SPECIFICATION

Calypso Terminal API

Storage Card

Version 2.0.0-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 Storage 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 Storage Card API MUST satisfy.

The Terminal Storage Card API extends the **Terminal Reader API** with the abstractions required to manipulate **storage cards** (MIFARE Ultralight, MIFARE Classic 1K/4K, ST25/SRT512), all contactless cards compliant with ISO/IEC 14443. It exposes a block-oriented memory image, an authentication mechanism for MIFARE Classic and a transaction manager that batches read and write commands.

Document Status

Reference	YYMMDD-SP-CNATerminalAPI-StorageCard
Short name	CNA-TSC-API
Version	2.0.0-SNAPSHOT
Revision date	2026-07-20
Editor	Calypso Networks Association
Source repository	https://github.com/calypsonet/calypsonet-terminal-storagecard-uml-api
Reference license	Creative Commons Attribution-NoDerivatives 4.0 International (CC BY-ND 4.0)

The present document specifies the 2.0.0-SNAPSHOT of the Terminal Storage 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
v2.0.0-SNAPSHOT 2026-07-20	Baseline.

2. Introduction

2.1. Purpose

The Terminal Storage Card API defines the contracts required to:

- select storage cards through the **Terminal Reader API** selection mechanism,
- perform block-level read and write operations,
- expose card-specific metadata (product type, UID, system block),
- authenticate against MIFARE Classic sectors using Key A or Key B.

Conforming implementations MUST guarantee interoperability with CNA-TR-API reader implementations.

2.2. Scope

This specification covers:

- the factory used to instantiate the public types of the API,
- the abstract representation of a storage card and its product types,
- the card-selection extension used to enrich a selection scenario with storage-card commands,
- the transaction manager used to exchange APDUs with the selected card,
- the family of storage-card-specific exceptions.

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

- the wire-level protocols specific to each storage card technology,
- the management of cryptographic key material for MIFARE Classic authentication.

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 4.3).

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`.
`byte array`	Ordered sequence of octets.	`byte[]`, `[u8]`, `bytes`.
`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.
- **Fluent composition.** Builder-style operations return the current instance so that calls can be chained.

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 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
Block	Smallest addressable storage unit on a storage card. The block size depends on the card product type.
Sector	Group of blocks sharing the same access conditions on a MIFARE Classic card.
System Block	Block containing card-specific metadata and configuration data. Available on selected card types only.
Storage Card	Card whose primary purpose is to store binary data in addressable blocks, by opposition to a Calypso or generic application card.
MIFARE Classic	NXP MIFARE Classic 1K/4K family of contactless storage cards using sector-based authentication.
UID	Unique IDentifier of the card.
OTP	One-Time-Programmable memory area whose bits can only transition from `0` to `1`.

3.2. Acronyms

Abbreviation	Expansion
APDU	Application Protocol Data Unit
CNA	Calypso Networks Association
OTP	One-Time-Programmable
PC/SC	Personal Computer / Smart Card (interoperability standard)
SPI	Service Provider Interface
UID	Unique IDentifier
UML	Unified Modelling Language

4. Architectural Overview

4.1. Functional positioning

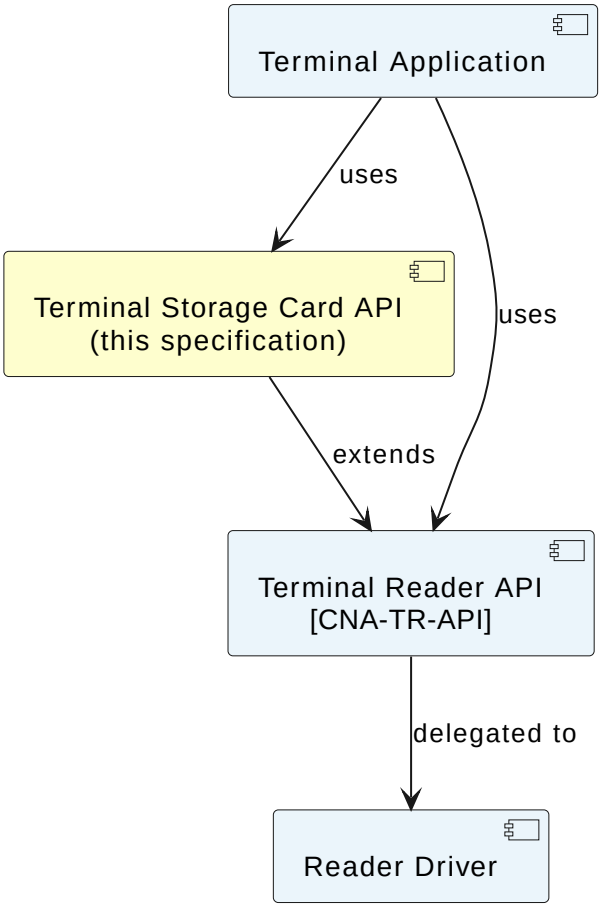

Figure 1. Terminal Storage Card API — Architecture Overview

4.2. Logical namespaces

Namespace	Role	Summary
`storagecard`	Public API	Factory, properties, key types and exceptions.
`storagecard.card`	Public API	Storage card abstraction, product type enumeration and selection extension.
`storagecard.transaction`	Public API	Transaction manager for storage cards.

4.3. UML class diagram

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

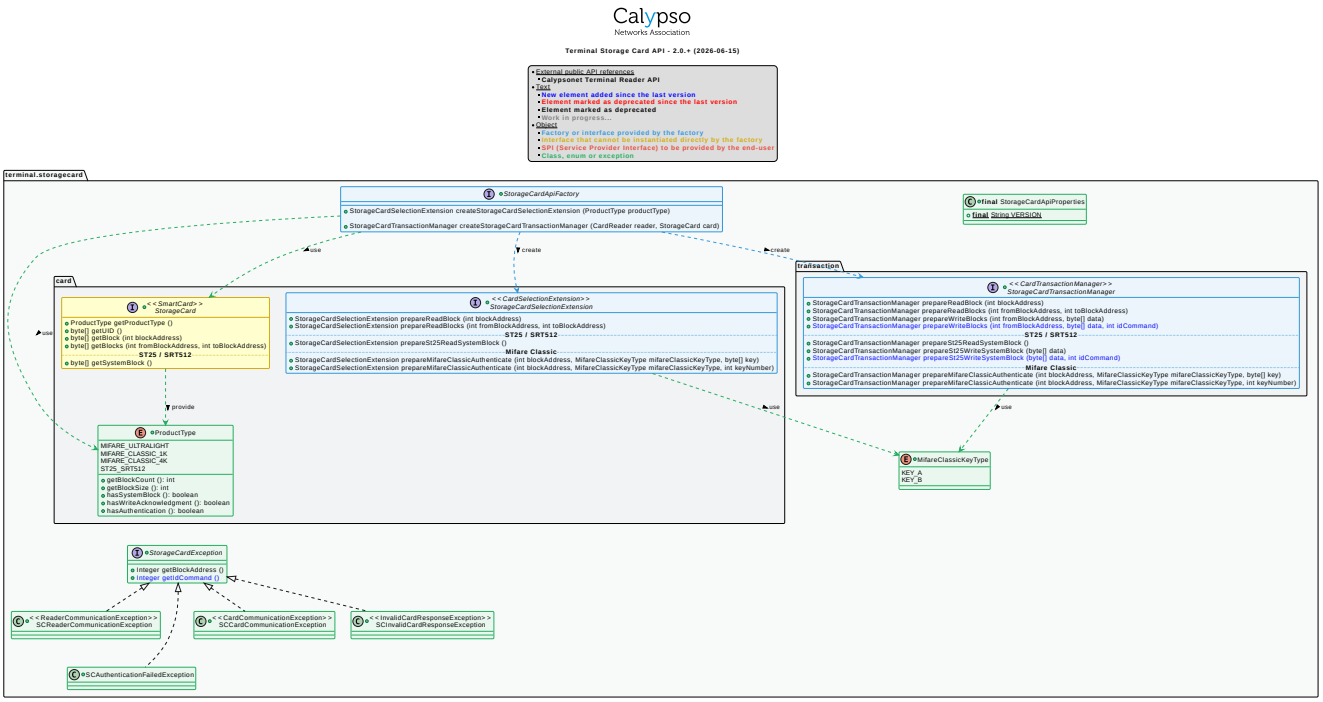

Figure 2. Terminal Storage Card API — UML Class Diagram

5. API Specification

5.1. Classes

5.1.1. Storage Card API Properties

Name	`StorageCardApiProperties`
Kind	Final class
Namespace	`storagecard`
Since	1.0
Purpose	Exposes the immutable properties of the Terminal Storage 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. `"2.0"`).

5.2. API Interfaces

5.2.1. Storage Card

Name	`StorageCard`
Kind	Interface
Namespace	`storagecard.card`
Extends	`SmartCard` (CNA-TR-API)
Since	1.0
Purpose	Represents a storage card and exposes operations to retrieve its memory image and metadata.

Get Block

Signature	`getBlock(blockAddress: integer) → byte array`
Since	1.0
Description	Returns the data block at the specified block address. If the block has not been previously read, the returned array is filled with zeros. A caller cannot distinguish an unread block (returned as zeros) from a block whose stored content is genuinely all-zero.
Parameters	`blockAddress` — the address of the block to retrieve.
Returns	A non-empty `byte array` containing the block data; zero-filled if the block has not been read.
Throws	`IndexOutOfBoundsException` — if the block address is out of range.

Get Blocks

Signature	`getBlocks(`
	`  fromBlockAddress: integer,`
	`  toBlockAddress: integer`
	`) → byte array`

Since	1.0
Description	Returns the data blocks within the specified range from the memory image of the storage card. The returned array contains the blocks in order, from `fromBlockAddress` to `toBlockAddress` (both inclusive). Blocks that have not been previously read appear as zero-filled sections. A caller cannot distinguish an unread block from a block whose stored content is genuinely all-zero.
Parameters	`fromBlockAddress` — the starting block address (inclusive). `toBlockAddress` — the ending block address (inclusive).
Returns	A non-empty `byte array` containing the data blocks in the specified range.
Throws	`IndexOutOfBoundsException` — if `fromBlockAddress` is greater than `toBlockAddress`, if either is negative, or if they exceed the memory image range.

Get Product Type

Signature	`getProductType() → ProductType`
Since	1.0
Description	Returns the product type of the storage card.
Returns	A `ProductType`.
Throws	None.

Get System Block

Signature	`getSystemBlock() → byte array?`
Since	1.0
Description	Returns the system block when it has been read. The system block contains card-specific metadata and configuration data. This feature is specific to ST25/SRT512 cards. The system block MUST have been previously read using `prepareSt25ReadSystemBlock()`.
Returns	A non-empty `byte array` containing the system block data, or `null` if the system block has not been read yet.
Throws	`UnsupportedOperationException` — if the current card type does not support system block access.

Get UID

Signature	`getUID() → byte array`
Since	1.0
Description	Returns the unique identifier (UID) of the storage card.
Returns	A non-empty `byte array` containing the UID.
Throws	None.

5.2.2. Storage Card API Factory

Name	`StorageCardApiFactory`
Kind	Interface
Namespace	`storagecard`

Since	1.0
Purpose	Factory used by the application to obtain instances of the public types provided by the API.

Create Storage Card Selection Extension

Signature	`createStorageCardSelectionExtension(`
	`  productType: ProductType`
	`) → StorageCardSelectionExtension`

Since	1.0
Description	Returns a new `StorageCardSelectionExtension` dedicated to the targeted product type.
Parameters	`productType` (`ProductType`) — the targeted storage card product type.
Returns	A `StorageCardSelectionExtension`.
Throws	None.

Create Storage Card Transaction Manager

Signature	`createStorageCardTransactionManager(`
	`  reader: CardReader,`
	`  card: StorageCard`
	`) → StorageCardTransactionManager`

Since	1.0
Description	Returns a new `StorageCardTransactionManager` bound to the supplied reader and to the initial card data provided by the selection process.
Parameters	`reader` (`CardReader` (CNA-TR-API)) — the reader through which the card is reached. `card` (`StorageCard`) — the initial card data provided by the selection process.
Returns	A `StorageCardTransactionManager`.
Throws	None.

5.2.3. Storage Card Selection Extension

Name	`StorageCardSelectionExtension`
Kind	Interface
Namespace	`storagecard.card`
Extends	`CardSelectionExtension` (CNA-TR-API)
Since	1.0
Purpose	Allows the application to enrich the selection scenario with optional read and authentication commands executed during the selection phase. An instance is obtained via `StorageCardApiFactory.createStorageCardSelectionExtension`.

Prepare MIFARE Classic Authenticate (Key Number)

Signature	`prepareMifareClassicAuthenticate(`
	`  blockAddress: integer,`
	`  mifareClassicKeyType: MifareClassicKeyType,`
	`  keyNumber: integer`
	`) → StorageCardSelectionExtension`

Since	1.1

Description	Prepares a MIFARE Classic authentication command using a key referenced by its storage index in the reader's key storage. This avoids transmitting the key value over the communication channel. This operation is specific to MIFARE Classic cards and MUST be prepared before reading from or writing to protected sectors. The authentication applies to the entire sector containing the supplied block address.
Parameters	`blockAddress` — the address of any block within the sector to authenticate. `mifareClassicKeyType` (`MifareClassicKeyType`) — the type of key to use (Key A or Key B). `keyNumber` — the index of the key in the reader's key storage.
Returns	The current instance (`StorageCardSelectionExtension`).
Throws	`IllegalArgumentException` — if the block address is out of range, or if the key number is invalid. `UnsupportedOperationException` — if the current card type does not support authentication.

Prepare MIFARE Classic Authenticate (Key)

Signature	`prepareMifareClassicAuthenticate(`
	`  blockAddress: integer,`
	`  mifareClassicKeyType: MifareClassicKeyType,`
	`  key: byte array`
	`) → StorageCardSelectionExtension`

Since	1.1
Description	Prepares a MIFARE Classic authentication command using the supplied 6-byte key. The authentication applies to the entire sector containing the specified block address. When the key value is supplied this way, it is forwarded to the reader to be stored as a **volatile** key at index 0 (see PC/SC **Load Key** command). The volatile key is erased after usage. **Security note:** this overload transmits the key value over the application-reader communication channel. Security-sensitive applications SHOULD use the overload accepting a key index instead.
Parameters	`blockAddress` — the address of any block within the sector to authenticate. `mifareClassicKeyType` (`MifareClassicKeyType`) — the type of key to use (Key A or Key B). `key` — the 6-byte key data for authentication.
Returns	The current instance (`StorageCardSelectionExtension`).
Throws	`IllegalArgumentException` — if the block address is out of range, or if the key is not exactly 6 bytes long. `UnsupportedOperationException` — if the current card type does not support authentication.

Prepare Read Block

Signature	`prepareReadBlock(blockAddress: integer) → StorageCardSelectionExtension`
Since	1.0
Description	Prepares the reading of a specific block. Block addresses start at `0`; the maximum value is `getBlockCount() - 1`. Once processed, the result is available in the `StorageCard`.
Parameters	`blockAddress` — the address of the block to be read.
Returns	The current instance (`StorageCardSelectionExtension`).
Throws	`IllegalArgumentException` — if the block address is out of range.
See also	`ProductType.getBlockCount`

Prepare Read Blocks

Signature	`prepareReadBlocks(`
	`  fromBlockAddress: integer,`
	`  toBlockAddress: integer`
	`) → StorageCardSelectionExtension`

Since	1.0
Description	Prepares the reading of a range of blocks. Block addresses start at `0`; the maximum value is `ProductType.getBlockCount` - 1. Once processed, the result is available in the `StorageCard` via `getBlock` / `getBlocks`.
Parameters	`fromBlockAddress` — the starting block address (inclusive). `toBlockAddress` — the ending block address (inclusive).
Returns	The current instance (`StorageCardSelectionExtension`).
Throws	`IllegalArgumentException` — if one of the arguments is out of range.
See also	`ProductType.getBlockCount`

Prepare ST25 Read System Block

Signature	`prepareSt25ReadSystemBlock() → StorageCardSelectionExtension`
Since	1.2
Description	Prepares the reading of the system block of an ST25/SRT512 storage card. This operation is specific to the ST25 and SRT512 card types, which expose a system block at address 255 carrying card-specific metadata and configuration data. Once processed, the result is available in the `StorageCard` via `getSystemBlock`.
Returns	The current instance (`StorageCardSelectionExtension`).
Throws	`UnsupportedOperationException` — if the current card type is not ST25/SRT512.
See also	`StorageCard.getSystemBlock`

5.2.4. Storage Card Transaction Manager

Name	`StorageCardTransactionManager`
Kind	Interface
Namespace	`storagecard.transaction`
Extends	`CardTransactionManager<StorageCardTransactionManager>` (CNA-TR-API)
Since	1.0
Purpose	Manages APDU exchanges with a storage card. It allows the application to prepare read and write operations, batch them into a single transaction, and control the underlying communication channel. An instance is obtained via `StorageCardApiFactory.createStorageCardTransactionManager`.

The inherited `processCommands()` operation sends every APDU corresponding to a prepared command, in the order in which it was prepared, and interrupts the process at the first failed command.

For read commands, it updates the `StorageCard` memory image with the data retrieved from the card;

for write commands, it updates the memory image with the data written **only** when the write is confirmed successful.

For card technologies that do not provide reliable status codes (e.g. ST25/SRT512), confirmation is obtained via an automatic verification read, and the memory image is updated only if this verification passes.

Prepare MIFARE Classic Authenticate (Key Number)

Signature	`prepareMifareClassicAuthenticate(`
	`  blockAddress: integer,`
	`  mifareClassicKeyType: MifareClassicKeyType,`
	`  keyNumber: integer`
	`) → StorageCardTransactionManager`

Since	1.1
Description	Prepares a MIFARE Classic authentication command using a key referenced by its storage index in the reader's key storage. This operation is specific to MIFARE Classic cards and MUST be prepared before reading from or writing to protected sectors. The authentication applies to the entire sector containing the specified block address. Referencing the key by index allows pre-configured keys to be used without transmitting their value over the communication channel, which provides enhanced security. Once authenticated, subsequent read and write operations within the same sector MAY be performed without re-authentication, until the card is removed from the field or another sector is accessed.
Parameters	`blockAddress` — the address of any block within the sector to authenticate. `mifareClassicKeyType` (`MifareClassicKeyType`) — the type of key to use (Key A or Key B). `keyNumber` — the index of the key in the reader's key storage.
Returns	The current instance (`StorageCardTransactionManager`).
Throws	`IllegalArgumentException` — if the block address is out of range, or if the key number is invalid. `UnsupportedOperationException` — if the current card type does not support authentication.

Prepare MIFARE Classic Authenticate (Key)

Signature	`prepareMifareClassicAuthenticate(`
	`  blockAddress: integer,`
	`  mifareClassicKeyType: MifareClassicKeyType,`
	`  key: byte array`
	`) → StorageCardTransactionManager`

Since	1.1
Description	Prepares a MIFARE Classic authentication command using the supplied 6-byte key. This operation is specific to MIFARE Classic cards and MUST be prepared before reading from or writing to protected sectors. The authentication applies to the entire sector containing the specified block address. Once authenticated, subsequent read and write operations within the same sector MAY be performed without re-authentication, until the card is removed from the field or another sector is accessed. When the key value is supplied this way, it is forwarded to the reader to be stored as a **volatile** key at index 0 (see PC/SC **Load Key** command). The volatile key is temporary and is erased after usage, when the reader is powered off. **Security note:** this overload transmits the key value over the application-reader communication channel. For production environments and security-sensitive applications, it is RECOMMENDED to use the `keyNumber` overload instead, which references a key already stored in the reader without transmitting its value.
Parameters	`blockAddress` — the address of any block within the sector to authenticate. `mifareClassicKeyType` (`MifareClassicKeyType`) — the type of key to use (Key A or Key B). `key` — the 6-byte key data for authentication.
Returns	The current instance (`StorageCardTransactionManager`).
Throws	`IllegalArgumentException` — if the block address is out of range, or if the key is not exactly 6 bytes long. `UnsupportedOperationException` — if the current card type does not support authentication.

Prepare Read Block

Signature	`prepareReadBlock(blockAddress: integer) → StorageCardTransactionManager`
Since	1.0
Description	Prepares the reading of a specific block. Block addresses start at `0`; the maximum value is `getBlockCount() - 1`. Once processed, the result is available in the `StorageCard`.
Parameters	`blockAddress` — the address of the block to be read.
Returns	The current instance (`StorageCardTransactionManager`).
Throws	`IllegalArgumentException` — if the block address is out of range.
See also	`ProductType.getBlockCount`

Prepare Read Blocks

Signature	`prepareReadBlocks(`
	`  fromBlockAddress: integer,`
	`  toBlockAddress: integer`
	`) → StorageCardTransactionManager`

Since	1.0
Description	Prepares the reading of a range of blocks. Block addresses start at `0`; the maximum value is `ProductType.getBlockCount` - 1. Once processed, the result is available in the `StorageCard`.
Parameters	`fromBlockAddress` — the starting block address (inclusive). `toBlockAddress` — the ending block address (inclusive).
Returns	The current instance (`StorageCardTransactionManager`).
Throws	`IllegalArgumentException` — if one of the arguments is out of range.
See also	`ProductType.getBlockCount`

Prepare ST25 Read System Block

Signature	`prepareSt25ReadSystemBlock() → StorageCardTransactionManager`
Since	1.1
Description	Prepares the reading of the system block of an ST25/SRT512 storage card. This operation is specific to the ST25 and SRT512 card types, which expose a system block at address 255 carrying card-specific metadata and configuration data. Once processed, the result is available in the `StorageCard` via `getSystemBlock`.
Returns	The current instance (`StorageCardTransactionManager`).
Throws	`UnsupportedOperationException` — if the current card type is not ST25/SRT512.
See also	`StorageCard.getSystemBlock`

Prepare ST25 Write System Block (ID Command)

Signature	`prepareSt25WriteSystemBlock(`
	`  data: byte array,`
	`  idCommand: integer`
	`) → StorageCardTransactionManager`

Since	2.0

Description	Prepares the writing of data to the system block of an ST25/SRT512 storage card, and attaches an application-supplied identifier to the prepared command. The data length MUST match the block size. The provided data MUST represent the **expected final state** of the system block. Because ST25/SRT512 cards do not provide reliable status codes, an automatic verification read is performed and the transaction fails if the physical state does not match the provided data. The identifier MAY later be obtained via `StorageCardException.getIdCommand` if this specific command causes an exception to be raised.
Parameters	`data` — the data to be written. `idCommand` — the application-supplied identifier of this command.
Returns	The current instance (`StorageCardTransactionManager`).
Throws	`IllegalArgumentException` — if the length of `data` does not match the block size. `UnsupportedOperationException` — if the current card type is not ST25/SRT512.

Prepare ST25 Write System Block (No ID Command)

Signature	`prepareSt25WriteSystemBlock(data: byte array) → StorageCardTransactionManager`
Since	1.1
Description	Prepares the writing of data to the system block of an ST25/SRT512 storage card. This operation is specific to the ST25 and SRT512 card types, which expose a system block at address 255 carrying card-specific metadata and configuration data. The data length MUST match the block size defined by the card's `ProductType`. **Important:** the application is responsible for ensuring that the write operations are coherent with the target card's technology (e.g. OTP bits). The provided data MUST represent the **expected final state** of the system block after the write. Because ST25/SRT512 cards do not provide reliable status codes, an automatic verification read is performed and the transaction fails if the physical state does not match the provided data.
Parameters	`data` — the data to be written to the system block (expected final state).
Returns	The current instance (`StorageCardTransactionManager`).
Throws	`IllegalArgumentException` — if the length of `data` does not match the block size. `UnsupportedOperationException` — if the current card type is not ST25/SRT512.
See also	`ProductType.getBlockSize`

Prepare Write Blocks (ID Command)

Signature	`prepareWriteBlocks(`
	`  fromBlockAddress: integer,`
	`  data: byte array,`
	`  idCommand: integer`
	`) → StorageCardTransactionManager`

Since	2.0
Description	Prepares the writing of one or more blocks starting from the specified block offset, and attaches an application-supplied identifier to the prepared command. The number of blocks written is determined by the length of `data` divided by the block size of the card. **Important:** the application is responsible for ensuring that the write operations are coherent with the target card's technology (e.g. OTP bits, counters). The provided data MUST represent the **expected final state** of the blocks after the write. For cards that do not provide reliable status codes, an automatic verification read is performed and the transaction fails if the physical state does not match the provided data. The identifier MAY later be obtained via `StorageCardException.getIdCommand` if this specific command causes an exception to be raised.

Parameters	`fromBlockAddress` — the offset at which the blocks are written. `data` — the data to be written. `idCommand` — the application-supplied identifier of this command.
Returns	The current instance (`StorageCardTransactionManager`).
Throws	`IllegalArgumentException` — if the length of `data` is not a multiple of the block size.

Prepare Write Blocks (No ID Command)

Signature	`prepareWriteBlocks(`
	`  fromBlockAddress: integer,`
	`  data: byte array`
	`) → StorageCardTransactionManager`

Since	1.0
Description	Prepares the writing of one or more blocks starting from the specified block offset. The number of blocks written is determined by the length of `data` divided by the block size of the card. **Important:** the application is responsible for ensuring that the write operations are coherent with the target card's technology (e.g. OTP bits, counters). The provided data MUST represent the **expected final state** of the blocks after the write. For cards that do not provide reliable status codes, an automatic verification read is performed and the transaction fails if the physical state does not match the provided data.
Parameters	`fromBlockAddress` — the offset at which the blocks are written. `data` — the data to be written (expected final state).
Returns	The current instance (`StorageCardTransactionManager`).
Throws	`IllegalArgumentException` — if the length of `data` is not a multiple of the block size.
See also	`ProductType.getBlockCount`

5.3. Enumerations

5.3.1. MIFARE Classic Key Type

Name	`MifareClassicKeyType`
Kind	Enumeration
Namespace	`storagecard`
Since	1.1
Purpose	Enumerates the MIFARE Classic key types used for authentication.

MIFARE Classic cards support two types of keys per sector. Each sector can be protected independently by these keys, allowing fine-grained access control.

Value	Since	Description
`KEY_A`	1.1	Primary key used for authentication to a MIFARE Classic sector. In most configurations, Key A has read/write permissions while Key B may have restricted permissions.
`KEY_B`	1.1	Secondary key used for authentication to a MIFARE Classic sector. Key B is often used for restricted operations or read-only access, depending on the sector's access conditions.

5.3.2. Product Type

Name	`ProductType`
Kind	Enumeration
Namespace	`storagecard.card`
Since	1.0
Purpose	Enumerates the storage card products supported by the API and exposes their structural properties.

Value	Since	Description
`MIFARE_ULTRALIGHT`	1.0	MIFARE Ultralight — 16 blocks of 4 bytes, no system block, with write acknowledgment, without authentication.
`MIFARE_CLASSIC_1K`	1.1	MIFARE Classic 1K — 64 blocks of 16 bytes, no system block, with write acknowledgment, with authentication.
`MIFARE_CLASSIC_4K`	1.1	MIFARE Classic 4K — 256 blocks of 16 bytes, no system block, with write acknowledgment, with authentication.
`ST25_SRT512`	1.0	ST25 SRT512 — 16 blocks of 4 bytes, with a system block, without write acknowledgment, without authentication.

Get Block Count

Signature	`getBlockCount() → integer`
Since	1.0
Description	Returns the number of blocks in the main memory area of the card. Additional **system** blocks MAY exist and are not included in this count.
Returns	The number of blocks.
Throws	None.

Get Block Size

Signature	`getBlockSize() → integer`
Since	1.0
Description	Returns the size of each block in bytes.
Returns	The block size.
Throws	None.

Has Authentication

Signature	`hasAuthentication() → boolean`
Since	1.1
Description	Indicates whether the product type requires authentication before read/write operations. When this operation returns `true`, the application MUST authenticate the targeted sector before issuing read or write commands.
Returns	`true` if the card requires authentication, `false` otherwise.
Throws	None.

Has System Block

Signature	`hasSystemBlock() → boolean`
Since	1.0
Description	Indicates whether the product type exposes an accessible system block. For ST25/SRT512 cards, the system block is accessible at address 255. When a system block is available, it MAY be read using the appropriate `prepare*` operations during selection or transaction processing.
Returns	`true` if the product type exposes a system block, `false` otherwise.
Throws	None.

Has Write Acknowledgment

Signature	`hasWriteAcknowledgment() → boolean`
Since	1.0
Description	Indicates whether the product type provides a reliable acknowledgment after write operations. When this operation returns `true`, a successful write response guarantees that the data has been correctly stored. When it returns `false`, a verification read is performed to confirm the write.
Returns	`true` if the card provides a reliable write acknowledgment, `false` otherwise.
Throws	None.

5.4. Exceptions

5.4.1. SC Authentication Failed Exception

Name	`SCAuthenticationFailedException`
Kind	Runtime exception
Namespace	`storagecard`
Extends	`CardCommunicationException` (CNA-TR-API)
Implements	`StorageCardException`
Since	1.1
Purpose	Indicates that an authentication attempt on a storage card has failed (typically due to incorrect key data or key type).

5.4.2. SC Card Communication Exception

Name	`SCCardCommunicationException`
Kind	Runtime exception
Namespace	`storagecard`
Extends	`CardCommunicationException` (CNA-TR-API)
Implements	`StorageCardException`
Since	1.0
Purpose	Indicates an input/output error during the dialog with the storage card — e.g. transmission failures, card removal during processing, or a failed automatic verification read after a write operation on cards lacking reliable write acknowledgment.

5.4.3. SC Invalid Card Response Exception

Name	`SCInvalidCardResponseException`
Kind	Runtime exception
Namespace	`storagecard`
Extends	`InvalidCardResponseException` (CNA-TR-API)
Implements	`StorageCardException`
Since	1.0
Purpose	Indicates that a command returned an unexpected or invalid status while interacting with the storage card.

5.4.4. SC Reader Communication Exception

Name	`SCReaderCommunicationException`
Kind	Runtime exception
Namespace	`storagecard`
Extends	`ReaderCommunicationException` (CNA-TR-API)
Implements	`StorageCardException`
Since	1.0
Purpose	Indicates a low-level reader communication failure (lost connection, hardware malfunction, driver issue) preventing the command from being transmitted to the storage card.

5.4.5. Storage Card Exception

Name	`StorageCardException`
Kind	Interface (marker for exceptions)
Namespace	`storagecard`
Since	1.0
Purpose	Interface implemented by every exception raised during the execution of a command on a `StorageCard`. Provides additional context about the block address involved in the error.

Get Block Address

Signature	`getBlockAddress() → integer?`
Since	1.0
Description	Returns the address of the block involved in the error, when applicable.
Returns	The block address that caused the error, or `null` if not relevant.
Throws	None.

Get ID Command

Signature	`getIdCommand() → integer?`
Since	2.0
Description	Returns the application-supplied identifier of the command that caused the exception, when the failing command was prepared with an `idCommand` overload.

Returns	The command identifier, or `null` if no identifier was supplied for the failing command or if the exception is not attached to a specific command.
Throws	None.