SPECIFICATION

Calypso Terminal API

Card

Version 3.0.0-SNAPSHOT, 2026-07-20

Warning. The present document cancels and replaces the previous release.

Link and Contact

Website
https://calypsonet.org

Support
support@calypsonet.org

Table of Contents

1. Abstract

This document is the official technical specification of the **Terminal Card API** standardised by the **Calypso Networks Association** (CNA). It defines, in a language-agnostic way, the interfaces, classes, enumerations, exceptions, structural relationships and behavioural requirements that any conforming implementation of the Terminal Card API MUST satisfy.

The Terminal Card API is the **underside** of the **Terminal Reader API** (CNA-TR-API). It defines the **internal** contract used by reader implementations and by card extension modules to exchange APDU-level data with cards, while the **Terminal Reader API** remains the public-facing surface used by terminal applications.

Document Status

Reference	YYMMDD-SP-CNATerminalAPI-Card
Short name	CNA-TC-API
Version	3.0.0-SNAPSHOT
Revision date	2026-07-20
Editor	Calypso Networks Association
Source repository	https://github.com/calypsonet/calypsonet-terminal-card-uml-api
Reference license	Creative Commons Attribution-NoDerivatives 4.0 International (CC BY-ND 4.0)

The present document specifies the 3.0.0-SNAPSHOT of the Terminal Card API. It defines a single, coherent baseline against which conforming implementations are evaluated; the **Since** column indicates the version in which each member was introduced.

Revision List

This section lists the high-level changes per version. The complete, fine-grained changelog is maintained in the `CHANGELOG.md` file at the root of the source repository.

Version / Date	Modifications
v3.0.0-SNAPSHOT 2026-07-20	Baseline.

2. Introduction

2.1. Purpose

The Terminal Card API defines the **internal** contract by which:

- a **reader implementation** MUST expose an APDU transmission interface (the **proxy reader**) so that selection managers and card transaction managers can exchange data with cards in a uniform way,
- a **card extension** MUST expose its selection process (request preparation and response parsing) so that the selection manager defined by CNA-TR-API can drive it,
- APDU-level exceptions carry the responses already received before the failure together with any duration information required to enforce the APDU exchange execution-time bounds.

The API is **internal**: terminal applications MUST NOT use it directly. It is the contract that allows reader implementations and card extensions written by independent vendors to interoperate.

2.2. Scope

This specification covers:

- the APDU request and response data carriers, including the per-request duration bound and per-response measured duration used to control the APDU exchange execution time,
- the selection request and response data carriers, including the multi-channel-aware channel information,
- the proxy-reader contract through which APDUs are transmitted, including the multi-channel transmission and closure operations,
- the SPI implemented by card extensions to drive selection,
- the SPI implemented by card extensions to represent a selected smart card, including the multi-channel variant,
- the family of exceptions raised at APDU exchange level.

The following topics are **out of scope**:

- the public reader interface exposed to terminal applications (covered by CNA-TR-API),
- the parsing of APDU payloads (deferred to specific card extension specifications),
- the protocol-level cryptographic security mechanisms handled by card extensions such as the Calypso Card API.

2.3. Conformance

A software product conforms to this specification if and only if:

1. it implements every interface and class defined in Chapter 5 with the operations and semantics prescribed in this document,
2. it raises exceptions exclusively of the types defined in Section 5.4 for the situations described,
3. it preserves the structural relationships described in Chapter 4.

Throughout this specification, the key words **MUST**, **MUST NOT**, **SHOULD**, **SHOULD NOT**, **RECOMMENDED**, **MAY** and **OPTIONAL** are to be interpreted as described in RFC 2119 and RFC 8174 when, and only when, they appear in all capitals.

In conformance with RFC 2119, **SHALL** is equivalent to **MUST**; this specification uses **MUST** exclusively in order to avoid ambiguity.

2.4. Document Conventions

2.4.1. Naming

Type names follow `UpperCamelCase`; operation, parameter and enumeration-member names follow `lowerCamelCase` except for enumeration values which use `UPPER_SNAKE_CASE`. Names defined by this specification are reproduced as-is in the UML class diagram (Section 4.5).

2.4.2. Language-agnostic types

Symbolic type	Description	Typical bindings
`string`	Sequence of characters, UTF-8 encoded by default.	`String`, `string`, `str`.
`boolean`	Two-state truth value.	`boolean`, `bool`.
`integer`	Signed 32-bit integer.	`int`, `Int32`.
`long`	Signed 64-bit integer.	`long`, `Int64`.
`byte array`	Ordered sequence of octets.	`byte[]`, `[u8]`, `bytes`.
`list<T>`	Ordered collection of `T` values, may contain duplicates.	`List<T>`, `Vec<T>`.
`set<T>`	Unordered collection of unique `T` values.	`Set<T>`, `HashSet<T>`.
`T?`	Nullable value: either a `T` value or the absence of value (`null`).	`Integer` (boxed), `Optional<T>`, `T?`.
`void`	Indicates that an operation returns no value.	`void`, `Unit`, `None`.

2.4.3. Type tables

Each interface, class, enumeration and exception defined in this document is described by a table of the form:

Name	The type's normalized name.
Kind	The category of the type (interface, class, enumeration, exception, etc.).
Namespace	The namespace in which the type is defined.
Extends	The type extended by this type, when applicable.
Implements	The interface implemented by this type, when applicable.
Since	The version of the API in which the type was introduced.
Purpose	A normative description of the type's responsibility.
See also	Cross-references to related operations or types, each a hyperlink to the corresponding table. Present only when applicable.

The **Purpose** row states, in one or two sentences, the responsibility of the type and the way an instance is obtained. It is meant to be scanned, not read: any content that requires structure or depth — rules, lists, notes, value tables or examples — is placed as prose below the table.

2.4.4. Operation tables

Each operation defined in this document is described by a table of the form:

Signature	The operation's language-agnostic signature.
Since	The version of the API in which the operation was introduced.
Description	A normative description of the operation's behaviour.
Parameters	A description of each input parameter.
Returns	A description of the returned value, if any.
Throws	A list of exceptional conditions, each mapped to a specific exception type.
See also	Cross-references to related operations or types, each a hyperlink to the corresponding table. Present only when applicable.

2.4.5. Operation contracts

To avoid repetition, the following rules apply to every operation table in Chapter 5 and are therefore not restated for each operation:

- **Non-`null` arguments.** Unless explicitly stated otherwise, every input parameter MUST be non-`null`. An implementation MUST raise `IllegalArgumentException` if a `null` value is supplied. This implicit `null` check is not repeated in the **Throws** row, which states `None.` when no other exception can occur.
- **Non-`null` results.** Unless the return type is marked nullable (`T?`) or the **Returns** row states otherwise, an operation returns a non-`null` value.
- **Collections.** Unless the return type is marked nullable (`T?`), an operation whose return type is a collection (e.g. `list`, `set`, `map`) returns an empty collection rather than `null`.

2.4.6. Concurrency

Unless explicitly stated otherwise, the stateful types defined by this specification are **not** thread-safe. A given instance MUST be accessed from a single thread at a time; concurrent use of the same instance without external synchronisation results in undefined behaviour. Distinct instances MAY be used concurrently.

2.5. References and Resources

2.5.1. Calypso References

References to CNA Terminal APIs designate the indicated **major** version; any later release within the same major is backward-compatible per that API's evolution policy.

Reference	Document
[CNA-TR-API]	*CNA Terminal API — Reader* (SP-CNATerminalAPI-Reader), version 3.0, Calypso Networks Association.

2.5.2. Normative References

Reference	Document
[RFC 2119]	*Key words for use in RFCs to Indicate Requirement Levels*, S. Bradner, March 1997.
[RFC 8174]	*Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words*, B. Leiba, May 2017.
[ISO/IEC 7816]	*Identification cards — Integrated circuit cards*.
[ISO/IEC 14443]	*Identification cards — Contactless integrated circuit cards — Proximity cards*.
[PC/SC]	*Interoperability Specification for ICCs and Personal Computer Systems*.

2.5.3. External Resources

Resource	Link
Calypso Networks Association	https://calypsonet.org
Terminal APIs website	https://terminal-api.calypsonet.org
Terminal APIs documentation	https://docs.terminal-api.calypsonet.org

3. Glossary and Acronyms

3.1. Glossary

Term	Definition
APDU	Application Protocol Data Unit, the message format used between the terminal and the card (ISO/IEC 7816-4).
Case 4 APDU	APDU command that carries data in both directions; the `Le` field is mandatory and MUST be set to `00h`.
Card Extension	Software module providing the card-specific dialog with a card family (e.g. Calypso).
Logical Channel	Software-level communication path established with a card after a successful selection.
Basic Channel	The default logical channel (channel `0`) of an ISO/IEC 7816-4 card.
Physical Channel	Hardware-level communication path between the reader and the card.
Proxy Reader	Reader-side facet exposing APDU transmission to extension layers, by opposition to the public reader interface presented to terminal applications.
Status Word (SW)	Two-byte trailer of an APDU response indicating the outcome of the command.
SPI	Service Provider Interface — a set of interfaces designed to be implemented by an extension module.

3.2. Acronyms

Abbreviation	Expansion
APDU	Application Protocol Data Unit
ATR	Answer To Reset
CNA	Calypso Networks Association
FCI	File Control Information
ISO	International Organization for Standardization
SPI	Service Provider Interface
SW	Status Word
UML	Unified Modelling Language

4. Architectural Overview

4.1. Functional positioning

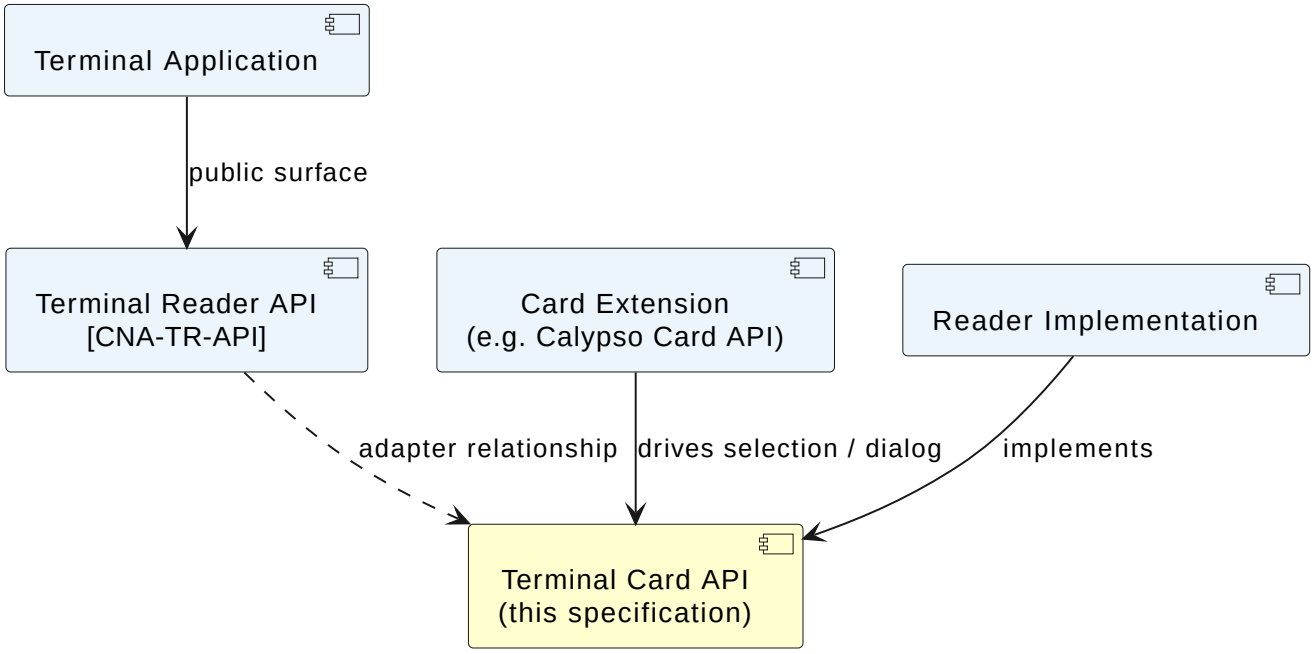

Figure 1. Terminal Card API — Architecture Overview

The **adapter relationship** depicted above (`TR ..> TC`) binds this specification to CNA-TR-API: an implementation exposes each contract of this API through an adapter that also implements the corresponding public-facing interface of the **Terminal Reader API**. Specifically:

- an adapter of `ProxyReaderApi` MUST also implement the `CardReader` interface of CNA-TR-API;
- an adapter of `CardSelectionExtensionSpi` MUST also implement the `CardSelectionExtension` interface of CNA-TR-API;
- an adapter of `SmartCardSpi` MUST also implement the `SmartCard` interface of CNA-TR-API.

4.2. Logical namespaces

Namespace	Role	Summary
`card`	Public API (internal-only audience)	Properties, parse exception, data carriers (APDU response, card response, selection response), proxy-reader interface and exceptions.
`card.spi`	SPI	Data carriers and contracts implemented by card extensions.

4.3. Multi-channel design

The Terminal Card API exposes the multi-channel capability of ISO/IEC 7816-4 cards through:

- a dedicated SPI sub-type `MultichannelSmartCardSpi` that carries the channel number of each active smart card,
- dedicated proxy-reader operations `transmitCardRequestAndCloseChannel` and `closeChannel` that target a specific channel by passing the corresponding `MultichannelSmartCardSpi`.

The single-channel transmission operation `transmitCardRequest` accepts the base `SmartCardSpi` and serves both the single-channel and the multi-channel use case (the implementation extracts the channel information from the dynamic sub-type when needed).

4.4. APDU exchange execution-time control

To enable software-level control of the APDU exchange execution time, this specification carries:

- an **optional** per-request duration bound on `ApduRequestSpi` (`getApduExchangeMaxDuration`),
- an **optional** per-response effective duration on `ApduResponseApi` (`getApduExchangeDuration`),
- a dedicated exception `ApduExchangeDurationExceededException` raised when the effective duration exceeds the declared bound.

4.5. UML class diagram

The following UML class diagram provides a visual representation of the types and relationships defined by this specification:

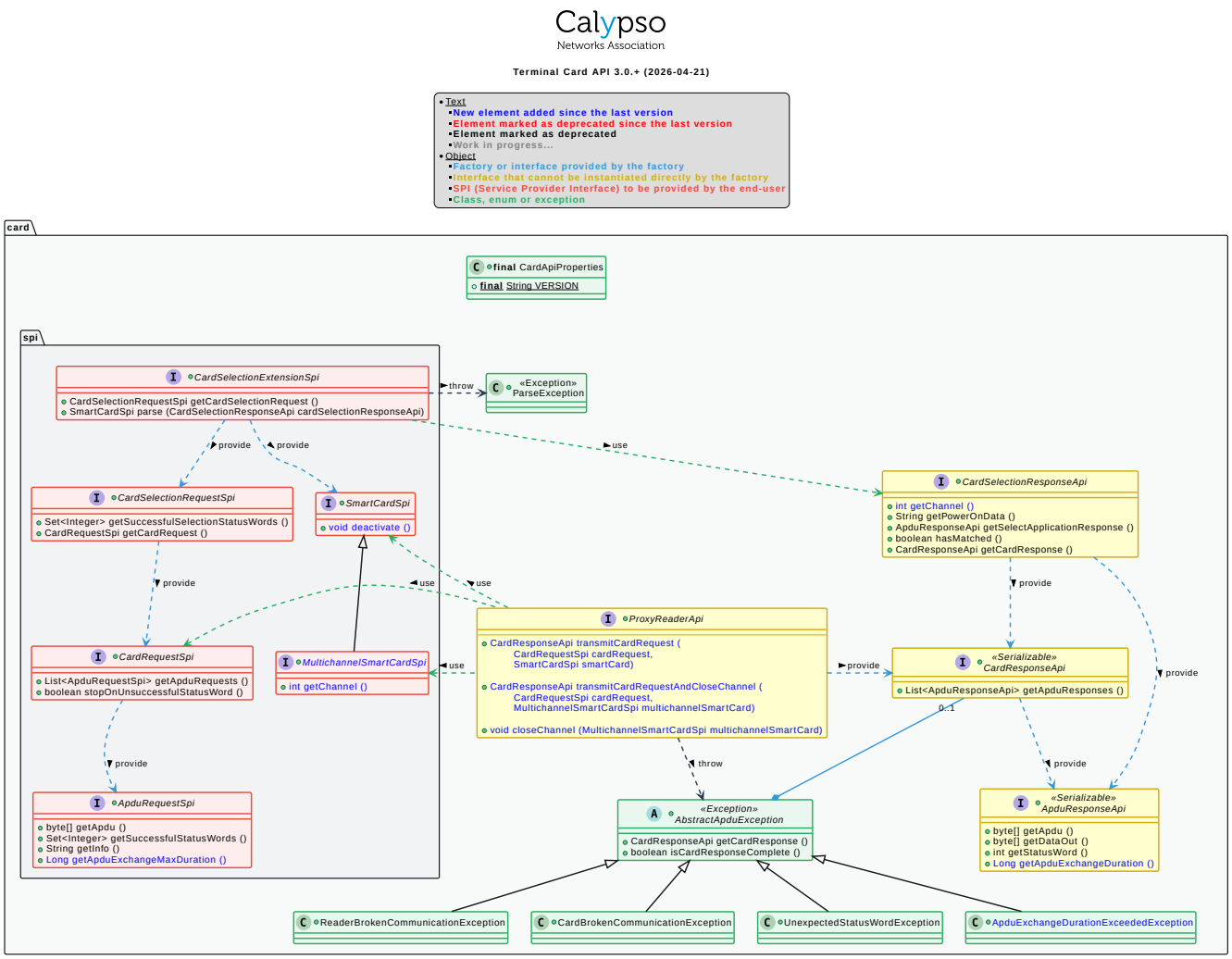

Figure 2. Terminal Card API — UML Class Diagram

5. API Specification

5.1. Classes

5.1.1. Card API Properties

Name	`CardApiProperties`
Kind	Final class
Namespace	`card`
Since	1.0
Purpose	Exposes the immutable properties of the Terminal Card API.

Constants

Name	Type	Since	Description
`VERSION`	`string`	1.0	String representation of the version of the API implemented by the conforming binding (e.g. `"3.0"`).

The `CardApiProperties` type cannot be instantiated; it exposes only static properties.

5.2. API Interfaces

5.2.1. APDU Response API

Name	`ApduResponseApi`
Kind	Interface (Serializable)
Namespace	`card`
Since	1.0
Purpose	Carries the data received in response to a single APDU command (variable-length payload followed by the status word).
See also	`ApduRequestSpi`

Instances are serialisable so that they can be transported across distributed boundaries.

Get APDU

Signature	`getApdu() → byte array`
Since	1.0
Description	Returns the raw response bytes received from the card, **including** the status word.
Returns	A `byte array` of at least 2 bytes.
Throws	None.

Get APDU Exchange Duration

Signature	`getApduExchangeDuration() → long?`

Since	3.0
Description	Returns the effective duration of the APDU exchange, in **milliseconds**, as measured by the underlying reader implementation.
Returns	The measured duration in milliseconds, or `null` if the reader does not provide the measurement.
Throws	None.

Get Data Out

Signature	`getDataOut() → byte array`
Since	1.0
Description	Returns the data part of the response (excluding the status word).
Returns	A `byte array`; empty if the response carries no data.
Throws	None.

Get Status Word

Signature	`getStatusWord() → integer`
Since	1.0
Description	Returns the status word as an `integer` between `0000h` and `FFFFh`.
Returns	An `integer` in the range `[0x0000, 0xFFFF]`.
Throws	None.

5.2.2. Card Response API

Name	`CardResponseApi`
Kind	Interface (Serializable)
Namespace	`card`
Since	1.0
Purpose	Groups multiple APDU responses received from the card after the execution of a `CardRequestSpi`.
See also	`CardRequestSpi`

A `CardResponseApi` MAY be embedded in an `AbstractApduException`; in that case some responses MAY be missing (e.g. the card was removed during processing, or processing stopped after an unsuccessful status word—see `CardRequestSpi.stopOnUnsuccessfulStatusWord`).

Get APDU Responses

Signature	`getApduResponses() → list<ApduResponseApi>`
Since	1.0
Description	Returns the responses received for the executed APDU requests, in the same order in which the requests were submitted.
Returns	A `list`; empty if there is no response.
Throws	None.

5.2.3. Card Selection Response API

Name	`CardSelectionResponseApi`
Kind	Interface
Namespace	`card`
Since	1.0
Purpose	Carries the data observed during the start-up phase with the card—the selection step itself, the channel on which the card has been placed and any additional commands that may have been executed afterwards.
See also	`CardSelectionRequestSpi`

Get Card Response

Signature	`getCardResponse() → CardResponseApi?`
Since	1.0
Description	Returns the responses received for any additional commands attached to the selection case.
Returns	The grouped response (`CardResponseApi`), or `null` if no additional command was executed.
Throws	None.

Get Channel

Signature	`getChannel() → integer`
Since	3.0
Description	Returns the logical channel number on which the card has been placed by the selection. For a single-channel selection scenario this is `0`. For a multi-channel selection scenario, the channel may be any value supported by the underlying ISO/IEC 7816-4 card.
Returns	A non-negative `integer`.
Throws	None.

Get Power On Data

Signature	`getPowerOnData() → string?`
Since	1.0
Description	Returns the card's power-on data, that is, the data retrieved by the reader when the card is inserted. For a contact reader, this is the Answer To Reset (ATR) defined by ISO/IEC 7816. For a contactless reader, the reader decides what this data is. Some contactless readers provide a virtual ATR (partially standardised by the PC/SC standard), but other devices MAY have their own definition, including for example elements from the anti-collision stage of the ISO/IEC 14443 protocol (ATQA, ATQB, ATS, SAK, etc.) or any proprietary definition. As this data varies from one reader to another, it is exposed as a `string`, which MAY be a hexadecimal string or any other relevant representation.
Returns	The non-empty power-on data, or `null` if no power-on data is available.
Throws	None.

Get Select Application Response

Signature	`getSelectApplicationResponse() → ApduResponseApi?`

Since	1.0
Description	Returns the response received from the card to the **Select Application** command.
Returns	The response (`ApduResponseApi`), or `null` if no **Select Application** command was executed.
Throws	None.

Has Matched

Signature	`hasMatched() → boolean`
Since	1.0
Description	Indicates whether the inserted card matches the selection filters.
Returns	`true` if the card matched, `false` otherwise.
Throws	None.

5.2.4. Proxy Reader API

Name	`ProxyReaderApi`
Kind	Interface
Namespace	`card`
Since	1.0
Purpose	Reader-side facet exposing APDU transmission and channel-closure operations to card extensions.

A `ProxyReaderApi` and the `CardReader` interface defined in CNA-TR-API are two facets of the same underlying reader; they are separated so that only the **internal** dialog with cards is exposed to extension modules. `ProxyReaderApi` is the **backside** of `CardReader`: it is the reader able to transmit card requests and having control over the physical channel. An adapter of this interface MUST therefore also implement `CardReader`.

The proxy reader uses the **smart card SPI** received as parameter to identify the target of every operation: it extracts the channel information (when the dynamic type is `MultichannelSmartCardSpi`) and verifies the active state of the smart card (see the role of the smart-card SPI parameter) before performing any exchange.

Role of the smart-card SPI parameter

The three operations of `ProxyReaderApi` receive a smart-card SPI parameter. This parameter is **not** a mere vehicle for the channel number. It plays up to three roles:

1. **Channel routing** (only when the dynamic type is `MultichannelSmartCardSpi`): the logical channel number is read via `getChannel()` and used to route the APDU exchange or the closure on the correct logical channel. When the parameter is a base `SmartCardSpi` (not a `MultichannelSmartCardSpi`), the exchange is addressed to the basic channel (channel `0`).
2. **Active-state check**: the active state of the smart card is verified before any exchange (`SmartCard.isActive()` is `true` per CNA-TR-API). If it is not active, a `CardBrokenCommunicationException` is raised without contacting the card.
3. **Deactivation**: when the situation requires it (typically after a communication failure or after an explicit channel closure), `SmartCardSpi.deactivate` is called so that the deactivation is immediately visible to the application via `SmartCard.isActive() == false`.

Roles 2 and 3 are common to `SmartCardSpi` and `MultichannelSmartCardSpi`; role 1 is specific to the multi-channel variant.

Close Channel

Signature	`closeChannel(multichannelSmartCard: MultichannelSmartCardSpi) → void`
Since	3.0
Description	Closes the logical channel of the smart card identified by `multichannelSmartCard` and deactivates the smart card. This operation is idempotent: calling it on an already-closed channel has no effect.
Parameters	`multichannelSmartCard` (`MultichannelSmartCardSpi`) — the multi-channel smart card SPI identifying the target.
Throws	`ReaderBrokenCommunicationException` — if communication with the reader is broken. `CardBrokenCommunicationException` — if communication with the card is broken.

Transmit Card Request

Signature	`transmitCardRequest(`
	`  cardRequest: CardRequestSpi,`
	`  smartCard: SmartCardSpi`
	`) → CardResponseApi`

Since	3.0
Description	Transmits the provided `CardRequestSpi` targeting the smart card identified by `smartCard`. The logical channel remains open after the operation. If the smart card is no longer active at the moment of the call, a `CardBrokenCommunicationException` is raised without any exchange. **Note:** if an error occurs while sending an APDU (communication error or unexpected status word), an `AbstractApduException` is thrown and the responses already collected from the previously transmitted APDUs are attached to it. This lets the application tolerate card tearing and retrieve the partial response, or keep strict control over the APDUs sent to the card (see `CardRequestSpi.stopOnUnsuccessfulStatusWord`).
Parameters	`cardRequest` (`CardRequestSpi`) — the request to transmit. `smartCard` (`SmartCardSpi`) — the smart card SPI identifying the target.
Returns	A `CardResponseApi`.
Throws	`ReaderBrokenCommunicationException` — if communication with the reader is broken. `CardBrokenCommunicationException` — if communication with the card is broken, or if the smart card is no longer active. `UnexpectedStatusWordException` — if a command returned an unexpected status word. `ApduExchangeDurationExceededException` — if a measured APDU duration exceeded the bound declared on the request.

Transmit Card Request And Close Channel

Signature	`transmitCardRequestAndCloseChannel(`
	`  cardRequest: CardRequestSpi,`
	`  multichannelSmartCard: MultichannelSmartCardSpi`
	`) → CardResponseApi`

Since	3.0
Description	Transmits the provided request targeting the smart card identified by `multichannelSmartCard` and, upon successful completion, closes the corresponding logical channel. The smart card is deactivated after the channel has been closed (see the role of the smart-card SPI parameter).
Parameters	`cardRequest` (`CardRequestSpi`) — the request to transmit. `multichannelSmartCard` (`MultichannelSmartCardSpi`) — the multi-channel smart card SPI identifying the target.
Returns	A `CardResponseApi`.

Throws	`ReaderBrokenCommunicationException` — if communication with the reader is broken. `CardBrokenCommunicationException` — if communication with the card is broken, or if the smart card is no longer active. `UnexpectedStatusWordException` — if a command returned an unexpected status word. `ApduExchangeDurationExceededException` — if a measured APDU duration exceeded the bound declared on the request.

5.3. SPI Interfaces

5.3.1. APDU Request SPI

Name	`ApduRequestSpi`
Kind	Interface (SPI)
Namespace	`card.spi`
Since	1.0
Purpose	Carries a single APDU request together with its successful status words, its information string and an optional duration bound used to control the APDU exchange execution time.
See also	`ApduResponseApi`

Get APDU

Signature	`getApdu() → byte array`
Since	1.0
Description	Returns the bytes of the APDU command (header and, when applicable, data field and `Le`). For case 4 APDU commands, the `Le` field is mandatory and MUST be set to `00h`.
Returns	A `byte array` of at least 4 bytes.
Throws	None.

Get APDU Exchange Max Duration

Signature	`getApduExchangeMaxDuration() → long?`
Since	3.0
Description	Returns the maximum tolerated duration of the APDU exchange, in **milliseconds**. The proxy reader MUST measure the effective duration of the exchange and raise `ApduExchangeDurationExceededException` if it exceeds this bound.
Returns	The maximum duration in milliseconds, or `null` if no bound is defined for this request.
Throws	None.

Get Info

Signature	`getInfo() → string?`
Since	1.0
Description	Returns an informational string describing the APDU, used by implementations for logging and error messages.
Returns	The non-empty informational `string`, or `null` if no information has been defined.
Throws	None.

Get Successful Status Words

Signature	`getSuccessfulStatusWords() → set<integer>`
Since	1.0
Description	Returns the set of status words that MUST be considered successful for this APDU.
Returns	A non-empty `set` of status words, containing at least `9000h`.
Throws	None.

5.3.2. Card Request SPI

Name	`CardRequestSpi`
Kind	Interface (SPI)
Namespace	`card.spi`
Since	1.0
Purpose	Aggregates an ordered list of `ApduRequestSpi` to be executed consecutively against the card, together with a flag indicating whether processing stops when one of the APDUs answers with an unexpected status word.
See also	`ApduResponseApi`

Get APDU Requests

Signature	`getApduRequests() → list<ApduRequestSpi>`
Since	1.0
Description	Returns the ordered list of APDU requests to be executed.
Returns	A non-empty `list`.
Throws	None.

Stop On Unsuccessful Status Word

Signature	`stopOnUnsuccessfulStatusWord() → boolean`
Since	1.0
Description	Indicates whether the iteration MUST stop at the first APDU whose response status word is not in its `getSuccessfulStatusWords()` set.
Returns	`true` to stop on the first failure, `false` to continue regardless.
Throws	None.

5.3.3. Card Selection Extension SPI

Name	`CardSelectionExtensionSpi`
Kind	Interface (SPI)
Namespace	`card.spi`
Since	2.0
Purpose	Drives a selection case from the card-extension side: builds the selection request and parses the resulting response into a smart card.

An implementation of `CardSelectionExtensionSpi` MUST also implement the `CardSelectionExtension` interface defined in CNA-TR-API.

Get Card Selection Request

Signature	`getCardSelectionRequest() → CardSelectionRequestSpi`
Since	1.0
Description	Returns the request to use when executing this selection case.
Returns	A `CardSelectionRequestSpi`.
Throws	None.

Parse

Signature	`parse(cardSelectionResponseApi: CardSelectionResponseApi) → SmartCardSpi`
Since	1.0
Description	Parses the response of an executed selection case to produce the corresponding smart-card SPI.
Parameters	`cardSelectionResponseApi` (`CardSelectionResponseApi`) — the card selection response.
Returns	A `SmartCardSpi`.
Throws	`ParseException` — if the response cannot be interpreted.

5.3.4. Card Selection Request SPI

Name	`CardSelectionRequestSpi`
Kind	Interface (SPI)
Namespace	`card.spi`
Since	1.0
Purpose	Carries the data needed to execute a single selection case: a card selector defining the target card profile, and an optional card request to be executed after a successful card selection.
See also	`CardSelectionResponseApi`

Get Card Request

Signature	`getCardRequest() → CardRequestSpi?`
Since	1.0
Description	Returns the optional follow-up card request to execute after a successful selection.
Returns	A `CardRequestSpi`, or `null` if no follow-up is needed.
Throws	None.

Get Successful Selection Status Words

Signature	`getSuccessfulSelectionStatusWords() → set<integer>`
Since	2.0
Description	Returns the set of status words that MUST be considered successful for this selection case.
Returns	A non-empty `set` of status words, containing at least `9000h`.
Throws	None.

5.3.5. Multichannel Smart Card SPI

Name	`MultichannelSmartCardSpi`
Kind	Interface (SPI)
Namespace	`card.spi`
Extends	`SmartCardSpi`
Since	3.0
Purpose	Multi-channel variant of the smart-card SPI, used by card extensions whose underlying card supports several active logical channels.

Get Channel

Signature	`getChannel() → integer`
Since	3.0
Description	Returns the logical channel number on which this smart card has been placed.
Returns	A non-negative `integer`.
Throws	None.

5.3.6. Smart Card SPI

Name	`SmartCardSpi`
Kind	Interface (SPI)
Namespace	`card.spi`
Since	1.0
Purpose	Card-extension facet of a successfully selected smart card. An implementation of this SPI MUST also implement the `SmartCard` interface defined in CNA-TR-API.

Deactivate

Signature	`deactivate() → void`
Since	3.0
Description	Marks this smart card as no longer active. After this operation has returned, the corresponding `SmartCard` exposed to the application via CNA-TR-API MUST report `isActive() == false`. MUST be idempotent: calling it twice MUST have the same observable effect as calling it once.
Throws	None.

5.4. Exceptions

5.4.1. Abstract APDU Exception

Name	`AbstractApduException`
Kind	Abstract checked exception
Namespace	`card`
Since	1.0

Purpose	Base class for every exception raised by `ProxyReaderApi.transmitCardRequest*` operations. Carries the response data received from the card until a communication failure occurs or an unexpected APDU status word is received, together with a completeness flag indicating whether every prepared APDU produced a response.

Get Card Response

Signature	`getCardResponse() → CardResponseApi?`
Since	1.0
Description	Returns the responses collected before the failure occurred. MAY be `null` if no response was collected.
Returns	A `CardResponseApi`, or `null`.
Throws	None.

Is Card Response Complete

Signature	`isCardResponseComplete() → boolean`
Since	1.0
Description	Indicates whether the embedded card response is complete (every prepared APDU produced a response) or partial.
Returns	`true` if complete, `false` if partial.
Throws	None.

5.4.2. APDU Exchange Duration Exceeded Exception

Name	`ApduExchangeDurationExceededException`
Kind	Checked exception
Namespace	`card`
Extends	`AbstractApduException`
Since	3.0
Purpose	Indicates that the effective duration of an APDU exchange exceeded the bound declared on the request via `getApduExchangeMaxDuration`. Higher-level extensions (e.g. Calypso) MAY intercept this exception, cancel any ongoing session and propagate the failure to the application as an `InvalidCardResponseException` (CNA-TR-API).

5.4.3. Card Broken Communication Exception

Name	`CardBrokenCommunicationException`
Kind	Checked exception
Namespace	`card`
Extends	`AbstractApduException`
Since	1.0
Purpose	Indicates that the communication with the card has failed during the execution of a card request, or that the targeted smart card is no longer active. Carries the response data received from the card until the failure occurred.

5.4.4. Parse Exception

Name	`ParseException`
Kind	Checked exception
Namespace	`card`
Since	2.0
Purpose	Raised by `CardSelectionExtensionSpi.parse` when the parsing of the card selection response has failed. The most likely reason is that the **Select Application** command returned an invalid FCI structure.

5.4.5. Reader Broken Communication Exception

Name	`ReaderBrokenCommunicationException`
Kind	Checked exception
Namespace	`card`
Extends	`AbstractApduException`
Since	1.0
Purpose	Indicates that the communication with the reader has failed during the execution of a card request. Carries the response data received from the card until the failure occurred.

5.4.6. Unexpected Status Word Exception

Name	`UnexpectedStatusWordException`
Kind	Checked exception
Namespace	`card`
Extends	`AbstractApduException`
Since	1.0
Purpose	Indicates that a command returned a status word that was not part of its `getSuccessfulStatusWords()` set. Carries the response data received from the card until that status word was received.