149818500183200183203613200

Definition of I/O AV sequence

Relative OID	7.1.8 to 7.1.14
Access	Read-write
Type	BYTES
Length	0, 17 to 209
Value	Byte string comprising a CBOR encoded audiovisual sequence.
Description	See "I/O configuration structure " on page 261.
Default value	EMPTY

Bonding between events and I/O AV sequences

Relative OID	7.2.0
Access	Read-write
Type	BYTES
Length	0 to 44
Value	Byte string containing a list of defined bondings.
Description	Content is a list of defined bondings. Each bonding is a record (group) consisting of three integers where: • First integer represents AV event ID (range: 0 - 10, see "System events" on page 38). • Second integer represents number of AV sequence slot (range: 0 - 14). • Third integer represents bonding priority (range: 0 - 4).
Default value	An empty value removes all bondings. 95090100080201040303070402050502020603030702 (see "Audio-visual sequences" on page 36).

User GPIO0 pin configuration

Relative OID	7.3.0
Access	Read-write
Type	ENUM
Value	0 = OUTPUT 1 = INPUT 2 = INPUT with pull-down 3 = INPUT with pull-up
Description	Configures direction and setup of a user pin.
Default value	0 = OUTPUT

User GPIO1 pin configuration

Relative OID	7.3.1
Access	Read-write
Type	ENUM
Value	0 = OUTPUT 1 = INPUT 2 = INPUT with pull-down 3 = INPUT with pull-up
Description	Configures direction and setup of a user pin.
Default value	1 = INPUT

Assignment of user GPIOs to OSDP

Relative OID	7.3.5
Access	Read-write
Type	ENUM
Value	0 = No pins used by OSDP 1 = GPIO0 used by OSDP 2 = GPIO1 used by OSDP 3 = GPIO0 and GPIO1 used by OSDP
Description	Determines if user-configurable pins should be used by OSDP. If the given pin is assigned to OSDP, its direction is determined by an appropriate pin configuration, and it cannot be used by ERA.
Default value	3 = GPIO0 and GPIO1 used by OSDP

Wiegand configuration structure

This section describes the OID items that contain information about the Wiegand configuration.

HID ROOT	ROOT	BRANCH				
hid (1)	omnikey (1)	config (1)	wiegand (8)	general (0)	pulse (0)	
						space (1)
						guard (2)

Wiegand interface pulse width

Relative OID	8.0.0
Access	Read-write
Type	INTEGER
Value	20 to 200, step 10
Description	Wiegand signal pulse width. Unit: microseconds.
Default value	40 = 40 μ s

Wiegand interface space width

Relative OID	8.0.1
Access	Read-write
Type	INTEGER
Value	200 to 20000, step 50
Description	Wiegand signal space width (bit time). Unit: microseconds.
Default value	2000 = 2 ms

Wiegand interface guard time

Relative OID	8.0.2
Access	Read-write
Type	INTEGER
Value	10 to 200, step 10
Description	Wiegand signal guard time. Unit: milliseconds.
Default value	10 = 10 ms

OSDP configuration structure

This section describes the OID items that contain information about the OSDP configuration.

HID ROOT	ROOT	BRANCH			
hid (1)	omnikey (1)	config (1)	osdp (9)	general (0)	baudrate (0)
					address (1)
					enforce secure (2)

OSDP interface baudrate

Relative OID	9.0.0
Access	Read-write
Type	ENUM
Value	0 = 9600 1 = 19200 2 = 38400 3 = 57600 4 = 115200 5 = 230400
Description	Configures the OSDP interface baud rate.
Default value	0 = 9600 baud

OSDP interface address

Relative OID	9.0.1
Access	Read-write
Type	INTEGER
Value	0 to 127
Description	Configures the OSDP reader address.
Default value	0

OSDP enforce secure

Relative OID	9.0.2
Access	Read-write
Type	BOOL
Value	0 = SC is not required 1 = SC is required
Description	Flag indicating whether a secure channel (SC) is required for communication.
Default value	0 = SC is not required

Connect Timer Interval

Relative OID	10.0.1
Access	Read-write
Type	INTEGER
Value	100 to 10000
Description	Application-specific timer guarding interval between the moment the connection is established and the first application-specific message arrives.
Default value	4030 = 4 s 30 ms

Allowed Advertising Interval Range

Relative OID	10.1.0
Access	Read-write
Type	BYTES
Value	0x20 to 0x4000 (must be less than the Maximal Advertising Interval)
Description	Contains a CBOR array with two integer elements linked to advertising, coded in 0.625 ms units. Allowed values range: 0x20 to 0x4000. Minimal Advertising Interval must be less than the Maximal Advertising Interval. Allowed byte string length: 5 to 7. Cannot be empty.
Default value	8218201828 min. advertising interval = 32 max. advertising interval = 40

Fragment Timer Interval

Relative OID	10.1.6
Access	Read-write
Type	INTEGER
Value	100 to 10000
Description	Timeout for error detection on the applicable fragmentation layer.
Default value	300ms

Message Timer Interval

Relative OID	10.1.7
Access	Read-write
Type	INTEGER
Value	100 to 10000
Description	Timeout for error detection on the applicable APDU layer in milliseconds.
Default value	1500 = 1.5 s

Include PPCP flag

Relative OID	10.1.9
Access	Read-write
Type	BOOL
Value	0 = OFF 1 = ON
Description	This binary flag relates to the presence of the GAP "Peripheral Preferred Connection Parameters" Characteristic in the GATT Server stack of Seos Lock.
Default value	1 = ON

Custom keys handling

Introduction

The OMNIKEY SE Reader Core provides the ability to read any custom data from any supported credential. To achieve custom data handling, custom keys must be injected into the device. The reader supports functionalities to inject any custom key using either:

- An ERA command.
- A config file.

To make custom key injection secure, the key must be encrypted with another encryption key called the Key-Encryption-Key (KEK). The device is provided to the customer with a default AES KEK. Since this key is distributed to customers, it is not safe to use it to inject any other key. Before any other key can be changed, the default key must be changed. Due to distribution of the default key, it is highly recommended to change the default key to a customer key in a secure environment.

AES KEK usage

Default AES KEK rolling

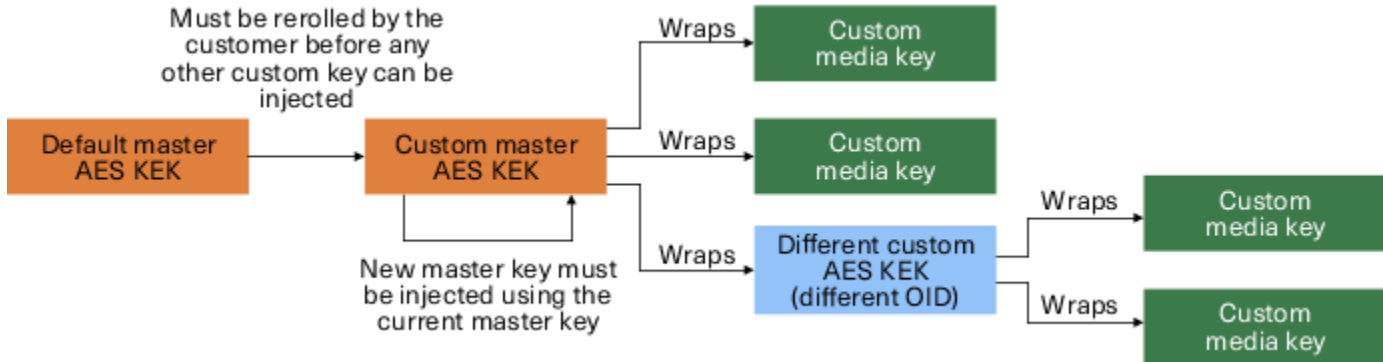
Before any other key can be injected, the default AES KEK key must be rolled using the "rollDefaultWrappingKey " on page 165 command as follows:

- When delivered, the reader contains default keys, and you cannot inject any custom keys until the default keys are rolled. No action is forced during the first power-on, but custom key injection is blocked.
- Changing the default AES KEK is possible only through the ERA command "rollDefaultWrappingKey " on page 165.
- Once the AES KEK is changed, custom key injection is unblocked. The reader will now accept config files which include wrapped custom keys, and the "setUserKey " on page 167 command may also be used.

Custom AES KEKs injection

When the default AES KEK is rolled, it is possible to inject other AES KEKs. The "setUserKey " on page 167 command is used, as for any other key. Other AES KEKs can be used to wrap other keys, so it can be used to diversify access to different keys at organization level. The master key can be rolled only using the currently uploaded master key.

KEK key wrapping



AES KEKs OIDs

When a new AES KEK is injected, it must be injected to the OID as stated in the memory map shown in the following table.

Key reference min	Key reference max	Description
0x030100	0x030119	OIDs reserved for user media keys.
0x03F000	0x03F00F	OIDs reserved for AES KEKs. You can use any of these OIDs as a target for new AES KEKs or as a wrapping key for another key. 0x03F000 is the OID of the default AES KEK. The device is shipped with the default value of AES KEK.

Only keys in the range of these OIDs can be used to wrap/unwrap other keys. If an empty (not set) OID is used, the reader rejects the operation.

User Key encoding

This section describes the structure of the user key encoding format that also applies to custom user key encoding.

Structure	Description
<code>user_key_package = { "0" : target_key_ref, "1" : wrap_key_ref, "2" : key, ? "3" : key_id }</code>	User key package.
<code>target_key_ref = bstr .size 3</code>	New key reference. If the operation is executed properly, the new key is accessible under this key reference.
<code>wrap_key_ref = bstr .size 3</code>	Key reference of a key which needs to be used to decrypt the new key.
<code>key = bstr</code>	Wrapped key material. Only keys of size 6, 8, 16, 24, or 32 will be accepted. If target_key_ref is AES_KEK (to inject a new AES KEK) then the key size must be of length = 16/24/32B.
<code>key_id = bstr .size 8</code>	ID of the key; customer handles the value. Length = 8.

Key ID

You might need to track which key has been uploaded to particular device. For this purpose, it is possible to set the key ID during key injection (and default key rolling). Later, it is possible to check the type and ID of a particular key using the "getUserKeyInfo " on page 169 command. The key ID is optional, so if no key ID has been set, no key ID is returned.

Reader update

Introduction

The OMNKEY SE Reader Core is capable of updating its configuration or firmware using a binary update file. The update file is a single binary file capable of storing firmware or configuration data for one or multiple internal components. It is possible to perform the update in a graphical environment using the OMNIKEY Reader Manager tool. Additionally, an update over UART can be performed manually using the External Reader API or the embedded SDK.

Update over ERA

This section describes the process of uploading the file and executing the update manually using the External Reader API. The procedure for configuration and firmware updates is identical, so the same set of instructions should be followed.

Update requirements

To perform an update, the following requirements must be met:

- The OMNIKEY SE Reader Core is operating in External Reader API mode (see "Modes of operation" on page 11).
- You have the update file ready, which should be provided by HID. The update file must be compatible with both the device type and the current firmware version. In the event of a mismatch, the update file will not be accepted by the device.
- You have established ERA communication, for example, over UART.

Reader update flow

The process of updating the reader can be divided into two phases:

- Uploading the update files.
- Executing the firmware update.

File upload stage

During this stage, the update file is uploaded into the internal memory of the reader. Throughout this process, the reader remains fully operational.

File uploads can be paused and resumed, eliminating the need to complete the process in a single session.

To upload the file:

1. Issue the "fileStore " on page 196 command with "action=INIT."
2. Use the "fileStore " on page 196 command with "action=CHUNK" to upload firmware chunks until the entire file has been transmitted.

If at any time the reader responds with "5.03 Service Unavailable", it indicates that the previous command is still being processed, and you should retransmit the current command.

Initial file validation occurs in real time during the upload of each chunk. As a result, it is possible to receive an error response for any given chunk. In this case, you can use the "fileGetUpdateStatus " on page 200 command to detect the reason for the failure. For detailed troubleshooting instructions, see "Troubleshooting" on page 282

Once the file is fully uploaded, it undergoes validation by the firmware, which may take several seconds. To verify the current status, use the "fileGetUpdateStatus " on page 200 command.

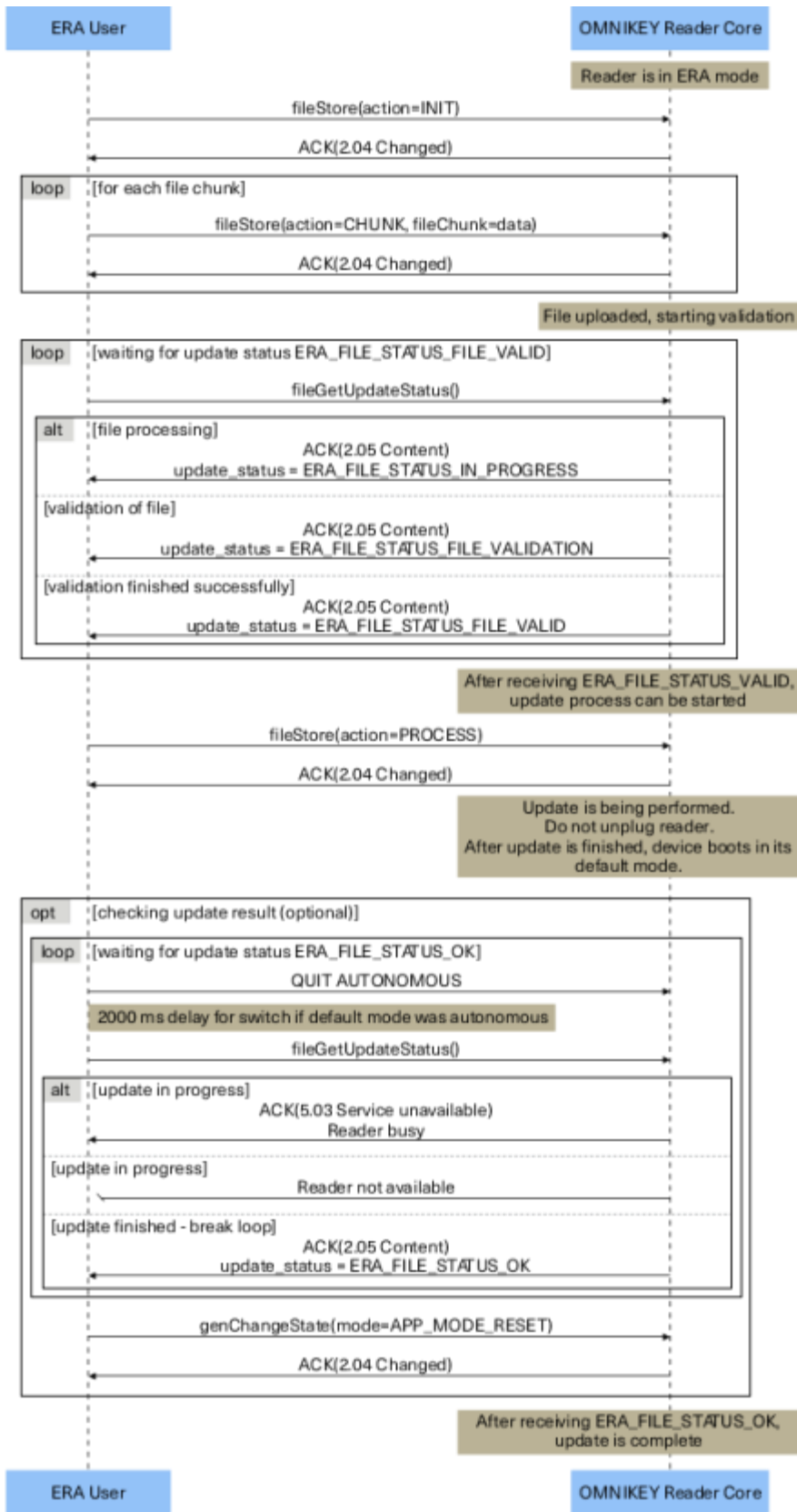
An update status of "ERA_FILE_STATUS_FILE_VALID" indicates successful file validation, allowing you to initiate the upgrade procedure. In the event of encountering any error codes at this stage, see "Troubleshooting" on page 282.

Reader update stage

When the update status is "ERA_FILE_STATUS_FILE_VALID," the reader update can be initiated by sending the "fileStore " on page 196 command with "action=PROCESS". The reader cannot process any other commands during the update procedure, so it may respond with "5.03 Service Unavailable" or not respond at all. The duration of the update can range from a few seconds to up to five minutes.

During the update, the reader may reset itself multiple times. Upon successful completion of the update, the status should change to "ERA_FILE_STATUS_OK." However, if any ERROR status is encountered, see "Troubleshooting" on page 282. The following diagram shows a successful file update flow.

File upload and update execution flow

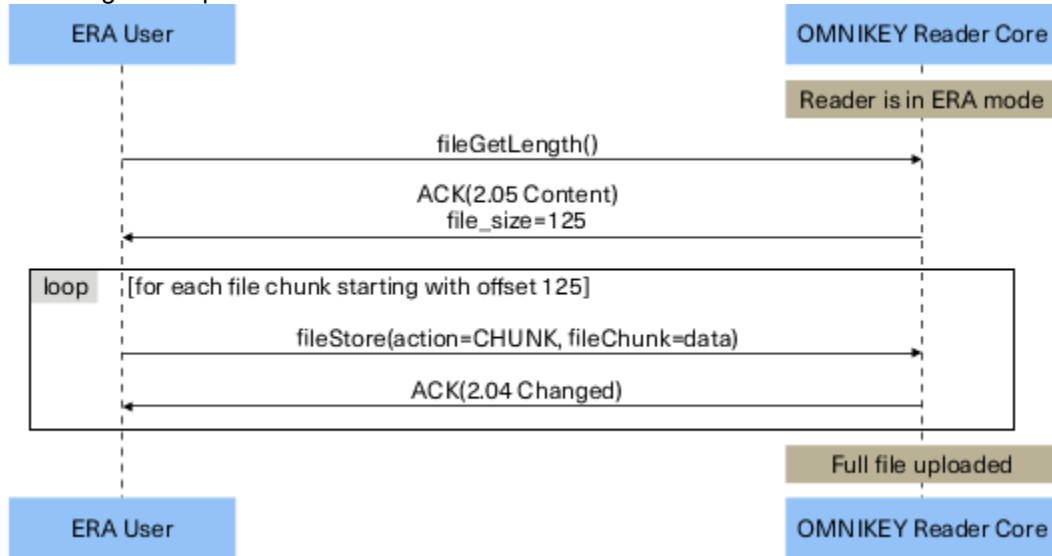


When the update process is complete, the reader automatically resets and boots into a start mode according to its configuration. If the start mode is different than the External Reader API, you should switch to External Reader API and confirm that the update status is "ERA_FILE_STATUS_OK". Once the status is confirmed, the host application should

reset the reader using the "genChangeState " on page 107 command with the "APP_MODE_RESET" parameter. This reset action ensures that the reader operates in its default mode as configured.
It is essential to perform a soft reset after checking the status to ensure proper operation after the update.

File upload resume

The file upload process can be paused at any time. When resuming the process, the "fileGetLength " on page 198 command can be used to determine how much has already been uploaded. The following diagram shows the flow for resuming a file upload.



File upload abort

The file upload process can be aborted at any time using the "fileClear " on page 199 command. This command removes the already uploaded file and resets the update status. After issuing this command, the upload stage should start again from the beginning.

Troubleshooting

This section describes potential reasons for problems that may occur during a reader update. If these suggestions do not resolve the problem, please contact [HID Global Support](#).

ERA status codes to CoAP response codes mapping

Problem	Explanation
Reader responds with 5.03 SERVICE UNAVAILABLE	The reader is busy processing the previous command. Resend the command.
Reader responds with 4.12 PRECONDITION FAILED for "fileStore " on page 196.	The upload has been rejected.
After file upload and validation, update_status is set to ERA_FILE_STATUS * ERROR.	The upload has been rejected.

Possible reasons for rejecting an update file are:

- The file's stored device class is different to the reader device class. The update file set is not intended for this device.
- Minimum version counter requirement is not met. The update file set cannot be applied. An important update may have been skipped.
- Signature validation did not pass. The file may be corrupted.
- The currently running version number is greater than or equal to the one in the update file. The device has already been updated with a newer file.

Security considerations

Introduction

During the integration of the OMNIKEY SE Reader Core into an end device, it is crucial to address the security considerations associated with communication between the embedded device and the host device. It is important to note that the currently implemented protocol does not include encryption or integrity checks for data exchanged between the two devices.

To ensure the overall security of the integrated system, you are strongly advised to focus on the physical security of the connection between the OMNIKEY SE Reader Core and the host device, and also hardening of the end device's housing. Implementing robust physical security measures protects against potential risks from unauthorized access or tampering.

Physical hardening

To mitigate the potential security risk of the interception of data transmitted between the devices, it is important to fortify the physical connection points. Implement the following measures to enhance the robustness of the interface:

- **Enclosure protection**
Ensure that the OMNIKEY SE Reader Core and its connecting cables are housed within a secure, tamper-resistant enclosure. This protects against unauthorized access and physical tampering.
- **Restricted access**
Access to the OMNIKEY SE Reader Core and associated components (physical access to the embedded devices) must be strictly limited and monitored. This prevents unauthorized individuals from directly interacting with the connection points.

Cable security

The integrity of the connecting cables is crucial for securing data communication against interception. The following safeguards are strongly advised:

- **Cable conduit**
Use cable conduits or protective channels to shield connecting cables from environmental factors and potential physical interference.
- **Secure routing**
Route cables in areas inaccessible to unauthorized personnel. Minimize cable exposure to deter tampering or interception.

Regular inspection

Create a scheduled inspection routine to monitor the physical integrity of the OMNIKEY SE Reader Core and host device connection points. Check for any signs of tampering, wear, or damage, and immediately resolve any issues.

Additional security measures

Consider supplementary security measures, such as tamper detection or security cameras, to enhance the overall resilience of the system.

By diligently addressing these physical security aspects, you can significantly improve the overall security of the integrated system. These precautions are essential for mitigating potential risks associated with the absence of communication encryption and integrity checks.

Handling custom keys

Rolling of the default AES KEK key must be executed in a secure, trusted premises so that the new master AES KEK cannot be intercepted. It is highly recommended to roll different master AES KEKs (and also other AES KEKs) to a unique key for each device (key diversification is recommended).

FCC Compliance Information

FCC-ID ZERSEC01

Changes or modifications not expressly approved by Access Ltd could void the user's authority to operate the equipment.

This device complies with part 15 of the FCC Rules. Operation is subject to the following two conditions: (1) This device may not cause harmful interference, and (2) this device must accept any interference received, including interference that may cause undesired operation.

Note:

This equipment has been tested and found to comply with the limits for a Class B digital device, pursuant to part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference in a residential installation. This equipment generates, uses and can radiate radio frequency energy and, if not installed and used in accordance with the instructions, may cause harmful interference to radio communications. However, there is no guarantee that interference will not occur in a particular installation. If this equipment does cause harmful interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try to correct the interference by one or more of the following measures:

- Reorient or relocate the receiving antenna.
- Increase the separation between the equipment and receiver.
- Connect the equipment into an outlet on a circuit different from that to which the receiver is connected.
- Consult the dealer or an experienced radio/TV technician for help.

Introduction

The ATR SE Reader Core is an embedded RFID reader module designed for integration exclusively into ATR215 products manufactured by Access Ltd. **This module is approved under FCC Limited Modular Approval and must not be installed in any other products or by any third parties.**

Applicable FCC Rules

FCC Part 15 Subpart C

CFR 47 15.225 Operation within the band 13.110-14.010MHz

Operational Use Conditions

Host Restriction: Integration is limited to ATR215 products manufactured by Access Ltd.

All ATR215 units share a common hardware architecture, with the only variations being

- The interchangeable interface board that supports different external connection options, such as USB, Ethernet, and RS232 and optional indicator LEDs.
- Alternative housing to accommodate desktop and OEM versions. (OEM version has a smaller interface board without indicator LEDs).

Limited Module Procedures

As described above, the ATR SE Reader core may only be integrated with the ATR215 products.

Antennas / Trace Antenna Design

Only the antenna design and layout with which the module was tested shall be used in all cases. This is the ATR215 antenna which was included in the FCC application.

RF Exposure

As described above, the ATR SE Reader core may only be integrated with the ATR215 products.

RF SAR report C16104TR1 included in the FCC application.

Labelling Information

The ATR215 must have a label stating “Contains FCC ID: ZERSEC01”

Test Plan

For each model of the ATR215, to ensure continued compliance with the FCC rules, the final product must be tested to according to:

- §15.207(a) AC Power Line Conducted Emissions Voltage
- § 15.209 Radiated emission limits; general requirements.
- §15.205, §15.209, 15.225 (a) (b) (c) Band Edge Measurement

Test method ANSIC63.10-2020/Cor1-2023 must be applied

The fundamental emission is not permitted to exceed the original level of 0.002mW.

Grantee Contact Information

The grantee may be contacted regarding C2PC filings or for other information via tanya.newell@hidglobal.com or by phoning +44118 9663333

Revision history

Date	Description	Revision
January 2026	ATR SE Reader Core – Draft. Derived from PLT-06682 A.6 - OMNIKEY SE Reader Core User Guide	01