**ID1 Developer Guide**

# Technical Description

**Document Release Version:** V1.0

# Table of Contents

# Introduction

The main aim of this document is to empower software engineers and developers to create applications that make use of the interface of the security chip of the latest iteration of EstEID smart card. It would be useful for the reader of this document to be familiar with smart card and chip application related topics but it's not strictly required. By exploring the general operations section and code examples together, readers with a background in software engineering should be able to follow along.

# Document scope

This document describes the basic use cases of the default application available on the chip such as:

- reading data from the chip
- changing and blocking/unblocking PINs
- signature computation
- authentication
- decryption

These operations are described as step by step instructions with code examples written in the Java programming language to accompany them. The commands that are used in examples are also explained in detail to provide context and to enable the reader to use variations of them to arrive at the same outcome. In its current iteration it does not however cover personalisation, configuration and maintenance. Also the reaction and behaviour of the application and the card to experiments, tests and attack attempts is out of scope of this document.

# References

ISO/IEC 7816-3 (2006): “Identification cards - Integrated circuit cards - Part 3: Cards with contacts — Electrical interface and transmission protocols".

ISO/IEC 7816-4 (2013): “Identification cards - Integrated circuit cards - Part 4: Organization, security and commands for interchange”.

ISO/IEC 7816-8 (2014): “Identification cards - Integrated circuit cards - Part 8: Commands for security operations”.

EstEID v. 3.5 (2013): "Estonian Electronic ID–card application specification".

NIST Special Publication 800-56A Revision 3 (2018): "Recommendation for Pair-Wise KeyEstablishment Schemes Using Discrete Logarithm Cryptography - Chapter 6: Key agreement schemes"

FIPS PUB 186-4 (2013): "Digital Signature Standard (DSS) - Chapter 6: The Elliptic Curve Digital Signature Algorithm (ECDSA)"

## Terms and operators

- **||** - concatenation operation
- **0x** - Marks that the following number is presented in hexadecimal format.
- **ADF** - Application Directory File
- **AID** - Application identifier
- **APDU** - Application protocol data unit
- **ASCII** - The standard 7-bit code table to present digitally the English alphabet and other keyboard symbols.
- **AT** - Authentication template
- **bit** - Marks that the number is presented in binary format.
- **CSE** - Current security environment
- **CRT** - Control reference template
- **CT** - Confidentiality template
- **dec** - Marks that the number is presented in decimal format.
- **DF** - Directory file
- **DST** - Digital signature template
- **ECDH** - Elliptic curve Diffie-Hellman is an encryption or more precisely a key-agreement protocol used in elliptic curve cryptography
- **ECDSA** - Elliptic curve digital signature algorithm
- **EF** - Elementary file
- **FCP** - File control parameter
- **FID** - File id
- **hex** - Marks that the number is presented in hexadecimal format.
- **MF** - The DF at the root is called the master file (MF). The MF is mandatory.

- **PIN** - Personal identification code
- **SDO** - Security data object
- **SHA** - Secure hash algorithm
- **TLV** - Binary data structure of Tag, Length and Value

# Chip and card application

## Card Platform

ID1 is the n-th Estonian eID platform that is implemented on top of ID-One™ Cosmo v8.1, which is certified as an open platform CC EAL5+. The platform includes an application loading mechanism, which has also CC EAL5+ level certification. ID-One™ Cosmo is installed on a separate domain on the java global platform. Additional applications are loaded onto a separate domain. Domains are protected by a firewall between them which ensures compliance with CC EAL5+ certification. As a result, even if a non-evaluated applet is loaded the security is not compromised and the certificate remains valid. The certification of an external application is also strongly simplified by this existing certificate by simple composition on the platform. The Cosmo platform is compliant with the latest international standards:

- JavaCard™ 3.0.4 Classic Edition
- Global Platform v2.2.1 (ID Configuration v1.0)
- ISO/IEC 7816 parts 1, 2, 3, 4, 5, 6, 8 and 9
- ISO/IEC 14443 Type A

## Answer to reset (Contact interface)

Every contact card responds to reset with sequence of bytes called Answer To Reset (ATR). The ATR gives information about the electrical communication protocol and the chip itself. It is mainly linked with the underlying integrated circuit (chip) and also the Java operating system on it. There is a block of historical bytes that can be used to indicate the purpose of the chip card.

The ATR can be different depending on if the reset is the first since power-up (Cold ATR) or not (Warm ATR). The meaningful info of ATR can be read from historical bytes of Cold ATR. Together with a category indicator byte the historical bytes form a string of 10 bytes with the value "00 12 23 3F 53 65 49 44 0F 90 $00_{hex}$".

| **TS** | 3B | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Protocol bytes**$_{hex}$ | **T0** | **TA1** | **TC1** | **TD1** | **TD2** | **TA3** | **TB3** | **TD3** | **TA15** |
| | DB | 96 | 00 | 80 | B1 | FE | 45 | 1F | 83 |
| **Historical bytes**$_{hex}$ | **T1** | **T2** | **T3** | **T4** | **T5** | **T6** | **T7** | **T8** | |
| | 00 | 31 | C1 | 64 | 084021 | | | | |
| **Additional bytes**$_{hex}$ | **STATE** | **SW1** | **SW2** | **TCK** | | | | | |
| | 00 | 90 | 00 | XX | | | | | |

The ATR resulting from default choices is detailed below:

- TA1=’96’: F=512, D=32, i.e. 307 200 bauds
- TC1=’00’: No Extra Guard time (specific to T=0 protocol - character time = 12 etu)
- TD1=’80’: Card bi-protocol T=0/T=1
- TA3=’FE’: IFSC=254 bytes (specific to T=1 protocol)
- TB3=’45’: Waiting times BWI=4, CWI=5 (specific to T=1 protocol)
- TA15=’83’: Clock stop indicator state H (high) and class A & B (class C is not supported)
- Historical Bytes: 0012233053654944 0F 9000
    - Category Indicator: 0x00
    - Country Code (ISO 3166-1): 0x233F (Estonia)
    - Card’s issuer data: 0x654944 (“eID”)
    - LCS: 0x0F (Termination State)
    - SW: 0x9000
- TCK: 0xF1

The resulting specific ATR to Estonia is: 3B DB 96 00 80 B1 FE 45 1F 83 00 12 23 3F 53 65 49 44 0F 9000 F1

# ATS (Contactless interface)

(ATS):

- speed rate (kbit/s): 848 / 424 / 212 / 106
- Historical bytes: Default (same as contact) / Other (between 0 and 15 bytes)

Default parameters are:

- Baud rate = symmetrical 848 kb/s
- FWI + CID:
    - FWI = ‘C’, FWT = 1.237s
    - CID supported
- Historical Bytes: 0012233053654944 0F 9000
    - Category Indicator: 0x00
    - Country Code (ISO 3166-1): 0x233F (Estonia)
    - Card’s issuer data: 0x654944 (“eID”)
    - LCS: 0x0F (Termination State)
    - SW: 0x9000

VHBR (Very High Baud Rate) is activated.

The resulting specific ATS to Estonia is: 3B 8B 80 01 00 12 23 3F 53 65 49 44 0F 90 00 A0

# PKI application

The application enabling PKI functionalities in Estonian eID Documents is IAS-ECC, a sophisticated but standardised solution conforming to CEN TS 15480-2 (European eID) with extra features. Everything detailed in the inter-industry standard “EUROPEAN CARD FOR e-SERVICES AND NATIONAL e-ID APPLICATIONS – Technical Specifications” (rev. 1.0.1).

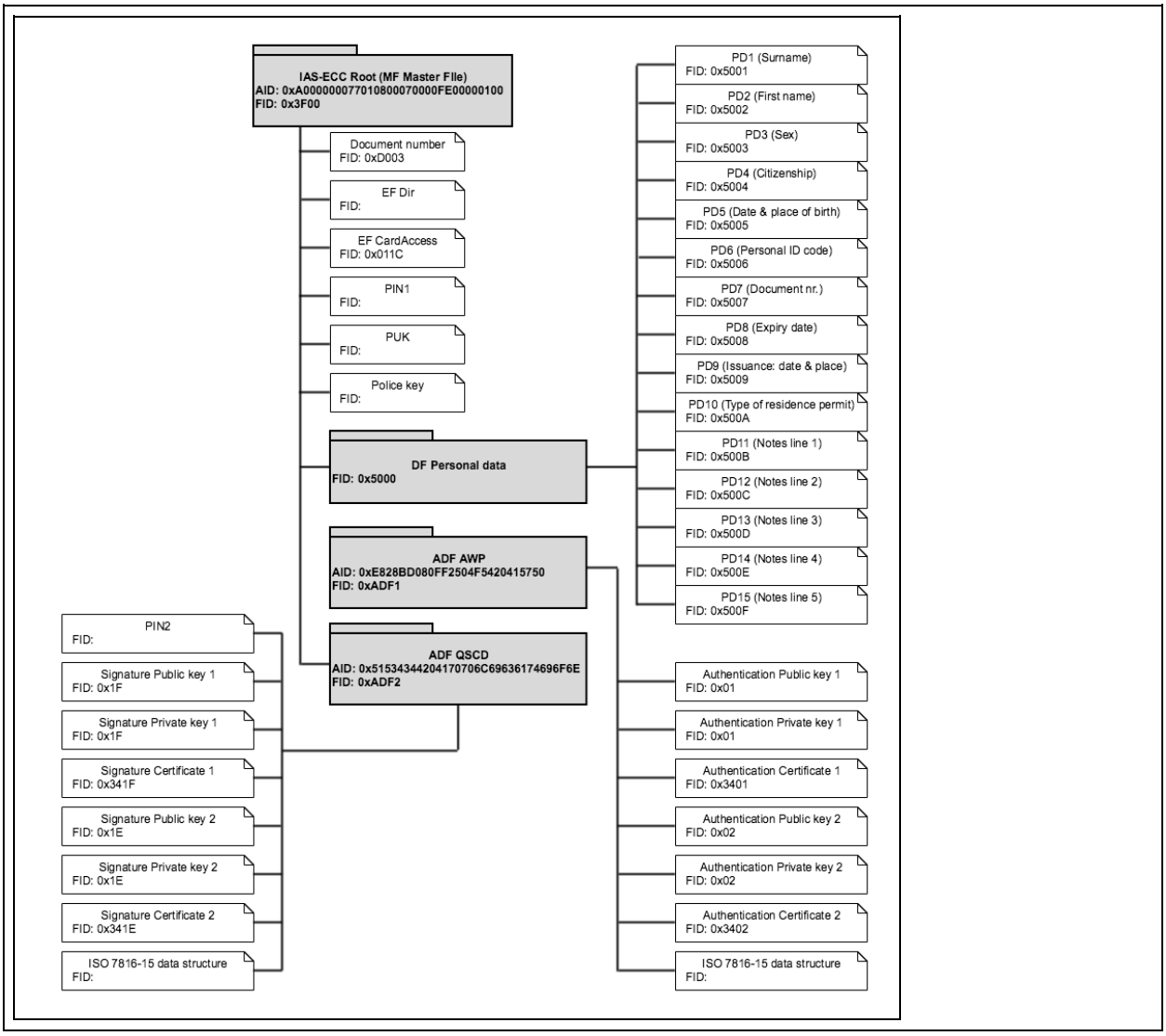
IAS-ECC, which stands for Identification Authentication Signature - European Citizen Card, is a PKI application which is QSCD certified according to the following Protection Profiles:

- CEN/EN 14169-2 (EN 419211-2) – Device with key generation
- CEN/EN 14169-3 (EN 419211-3) – Device with key import
- CEN/EN 14169-4 (EN 419211-4) – Extension for device with key generation and trusted communication with certificate generation application
- CEN/EN 14169-5 (EN 419211-5) – Extension for device with key generation and trusted communication with signature creation application
- CEN/EN 14169-6 (EN 419211-6) – Extension for device with key import and trusted communication with signature creation application

The several features available in IAS-ECC are for final user or for securing the usage on field.

# PKI Data Structure

Filesystem diagram:



# Card application objects and general operations

In this chapter we'll cover card application objects and main use cases for interacting with them. For more detailed information on the command- and response APDUs used in this chapter take a look at the Command APDUs section in this document. Some of the operations can be performed in multiple ways which are not all covered in this section. For more information on these ways you can again refer to the Command APDUs section or to ISO-7816-4. A good example of this is using the SELECT operation to navigate through the filesystem. Depending on your intent you can use different parameters to select DF, ADF or EF files or control whether the card application should or should not return file control parameters (FCP).

All the operations described here also have corresponding java code samples included in the Java code examples for general operations chapter.

## Document number

It is a transparent file holding the document number as defined by Estonian legislation: two prefix letters and a seven digits unique number for the given prefix.

The Document Number is generated during personalisation phase.

Example: AS9991044

The document number is stored in a binary file as a TLV value, where the tag has $04_{hex}$ value and document number value is ASCII encoded.

### Reading document number from card application

Let the document number in our case be $AS9991044_{ASCII}$ - $41533939393130343 4_{hex}$

1. Select Master file by issuing a SELECT FILE command

| CLA | INS | P1 | P2 |
|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $00_{hex}$ | $0C_{hex}$ |

2. Select document number EF by issuing the following SELECT FILE command.

| CLA | INS | P1 | P2 | Lc | Data (FID) |
|---|---|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $02_{hex}$ | $0C_{hex}$ | $02_{hex}$ | $D003_{hex}$ |

3. To read data from the selected file issue a READ BINARY command. P1 and P2 are used to specify an offset in the selected file to start reading from.

| CLA | INS | P1 | P2 | Le |
|---|---|---|---|---|
| $00_{hex}$ | $B0_{hex}$ | $00_{hex}$ | $00_{hex}$ | $00_{hex}$ |

4. In case of a successful read the card responds with Response APDU holding a TLV value:

| Data (Document number) | SW1 | SW2 |
|---|---|---|
| $04094153393939313034 34_{hex}$ | $90_{hex}$ | $00_{hex}$ |

NB! The document number can also be read from Personal Data file (See next chapter).

# Personal data file

The same personal information that is visible on the card can also be read from the card application (except photo and signature image). Cardholders personal information is held in a Dedicated File (DF) with file identifier $5000_{hex}$. All personal data records are Elementary files (EF) with a transparent structure (Transparent EF) which means the EF is seen as a sequence of data units. Personal data files are all mandatorily present on the card but some fields may be empty. In case the personal data field is empty, the corresponding Data File exists, has a one-byte size and contains the 0x00 terminator byte. In case the personal data field is present, the data length defines personal data file size.

## ID card Personal data file example

| File | Content | Object ID | Encoding | Sample value | Data format |
|---|---|---|---|---|---|
| PD1 | Surname | $5001_{hex}$ | ASCII/UTF-8 | JÕEORG | Xn |
| PD2 | First name | $5002_{hex}$ | ASCII/UTF-8 | JAAK-KRISTJAN | Xn |
| PD3 | Sex | $5003_{hex}$ | ASCII/UTF-8 | M | X |
| PD4 | Citizenship | $5004_{hex}$ | ASCII/UTF-8 | EST | XXX |
| PD5 | Date and place of birth | $5005_{hex}$ | ASCII/UTF-8 | 08 01 1980 EST | DD MM YYYY XXX |

| | | | | | |
|---|---|---|---|---|---|
| PD6 | Personal identification code | $5006_{hex}$ | ASCII/UTF-8 | 38001085718 | 99999999999 |
| PD7 | Document number | $5007_{hex}$ | ASCII/UTF-8 | AS9991044 | XX9999999 |
| PD8 | Expiry date | $5008_{hex}$ | ASCII/UTF-8 | 18 10 2023 | DD MM YYYY |
| PD9 | Date of issuance | $5009_{hex}$ | ASCII/UTF-8 | 18 10 2018 | DD MM YYYY |
| PD10 | Type of residence permit | $500A_{hex}$ | ASCII/UTF-8 | | |
| PD11 | Notes line 1 | $500B_{hex}$ | ASCII/UTF-8 | | |
| PD12 | Notes line 2 | $500C_{hex}$ | ASCII/UTF-8 | | |
| PD13 | Notes line 3 | $500D_{hex}$ | ASCII/UTF-8 | | |
| PD14 | Notes line 4 | $500E_{hex}$ | ASCII/UTF-8 | | |
| PD15 | Notes line 5 | $500F_{hex}$ | ASCII/UTF-8 | | |

## Digital Identity Card (eResident) Personal data file example

| File | Content | Object ID | Encoding | Sample value | Data format |
|---|---|---|---|---|---|
| PD1 | Surname | $5001_{hex}$ | ASCII/UTF-8 | JÕEORG | Xn |
| PD2 | First name | $5002_{hex}$ | ASCII/UTF-8 | JAAK-KRISTJAN | Xn |
| PD3 | Sex | $5003_{hex}$ | ASCII/UTF-8 | | |
| PD4 | Citizenship | $5004_{hex}$ | ASCII/UTF-8 | | |
| PD5 | Date and place of birth | $5005_{hex}$ | ASCII/UTF-8 | | |

| | | | | | |
|---|---|---|---|---|---|
| PD6 | Personal identification code | $5006_{hex}$ | ASCII/UTF-8 | 38001085718 | 99999999999 |
| PD7 | Document number | $5007_{hex}$ | ASCII/UTF-8 | NS0000009 | XX9999999 |
| PD8 | Expiry date | $5008_{hex}$ | ASCII/UTF-8 | 18 10 2023 | DD MM YYYY |
| PD9 | Date of issuance | $5009_{hex}$ | ASCII/UTF-8 | 18 10 2018 | DD MM YYYY |
| PD10 | Type of residence permit | $500A_{hex}$ | ASCII/UTF-8 | | |
| PD11 | Notes line 1 | $500B_{hex}$ | ASCII/UTF-8 | | |
| PD12 | Notes line 2 | $500C_{hex}$ | ASCII/UTF-8 | | |
| PD13 | Notes line 3 | $500D_{hex}$ | ASCII/UTF-8 | | |
| PD14 | Notes line 4 | $500E_{hex}$ | ASCII/UTF-8 | | |
| PD15 | Notes line 5 | $500F_{hex}$ | ASCII/UTF-8 | | |

## Residence Permit Card Personal data file example

| File | Content | Object ID | Encoding | Sample value | Data format |
|---|---|---|---|---|---|
| PD1 | Surname | $5001_{hex}$ | ASCII/UTF-8 | JÕEORG | Xn |
| PD2 | First name | $5002_{hex}$ | ASCII/UTF-8 | JAAK-KRISTJAN | Xn |
| PD3 | Sex | $5003_{hex}$ | ASCII/UTF-8 | M | X |
| PD4 | Citizenship | $5004_{hex}$ | ASCII/UTF-8 | UKR | XXX |
| PD5 | Date and place of birth | $5005_{hex}$ | ASCII/UTF-8 | 08 01 1980 UKR | DD MM YYYY XXX |

| | | | | | |
|---|---|---|---|---|---|
| PD6 | Personal identification code | $5006_{hex}$ | ASCII/UTF-8 | 38001085718 | 99999999999 |
| PD7 | Document number | $5007_{hex}$ | ASCII/UTF-8 | PS0000038 | XX9999999 |
| PD8 | Expiry date | $5008_{hex}$ | ASCII/UTF-8 | 18 10 2023 | DD MM YYYY |
| PD9 | Date and place of Issuance | $5009_{hex}$ | ASCII/UTF-8 | 10 2018 PPA | MM YYYY XXX |
| PD10 | Type of residence permit | $500A_{hex}$ | ASCII/UTF-8 | PIKAAJALINE ELANK | Xn |
| PD11 | Notes line 1 | $500B_{hex}$ | ASCII/UTF-8 | | |
| PD12 | Notes line 2 | $500C_{hex}$ | ASCII/UTF-8 | | |
| PD13 | Notes line 3 | $500D_{hex}$ | ASCII/UTF-8 | | |
| PD14 | Notes line 4 | $500E_{hex}$ | ASCII/UTF-8 | | |
| PD15 | Notes line 5 | $500F_{hex}$ | ASCII/UTF-8 | | |

## Diplomatic Identity Card Personal data file example

| File | Content | Object ID | Encoding | Sample value | Data format |
|---|---|---|---|---|---|
| PD1 | Surname | $5001_{hex}$ | ASCII/UTF-8 | THOMPSON | Xn |
| PD2 | First name | $5002_{hex}$ | ASCII/UTF-8 | STEVEN PAUL | Xn |
| PD3 | Sex | $5003_{hex}$ | ASCII/UTF-8 | | |
| PD4 | Citizenship | $5004_{hex}$ | ASCII/UTF-8 | | |
| PD5 | Date of birth | $5005_{hex}$ | ASCII/UTF-8 | 11 08 1975 | DD MM YYYY |

| | | | | | |
|---|---|---|---|---|---|
| PD6 | Personal identification code | $5006_{hex}$ | ASCII/UTF-8 | 37508110387 | 99999999999 |
| PD7 | Document number | $5007_{hex}$ | ASCII/UTF-8 | A19000195 | XX9999999 |
| PD8 | Expiry date | $5008_{hex}$ | ASCII/UTF-8 | 18 10 2023 | DD MM YYYY |
| PD9 | Date and place of Issuance | $5009_{hex}$ | ASCII/UTF-8 | 18 10 2018 | DD MM YYYY |
| PD10 | Type of residence permit | $500A_{hex}$ | ASCII/UTF-8 | | |
| PD11 | Notes line 1 | $500B_{hex}$ | ASCII/UTF-8 | | |
| PD12 | Notes line 2 | $500C_{hex}$ | ASCII/UTF-8 | | |
| PD13 | Notes line 3 | $500D_{hex}$ | ASCII/UTF-8 | | |
| PD14 | Notes line 4 | $500E_{hex}$ | ASCII/UTF-8 | | |
| PD15 | Notes line 5 | $500F_{hex}$ | ASCII/UTF-8 | | |

## Reading personal info from card application

To read the personal information a combination of SELECT FILE and READ BINARY operations must be performed.

1. Select Master file by issuing a SELECT FILE command

| CLA | INS | P1 | P2 |
|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $00_{hex}$ | $0C_{hex}$ |

2. Select Personal data DF by issuing the following SELECT FILE command. The $P1=01_{hex}$ value means we are selecting a child DF.

| CLA | INS | P1 | P2 | Lc | Data (FID) |
|---|---|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $01_{hex}$ | $0C_{hex}$ | $02_{hex}$ | $5000_{hex}$ |

3. Select the personal data transparent EF that you wish to read data from by issuing another SELECT FILE command. The P1=$02_{hex}$ value means we are selecting a child EF. In the example below we are selecting the first name record but by replacing the object ID the same command can be used to select any personal file record.

| CLA | INS | P1 | P2 | Lc | Data (FID) |
|---|---|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $02_{hex}$ | $0C_{hex}$ | $02_{hex}$ | $5002_{hex}$ |

4. To read data from the selected file issue a READ BINARY command. P1 and P2 are used to specify an offset in the selected file to start reading from.

| CLA | INS | P1 | P2 | Le |
|---|---|---|---|---|
| $00_{hex}$ | $B0_{hex}$ | $00_{hex}$ | $00_{hex}$ | $00_{hex}$ |

5. In case of a successful read the card responds with Response APDU:

| Data | SW1 | SW2 |
|---|---|---|
| $4A41414B2D4B524953544A414E_{hex}$ | $90_{hex}$ | $00_{hex}$ |

6. **NB!** You can repeat 3. and 4. in a loop to read all personal data from the chip - Just increment the Data portion of the Command APDU in part 3 from $5001_{hex}$ to $500F_{hex}$

# PIN1, PIN2 and PUK code operations

For PIN and PUK examples let's assume the following code values

| Type | Text value | Hex value |
|---|---|---|
| PIN1 | $1234_{ASCII}$ | $31323334_{hex}$ |
| PIN2 | $12345_{ASCII}$ | $3132333435_{hex}$ |
| PUK | $12345678_{ASCII}$ | $3132333435363738_{hex}$ |

## Using the VERIFY command to read the remaining tries counter for PIN1, PIN2 and PUK codes

By issuing the VERIFY command without the actual code value (empty data field) the retry count can be read from the status bytes (SW1-SW2) of the Response APDU. When the body is empty, the command may be used either to retrieve the number 'X' of further allowed retries (SW1-SW2='$63CX_{hex}$') or to check whether the verification is not required (SW1-SW2='$9000_{hex}$').

### PIN1 or PUK tries left

1. Select Master file by issuing a SELECT FILE command

| CLA | INS | P1 | P2 |
|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $00_{hex}$ | $0C_{hex}$ |

2. Issue the following VERIFY command

   to verify **PIN1**

| CLA | INS | P1 | P2 |
|---|---|---|---|
| $00_{hex}$ | $20_{hex}$ | $00_{hex}$ | $01_{hex}$ |

   to verify **PUK**

| CLA | INS | P1 | P2 |
|---|---|---|---|
| $00_{hex}$ | $20_{hex}$ | $00_{hex}$ | $02_{hex}$ |

The card should respond with the following response APDU where **X** is the number of further allowed retries. For example if the retry count is 2 then SW1-SW2=63C2$_{hex}$ and if the retry count is 3 then SW1-SW2=63C3$_{hex}$ etc.

| **Data** | **SW1** | **SW2** |
|---|---|---|
| empty | 63$_{hex}$ | C**X**$_{hex}$ |

**PIN2 tries left**

PIN2 unlike PIN1 and PUK is located on the QSCD ADF so navigating to it requires an extra SELECT command

1. Select Master file by issuing a SELECT FILE command

| **CLA** | **INS** | **P1** | **P2** |
|---|---|---|---|
| 00$_{hex}$ | A4$_{hex}$ | 00$_{hex}$ | 0C$_{hex}$ |

2. Select QSCD application DF by issuing the following SELECT FILE command. The data field is the file id (FID) of the QSCD application DF.

| **CLA** | **INS** | **P1** | **P2** | **Lc** | **Data (FID of QSCD ADF)** |
|---|---|---|---|---|---|
| 00$_{hex}$ | A4$_{hex}$ | 01$_{hex}$ | 0C$_{hex}$ | 10$_{hex}$ | ADF2$_{hex}$ |

3. Issue the following VERIFY command

| **CLA** | **INS** | **P1** | **P2** |
|---|---|---|---|
| 00$_{hex}$ | 20$_{hex}$ | 00$_{hex}$ | 85$_{hex}$ |

4. The card should respond with the following response APDU where **X** is the number of further allowed retries. For example if the retry count is 2 then SW1-SW2=63C2$_{hex}$ and if the retry count is 3 then SW1-SW2=63C3$_{hex}$ etc.

| **Data** | **SW1** | **SW2** |
|---|---|---|
| empty | 63$_{hex}$ | C**X**$_{hex}$ |

## Changing PIN1, PIN2 or PUK code.

The values of PIN1, PIN2 and PUK codes can be replaced by issuing the CHANGE REFERENCE DATA command if the given code is not blocked.

**NB!** As can be seen in the tables below the length of the codes must be 12 bytes. To achieve this the codes have to be padded with $FF_{hex}$ on the right side. So in our PIN1 example $1234_{ASCII}$ is $31323334_{hex}$ and $31323334FFFFFFFFFFFFFFFF_{hex}$ including padding. Since we also need to send the new code value of $4321_{ASCII}$ which is $34333231_{hex}$ and $34333231FFFFFFFFFFFFFFFF_{hex}$ including padding. The combined command data value becomes $31323334FFFFFFFFFFFFFFFF34333231FFFFFFFFFFFFFFFF_{hex}$. The same principle applies to PIN2 and PUK codes.

### Change PIN1 or PUK codes

1. Select Master file by issuing a SELECT FILE command

| CLA | INS | P1 | P2 |
|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $00_{hex}$ | $0C_{hex}$ |

2. Issue the following CHANGE REFERENCE DATA command

   to change **PIN1** from 1234 to 4321

| CLA | INS | P1 | P2 | Lc | Data (existing PIN1 \|\| new PIN1) |
|---|---|---|---|---|---|
| $00_{hex}$ | $24_{hex}$ | $00_{hex}$ | $01_{hex}$ | $18_{hex}$ | $31323334FFFFFFFFFFFFFFFF34333231FFFFFFFFFFFFFFFF_{hex}$ |

   to change **PUK** from 12345678 to 87654321

| CLA | INS | P1 | P2 | Lc | Data (existing PUK \|\| new PUK) |
|---|---|---|---|---|---|
| $00_{hex}$ | $24_{hex}$ | $00_{hex}$ | $02_{hex}$ | $18_{hex}$ | $3132333435363738FFFFFFFF3837363534333231FFFFFFFF_{hex}$ |

**Change PIN2 code**

PIN2 unlike PIN1 and PUK is located on the QSCD ADF so navigating to it requires an extra SELECT command

1. Select Master file by issuing a SELECT FILE command

| CLA | INS | P1 | P2 |
|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $00_{hex}$ | $0C_{hex}$ |

2. Select QSCD application DF by issuing the following SELECT FILE command. The data field is the file id (FID) of the QSCD application DF.

| CLA | INS | P1 | P2 | Lc | Data (FID of QSCD ADF) |
|---|---|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $01_{hex}$ | $0C_{hex}$ | $10_{hex}$ | $ADF2_{hex}$ |

3. Issue the following CHANGE REFERENCE DATA command to change **PIN2** from 12345 to 54321

| CLA | INS | P1 | P2 | Lc | Data (existing PIN2 \|\| new PIN2) |
|---|---|---|---|---|---|
| $00_{hex}$ | $24_{hex}$ | $00_{hex}$ | $01_{hex}$ | $18_{hex}$ | $3132333435FFFFFFFFFFFFFF3534333231FFFFFFFFFFFFFF_{hex}$ |

## Unblocking PIN1 and PIN2 code

When PIN codes retry counter values has decremented to value 0 and gets blocked, it is possible to unblock them by resetting the retry counter. This operation is possible with command RESET RETRY COUNTER.

**Unblocking PIN1**

1. Select Master file by issuing a SELECT FILE command

| CLA | INS | P1 | P2 |
|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $00_{hex}$ | $0C_{hex}$ |

2. Verify PUK code

| CLA | INS | P1 | P2 | Lc | Data (PUK code '12345678' padded) |
|---|---|---|---|---|---|
| $00_{hex}$ | $20_{hex}$ | $00_{hex}$ | $02_{hex}$ | $0C_{hex}$ | $3132333435363738FFFFFFFF_{hex}$ |

3. After given command is executed and successful response (SW1-SW2=9000) returned it is possible to reset retry counter of PIN codes
4. Reset PIN1 with RESET RETRY COUNTER command. This resets the PIN1 with the new given value.

| CLA | INS | P1 | P2 | Lc | Data (new PIN1 code '1234' padded) |
|---|---|---|---|---|---|
| $00_{hex}$ | $2C_{hex}$ | $02_{hex}$ | $01_{hex}$ | $0C_{hex}$ | $31323334FFFFFFFFFFFFFFFF_{hex}$ |

**Unblocking PIN2**

PIN2 unlike PIN1 and PUK is located on the QSCD ADF. This means that after PUK verification it's needed to navigate to QSCD application DF before issuing the RESET RETRY COUNTER COMMAND.

1. Select Master file by issuing a SELECT FILE command

| CLA | INS | P1 | P2 |
|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $00_{hex}$ | $0C_{hex}$ |

2. Verify PUK code

| CLA | INS | P1 | P2 | Lc | Data (PUK code '12345678' padded) |
|---|---|---|---|---|---|
| $00_{hex}$ | $20_{hex}$ | $00_{hex}$ | $02_{hex}$ | $0C_{hex}$ | $3132333435363738FFFFFFFF_{hex}$ |

3. After given command is executed and successful response (SW1-SW2=9000) returned it is possible to reset retry counter of PIN codes
4. Select QSCD application DF by issuing the following SELECT FILE command. The data field is the file id (FID) of the QSCD application DF.

| CLA | INS | P1 | P2 | Lc | Data (FID of QSCD ADF) |
|---|---|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $01_{hex}$ | $0C_{hex}$ | $10_{hex}$ | $ADF2_{hex}$ |

5. Reset PIN2 with RESET RETRY COUNTER command. This resets the PIN2 with the new given value.

| CLA | INS | P1 | P2 | Lc | Data (new PIN2 code '12345' padded) |
|---|---|---|---|---|---|
| $00_{hex}$ | $2C_{hex}$ | $02_{hex}$ | $85_{hex}$ | $0C_{hex}$ | $3132333435FFFFFFFFFFFFFF_{hex}$ |

# Reading certificates

The chip contains 2 certificates. One for cardholder authentication, encrypt and decrypt operations and the second for cardholder digital signing operations. The certificates are located on different application definition files (ADF) with the authentication certificate on AWP application (FID = $ADF1_{hex}$) and the signing certificate on QSCD application (FID = $ADF2_{hex}$). Certificate files are transparent files and can be read with the READ BINARY command.

First navigate to the correct file

1. Select Master file by issuing a SELECT FILE command

| CLA | INS | P1 | P2 |
|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $00_{hex}$ | $0C_{hex}$ |

2. Select the correct ADF for certificate by issuing a SELECT FILE command
    1. For signing certificate select QSCD application

| CLA | INS | P1 | P2 | Lc | Data (FID) |
|---|---|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $01_{hex}$ | $0C_{hex}$ | $02_{hex}$ | $ADF2_{hex}$ |

    2. For authentication certificate select AWP application

| CLA | INS | P1 | P2 | Lc | Data (FID) |
|---|---|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $01_{hex}$ | $0C_{hex}$ | $02_{hex}$ | $ADF1_{hex}$ |

3. Select the certificate file by issuing another SELECT FILE command
    1. For signing certificate the object ID is $341F_{hex}$

| CLA | INS | P1 | P2 | Lc | Data (FID) |
|---|---|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $02_{hex}$ | $0C_{hex}$ | $02_{hex}$ | $341F_{hex}$ |

    2. For authentication certificate the object ID is $3401_{hex}$

| CLA | INS | P1 | P2 | Lc | Data (FID) |
|---|---|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $02_{hex}$ | $0C_{hex}$ | $02_{hex}$ | $3401_{hex}$ |

    3.

Now the certificate is ready for reading operations. The method described next is reading the file by sending multiple READ BINARY commands and changing the file reading offset for every command until the whole file has been read. Data of the result has to be concatenated together.

1. Send the first READ BINARY command in sequence

| **CLA** | **INS** | **P1** | **P2** | **Le** |
|---|---|---|---|---|
| $00_{hex}$ | $B0_{hex}$ | $00_{hex}$ | $00_{hex}$ | $00_{hex}$ |

2. Response:

| **Data** | **SW1** | **SW2** |
|---|---|---|
| part1 of certificate | $90_{hex}$ | $00_{hex}$ |

3. Continue sending READ BINARY commands while increasing the file reading offset by the data size of the previous read binary response every time until the chip responds with an error SW1-SW2=$6B00_{hex}$ (Wrong parameter(s) P1-P2) indicating that our pointer is farther than the file length.

| **CLA** | **INS** | **P1** | **P2** | **Le** |
|---|---|---|---|---|
| $00_{hex}$ | $B0_{hex}$ | $00_{hex}$ | $E7_{hex}$ | $00_{hex}$ |
| **CLA** | **INS** | **P1** | **P2** | **Le** |
| $00_{hex}$ | $B0_{hex}$ | $01_{hex}$ | $CE_{hex}$ | $00_{hex}$ |
| **CLA** | **INS** | **P1** | **P2** | **Le** |
| $00_{hex}$ | $B0_{hex}$ | $02_{hex}$ | $B5_{hex}$ | $00_{hex}$ |
| **CLA** | **INS** | **P1** | **P2** | **Le** |
| $00_{hex}$ | $B0_{hex}$ | $03_{hex}$ | $9C_{hex}$ | $00_{hex}$ |
| **CLA** | **INS** | **P1** | **P2** | **Le** |
| $00_{hex}$ | $B0_{hex}$ | $04_{hex}$ | $07_{hex}$ | $00_{hex}$ |

6. Error response (Wrong parameter(s) P1-P2) indicating that the pointer is farther than the file length and the file has been read:

| **Data** | **SW1** | **SW2** |
|---|---|---|
| empty | $6B_{hex}$ | $00_{hex}$ |

# Computing digital signature

Card application enables the calculation of the electronic signature using the EC key in two ways:

- Providing card application with already calculated hash for signing procedure.
- Providing card application with data to be hashed before the signature procedure.

## Computing digital signature for pre-calculated hash

This chapter describes the signing method where hash is calculated by the host application and provided to card application prior to signing operation. For authorising cardholder for given operation it is needed to authenticate the user with PIN2.

For this chapter let PIN2 code be $12345_{ascii}$ - $3132333435_{hex}$

1. Select Master file by issuing a SELECT FILE command

| CLA | INS | P1 | P2 |
|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $00_{hex}$ | $0C_{hex}$ |

2. Select QSCD Application DF by issuing SELECT FILE command

| CLA | INS | P1 | P2 | Lc | Data (FID) |
|---|---|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $01_{hex}$ | $0C_{hex}$ | $02_{hex}$ | $ADF2_{hex}$ |

3. Verify PIN2 using the VERIFY command

| CLA | INS | P1 | P2 | Lc | Data (PIN2 code '12345' padded) |
|---|---|---|---|---|---|
| $00_{hex}$ | $20_{hex}$ | $00_{hex}$ | $85_{hex}$ | $0C_{hex}$ | $3132333435FFFFFFFFFFFFFF_{hex}$ |

4. Prepare the card security environment for ECDSA operation with sign key by executing command MANAGE SECURITY ENVIRONMENT

| CLA | INS | P1 | P2 | Lc | Data (Cryptographic mechanism ref \|\| len \|\| value (ECDSA SHA-384) \|\| Private |
|---|---|---|---|---|---|

| | | | | | **key ref \|\| len \|\| value)** |
|---|---|---|---|---|---|
| $00_{hex}$ | $22_{hex}$ | $41_{hex}$ | $B6_{hex}$ | $09_{hex}$ | $80_{hex}$ \|\| $04_{hex}$ \|\| $FF150800_{hex}$ \|\| $84_{hex}$ \|\| $01_{hex}$ \|\| $9F_{hex}$ |

5. Compute digital signature by executing command COMPUTE DIGITAL SIGNATURE

| **CLA** | **INS** | **P1** | **P2** | **Lc** | **Data (Hash)** | **Le** |
|---|---|---|---|---|---|---|
| $00_{hex}$ | $2A_{hex}$ | $9E_{hex}$ | $9A_{hex}$ | $30_{hex}$ | Hash algorithm output | $00_{hex}$ |

**Using hash algorithms with shorter output than $30_{hex}$**

Hash algorithm outputs that have shorter bit length than the EC key bit-length (384) need to be padded with zeroes from the left.

- First let's look at SHA-256 hash algorithm
  SHA-256("JÕEORG"): 1DECA99947932296BC91E03831FF411C7437203F53351FB8532B1AC4B4C2257B$_{hex}$
  This has a length of $20_{hex}$ so we need to pad it with zeroes:
  0000000000000000000000000000001DECA99947932296BC91E03831FF411C7437203F53351FB8532B1AC4B4C2257B00$_{hex}$
  And the resulting command APDU would be:

| **CLA** | **INS** | **P1** | **P2** | **Lc** | **Data (Has** |
|---|---|---|---|---|---|
| $00_{hex}$ | $2A_{hex}$ | $9E_{hex}$ | $9A_{hex}$ | $30_{hex}$ | 0000000000000000000000000000001DECA99947932296BC91E0 |

- And now the same with SHA-384

  SHA-384("JÕEORG"): A053E7B6A279D215B67407E392ED62684B6D65965B7B2191AEA33638607BDE2B30B6015D843032D1824BC03888C89762$_{hex}$
  This has a length of $30_{hex}$ so no padding is necessary and the resulting command APDU would be:

| **CLA** | **INS** | **P1** | **P2** | **Lc** | **Data (Ha** |
|---|---|---|---|---|---|
| $00_{hex}$ | $2A_{hex}$ | $9E_{hex}$ | $9A_{hex}$ | $30_{hex}$ | A053E7B6A279D215B67407E392ED62684B6D65965B7B2191AEA |

**Using hash algorithms with longer output than $30_{hex}$**

When the length of the output of the hash function is greater than the bit length of curves order *n*, then the leftmost *n* bits of the hash function output block shall be used in any calculation using the hash function output during the generation or verification of a digital signature.

This means that hash algorithm outputs that have a longer bit-length than EC key bit-length need to be truncated and the data sent with the compute digital signature command would be the $384_{dec}$ leftmost bits of the hash.

Let's look at an example using the SHA-512 hash algorithm:

SHA-
512("JÕEORG"): 3D5D6073666A36A7CD68A1B1DD0A4CBEF3197DDE32AFEE5DF6001F
96D6FA1C65146212EB53C44FDED7333318D4E328C29A5EA6E910BB2F5F1EE0309C7A55
168E$_{hex}$
The length of our generated hash is $64_{dec} = 40_{hex}$ which is longer than we need and so we have to truncate it by taking the leftmost $48_{dec} = 30_{hex}$ bytes:
3D5D6073666A36A7CD68A1B1DD0A4CBEF3197DDE32AFEE5DF6001F96D6FA1C651462
12EB53C44FDED7333318D4E328C2$_{hex}$

And the resulting command APDU would be:

| CLA | INS | P1 | P2 | Lc | Data (Hash) |
|---|---|---|---|---|---|
| $00_{hex}$ | $2A_{hex}$ | $9E_{hex}$ | $9A_{hex}$ | $30_{hex}$ | 3D5D6073666A36A7CD68A1B1DD0A4CBEF3197DDE32AFEE5DF6001F |

# Calculating response for TLS challenge

This chapter covers the use of the INTERNAL AUTHENTICATE command to calculate the response for a TLS challenge. This command is used for the client/server authentication. To authorise cardholder to perform this operation authentication with PIN1 is required. In the context of this chaper let PIN1 code be $1234_{ascii}$ - $31323334_{hex}$.

1. Select Master file by issuing a SELECT FILE command

| CLA | INS | P1 | P2 |
|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $00_{hex}$ | $0C_{hex}$ |

2. Select AWP Application DF by issuing SELECT FILE command

| CLA | INS | P1 | P2 | Lc | Data (FID) |
|---|---|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $01_{hex}$ | $0C_{hex}$ | $02_{hex}$ | $ADF1_{hex}$ |

3. Authenticate cardholder with PIN1 using the VERIFY command to authorise executing command INTERNAL AUTHENTICATE

| CLA | INS | P1 | P2 | Lc | Data (PIN1 code '1234' padded) |
|---|---|---|---|---|---|
| $00_{hex}$ | $20_{hex}$ | $00_{hex}$ | $01_{hex}$ | $0C_{hex}$ | $31323334FFFFFFFFFFFFFFFF_{hex}$ |

4. Prepare the card security environment for ECDSA operation with auth key by executing command MANAGE SECURITY ENVIRONMENT

| CLA | INS | P1 | P2 | Lc | Data (Cryptographic mechanism ref \|\| len \|\| value \|\| Private key ref \|\| len \|\| value) |
|---|---|---|---|---|---|
| $00_{hex}$ | $22_{hex}$ | $41_{hex}$ | $A4_{hex}$ | $09_{hex}$ | $80_{hex}$ \|\| $04_{hex}$ \|\| $FF200800_{hex}$ \|\| $84_{hex}$ \|\| $01_{hex}$ \|\| $81_{hex}$ |

5. Calculate the response for TLS challenge by executing command INTERNAL AUTHENTICATE

| CLA | INS | P1 | P2 | Lc | Data | Le |
|---|---|---|---|---|---|---|
| $00_{hex}$ | $88_{hex}$ | $00_{hex}$ | $00_{hex}$ | TLS challenge length$_{hex}$ | TLS challenge | $00_{hex}$ |

# Decrypting public key encrypted data

Current chapter describes deriving the shared secret necessary for decryption by combining (ephemeral) public key with the authentication private key on the card. This is performed by the DECIPHER operation. To authorise cardholder to perform this operation authentication with PIN1 is required. In the context of this chapter let PIN1 code be $1234_{ascii}$ - $31323334_{hex}$.

1. Select Master file by issuing a SELECT FILE command

| CLA | INS | P1 | P2 |
|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $00_{hex}$ | $0C_{hex}$ |

2. Select AWP Application DF by issuing SELECT FILE command

| CLA | INS | P1 | P2 | Lc | Data (FID) |
|---|---|---|---|---|---|
| $00_{hex}$ | $A4_{hex}$ | $01_{hex}$ | $0C_{hex}$ | $02_{hex}$ | $ADF1_{hex}$ |

3. Authenticate cardholder with PIN1 using the VERIFY command to authorise executing command DECIPHER

| CLA | INS | P1 | P2 | Lc | Data (PIN1 code '1234' padded) |
|---|---|---|---|---|---|
| $00_{hex}$ | $20_{hex}$ | $00_{hex}$ | $01_{hex}$ | $0C_{hex}$ | $31323334FFFFFFFFFFFFFFFF_{hex}$ |

4. Prepare the card security environment for ECDH operation with auth private key by executing command MANAGE SECURITY ENVIRONMENT

| CLA | INS | P1 | P2 | Lc | Data (Cryptographic mechanism ref \|\| len \|\| value \|\| Private key ref \|\| len \|\| value) |
|---|---|---|---|---|---|
| $00_{hex}$ | $22_{hex}$ | $41_{hex}$ | $B8_{hex}$ | $09_{hex}$ | $80_{hex}$ \|\| $04_{hex}$ \|\| $FF300400_{hex}$ \|\| $84_{hex}$ \|\| $01_{hex}$ \|\| $81_{hex}$ |

5. Obtain shared secret by executing command DECIPHER

| CLA | INS | P1 | P2 | Lc | Data | Le |
|---|---|---|---|---|---|---|
| $00_{hex}$ | $2A_{hex}$ | $80_{hex}$ | $86_{hex}$ | Data $length_{hex}$ | Public key (Ephemeral) used when deriving shared secret | $00_{hex}$ |

# APDU protocol

Communication between a chip card and a host application is performed over application-level APDU protocol. Current chapter and subchapters gives the basics of APDU protocol usage. APDU protocol itself is specified in ISO 7816-4 standard.

APDU messages compromise two structures: one used by the host application to send commands to the card whereas one is used by the chip to send command response back to the host application. Data transmission between two ends is performed as master-slave communication where a host application is the master and a chip is the slave.

Command sent by a host application is called Command APDU (C-APDU) or simply APDU. Command sent by the chip as a response to C-APDU is called Response APDU (R-APDU).

APDU messages can be transmitted with two different transmission-level Transmission Protocol Data Units (TPDU) which are T0 and T1. The T0 and T1 protocols are used to support APDU protocols transmission between chip reader and chip itself. APDU protocol is used between the chip application and the chip reader.

T1 is block oriented protocol which enables blocks or grouped collections of data to be transferred. These data groups are transferred as a whole between chip and reader. The theoretical maximum length of T1 grouped collections for C-APDU is $65535_{dec}$ and for R-APDU is $65536_{dec}$ bytes. The practical maximum length depends on the used chip platform on which the application runs.

T0 is byte oriented protocol which means that the minimum data that can be transferred has a length of one byte. The maximum length of data structure that can be transferred with this protocol for C-APDU is $255_{dec}$ and for R-APDU is $256_{dec}$ bytes.

# APDU structure and contents

APDU structure defined in ISO 7816-4 standard is very similar to TDPU structure used in T0. When APDU is transmitted with T0, the elements of APDU exactly overlay the elements of TPDU.

## C-APDU structure

<table>
<tr><th colspan="4">Header</th><th colspan="3">Body</th></tr>
<tr><td>CLA</td><td>INS</td><td>P1</td><td>P2</td><td>Lc</td><td>Data</td><td>Le</td></tr>
<tr><td colspan="4"></td><td colspan="2">Optional for T1</td><td></td></tr>
<tr><td colspan="4"></td><td colspan="3">Optional for T0</td></tr>
</table>

## C-APDU contents

| Code | Name | Length | Description |
|---|---|---|---|
| CLA | Class | 1 | Instruction class - indicates the type of command, e.g. interindustry or proprietary |
| INS | Instruction | 1 | Instruction code - indicates the specific command, e.g. "write data" |
| P1 | Parameter 1 | 1 | Instruction parameter 1 |
| P2 | Parameter 2 | 1 | Instruction parameter 2 |

| | | | |
|---|---|---|---|
| Lc | Length | 0, 1 or 3 | Encodes the number (Nc) of bytes of command data to follow<br>• 0 bytes denotes Nc=0<br>• 1 byte with a value from 1 to 255 denotes Nc with the same value<br>• Extended APDU - 3 bytes, the first of which must be 0, denotes Nc in the range 1 to 65 535 (all three bytes may not be zero) |
| Data | Command data | Variable, equal to Lc | String of bytes sent in the data field of the command |

| | | | |
|---|---|---|---|
| Le | Length | 0, 1, 2 or 3 | Encodes the maximum number (Ne) of response bytes expected<br><br>• 0 bytes denotes Ne=0<br>• 1 byte in the range 1 to 255 denotes that value of Ne, or 0 denotes Ne=256<br>• 2 bytes (if extended Lc was present in the command) in the range 1 to 65 535 denotes Ne of that value, or two zero bytes denotes 65 536<br>• Extended APDU - 3 bytes (if Lc was not present in the command), the first of which must be 0, denote Ne in the same way |

| | | | |
|---|---|---|---|
| | | | as two-byte Le |

Keep in mind that by using T1 protocol either Le or data field has to be present always. When there is no specific value for Le or data field while using T1, then Le field must be set to value $00_{hex}$.

## R-APDU structure

| **Body** | **Trailer** | |
|---|---|---|
| Data | SW1 | SW2 |

## R-APDU contents

| **Code** | **Name** | **Length** | **Description** |
|---|---|---|---|
| Data | Data | Variable (at most Le if was present in C-APDU) | Sequence of bytes received in the data field of the response (Optional field) |
| SW1 | Status byte 1 | 1 | Command processing status |
| SW2 | Status byte 2 | 1 | Command processing qualifier |

## R-APDU indicating the operation was successful

The application responds with the following R-APDU to indicate the successful processing of C-APDU:

| Code | Name | Value |
|---|---|---|
| Data | Data | Optional - depending on the command this may or may not be present |
| SW1 | Status byte 1 | $90_{hex}$ |
| SW2 | Status byte 2 | $00_{hex}$ |

# Command APDUs

The following is a list and description of all the C-APDUs that are used within the context of this document in Card application objects and general operations. Card application APDU commands are derived from ISO 7816-4 but might not implement all given specification requirements.

## SELECT FILE

This command is used to select a file (EF, DF), the MF or an application (ADF). After a successful selection the file selected becomes the current file. After reset the current DF is the MF and no EF is selected.

The following rules shall apply:

- After a successful selection of a DF there is no selected EF
- After a successful selection of an ADF, the associated application is selected and becomes the current application, there is no selected EF
- If the selection is aborted due to an error, the current files selection is unchanged
- When selecting an EF, the current DF becomes the parent DF of the selected EF
- Following an ADF selection, the current DF is the ADF, and there is no current EF
- Upon IFD request, the command may return file FCP

| SELECT | |
|---|---|
| **Command Parameter** | **Meaning** |
| **CLA** | $00_{hex}$ |
| **INS** | $A4_{hex}$ |
| **P1** | Selection control - See table 1 below |
| **P2** | Selection control - See table 2 below |
| **Lc field** | Absent or length of data field<br>• 02 - to pass a FID<br>• 'xx' - to pass DF name or relative path |
| **Data field** | FID, DF name or relative path |
| **Le field** | empty or $00_{hex}$ or maximum length of data expected in response |

**Table 1: Selection, file and life cycle commands P1 possible bit values**

| b8 | b7 | b6 | b5 | b4 | b3 | b2 | b1 | Meaning | Command data field |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | x | x | **Selection by file identifier** | |
| | | | | | | 0 | 0 | Select MF | MF identifier or empty |
| | | | | | | 0 | 1 | Select child DF | DF identifier |
| | | | | | | 1 | 0 | Select EF under current DF | EF identifier |
| | | | | | | 1 | 1 | Select parent DF of the current DF. Upper limit = ADF or MF | None |
| | | | | 0 | 1 | x | x | **Selection by name** | |
| | | | | | | 0 | 0 | Select by DF name (ADF or MF) | AID |
| | | | | 1 | 0 | x | x | **Selection by path** | |
| | | | | | | 0 | 1 | Select from the current DF | Path without the current DF identifier |

**Table 2: Selection, file and life cycle commands P2 possible bit values**

| b8 | b7 | b6 | b5 | b4 | b3 | b2 | b1 | Meaning |
|---|---|---|---|---|---|---|---|---|

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | - | - | x | x | **File occurence** |
| | | | | - | - | 0 | 0 | First only occurence |
| | | | | x | x | - | - | **File control parameters (FCP)** |
| | | | | 0 | 0 | - | - | Not supported |
| | | | | 0 | 1 | - | - | Return FCP template, mandatory use of FCP tag and length |
| | | | | 1 | 1 | - | - | No data in response field |

| **SELECT response** | |
|---|---|
| **Response parameter** | **Meaning** |
| Data field | Absent or FCP (See table 3 below) |

| | |
|---|---|
| SW1-SW2 | All SW values are in hexadecimal format<br>• '6283' - Warning Selected file deactivated<br>• '6285' - The Selected file is in Terminate state<br>• '6985' - Current DF is "deactivated" or "terminated" / MF was not created<br>• '6A82' - File or application not found<br>• '6A86' - Incorrect parameters P1-P2<br>• '6A87' - Lc inconsistent with parameters P1-P2 |

**Table 3: FCP returned upon file selection**

| Template | Length | Value field | | | Presence | | |
|---|---|---|---|---|---|---|---|
| | | **Tag** | **Length** | **Content** | **ADF** | **DF** | **EF** |
| 62 | L62 | $80_{hex}$ | $02_{hex}$ | File length | - | - | **M** |
| | | $82_{hex}$ | $01_{hex}$ | File descriptor byte<br>• EF - $01_{hex}$<br>• DF - $38_{hex}$<br>• ADF/MF - $38_{hex}$ | **M** | **M** | **M** |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | $83_{hex}$ | $02_{hex}$ | File identifier | **M** | **M** | **M** |
| | | $84_{hex}$ | $05_{hex}$ to $10_{hex}$ | DF name (AID) | **M** | - | - |
| | | $88_{hex}$ | $00_{hex}$ or $01_{hex}$ | Short file identifier | - | - | **O** |
| | | $8A_{hex}$ | $01_{hex}$ | Life cycle status byte<br>• activated - $05_{hex}$<br>• deactivated - $04_{hex}$<br>• terminated - $0C_{hex}$ | **M** | **M** | **M** |
| | | $A1_{hex}$ | Var. | Security attributes in proprietary format | **M** | **M** | **M** |
| | | $A5_{hex}$ | Var. | Issuer discretionary data in BER-TLV format | **O** | **O** | **O** |
| | | $85_{hex}$ | Var. | Issuer discretionary data in NON BER-TLV format | **O** | **O** | **O** |

Legend:

- **-** means that the parameters is not returned
- **M** means the parameter is present
- **O** means the parameter is present if it was set at creation

## READ BINARY

The READ BINARY command is used to read binary data from Transparent EF.

| **READ BINARY** | |
|---|---|
| **Command Parameter** | **Meaning** |
| **CLA** | $00_{hex}$ |
| **INS** | $B0_{hex}$ |
| **P1-P2** | Offset to start reading from file - See table 1 below |
| **Le field** | number of bytes to read |

| **Table 1: P1 & P2 management - possible bit values** | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **P1** | | | | | | | | **P2** | | | | | | | |
| **b8** | **b7** | **b6** | **b5** | **b4** | **b3** | **b2** | **b1** | **b8** | **b7** | **b6** | **b5** | **b4** | **b3** | **b2** | **b1** |

| | | | | |
|---|---|---|---|---|
| 0 | Offset in the currently selected file over 15 bits $00_{hex}$ <= Offset <= $7FFF_{hex}$ | | | |
| 1 | 0 | 0 | Short File Identifier 1 <= SFI <= 30 | Offset in the file over 8 bits |

| **READ BINARY response** | |
|---|---|
| **Response parameter** | **Meaning** |
| Data field | Binary data |

| | |
|---|---|
| SW1-SW2 | All SW values are in hexadecimal format<br><br>• 6282 - End of file reached before reading ‘Ne’ bytes<br>• 6981 - Command incompatible with file structure<br>• 6982 - Security status not satisfied<br>• 6985 - Current DF is "deactivated" or "terminated" / MF was not created<br>• 6A80 - Wrong data<br>• 6A82 - File not found (no current EF)<br>• 6B00 - Wrong parameters P1-P2 : Offset + length is beyond the end of file<br>• 6700 - Wrong length (wrong Le field) |

## VERIFY

The VERIFY command is used to authenticate cardholder through PIN1, PIN2 or PUK code or to devalidate PIN1, PIN2, PUK. Codes must be provided in communication as ASCII character numbers.

| VERIFY | |
|---|---|
| **Command Parameter** | **Meaning** |
| **CLA** | $00_{hex}$ |
| **INS** | $20_{hex}$ |
| **P1** | $00_{hex}$ for verification or $FF_{hex}$ for devalidation |
| **P2** | See table 1 below |
| **Lc field** | $0C_{hex}$ or Absent or $00_{hex}$ |
| **Data field** | Candidate PIN/PUK (P1 = '$00_{hex}$' and the command is used to submit a PIN/PUK)<br>Or<br>Empty (P1 = '$FF_{hex}$' or P1 = '$00_{hex}$' and the command is used to audit the validation status)<br><br>If present The candidate PIN/PUK must be padded with $FF_{hex}$ on the right side so that the total length of the data field is $0C_{hex}$ e.g. if Code is $3132333435_{hex}$ then the data field value will be $3132333435FFFFFFFFFFFFFF_{hex}$ |
| **Le field** | Absent |

**Table 1: P2 encoding - possible bit values**

| b8 | b7 | b6 | b5 | b4 | b3 | b2 | b1 | Meaning |
|---|---|---|---|---|---|---|---|---|
| 1 | - | - | - | - | - | - | - | Local reference data Application |
| 0 | - | - | - | - | - | - | - | Global reference data (Card) |
| - | - | x | x | x | x | x | x | User authentication DO reference (Code reference) |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Forbidden |

| VERIFY response | |
|---|---|
| **Response parameter** | **Meaning** |
| Data field | Absent |

| | |
|---|---|
| SW1-SW2 | All SW values are in hexadecimal format<br><br>• 6A86 - P1 ≠ '00' and P1 ≠ 'FF'<br>• 6700 - PIN length is out of valid boundaries [2*Lmin - 2*Lmax]"<br>• 6A88 - Referenced PIN not found<br>• 6982 - Security Status not satisfied<br>• 6983 - Referenced PIN not successfully verified AND no subsequent tries are allowed (remaining tries counter reached 0)<br>• 6984 - Referenced PIN usage counter reached 0<br>• 6300 - No retry limit : User authentication failed (if Pin verification) or PIN is not validated (if Lc=0) |

| | |
|---|---|
| | • 63Cx - x = remaining tries : User authentication failed (if Pin verification) or PIN is not validated (if Lc=0).<br>• 9000 - user authentication successful. |

## CHANGE REFERENCE DATA

This command allows changing PIN1, PIN2 and PUK codes. To change PIN1, PIN2 or PUK codes you need to know the currently active code. PIN1, PIN2 and PUK codes must be provided in communication as ASCII character numbers.

| **CHANGE REFERENCE DATA** | |
|---|---|
| **Command Parameter** | **Meaning** |
| **CLA** | $00_{hex}$ |
| **INS** | $24_{hex}$ |
| **P1** | $00_{hex}$ |
| **P2** | See table 1 below |
| **Lc field** | $18_{hex}$ |
| **Data field** | Current Code \|\| New Code<br><br>The current code and new code must be padded with $FF_{hex}$ on the right side so that the total length of the data field is $18_{hex}$<br>e.g. if Code is $3132333435_{hex}$ and the new code is $35343231_{hex}$ then the data field value will be $3132333435FFFFFFFFFFFFFF35343231FFFFFFFFFFFFFF_{hex}$ |
| **Le field** | Absent |

| **Table 1: P2 encoding - possible bit values** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **b8** | **b7** | **b6** | **b5** | **b4** | **b3** | **b2** | **b1** | **Meaning** |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | - | - | - | - | - | - | - | Local reference data Application |
| 0 | - | - | - | - | - | - | - | Global reference data (Card) |
| - | - | x | x | x | x | x | x | User authentication DO reference (Code reference) |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Forbidden |

| **CHANGE REFERENCE DATA response** | |
|---|---|
| **Response parameter** | **Meaning** |
| Data field | Absent |

| | |
|---|---|
| SW1-SW2 | All SW values are in hexadecimal format<br><br>• 6A81 - Command not supported (state selectable)<br>• 6A86 - P1 ≠ '00'<br>• 6A88 - Referenced PIN not found<br>• 63Cx - Referenced PIN not successfully verified AND subsequent tries are allowed (error counter not null), x = remaining tries allowed<br>• 6700 - Lc ≠ '00' – PIN length is out of valid boundaries.<br>• 6983 - Referenced PIN not successfully verified AND no subsequent tries are allowed (remaining tries counter reached 0)<br>• 6984 - Referenced PIN usage |

| | |
|---|---|
| | counter reached 0<br>• 6982 - Security status not satisfied |

## RESET RETRY COUNTER

The RESET RETRY COUNTER command is used to unblock, devalidate or unblock and change PIN1 or PIN2 code. In order to unblock code PUK verification operation must be performed first.

| RESET RETRY COUNTER | |
|---|---|
| **Command Parameter** | **Meaning** |
| **CLA** | $00_{hex}$ |
| **INS** | $2C_{hex}$ |
| **P1** | $02_{hex}$ (to unblock and change PIN1/PIN2) or $03_{hex}$ (to unblock only) or $FF_{hex}$ (to devalidate PIN1/PIN2) |
| **P2** | See table 1 below |
| **Lc field** | Variable |
| **Data field** | Absent (P1 = '$03_{hex}$' or '$FF_{hex}$') or new reference data (P1 = '$02_{hex}$')<br><br>If present the new code must be padded with $FF_{hex}$ on the right side so that the total length of the data field is $0C_{hex}$ e.g. if new code is $3132333435_{hex}$ then the data field value will be $3132333435FFFFFFFFFFFFFF_{hex}$ |
| **Le field** | Absent |

| Table 1: P2 encoding - possible bit values | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **b8** | **b7** | **b6** | **b5** | **b4** | **b3** | **b2** | **b1** | **Meaning** |
| 1 | - | - | - | - | - | - | - | Local reference data Application |
| 0 | - | - | - | - | - | - | - | Global reference data (Card) |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| - | - | x | x | x | x | x | x | User authentication DO reference (Code reference) |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Forbidden |

| **RESET RETRY COUNTER response** | |
|---|---|
| **Response parameter** | **Meaning** |
| Data field | Absent |

| | |
|---|---|
| SW1-SW2 | All SW values are in hexadecimal format<br><br>• 6A81 - Command not supported (state selectable)<br>• 6A86 - P1 ≠ '02', P1 ≠ '03' and P1 ≠ 'FF'<br>• 6A88 - Referenced PIN not found<br>• 6700 - The length of the new reference data doesn't match with the length of the PIN reference container length (P1 = '02') or Lc ≠ '00' (P1 = '03' or 'FF').<br>• 6984 - Reference data not usable – Usage counter of referenced PIN raised 0.<br>• 6982 - Security status not satisfied |

## MANAGE SECURITY ENVIRONMENT (Set)

This command is used to change the currently active pointers to keys for security operations.

At any time, a current cryptographic context – named current security environment (CSE) - is available. This context gathers all the informations required to perform any possible cryptographic operation offered by the application:

- Authentication
- Signature/verification of signature
- Encryption/decryption
- Hashing
- Key agreement

For each cryptographic operation, the current security environment contains:

- The security object reference
- The algorithm identifier indicating the usage
- The mode of operation (signature/verification of signature, ...)

Before performing any cryptographic operations such as digital signature, authentication protocols,....the application looks into the current security environment to find the data needed (security object reference, algorithm identifier indicating the usage, mode of operation)

| **MANAGE SECURITY ENVIRONMENT (Set)** | |
|---|---|
| **Command Parameter** | **Meaning** |
| **CLA** | $00_{hex}$ |
| **INS** | $22_{hex}$ |
| **P1** | $41_{hex}$ |
| **P2** | See CRT chapter below |
| **Lc field** | Variable |

| | |
|---|---|
| **Data field** | Control reference template (CRT) See CRT chapter below |
| **Le field** | Absent |

**Control Reference Template (CRT)**

The Control Reference Template is used in the MANAGE SECURITY ENVIRONMENT Set command. Following is a description of CRT's and the corresponding P1-P2 parameters that must be used when security environment is set for signature computation, internal authentication/calculating response to TLS challenge and decryption. CRT's covered within this chapter are:

- AT - Authentication template (internal authentication/calculating response to TLS challenge)
- DST - Digital signature template (signature computation)
- CT - Confidentiality template (encryption key decipherment)

**Digital signature template (DST) for computing digital signature with EC private key**

| P1 | P2 | Length | Tag | Length | Value | Presence |
|---|---|---|---|---|---|---|
| $41_{hex}$ | $B6_{hex}$ | Variable | $80_{hex}$ | Variable | Algorithm identifier | Mandatory |
| | | | $84_{hex}$ | $01_{hex}$ | Asymmetric key object (private portion) reference (EC private key) | Mandatory |

| **Algorithm identifier values in DST for computing digital signature** | |
|---|---|
| **Algorithm identifier** | **Usage** |
| $14_{hex}$ or $FF110800_{hex}$ | Digital signature with ECDSA SHA-1 |
| $34_{hex}$ or $FF130800_{hex}$ | Digital signature with ECDSA SHA-224 |
| $44_{hex}$ or $FF140800_{hex}$ | Digital signature with ECDSA SHA-256 |
| $54_{hex}$ or $FF150800_{hex}$ | Digital signature with ECDSA SHA-384 |
| $64_{hex}$ or $FF160800_{hex}$ | Digital signature with ECDSA SHA-512 |

**Authentication template (AT) for calculating response to TLS challenge (client/server authentication)**

| P1 | P2 | Length | Tag | Length | Value | Presence |
|---|---|---|---|---|---|---|
| $41_{hex}$ | $A4_{hex}$ | Variable | $80_{hex}$ | Variable | Algorithm identifier | Mandatory |
| | | | $84_{hex}$ | $01_{hex}$ | Asymmetric key object (private portion) reference (EC private key) | Mandatory |

| Algorithm identifier values in AT for calculating response to TLS challenge | |
|---|---|
| **Algorithm identifier** | **Usage** |
| $04_{hex}$ or $FF200800_{hex}$ | Authentication with ECDSA without any data hashing |

**Confidentiality template (CT) for encryption key decipherment**

| P1 | P2 | Length | Tag | L | Value | Presence |
|---|---|---|---|---|---|---|
| $41_{hex}$ | $B8_{hex}$ | Variable | $80_{hex}$ | Variable | Algorithm identifier | Mandatory |
| | | | $84_{hex}$ | $01_{hex}$ | Asymmetric key object (private portion) reference (EC private key) | Mandatory |

| Algorithm identifier values in AT for calculating response to TLS challenge | |
|---|---|
| **Algorithm identifier** | **Usage** |
| $0B_{hex}$ or $FF300400_{hex}$ | Encryption key decipherment with ECDH |

## PERFORM SECURITY OPERATION - COMPUTE DIGITAL SIGNATURE

This command performs the digital signature creation using the signature EC private key on the card. To use this command the security environment must be set to use ECDSA operation with signature private key on QSCD application. Also PIN2 needs to be verified.

| **PERFORM SECURITY OPERATION - COMPUTE DIGITAL SIGNATURE** | |
|---|---|
| **Command Parameter** | **Meaning** |
| **CLA** | $00_{hex}$ |
| **INS** | $2A_{hex}$ |
| **P1** | $9E_{hex}$ |
| **P2** | $9A_{hex}$ |
| **Lc field** | Data field length (in case of hash off card): 0x30 (SHA-384 hash length)<br><br>For last round hashing and on card hashing: empty |
| **Data field** | Hash of data or absent<br><br>Hash algorithm outputs that have shorter bitlength than the EC key bitlength (384) - 0x30 need to be padded with zeroes from the left until the have a length of 0x30 including padding.<br><br>Hash algorithm outputs that have longer bitlength than the EC key bitlength (384) - 0x30 need to be truncated so that they have a length of 0x30. See https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf (chapter 6.4) for information on truncating a longer hash. |
| **Le field** | $00_{hex}$ or absent |

| **PERFORM SECURITY OPERATION - COMPUTE DIGITAL SIGNATURE response** | |
|---|---|
| **Response parameter** | **Meaning** |

| Data field | Digital signature |
|---|---|
| SW1-SW2 | All SW values are in hexadecimal format<br>• 6984 - Security Data Object (SDO) not usable<br>• 6985 - No hash available<br>• 6A81 - Command not supported (state selectable)<br>• 6A88 - Current Security environment problem<br>• 6A86 - Incorrect P1-P2<br>• 6982 - Security status not satisfied |

## PERFORM SECURITY OPERATION - DECIPHER

This command performs the encryption key decipherment - deriving the shared secret using the authentication EC private key on card and public key in the command data field. To use this command the security environment must be set to use encryption key decipherment with ECDH with authentication private key. Also PIN1 needs to be verified.

| **PERFORM SECURITY OPERATION - DECIPHER** | |
|---|---|
| **Command Parameter** | **Meaning** |
| **CLA** | $00_{hex}$ |
| **INS** | $2A_{hex}$ |
| **P1** | $80_{hex}$ |
| **P2** | $86_{hex}$ |
| **Lc field** | Variable (length of data field) |
| **Data field** | $00_{hex}$ \|\| Public key or Ephemeral public key |
| **Le field** | $00_{hex}$ or absent |

| **PERFORM SECURITY OPERATION - DECIPHER response** | |
|---|---|
| **Response parameter** | **Meaning** |
| Data field | Shared secret |

| | |
|---|---|
| SW1-SW2 | All SW values are in hexadecimal format<br><br>• 6A80 - Padding verification error. Length of data is too big or the public key is not in uncompressed format, its value is inconsistent or has a wrong length<br>• 6A81 - Command not supported (state selectable)<br>• 6A88 - Current Security environment problem<br>• 6A86 - Incorrect P1-P2<br>• 6982 - Decipher key access conditions not fulfilled, or extraction error<br>• 6984 - Security Data Object (SDO) not usable<br>• 6985 - Condition not satisfied<br>• 6982 - Security |

| | |
|---|---|
| | status not satisfied |

## INTERNAL AUTHENTICATE for client/server authentication

This command is used to authenticate the cardholder by the host side. Data field in C-APDU must contain challenge that will be encrypted with private key stored in the card. Response can be verified by using public key of the same key pair.

| **INTERNAL AUTHENTICATE** | |
|---|---|
| **Command Parameter** | **Meaning** |
| **CLA** | $00_{hex}$ |
| **INS** | $88_{hex}$ |
| **P1** | $00_{hex}$ |
| **P2** | $00_{hex}$ |
| **Lc field** | Variable (length of data field) |
| **Data field** | TLS Challenge<br><br>For ECDSA scheme the length of data shall not exceed the length in bits of the order of the generator |
| **Le field** | $00_{hex}$ or absent |

| **INTERNAL AUTHENTICATE response** | |
|---|---|
| **Response parameter** | **Meaning** |
| Data field | Authentication cryptogram |

| | |
|---|---|
| SW1-SW2 | All SW values are in hexadecimal format<br><br>• 6700 - Wrong length; no further indication.<br>• 6982 - Security status not satisfied<br>• 6984 - Security data object (SDO) not usable<br>• 6985 - Security environment content doesn’t allow processing the command.<br>• 6A81 - Command not supported (state selectable)<br>• 6A86 - P1P2 ≠ ‘0000’<br>• 6A88 - Reference data needed for internal authenticate not found |

# Java code examples for general operations

These are code examples written in Java. All of them use classes from javax.smartcardio.* package to establish a channel and communicate with the smartcard. The first chapter shows how to establish a channel to communicate with the smartcard. All other examples assume that the channel is established and we can use its *transmit* function.

# Establishing a channel

```
//Get a list of available card terminals
List<CardTerminal> terminals =
TerminalFactory.getDefault().terminals().list();

//You can select the correct terminal by filtering the list by name or
whether it has a card present
//For the purposes of this example let's just choose the first one from the
list
CardTerminal terminal = terminals.get(0);

//Connect with card (using the T=1 protocol)
Card card = terminal.connect("T=1");

//Establish a channel
CardChannel channel = card.getBasicChannel();
```

# Helper functions

we will be using 2 helper functions for concatenation and padding codes:

```
private static byte[] concat(byte[] ... byteArrays) throws IOException {
    ByteArrayOutputStream stream = new ByteArrayOutputStream();
    for (byte[] byteArray : byteArrays) {
        stream.write(byteArray);
    }
    return stream.toByteArray();
}

private static byte[] padCode(byte[] code) {
    byte[] padded = Arrays.copyOf(code, 12);
    Arrays.fill(padded, code.length, padded.length, (byte) 0xFF);
    return padded;
}
```

# Reading document number from card application

```
CardChannel channel = card.getBasicChannel();

//Select master file
CommandAPDU selectMF = new CommandAPDU(0x00, 0xA4, 0x00, 0x0C);
channel.transmit(selectMF);
//Select document number EF
CommandAPDU selectDocumentNumberEF = new CommandAPDU(0x00, 0xA4, 0x02, 0x0C,
new byte[]{(byte)0xD0, 0x03});
channel.transmit(selectDocumentNumberEF);
//Read binary and convert the response data into an UTF-8 string
CommandAPDU readBinary = new CommandAPDU(new byte[]{0x00, (byte)0xB0, 0x00,
0x00, 0x00});
ResponseAPDU response = channel.transmit(readBinary);

String documentNumber = new String(response.getData(), Charset.forName("UTF-
8")).trim();
```

# Reading personal info from card application

Reading first name value:

```
CardChannel channel = card.getBasicChannel();

//Select master file
CommandAPDU selectMF = new CommandAPDU(0x00, 0xA4, 0x00, 0x0C);
channel.transmit(selectMF);
//Select personal data DF
CommandAPDU selectPersonalDataFile = new CommandAPDU(0x00, 0xA4, 0x01, 0x0C,
new byte[]{0x50, 0x00});
channel.transmit(selectPersonalDataFile);
//Select first name record
CommandAPDU selectFirstNameRecord = new CommandAPDU(0x00, 0xA4, 0x02, 0x0C,
new byte[]{0x50, 0x02});
channel.transmit(selectFirstNameRecord);

//Read binary and convert the response data into an UTF-8 string
CommandAPDU readBinary = new CommandAPDU(new byte[]{0x00, (byte)0xB0, 0x00,
0x00, 0x00});
ResponseAPDU response = channel.transmit(readBinary);
String firstName = new String(response.getData(), Charset.forName("UTF-
8")).trim();
```

Reading all personal data records

```
CardChannel channel = card.getBasicChannel();

//Select master file
CommandAPDU selectMF = new CommandAPDU(0x00, 0xA4, 0x00, 0x0C);
channel.transmit(selectMF);
//Select personal data DF
CommandAPDU selectPersonalDataFile = new CommandAPDU(0x00, 0xA4, 0x01, 0x0C,
new byte[]{0x50, 0x00});
channel.transmit(selectPersonalDataFile);

//Read all records and add them to a list of strings
List<String> allRecords = new ArrayList<>();
CommandAPDU readBinary = new CommandAPDU(new byte[]{0x00, (byte)0xB0, 0x00,
0x00, 0x00});
for (int i = 1; i <= 15; i++) {
    CommandAPDU selectChildEF = new CommandAPDU(0x00, 0xA4, 0x02, 0x0C, new
byte[] {0x50, (byte) i});
    channel.transmit(selectChildEF);
    ResponseAPDU response = channel.transmit(readBinary);
    String record = new String(response.getData(), Charset.forName("UTF-
8")).trim();
    allRecords.add(record);
}
```

# Read remaining tries counter for PIN1, PIN2, PUK

## PIN1

```
CardChannel channel = card.getBasicChannel();
//Select master file
CommandAPDU selectMF = new CommandAPDU(0x00, 0xA4, 0x00, 0x0C);
channel.transmit(selectMF);
//Verify PIN1
CommandAPDU verifyPin1 = new CommandAPDU(0x00, 0x20, 0x00, 0x01);
ResponseAPDU response = channel.transmit(verifyPin1);
//Get retry count from SW
String sw = Integer.toHexString(response.getSW());
int retryCount = Integer.parseInt(sw.substring(sw.length() - 1));
```

## PUK

```
CardChannel channel = card.getBasicChannel();
//Select master file
CommandAPDU selectMF = new CommandAPDU(0x00, 0xA4, 0x00, 0x0C);
channel.transmit(selectMF);
//Verify PUK
CommandAPDU verifyPuk = new CommandAPDU(0x00, 0x20, 0x00, 0x02);
ResponseAPDU response = channel.transmit(verifyPuk);
//Get retry count from SW
String sw = Integer.toHexString(response.getSW());
int retryCount = Integer.parseInt(sw.substring(sw.length() - 1));
```

## PIN2

```
CardChannel channel = card.getBasicChannel();
//Select master file
CommandAPDU selectMF = new CommandAPDU(0x00, 0xA4, 0x00, 0x0C);
//Select QSCD ADF
byte[] qscdFid = new byte[] {(byte)0xAD, (byte)0xF2};
CommandAPDU selectQSCDAdf = new CommandAPDU(0x00, 0xA4, 0x01, 0x0C, qscdFid);
channel.transmit(selectQSCDAdf);
//Verify PIN2
CommandAPDU verifyPin2 = new CommandAPDU(0x00, 0x20, 0x00, 0x85);
ResponseAPDU response = channel.transmit(verifyPin2);


//Get retry count from SW
String sw = Integer.toHexString(response.getSW());
int retryCount = Integer.parseInt(sw.substring(sw.length() - 1));
```

# Changing PIN1, PIN2 or PUK code.

The values of PIN1, PIN2 and PUK codes can be replaced by issuing the CHANGE REFERENCE DATA command if the given code is not blocked.

### PIN1

```
CardChannel channel = card.getBasicChannel();
//Select master file
CommandAPDU selectMF = new CommandAPDU(0x00, 0xA4, 0x00, 0x0C);
channel.transmit(selectMF);
//Change PIN1 from 1234 to 4321
CommandAPDU changePin1 = new CommandAPDU(0x00, 0x24, 0x00, 0x01,
concat(padCode("1234".getBytes()), padCode("4321".getBytes())));
channel.transmit(changePin1);
```

### PUK

```
CardChannel channel = card.getBasicChannel();
//Select master file
CommandAPDU selectMF = new CommandAPDU(0x00, 0xA4, 0x00, 0x0C);
channel.transmit(selectMF);
//Change PUK from 12345678 to 87654321
CommandAPDU changePuk = new CommandAPDU(0x00, 0x24, 0x00, 0x02,
concat(padCode("12345678".getBytes()), padCode("87654321".getBytes())));
channel.transmit(changePuk);
```

### PIN2

```
CardChannel channel = card.getBasicChannel();
//Select master file
CommandAPDU selectMF = new CommandAPDU(0x00, 0xA4, 0x00, 0x0C);
//Select QSCD ADF
byte[] qscdFid = new byte[] {(byte)0xAD, (byte)0xF2};
CommandAPDU selectQSCDAdf = new CommandAPDU(0x00, 0xA4, 0x01, 0x0C, qscdFid);
channel.transmit(selectQSCDAdf);
//Change PIN2 from 12345 to 54321
CommandAPDU changePuk = new CommandAPDU(0x00, 0x24, 0x00, 0x85,
concat(padCode("12345".getBytes()), padCode("54321".getBytes())));
channel.transmit(changePuk);
```

# Unblocking PIN1 and PIN2 code

When PIN codes retry counter values has decremented to value 0 and gets blocked, it is possible to unblock them by resetting the retry counter. This operation is possible with command RESET RETRY COUNTER.

## PIN1

```
CardChannel channel = card.getBasicChannel();
//Select master file
CommandAPDU selectMF = new CommandAPDU(0x00, 0xA4, 0x00, 0x0C);
channel.transmit(selectMF);
//Verify PUK code
CommandAPDU verifyPuk = new CommandAPDU(0x00, 0x20, 0x00, 0x02,
padCode("12345678".getBytes()));
ResponseAPDU response = channel.transmit(verifyPuk);
//If PUK verification was successful then reset PIN1 with value 1234
if ("9000".equalsIgnoreCase(Integer.toHexString(response.getSW()))) {
    CommandAPDU resetRetryCounter = new CommandAPDU(0x00, 0x2C, 0x02, 0x01,
padCode("1234".getBytes()));
    channel.transmit(resetRetryCounter);
}
```

## PIN2

```
CardChannel channel = card.getBasicChannel();
/Select master file
CommandAPDU selectMF = new CommandAPDU(0x00, 0xA4, 0x00, 0x0C);
channel.transmit(selectMF);
//Verify PUK code
CommandAPDU verifyPuk = new CommandAPDU(0x00, 0x20, 0x00, 0x02,
padCode("12345678".getBytes()));
ResponseAPDU response = channel.transmit(verifyPuk);
//If PUK verification was successful then select QSCD ADF and reset PIN2 with
value 12345
if ("9000".equalsIgnoreCase(Integer.toHexString(response.getSW()))) {
        byte[] qscdFid = new byte[] {(byte)0xAD, (byte)0xF2};
        CommandAPDU selectQSCDAdf = new CommandAPDU(0x00, 0xA4, 0x01, 0x0C,
qscdFid);
        channel.transmit(selectQSCDAdf);
    CommandAPDU resetRetryCounter = new CommandAPDU(0x00, 0x2C, 0x02, 0x85,
padCode("12345".getBytes()));
    channel.transmit(resetRetryCounter);
}
```

# Reading certificates

The chip contains 2 certificates. One for cardholder authentication operations and the second for cardholder digital signing operations. The certificates are located on different application definition files (ADF) with the authentication certificate on AWP application (ADF1) and the signing certificate on QSCD application (ADF2). Certificate files are transparent files and can be read with the READ_BINARY command.

## Read authentication certificate

```
CardChannel channel = card.getBasicChannel();
//Select master file
CommandAPDU selectMF = new CommandAPDU(0x00, 0xA4, 0x00, 0x0C);
channel.transmit(selectMF);

//Select AWP application (fid = ADF1)
CommandAPDU selectADF1 = new CommandAPDU(0x00, 0xA4, 0x01, 0x0C, new
byte[]{(byte)0xAD, (byte)0xF1});
channel.transmit(selectADF1);
//Select auth certificate (fid=3401)
CommandAPDU selectCertificate = new CommandAPDU(0x00, 0xA4, 0x02, 0x0C, new
byte[] {0x34, 0x01});
channel.transmit(selectCertificate);

//Read certificate
ByteArrayOutputStream stream = new ByteArrayOutputStream();
boolean doneReading = false;
while (!doneReading) {
    CommandAPDU readBinary = new CommandAPDU(new byte[]{0x00, (byte)0xB0,
(byte)(stream.size() >> 8), (byte)stream.size(), 0x00});
    ResponseAPDU response = channel.transmit(readBinary);
    stream.write(response.getData());
    String sw = Integer.toHexString(response.getSW());
    if ("6B00".equalsIgnoreCase(sw)) {
        doneReading = true;
    }
}
byte[] cert = stream.toByteArray();
//Optionally create an X509Certificate
X509Certificate x509Certificate = (X509Certificate)
CertificateFactory.getInstance("X.509")
    .generateCertificate(new ByteArrayInputStream(cert));
```

## Read digital signature certificate

```
CardChannel channel = card.getBasicChannel();
//Select master file
CommandAPDU selectMF = new CommandAPDU(0x00, 0xA4, 0x00, 0x0C);
channel.transmit(selectMF);

//Select QSCD application (fid = ADF2)
CommandAPDU selectADF1 = new CommandAPDU(0x00, 0xA4, 0x01, 0x0C, new
byte[]{(byte)0xAD, (byte)0xF2});
channel.transmit(selectADF1);
//Select digital signature certificate (fid=341F)
CommandAPDU selectCertificate = new CommandAPDU(0x00, 0xA4, 0x02, 0x0C, new
byte[] {0x34, 0x1F});
channel.transmit(selectCertificate);

//Read certificate
ByteArrayOutputStream stream = new ByteArrayOutputStream();
boolean doneReading = false;
while (!doneReading) {
    CommandAPDU readBinary = new CommandAPDU(new byte[]{0x00, (byte)0xB0,
(byte)(stream.size() >> 8), (byte)stream.size(), 0x00});
    ResponseAPDU response = channel.transmit(readBinary);
    stream.write(response.getData());
    String sw = Integer.toHexString(response.getSW());
    if ("6B00".equalsIgnoreCase(sw)) {
        doneReading = true;
    }
}
byte[] cert = stream.toByteArray();
//Optionally create an X509Certificate
X509Certificate x509Certificate = (X509Certificate)
CertificateFactory.getInstance("X.509")
    .generateCertificate(new ByteArrayInputStream(cert));
```

# Computing digital signature for pre-calculated hash

```
CardChannel channel = card.getBasicChannel();
//Select master file
CommandAPDU selectMF = new CommandAPDU(0x00, 0xA4, 0x00, 0x0C);
//Select QSCD application
byte[] qscdFid = new byte[] {(byte)0xAD, (byte)0xF2};
CommandAPDU selectQSCDAdf = new CommandAPDU(0x00, 0xA4, 0x01, 0x0C, qscdFid);
channel.transmit(selectQSCDAdf);
//Verify PIN2
CommandAPDU verifyPin2 = new CommandAPDU(0x00, 0x20, 0x00, 0x85,
padCode("12345".getBytes()));
ResponseAPDU response = channel.transmit(verifyPin2);
//Proceed with signature calculation if pin2 verification was successful
if ("9000".equalsIgnoreCase(Integer.toHexString(response.getSW()))) {
        //Set Security environment
    CommandAPDU setEnvironment = new CommandAPDU(0x00, 0x22, 0x41, 0xB6, new
byte[] {(byte) 0x80, 0x04, (byte) 0xFF, 0x15, 0x08, 0x00, (byte) 0x84, 0x01,
(byte) 0x9F});
    channel.transmit(setEnvironment);
        //SHA-256 hash
    String text = "JÕEORG";
    byte[] sha256DigestValue = MessageDigest.getInstance("SHA-
256").digest(text.getBytes());
        //pad with zeroes
    byte[] padded = padWithZeroes(sha256DigestValue);
        //calculate digital signature
    CommandAPDU securityOperationComputeSignature =
        new CommandAPDU(concat(new byte[]{0x00, 0x2A, (byte)0x9E, (byte)0x9A,
(byte)padded.length}, padded, new byte[]{0x00}));
    byte[] signature =
channel.transmit(securityOperationComputeSignature).getData();
}

//Helper function to pad hash with zeroes
private static byte[] padWithZeroes(byte[] hash) throws IOException {
    if (hash.length >= 48) {
        return hash;
    }
    try (ByteArrayOutputStream toSign = new ByteArrayOutputStream()) {
        toSign.write(new byte[48 - hash.length]);
        toSign.write(hash);
        return toSign.toByteArray();
    }
}
```

# Calculating response for TLS challenge

```
CardChannel channel = card.getBasicChannel();
//Select master file
CommandAPDU selectMF = new CommandAPDU(0x00, 0xA4, 0x00, 0x0C);
channel.transmit(selectMF);
//Select AWP application
CommandAPDU selectADF1 = new CommandAPDU(0x00, 0xA4, 0x00, 0x0C, new
byte[]{(byte)0xAD, (byte)0xF1});
channel.transmit(selectADF1);
//Verify PIN1
CommandAPDU verifyPin1 = new CommandAPDU(0x00, 0x20, 0x00, 0x01,
padCode("1234".getBytes()));
ResponseAPDU verifyResponse = channel.transmit(verifyPin1);
//proceed if pin verification was successful
if ("9000".equalsIgnoreCase(Integer.toHexString(verifyResponse.getSW()))) {
        //Set security environment
    CommandAPDU setEnvironment =
        new CommandAPDU(0x00, 0x22, 0x41, 0xA4, new byte[] {(byte) 0x80,
0x04, (byte) 0xFF, 0x20, 0x08, 0x00, (byte) 0x84, 0x01, (byte) 0x81});
    channel.transmit(setEnvironment);
        //Use JÕEORG as challenge
    byte[] challenge = "JÕEORG".getBytes();
    //Send the INTERNAL AUTHENTICATE command and read response from
ResponseAPDU
    CommandAPDU internalAuthenticate =
        new CommandAPDU(concat(new byte[]{0x00, (byte)0x88, (byte)0x00,
(byte)0x00, (byte)challenge.length}, challenge, new byte[]{0x00}));
    byte[] response = channel.transmit(internalAuthenticate).getData();
}
```

# Decrypting public key encrypted data

```
CardChannel channel = card.getBasicChannel();
//Select master file
CommandAPDU selectMF = new CommandAPDU(0x00, 0xA4, 0x00, 0x0C);
channel.transmit(selectMF);
//Select AWP application
CommandAPDU selectADF1 = new CommandAPDU(0x00, 0xA4, 0x00, 0x0C, new
byte[]{(byte)0xAD, (byte)0xF1});
channel.transmit(selectADF1);
//Verify PIN1
CommandAPDU verifyPin1 = new CommandAPDU(0x00, 0x20, 0x00, 0x01,
padCode("1234".getBytes()));
channel.transmit(verifyPin1);
//Set security environment
CommandAPDU setEnvironment =
    new CommandAPDU(0x00, 0x22, 0x41, 0xB8, new byte[] {(byte) 0x80, 0x04,
(byte) 0xFF, 0x30, 0x04, 0x00, (byte) 0x84, 0x01, (byte) 0x81});
channel.transmit(setEnvironment);

//The encrypted data transmitted to the card application must be pre-padded
with 00hex
byte[] prefix = new byte[] {0x00};

//We need the ephemeral public key data for the card application to derive
the shared secret
byte[] publicKeyData = getEphemeralPublicKeyRawBitsFromSomewhere();
byte[] data = concat(prefix, publicKeyData);

//Derive shared secret
byte[] header = new byte[]{0x00, 0x2A, (byte)0x80, (byte)0x86};
byte[] headerWithLc = concat(header, new byte[]{(byte)data.length});
byte le = 0x00;

CommandAPDU deriveSharedSecret = new CommandAPDU(concat(headerWithLc, data,
new byte[]{le}));

byte[] sharedSecret = channel.transmit(deriveSharedSecret).getData();
```