Smartcards and contactless smartcards

Integrators's and Implementer's Guide

www.springcard.com

Ref. PMD17041-AA

Document identification

Category	Developer's Manual
Classification	Public
Reference	PMD17041
Version	AA
Status	Approved
Keywords	
Abstract	
File name	[PMD17041-AA] Smartcards and contactless smartcards- Integrator's and Implementers's Guide.odt
Print date	19/01/18

Revision history

Ver.	Date	Author	Valid. by		Approv. by	Details
			Tech.	Qual.		
AA	08/01/18	JDA		CMA	JDA	Creation

Contents

Illustration index

Table index

This guide book has been written for SpringCard by Johann Dantant.

Thanks to Laetitia Cochet for her contributions, Claire Maillet for her careful re-reading, and all the SpringCard R&D team for the technical inputs.

Warning

A few parts of this guide make reference to future documents, features or products, that are not yet available at the date of writing.

Such items are written in blue and associated to the "coming soon" or "TBD" (to be done) words.

1. Introduction

1.1. Overview

This is a guide for system integrators, software designers and developers who are creating smartcard-aware computer applications, especially applications that communicates with contactless cards, RFID labels or NFC tags through a **SpringCard coupler**.

This document aims to give the reader the basic skills that are required to conceive and implement a solution involving a smartcard or a contactless card. It contains a large part related to PC/SC-development, for PC/SC is the most widely-adopted API to create an application that communicates with a smartcard. Even if your solution does not involve PC/SC, the concepts will always be the same.

For a more general overview, or to go further into some features not covered by this handbook, consult the following references:

- **Smart Card Programming**, Ugo Chirico, Lulu Press Inc, 2014,
- **Smart Cards: The Developer's Toolkit**, Timothy M. Jurgensen and Scott B. Guthery, Pearsons Educations, 2002,
- **Les cartes à puce, théorie et mise en oeuvre**, Christian Tavernier, Dunod, 2011 *(in french),*
- **RFID and Contactless Smart Card Applications**, Dominique Paret, Wiley-Blackwell, 2005.

1.2. Audience

Readers of this guide are assumed to be familiar with computer application programming.

1.3. Related documents

This guide introduces many concepts, but does not aim to cover all aspects of them. The following documents are the related reference manuals or applications notes that should be read together with this guide:

Doc #	Title
PMDZ061	PC/SC Simplified documentation of the API
PMD15305	PC/SC Zero-driver – CCID low-level implementation
PMD17182	SpringCard PC/SC couplers – embedded APDU Processor
Coming soon	NFC peer-to-peer with SpringCard PC/SC couplers
Coming soon	Card and NFC tag emulation with SpringCard PC/SC couplers
Coming soon	Advanced control and configuration of SpringCard PC/SC couplers

Also refer to the Getting Started Guide accompanying the product you are using.

SpringCard publishes updates or new documents frequently. Please verify that you have always the latest version of every document.

1.4. Product listing

The following **SpringCard PC/SC couplers** are covered by this document:

H663 Group (USB)

Prox'N'Roll PC/SC HSP	Desktop USB PC/SC coupler for contactless/RFID/NFC smartcards
Prox'N'Roll PC/SC HSP OEM	OEM USB PC/SC coupler for contactless/RFID/NFC smartcards
CSB HSP	Multi-interface desktop USB PC/SC coupler (contactless/RFID/NFC smartcards + contact cards + 1 to 3 SIM/SAM)
CrazyWriter HSP	Multi-interface OEM USB PC/SC coupler (contactless/RFID/NFC smartcards with remote antenna + contact cards + 1 to 4 SIM/SAM)
TwistyWriter HSP	Turnkey OEM USB PC/SC contactless/RFID/NFC coupler solution (remote antenna + 1 SAM slot)
H663-USB	Turnkey OEM USB PC/SC contactless/RFID/NFC coupler solution
H663	OEM USB PC/SC contactless/RFID/NFC coupler

	module (without antenna)
H512 Group (USB)	
H512-USB	Turnkey OEM USB PC/SC NFC coupler solution
H512	OEM USB PC/SC NFC coupler module (without antenna)
E663 Group (network)	
FunkyGate-IP PC/SC	Ready-to-use contactless/RFID/NFC PC/SC over Ethernet wall coupling device
TwistyWriter-IP PC/SC	OEM contactless/RFID/NFC PC/SC over Ethernet couplers (remote antenna)
K663 Group (serial)	
Devices based on the K663 module don't have an actual PC/SC driver, but could be operated in PC/SC-like mode based on the CCID zero-driver implementation	
CSB4.8S	Contactless/RFID/NFC desktop coupler, serial interface
CSB4.8U	Contactless/RFID/NFC desktop coupler, USB serial port emulation
TwistyWriter-TTL or 232 or 485	OEM contactless/RFID/NFC serial interfaced couplers (remote antenna)
K663-TTL or 232 or 485	OEM contactless/RFID/NFC serial communication coupler module
CSB6 Group	
The CSB6 Group is no longer in production	
Prox'N'Roll PC/SC	Desktop USB PC/SC coupler for contactless/RFID smartcards
CSB6	Multi-interface desktop USB PC/SC coupler (contactless/RFID smartcards + contact cards + SIM/SAM)
CrazyWriter	Multi-interface OEM USB PC/SC coupler (contactless/RFID smartcards with remote antenna + 2 SIM/SAM)
CrazyWriter	OEM USB PC/SC coupler for contactless/RFID/NFC smartcards (remote antenna)
NFC'Roll	Desktop USB PC/SC NFC coupler

Please refer to each product’s page on www.springcard.com to download the relevant datasheet, getting started guide, and associated documents.

SpringCard introduces new products frequently. Latests products may not appear on the list, while being also covered by this document.

1.5. Reference documents

1.5.1. International standards

Standard	Description
ISO/IEC 7810	Identification cards – Physical characteristics
ISO/IEC 7816-2	Identification cards – Integrated circuit cards Part 2: Cards with contacts – Dimensions and location of the contacts
ISO/IEC 7816-3	Identification cards – Integrated circuit cards Part 3: Cards with contacts – Electrical interface and transmission protocols
ISO/IEC 7816-4	Identification cards – Integrated circuit cards Part 4: Organization, security and commands for interchange
ISO/IEC 7816-5	Identification cards – Integrated circuit cards Part 5: Registration of application providers
ISO/IEC 7816-6	Identification cards – Integrated circuit cards Part 6: Interindustry data elements for interchange
ISO/IEC 18000-3	Information technology – Radio frequency identification for item management Part 3: Parameters for air interface communications at 13.56 MHz

1.5.2. Public specifications

Reference	Publisher	Title
CCID	**USB Workgroup**	Universal Serial Bus Device Class: Smart Card Specification for Integrated Circuit(s) Cards Interface Devices Rev 1.1 – 22/04/2005 Download link: http://www.usb.org/developers/docs/devclass_docs/DWG_Smart-Card_CCID_Rev110.pdf
PC/SC	**PC/SC Workgroup**	Interoperability Specification for ICCs and Personal Computer Systems Revision 2 Download link: https://www.pcscworkgroup.com/specifications/download/

1.6. Support and updates

Useful related materials (product datasheets, application notes, sample software, HOWTOs and FAQs…) are available at SpringCard's website:

www.springcard.com

Updated versions of this document and others are posted on this website as soon as they are available.

For technical support enquiries, please refer to SpringCard support page:

www.springcard.com/support

1.7. Conventions used in this document

1.7.1. Typographic conventions for numbers

1.7.1.1. Hex notation

All hex numbers are prefixed by a small "h". Examples:

$_{h}$9AE4
$_{h}$0100

1.7.1.2. Binary notation

All binary numbers are prefixed by a small "b". Examples:

$_{b}$0001 1001
$_{b}$01

1.7.1.3. Decimal notation

Numbers without any prefix are decimal. Examples:

16
255

When absolutely required for clarity, decimal numbers are prefixed by a small "d". Examples:

$_{d}$16
$_{d}$255

1.7.2. Object size

The following designations are used when referring to the size of data objects:

- A **byte** is an 8-bit object,
- A **word** is a 16-bit object,
- A double-word or **dword** is a 32-bit object.

1.7.3. Iconography

Icon	Description
⚠	Warning: source of frequent errors or confusions
ℹ	Useful supplementary information, advice
↗	External link – Reference to a document or a page available on the web

1.8. Glossary and acronyms

The terms used in this document as generally defined on the first time they are used. The most common terms are listed below:

APDU	Application Protocol Datagram Unit Refer to a ISO/IEC 7816-4 command (C-APDU) or response (R-APDU)
ATR	Answer To Reset
CLA	Class The 1st byte of a ISO/IEC 7816-4 command (C-*APDU*)
CSN	Card Serial Number (see also *UID*, *RID*, *PUPI*)
INS	Instruction The 2nd byte of a ISO/IEC 7816-4 command (C-*APDU*)
NFC	Near Field Communication
PCD	Proximity Coupling Device
PICC	Proximity Integrated Circuit Card
PPS	Parameter & Protocol Selection
RFID	Radio Frequency IDentification
SW	Status Word The 2 status bytes at the end of a ISO/IEC 7816-4 response (R-*APDU*)
TPDU	Transport Protocol Datagram Unit Refer to a ISO/IEC 7816-3 or ISO/IEC 14443-4 block
VCD	Vicinity Coupling Device
VICC	Vicinity Integrated Circuit Card

A complete glossary is available on SpringCard's Technical Blog:

http://tech.springcard.com/glossary/

2. Smartcards and couplers – Concepts and definitions

2.1. What is a smartcard?

According to Wikipedia, "*a **smartcard**, **chip card** or **integrated circuit card** (**ICC**) is any pocket-sized card that has embedded integrated circuits*".

https://en.wikipedia.org/wiki/Smart_card

This is a very open definition, which obviously would cover not only the objects you would spontaneously name "smartcard", but also the SD card of your digital camera or mobile phone, the cartridges of the gaming console you used to play with as a child, and possibly a flat USB flash drive. Actually, all of these are "pocket-sized cards that has embedded integrated circuits" as well. But they are no smartcards.

At first, let's circumscribe the subject to what the engineers and the industry have standardised under the technical name 'smartcard': an IC card, having a standardised size, featuring a standardised electrical interface, implementing a standardised protocol, and finally exposing standardised software interfaces.

All these standards are grouped in the ISO/IEC 7816 cluster, entitled "identification cards – integrated circuit cards".

From now on, we are leaving Wikipedia's open definition behind us, and will consider that a **smartcard is an electronic device that obeys to (at least some of) the ISO/IEC 7816 standards**.

2.2. What is a smartcard, according to ISO/IEC 7816

2.2.1. Form-factor and electrical interface

ISO/IEC 7816-1 says the form-factor of a smartcard must be **ID-1**, as defined in ISO 7810, i.e. the "credit card" size we all know (85.60×63.98 mm, 3.37×2.13 in).

A few smaller form-factors have been introduced year after year by mobile phones manufacturer, and are now widely adopted:

- **2FF** "mini SIM" for the mobile phone manufacturers (25×15 mm, 0.98×0.59 in) has also been adopted by the smartcard industry for security coprocessors (SAM) under the name **ID-000**,

- **3FF** "micro SIM" (15×12 mm) and **4FF** "nano SIM" (12.30×8.80 mm) are used only in smartphones.

Illustration 1 below shows the different form-factors.

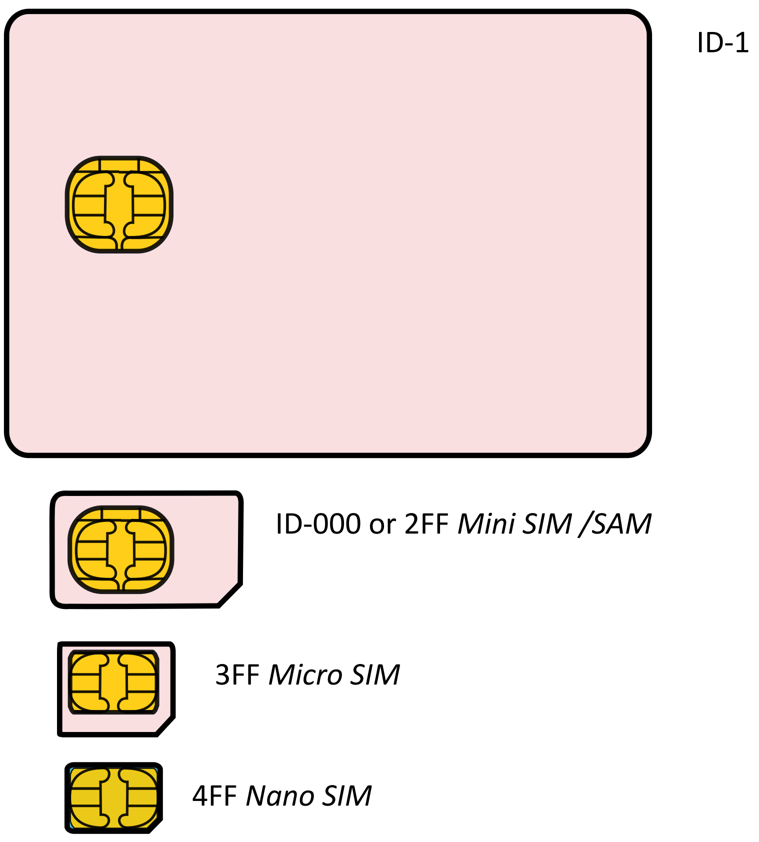


Illustration 1: Smartcard formats

The card communicates with the external world using a set of physical contacts, which also provides power and clock (ISO/IEC 7816-2, illustration 2).

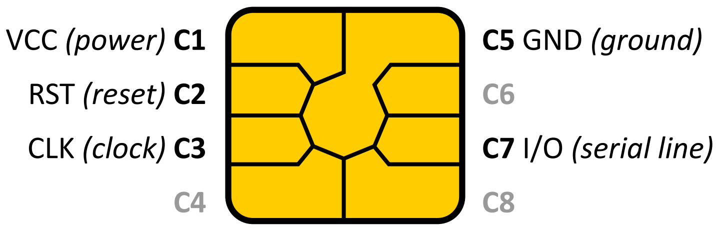


Illustration 2: Smartcard contacts

We'll no go deep into details concerning only the engineers working on the electrical interface, but there are three facts worth noticing:

- There are 3 power classes. Class A stands for VCC=5 V, class B for VCC=3.3 V and class C for VCC=1.8 V. A "general purpose" smartcard reader is responsible for powering the card with increasing voltages, until the card comes to life. A reader that is designed to always read the same card, or the same card family, could be optimized to support only a single power class.
- Contacts C4, C6 and C8 were required in the 80's, because the IC technology of the time required a higher voltage when programming (erase / write) the memories than when reading them. These contacts have been deprecated in the 90's, but are now gaining new roles: some cards may implement USB on C4 and C8[1], and, in some NFC-enabled smartphones, C6 is a direct link between the SIM card's chip and the NFC contactless front-end.
- In early generation cards, CLK was the unique clock source for the chip's processor. Nowadays, most cards have an internal clock source, yet the CLK line is still required to clock the serial communication. The reader must supply a constant clock signal[2].

2.2.2. Protocol

ISO/IEC 7816-3 specifies that the card implements an asynchronous serial transport protocol over the I/O pin; the bitrate is defined as a division of the input clock (CLK).

The smartcard protocol must put an emphasis on the error detection and recovery schemes, in order to improve the overall reliability of the link, because dirty or even damaged contacts (either on the card-side or on the reader-side) are the cause of frequent communication errors.

Actually, there are two protocols:

- **T=0** is a character-oriented protocol. Error detection and recovery take place after every byte (parity bit + acknowledge). This is a legacy of the early-80's, a slow protocol, yet usable even in harsh conditions.
- **T=1** is a block-oriented protocol. Blocks are 16 to 256 byte-long. It is lots faster than T=0 in the nominal situation where no communication error ever occurs, but, on the other hand, the error recovery process takes more time.

Since there is only one single I/O pin shared by both input and output lines, both protocols are half-duplex. The communication uses a *command/response* model: the reader sends down a *command*, and the card must provide its *response* in a given time frame.

The only exception to this half-duplex scheme is the initial power-up and reset sequence, when the card sends its first message spontaneously. This message is the card's *Answer to Reset* or **ATR.**

[1] You can see this kind of cards as the "combo" between a USB smartcard reader and the smartcard itself.
[2] The clock frequency could be selected by the reader between 1 and 5 MHz. SpringCard contact couplers clock the cards at 4 MHz.

Illustration 3 on page 20 summarizes this workflow.

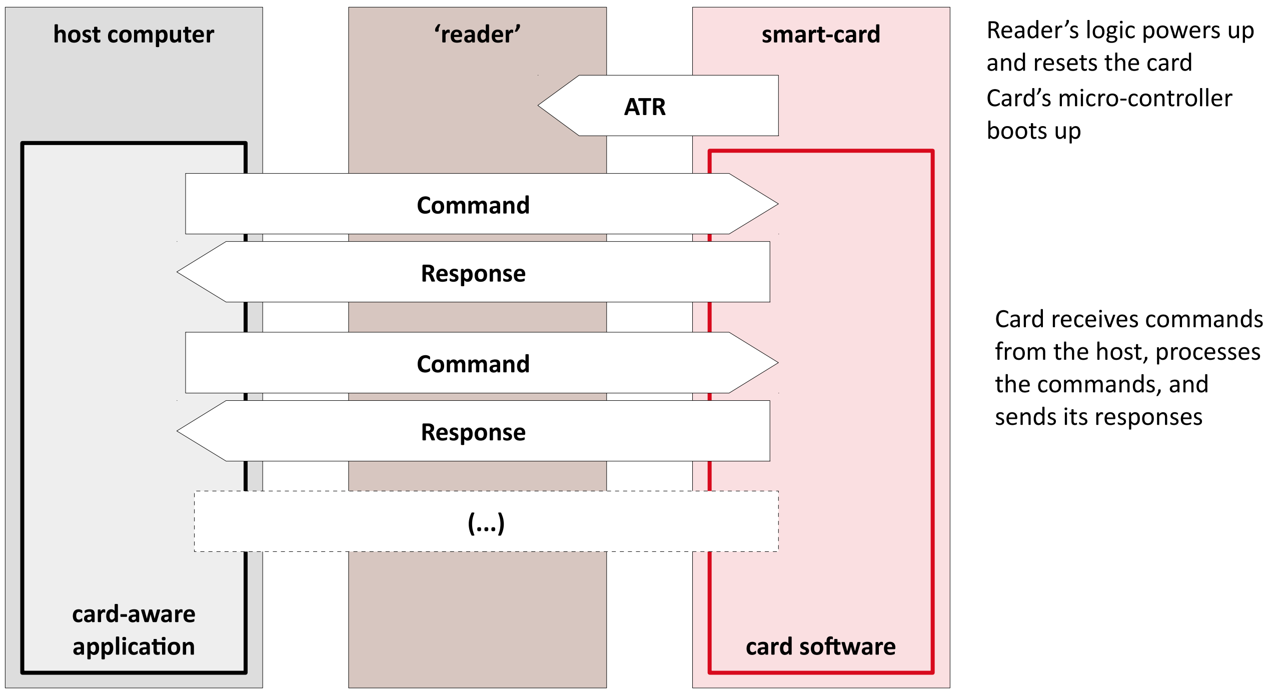

host computer
'reader'
smart-card
Reader's logic powers up and resets the card
Card's micro-controller boots up
ATR
Command
Response
Command
Response
(...)
Card receives commands from the host, processes the commands, and sends its responses
card-aware application
card software

Illustration 3: Principles of card operation:
power-up, ATR, and command/response sequences

The ATR contains all the technical information the reader needs to operate the card: the protocol(s) the card supports, the maximal bitrate and the expected timings. The ATR also conveys a few free bytes (max 15), named *Historical Bytes*, that the developer of the software running in the card's microcontroller may use to expose some information or meta-data.

In a way, the ATR is the fingerprint of the card family. The ATR allows a card-aware application (running in the host computer) to determine whether a card that has just been inserted in the reader is (possibly) the one card that the application was waiting for, or could be ignored.

Ludovic Rousseau, a smartcard professional and open-source enthusiast, has been establishing and maintaining for years a list of smartcard ATRs.

The complete list:

http://ludovic.rousseau.free.fr/softwares/pcsc-tools/smartcard_list.txt

He also provides an online tool to decode the ATR's technical data and query his list of ATRs:

https://smartcard-atr.appspot.com/

⚠ SpringCard is not connected with and does not sponsor or endorse 3rd party open-source developers.

2.2.3. Software (application) interface

The T=0 and T=1 transport protocols as standardised in ISO/IEC 7816-3 convey (or "transport") commands and responses between the card-aware software running in the host and the software running in the smartcard.

ISO/IEC 7816-4 defines both the grammar and the vocabulary of the commands and responses exchanged at application level, which are named *application protocol datagram units*, or APDU[3]. The grammar is how the commands and responses are formatted. The vocabulary is the list of commands that shall/should be supported, and also the values of the success/error status in the responses.

To clarify the difference between the transport protocol and the application-level grammar/vocabulary, let's have a look on HTTP, the application protocol behind the world-wide web.

The web browser (HTTP client) expects that the HTTP server answers with a numerical status code, followed by some text data. This is the grammar. Status code is defined to 200 for "OK, please display the following text nicely", and 404 for "Not found, the following text is an error message". This is the vocabulary.

The command and the responses are conveyed by an underlying transport protocol, TCP, that is agnostic about the grammar/vocabulary used at application level.

[3] The notion of APDU is not specific to the smartcard field. It has been introduced by network engineers in the early 70's and standardised by ISO 7498 "Information Processing – Open Systems Interconnection – Basic Reference Model", a standard that most network and system engineers know only as "the OSI model". In the OSI model, T=0 and T=1 are transport protocols (4th layer); they convey TPDUs. The applications (7th layer) do exchange APDUs. But in most situations, the developer of the application would simply say "command / response" instead of "C-APDU / R-APDU".

2.2.4. The grammar

2.2.4.1. Commands

ISO/IEC 7816-4 specifies that the Command APDU starts with a 4 byte header.

- 1st byte is named *class* or **CLA** and is intended to directly route the command to one of the card's applications (in case it has more than one), or to control a secure communication channel. CLA=$_h$00 in most situations.
- 2nd byte is named *instruction* or **INS** and identifies the command that has to be executed by the card's application. Next paragraph "vocabulary" lists the ISO-defined values for the INS byte.
- 3rd and 4th bytes are named *parameters* or **P1, P2**. They convey the parameter(s) to the instruction. It could be a combination of flags or a 16-bit number (for instance a record number or an offset inside a file), depending on the instruction.

After the header comes the optional **data** field. The length of the data is specified by the **L_C** field (length of command).

There are two kind of APDUs: short APDUs are limited to 254 B of data, with **L_C** on one byte. Extended APDUs go up to 64 kB of data, with **L_C** on three bytes (constant value $_h$00 then actual length of 2 bytes)[4].

The C-APDU is terminated by an optional **L_E** (length expected). It tells the card how many bytes the caller is waiting for. Symmetrically, **L_E** could be on 1 or 3 bytes.

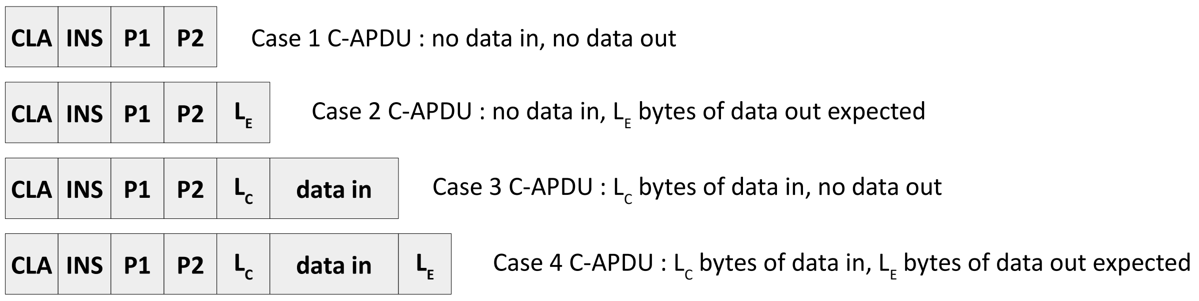

CLA
INS
P1
P2
Case 1 C-APDU : no data in, no data out
CLA
INS
P1
P2
L_E
Case 2 C-APDU : no data in, L_E bytes of data out expected
CLA
INS
P1
P2
L_C
data in
Case 3 C-APDU : L_C bytes of data in, no data out
CLA
INS
P1
P2
L_C
data in
L_E
Case 4 C-APDU : L_C bytes of data in, L_E bytes of data out expected

Illustration 4: Format of C-APDUs

[4] Most smartcards do not support, and do no need to support, the Extended APDUs, because they store and exchange only a limited volume of data. Extended APDUs have been introduced recently to fulfill the need of e-ID cards (including contactless passports), where fingerprints, pictures, certificate chains must be read as quickly as possible. A lot of readers still do not support Extended APDUs at all, or supports them by chaining TPDUs in the driver (that runs in the host computer), and this is dramatic in terms of speed. SpringCard H663 supports Extended APDUs directly into the reader, but with a limit of 8kB of data. The next generation of SpringCard readers will implement extended APDUs with no limit.

2.2.4.2. Responses

The Response APDU has no header and no length field. It conveys an optional **data** field; if this field is present, its length must be consistent with **L_E**. It is terminated by a two-byte *status word:* **SW1, SW2**.

The status word tells whether the instruction has been correctly executed, or not. Next paragraph "vocabulary" lists the ISO-defined values.

SW1	SW2	Status Word only, no data out

data out	SW1	SW2	Status Word and data out

Illustration 5: Format of R-APDUs

2.2.5. The vocabulary

2.2.5.1. Principles

According to ISO/IEC 7816-4, the card organizes its data in files. The files are in turn organized in directories. Security schemes (authentication, access control and read/write protection, secure communication) are generally implemented at directory level. The standard names a directory a *dedicated file* or DF.

Managing the security of the card's content at directory level, and not at the global card level, implies that an application may store (and protect) its data into a smartcard's directory, independently of any other application that will store its data into an other directory. This is the basis of so-called "**multi-application**" cards.

In a multi-application card, many service providers (bank, transport network, physical access control system…) share a single card, each provider having its own directory in the card.

The complexity of such a scheme lies on the top-level administration of the card: which of the service providers is responsible (and allowed!) to create (and maybe delete) the other's directories?

Most of the time, it is cheaper for every service provider to issue its own card than to cooperate with numerous partners in a multi-application card. Generally speaking, such a card also involves sharing sensitive data with competitors or far subcontractors. The same disincentives are also at work when it comes to emulating many smartcards by a single NFC mobile phone[5].

[5] Contactless smartcards, of course. But the principles are exactly the same. And the SIM, that actually hosts the card applications, is a contact card!

Under the directories or *dedicated files (DF)*, there are *elementary files* (EF). Smartcard's EFs are very different from regular computer files.

Firstly, they are not referred to by a file-name, but only by a short number (2 or even 1 byte), because a smartcard is a small, resource constraint, chip.

Secondly, EFs are "smart" files: the instruction to read / write a file does not map directly to "hey, please access this memory address and get / set this buffer there", but is a (possibly complex) set of operations executed by the card's software. This allows a few interesting features:

- **Record files** could be seen as a kind of SQL table. The card's software manages the record structure and controls finely the access: depending on the password or key that has been used for authentication, the granted access could either be read only, or read + insert, or read + update, etc. The card's software generally implements a *transaction* system with the record files (more about that in paragraph 2.2.5.2).
- **Cyclic files** are the same as record files, but when a new record is inserted, the oldest record is automatically overwritten. This is a must-have to implement a transaction log efficiently in a size-constraint file-system!
- **Value** or **Counter files** are single record-files that store a numeric value. They are associated to atomic, access-controlled operations (increase, decrease), all with boundary check.
- **Backup files** are more like the standard files of your computer, but mirrored and associated to a powerful *anti-tearing* mechanism (more about that in paragraph 2.2.5.3).
- **Standard files** do exists anyway; they are generally used to store large amount of data that don't change in the field.
- **Key files** and a few other **hidden files** store the secret keys or passwords, and all the meta-data used by the card's software to manage the access rights for the other files. These files are never readable, and special management instructions must be used to change their content.

2.2.5.2. Transactions

Some card software feature a *transaction* system. It isn't as complicated as a transactional SQL server, but implements the very same concept: maintaining the coherence of the data stored in the card.

For example, if the card implements an electronic wallet; the merchant's terminal must take the money from the wallet (decrease a value file) and write an event in the log (insert a record in a cyclic file). Without a transaction system, there's a risk that only one of the two operations is performed. But the transaction system ensures that either both operations are performed at once (*commit*), or they are both canceled (*rollback*).

A *secure transaction* system is at work in some cards and extends the principle one step further. It uses cryptographic techniques, generally a CMAC (*cryptographic-message authentication code*) computed other all the commands and data involved in the transaction. The CMAC is computed symmetrically in the smartcard and by the host's card-aware application[6]. The host application sends its commit instruction together with the CMAC, and the card compares this input with its own; the whole transaction is canceled if the authentication cryptograms don't match.

Doing so, even a defrauding software, running in the same host computer as the genuine card-aware application, will not be able to change the card's content, because the forged commands will not fit into the CMAC provided by the genuine application.

And finally, the card may send back another CMAC in answer, as a proof, for the back-end system, that the transaction has actually been performed.

2.2.5.3. Anti-tearing

The *anti-tearing* system is also an important feature to maintain the coherence of the data. If the card is teared away from the reader, or in the event of power loss, records or values could be partially erased or partially written, which makes the data invalid.

To prevent this dramatic situation to occur, some cards feature a dedicated anti-tearing hardware unit. Basically, it's a kind of *cache* memory, associated with a capacitor to accumulate the energy and a power-supply monitoring system.

At first the data is only written into the cache; then the card's logic ensures that there is enough energy in the capacitor to actually write all the data into the persistent memory (E2PROM or flash) even in case the external power-supply is lost. If this condition is met, the writing starts – and will eventually succeed. Otherwise, the existing data remain unaltered.

2.2.5.4. Instructions

Based on this file-system model, ISO/IEC 7816-4 defines about 40 commands.

Table 1 starting next page provides an overview of the most frequently used commands, with a few explanations.

[6] Possibly with the help of a dedicated smartcard, a SAM (*secure access module*), i.e. a ID-000 sized card whose unique role is to validate the transaction with the user's cards, suppressing the need to store the security keys in the host software, which would make them vulnerable to hacking. Nowadays, SAM cards could be replaced by *secure elements*, or HSM (hardware security module). More about that in paragraphs 2.3.1 and 2.3.2.

Command name	INS	Usage / Notes
SELECT	$_h$A4	The SELECT instruction is the "good at everything" instruction to navigate in the card's directory structure and select either directories or files. It has 3 different flavors, based on the flags in P1,P2: • *Selection by file identifier* is the basic form and uses 2-B IDs for most situations, or a 1-B short ID if the card supports it for a few files (to speed up the transaction), • *Selection by path* allows both to access a file with an absolute path, and to navigate among the tree (up one level, select first/next/previous file at current level, and so on), • *Selection by directory name* (or SELECT APPLICATION) gives the ability to select a directory using either a friendly, human-readable name, or a long, unique identifier assigned to the application in an open-loop, inter-operable context[7].
READ BINARY UPDATE BINARY	$_h$B0 $_h$D6	These are the 2 basic instructions to read/write into standard (and backup) files. P1,P2 specifies the offset inside the file. • For the READ BINARY instruction, L_E bytes are returned. • For the UPDATE BINARY instruction, L_C bytes are written (care must be taken that a few card technologies mandates a constant values for L_C, to match the memory's internal block size).
ERASE BINARY WRITE BINARY	$_h$0E $_h$D0	These 2 instructions are a legacy of the old memory technologies, where the memory cells or block can not be freely "updated", but only cleared and programmed again. The ERASE BINARY restore a portion of the file to the cleared memory state (either all 1s or all 0s depending on the underlying technology). The WRITE BINARY performs and exclusive OR or AND to change the value, but in one-way only.
READ RECORD(S) UPDATE RECORD APPEND RECORD	$_h$B2 $_h$DC $_h$E2	Read one or many records from a Record or Cyclic file. Change the content of an existing record. Insert a new record. On a Cyclic file, this overwrites the oldest record.
ERASE RECORD WRITE RECORD	$_h$0C $_h$D2	Restore one or many records to the cleared-memory state. Perform a logical OR or AND over the previous record value, in one-way only.

[7] Interoperable *Application IDentifiers* (AIDs) are covered by ISO/IEC 7816-5. Every card solution provider has its own *issuer identification number.* All the applications offered by this issuer uses this number as the first bytes of their AIDs. SpringCard's issuer identification number is $_h$A000000614.

Command name	INS	Usage / Notes
GET CHALLENGE	$_h$84	Generate a random number (challenge or nonce) and prepare the EXTERNAL AUTHENTICATE.
EXTERNAL AUTHENTICATE	$_h$82	Sends to the card the response to its challenge, computed using a secret key (or a password). This allows the card to verify that the "external world" (i.e. the application using the card) could be trusted (because it knows the expected secret), and to give read/write accesses accordingly.
INTERNAL AUTHENTICATE	$_h$88	Sends a challenge to the card, and retrieve the card's response, computed using a secret key. This allows the "external world" to verify that the card could be trusted (because it knows the expected secret).
GENERAL AUTHENTICATE	$_h$86	Alternative instruction for some compound authentication schemes. Note that ISO/IEC 7816-4 does not specify any cryptographic primitive for the authentication system, only the instruction to convey the authentication datagrams. The authentication could either be based on a symmetrical algorithm (such as AES or DES: both the card and the application using the card share the same secret key) or an asymmetrical algorithm (such as RSA or ECC: both the card and the application have their own secret, private key, and share only their public key with the other part)
GET DATA PUT DATA	$_h$CA $_h$DA	These commands give access to the meta-data of the card, directory or file.
GET RESPONSE	$_h$C0	This command is specific to the T=0 protocol. T=0 does not support the Case 4 APDUs (command + data in → OK + data out, so it is implemented as command + data in → OK, then GET RESPONSE → data out. Most of today's smartcard readers work at APDU level, and therefore perform the GET RESPONSE automatically when required.

Table 1: Excerpt of the ISO/IEC 7816-4 of INS values

It must be clearly stated that none of these commands is mandatory. The ISO standard says to the developer of the card's software "your select file instruction should be INS=$_h$A4". It does neither say "your select file instruction MUST be..." nor "you MUST provide a select file instruction". If the card developer wants to use a different INS for select file, he is allowed to do so. If the card has a single file and therefore does not implement the select instruction, this is allowed as well.

Therefore, there's no general way to "explore" a smartcard or to "read" its content. Without the card's specification, without knowing the actual list of instructions (and their parameters) that the card supports, without knowing the actual list of directories and files, and how are the data organised in them, there's nothing one can do reasonably with a smartcard.

More than that, smartcards are designed to protect their content by security systems (password protection or cryptographic authentication). If you do not know the key or the password, you will not access the data, period[8].

Do not start a smartcard project until you have the complete specification of the smartcard you will be communicating with. Even resist the temptation to quote a software development effort before gathering all the information you will need to complete the project.

Complete specification means: the detailed documentation of the card's instruction-set, the data model of the application, and the security model, and keys, to access them. The development team will also need to have tests cards. It is a good practice to use a different key-set in the test cards than in the final cards.

Unless you are a manufacturer of readers (like SpringCard is), you don't care of the card's hardware and low-level specification[9].

Just make sure that the card is supported by the reader at protocol level (ISO/IEC 7816-3 T=0 & T=1, or ISO/IEC 14443-4 "T=CL" for contactless cards), and everything will be fine.

[8] This introductory document could not go into the technical details on how the card protects its content. Only a few words on the subject: a secure card combines passive security (the memory is not a well ordered matrix, but a complex labyrinth where the bit-cell are dispatched here and there, so even an attacker with a physical access to the memory will not understand anything) and a lot of active countermeasures. Of course nothing is invulnerable, and a few old cards have been defeated by practical attacks, but this remain very rare. Most of the attacks remain purely theoretical, or at least too expensive to be reproduced out of a few well-equipped university labs.

[9] Some cards use an operating system, such as JavaCard. This makes a strong difference for the developer of the application in the card, of course, but for the developer of the host application, and even for the reader itself, whether the card runs a code written in Java or in C or in assembler remains undistinguishable (at least until it comes to benchmarking the transaction speed).

2.2.5.5. Status Words

It is mandatory for the card to return at least 2 bytes of Status Word (SW1,SW2) after every command. The first nibble of the Status Word (4 high-order bits of SW1) must be either 6 (error) or 9 (success). Any other value is a violation of the protocol.

Table 2 below provides a list of the most frequently used commands, with a few explanations.

SW1 SW2	Meaning
Correct execution	
h90 00 h9x xx	Success Success + vendor specific information
h61 xx	T=0 case 4 APDU only: xx bytes of data out are available, the reader shall issue a GET RESPONSE command with L_E=xx
Checking errors	
h6C xx	L_E not supported – repeat the same command with L_E=xx
h67 00	Wrong length L_C – the C-APDU is malformed
h6E 00	CLA not supported
h68 xx	Secure messaging / functions in CLA not supported
h6D 00	INS not supported
h6B 00 h6A xx	Wrong parameter P1, P2
h69 xx	Command not allowed
Other errors	
h6F 00 h6F xx	Generic error, no precise diagnostic Vendor specific error code

Table 2: Excerpt of the ISO/IEC 7816-4 listing of status words

2.3. Variations around ISO/IEC 7816

2.3.1. SAM and HSM

A *secure authentication module* (SAM) is a ID-000 sized smartcard that plays only a single role: store secret keys, and compute cryptograms. As all the other smartcards, a SAM is carefully designed to protect its secrets against all known attacks (and, ideally, against most future attacks).

Some transportation networks install a SAM at every gate to authenticate the user's (contactless) smartcard very quickly; the gate's performance is independent from the latency of the network, and, more than that, the gate keeps working even in the event when the network is down. Some electronic purse (e-payment) systems also use a SAM the merchant terminals for the same reasons.

A *hardware security module* (HSM) is a network appliance that provides the same features and the same security level as a SAM card, but with a dramatically higher throughput.

HSMs are typically used to authenticate the secure transactions (i.e. compute a CMAC, see 2.2.5.2) in large systems. They are also widely used in *public key infrastructures* (PKI) or to implement the secure communication layer (TLS, HTTPS) in performance-critical servers or VPNs.

2.3.2. Secure elements and other "smartcard chips without card"

The last years have seen a fast development of complex, interconnected networks of high-end technology artifacts, which can not be named "computers" anymore because they do not have a screen or a keyboard. Most of them don't even have a user!

Smart-watches and other wearables, sensors connected to the Internet from your own or from the streets of the *smart-city* where you live, counters and actuators of the *smart-grid*, the *IoT* (Internet of Things) is built on top of secure, trusted, machine-to-machine communication. Even the cheapest node in such a complex, interconnected system, must have its own secret keys and be tamper-proof; otherwise, the whole system is vulnerable to data leakage, denial or service attacks, injection of counterfeit data, and to many other unpleasant interactions.

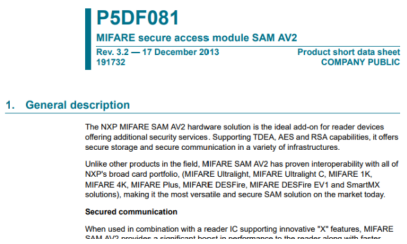
The smartcard manufacturers have a strong experience in tamper-proof silicons and secure software designs, which makes them key actors of the IoT market. All they have to do is offering their smartcard chips in the form of SMD components, evading the ISO-specified form-factor and contacts.

Such chips are named "secure companion chip", "trusted platform coprocessor" or more commonly "secure element".

A good definition of a **secure element** is: “a smart chip that securely stores and manage information”.

Since most of them are developed by smartcards vendors and/or designed to behave the same way as a smartcard, they generally use ISO/IEC 7816-4 as software interface.

The NXP A70CM is a SMD component, based on a secure MCU and running a JavaCard OS.
It is compliant with ISO/IEC 7816-4 but uses a custom version of T=1 on top of a classical I2C bus.
(image NXP)



P5DF081

MIFARE secure access module SAM AV2

Rev. 3.2 — 17 December 2013 Product short data sheet
191732 COMPANY PUBLIC

1. General description

The NXP MIFARE SAM AV2 hardware solution is the ideal add-on for reader devices offering additional security services. Supporting TDEA, AES and RSA capabilities, it offers secure storage and secure communication in a variety of infrastructures.

Unlike other products in the field, MIFARE SAM AV2 has proven interoperability with all of NXP's broad card portfolio, (MIFARE Ultralight, MIFARE Ultralight C, MIFARE 1K, MIFARE 4K, MIFARE Plus, MIFARE DESFire, MIFARE DESFire EV1 and SmartMX solutions), making it the most versatile and secure SAM solution on the market today.

Secured communication

When used in combination with a reader IC supporting innovative "X" features, MIFARE SAM AV2 provides a significant boost in performance to the reader along with faster

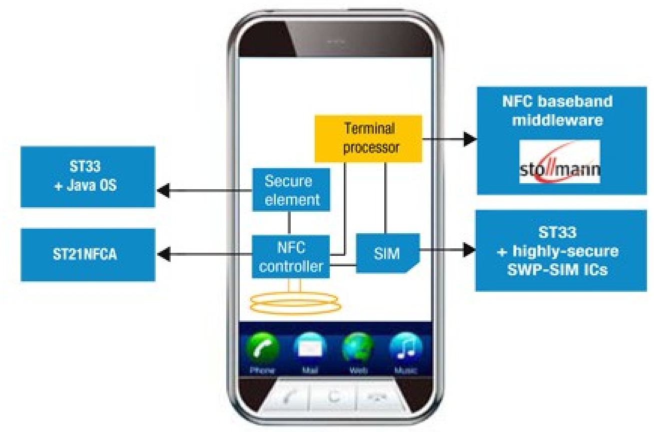
The NXP AV2 is a secure access module (SAM), offered either as a ID-000 SIM/SAM smartcard or a SMD chip.
It is compliant with ISO/IEC 7816-4 and uses ISO/IEC 7816-3 T=1 as transport protocol.
(image NXP)

Embedded in the form of a USB stick, they are the basis of strong authentication tokens.

Gemalto SafeNet is a family of tokens to remotely access a critical system or a private network. Basically, it is a secure chip, in the form of a smartcard, or a smartcard with its own USB reader, and possibly a small LCD as user interface.
(image Gemalto)

The smartphone market is also pushing for secure elements to store sensitive data. For example, when you pay with Apple Pay or Google Wallet, the payment application that emulates a credit or debit card runs in a secure element. But the same secure element may also host a transit network ticket, your company’s access badge, the key to your hotel room…

In a way, the secure element of a NFC smartphone allows to *virtualize* many smartcards; the smartphone's processor (and the network behind it) uses it through a classical contact interface, and the external world through a NFC, contactless interface. This pushes the "multi-application card" concept to a next level!



A smartphone featuring a SE and a SIM.
(image ST)

2.3.3. Wired-logic, storage only card

A wired-logic, storage only card, frequently called a **synchronous card**, is nothing more than a E2PROM memory which is embedded in an **ID-1** plastic card, and can be accessed through physical contacts located at the position defined by ISO/IEC 7816-2. But since the card does not embed a microcontroller, it does not implement neither the T=0 or T=1 protocols of ISO/IEC 7816-3, nor the command/response grammar and vocabulary of ISO/IEC 7816-4.

There are three major kinds of synchronous cards, based on the technology of the memory chip they use:

- I_2C ("S=8"),
- SPI with separated MOSI/MISO lines (3 wires or "S=9"),
- SPI with a shared I/O line (2 wires or "S=10").

Unfortunately, there are also plenty of proprietary variations, and it has became almost impossible for a reader to detect what type of card has been inserted, without being manually instructed of the protocol to use[10].

[10] To be complete, we must mention ISO/IEC 7816-10 "Electronic signals and answer to reset for synchronous cards" (1999) that is an effort to ensure interoperability of wired-logic storage cards. Unfortunately, most of them have been designed and issued before the release of the standard. The largest part of the card on the field today are still not compliant with any standard.

Most today’s PC/SC smartcard readers do not support synchronous contact cards[11].

Notably, SpringCard products do not (unless they are flashed with a customer-specific firmware).

2.3.4. Contactless cards

A *contactless smartcard* is a smartcard that uses inductive coupling to communicate with its “reader”.

Inductive coupling is the technology behind NFC (Near Field Communication) and short-range RFID (Radio Frequency Identification) in the 13.56 MHz (HF) radio band.

“Proximity” contactless smartcards are defined by the ISO/IEC 14443 set of international standards. The ISO/IEC 14443 is divided into 4 layers, the upper layer (ISO/IEC 14443-4) specifies the **contactless transport protocol**, sometimes referred to as “T=CL”. The application layer that comes on top of the contactless protocol is supposed to be the very same ISO/IEC 7816-4 that comes on top of the T=0 or T=1 transport protocols.

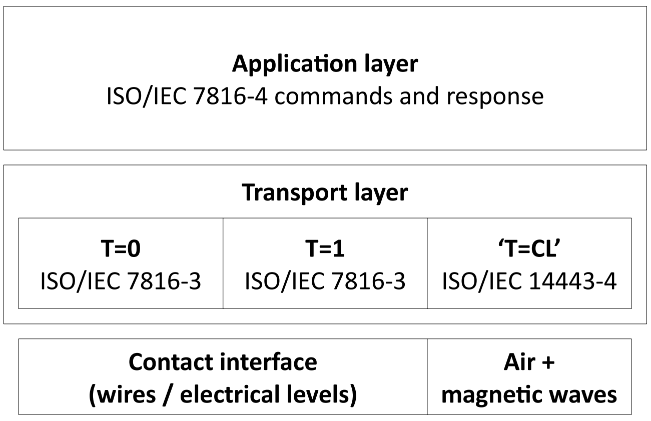


Illustration 6: The contact and contactless protocol stacks

Contactless smartcards are discussed in detail in chapter 3.1.

[11] CT-API (CardTerminal API) is an alternative to PC/SC that focuses on synchronous cards. A few legacy readers are still available on the market, supporting this API.

2.3.5. Wired-logic, storage only contactless cards

A lot of contactless cards on the market today are not “smartcards” but wired-logic memories with an RF interface. The marketing gives them different names: RFID labels, NFC tags, contactless tickets… but they are all based on the same physical principle: inductive coupling at 13.56MHz, and are all (more or less) compliant with either ISO/IEC 14443 standard for proximity cards or ISO/IEC 15693 standard for vicinity (hand free) cards.

Wired-logic, storage only contactless cards are discussed in detail in chapter 3.2.

2.4. The coupling device or coupler

Now that we know that a smartcard is a basically a (secure) microcontroller communicating with the external world through a (sort of) serial line, we must find a better name than “reader” for the device that ensures the connectivity and gives access to the smartcard instructions.

Even if the term *smartcard reader* has been popularized by the PC/SC and CCID specifications, it does not reflect the technical reality, because a card’s instruction set opens lots more feature than just “reading” some data.

“Reading” a smartcard makes not more sense than “reading” a remote web server, or “reading” a SQL database; you do not “read” a raw memory; your application sends some commands to get authenticated, sends other commands to explore a file system, sends commands to insert / update / delete the data… and read it, of course, but not only.

Therefore, the device in which the smartcard is inserted plays the role of a pass-through gateway between a software running in the host computer (or host terminal) and the software running inside the card’s microcontroller.

This pass-through gateway translates commands coming from the host application through either USB, Serial, Ethernet, or any other computer interconnection technology, into electrical signals that are compliant with the asynchronous, half-duplex serial T=0 or T=1 transport protocol. But this gateway does not add any processing logic; its position is to couple the smartcard with the computer.

As a consequence, the standard name, as defined by ISO/IEC 7816, is *coupling device* (CD) or simply *coupler*.

Contactless couplers are in turn named *proximity coupling device* (PCD) by ISO/IEC 14443 or *vicinity coupling device* (VCD) by ISO/IEC 15693.

3. Contactless cards, RFID, NFC – concepts and standards

3.1. ‘Proximity’ contactless smartcards

3.1.1. Basics

A *contactless smartcard* is a smartcard that uses inductive coupling to communicate with its coupler. Inductive coupling is the principle behind electric transformers: a primary coil is powered by a AC voltage and creates a magnetic field. The magnetic field induces a current in the secondary coil.

In the case of contactless cards, the primary coil is driven by a sinusoidal voltage at 13.56 MHz, and hence creates a magnetic field with a 13.56 MHz carrier frequency (HF radio band). This magnetic field is not constrained in a confined volume, as it would be the case in an actual transformer. Instead, the field floods virtually freely within a part of the open air, limited only by the directivity of the coil and the (fast) decrease of the magnetic waves.

This coil and its driving circuit, forming the primary of a virtual transformer, is named the *Proximity Coupling Device* (PCD).

When a (mobile) secondary coil is moved into the part of the space where the PCD’s field floods – say, comes in *proximity* to the primary circuit *(illustration 7)* – the transformer, once virtual, now becomes real. The flow of the magnetic field through the secondary coil provides electrical power to a secondary, passive, circuit.

As you may have guessed, this mobile part of the transformer, made of a coil and of an electronic circuit (or a chip) is the *Proximity Integrated Circuit Card* (PICC).

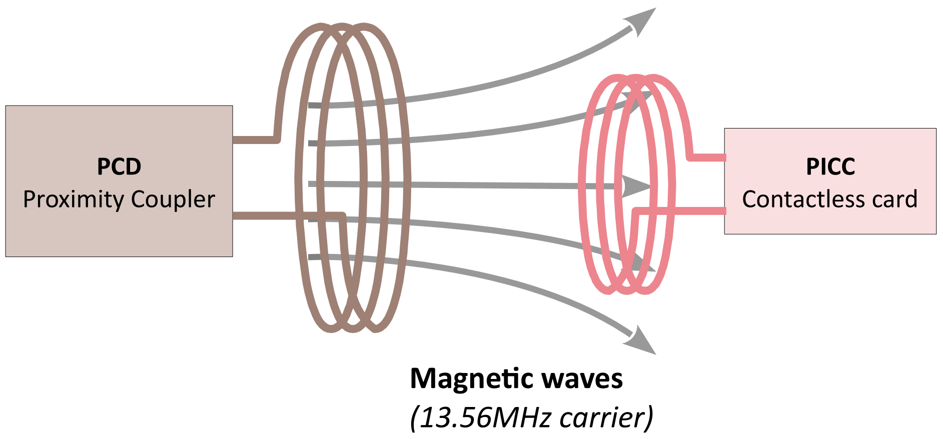


Illustration 7: PCD → PICC remote power principle

The magnetic field emitted by the coupler not only provides a power supply to the card, but is also suitable to convey data in both directions *(illustration 8)*:

- The coupler modulates its carrier to transmit information to the card;
- The card uses load-modulation to answer.

Load-modulation means that the card is able to vary its impedance, i.e. the *load* it represents for the electrical circuits.

According to Ohm’s low, under a fixed voltage, any variation of the impedance causes a variation in the current. And, as this is the case for any transformer, a variation of the current in the secondary circuit is also noticeable from the primary circuit – this is how the PCD “sees” the PICC’s answer.

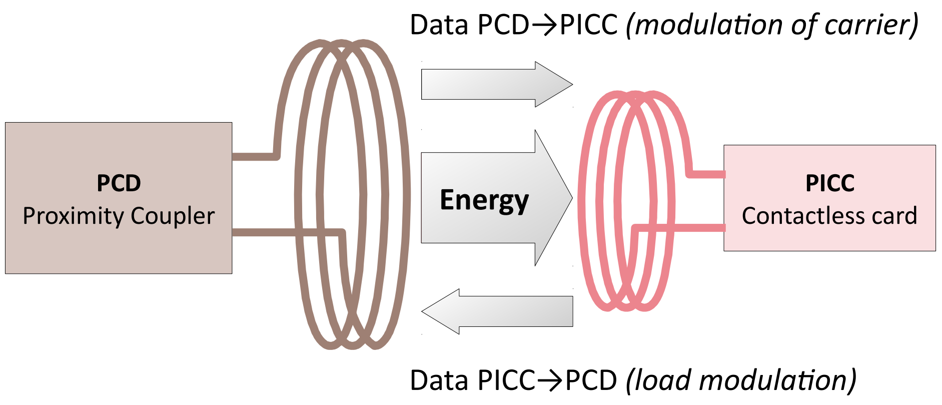

Data PCD→PICC (modulation of carrier)
PCD
Proximity Coupler
Energy
PICC
Contactless card
Data PICC→PCD (load modulation)

Illustration 8: How the card and the coupler communicates

This principle of operation could be summarized as being *near-field, passive RFID*:

- For electromagnetic scientists, *near field* is the part of the space where the distance to the emitter's antenna is lots smaller than the wave-length, which is always the case with magnetic waves[12],
- *Passive RFID*, because the contactless card does not have an emitter. The card is only allowed to alter the incoming wave, using load-modulation[13].

A 125 kHz or 135 kHz reader/transponder system is in the same *near-field, passive RFID family.*

The key difference is the frequency band (LF *vs* HF) which lead to faster bitrates for 13.56 MHz systems: 106 kbit/s to 848 kbit/s (ISO/IEC 14443) *vs* 2 to 15 kbit/s at 125/135 kHz.

3.1.2. The standards for proximity cards

The standard for **proximity cards** and couplers (PICCs and PCDs) is ISO/IEC 14443, which is divided into 4 layers:

- ISO/IEC 14443-1 covers the physical aspects (most of them are detailed in paragraph 3.3.3).
- ISO/IEC 14443-2 defines the field level and bit-level modulation.
- ISO/IEC 14443-3 tells how the bits are assembled in bytes and then in frames. There are two options: ISO/IEC 14443 type A uses *On/Off Keying modulation* (OOK) and Manchester coding, ISO/IEC 14443 type B uses a 10% *Amplitude Shift Keying modulation* (ASK) and NRZ-L coding[14]. The coupler (PCD) shall implement both types, but the card (PICC) may choose to implement only one of them[15].
- ISO/IEC 14443-4 tops both types with a block-oriented transport protocol, not far from the T=1 protocol of ISO/IEC 7816-3. This protocol is often referred to as "T=CL", short for *Transport = Contact-Less*, although this name does not appear officially in the standards.

[12] At 13.56 MHz, the wave-length is 22m. That gives a limit near field/far field > 3m. Due to EMC rules, at this distance, the magnetic field is already too low to be used.

[13] *Passive* means that the card does not emit radio wave (only the PCD is active). But this does not prevent a card from having its own power supply. Typically, a mobile phone running in card emulation mode remains a *passive* device, even if it would have been unable to achieve a transaction without the battery to power its CPU and SIM card.

[14] ISO/IEC 14443-A comes from the work of Philips Semiconductor (NXP) under the MIFARE brand name; the history is roughly summarized in 3.6.1. ISO/IEC 14443-B is a fork of the Innovatron radio protocol, developed by Roland Moreno / RATP / SNCF for the public transport market (Paris' NAVIGO card).

[15] And this is actually the case in 100% of the 'true' PICC. Only NFC objects running in card emulation mode, such as some mobile phones, are likely to implement both types in PICC mode.

Vicinity cards, and the difference between Proximity and Vicinity, are the subject of paragraph 3.4.

3.1.3. Polling

In a contact coupler, a physical switch tells the coupler when a card has been inserted into position and is ready for operation (at least *may be ready* – it could be a dumb plastic card, or it could have been inserted upside down…). But a contactless coupler has no equivalent way to be notified when a card arrives or leaves. There is no choice for the contactless coupler but to broadcast repeatedly a message like "is there someone here?", in the hope that a card would be near enough to answer.

A PCD supporting both types A & B of ISO/IEC 14443 broadcasts *REQA* (Request A) or *WUPA* (WakeUp A) and waits for a few milliseconds, in the hope that a type A card may answer in the opened time-window. If there is no answer, the PCD broadcasts *REQB* (Request B) or *WUPB* (WakeUp B), and opens another time-window. If there is still no answer, the PCD tries again with type A, and so on.

This endless process is called the coupler's *polling loop.* In a multi-technology coupler, other protocols (ISO/IEC 15693, FeliCa, proprietary...) may also be part of the loop.

When a card is in position for communication, the card answers to either *REQA*/*WUPA* or *REQB*/*WUPB* by *ATQA* (Answer To Query A) or *ATQB* (Answer To Query B) during the time-window opened by the coupler. The coupler then reports the presence of the card to the host computer, and an application running in the host computer may start using the card.

3.1.4. Anticollision

Now let’s consider the case of two or more contactless cards entering the PCD’s RF field in the same time.

If all the cards use different protocols, the coupler “sees” one answer to every request frame. But if at least two cards use the same protocol, their answers collide in the same time-window. The coupler is then unable to discriminate among the two answers. The PCD must then run a collision-resolution algorithm – the *anticollision loop* – to enumerate all the cards.

ISO/IEC 14443 type A features a *deterministic anticollision method*, which means that all cards could be enumerated in a deterministic, reliable sequence.

ISO/IEC 14443 type B features a *probabilistic anticollision method*, where the cards generate random numbers to answer in random time-slots. This method is globally slower, and can not guarantee that all cards will eventually be enumerated.

3.1.5. Single card approach

But is it really interesting for a PCD to enumerate all the nearby PICCs? In most real-world applications, the answer is simply no.

Of course, when the PCD is part on a warehouse inventory system, or when there is by design a high probability of collisions (this is for instance the case for electronic passports with added electronic visa labels), there is no choice but to enumerate all the PICCs present, and let the top-level application decide the one(s) it wants to process.

On the other hand, a *point of sale* terminal (POS) must be able to know precisely *who* is willing to pay for a service or a good. If you put in front of the POS’ coupler a wallet containing two debit or credit cards, the POS has no way to decide which one should be charged. Therefore, the POS cancels the transaction and prompts you to place one card, and only one, in front of its antenna. Therefore, the POS coupler does not have to implement any kind of anticollision – all it has to do is reporting that a collision has occurred.

And the same applies for a PC/SC contactless coupler. PC/SC has been primarily designed for contact smartcards. There is no chance that two smartcards could ever be introduced – and powered – in a single contact slot. Therefore, the PC/SC infrastructure relies on the paradigm “one coupler = one card”.

A PC/SC-compliant PCD may know that there are two (or more) cards there, but the PC/SC middleware and the host applications ‘sees’ only one card at once.

3.1.6. Transport and application protocols

3.1.6.1. Full ISO stack

From the point of view of a software developer, contactless smartcards are, like their contact counterpart, nothing more than a microcontroller exposing its services through a command/response scheme.

ISO/IEC 14443-4 "T=CL" makes no assumption on the format of commands and responses that could be exchanged by the top-level application (see paragraph 2.3.4). Anyway, keeping in touch with the well-established standards, even at application level, is always better for interoperability.

Therefore, virtually all high-end contactless smartcards use ISO/IEC 7816-4 APDUs at application level. This is specially true for payment, eID, public transport cards, and more, where the same card software could run indifferently in a contact or a contactless smartcards.

In most situations, the developer of the application does not even have to care whether the card and the coupler use T=CL, T=0 or T=1 to communicate together – he just sends ISO/IEC 7816-4 commands and process ISO/IEC 7816-4 responses, with a complete abstraction of the underlying protocol.

3.1.6.2. Vendor-specific command sets

There's also a market for lower-end contactless smartcards that use a proprietary command/response set on top of T=CL.

The motivation could be either the price of the chip – a proprietary command set may be chosen to have the smallest memory footprint – or the legacy of an earlier technology, or both.

An example of the legacy of an earlier technology is NXP MIFARE Plus, a modern card with a microcontroller core and a high security level (based on AES). The card has been introduced in 2009 to supersede the popular MIFARE Classic wired-logic card (dated 1994)[16]. Since the new card aims to replace a previous one, it must be easy for the implementers to take the move while preserving most of his existing applications and products.

Therefore, the new MIFARE Plus chip uses a proprietary function set that is derived in straight line from the one of the MIFARE Classic chip, the novelty being limited to the use of T=CL blocks (ISO/IEC 14443-4) to convey the commands/responses, instead of relying on 'raw' ISO/IEC 14443-3 frames as earlier.

Between the two ends, NXP DESFire is the example of a contactless smartcard born with a proprietary command set only (2002, FW version 0.4) that has quickly evolved to convey the proprietary commands embedded into ISO/IEC 7816-4 compliant

[16] The MIFARE Classic has been deprecated following the disclosure of numerous practical attacks against its security scheme. More about than in paragraph 3.6.1.2.

messages (DESFire EV0, 2004, FW version 0.6) before eventually plunging into the ISO/IEC 7816-4 command set (DESFire EV1, 2008), at least partially[17].

3.1.7. Contactless smartcards and PC/SC

The ISO/IEC 14443-4 "T=CL" protocol is very close to the T=1 contact protocol, so the contactless smartcards are operated as if they were T=1 cards. It makes no difference from the application's point of view.

In PC/SC, the card must also have an ATR. The PC/SC standard therefore specifies how the contactless coupler build a virtual ATR from the contactless card's ISO/IEC 14443-4 meta-data. This is detailed in paragraph 6.4.1.

3.1.8. Contactless only, dual, two-chip cards

Most contactless cards are 'contactless only', meaning that the chip's only interface is its antenna and a RF 'modem'. Some cards feature both contact and contactless interfaces.

This is typically the case of corporate badges: on the one hand, the access-control readers at the entrance of the building or rooms use the card's contactless interface. A private e-purse may also be handled this way by the ATMs or at the company's restaurant. On the other hand, the IT-related services – session opening on the desktop or laptop, encryption of the hard disks, single-sign-on (SSO) on the Intranet or corporate applications, digital signature – use the card through its contact interface.

Most credit or debit cards delivered today by the banking industry also feature two interfaces: contactless payment is enabled for low amounts and does not require the user's PIN; for larger amounts, the card (or the terminal) enforces contact operation and PIN entry.

But there's a slight difference here: in the case of a credit or a debit card, a single chip provides the same service (payment) through the two interfaces. In the case of the corporate badge, there is no evidence that the contactless world (access control, e-purse) and the contact world (IT) have any data to share.

More than that, opening security-sensitive corporate processes (like digital signature or SSO), through a contactless interface, could be seen as a potential security breach. As a consequence, a lot of contact+contactless cards are not actually based on *dual-access* chips, but embed two chips (one for contact, one for contactless) that have nothing in common.

[17] DESFire EV1 provides ISO/IEC 7816-4 commands to handle the data (SELECT, READ, UPDATE) but not much more. Formatting the card or using efficiently its various security features implies going back to the proprietary command set. The new DESFire EV2 (2016) goes one step further with new ISO/IEC 7816-4 commands added.

Illustration 9: A first-generation's dual-access card, featuring a copper coil welded behind the ISO/IEC 7816 module

3.2. Wired-logic proximity contactless cards

As this is the case in the contact world with synchronous cards, there are plenty of contactless cards that does not embed a processor. They could be seen as a raw memory storage, with some kind of digital-logic (or an ASIC) exposing a very limited set of functions to read / write data, and maybe to offer basic security features.

3.2.1. Support of wired-logic cards by standard PCDs

In the world of contact cards, the standards have appeared (long) after most synchronous cards have been designed – and released.

But in the world of contactless cards and RFID, the ISO/IEC 14443 standard has been written early in the process, and is, by design, very close to the proprietary system that was leader of the market at the time (MIFARE)[18].

As a consequence, most wired-logic contactless cards available today rely on ISO/IEC 14443-3 (or at least ISO/IEC 14443-2), and it's not a challenge for a PCD that is compliant with the standard to support them as well.

There are 2 noticeable exceptions: 2 companies that had already developed their own contactless protocols before the adoption of ISO/IEC 14443 in 2001, and that didn't take the move to the standard protocol since then.

The 1st of them is Sony; the Sony FeliCa family of cards is still using a Japanese protocol (JIS X 6319-4) that has been declined by the ISO/IEC 14443 committee. But Sony has managed to push this protocol into NFCIP-1 (ISO/IEC 18092) in 2003, and

[18] For more information regarding the MIFARE family, read 3.6.1.

to have the FeliCa cards adopted by NFC Forum as Type 3 Tags. Therefore, this protocol is now supported by most mainstream couplers, at least partially.

The 2nd is LEGIC. The LEGIC Prime is a family of cards that rely on a proprietary, undisclosed protocol. But at last, the new generation of cards (LEGIC Advant) is going full ISO.

SpringCard contactless couplers support a large number of wired-logic chips. Anyway, some hardware are (by design) unable to support some of them, and old firmware versions may provide only a small part of the features offered by an up-to-date firmware. Always refer to each product's datasheet to know for sure which technologies are actually supported by the device you intend to use.

3.2.2. Support of wired-logic contactless cards under PC/SC

The PC/SC standard specifies 4 commands, taken from the ISO/IEC 7816-4 standard, to work with wired-logic contactless cards.

These commands are not forwarded to the card, since a wired-logic card would not support them. They are processed, or interpreted, by the coupler's microcontroller, that is responsible for translating them into the equivalent commands in the card's specific function set and protocol.

The 4 commands are:

- GET DATA (INS=$_h$CA) is used to retrieve the card's protocol-level identifier (UID/PUPI/serial number/random ID),
- READ BINARY (INS=$_h$B0) to read a data block,
- UPDATE BINARY (INS=$_h$D6) to write a data block,
- GENERAL AUTHENTICATE (INS=$_h$86) to get authenticated onto the card, if the authentication algorithm and keys are implemented in the coupler itself, not in the host application. This is only the case for the CRYPTO1 algorithm of MIFARE Classic[19].

These commands, and the other commands exposed by the embedded APDU Processor, are documented in paragraph 6.6.

In PC/SC every card must also have an ATR. The PC/SC standard therefore specifies how the contactless coupler build a virtual ATR from the wired-logic contactless cards protocol information. The ATR's historical bytes exposes the card's technical data (active protocol, card family). This is detailed in paragraph 6.4.1.

[19] CRYPTO1 is a proprietary algorithm that is embedded only in NXP reader chips (and cards, of course). Therefore the host application is not able to implement the authentication itself, as it would be the case for other card technologies, and must rely on the reader's internal logic to do the job, hence the GENERAL AUTHENTICATE instruction.

3.3. Operating distance: size matters

3.3.1. Decrease of the RF field with the distance to the PCD

Illustration 10 shows how the RF field decreases with the distance to the coupler's antenna, for different antenna radius, with the same RF field level at the origin. Spoiler: the smaller the antenna, the faster the decrease.

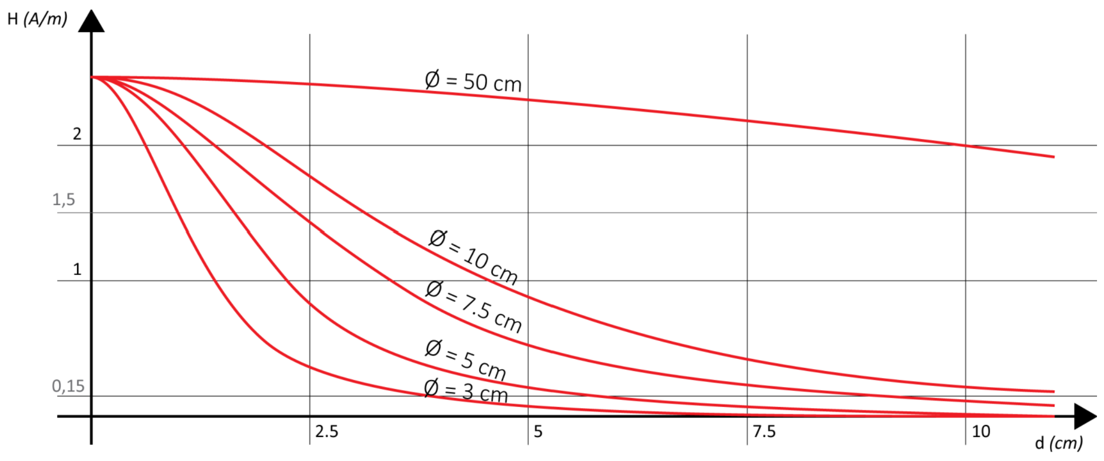


Illustration 10: RF field level against distance to the antenna, with the same H_0, for different diameters

In this illustration, the field level at the center (H_0) of the antenna is fixed (2.5A/m) whatever the diameter, and the field level is evaluated along the axis. This is not a practical situation, because there's also are two strong links between H_0 and diameter of the antenna.

The first link is purely technical: the H_0 is a function of the current in the antenna's coil:

$$H_0 = \frac{N\,I}{\varnothing}$$

where ***N*** is the number of loops of the coil, and *I* the current in the coil, and **Ø** its diameter.

For a given reader architecture, ***I*** is limited to the capabilities of the antenna driver IC. Therefore, it is not possible to increase ***I*** freely, and, as a consequence, a larger diameter **Ø** leads to a smaller field level at the center ($\boldsymbol{H_0}$).

The second link is related to the *electromagnetic compatibility* (EMC) mandatory limits[20]. It is forbidden by law to exceed a certain field level. The measure is taken at 10 m of the antenna.

[20] Europe: ETSI 300-330 / US: FCC 47 part 45.

As a consequence, an antenna with a "flat" field curve (**Ø = 50 cm** in illustration 10) must have a very small ***H_0***, to stay in the allowed limits.

Illustration 11 shows the RF field curves for "practical" HF readers.

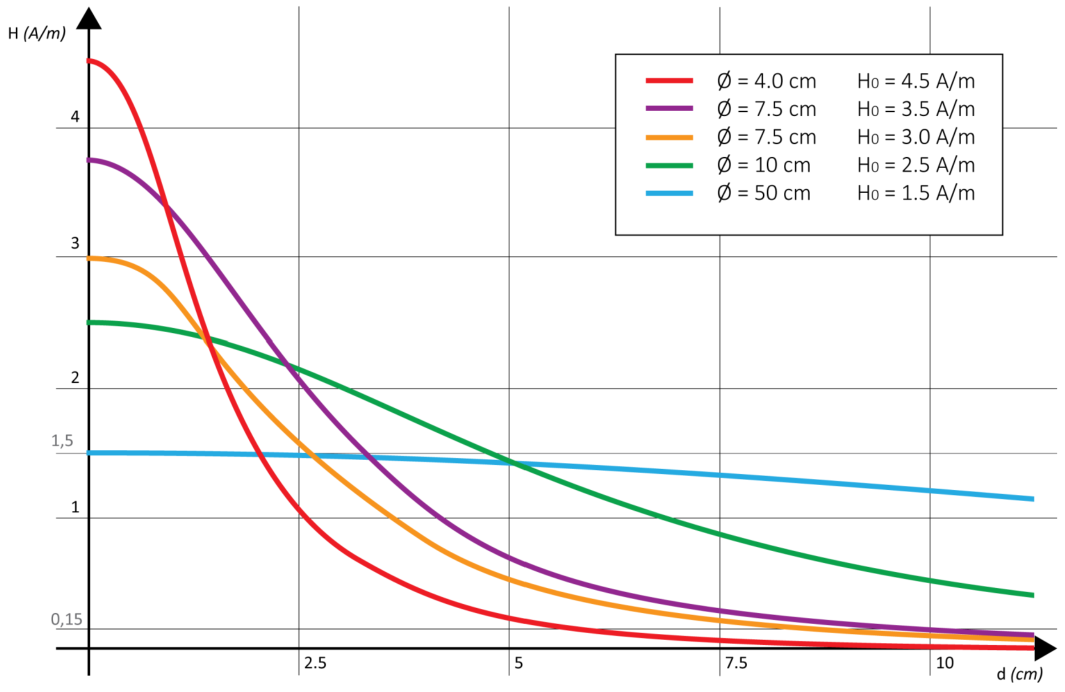


Illustration 11: RF field level against distance to the antenna, for different actual readers

3.3.2. Field level required by the PICC

Of course, a PICC needs a decent RF level to receive enough energy to be powered. The RF level is measured in amperes per meter (A/m); ISO/IEC 14443-2 states that the PICC shall be able to operate at 1.5 A/m. Most PICCs are even able to operate at weakest field levels, but no assumption should be made in general case. This is the first limiting factor in a PCD/PICC system.

The 1.5 A/m limit is shown in gray on illustrations 10 and 11.

This gives a basic estimation of the operating range of a given contactless reader: the reader associated to the purple curve in illustration 11 (Ø = 7.5 cm, H_0 = 3.5 A/m) will provide enough power to all standard-compliant PICCs up to 3 cm. Most today's wired-logic PICCs are able to work on a field as low as 0.5A/m, increasing the actual range up to 6 cm with the same device.

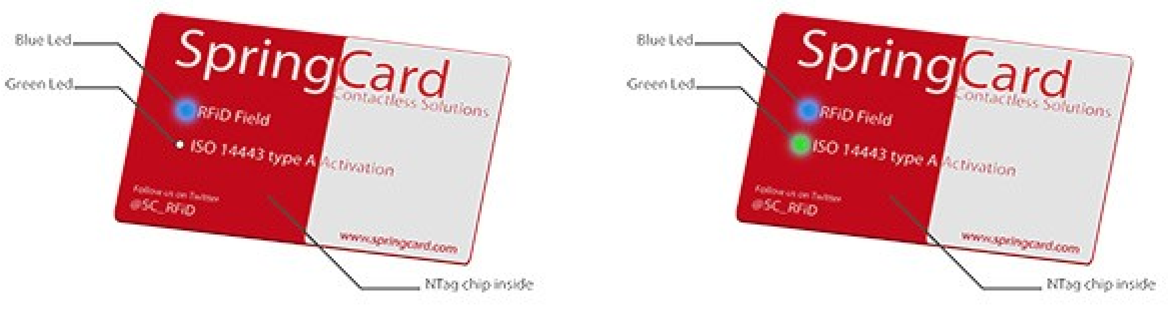


Illustration 12: The SpringField Florida is a small testing tool, based on a NXP NTAG chip, that shows how far the RF field is strong enough to power a contactless card.

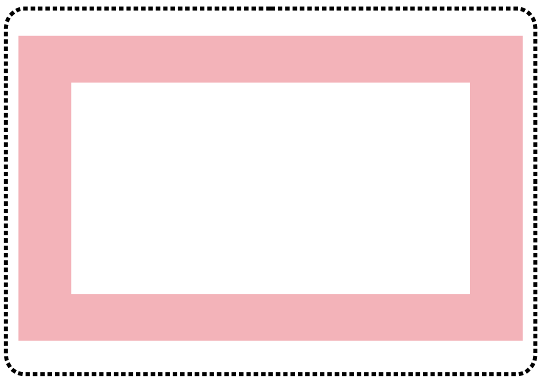
3.3.3. Size of the PICC

The size of the PICC is a second limiting factor. As this is the case with any electrical transformer, the coupling ratio is directly tied to the relative size of both antennas. When both antennas have exactly the same size, the flow of the RF field could be maximized. But when one of the antennas is lots smaller than the other, there's a chance that the coupling factor between both circuits becomes two weak for proper operation.

To promote interoperability, ISO/IEC 14443-1 defines 6 classes of contactless antennas, depicted in illustrations 13 and 14. The standard strongly advices that all PICCs conform to one of these classes. It also states that the PICC's antenna shall in no case exceed 86×54 mm.

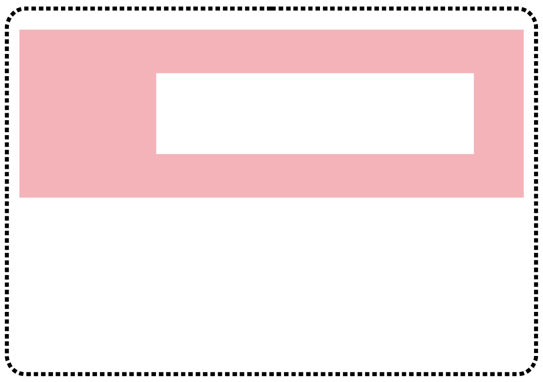
.../...

In the illustrations below and continuing next page, the dashed line shows the size of a ID-1 card (drawings are not to scale). The antenna must fit totally inside the colored areas.



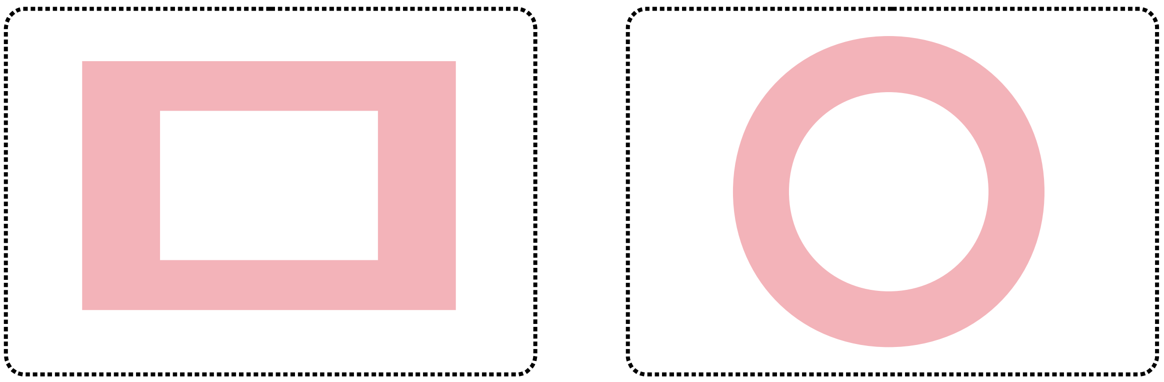
Class 1

The antenna occupies the ID-1 surface at best.



Class 2

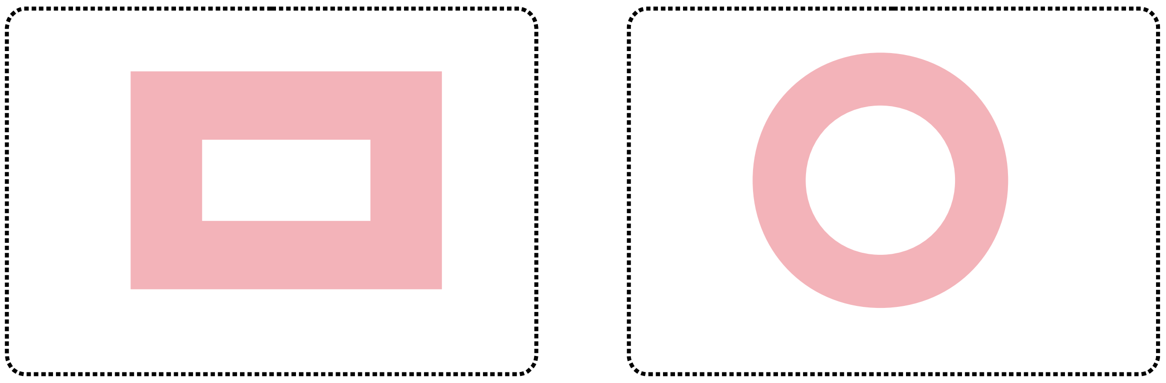
The spare area on the bottom is where a credit card is embossed.



Class 3

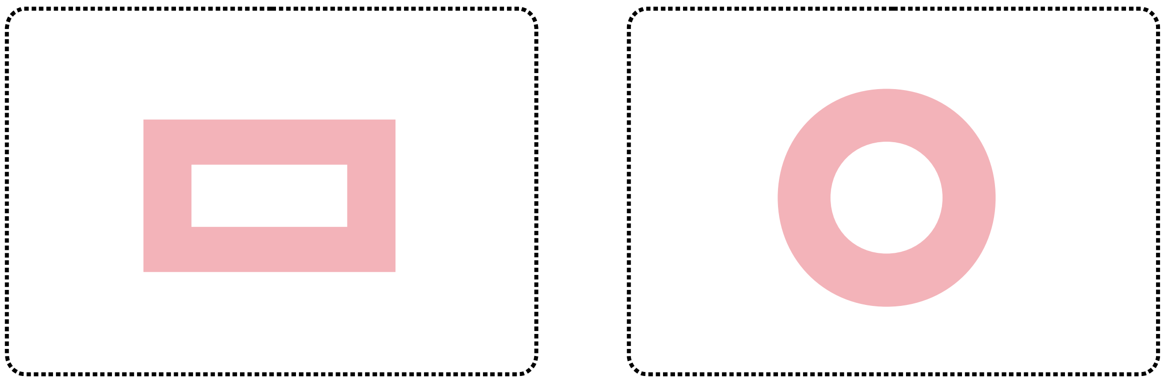
These 2 formats are very frequent for inlays or contactless stickers.
An inlay is a contactless chip glued on a very thin plastic film. The antenna is printed onto the plastic with conductive inks. Most today's low cost contactless-only cards are manufactured by embedding a Class 3 inlay between two pieces of white plastic.

Illustration 13: Definition of PICC classes 1, 2 and 3



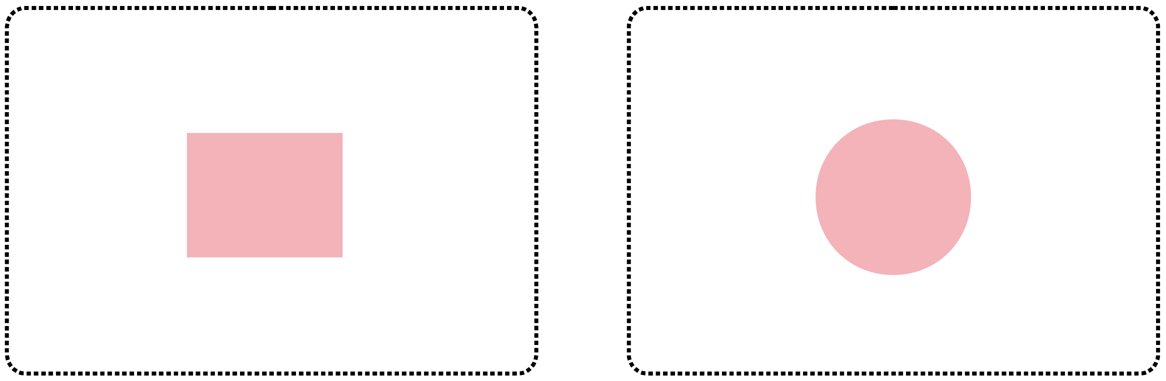
Class 4

2 smaller formats widely adopted by NFC Tags.



Class 5

These 2 formats are also now very frequent for inlays embedded in NFC Tags, wristbands or watches.



Class 6

These smallest formats are very interesting to "tag" small objects in RFID applications.

Illustration 14: Definition of PICC classes 4, 5 and 6

3.3.4. Which classes a coupler has to support?

According to ISO/IEC 14443-1, only the support of Classes 1, 2 and 3 is mandatory for all contactless couplers (PCDs).

All out-of-shell SpringCard contactless couplers support classes 1 to 5[21].

Last generation couplers do also support class 6 with a decent operating distance (at least 1.5 to 2 cm). Customers shall be aware that couplers of earlier generations may be unable to operate class 6 cards at more than 0.5 cm – which may be considered as too short for a satisfying user experience.

The standard does not claims that a PICC outside of the 6 classes should not or could not be supported, but the operation of a compliant PCD could only be guaranteed with PICCs belonging to one of the 6 classes.

3.3.5. Actual operating distance

“What is the operating distance of this coupler” is the most frequent question ever heard by a manufacturer of contactless couplers.

And it is also the question one couldn’t answer, because the answers comes not only from the coupler, but also from a lot of parameters coming the card itself:

- The size of its antenna, of course,
- How much power it requires from the RF field,
- How its RF circuit (antenna + matching capacitor) has been designed.

As a rule of thumb, you may consider that with a decently manufacturer wired-logic class 1 card, the maximum operating distance is something between 150 and 200% of the diameter (or diagonal) of the coupler’s antenna[22].

A contactless smartcard having the same format, but with a microcontroller that requires a little more power than the memory-only chip, will behave correctly only up to 125 or 150% of the diameter (or diagonal) of the coupler’s antenna, at least at slow speed (106 kbit/s). At a higher speed, the chip requires more power, and the signal over noise ration is decreased, so the operating distance decreases when the bitrate increases.

It may be noticed that performances are generally a little better or at least more repeatable in ISO/IEC 14443-A than in ISO/IEC 14443-B, due to a more robust modulation scheme.

[21] Some customer-specific products have been especially created to support very small tags (smaller than class 6) and are by design unable to power cards that are larger than class 6. But this devices are not made available to other customers.

[22] This is a little more than we have announced at first in 3.3.1, because most wired-logic cards may start running on a field as low as 0.8 A/m (instead of the 1.5 A/m limit considered in first approach).

Most SpringCard OEM couplers are provided with a 69×45 mm antenna board. The antenna is balanced and its active diagonal is approx. 65 mm long.

In optimal conditions, this antenna is able to operate a Class 1 MIFARE Classic card (genuine NXP chip, high-end card manufacturer) up to 12 or 13 cm.

The Prox'N'Drive is typically placed behind a windscreen.

Since glass has a different magnetic permittivity than air, the Prox'N'Drive's antenna has a very different tuning than other couplers'.

Symetrically, the antenna of a card or tag will be detuned if you place it on a windscreen or a glass bottle; if the card is not to be used "in free air", it must be chosen or designed accordingly.

3.4. 'Vicinity' contactless cards

3.4.1. The need for hand free systems

While the ISO/IEC 14443 standard covering *proximity* cards in the HF band was discussed in the early 2000's, companies interested in hand free access were simultaneously working in a so-called *vicinity* standard.

There are many use cases for such a system: ski passes in the winter resorts, wearable tags (bracelets, rings) for user identification in the leisure or gaming industries, tagging books in libraries to implement both inventory and anti-theft solutions, and more.

We have introduced in 3.1.1 the proximity coupler and the proximity card as the two circuits of a mobile, unconstrained transformer.

Proximity means the secondary coil remains close to the primary coil[23]. This location ensures a descent coupling between both coils. The chip in the card receives enough energy to perform demanding computational operations, and high-speed modulation schemes could be used. There are only little concerns regarding the signal/noise ratio and EMC compliance.

On the other hand, *vicinity* means that the two coils are farther. This has two consequences:

- since the field decreases very quickly with the distance, the card receives lots less power; increasing the coupler's RF power is not an option since it would overrule EMC limitations,
- due to the worse coupling, card's load-modulation is likely to fell down under the signal/noise ratio of the receiver.

The only viable approach is:

1. Choose a chip that requires less power, i.e. that implements less features and communicates slowly. Also a robust modulation scheme is required to ease the receiver's job,
2. Increase the size of the coupler's antenna, so the part of the space where the RF field floods could be significantly larger, hence allowing a wider operating volume with the same nominal RF level.

As you may have seen in ski resorts, swimming pools or leisure centers, the couplers used for hand free access are definitively larger than the PCDs used with contactless smartcards!

[23] The distance matters will be discussed in 3.5

3.4.2. The standards

The result of this effort on hand free systems is the ISO/IEC 15693 standard, introducing the concept of *vicinity integrated circuit card* (VICC) and of course *vicinity coupling device* (VCD).

3.4.2.1. Size

As it is the case for ISO/IEC 14443-1, ISO/IEC 15693-1 states that the VICC's antenna shall in no case exceed 86×54 mm. There is no other explicit size constraint or even advice for VICCs. Anyway, the VCD test bench uses only ID-1 VICCs, and that implies that the behavior with a VICC smaller or bigger than ID-1 could not be tested using the standard test tools.

3.4.2.2. Field level

The field level is defined in ISO/IEC 15693-2. The VICC must be able to operate on a field as low as 0.15 A/m.

Let's go back to illustration 11 on page 45: we have written then that PCB corresponding to the purple curve (Ø = 7.5 cm, H_0 = 3.5 A/m) was able to power all PICCs up to 3 cm and most of them up to 6 cm.

The very same device, in VCD mode, will be able to power all VICCs up to 10 cm.

A coupler could be designed to communicate both with proximity and vicinity cards. Using the standards' wording, it is both a PCD and VCD.

Due to EMC limitations, such a coupler must have the (small) size of a PCD, overwise it would not be allowed to implement the proximity modulation. Therefore, its typical operating range is not really longer in VCD mode than it is in PCD mode.

A larger reader (green curve, Ø = 10 cm, H_0 = 2.5 A/m or blue curve, Ø = 50 cm, H_0 = 1.5 A/m) is required for actual hand free mode (illustration 11 and 15).

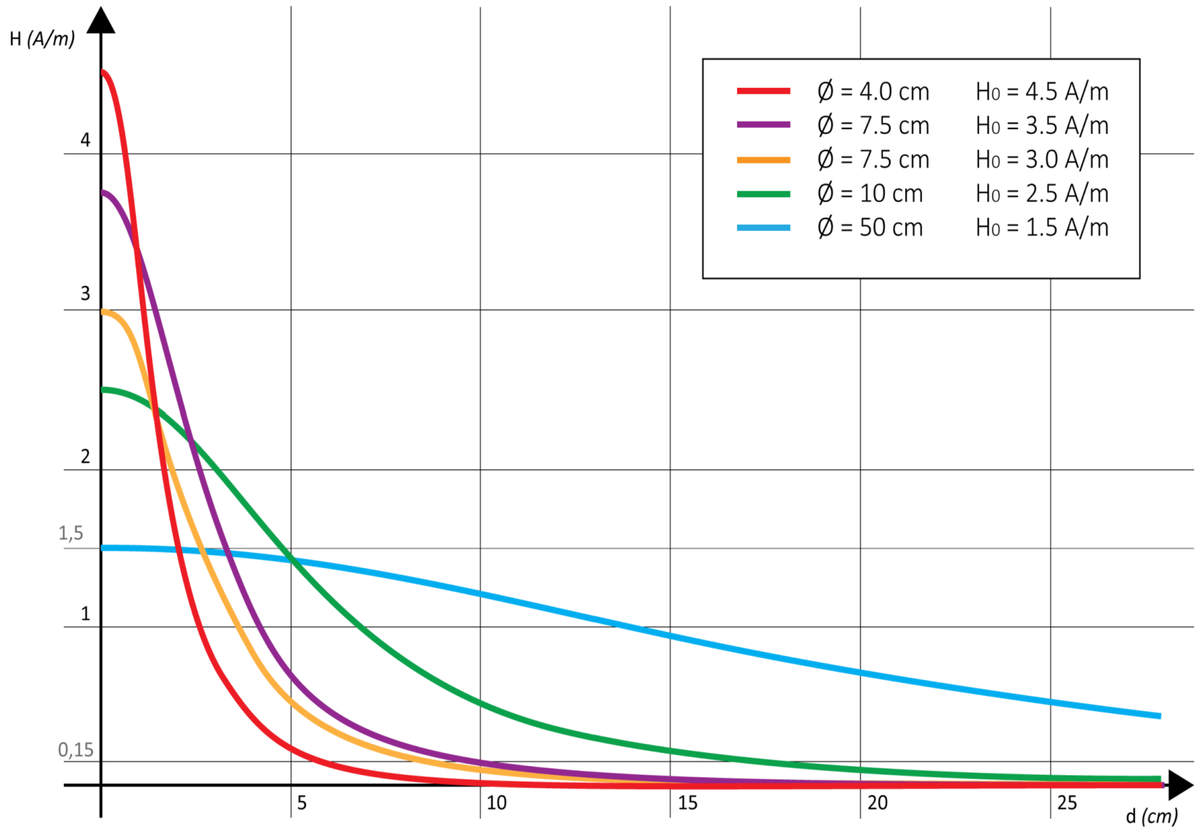


Illustration 15: RF field level against distance to the antenna, for different readers, up to 25 cm

3.4.2.3. Communication protocol

The field level and bit-level modulation are defined in ISO/IEC 15693-2. The very same 13.56 MHz carrier as defined in ISO/IEC 14443-2 is used, and the field level ranges allowed by both standards overlap for a wide part.

There are a few strong differences with the proximity standard:

- The VCD is responsible for choosing the communication bitrates and options in both direction, in order to meet the local EMC limitations (FCC 47 part 15 in the US, ETSI 300-330 in Europe), depending on the size of its antenna and the nominal RF power it provides,
- For VCD to VICC communication, a 30% *Amplitude Shift Keying modulation* (ASK) is used (instead of OOK or 10% ASK for proximity),
- The available bitrates for VCD → VICC are 26.5 kbit/s for small couplers where the card remains at a short distance (when the VCD is also a PCD, basically) and 1.6 kbit/s for the larger couplers actually implementing the hand free mode,

- The available bitrates for VICC → VCD are 26.5 kit/s (short distance) and 6.6 kbit/s (hand free). The VCD may choose between two different load-modulation schemes, depending on its technology and expected signal/noise ratio of the surroundings.

3.4.2.4. Application-level protocol

The stack is topped by ISO/IEC 15693-3 that defines the VICC as a direct-access memory, divided into blocks, making it very close to a wired-logic proximity card.

This layer of the standard introduces the following concepts:

- Every card is identified by a unique serial number, the UID. Every card manufacturer has its own Manufacturer ID, and is responsible of the unicity of the UIDs attributed under this Manufacturer ID *(*see 6.3.2),
- The card is basically an unsecure storage. Securing the access – and maybe the communication – is possible only through proprietary commands (some cards may be password-protected, some other may feature a more advanced authentication scheme based on cryptographic functions),
- The memory is divided into blocks of fixed size – but every manufacturer chooses freely the size of its blocs (a lots of cards available on the market today have 4-B blocks, some others are 1-B or 2-B only, some 8-B or 16-B, and there's no reason not to see larger blocks in the future),
- The memory is read and written at block-level. Optional functions make it possible to read/write more than one block at once (but never less),
- The card may provide a lock mechanism to turn every block permanently read-only – given the lack of mandated security, this is the minimum feature to be sure that the data could not be modified after writing.

There is no provision in the standard to implement smartcard features (APDUs and file-system) on top of the ISO/IEC 15693 stack, because running a card microcontroller, even very small, on the weak RF field of a VCD located 1.5m away, is not possible with today's technology (or at least not possible at an acceptable cost).

But the technology evolves constantly, and so do the standards.

3.4.3. Vicinity contactless cards *vs* RFID HF tags or labels

The ISO *smartcard* workgroup is responsible for all the standards discussed until then. There are many other workgroups at ISO's. One of them is in charge of RFID, *radio-frequency identification*.

The RFID workgroup is driven by key actors of logistics, warehouse and retails. They focus on open-loop database-centric systems, in which the RFID chip is no more than the pointer to a record in a database. Storing a large amount of data or implementing complex cryptographic features in the chip is totally out of their scope. Basically, their first need is a (very) cheap technology, because they will use billions of disposable tags. Also, a fast and reliable anticollision scheme is a must have, to be able to read dozens of tags at the same time.

The result of their work is the ISO/IEC 18000 family of standards; every radio-frequency band that could potentially be used for RFID applications is covered by its own entry in the family. This includes for instance ISO/IEC 18000-2 for the LF ISM band (125/135 kHz) and ISO/IEC 18000-6 for the UHF ISM band (860/960 MHz).

Closer to our focus, ISO/IEC 18000-3 covers the HF ISM band, i.e. the 13.56 MHz frequency. This standard is itself divided in three modes that are not interoperable[24]:

- Mode 1 (ISO/IEC 18000-3M1) is very close to ISO/IEC 15693. A VICC compliant with ISO/IEC 15693 may be used together with a ISO/IEC 18000-3M1 interrogator,
- Mode 2 (ISO/IEC 18000-3M2) uses *phase-jitter modulation* (PJM) for the coupler-to-tag channel, which is very different from the ASK or OOK modulations used by the other HF standards,
- Mode 3 (ISO/IEC 18000-3M3) is the ISO transcription of the *GS1 EPC HF RFID Air Interface*. GS1 is the child organization born from EAN (Europe Article Numbering) and UCC (Uniform Code Council), the organizations behind the standards for barcodes. The aim of GS1 is to replace barcodes by *electronic product codes* (EPC). The ISO/IEC 18000-3M3 protocol has been designed by GS1 to offer the best anticollision speed[25].

SpringCard couplers that are ISO/IEC 15693 VCD could be used with ISO/IEC 18000-3 Mode 1 tags.

Mode 3 tags will be supported by future products (UID reading only).

There is no plan to support Mode 2 tags.

[24] This means that a given system shall implement only 1 of the 3 modes, and that a system running another mode will cause significant interferences.

[25] A high-end dedicated RFID reader is likely to identify up to 700 tags per second using ISO/IEC 18000-3M3, against up to 60 tags per second using ISO/IEC 18000-3M1.

3.5. NFC Tags

NFC Forum Tags are not a new technology, but a way of using contactless cards to exchange in open-loop, mass-market applications.

3.5.1. NFC and the NFC Forum

In 2003, Philips (the provider of the MIFARE cards, NXP being in the past the Philips Semiconductors division) and Sony (the provider of the FeliCa cards) introduced a new concept of short range wireless network, using inductive coupling (i.e. near field technology) in the 13.56 MHz ISM band.

This concept was entitled *near field communication – interface and protocol – 1* (NFCIP-1). It has been adopted in 2005 as international standard ISO/IEC 18092.

This standard completes the classical *active coupler/passive card* scenario by introducing *active/active* communication: both the coupler (or *initiator*) and the card (or *target*) may now generate the 13.56 MHz carrier alternatively. It also introduces a *peer-to-peer* communication mode, superseding the earlier command/response (or master/slave) schemes.

The peer-to-peer mode is theoretically ready to convey any kind of higher-level network and application protocols, hence the analogy between NFCIP-1 and TCP/IP.

Then, Philips and Sony partnered with Nokia (at that time the leader of the mobile phone market) to create NFC Forum. Their goal was to develop an ecosystem of added-value applications, combining all the technologies based on inductive coupling in the 13.56MHz band: the new ISO/IEC 18092 peer-to-peer implementation, and the pre-existing ISO/IEC 14443 and ISO/IEC 15693 coupler (or reader/writer) and card implementations.

Putting everything together, and after 10 years of existence, NFC Forum provides a consistent set of specifications, largely adopted by mobile phone manufacturers and also by mobile network operators, to do the following:

- Have a mobile phone emulate a contactless smartcard – and be able to use its SIM card (or UICC) as a secure execution environment; this has led to Apple Pay, Google Wallet and a few alternative contactless payment systems, and to the virtualization of public transport tickets into the phone,

- Have two mobiles phones communicate together using NFCIP-1 "peer-to-peer" to exchange small pieces of information: a business card or a contact entry, a link to the running application or to the currently active web page, etc. This system is known as *Windows Proximity Networking* in Microsoft's world, and *Android Beam* in Android's world[26].
- Have a mobile phone fetch small pieces of information from a static object: a NFC Tag.

The later is the concept that is detailed in the next paragraphs.

Developing in the mobile phone is out-of-the scope of this guide since it has little in common with PC/SC and is strongly tied to the mobile system's API. But at least, the concepts at work when exchanging an APDU with a smartcard are the same.

Implementing card emulation or peer-to-peer using SpringCard couplers will be addressed in a future guide.

3.5.2. The concept behind NFC Tags

Basically, NFC Tags are contactless cards holding some publicly-readable data. These data are organized according to an open format. They should be understood and handled the same way by all the "readers". In most situations, the "reader" is a software component provided by the terminal's operating system itself, namely Android or iOS for smartphones.

Most frequent usages of NFC Tags are opening a web page in the smartphone's browser, but there are many other innovative use cases as well: initiate a phone call, add a business card to the list of contacts (vCard), configure a wireless communication channel (WiFi or Bluetooth automated pairing), launch and application and/or execute a custom action on the phone.

In short, NFC Tags provides the same functionality to end-users as 2D barcodes (QRCode / flashcode), but with a faster – and more user-friendly – gesture.

3.5.3. NFC Forum Data Exchange Format and Record Types

The public content that is stored in the card's memory is organized according to a set of specifications edited by the NFC Forum.

[26] NFC communication requires that the two phones stays in close proximity, which is a terrible user experience when the duration of the exchange increases. For this reason, NFC peer-to-peer is limited to 1 or a few KB of data only. When more data are to be transmitted, both peers negotiate the opening of a new communication channel, based on Bluetooth or WiFi. The NFC channel is then used only for *handshaking*, the devices do a *handover* and communicate through the faster, longer-distance 2.4 GHz wireless channel.

The top level specification is called a NFC Forum Data Exchange Format (NDEF). A NDEF *message* contains one or a few *records*. The specification of the records are named Record Type Definitions (RTD).

The commonly used RTDs are

- *URI* or *URL* (*http:*, *https:*, *mailto:*, *callto:*, *smsto:* …),
- *Text*, to store an informative text that could be displayed to the user,
- Arbitrary *MIME data* (for instance to store a small icon using the *image/png* type, or a business card with the *text/x-vCard* type),
- *Connection handover* (WiFi or Bluetooth automated pairing),
- *Signature*, holding the proof that the NFC Tag has been created by a certain issuer, and that the NDEF message has not been altered.

The *smartposter* RTD refers to a compound record, holding an *URL*, a *Text* and an action verb (*open*, *save*, *print*..).

3.5.4. List of compliant PICCs / VICCs

NFC Forum content could be stored on any card that has enough memory to store the NDEF message itself and the associated headers[27].

Following the urge for a total interoperability between any NFC Tag and any NFC-enabled smartphone, the NFC Forum has limited its list of 'officially compliant' cards providers to only 5 families of contactless chips[28].

3.5.4.1. Type 1 Tag (T1T)

The NFC Forum T1T specification is a copy/paste of the datasheet of the "Topaz" card, a low-cost, wired-logic PICC using a custom protocol based on ISO/IEC 14443-2 type A bit-level modulation. The card has been developed by Innovision Research & Technology, now a part of Broadcom.

As any other wired-logic card, Type 1 Tags are supported by PC/SC couplers thanks to the embedded APDU-Processor, through the READ BINARY and UPDATE BINARY instructions. Once stored in the Tag, the NDEF message may be write-protected by setting a lock byte.

[27] They could also be pushed through a peer-to-peer communication channel (NFC beam)
[28] Until 2016, only types 1 to 4 were defined, therefore a lots of implementation may still lack support for type 5

3.5.4.2. Type 2 Tag (T2T)

The NFC Forum T2T specification is a copy/paste of the datasheet of the "MIFARE UltraLight" card by NXP (formerly Philips Semiconductors, Philips being one of the founders of the NFC Forum)[29]. This card is also a low-cost, wired-logic PICC, that has been designed with disposable transport tickets in mind. It uses a custom protocol on top of ISO/IEC 14443 up to-3, type A.

The "up to -3" makes it faster and more interoperable with old generations of couplers than the T1T.

The initial MIFARE UltraLight card was limited to 48 bytes of storage, but NXP now offers under the NTAG brand capacities going up to 2KB. Infineon also has an interesting portfolio of chips and cards compliant with T2T (*my-d NFC*).

The PC/SC embedded APDU-Processor gives access to Type 2 Tags through the same READ BINARY and UPDATE BINARY instructions. Also special lock bytes make it possible to turn the Tag read-only.

3.5.4.3. Type 3 Tag

The NFC Forum T3T specification is a copy/paste of the datasheet of the "FeliCa Lite" card by Sony. This is a light-weight version of Sony's original FeliCa card, with the security features removed. It uses a Japanese protocol (JIS X 6319-4) named "NFC-F" in NFC Forum's documentations).

The PC/SC embedded APDU-Processor gives access to Type 3 Tags through the same READ BINARY and UPDATE BINARY instructions.

3.5.4.4. Type 4 Tag

The NFC Forum T4T specification defines how to use a generic, interoperable, contactless smartcard to store a NDEF message.

The card shall support both ISO/IEC 14443 up to -4 (either type A or B, respectively named "NFC-A" and "NFC-B" in NFC Forum's documentations). On top of that, the card shall support the ISO/IEC 7816-4 command set (SELECT APPLICATION, SELECT FILE, READ BINARY) and provide 2 files to store a header and the NDEF message itself.

The specification does not describe how the files are created, nor how to write their contents, since NFC Forum focuses on the use (reading) and not on the issuing process.

Virtually any contactless smartcard may therefore be used to implement a T4T. Notably, SpringCard's SDK has an example on how to format a NXP Desfire EV1 to become a valid T4T.

STMicroElectronics also has a portfolio of smartcards already formatted as T4T.

[29] For more information regarding the MIFARE family, read 3.6.1.

The NDEF may be locked, or its write access may be controlled by a password or any kind of authentication method, depending on the features the card supports – and on the features the developer has decided to use.

3.5.4.5. Type 5 Tag

Adopted in 2016, the NFC Forum T5T specification defines how to use an ISO/IEC 15693 VICC to store a NDEF message ("NFC-V" technology in NFC Forum's documentations).

There is not much to add here: PC/SC couplers have been supporting for VICCs for long. The memory is accessed through READ BINARY and UPDATE BINARY instructions translated by the embedded APDU Processor to the Read Block / Write Block commands defined in ISO/IEC 15693-3.

Compliant cards could be found at NXP's (*ICODE-SLI* family), STMicroElectronics' (*LR* family), Infineon's (*my-d Vicinity* family) and Texas Intrument (*Tag-IT* family).

3.6. Key actors and brands

3.6.1. NXP, ex Philips Semiconductors

3.6.1.1. MIFARE – The historical background

In the early 1990's, an Austrian company called Mikron introduced an emerging technology: a wired-logic card featuring semi-structured storage and some kind of security, using inductive coupling at 13.56 MHz to communicate with its coupler.

As the 1st expected marked was the automatic fare collection, the card was named MIFARE for *MIkron FARE collection*. In 1994, Mikron started working with the ISO smartcard committee on the development of what would become ISO/IEC 14443 (type A).

In 1995, Philips purchased Mikron and the MIFARE technology. Later on, Philips decided to split its assets and Philips Semiconductors became NXP.

Philips / NXP decided to use the brand as a common name for an ever growing family of PICCs. Most of these products are still wired-logic cards (MIFARE Classic, MIFARE UltraLight), but some others are microprocessor-based cards (MIFARE Pro, ProX, SmartMX, MIFARE DESFire, MIFARE Plus).

3.6.1.2. MIFARE Classic

Illustration 16: Outprint of a MIFARE Classic promotional card

The early version of the MIFARE card is now known as MIFARE Classic. It comes in two flavors[30].

- The MIFARE Classic 1K features 64 blocks of 16 bytes. Over these 64 blocks, 16 are reserved to store security keys and control the access rights on 16 parts of the memory, known as sectors, and the 1st block store read-only manufacturers data (such as a unique serial number or UID). This makes 752 bytes available for user's data.
- The MIFARE Classic 4K features 256 blocks of 16 bytes, divided in 40 sectors. The sectors below 2K use the same "3+1 blocks" mapping as the 1K card, where the sectors above 2K use a "15+1 blocks" mapping. This makes 3440 bytes available for user's data.

The card uses a proprietary command set on top of ISO/IEC 14443-3 type A, but with a noticeable difference: MIFARE Classic's security relies on a proprietary stream cipher algorithm, CRYPTO1, and CRYPTO1 breaks the layered model by ciphering the parity bits and the CRC of every frame (instead of ciphering the data before computing the parity and the CRC).

CRYPTO1 uses 48-bit keys – which is very weak – and all cards manufactured until recently are affected by a vulnerability in the random number generator. The security scheme of MIFARE Classic has been totally broken in 2009-2010, and the card shall be considered for what it is: a low-cost, insecure storage memory.

3.6.1.3. MIFARE Plus

Following the security breach of CRYPTO1, NXP has introduced the MIFARE Plus, a new generation of contactless cards based on a small microcontroller, and offering a strong AES (128 bits) security, yet using (basically) the same command set and memory mapping as MIFARE Classic to allow a smooth migration.

The card's issuer is responsible for managing its life-cycle over 3 steps called "security levels" (SL), going from MIFARE Classic emulation (SL1) up to highly secure SL3 mode.

There are 2 subfamilies:

- MIFARE Plus X has all the security features. It is export-controlled,
- MIFARE Plus S has less security features, but could be exported freely.

Both cards exist in 2K and 4K memory size.

Two interesting security features of MIFARE Plus X are the *proximity check*, the use of a *distance-bounding protocol* to prevent relay attacks, and the *virtual card* concept, a way for the coupler to pre-select a particular virtual card in a NFC object that is able to emulate many.

[30] There used to be a 3rd flavour, the MIFARE Classic Lite which is more-or-less a 1K with only 4 sectors instead of 16, but it has been abandoned as prices of the 1K dropped down with maturity.

Also the *virtual card* concept increases the level of privacy of the contactless system, because the card may use a *random* protocol-level *identifier* (RID) instead of sending its a *unique identifier* (UID) in response to the coupler's polling loop.

Only a coupler belonging to the same "owner" as the card – and therefore sharing a secret key with the card – will then be able to retrieve the actual UID and start processing the card.

3.6.1.4. MIFARE UltraLight and NTAG

The low-end of the family is the MIFARE UltraLight, a chip with only 48 bytes available for user's data, and no security. It targets low-cost applications, where the card could be disposed-of after a single or a few uses only.

Being a cheap yet fast wired-logic card, the MIFARE UltraLight is the foundation of the NFC Forum Tag 2 standard (see paragraph 3.5.4.2). Chips fully compliant with MIFARE UltraLight, but with a larger memory capacity, are now sold by NXP under the NTAG brand.

Another derivate of the MIFARE UltraLight is the MIFARE UltraLight C that includes an authentication scheme based on the 3DES algorithm (112 bits). This card targets the market of disposable tickets, typically in the public transport field.

3.6.1.5. SmartMX

SmartMX is NXP's current family in secure microcontrollers, based on an 8051 (8-bit) architecture. It is marketed to host eGov applications (passports, ID card, health card…) or to power NFC-aware objects in the IoT world, and supersedes the earlier MIFARE Pro and MIFARE ProX families.

NXP also maintains JCOP, a JavaCard operating system initially developed together with IBM. JCOP allows card-application developers to develop Java "cardlets" for the SmartMX chip.

3.6.1.6. DESFire

The DESFire contactless smartcard is based on a core similar to the SmartMX, but is not open to card-applications developers. The chip comes with a general purpose operating system in ROM, that offers a simple directory structure and files.

The card communicates using ISO/IEC 14443-4 type A. The command set is proprietary, but could be encapsulated to comply with the APDU format of ISO/IEC 7816-4. A few ISO/IEC 7816-4 instructions have been introduced in version EV1 to ease the adoption of the DESFire card in open, interoperable architectures.

Last version is EV2, that adds the proximity check concept (introduced on MIFARE Plus) and more notably a delegated administration model, that solves most of the issues of multi-application card schemes.

Regarding the security aspects, the 1st generation rely on 3DES with 2 keys (112 bits) for mutual authentication and ciphering. EV1 and EV2 also support AES (128 bits) and 3DES with 3 keys (168 bits).

3.6.1.7. ICODE

ICODE is NXP's brand name for vicinity and RFID products. The pre-standard ICODE-1 has been discontinued for long, and three families exist today:

- ICODE-SLIX and ICODE SLIX 2 are ISO/IEC 15693 VICCs, also compliant with ISO/IEC 18000-3M1,
- ICODE DNA are also ISO/IEC 15693 VICCs, but they introduce high-end security (AES-based dynamic mutual authentication) that was previously available only on PICCs,
- ICODE ILT are compliant with ISO/IEC 18000-3M3 and EPC HF.

3.6.2. STMicroElectronics

3.6.2.1. Wired-logic PICCs

STMicroElectronics SR chips offer a short-range RFID interface compatible with ISO/IEC 14443-2 (bit-level modulation) type B, and the frame format of ISO/IEC 14443-3 type B. The anticollision scheme as well as the top-level command set are proprietary.

The 1st generation (SR176) and its descendants (SRT512, SRI512, SRI2K, SRI4K) are used as single-trip ticket in the public transport field, mostly embedded in disposable paper cards.

The SR series is now superseded by the ST25TB family (ST25TB512, ST25B02K, ST25B04K). Chips in this family are compatible with the previous generation from the coupler point of view.

The ST25TA family are contactless smartcards (ISO/IEC 14443, type A) already formatted to be an NFC Forum Type 4 Tag.

3.6.2.2. MCUs for PICCs

STMicroelectronics offers secure microcontrollers (ST21, ST33), based on ARM cores, to smartcard developers. They are for instance the platform powering a lot of Calypso cards.

3.6.2.3. VICCs

The LR series, as the initials suggest, is ST's long-range RFID family. They are fully compliant with ISO/IEC 15693.

The LR series will be shortly superseded by the new ST25TV family.

3.6.3. Infineon

3.6.3.1. Wired-logic PICCs

Infineon SRF 66R "my-d proximity" are ISO/IEC 14443-3 type A PICCs. They use the same command set as NFC Type 2 Tags, plus a proprietary authentication scheme.

The family includes products branded "my-d NFC" with no authentication and the memory initialized as a NDEF container.

Infineon is also a MIFARE licensee and offers a clone of the Classic 1K (SLE 66R35).

3.6.3.2. MCUs for PICCs

Infineon has a wide range of secure microcontrollers (16-bit core) with ISO/IEC 14443 interface (SLC32, SLE77, SLE78). They are the basis of many payment or transport cards. Some of them may incorporate a MIFARE Classic emulation.

3.6.3.3. VICCs

Infineon SRF 55V "my-d vicinity" are ISO/IEC 15693 + ISO/IEC 18000-3M1 VICCs. Some chips in the family include a proprietary authentication scheme.

3.6.4. Texas Instrument

3.6.4.1. VICCs

The Tag-it HF-I chips are fully compliant with ISO/IEC 15693. They can be used as NFC Type 5 Tags as well. Texas Instrument offers a very wide choice of ready-to-use inlays based on these chips.

3.6.5. Atmel (now Microchip)

3.6.5.1. Wired-logic PICCs

Atmel CryptoRF (AT88SCxxxxCRF) are wired-logic contactless chips or ready-to-use tags, using ISO/IEC 14443-2 (bit-level modulation) and ISO/IEC 14443-3 (frames and anticollision) type B.

4. PC/SC Stack – role, specificities, alternatives

4.1. Introduction

PC/SC stands for “Personal Computer / Smart Card”. It’s the *de facto* standard to use smartcards – and smartcard couplers – from a mainstream computer environment.

PC/SC has been written and is maintained since 1996 by the PC/SC Workgroup.

The PC/SC Workgroup’s website is: www.pcscworkgroup.com

PC/SC provides a complete abstraction of the underlying coupler and driver. All SpringCard PC/SC couplers comply with this standard. It ensures interoperability among manufacturers: a SpringCard coupler may be used with a PC/SC-aware application written with another coupler in mind, and vice-versa.

Interestingly, PC/SC makes no difference between a contact coupler (CD) and a contactless coupler (PCD/VCD), and hides most of the specificities of either technologies. Last but not least, PC/SC makes it possible to operate wire-logic cards (and more specifically wired-logic contactless cards, RFID labels and NFC tags in the case of SpringCard contactless couplers) using “classical” APDUs, as if they were microcontroller-based smartcards.

The possible alternatives to PC/SC are covered by chapter 8.

4.2. The PC/SC architecture (and vocabulary)

PC/SC relies on a layered architecture, going from the coupler (*smartcard reader*) and its driver up to the high-level *application programming interface* (API) through a *middleware* (system-provided processes and libraries) that implements both the device abstraction facility and a strong isolation between the client card-aware applications – for security reasons[31].

Illustration 17 (page 70) shows the PC/SC stack.

4.2.1. Smartcards, readers and drivers

The smartcard or *ICC* (Integrated Circuit Card) is inserted into a reader or *IFD* (InterFace Device). This is, strictly-speaking, a “coupler” and not a reader.

The IFD, a hardware product, is associated to a reader driver or *IFD handler*.

[31] Imagine you enter your PIN in an application to unlock your credit card, or your phone’s SIM card, to explore its content. Do you really want that another application (maybe a malware) could also communicate with your now unlocked card?

The manufacturer of the product has two options: either provide a custom driver for "all" the operating systems, or implements its product following an open specification, in the hope that the operating systems will provide a generic driver based on the same open specification. This is the option opened by the USB workgroup with the USB *contact card interface device* (CCID) specification, and this is the option chosen by SpringCard for its PC/SC couplers.

Thanks to this USB CCID profile, SpringCard USB PC/SC couplers are supported by a generic, open source, USB CCID driver, that is available on Linux and many other Unix-like systems. Apple also uses the source code of this driver to support the CCID USB PC/SC couplers in Mac OS X.

Microsoft Windows also provides a generic USB CCID driver in Windows, but unfortunately this driver as a lot of limitations. Therefore, SpringCard provides its own USB CCID driver for Windows.

4.2.2. The middleware

The drivers are under the control of the PC/SC middleware or *ICC resource manager*.

This middleware is responsible for maintaining the list of currently connected couplers, notifying the applications when a smartcard is inserted in a coupler, and preventing two (or more) applications from accessing the same card simultaneously.

On Windows, the PC/SC middleware is implemented by a system service named "Smart Card Service" (scardsvr.exe). On Linux/Unix-likes, it is implemented in the *pcscd* daemon.

4.2.3. The API

An application that aims to use smartcards – a *ICC-aware application* – loads a user-land dynamic library to invoke the services provided by the PC/SC middleware through entry-points specified as the "PC/SC API".

On Windows, the shared PC/SC library is *winscard.dll*, and *libpcsclite.so* on Linux/Unix-likes. There are various wrappers for managed platforms (such as .NET or Java) or script engines (Python, Perl, Lua, JS, etc).

4.2.4. Helpers

In-between the middleware and the application, the 'helpers' are optional software components that give a more abstract view to the application developer.

For instance, a card providing general-purpose cryptographic primitives (ciphering, deciphering, digital signature) is likely to be associated with a 'card helper' that exposes its primitives with a high-level, interoperable, object-oriented approach.

Thanks to the helper, the application developer does not have to deal with the card at APDU-level; his application may even support different cards easily, provided that they offer the same primitives – even if this is done with different implementations.

Typical examples are the cards used for digital signature or secure login onto the computer or the Active Directory infrastructure, that are supported through Windows' CryptoAPI

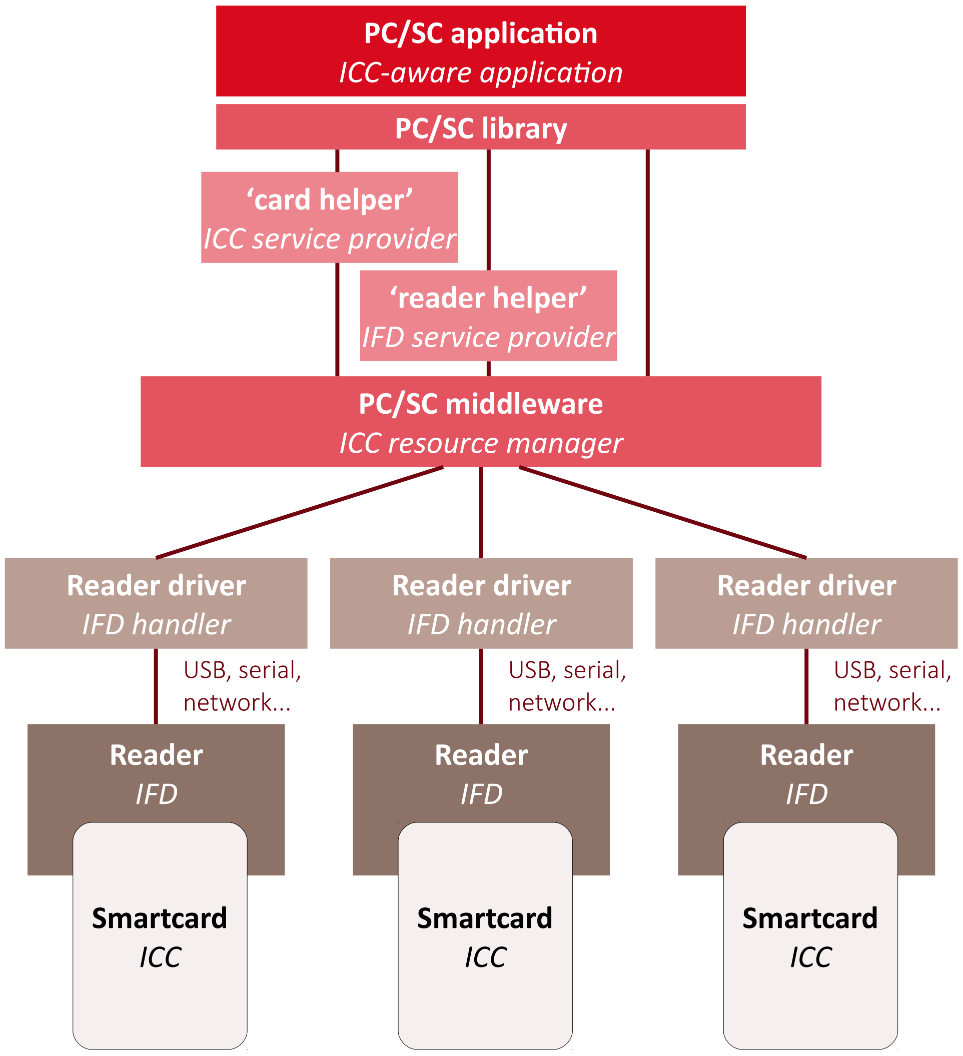


Illustration 17: The PC/SC stack

4.3. PC/SC on Microsoft Windows

Microsoft was one of the founders of the PC/SC workgroup, and all desktop systems since Windows 95 and Windows NT 4.0 do include the middleware. This is not so clear for mobile or embedded systems (Windows CE, Windows Phone, Windows IoT Core) where the middleware may be installed or not, depending on how the system image is generated.

Following the deep integration of NFC technologies into the operating system of a today's mobile phone, Microsoft has introduced in its Windows Phones and Surface Tablet a new *Proximity* API, that is an alternative to PC/SC to communicate with contactless cards.

But to-date, the *Proximity* API is strictly tied to a few specific hardware, where the NFC interface is directly connected to the core CPU, and not a remote (USB / network / other?) device.

Also, the *Proximity* API is limited to NFC Forum's specifications, and therefore offers little to no support for the many families of wired-logic contactless cards that have not been endorsed by the NFC Forum.

Hence, PC/SC remains the interface of choice for pluggable contactless couplers.

4.3.1. Official documentation

The documentation related to PC/SC is sprinkled among many pages of Microsoft's websites. We list here only the two main entry points:

A general introduction to using the smartcard under Windows, simply entitled "Smart Card" is available at:

https://msdn.microsoft.com/en-us/library/bb742533.aspx

The article "Accessing a Smart Card" is the entry-point to explore the API, as it is implemented by Microsoft, by clicking "SCardConnect" after having read the introductory text:

https://msdn.microsoft.com/en-us/library/windows/desktop/aa374709(v=vs.85).aspx

4.3.2. Technical implementation

The PC/SC API is exposed through a single dynamically callable library, *winscard.dll* .

Key functions of the PC/SC API are introduced in chapter 5, and more completely documented in doc. PMDZ061.

C and C++ developers building native Windows applications use the *winscard.h* header and link their program against *winscard.lib*, which provide the static entry points to use *winscard.dll* . Care must be taken that for every function that takes a string as parameter (for instance, the name of the coupler in *SCardConnect*), the library exposes both ASCII and UNICODE versions (*SCardConnectA* and *ScardConnectW*).

Managed applications must use a wrapper to access the PC/SC stack through *winscard.dll* ; a Java application is likely to use the standard wrapper named *javax.smartcardio*.

For .NET applications, there is no standard wrapper. SpringCard proposes its own wrapper for convenience (look for the `SpringCard.PCSC` namespace in the SDK). There are also plenty of open-source alternatives, and also a few paid solutions (including ActiveX components for legacy development environments).

4.3.3. Writing and using card helpers

Microsoft recommends to write a “card helper”, or *ICC service provider,* a reusable software component that provides high-level access to the services offered by the smartcard. This simplifies the job of the top-level application developer, who does not have to manipulate the APDUs and any raw data in his business logic.

Vendors of cryptographic smartcards, used for digital signature and/or to logon the system, typically do so by providing either a Smart Card Cryptographic Provider (CSP) or a Smart Card Mini Driver.

Smart Card Cryptographic Service Provider specification:
https://msdn.microsoft.com/en-us/library/ms953432.aspx

Smart Card Mini Driver specification:
https://msdn.microsoft.com/en-us/windows/hardware/drivers/smartcard/smartcard-minidrivers

Anyway, in most projects, it is much simpler – and perfectly fine for the developer(s) – to implement the communication with the smartcard directly into the application. It is also possible to implement a basic helper through a DLL or a Class Library, without bothering with CSP or Mini Driver specifications.

Doc. TBD introduces such a Class Library: `SpringCard.NfcForum.Tags`; written in C#, it allows an application developer to read or write NFC Forum Tags at a high abstraction level, on top of the `SpringCard.PCSC` classes.

4.3.4. Limitations

Some parts of Windows' PC/SC stack have always been the source of issues when more than **10 couplers** are connected to the computer.

Starting with Windows 8, Microsoft officially limits the use of its PC/SC stack to ten couplers.

You may attach more than ten coupling devices to the computer, but *SCardListReaders*, the command that enumerates the couplers, will never return more than ten. The other couplers are ignored.

Please read Microsoft KB #3144446 for reference and explanations.

Some SpringCard PC/SC couplers, like the CSB HSP or the CrazyWriter HSP, could host in a single USB device as many as 5 or 6 PC/SC couplers (1 contactless slot, 1 ID-1 slot, 3 or 4 SIM/SAM slots). It is very easy to overflow the system's limitations with these products.

4.4. Linux and other UNIX-like systems

A group of volunteers has created MUSCLE, Movement for the Use of Smart Cards in a Linux Environment. They have developed, and now maintain, PCSC-Lite, the open-source PC/SC stack for Linux and other UNIX-like systems.

MUSCLE PCSC-Lite project: http://www.musclecard.com

Direct link to PC/SC stack: http://pcsclite.alioth.debian.org

The PC/SC middleware is implemented in a daemon named *pcscd*. The PC/SC API is exposed through a single dynamically callable library, *libpcsclite.so.1* . The corresponding header file is named *winscard.h* for compatibility with Windows applications at source code level.

SpringCard USB PC/SC couplers are supported thanks to the open-source CCID driver developed for PCSC-Lite by Ludovic Rousseau, as documented here:

Direct link to PCSC-Lite's USB CCID driver:
https://pcsclite.alioth.debian.org/pcsclite.html

For SpringCard non-USB couplers (for instance, TwistyWriter-IP PC/SC relies on a CCID over TCP over Ethernet implementation), SpringCard provides an open-source driver[32].

SpringCard NetPCSC driver for PCSC-Lite:
http://tech.springcard.com/2016/springcard-netpcsc-for-pcsc-lite/

Ludovic Rousseau also authors a very interesting blog related to smartcard development in the UNIX-world. It also documents how to access the PC/SC API from various scripting languages (Lua, Python, PERL, Node.js, etc).

Ludovic Rousseau’s blog: https://ludovicrousseau.blogspot.fr/

⚠ SpringCard is not connected with and does not sponsor or endorse 3rd party open-source developers.

4.5. macOS X

Apple provides a fork of the PCSC-Lite stack (and of the open-source USB CCID driver) within their UNIX-like system. The dynamically callable library is renamed *PCSC.framework/PCSC*.

The documentation is hosted on GitHub:

Apple Smart Card Services: https://smartcardservices.github.io/

Once again, Ludovic Rousseau’s blog is a very interesting source of information, with many tips (or bug explanations) related to Apple Mac OS X:

The ‘”Apple” keyword at Ludovic Rousseau’s:
https://ludovicrousseau.blogspot.fr/search?q=apple

⚠ SpringCard is not connected with and does not sponsor or endorse 3rd party open-source developers.

[32] SpringCard drivers for PCSC-Lite are forks of the original USB CCID driver for PCSC-Lite, and are released under the same LGPL license.

4.6. Android

As a Linux-based system, it would be logical that Android includes its version of PCSC-Lite. But this is not the case in out-of-the-box system images.

Giesecke & Devrient, a german manufacturer of smartcards, promotes a complex project named SEEK for Android, SEEK being “Secure Element Evaluation Kit”, that includes some efforts to port PCSC-Lite on Android.

Unfortunately, their promising *pcscdroid* project is not maintained anymore, and is strictly limited to custom system images or to ‘rooted’ devices, which prevent any adoption as a mainstream solution.

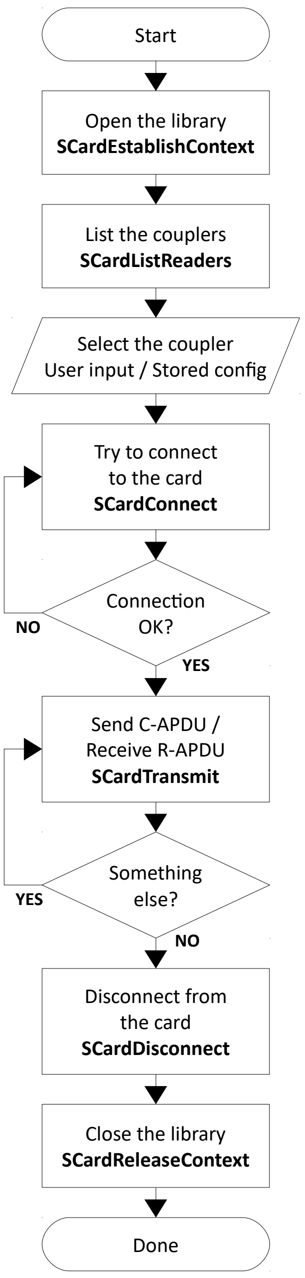
SEEK for Android main page: http://seek-for-android.github.io/

The Pcscdroid project:
https://github.com/seek-for-android/pool/wiki/%5BUNMAINTAINED%5D-Pcscdroid

⚠ SpringCard is not connected with and does not sponsor or endorse 3rd party open-source developers.

5. An application that uses PC/SC – Introduction to the API and typical workflow

5.1. Introduction



The illustration on the left shows the typical workflow of a PC/SC application.

The functions are introduced in the paragraphs hereafter. For a more complete documentation, please refer to doc. PMDZ061 or to Microsoft's documentation available on MSDN.

In most situations, it is possible to use a PC/SC coupler from an object-oriented language using higher-level objects. This is for instance the case in .NET thanks to SpringCard's `SpringCard.PCSC` class library, and to `javax.smartcardio` in Java.

Care must be taken anyway that, whatever the depth of OOP concepts and classes under your application, it all summarizes as passing calls to the underlying PC/SC API. Therefore, it is better to have a clear understanding of the mechanisms at work, in order to avoid a few design flaws that could easily turn into a bad user experience.

Chapter 7 gives more details about this.

We must also put emphasis (once again) on the fact that the PC/SC coupler is not much more than a pass-through device between the application and a smartcard. The coupler has no clue of what you (or what the application you are developing) want to do with the smartcard. It does even not now how to "read" or "write" a card in general case; always refer to the smartcard's documentation!

Yet, to support wired-logic contactless cards (or RFID labels, or NFC tags), the PC/SC coupler embeds some processing logic that translates a few chosen C-APDUs (READ BINARY, UPDATE BINARY) into commands the card will understand (more on this in paragraph 6.6).

5.2. Establish a PC/SC context

The first step is to initialize the client part of the PC/SC API by calling the *SCardEstablishContext* function.

Within *winscard.dll* (or its counterpart for PCSC-Lite) this has the effect of opening a communication channel with the middleware and to instantiate the variables.

In a multi-threaded application, every thread (that calls functions from the PC/SC API) shall have its own context.

```
LONG SCardEstablishContext(IN DWORD dwScope,
                           IN LPCVOID *pvReserved1,
                           IN LPCVOID *pvReserved2,
                           OUT LPSCARDCONTEXT phContext);
```

For those non-familiar with the Hungarian notation used by Microsoft, let's translate:

- *dwScope* is a double-word (4 bytes) input value. Set it to the constant *SCARD_SCOPE_SYSTEM* (value = 2).

In the context of a terminal server or remote desktop session, *dwScope* tells the PC/SC middleware whether the application wants to use the server's resources, the terminal's, or the local user's.

In any other contexts, use *SCARD_SCOPE_SYSTEM* with no hesitation.

- *pvReserved1* and *pvReserved2* are constant input pointers to anything (*LPCVOID* stands for *long pointer to a const void*). Set them to *NULL*.
- *phContext* is a pointer to a *SCARDCONTEXT (LPSCARDCONTEXT* is equivalent to *void *SCARDCONTEXT)*, that will hold the output data after a successful call.

SCARDCONTEXT is an opaque handle. With a non-C/C++ language, use a generic pointer (*System.IntPtr* on .NET). This makes *phContext* a pointer to a generic pointer (*ref System.IntPtr* on .NET).

SCardEstablishContext, as well as all other functions from the PC/SC API, returns a *LONG* (signed long integer). On success, its value is *SCARD_S_SUCCESS* (value = 0).

Refer to paragraph 7.4 for a few information regarding the error codes and their understanding.

5.3. List the PC/SC couplers

Get a list of the available PC/SC couplers using the *SCardListReaders* function.

```
LONG SCardListReaders(IN SCARDCONTEXT hContext,
                      IN LPCTSTR mszGroups,
                      OUT LPTSTR mszReaders,
                      IN OUT LPDWORD pcchReaders);
```

Let’s explain the Hungarian notation and the parameters again:

- *hContext* is the pointer you got as output to *SCardEstablishContext*,
- *mszGroups* is a constant input pointers to a string: *long pointer to a const “T” string*, the “T” meaning that the string could use either single-byte characters (char) or multibyte characters (wchar_t)[33]. Set this parameter to *NULL*.
- *mszReaders* is the output string that will hold the list of couplers after a successful call. It shall have been allocated by the caller application,
- *pcchReaders* is a pointer to the length of the *mszReaders* string (in number of characters, not number of bytes). On input, the caller application sets the pointed value to the allocated length. On output, the library tells how many characters are used by the string.

Classical examples use *SCardListReaders* twice: upon first call *mszReaders* is set to *NULL*, so *pcchReaders* provides the number of characters that must be allocated. Then the caller application allocates a string of the given length, and calls *SCardListReaders* a second time with it.

On modern systems, and given the limitation to 10 couplers on Windows, just allocate a large string (say, at least 1024 characters) once for all.

On output, *mszReaders* holds the list of couplers, in the form of a multi-string string. Now, what is a multi-string string? It’s an array of characters (*char* or *wchar_t* depending). As any other string, every string in the array is zero-terminated (*‘\0’* character). The end of the array is marked by a double zero (*“\0\0”*).

`javax.smartcardio` implements *SCardListReaders* through the *CardTerminals* class.

`SpringCard.PCSC` for .NET implements *SCardListReaders* through the *SCardReaderList* class.

[33] Actually, the *SCardListReaders* function has no existence. It’s an alias to either *SCardListReadersA* (ASCII implementation, the strings being handled as *char[]*) or *SCardListReadersW* (UNICODE implementation, the strings being handled as *wchar_t[]*). A C/C++ source including *winscard.h* will use either implementation depending on whether *_UNICODE* is defined or not.

```
BOOL print_readers(void)
{
  SCARDCONTEXT hContext;
  char *pReader, *szReaders = NULL;
  DWORD dwReadersSz = 1024;
  int cReaders = 0;
  LONG rc;

  rc = SCardEstablishContext(SCARD_SCOPE_SYSTEM, NULL, NULL, &hContext);
  if (rc != SCARD_S_SUCCESS)
  {
    printf("SCardEstablishContext error %08lX\n", rc);
    return FALSE;
  }

  szReaders = calloc(dwReadersSz, sizeof(char));
  if (szReaders == NULL)
  {
    printf("Allocation failed\n");
    goto failed;
  }

  rc = SCardListReaders(hContext,
                        NULL,           /* Any group */
                        szReaders,
                        &dwReadersSz);
  if (rc != SCARD_S_SUCCESS)
  {
    printf("SCardListReaders error %08lX\n",rc);
    goto failed;
  }

  /* Iterate through the readers */
  pReader = szReaders;
  while (*pReader != '\0')              /* End of array reached */
  {
    printf("Reader %d: %s\n", pReader); /* Print the reader name */
    cReaders++;                         /* Increment count of readers */
    pReader += strlen(pReader) + 1;     /* Next string in multi-string array */
  }

  /* Done, success */
  free(szReaders);
  SCardReleaseContext(hContext);
  return TRUE;

failed:
  /* Done, error */
  if (szReaders != NULL) free(szReaders);
  SCardReleaseContext(hContext);
  return FALSE;
}
```

Illustration 18: A minimal C function to list the PC/SC couplers

5.4. Is there a card in the coupler?

Use the *SCardGetStatusChange* function to read the status of the coupler.

```
LONG SCardGetStatusChange(IN SCARDCONTEXT hContext,
                          IN DWORD dwTimeout,
                          IN OUT LPSCARD_READERSTATE rgReaderState,
                          IN DWORD cReaders);
```

Where:

- *hContext* is the pointer you got as output to *SCardEstablishContext*,
- *dwTimeout* is the time to wait for an event (in milliseconds). In this part of the document, we set it to 0 (return immediately),

SCardGetStatusChange could be used in two modes: non-blocking (query only), with the *dwTimeout* parameter set to 0, or blocking until an event occurs is *dwTimeout* is non-0. The later will be shown in 7.2 “Using background threads”.

In this chapter, we remain very basic: the user has a button to click to initiate the card transaction, and we use *SCardGetStatusChange* to verify that there is actually a card in the coupler only when he clicks this button.

- *rgReaderState* is an array of *SCARD_READERSTATE* structures – one entry per coupler you want to monitor, i.e. only one entry when you are working with one coupler only. On input, you write into the structure the name of the coupler you are working with and its assumed current status (“don’t know” is a good assumption on first call!). On output, the structure contains the actual status of the coupler (no card, card present, card in use…). If there is a card in the coupler, the structure also provides its ATR.
- *chReader* is the number of entries in the *rgReaderState* array. With one coupler only, we set it to 1.

Things will be clearer with an example: see illustration 19 on next page.

`javax.smartcardio` implements *SCardGetStatusChange* as *CardTerminal.isCardPresent()* (no wait) or *CardTerminal.waitForCardAbsent(long timeout)* (blocking).

`SpringCard.PCSC` implements *SCardGetStatusChange* as *SCardReader.CardPresent*. The algorithm “is there a card available in the coupler” that appears in illustration 19 is implemented as *SCardReader.CardAvailable*.

```
BOOL is_card_available(SCARDCONTEXT *hContext, const char *reader_name)
{
  SCARD_READERSTATE reader_state;
  LONG rc;

  memset(&reader_state, 0, sizeof(reader_state));
  reader_state.szReader = reader_name;
  reader_state.dwCurrentState = SCARD_STATE_UNAWARE; /* "Don't know" */

  rc = SCardGetStatusChange(hContext, 0, &reader_state, 1);
  if (rc != SCARD_S_SUCCESS)
  {
    printf("SCardGetStatusChange error %08lX\n", rc);
    return FALSE;
  }

  /* Status indicating an error */
  if (reader_state.dwEventState & SCARD_STATE_IGNORE)
  {
    printf("Ooops, the reader must be ignored?\n");
  } else
  if (reader_state.dwEventState & SCARD_STATE_UNKNOWN)
  {
    printf("Ooops, the reader doesn't exists?\n");
  } else
  if (reader_state.dwEventState & SCARD_STATE_UNAVAILABLE)
  {
    printf("Ooops, the reader has just been removed?\n");
  } else
  /* Normal status */
  if (reader_state.dwEventState & SCARD_STATE_PRESENT)
  {
    printf("There is a card in the reader\n");
    if (reader_state.dwEventState & SCARD_STATE_MUTE)
    {
      printf("The card is mute\n");
    } else
    if (reader_state.dwEventState & SCARD_STATE_IN_USE)
    {
      printf("The card is already used by another application\n");
    } else
    {
      printf("The card is available!\n");
      return TRUE;
    }
  } else
  {
    printf("No card in the reader\n");
  }
  return FALSE;
}
```

Illustration 19: Basic implementation of SCardGetStatusChange

5.5. Connect to the card

The *SCardConnect* function opens a logical communication channel between the application and the card contained by a specific coupler.

```
LONG SCardConnect(IN SCARDCONTEXT hContext,
                  IN LPCTSTR szReader,
                  IN DWORD dwShareMode,
                  IN DWORD dwPreferredProtocols,
                  OUT LPSCARDHANDLE phCard,
                  OUT DWORD pdwActiveProtocol);
```

Where:

- *hContext* is the pointer you got as output to *SCardEstablishContext*,
- *szReader* is the name of the coupler,
- *dwShareMode* shall be set to *SCARD_SHARE_EXCLUSIVE* for convenience,
- *dwPreferredProtocols* shall be set to SCARD_PROTOCOL_T0| SCARD_PROTOCOL_T1 (logical OR) to let the coupler (or driver) decide which protocol is the best for card communication,
- *phCard* is a pointer to a *SCARDHANDLE*, that will hold the logical channel's handle after a successful call,
- *pdwActiveProtocol* is a pointer to a DWORD that will tell which protocol has been selected.

SCARDHANDLE is an opaque handle. With a non-C/C++ language, use a generic pointer (*System.IntPtr* on .NET). This makes *phCard* a pointer to a generic pointer (*ref System.IntPtr* on .NET).

In a multi-threaded application, it is a bad idea to share the *hCard* handle among different threads.

If the connection channel has been successfully opened, the function returns *SCARD_S_SUCCESS; *phCard* is populated with the channel's handle, and *pdwActiveProtocol* is populated with the transport protocol that has been selected by the coupler (or driver). It is either

- *SCARD_PROTOCOL_T0* : for ISO/IEC 7816-3 T=0
- *SCARD_PROTOCOL_T1 :* for ISO/IEC 7816-3 T=1, ISO/IEC 14443-4 (because the contactless block protocol "T=CL" is very close to T=1), and for all wired-logic contactless cards (because the coupler's embedded APDU- Processor behaves as a T=1 card).

If there is no card in the coupler, the function returns *SCARD_E_NO_SMARTCARD*. If the card is already used by another application, it returns

SCARD_E_SHARING_VIOLATION. Of course other error codes are also possible in case a technical error occurs.

Refer to paragraph 7.4 for a few information regarding the other possible error codes and their understanding.

`javax.smartcardio` implements *SCardConnect* and the *hCard* as a *Card* object, returned by the *CardTerminal.connect(String protocol)* method, and the subsequent communication channel as a *CardChannel* channel..

`SpringCard.PCSC` implements the communication channel as a *SCardChannel* object, the implementation of *SCardConnect* being the *SCardChannel.Connect()* method.

5.6. Send commands to the card – and receive its responses

The *SCardTransmit* function sends a command (C-APDU) to the smartcard, expecting to receive back a response (R-APDU).

```
LONG SCardTransmit(IN SCARDHANDLE hCard,
                   IN LPCSCARD_IO_REQUEST pioSendPci,
                   IN LPCBYTE pbSendBuffer,
                   IN DWORD cbSendLength,
                   OUT LPSCARD_IO_REQUEST pioRecvPci,
                   OUT LPBYTE pbRecvBuffer,
                   IN OUT LPDWORD pcbRecvLength);
```

Where:

- *hCard* is the handle to the card channel that you got as output to *SCardConnect*,
- *pbSendBuffer* is the C-APDU, an array of bytes,
- *cbSendLength* is the number of bytes of *pbSendBuffer*,
- *pbRecvBuffer* is the R-APDU, an array of bytes, that must be allocated by the caller,
- *pcbRecvLength* is a pointer to the length of the *pbRecvBuffer* array. On input, the caller application sets the pointed value to the size of the array. On output, the library tells how many bytes have been returned by the card (the function returns *SCARD_E_INSUFFICIENT_BUFFER* if the response does not fit in the array).

You must have noticed that we have set apart the *pioSendPci* and *pioRecvPci* parameters. They are mandatory parameters, but not really significant for the developer. Use *SCARD_PCI_T0* if the card uses the T=0 protocol, or *SCARD_PCI_T1* for T=1 (check the value returned in *pdwActiveProtocol* by *SCardConnect*, to know the protocol cf paragraph 5.5).

Non-C/C++ languages are not able to access directly the *SCARD_PCI_T0* and *SCARD_PCI_T1* symbols, that are defined as *extern* variables in *winscard.h*, and resolved at link-time to some symbols coming from the PC/SC dynamically loadable library.

Instead, the non-C/C++ language must use a specific system call to get explicitly the address of these symbols at run-time (*GetProcAddress* on Windows, *dlsym* on UNIX systems).

`javax.smartcardio` implements *SCardTransmit* as *CardChannel.transmit(...)* method.

`SpringCard.PCSC` implements *SCardTransmit* as *SCardChannel.Transmit(...)* method.

We use the (...) to denote that, in both languages, there are a few overloads for the *transmit* or *Transmit* methods, allowing the developer to use either raw array of bytes, or higher level C-APDU and R-APDU objects.

5.7. Retrieve a coupler’s (or driver’s) meta-data

Use the *SCardGetAttrib* function to read one of the coupler’s or driver’s meta-data. This is particularly useful to retrieve the coupler’s serial number, as shown in the sample source code.

```
LONG SCardGetAttrib(IN SCARDHANDLE hCard,
                    IN DWORD dwAttrId,
                    OUT LPBYTE pbAttr,
                    IN OUT LPDWORD pcbAttrLength);
```

Where:

- *hCard* is the handle to the card channel that you got as output to *SCardConnect.* Note that you may retrieve the meta-data of a coupler even when there’s no card in it. In this case, you must call *SCardConnect* specifying *dwShareMode=SCARD_SHARE_DIRECT* and *dwPreferredProtocols=0*. The *hCard* that is returned in this case is not the handle of a card channel, but a specific handle giving a direct access to the coupler.
- *dwAttrId* is the attribute you want to get. A complete list of the allowed values is documented by Microsoft. For instance, the symbolic value *SCARD_ATTR_IFD_SERIAL_NO* returns the coupler’s serial number, and *SCARD_ATTR_VENDOR_IFD_VERSION* its firmware version.
- *pbAttr* is the buffer to receive the response; it must be allocated by the caller,
- *pcbAttrLength* is a pointer to the length of the *pbAttr* array. On input, the caller application sets the pointed value to the size of the array. On output, the library tells how many bytes have been returned by the card (the function returns *SCARD_E_INSUFFICIENT_BUFFER* if the response does not fit in the array).

`SpringCard.PCSC` implements this function as *SCardChannel.GetAttrib*. *SCardReader.GetAttrib* is also possible, and creates a temporary connection to the coupler.

5.8. How to retrieve the card’s ATR?

There are three ways to know the ATR of the smartcard that is in the coupler:

1. Call *SCardGetAttrib* with *dwAttrId=SCARD_ATTR_ATR_STRING*. In this case, the *hCard* must be either an actual handle to a card, or a handle giving direct access to the coupler,
2. Call *SCardGetStatusChange* and retrive the ATR in the *SCARD_READERSTATE* structure when the flags say that a card is present (*dwEventState & SCARD_STATE_PRESENT*),
3. Call *SCardStatus* with *hCard* being an actual handle to the card.

5.8.1. *SCardGetAttrib* method

To retrive the card’s ATR, invoke *SCardGetAttrib* as follow:

```
/* This function assumes that hCard has already been set by a */
/* successful call to SCardConnect                            */
BOOL get_card_atr1(SCARDHANDLE hCard, BYTE abAtr[], DWORD dwMaxAtrSz,
                                                    DWORD *dwActualAtrSz)
{
  LONG rc;
  DWORD dwAtrSz = dwMaxAtrSz;

  rc = SCardGetAttrib(hCard,
                      SCARD_ATTR_STRING,
                      abAtr,
                      dwAtrSz);
  if (rc != SCARD_S_SUCCESS)
  {
    printf(" SCardGetAttrib(ATR) error %08lX\n", rc);
    return FALSE;
  }

  if (dwActualAtrSz != NULL)
    *dwActualAtrSz = dwAtrSz;

  return TRUE;
}
```

5.8.2. *SCardGetStatusChange* method

To retrive the card’s ATR, invoke *SCardGetStatusChange* as follow:

```
BOOL get_card_atr2(SCARDCONTEXT *hContext, const char *reader_name,
                                           BYTE abAtr[],
                                           DWORD dwMaxAtrSz,
                                           DWORD *dwActualAtrSz)
{
  SCARD_READERSTATE reader_state;
  LONG rc;

  memset(&reader_state, 0, sizeof(reader_state));
  reader_state.szReader = reader_name;
  reader_state.dwCurrentState = SCARD_STATE_UNAWARE; /* "Don’t know" */

  rc = SCardGetStatusChange(hContext, 0, &reader_state, 1);
  if (rc != SCARD_S_SUCCESS)
  {
    printf("SCardGetStatusChange error %08lX\n", rc);
    return FALSE;
  }

  /* Status indicating an error */
  if (!(reader_state.dwEventState & SCARD_STATE_IGNORE)
    && (reader_state.dwEventState & SCARD_STATE_PRESENT))
  {
    if (!(reader_state.dwEventState & SCARD_STATE_MUTE))
    {
      if ((abAtr != NULL) && (dwMaxAtrSz <=  reader_state.cbAtr))
        memcpy(abAtr, reader_state.rgbAtr, reader_state.cbAtr);
      if (dwActualAtrSz != NULL)
        *dwActualAtrSz =  reader_state.cbAtr;
      return TRUE;
    }
  }
  return FALSE;
}
```

5.8.3. *SCardStatus* method

To retrive the card’s ATR, invoke *SCardGetStatus* as follow:

```
/* This function assumes that hCard has already been set by a */
/* successful call to SCardConnect                            */
BOOL get_card_atr3(SCARDHANDLE hCard, BYTE abAtr[], DWORD dwMaxAtrSz,
                                                    DWORD *dwActualAtrSz)
{
  LONG rc;
  DWORD dwAtrSz = dwMaxAtrSz;

  rc = SCardStatus(hCard,
                   NULL, NULL,

                      abAtr,
                      dwAtrSz);
  if (rc != SCARD_S_SUCCESS)
  {
    printf(" SCardGetAttrib(ATR) error %08lX\n", rc);
    return FALSE;
  }

  if (dwActualAtrSz != NULL)
    *dwActualAtrSz = dwAtrSz;

  return TRUE;
}
```

5.9. Disconnect from the card

The *SCardDisconnect* function closes the logical communication channel that has been opened by *SCardConnect*.

```
LONG SCardDisconnect(IN SCARDHANDLE hCard,
                     IN DWORD dwDisposition);
```

Where:

- *hCard* is the *SCARDHANDLE*, that has been opened by *SCardConnect*. Following the call to *SCardDisconnect*, this handle becomes invalid, and shall not be used anymore;
- *dwDisposition* tells the coupler whether it should leave the card powered (*SCARD_LEAVE_CARD*), reset the card (*SCARD_RESET_CARD*) or unpower the card (*SCARD_UNPOWER_CARD*). In contact couplers used in ATMs or automatic pay stations, it is also possible to drive the coupler's motor to send back the card to the user (*SCARD_EJECT_CARD)* or to swallow it (*SCARD_CONFISCATE_CARD*). This is obviously not possible with a contactless coupler, nor with desktop contact couplers.

Leaving the card in the powered state without resetting it (*SCARD_LEAVE_CARD*) opens potential security vulnerabilities: if the application uses external or mutual authentications, or even only sends a PIN to the card, it gains an access to some of the card's protected resources. This access should always be closed when the application releases the card.

Otherwise, any other – possibly rogue – application may access the card's protected resources too.

For contactless cards *SCARD_UNPOWER_CARD* does not actually removes the power from the card, because the RF field must remain active to detect when the card is physically removed from the proximity of the coupler's antenna, or when another card arrives.

This is not the case for a contact coupler, where the VCC line may actually go inactive, since the removal of the card will be detected by the presence switch.

`javax.smartcardio` implements *SCardDisconnect* through the *Card.disconnect(boolean reset)* method. Setting the *reset* parameter to *true* is the same as *SCARD_RESET_CARD* and to *false* the same as *SCARD_LEAVE_CARD*.

With `SpringCard.PCSC` use the *SCardChannel.Disconnect()* method or one of its overloads.

5.10. Release the PC/SC context

The last step of using the PC/SC API is calling the *SCardReleaseContext* function. This function closes the communication channel with the middleware and ensures that all allocated resources are freed correctly.

In a multi-threaded application, every thread (that calls functions from the PC/SC API) shall close its own context when terminating.

```
LONG SCardReleaseContext(SCARDCONTEXT hContext);
```

In both `javax.smartcardio` and `SpringCard.PCSC`, the *SCardReleaseContext* function is implicitly called when the PC/SC objects are free by the garbage collector.

6. Using contactless cards with PC/SC

6.1. Introduction

By design, the standards and the PC/SC specification make it possible to operate an ISO/IEC 7816-4 smartcard without even noticing whether it used the T=1 contact transport protocol or the "T=CL" contactless transport protocol.

Unfortunately, when the contactless card is not compliant with ISO/IEC 7816-4 – which is obviously the case of RFID labels, most NFC Tags and most entry-level contactless cards – things are not that easy. The coupler's embedded APDU Processor exposes the card's read and write functions through 7816-4 APDUs, but this is not straightforward.

More than that, even the most classical 7816-4 smartcard may require some added precautions or efforts when operated over ISO/IEC 14443.

Indeed, for the user is free to remove the card from the RF field at any moment, the likelihood of a fatal error during the transaction is lots higher in contactless mode than in contact mode. And, even if the transaction goes to the end, there is also a strong probability that the user removes the card and inserts it again – and in most situations this shall not start another transaction[34].

The role of this chapter is to expose a few ideas that could dramatically increase the user experience when correctly implemented by the application in the terminal.

6.2. Connecting to a contactless card

6.2.1. Protocol

We will see in next paragraph (6.4) that all contactless cards, either a wired-logic card or an actual smartcard, has an ATR that tells the PC/SC middleware and stack that the card does support both the T=0 and the T=1 protocols (TD1 and TD2 bytes of the ATR).

Since the card is said to support the two protocols, which one shall be selected by the application when calling *SCardConnect*?

[34] Imagine the situation where you are boarding a bus with a friend and intend to pay the two travels with your contactless transport card. You have to place the card in front of the validator twice to debit two tickets. If you remove the card too early, the validator instructs you present the card again – but it shall not start debiting a third ticket. And if you keep the card in front of the validator, or even if you are shivering in front of it, it shall also not debit a third ticket. In other words, an ideal validator's application should be able to make the difference, only by software, between the insert/remove/insert again sequence and the insert/shiver/shiver/shiver… sequence.

The answer is clear: the application shall always specify to use T=1 only, for (at least) two good reasons:

- The ISO/IEC 14443-4 protocol is very close to T=1, and the card is allowed to return some data after any instruction (there is no difference between case 3 and case 4 APDU: the L_E byte is optional in contactless and GET RESPONSE is not implemented). But if the card is connected in T=0 and returns some data after a case 3 APDU, this will be considered as a fatal protocol violation by the PC/SC middleware, and the answer will be lost.
- The same applies for the coupler's embedded APDU Processor, that is responsible for exposing wired-logic cards through ISO/IEC 7816-4 instructions. It uses the T=1 protocol only and makes no difference between a case 3 and a case 4 APDU, which is considered to be a fatal violation of the T=0 protocol by the PC/SC middleware.

6.2.2. Share mode

There is another parameter that the application is responsible to choose when calling *SCardConnect*: *dwShareMode*. This parameters tells whether the application accepts to share the card with another application, or not.

It is never a good idea to share a smartcard – nor any security-sensitive resource.

If the application uses external or mutual authentications, or even only sends a PIN to the card, it gains an access to some of the card's protected resources. If the application accepts to share the card, any other – possibly rogue – application may access the card's protected resources too.

As a consequence, setting *dwShareMode* to *SCARD_SHARE_EXCLUSIVE* is the normal behavior.

But this has a side effect: if another application is already connected to the card, the call to *SCardConnect* in your application will fail with error *SCARD_E_SHARING_VIOLATION*. And, on a standard Windows computer, the system will always be the first to connect to the card, in order to try and recognize it, with the aim to find it suitable for one of its PC/SC helper libraries or smartcard mini-driver.

Therefore, your application must be designed to recover nicely from a sharing violation, without any annoyance for the user.

6.2.3. Sample code

As a conclusion of the two above paragraphs, a contactless card shall always be connected as shown below:

6.2.3.1. SCardConnect for a contactless card – C example

```
/* This function assumes that hContext has already been set by a */
/* successful call to SCardEstablishContext                       */
BOOL connect(SCARDCONTEXT hContext, char *szReader)
{
  LONG rc;
  DWORD dwProtocol;
  int retry = 5;

  for (;;)
  {
    rc = SCardConnect(hContext,
                      szReader,
                      SCARD_SHARE_EXCLUSIVE,
                      SCARD_PROTOCOL_T1,
                      &hCard,
                      &dwProtocol);

    if (rc == SCARD_S_SUCCESS)
    {
      /* We are connected, great */
      return TRUE;
    }

    if ((rc == SCARD_E_SHARING_VIOLATION) && (retry-- > 0))
    {
      /* The card is already in use, let's wait a little before retrying */
      Sleep(250); // usleep(250000) on Unix
    } else
    {
      break;
    }
  }

  printf("An error has occured");
  return FALSE;
}
```

6.2.3.2. SCardConnect for a contactless card – C# example

```
public SCardChannel channel;

void connect(string readerName)
{
  SCardReader reader = new SCardReader(readerName);

  channel = new SCardChannel(reader);
  channel.ShareMode = SCARD.SHARE_EXCLUSIVE;
  channel.Protocol = SCARD.PROTOCOL_T1;

  int retry = 5;

  for (;;)
  {
    if (channel.Connect())
    {
      /* We are connected, great */
      return;
    }

    if ((channel.LastError == SCARD.E_SHARING_VIOLATION) && (retry-- > 0))
    {
      /* The card is already in use, let's wait a little before retrying */
      System.Threading.Thread.Sleep(250);
    } else
    {
      break;
    }
  }

  throw new Exception("An error has occured");
}
```

6.3. Retrieving the card’s protocol level ID

6.3.1. Motivation

Whatever the low-level protocol in use (ISO/IEC 14443 A or B, ISO/IEC 15693, FeliCa, or any current vendor-specific implementation on top of a 13.56MHz carrier), the card transmits an ID in response to the coupler’s lookup frames. This is the card’s protocol level ID.

- A lot of legacy applications use the protocol level ID of wired-logic contactless cards are their primary source of information. This is a bad design, because the protocol level ID, being (of course) a publicly-readable data, is easy to clone on a card emulator, or even in an other card with “hacking” features[35]. Yet even new applications may have to re-use an existing architecture, whatever their design flaws,
- Even if it is not the primary source of information, the protocol level ID is frequently one of the key data. For instance, the data stored in the card may include a digital signature computed over this ID, or the card’s authentication key(s) may be computed from a root key and this ID,
- There is one more motivation of retrieving the card’s protocol level ID, and maybe it is the most important with the user experience in mind: this ID tells the application whether it is ready to process the same card again, or the user has inserted another. This is the basic feature to be able to restart an interrupted transaction efficiently, or to prevent processing the same card twice.

There are nowadays a lot of inexpensive RFID/NFC “security investigation tools” that allows to emulate virtually any PICC or VICC[36], including cloning the ID and all the data that the card contents.

Designing a secure solution starts by choosing a card chip providing strong security features (at least cryptographic-level authentication and message digest, possibly ciphered communication) and using them correctly.

[35] For instance, it is easy to find on the web so-called “MIFARE Chinese Magic Cards”. The chip is a clone of NXP’s MIFARE Classic IC, but its protocol level ID could be written freely. This card is a scarecrow for a lots of legacy access control systems.

[36] Google for: ProxMark iii and Chameleon RFID.

6.3.2. ID name, length and construction among the standards

The protocol level ID has different length and construction rules among the contactless standards. They are even name differently! This is summarized in table 3.

Standard	Name of the ID	Length	Rules / remark
ISO/IEC 14443 A	UID (Unique ID)	10 B 7 B 4 B	7 and 10-B UIDs start with a 1-B manufacturer code No rule for 4-B UIDs (only a few reserved values)
ISO/IEC 14443 B	PUPI (Pseudo-Unique ID)	4 B	No rule. No assumption shall be made over the uniqueness of the ID
ISO/IEC 15693	UID (Unique ID)	8 B	Ending with a 1-B manufacturer code and constant value $_h$0E *(see the notice below)*.
FeliCa	IDm (Manufacture ID)	8 B	
Innovatron	DIV (Diversifier)	4 B	4 low-order bytes of the 8-B card's serial number

Table 3: Understanding of the protocol level ID for the different standards

The ISO/IEC 15693 standard says that the UID's most significant byte (MSB) is $_h$0E. Then comes the manufacturer code. The same standard shows in another paragraph that the UID is transmitted LSB-first ($_h$0E is the last byte sent by the card).

But the PC/SC standard states that the bytes of the UID shall be understood ***in the order they are transmitted over the RF channel*** (MSB first). Following the PC/SC convention, any PC/SC compliant coupler returns ISO/IEC 15693 UIDs with $_h$0E at the end, not at the beginning.

This is only a matter of convention, but it could be very disturbing for the beginners…

There is a similar problem with ISO/IEC 14443 A short (4 bytes) IDs: both PC/SC and ISO say that the 1st byte (uid_0) comes first. But in all the old MIFARE application notes the inverse convention is used, the last byte (uid_3) being shown first.

Both ISO/IEC 14443 A and ISO/IEC 15693 IDs include a manufacturer code. Table 4 below is an excerpt of the listing of IC Manufacturers, maintained by ISO/IEC JTC1/SC17. The complete listing is available on the SC17 website, under title "Register of IC Manufacturers":

- ISO/IEC JTC1/SC17 public document listing
- Register of IC Manufacturers *(warning: the URL is likely to change everytime a new version is released)*

ICM	IC Manufacturer according to [ISO7816-6]
$_h$02	ST Microelectronics
$_h$04	Philips Semiconductors → NXP
$_h$05	Infineon (formerly Siemens)
$_h$07	Texas Instrument
$_h$12	Inside Technology
$_h$15	Atmel → Microchip
$_h$16	EM Microelectronic Marin

Table 4: List of manufacturer codes (partial)

6.3.3. Are ISO/IEC 14443 A 4-byte UIDs really unique?

When the MIFARE card has been launched in the late 1990s with a 4-byte UID, people were certainly thinking that it would take decades before the 4 billion of possible IDs were exhausted. But it took only a little more than one decade, actually.

Nowadays, applications should treat ISO/IEC 14443 type A 4-byte UID very carefully, because new MIFARE cards are likely to re-use UIDs that have already been issued in the past.

6.3.4. Random IDs

Cards with a random protocol level ID are more and more present in the field.

Firstly, in some countries the concerns regarding the privacy of RFID and related technologies are getting stronger and stronger. In new application designs, all user sensitive data is protected by a mutual authentication and a secure (ciphered) communication channel. Configuring the card with a random protocol level ID suppresses the last "publicly readable" data from the card.

Secondly, an increasing number of smartphones feature the NFC card emulation mode. Since the smartphone is able to emulate more than one card at once (payment, transport, access control…) there is technically speaking no constant protocol level ID to be exposed to the couplers. Given the privacy concerns, it has been chosen to implement only random IDs in all today's smartphones.

In ISO/IEC 14443 A, a random ID is a 4-B UID with the first byte set to $_h$08.

In ISO/IEC 14443 B, any PUPI value could potentially be random.

The standards say that the card shall not change its ID while the RF field remain active (in other words, the card should not generate a new random ID until it is explicitly taken away from the coupler, and inserted again). Unfortunately, due to the intrinsic weakness of the communication link – including the movements from the user – it is frequent that the coupler "sees" a card changing its ID after a communication error. In the case of a smartphone, it may as well change its protocol (from ISO/IEC 14443 A to B, or from B to A) yet expose the very same applications and data.

Therefore, the application in the terminal must be adapted to identify the card based on its private data, and not to rely on the protocol level data anymore.

6.3.5. The GET DATA (ID) instruction

The coupler's embedded APDU Processor handles all instructions with the **CLA** byte set to $_h$**FF**. Using the **INS =** $_h$**CA** (GET DATA) and **P1,P2 =** $_h$**0000** it is possible to retrieve the card's protocol level identifier.

6.3.5.1. GET DATA (ID) – C example

```
/* The hCard handle has been connected in § 6.2.3.1 */

const BYTE get_data_command[] = { 0xFF, 0xCA, 0x00, 0x00, 0x00 };
BYTE response[256];
DWORD length = sizeof(response);
LONG rc;

rc = SCardTransmit(hCard,
                   SCARD_PCI_T1,
                   get_data_command,
                   sizeof(get_data_command),
                   NULL,
                   response,
                   &length);

if (rc != SCARD_S_SUCCESS)
  /* ... an error has occured - exit here */

if (length < 2)
  /* ... response is not correctly formatted (no SW1 SW2) - exit here */

if ((response[length-2] != 0x90) || (response[length-1] != 0x00))
  /* ... response denotes an error (SW1 SW2 != 90 00) - exit here */

printf("ID=");
for (int i=0; i<length-2; i++)
  printf("%02X ", response[i]);
printf("\n");
```

6.3.5.2. GET DATA (ID) – C# example

```
/* The channel object has been created and connected in § 6.2.3.2 */

RAPDU response = channel.Transmit(new CAPDU(0xFF, 0xCA, 0x00, 0x00, 0x00));

if ((response == null) || (response.data == null))
  throw new Exception("An error has occured");

Console.WriteLine("ID=" + response.data.AsString(" "));
```

6.4. Recognizing the contactless card type

Some applications have to process different types of cards. Their first step is to recognize the card family, before branching to the appropriate workflow.

There are 2 ways of recognizing the card family:

- Recognize, or analyze the card's ATR. The ATR is the first frame sent over the serial line for a contact card, but for a contactless card, the ATR is a "virtual" frame that has to be constructed by the coupler; how is this ATR constructed is an important part of the PC/SC specification. Retrieving the ATR has already been dealt with in § 5.8. The understanding of the ATR's data is the subject of the next paragraphs 6.4.1 and 6.4.2,
- Connect to the card at first (*SCardConnect*) and reads its protocol data by sending the appropriate instruction to the embedded APDU Processor – and processing its response. This is the subject of paragraph 6.4.3.

6.4.1. The ATR of a wired-logic contactless card

The ATR of a wired-logic contactless card (ISO/IEC 14443 below layer 4, ISO/IEC 15693, etc) is constructed as follow:

Byte #	Name	Value	Description
0	TS	$_h$3B	Direct convention
1	T0	$_h$8F	Higher nibble 8 means: no TA1, no TB1, no TC1. TD1 to follow Lower nibble is the number of historical bytes (15)
2	TD1	$_h$80	Higher nibble 8 means: no TA2, no TB2, no TC2. TD2 to follow Lower nibble 0 means: protocol T=0
3	TD2	$_h$01	Higher nibble 8 means: no TA3, no TB3, no TC3, no TD3 Lower nibble 1 means: protocol T=1
4	Historical Bytes	$_h$80	
5		$_h$4F	Application identifier presence indicator
6		$_h$0C	Length to follow (12 bytes)
7		$_h$A0	Registered Application Provider Identifier A0 00 00 03 06 is for PC/SC workgroup
8		$_h$00	
9		$_h$00	
10		$_h$03	
11		$_h$06	
12			PIX.SS: Card protocol (table 6)
13			PIX.NN: Card name (table 7)
14			
15		$_h$00	RFU
16		$_h$00	
17		$_h$00	
18		$_h$00	
19	TCK		Checksum (XOR of bytes 1 to 18)

Table 5: ATR of a wired-logic contactless card

The application could use PIX.SS and PIX.NN to recognize the card family – with some precautions.

6.4.1.1. Protocol

The underlying standard is provided in the PIX.SS byte.

b7	b6	b5	b4	b3	b2	b1	b0	Value	Description
0	0	0	0	0	0	0	0	$_h$00	No information given
0	0	0	0	0	0	0	1	$_h$01	ISO/IEC 14443 A, level 1
0	0	0	0	0	0	1	0	$_h$02	ISO/IEC 14443 A, level 2
0	0	0	0	0	0	1	1	$_h$03	ISO/IEC 14443 A, level 3 or 4 (and Mifare) ISO/IEC 18092 @ 106 kbit/s “NFC-A”
0	0	0	0	0	1	0	1	$_h$05	ISO/IEC 14443 B, level 1
0	0	0	0	0	1	1	0	$_h$06	ISO/IEC 14443 B, level 2
0	0	0	0	0	1	1	1	$_h$07	ISO/IEC 14443 B, level 3 or 4
0	0	0	0	1	0	0	1	$_h$09	ISO/IEC 15693, level 2
0	0	0	0	1	0	1	1	$_h$0B	ISO/IEC 15693, level 3
0	0	0	1	0	0	0	1	$_h$11	JIS:X6319-4 (and FeliCa) ISO/IEC 18092 @ 212 or 424 kbit/s “NFC-F”

Table 6: Values for the ATR's PIX.SS byte

6.4.1.2. Card name

The card name is provided in the PIX.NN word.

The PIX.NN is not a technical data coming from the card, but a data constructed by the coupler based on the few information it could gather from the card. It is not always possible to recognize one card from a card that provides more-or-less the same features (typical example is NFC Forum type 2 Tags). More than that, new cards appear on the market regularly, and a coupler running an “old” firmware is likely to identify it as an older card.

PIX.NN	Card name
$_h$00 $_h$00	Unrecognised card[37]
$_h$00 $_h$01	NXP Mifare Classic 1k
$_h$00 $_h$02	NXP Mifare Classic 4k

[37] By default, SpringCard Couplers don’t use the “Unrecognised card” value for PIX.NN, but one of the proprietary values defined at the end of the table. It is possible to revert to the PC/SC-defined mode by loading a custom configuration in the Coupler.

PIX.NN	Card name
$_{h}$00 $_{h}$03	NXP Mifare UltraLight NFC Forum Type 2 Tag with a capacity <= 64 bytes
$_{h}$00 $_{h}$06	ST Micro Electronics SR176
$_{h}$00 $_{h}$07	ST Micro Electronics SRI4K, SRIX4K, SRIX512, SRI512, SRT512
$_{h}$00 $_{h}$0A	Atmel AT88SC0808CRF
$_{h}$00 $_{h}$0B	Atmel AT88SC1616CRF
$_{h}$00 $_{h}$0C	Atmel AT88SC3216CRF
$_{h}$00 $_{h}$0D	Atmel AT88SC6416CRF
$_{h}$00 $_{h}$12	Texas Instruments TAG IT
$_{h}$00 $_{h}$13	ST Micro Electronics LRI512
$_{h}$00 $_{h}$14	NXP ICODE SLI
$_{h}$00 $_{h}$17	Inside Secure PicoPass 2K
$_{h}$00 $_{h}$18	Inside Secure PicoPass 2KS
$_{h}$00 $_{h}$19	Inside Secure PicoPass 16K
$_{h}$00 $_{h}$1A	Inside Secure PicoPass 16KS
$_{h}$00 $_{h}$1B	Inside Secure PicoPass 16K (8x2)
$_{h}$00 $_{h}$1C	Inside Secure PicoPass 16KS (8x2)
$_{h}$00 $_{h}$1D	Inside Secure PicoPass 32KS (16+16)
$_{h}$00 $_{h}$1E	Inside Secure PicoPass 32KS (16+8x2)
$_{h}$00 $_{h}$1F	Inside Secure PicoPass 32KS (8x2+16)
$_{h}$00 $_{h}$20	Inside Secure PicoPass 32KS (8x2+8x2)
$_{h}$00 $_{h}$21	ST Micro Electronics LRI64
$_{h}$00 $_{h}$24	ST Micro Electronics LR12
$_{h}$00 $_{h}$25	ST Micro Electronics LRI128
$_{h}$00 $_{h}$26	NXP Mifare Mini
$_{h}$00 $_{h}$2F	Broadcom Jewel
$_{h}$00 $_{h}$30	Broadcom Topaz NFC Forum Type 1 Tag
$_{h}$00 $_{h}$34	Atmel AT88RF04C
$_{h}$00 $_{h}$35	NXP ICODE SL2
$_{h}$00 $_{h}$36	NXP Mifare Plus 2K SL1

PIX.NN	Card name
$_h$00 $_h$37	NXP Mifare Plus 4K SL1
$_h$00 $_h$38	NXP Mifare Plus 2K SL2
$_h$00 $_h$39	NXP Mifare Plus 4K SL2
$_h$00 $_h$3A	NXP Mifare UltraLight C, NXP NTAG NFC Forum Type 2 Tag with a capacity > 64 bytes
$_h$00 $_h$3B	FeliCa NFC Forum Type 3 Tag
$_h$00 $_h$3D	NXP Mifare UltraLight EV1
The values below are specific to SpringCard (not in the PC/SC specification)	
$_h$FF $_h$A0	Generic/unknown 14443-A card
$_h$FF $_h$A1	ThinFilm RF Barcode
$_h$FF $_h$B0	Generic/unknown 14443-B card
$_h$FF $_h$B1	ASK CTS 256B
$_h$FF $_h$B2	ASK CTS 512B
$_h$FF $_h$B3	ST Micro Electronics SRI 4K
$_h$FF $_h$B4	ST Micro Electronics SRI X512
$_h$FF $_h$B5	ST Micro Electronics SRI 512
$_h$FF $_h$B6	ST Micro Electronics SRT 512
$_h$FF $_h$B7	Inside Contactless PicoTag/PicoPass
$_h$FF $_h$B8	Generic Atmel AT88SC / AT88RF card
$_h$FF $_h$C0	Calypso card using the Innovatron protocol
$_h$FF $_h$D0	Generic ISO/IEC 15693 from unknown manufacturer
$_h$FF $_h$D1	Generic ISO/IEC 15693 from EM Marin
$_h$FF $_h$D2	Generic ISO/IEC 15693 from ST Micro Electronics, block number on 8 bits
$_h$FF $_h$D3	Generic ISO/IEC 15693 from ST Micro Electronics, block number on 16 bits
$_h$FF $_h$D5	Generic ISO/IEC 15693 from Infineon
$_h$FF $_h$D6	EM MicroElectronic Marin EM4134 chip
$_h$FF $_h$FF	Virtual card (test only)

Table 7: Values for the ATR's PIX.NN word

6.4.2. The ATR of a contactless smartcard

The ATR of a contactless smartcard (ISO/IEC 14443 up to layer 4) is constructed as follow:

Byte #	Name	Value	Description
0	TS	$_h$3B	Direct convention
1	T0	$_h$8n	Higher nibble 8 means: no TA1, no TB1, no TC1. TD1 to follow Lower nibble (n) is the number of historical bytes (0 to 15)
2	TD1	$_h$80	Higher nibble 8 means: no TA2, no TB2, no TC2. TD2 to follow Lower nibble 0 means: protocol T=0
3	TD2	$_h$01	Higher nibble 8 means: no TA3, no TB3, no TC3, no TD3 Lower nibble 1 means: protocol T=1
4 to 3+n	Historical Bytes	$_h$80	Historical bytes, providing the protocol data returned by the PICC during activation: • ISO/IEC 14443-4 type A: the historical bytes from the ATS response • ISO/IEC 14443-4 type B: the historical bytes from the ATTRIB response
4+n	TCK		Checksum (XOR of bytes 1 to 3+n)

Table 8: ATR of a contactless smartcard

It shall be noticed that there is no explicit flag saying that the ATR actually belongs to a contactless smartcard. It is absolutely possible for a contact card to have the very same ATR as a contactless smartcard, yet the application is able to know, or at least to guess (from the coupler name or even from the hardware context) whether the smartcard is contact or contactless.

6.4.3. Obtaining technical data through the GET DATA instruction

This part is specific to SpringCard (not in the PC/SC specification)

The coupler's embedded APDU Processor handles all instructions with the **CLA** byte set to $_{h}$**FF**. Using the **INS** = $_{h}$**CA** (GET DATA) it is possible to retrieve most of the technical information used by the coupler to communicate with the card (including the card's serial number as shown in § 6.3).

The table below lists the P1,P2 values that are supported by SpringCard couplers.

CLA	INS	P1	P2	L_E	Returns
$_{h}$FF	$_{h}$CA	$_{h}$F1	$_{h}$00	$_{h}$00	3 bytes: **PIX.SS** (byte 0) and **PIX.NN** (bytes 1 and 2)
$_{h}$FF	$_{h}$CA	$_{h}$F1	$_{h}$01	$_{h}$00	1 byte: NFC Forum Tag type: - $_{h}$00 → not compliant with any NFC Forum Tag specification - $_{h}$01 → compliant with NFC Forum Type 1 Tag at protocol level - $_{h}$02 → compliant with NFC Forum Type 2 Tag at protocol level - $_{h}$03 → compliant with NFC Forum Type 3 Tag at protocol level - $_{h}$05 → compliant with NFC Forum Type 5 Tag at protocol level NB: Type 4 Tags are ISO/IEC 7816-4 smartcards, compliance should be tested at application level
$_{h}$FF	$_{h}$CA	$_{h}$FC	$_{h}$00	$_{h}$00	ISO/IEC 14443 communication indexes on 2 bytes: **DSI** (byte 0) and **DRI** (byte 1)
$_{h}$FF	$_{h}$CA	$_{h}$FC	$_{h}$01	$_{h}$00	Actual communication bitrate card → coupler (2 bytes, MSB first, expressed in kbit/s)
$_{h}$FF	$_{h}$CA	$_{h}$FC	$_{h}$02	$_{h}$00	Actual communication bitrate coupler → card (2 bytes, MSB first, expressed in kbit/s)

Table 9: GET DATA: P1,P2 specific values

6.5. Exchanging APDUs with a contactless smartcard

Exchanging APDUs with smartcards is the basic feature of PC/SC and is done using the *SCardTransmit* instruction (§ 5.6).

There is not much to add here, provided that the card's commands (and responses) are compliant with the ISO/IEC 7816-4 format (CLA, INS, P1, P2, Data for command, Data, SW1, SW2 for response). Otherwise, the coupler and the card will be perfectly able to communicate, but the PC/SC middleware is likely to report a protocol violation. Paragraph 6.5.2 shows how to overcome this limitation.

6.5.1. Case of a smartcard fully compliant with ISO/IEC 7816-4

This is the standard situation; there is no difference between a contactless smartcard and a contact smartcard. The following examples show a SELECT APPLICATION over the NFC Forum type 4 Tag application.

6.5.1.1. APDU exchange – C example

```
/* The hCard handle has been connected in § 6.2.3.1 */

const BYTE select_nfc_app_command[] = {
  0x00, 0xA4, 0x04, 0x00, 0x07, /* CLA, INS=SELECT, P1, P2, Lc=7 */
  0xD2, 0x76, 0x00, 0x00, 0x85, 0x01, 0x01, /* Application name */
  0x00 /* Le */
};
BYTE response[256];
DWORD length = sizeof(response);
LONG rc;

rc = SCardTransmit(hCard,
                   SCARD_PCI_T1,
                   select_nfc_app_command,
                   sizeof(select_nfc_app_command),
                   NULL,
                   response,
                   &length);

if (rc != SCARD_S_SUCCESS)
  /* ... an error has occured – exit here */

if (length < 2)
  /* ... response is not correctly formatted (no SW1 SW2) – exit here */

if ((response[length-2] != 0x90) || (response[length-1] != 0x00))
  /* ... response denotes an error (SW1 SW2 != 90 00) – exit here */

printf("NFC Forum type 4 Tag application selected!\n");
```

6.5.1.2. APDU exchange – C# example

```
/* The channel object has been created and connected in § 6.2.3.2 */

CardBuffer application_id = new CardBuffer("D2760000850101");
RAPDU response = channel.Transmit(new CAPDU(0x00, 0xA4, 0x04, 0x00,
                                            application_id.GetBytes(),
                                            0x00));

if (response == null)
  throw new Exception("An error has occured");

if (response.SW != 0x9000)
  throw new Exception("Status word is not 90 00");

Console.WriteLine("NFC Forum type 4 Tag application selected!");
```

6.5.2. Case of a smartcard having a custom APDU format

A few contactless smartcards, such as the NXP MIFARE Plus and the early version of the NXP DESFire, use a vendor-specific format for commands and responses on top of the ISO/IEC 14443-4 transport protocol.

For instance, the DESFire “GET VERSION” command is documented as follow:

Terminal		Card
$_h$60	→	
	←	$_h$AF *<1st part of the version string>*
$_h$AF	→	
	←	$_h$AF *<2nd part of the version string>*
$_h$AF	→	
	←	$_h$00 *<3rd (and last) part of the version string>*

Table 10: The DESFire GET VERSION command

This exchange is not compliant with ISO/IEC 7816-4 because:

- The command is on one byte only (and not CLA, INS, etc)
- The response does not end with a status word (SW1, SW2) with SW1 equal to either $_h$6x or $_h$9x.

Therefore, the command shall be encapsulated in a standard-compliant APDU, processed by the coupler’s embedded APDU Processor. In turn, the response will be encapsulated accordingly. Doing so, the PC/SC middleware will not report any protocol violation.

The ENCAPSULATE instruction uses **CLA =** $_h$**FF** and **INS =** $_h$**FE**.

6.5.2.1. Encapsulated APDU exchange – C example

```
/* The hCard handle has been connected in § 6.2.3.1 */

const BYTE desfire_get_version_command[] = {
  0xFF, 0xFE, 0x00, 0x00, 0x01, /* CLA=embedded, INS=ENCAPSULATE, P1,P2, Lc=1 */
  0x60, /* Desfire get version */
  0x00 /* Le */
};
const BYTE desfire_next_part_command[] = {
  0xFF, 0xFE, 0x00, 0x00, 0x01, /* CLA=embedded, INS=ENCAPSULATE, P1,P2, Lc=1 */
  0xAF, /* Desfire get next part */
  0x00 /* Le */
};

BYTE version[256];
DWORD version_length = 0;

BYTE response[256];
DWORD length;
LONG rc;

/* First part */
/* ---------- */

length = sizeof(response);
rc = SCardTransmit(hCard,
                   SCARD_PCI_T1,
                   desfire_get_version_command,
                   sizeof(desfire_get_version_command),
                   NULL,
                   response,
                   &length);

if (rc != SCARD_S_SUCCESS)
  /* ... an error has occured – exit here */
if (length < 2)
  /* ... response is not correctly formatted (no SW1 SW2) – exit here */
if ((response[length-2] != 0x90) || (response[length-1] != 0x00))
  /* ... response denotes an error (SW1 SW2 != 90 00) – exit here */

if (response[0] != 0xAF)
  /* ... the Desfire card has returned an error – exit here */

// TODO: check bounds

memcpy(&version[version_length], &response[1], length – 3);
version_length += length – 3;


                                                                     .../...
```

```
/* Next part */
/* ---------- */

length = sizeof(response);
rc = SCardTransmit(hCard,
                   SCARD_PCI_T1,
                   desfire_next_part_command,
                   sizeof(desfire_get_version_command),
                   NULL,
                   response,
                   &length);

// TODO: Check rc and SW - Same as above

if (response[0] != 0xAF)
  /* ... the Desfire card has returned an error – exit here */

// TODO: check bounds

memcpy(&version[version_length], &response[1], length – 3);
version_length += length – 3;

/* Last part */
/* ---------- */

length = sizeof(response);
rc = SCardTransmit(hCard,
                   SCARD_PCI_T1,
                   desfire_next_part_command,
                   sizeof(desfire_get_version_command),
                   NULL,
                   response,
                   &length);

// TODO: Check rc and SW - Same as above

if (response[0] != 0x00)
  /* ... the Desfire card has returned an error – exit here */

// TODO: check bounds

memcpy(&version[version_length], &response[1], length – 3);
version_length += length – 3;

printf("Desfire version=");
for (int i=0; i<version_length; i++)
  printf("%02X", version[i]);
printf("\n");
```

6.5.2.2. Encapsulated APDU exchange – C# example

```
/* The channel object has been created and connected in § 6.2.3.2 */

CardBuffer raw_cmd1 = new CardBuffer(0x60);
CardBuffer raw_cmd2_cmd3 = new CardBuffer(0xAF);
CardBuffer version = new CardBuffer();
RAPDU response;

response = channel.Transmit(new CAPDU(0xFF, 0xFE, 0x00, 0x00,
                                      raw_cmd1.GetBytes(),
                                      0x00));

if ((response == null) || (response.data == null) || response.SW != 0x9000)
  throw new Exception("Encapsulation or communication error");
if (response.data.GetByte(0) != 0xAF) /* First answer must start with AF */
  throw new Exception("Desfire error");

version.Append(response.data.GetBytes(1, response.data.Length – 1));

response = channel.Transmit(new CAPDU(0xFF, 0xFE, 0x00, 0x00,
                                      raw_cmd2_cmd3.GetBytes(),
                                      0x00));

if ((response == null) || (response.data == null) || response.SW != 0x9000)
  throw new Exception("Encapsulation or communication error");
if (response.data.GetByte(0) != 0xAF) /* Next answer must start with AF */
  throw new Exception("Desfire error");

version.Append(response.data.GetBytes(1, response.data.Length – 1));

response = channel.Transmit(new CAPDU(0xFF, 0xFE, 0x00, 0x00,
                                      raw_cmd2_cmd3.GetBytes(),
                                      0x00));

if ((response == null) || (response.data == null) || response.SW != 0x9000)
  throw new Exception("Encapsulation or communication error");
if (response.data.GetByte(0) != 0x00) /* Last answer must start with 00 */
  throw new Exception("Desfire error");

version.Append(response.data.GetBytes(1, response.data.Length – 1));

Console.WriteLine("Desfire version: " + version.AsString());
```

6.6. The embedded APDU Processor for wired-logic cards

Doc. PMD17182 is the reference guide for the embedded APDU Processor.

The paragraphs below give an overview of the features that are made available to the host application through *SCardTransmit* calls.

6.6.1. Generic wired-logic card read/write instructions

The READ BINARY and UPDATE BINARY instructions map to the READ/WRITE commands of most supported wired-logic cards:

- MIFARE Classic,
- NFC Forum type 2 Tags, MIFARE UltraLight & NTAG families,
- NFC Forum type 1 Tags, Jewel/Topaz cards,
- NFC Forum type 3 Tags,
- ISO/IEC 15693-3, NFC Forum type 5 Tags,
- ...

CLA	INS	P1	P2	L_C/L_E	Data In	Action / returns
$_h$FF	$_h$B0	Address		$_h$00		READ
$_h$FF	$_h$D6	Address		xx	Data	WRITE

Table 11: PC/SC READ/WRITE instructions for wired-logic cards

6.6.2. MIFARE Classic authentication and keys

The PC/SC standard specifies one instruction to get authenticated onto MIFARE Classic cards, and another to manage the keys known by the coupler. It is possible that future products use the same instructions to implement the security of other chips.

CLA	INS	P1	P2	L_C	Data In	Action
$_h$FF	$_h$86	$_h$00	$_h$00	$_h$05	$_H$01 $_h$00 Address, Group, Index	GENERAL AUTHENTICATE
$_h$FF	$_h$82	Group	Index	$_h$06	Key value	LOAD KEY

Table 12: PC/SC implementation of MIFARE Classic security

6.6.3. ISO/IEC 15693-3 instructions

This part is specific to SpringCard (not in the PC/SC specification)

The following instructions exports the complete function set defined by ISO/IEC 15693-3. Note that all the READ and WRITE functions ($_h$20, $_h$21, $_h$23, $_h$24) are used at a higher level by the generic READ and WRITE instructions defined in § 6.6.1.

CLA	INS	P1	P2	L_C/L_E	Data In	Action / returns
$_h$FF	$_h$F6	$_h$20	$_h$00	$_h$01	Address	READ SINGLE BLOCK
$_h$FF	$_h$F6	$_h$21	$_h$00	xx	Address, Data	WRITE SINGLE BLOCK
$_h$FF	$_h$F6	$_h$22	$_h$00	$_h$01	Address	LOCK BLOCK
$_h$FF	$_h$F6	$_h$23	$_h$00	$_h$02	Address, Count	READ MULTIPLE BLOCKS
$_h$FF	$_h$F6	$_h$24	$_h$00	xx	Address, Count, Data	WRITE MULTIPLE BLOCKS
$_h$FF	$_h$F6	$_h$27	$_h$00	$_h$01	AFI	WRITE AFI
$_h$FF	$_h$F6	$_h$28	$_h$00	$_h$00		LOCK AFI
$_h$FF	$_h$F6	$_h$29	$_h$00	$_h$01	DSFID	WRITE DSFID
$_h$FF	$_h$F6	$_h$2A	$_h$00	$_h$00		LOCK DSFID
$_h$FF	$_h$F6	$_h$2B	$_h$00	$_h$00		GET SYSTEM INFORMATION
$_h$FF	$_h$F6	$_h$2C	$_h$00	$_h$02	Address, Count	GET MULTIPLE BLOCKS SECURITY STATUS

Table 13: Embedded APDU Processor: mapping of ISO/IEC 15693-3 instructions

7. Creating efficient and robust PC/SC applications

7.1. Connecting to the right coupler

7.1.1. Coupler names (and the issues behind that)

The PC/SC drivers are responsible for providing a unique name for every coupler. The naming convention is:

`{VendorName} {ProductName [SlotName]} {Number}`

- The **`SlotName`** part is used for multi-interfaces couplers, such as SpringCard CrazyWriter HSP or CSB HSP that have a contactless slot and many contact slots.
- On Microsoft Windows, the **`Number`** part is a trivial counter that prevents two couplers to have the same name. On PCSC-Lite implementation, the *Number* part is in the form **`{BusNumber} {UnitNumber}`** (example: *"SpringCard H663 00 00"*).

There are two frequent issues related to the **SCardListReaders** function call:

1. The naming convention is different from one PC/SC stack to another, and from one PC/SC driver to another.

For instance, SpringCard's USB PC/SC driver for Microsoft Windows enumerates the Prox'N'Roll PC/SC HSP as *"SpringCard Prox'N'Roll Contactless {N}"*.

But this device is also supported by Microsoft's generic USB CCID driver, that names it only "*SpringCard Prox'N'Roll {N}*".

On the other hand, on a Linux or macOS X computer, the open source PCSC-Lite CCID driver names the same device *"SpringCard H663 {NN NN}"*.

2. The number attributed to a coupler by the operating system depends on the order of the plug'n'play enumeration, that depends on the user actions (plug/unplug a coupler) and also behaves differently every time the computer starts.

Suppose you connect two Prox'N'Roll HSP PC/SC to your running computer, one after the other. The first one will be named "*SpringCard Prox'N'Roll Contactless 0*", and the second one "*SpringCard Prox'N'Roll Contactless 1*".

Now reboot the computer. You will have two couplers bearing the same name, but there is a 50% chance that they have been swapped.

As a consequence, any application where the coupler's name is hard-coded is likely to fail on some computers or when another PC/SC device is added to the computer.

To avoid any problem, after calling **SCardListReaders**, the application shall either:

- Always present a list of the currently available couplers to the user (in a drop-down list or so), and let the user decide which coupler he wants to use,
- "Guess" which one is the most suitable coupler and select this coupler automatically (see next paragraph 7.1.2 for a possible strategy),
- Select the coupler based on its serial number (and store this serial number, not the coupler's name, in the application persistent settings). Paragraph 5.7 shows how to retrieve the coupler's serial number,
- Monitor all the couplers, and try to recognize the card(s) that the application supports, based on the ATR. Paragraph 5.8.2 shows how to use *SCardGetStatusChange* to get notified when a card arrives in any coupler and obtain its ATR

7.1.2. Identifying a SpringCard PC/SC contactless coupler

Use the following algorithm to select the first SpringCard PC/SC contactless coupler:

- Create a lower case copy of the coupler's name,
- Does this string starts with *"springcard"*?
  → this is a SpringCard PC/SC coupler
- Does this string contains *"contactless"* or *"nfc"*?
  → this is the contactless slot of the SpringCard PC/SC coupler, on a Windows system and using the driver supplied by SpringCard,
- Does this string ends with *"00"*?
  → this is the first slot, hence the contactless slot, of the SpringCard PC/SC coupler, on a PCSC-Lite system,
- Else:
  → this is the only slot, hence the contactless slot, of the SpringCard PC/SC coupler, on a Windows system and using the driver supplied by Microsoft.

7.2. Using background threads

By default, application code runs in the main thread. Every statement is therefore executed in sequence. If you perform a long lasting operation, the application blocks until the corresponding operation has finished.

Any I/O operation that could potentially take more than a few milliseconds to run should be executed in a background thread, to prevent any “lag” or blocking of the application’s window, which is a very bad user experience.

Although most smartcard-related I/O operations (*SCardConnect*, *SCardTransmit*) will take only a few milliseconds in most situations, there are some situations where they could block up to a few seconds, maybe up to one minute[38].

Therefore, a windowed application should always implement the smartcard-related I/O operations in a background thread.

The need to implement nicely “long lasting” tasks more easily from the developer’s point of view has led to the introduction of thread pools and/or to an asynchronous programming paradigm in modern development languages.

Next version `SpringCard.PCSC` will allow using the *async/await* keywords of the .NET environment.

7.2.1. Monitoring the coupler(s) and card(s) in background

Using *SCardConnect* as shown in paragraph 5.4 “Is there a card in the coupler?” is not very satisfying, nor is it efficient to say the least.

In a naive implementation, it relies on the user repeatedly clicking on a “start” button until the card is there, and the transaction could actually start.

Moving to an unattended implementation means calling *SCardConnect* periodically, from a software timer, until it returns *SCARD_S_SUCCESS* – OK.

But if the timer is too fast, its a pure waste of computer resources – and if it is too slow, a significant delay is introduced between the insertion of the card and the beginning of the transaction.

Fortunately, the PC/SC API offers the *SCardGetStatusChange* function, that, despite the lack of ‘Wait’ in the name, is a blocking call, with a timeout. The caller stops consuming any resource, and is resumed by the middleware only when the coupler fires an event. *SCardConnect* may then be called only when a card has just been inserted.

A few other events are also fired by the middleware to ease the synchronisation between two or more applications: an application waiting on *SCardGetStatusChange*

[38] See examples in § 7.2.3.

is notified when another application gains access to the card, and when the card is made available again.

But as any other blocking function, *SCardGetStatusChange* shall never be called from the application's main thread.

Therefore, a card-aware application using *SCardGetStatusChange* shall always implement this function call in a background thread.

7.2.2. *SCardConnect* "loop" in the background

As we have seen in paragraph 6.2.3, the application must be ready to retry its call to *SCardConnect* if the card is used by another application. This is done by catching the *SCARD_E_SHARING_VIOLATION* error and retrying after a few hundreds of milliseconds.

Therefore, this *SCardConnect* + *Sleep* loop shall be implemented in a background thread for a correct user experience.

7.2.3. *SCardTransmit* in the background

Even if the execution of a typical command by a smartcard is rather fast, there are some situations where it could be dramatically slow:

- In the contactless world, an ISO/IEC 14443-4 smartcard may report that it needs more than 5 seconds to process a command. If the user removes such a card while there is a running command, the coupler will wait for a first 5-second timeout. The coupler will then run its error-recovery algorithm, which involve retrying up to 3 times. As a consequence, such a card will be reported as removed only after 20 seconds or so;
- In the contact world, some (old) cryptographic cards take up to 60 seconds only to generate a new RSA key-pair.

And, more than that, a typical card transaction involves more than one exchanges. Even if every exchange take only 20ms or so, a sequence of 50 exchanges is already 1s in best case.

Therefore, the sequence of *SCardTransmit* that represent a card transaction shall be implemented in a background thread for a correct user experience.

7.3. Recommended flowchart with 1+2 threads

The conclusion to paragraph 7.2 is that we should add two threads to the application's main thread:

- one thread to monitor the coupler, looping around *SCardGetStatusChange*,
- one thread implement the transaction, looping around *SCardTransmit.*

The basic flowchart of paragraph 5.1 should become the more complex flowchart below:

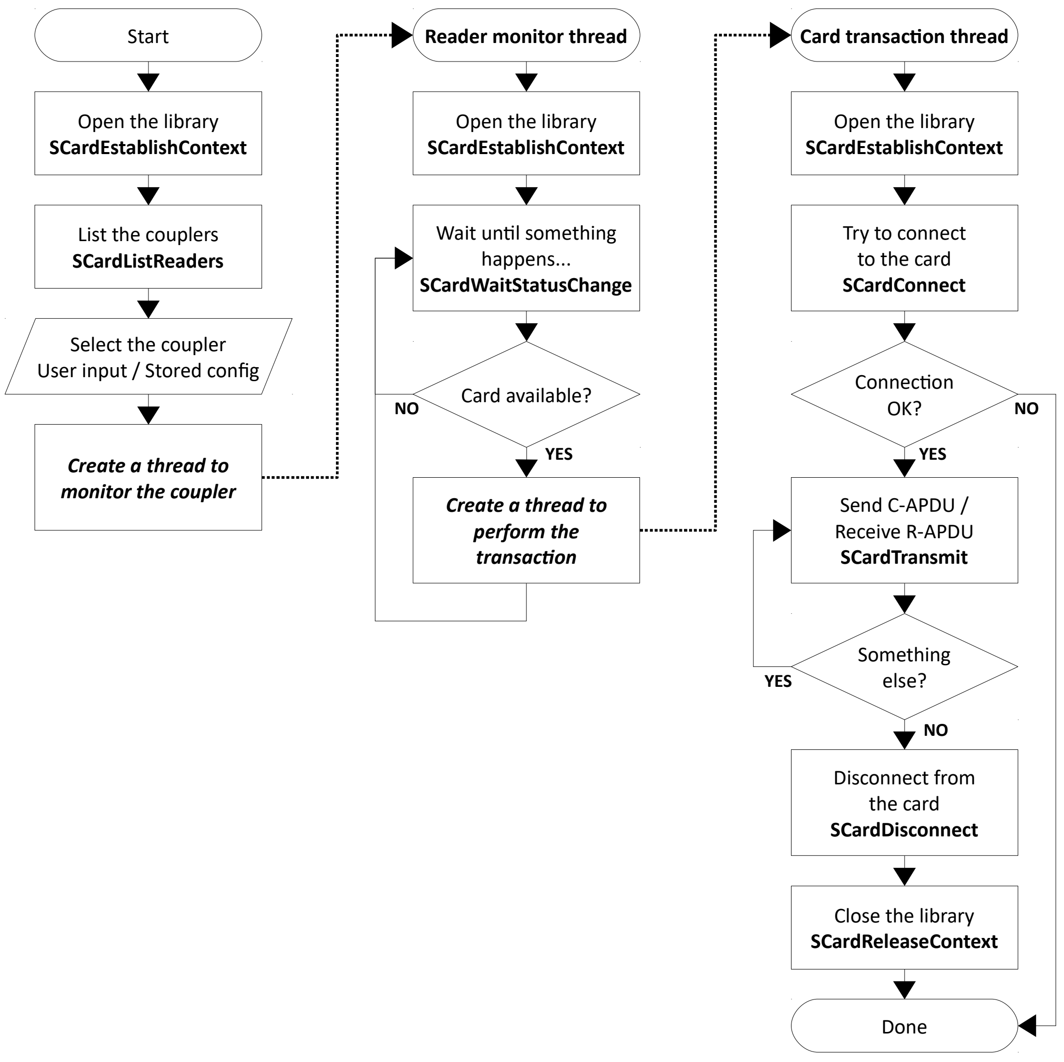


Illustration 20: Flowchart of a PC/SC application with 3 threads

There are a few things worth noticing in this flowchart:

- Every thread gets its own handle to the library by calling *SCardEstablishContext*. This is mandatory in PCSC-Lite, and considered a good practice on Windows too. Otherwise, the application can not stop the background threads individually.
- Actually, the flowchart doesn't show either how the *Reader monitor* thread and the *Card transaction* thread could be stopped. There is a single function to do so: *SCardCancel*. Just call *SCardCancel* from the main thread, providing as single parameter the *hContext* belonging to the thread you want to stop. Any blocking call (*SCardWaitStatusChange*, long *SCardTransmit*) will terminate with error *SCARD_E_CANCELLED*. That is why you should have a single hContext for every thread.
- The flowchart doesn't show either how the *Reader monitor* thread and the *Card transaction* thread interact with the user. This is often the most tricky part in such a windowed-application, because the GUI belongs to the main thread, and the background threads must use callbacks (or messages, or delegates) to have the main thread manipulate the GUI for them. If you are not familiar with this concept, read this page at Microsoft's:

https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/calling-synchronous-methods-asynchronously

7.4. Understanding the errors (and implementing a smart recovery)

The PC/SC API defines a lot of errors. It is not in the aim of this guide to document all of them. Nevertheless, the developer will find in the tables some tips to understand – and avoid – a few frequent mistakes, or to correctly handle "user related" errors.

7.4.1. Errors that should be recovered nicely by the application

Error code	Explanation	Remark/Recommended recovery
SCARD_E_CANCELLED	A blocking call has been canceled	This error is created by another thread of the same application calling *SCardCancel*
SCARD_E_UNKNOWN_READER	Invalid *szReader*	Either the coupler has been removed by the user, or the application's configuration is out of date
SCARD_E_SHARING_VIOLATION	The card (or coupler) is already reserved by another application	Try again later
SCARD_E_NO_SMARTCARD	*SCardConnect* called to an empty coupler	Monitor the coupler Try again later
SCARD_W_REMOVED_CARD (contactless card)	The card has been removed	Try again silently if the card is inserted again Prompt the user to re-insert the card

7.4.2. "User related" errors

Error code	Explanation	Remark/Recommended recovery
SCARD_E_READER_UNAVAILABLE	A coupler has been removed while being monitored by *SCardWaitStatusChange*	Prompt the user to re-plug or to select another
SCARD_E_NO_READERS_AVAILAB LE	There is no coupler (at all) connected to the computer	Prompt the user to plug a coupler
SCARD_W_REMOVED_CARD (contact card)	The card has been removed	Prompt the user to re-insert the card
SCARD_W_UNRESPONSIVE_ CARD (contact card)	The cars is mute	Prompt the user to check the card's orientation and to clean the contact

7.4.3. Errors that shall never occur after the application has been debugged...

Error code	Explanation	Remark
SCARD_E_INSUFFICIENT_BUFFER	*dwRecvLength* too small in *SCardTransmit*	Use 260 B-buffers if the card/coupler supports short APDUs only. Allocate 64 kB-buffers with extended APDUs
SCARD_E_INVALID_HANDLE	*hCard* or *hContext* is invalid	Check the application's flowchart...
ERROR_INVALID_HANDLE (Windows only, code 6)	*hContext* is invalid	Check the application's flowchart...
SCARD_E_PROTO_MISMATCH	Invalid protocol parameter	Always use T=1 for contactless and for contact cards supporting it. Use T=0 only with contact cards not supporting T=1.
SCARD_E_PCI_TOO_SMALL	Wrong *pioSendPci* (or *pioRecvPci*) in *SCardConnect* call	Use SCARD_PCI_T0 for T=0, SCARD_PCI_T1 for T=1
SCARD_W_RESET_CARD	The card has been reset (*SCardDisconnect* or *SCardReconnect* called by this application or another on the same card)	Check the application's flowchart... Don't share the card.

7.4.4. System errors

Error code	Explanation	Action
SCARD_E_NO_SERVICE	Failed to connect to the PC/SC middleware	Check that the PC/SC middleware is correctly installed, and that the user account is allowed to use it
SCARD_E_SERVICE_STOPPED	The PC/SC middleware has been stopped	

8. Smartcard applications without PC/SC

Developing a smartcard-aware application is not limited to high-end operating systems, such as Microsoft Windows, Linux or macOS X. In many OEM integrations, SpringCard couplers are driven by an industrial PLC or maybe a microcontroller unable to run a complete PC/SC stack. There is also the case of Android, where PC/SC is not available on mainstream devices.

More than that, even on a high-end host, the PC/SC driver and middleware are software layers that provide useful services, but at the price of a significant overhead that increases the time it takes to perform a transaction with the card. In such situation, notably when the coupler uses a simple serial or TCP channel to communicate, it could be more efficient to drive the coupler directly from the application. This is of course an added complexity for the developer, but object oriented programming allows to mask this complexity somehow.

This chapter introduces two architectures that are used by SpringCard in different projects as an alternative to a "genuine" PC/SC stack.

8.1. SpringCard zero-driver CCID implementation

Doc. PMD15282 is the reference guide for the zero-driver CCID implementation.

CCID is the USB specification for *Chip Card Interface Devices*. In clear words, it is a recommended protocol for implementers of USB-attached PC/SC couplers.

SpringCard not only uses this protocol in all its USB couplers, but also uses CCID as the foundation of the communication protocols created over other mediums: TCP, serial, and even Bluetooth Smart (also known as BLE, Bluetooth Low Energy).

The CCID protocol introduces a set of commands, that maps immediately into the function offered by the PC/SC API. It is therefore not difficult for a developer with a good understanding of the concepts documented in this guide to develop a smartcard-aware application directly on top of CCID. Table 14 lists the CCID commands, and their mapping into PC/SC functions.

PC/SC function	See paragraph	CCID Command	CCID Response
SCardGetStatusChange	5.4	*PC_To_RDR_GetSlotStatus*[39]	*RDR_To_PC_SlotStatus*
SCardConnect	5.5	*PC_To_RDR_IccPowerOn*	Card absent: *RDR_To_PC_SlotStatus* Card present (returns the ATR): *RDR_To_PC_DataBlock*
SCardTransmit	5.6	*PC_To_RDR_XfrBlock*	*RDR_To_PC_DataBlock*
SCardDisconnect	5.9	*PC_To_RDR_IccPowerOff*	*RDR_To_PC_SlotStatus*
SCardControl		*PC_To_RDR_Escape*	*RDR_To_PC_Escape*

Table 14: CCID commands/responses

`SpringCard.PCSC` for .NET is able to communicate with SpringCard network-attached PC/SC couplers (E663 family) and serial-attached PC/SC couplers (K663 family) without going through a PC/SC driver and stack[40]. Look for namespace `SpringCard.PCSC.ZeroDriver` in the SDK.

8.2. Android lightweight CCID implementation

8.2.1. Motivation

Given the lack of support for PC/SC in mainstream Android-based devices (see paragraph 4.6), and in an effort to address the growing demand to associate a contactless coupler to an Android terminal, SpringCard has developed a pure-Java solution that bridges Android applications with a USB PC/SC coupler (illustration 21, next page).

8.2.2. Technical architecture

This solution relies on a very simplified, 2-tiers architecture:

- A user-mode service, directly available on Google Play, implements the CCID protocol on top of Android’s USB Host API (*android.hardware.usb*),
- An open-source Java library ties the client application to the service. It could be ported as a plug-ins for non-native development.

[39] CCID also features an “interrupt” system, when the computer can be notified of card movements (insert/remove) without having to query the coupler with *PC_To_RDR_GetSlotStatus* periodically.

[40] The E663 family has a PC/SC driver for Windows and PCSC-Lite, but the K663 does not even have such a driver and is always operated in zero-driver mode, most of the time through the legacy SpringProx API.

An introduction to the SpringCard "PC/SC" solution for Android
http://tech.springcard.com/2015/springcard-pcsc-solution-for-android-released/

Please remember that this solution, although bearing the "PC/SC" word in its name, is very far from a standard implementation, yet it is the only solution that does not involve rooting your tablet or mobile phone, or building a custom image of the Android system.

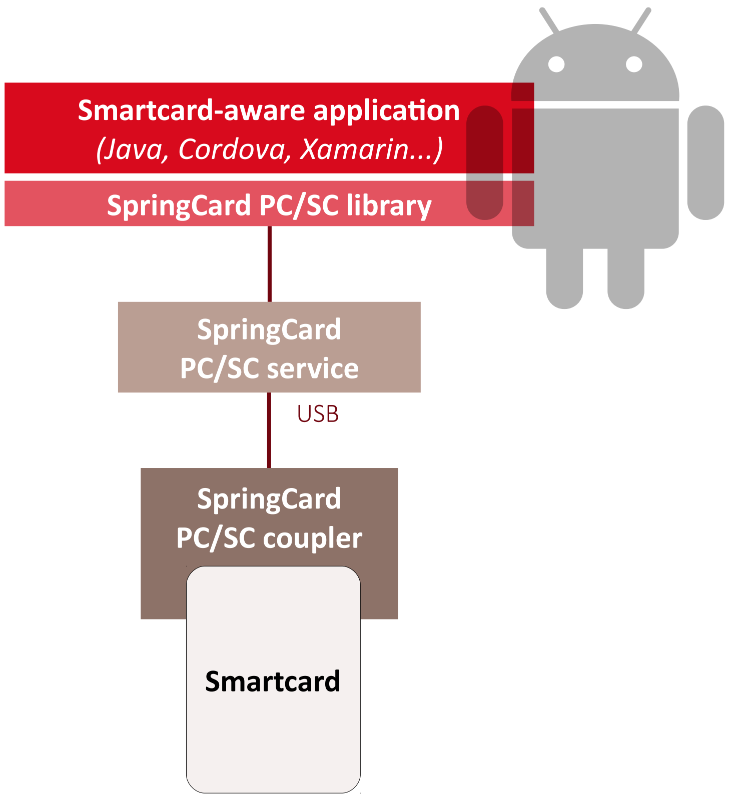


Illustration 21: The lighweight "PS/SC Like" architecture proposed by SpringCard for Android

Illustration 22: A Nexus 9 Android Tablet with a Prox'N'Roll PC/SC HSP

8.2.3. Frequent issues with mainstream Android tablets

There is a frequent issue, one must be really aware off, with Android devices: most of them don't feature a real USB host port (with a type A connector) as it is the case for a PC.

Instead, they have a single, multi-role, connector, primarily designed to be a USB device port (to connect the tablet as a slave device to a computer) and a charging port. And unfortunately, the last role of being a USB host is very often poorly implemented, with blocking issues being related to power supply:

- The Android host may be unable to supply more than 4.3 or 4.4V (the USB standard says it should be 5V),
- The Android host may be unable to provide more than 50 or 100mA (a USB PC/SC coupler typically requires 150 mA @ 5V, with peaks up to 200mA. And @ 4.5V, the current requirement increases to compensate the lower voltage),
- It could be impossible to power the Android tablet from a mains adapter and to use it has a USB host simultaneously.

In other words, if your project involves using an Android tablet with a USB PC/SC coupler, buy a few tablets for testing purposes, and check carefully that everything is fine before going into production stage!

“Terminating a development-related book after exactly 2^7 pages is a kind of fullness!”

LEGAL INFORMATION

Editor

SPRINGCARD SAS au capital de 227 000 €

RCS EVRY B 429 665 482

Parc Gutenberg, 13 voie La Cardon, 91120 Palaiseau

FRANCE

Contact

For more information and to locate our sales office or distributor in your country or area, please visit:

www.springcard.com