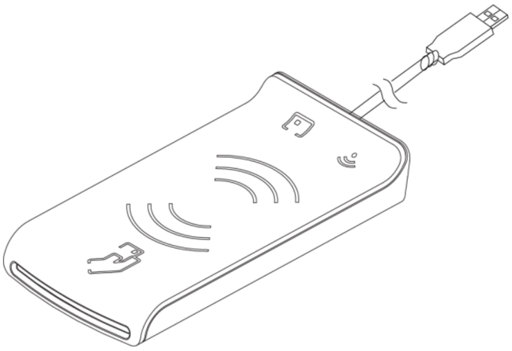
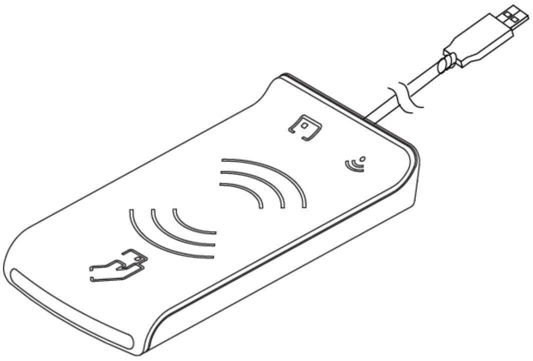
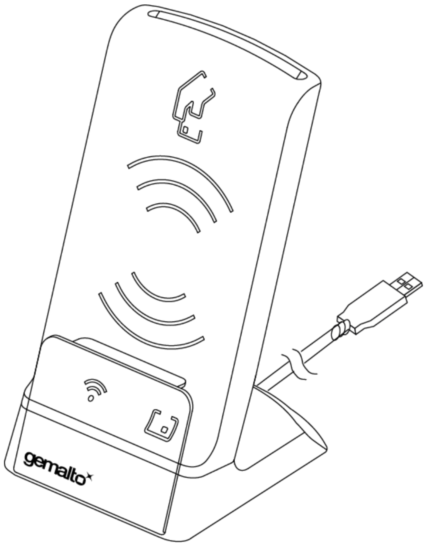


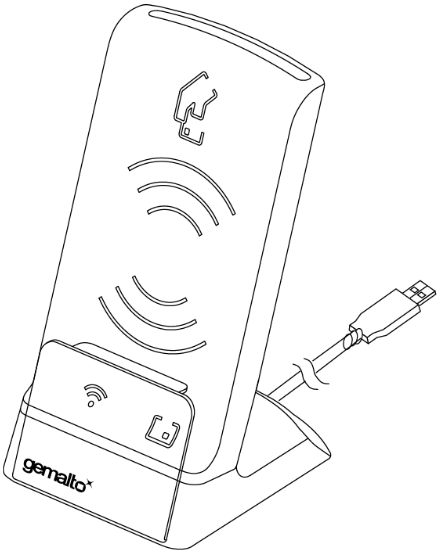


Prox–DU & Prox–SU

Dual interface USB smart card reader

PC/SC Guide





All information herein is either public information or is the property of and owned solely by Gemalto NV. and/or its subsidiaries who shall have and keep the sole right to file patent applications or any other kind of intellectual property protection in connection with such information.

Nothing herein shall be construed as implying or granting to you any rights, by license, grant or otherwise, under any intellectual and/or industrial property rights of or concerning any of Gemalto’s information.

This document can be used for informational, non-commercial, internal and personal use only provided that:

- The copyright notice below, the confidentiality and proprietary legend and this full warning notice appear in all copies.

- This document shall not be posted on any network computer or broadcast in any media and no modification of any part of this document shall be made.

Use for any other purpose is expressly prohibited and may result in severe civil and criminal liabilities.

The information contained in this document is provided “AS IS” without any warranty of any kind. Unless otherwise expressly agreed in writing, Gemalto makes no warranty as to the value or accuracy of information contained herein.

The document could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Furthermore, Gemalto reserves the right to make any change or improvement in the specifications data, information, and the like described herein, at any time.

Gemalto hereby disclaims all warranties and conditions with regard to the information contained herein, including all implied warranties of merchantability, fitness for a particular purpose, title and non-infringement. In no event shall Gemalto be liable, whether in contract, tort or otherwise, for any indirect, special or consequential damages or any damages whatsoever including but not limited to damages resulting from loss of use, data, profits, revenues, or customers, arising out of or in connection with the use or performance of information contained in this document.

Gemalto does not and shall not warrant that this product will be resistant to all possible attacks and shall not incur, and disclaims, any liability in this respect. Even if each product is compliant with current security standards in force on the date of their design, security mechanisms' resistance necessarily evolves according to the state of the art in security and notably under the emergence of new attacks. Under no circumstances, shall Gemalto be held liable for any third party actions and in particular in case of any successful attack against systems or equipment incorporating Gemalto products. Gemalto disclaims any liability with respect to security for direct, indirect, incidental or consequential damages that result from any use of its products. It is further stressed that independent testing and verification by the person using the product is particularly encouraged, especially in any application in which defective, incorrect or insecure functioning could result in damage to persons or property, denial of service or loss of privacy.

GEMALTO, B.P. 100, 13881 GEMENOS CEDEX, FRANCE.

Tel: +33 (0)4.42.36.50.00 Fax: +33 (0)4.42.36.50.90

Printed in France.

REVISION HISTORY

Date	Release	Comments
November 2010	V0	Creation - Draft
February 2011	A	First release

TABLE OF CONTENTS

TABLE LIST

FIGURE LIST

Introduction

This guide provides information on the use of the Prox–DU and the Prox–SU dual interface (contactless and contact) USB smart card reader/writer in the PC/SC environment.

This document is applicable to following reference, revision C and later:

Model	Reference	Comments
Prox–DU	HWP118184	Dual interface USB smart card reader Contact & contactless
Prox–SU	HWP118185	Contactless interface USB smart card reader With optional SIM/SAM slot
Prox–DU with stand	HWP118830	Prox–DU with a stand for vertical use
Prox–SU with stand	HWP118831	Prox–SU with a stand for vertical use

Table 1 – Dual interface USB smart card reader/writer models

For information on installation, please refer to the "*Installation Guide"* document.

For information on detailed operation about the smart card reader/writer, please refer to the *"Reference Manual"* document.

For information on how to install the smart card reader/writer in a computer, please refer to the *"Computer Installation Guide"* document.

Who Should Read This Book

This reference manual is designed for developers of PC/SC smart card application.

Conventions

Bit Numbering

A byte consists of 8 bits, b7 to b0, where b7 is the most significant bit and b0 is the least significant bit.

One byte	b7	b6	b5	b4	b3	b2	b1	b0

Byte Numbering

A string of n bytes consists of n number of concatenated bytes: Bn…B3…B0.

Bn is the most significant byte and B0 is the least significant byte:

A string of n bytes	Bn	Bn-1	-	-	-	B2	B1	B0

Contact Our Hotline

If you do not find the information you need in this document, or if you find errors, contact the Gemalto hotline at http://support.gemalto.com/.

Please note the document reference number, your job function, and the name of your company. (You will find the document reference number at the bottom of the document.)

Overview

PC/SC (short for “Personal Computer/Smart Card”) is a specification for smart-card integration into computing environments.

For detailled information about PC/SC specification, please refer to the PC/SC workgroup website: http://www.pcscworkgroup.com

Microsoft has implemented PC/SC in Microsoft Windows operating systems:

- The Winscard Smart Card API functions available in Microsoft Windows operating systems are defined in the following website: http://msdn.microsoft.com

A free implementation of PC/SC, PC/SC Lite, is available for Linux and other unixes.

A forked version comes bundled with Mac OS X.

- The Winscard Smart Card API functions available in Linux or MAC OS X operating systems are defined in the following website: http://pcsclite.alioth.debian.org

This document will provide detailed information about the PC/SC Winscard API functions and will give some implementation examples.

The Prox–DU and the Prox–SU devices were designed to be fully compliant with the latest PC/SC V2.0 specification.

Using PC/SC application

PC/SC Overview

The PC/SC specification describes the minimum functionality required of smart cards, smart card readers, and PCs to allow interoperability among compliant elements as provided by a variety of vendors.

The specification as a whole seeks to achieve the following objectives:

- Maintain consistency with existing smart card-related and PC-related standards while expanding upon them where necessary and practical.
- Enable interoperability among components running on various platforms (platform neutral).
- Enable applications to take advantage of products and components from multiple manufacturers (vendor neutral).
- Enable the use of advances in technology without rewriting application-level software (application neutral).
- Facilitate the development of standards for application-level interfaces to smart card services in order to enhance the fielding of a broad range of smart card-based applications in the PC environment.
- Support an environment that encourages the widest possible use of smart cards as an adjunct to the PC environment.

The interoperability specification for smart cards and personal computer systems is composed of nine parts. These are intended to apply only to devices and software intended to operate as a part of an overall system that includes a personal computer.

These documents include:

- Part 1: Introduction and architecture overview
- Part 2: Interface requirements for compatible smart cards and interface devices
- Part 3: Requirements for PC-connected interface devices
- Part 4: Interface devices design considerations and reference design information. This part is here as an example reference only.
- Part 5: Smart card resource manager definition
- Part 6: Smart card service provider interface definition
- Part 7: Application domain/developer design considerations
- Part 8: Recommendation for implementation of security and privacy smart card devices
- Part 9: Interface devices with extended capabilities

The PC/SC Winscard API functions detailed hereafter are related to the Part 5 : Smart card resource manager definition.

The next figure shows the PC/SC architecture:

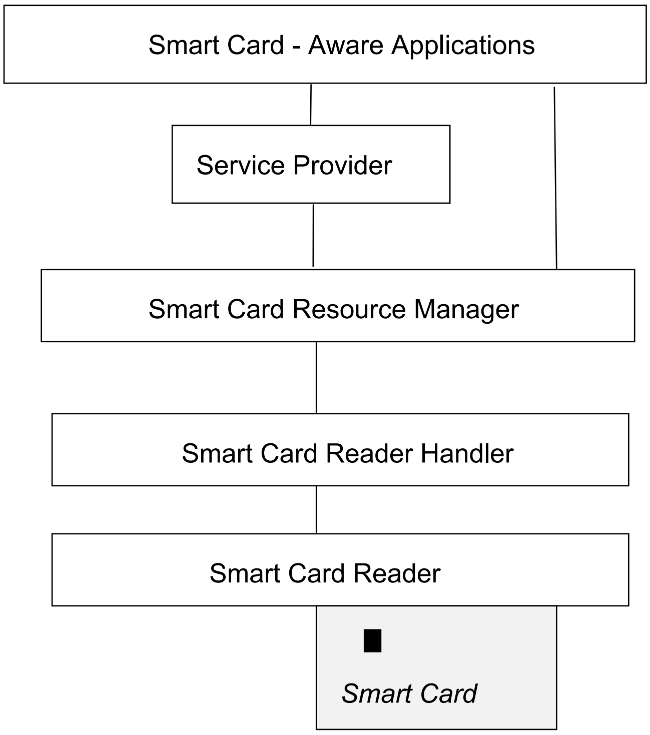

Smart Card - Aware Applications
Service Provider
Smart Card Resource Manager
Smart Card Reader Handler
Smart Card Reader
Smart Card

Figure 1 – PC/SC Architecture

The software design considerations presented in the PC/SC specification address the development of applications built on the architecture presented in the figure below.

This paragraph describes the way smart card-aware applications can use the functionality provided by the smart card. By using the smart card resource manager and the service provider layers, an application can use smart card functionality with some level of independence from a specific reader, or to some extent, from a specific smart card.

Resource Manager

The resource manager is a key component of the PC/SC architecture. It is responsible for managing the other smart card-relevant resources within the system and for supporting controlled access to smart card readers and, through them, individual smart cards. The resource manager is assumed to be a system-level component of the architecture. It must be present and will most likely be provided by the operating system vendor. There should be only a single resource manager within a given system.

The resource manager solves three basic problems in managing access to multiple readers and smart cards.

First, it is responsible for identification and tracking of resources. This includes:

- Tracking installed readers and making this information accessible to other applications.
- Tracking known smart card types, along with their associated service providers and supported Interfaces, and making this information accessible to other applications.
- Tracking smart card insertion and removal events to maintain accurate information on available smart cards within the readers.

Second, it is responsible for controlling the allocation of reader resources (and hence access to smart cards) across multiple applications. It does this by providing mechanisms for attaching to specific readers in shared or exclusive modes of operations.

Finally, it supports transaction primitives on access to services available within a given smart card. This is extremely important because current smart cards are single-threaded devices that often require execution of multiple commands to complete a single function. Transactions allow multiple commands to be executed without interruption, ensuring that intermediate state information is not corrupted.

Service Provider

The service provider is optional and is not described in this document because it is smart card dependant. However some information is given in the next paragraph.

The service provider is the mechanism through which a smart card-specific set of functionalities (in the form of an API) is made accessible to smart card-aware application software. For every smart card, there will be at least one service provider; it is through this service provider that an application can access data or services on that specific smart card.

The following three classes of services are widely implemented within existing smart cards:

- File services
- Authentication Services
- Cryptographic Services

These services, when present, have a high degree of functionality in common across smart cards. Consequently, it is beneficial to standardize interfaces to these services so that application development and maintenance are simpler. This specification defines such interfaces as well as a standard interface for controlling basic access to a smart card.

Additional smart card services tend to reflect the needs of specific application domains (EMV, GSM, and so on). It is believed most appropriate for groups within specific industries to standardize interfaces in such areas. This architecture fully supports the addition of such interfaces to the core set identified above.

The definition of the API's exposed by a specific service provider generally comes from a third-party workgroup (EMV, GSM, PC/SC, etc.). The service provider, by definition, has an intimate knowledge of the smart card to which it provides access. The implementation of these API's might be expected to come from a variety of sources, including (but not limited to) the following:

- The smart card supplier who wants to enable smart card use within the PC environment. Providing a service provider makes accessing the smart card an application software development effort that can be pursued by application developers with no specific expertise in smart card technology (either smart cards or readers).
- The smart card issuer, who might layer a “personalized” service provider on top of the service provider provided by its smart card supplier.
- A smart card-Aware Application supplier who wishes to define the level of functionality required of a smart card to adequately support the application. In defining the API, the application supplier enables one or more smart card suppliers to provide a smart card and the service provider that implements the API defined by the application supplier.
- One or more parties interested in a specific domain, who wish to enable the development of both applications and smart cards to support those applications within a domain of interest.

The smart card Service Provider

The smart card service provider is one of two possible sub-components of the service provider. It is responsible for exposing high-level interfaces to non-cryptographic services. This exposure is expected to include common interfaces, defined in this specification, for managing connections to a specific smart card, as well as access to file and authentication services. In addition, the smart card service provider may implement interfaces that the vendor defines for features specific to the application domain.

All smart card service providers shall implement the interface for managing connections to a smart card as defined herein (see Section 3). This interface provides mechanisms for connecting and disconnecting to a smart card.

In addition, to be compliant with this specification, smart card service providers that expose file access and authentication services shall do so using the interfaces defined herein (see Section 3.4). These interfaces encapsulate functionality defined by ISO 7816-4, along with natural extensions for functionality such as file creation and deletion.

The file access interface defines mechanisms for the following tasks:

- Locating files by name
- Creating or opening files
- Reading and writing file contents
- Closing a file
- Deleting files
- Managing file attributes

The authentication interface defines mechanisms for the following tasks:

- Cardholder verification
- smart card authentication
- Application authentication to the smart card

The Cryptographic Service Provider

The cryptographic service provider is a sub-component of the service provider. In contrast to the smart card service provider, the cryptographic service provider isolates cryptographic services because existing regulations imposed by various governments affect import and export. The cryptographic service provider allows applications to make use of cryptographic services in a manner that compartmentalizes the sensitive elements of cryptographic support into a well-defined and independently installable software package.

The cryptographic service provider encapsulates access to cryptographic functionality provided by a specific smart card through high level programming interfaces. Its purpose is to expose available cryptographic functions to applications running on a PC. All other functionality should be implemented in the smart card service provider.

Interfaces are defined in this specification for the following general-purpose cryptographic services:

- Key generation
- Key management
- Digital signatures
- Hashing (or message digests)
- Bulk encryption services
- Key import and export

The PC/SC Winscard API functions are listed the following tables:

Smart Card Database Query Functions: Query the smart card database	
SCardGetProviderId	Retrieve the identifier (GUID) of the primary service provider for the given card
SCardListCards	Retrieve a list of cards previously introduced to the system by a specific user
SCardListInterfaces	Retrieve the identifiers (GUIDs) of the interfaces supplied by a given card
SCardListReaderGroups	Retrieve a list of reader groups that have previously been introduced to the system
SCardListReaders	Retrieve the list of readers within a set of named reader groups

Table 2 – Smart Card Database Query Functions

Smart Card Database Management Functions: Manage the smart card database	
SCardAddReaderToGroup	Add a reader to a reader group
SCardForgetCardType	Remove a smart card from the system
SCardForgetReader	Remove a reader from the system
SCardForgetReaderGroup	Remove a reader group from the system
SCardIntroduceCardType	Introduce a new card to the system
SCardIntroduceReader	Introduce a new reader to the system
SCardIntroduceReaderGroup	Introduce a new reader group to the system
SCardRemoveReaderFromGroup	Remove a reader from a reader group

Table 3 – Smart Card Database Management Functions

Resource Manager Context Functions: Manage the context of the resource manager's database operations	
SCardEstablishContext	Establishes a context for accessing the smart card database
SCardReleaseContext	Closes an established context

Table 4 – Resource Manager Context Functions

Resource Manager Support Function: Release allocated memory	
SCardFreeMemory	Release memory returned through the use of SCARD_AUTOALLOCATE

Table 5 – Resource Manager Support Function

Smart Card Tracking Functions: Track smart cards within readers	
SCardLocateCards	Search for a card whose ATR string matches a supplied card name

SCardGetStatusChange	Block execution until the current availability of cards changes
SCardCancel	Terminate outstanding actions

Table 6 – Smart Card Tracking Functions

Smart Card and Reader Access Functions: Connect to and communicate with a smart card, including transferring data using *T=0*, *T=1*, and raw protocols	
SCardConnect	Connect to a card
SCardReconnect	Reestablish a connection
SCardDisconnect	Terminate a connection
SCardBeginTransaction	Start a transaction, blocking other applications from accessing a card
SCardEndTransaction	End a transaction, allowing other applications to access a card
SCardStatus	Provide the current status of the reader
SCardTransmit	Requests service and receives data back from a card using T=0, T=1, and raw protocols

Table 7 – Smart Card and Reader Access Functions

Direct Card Access Functions: Communicate with cards that may not conform to the ISO 7816 specifications	
SCardControl	Provide direct control of the reader
SCardGetAttrib	Get reader attributes
SCardSetAttrib	Set reader attribute

Table 8 – Direct Card Access Functions

The next paragraphs give more information about these commands (extracts from the MSDN website: http://msdn.microsoft.com)

Most of the functions are defined in the “Part 5: Smart card resource manager definition” of the PC/SC specification.

WinScard API

The next paragraph contains information retrieved from the http://msdn.microsoft.com web site.

SCardGetProviderId Function

The SCardGetProviderId function returns the identifier (GUID) of the primary service provider for a given card.

The caller supplies the name of a smart card (previously introduced to the system) and receives the registered identifier of the primary service provider GUID, if one exists.

Syntax:

```
LONG WINAPI SCardGetProviderId(
  __in   SCARDCONTEXT hContext,
  __in   LPCTSTR szCard,
  __out  LPGUID pguidProviderId
);
```

Parameters:

hContext [in]

> Handle that identifies the resource manager context for the query. The resource manager context can be set by a previous call to SCardEstablishContext. This parameter cannot be NULL.

szCard [in]

> Name of the card defined to the system.

pguidProviderId [out]

> Identifier (GUID) of the primary service provider. This provider may be activated using COM, and will supply access to other services in the card.

Return Value:

This function returns different values depending on whether it succeeds or fails:

> Success: SCARD_S_SUCCESS.
>
> Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

The SCardGetProviderId function is a database query function. For more information on other database query functions, see Smart Card Database Query Functions.

Examples:

The following example shows how to get the provider ID for the specified card. The example assumes that hContext is a valid handle obtained from a previous call to the SCardEstablishContext function and that "MyCardName" was introduced by a previous call to the SCardIntroduceCardType function.

```
GUID    guidProv;
LONG    lReturn;

lReturn = SCardGetProviderId(hContext,
                              L"MyCardName",
                              &guidProv);
if ( SCARD_S_SUCCESS != lReturn )
    printf("Failed SCardGetProviderId - %x\n", lReturn);
else
{
    // Use the provider GUID as needed.
    // ...
}
```

SCardListCards Function

The SCardListCards function searches the smart card database and provides a list of named cards previously introduced to the system by the user.

The caller specifies an ATR string, a set of interface identifiers (GUIDs), or both. If both an ATR string and an identifier array are supplied, the cards returned will match the ATR string supplied and support the interfaces specified.

Syntax:

```
LONG WINAPI SCardListCards(
  __in      SCARDCONTEXT hContext,
  __in_opt  LPCBYTE pbAtr,
  __in      LPCGUID rgguidInterfaces,
  __in      DWORD cguidInterfaceCount,
  __out     LPTSTR mszCards,
  __inout   LPDWORD pcchCards
);
```

Parameters:

hContext [in]

Handle that identifies the resource manager context for the query. The resource manager context can be set by a previous call to SCardEstablishContext.

pbAtr [in, optional]

Address of an ATR string to compare to known cards, or NULL if no ATR matching is to be performed.

rgguidInterfaces [in]

Array of identifiers (GUIDs), or NULL if no interface matching is to be performed. When an array is supplied, a card name will be returned only if all the specified identifiers are supported by the card.

cguidInterfaceCount [in]

Number of entries in the rgguidInterfaces array. If rgguidInterfaces is NULL, then this value is ignored.

mszCards [out]

Multi-string that lists the smart cards found. If this value is NULL, SCardListCards ignores the buffer length supplied in pcchCards, returning the length of the buffer that would have been returned if this parameter had not been NULL to pcchCards and a success code.

pcchCards [in, out]

Length of the mszCards buffer in characters. Receives the actual length of the multi-string structure, including all trailing null characters. If the buffer length is specified as SCARD_AUTOALLOCATE, then mszCards is converted to a pointer to a byte pointer, and receives the address of a block of memory containing the multi-string structure. This block of memory must be deallocated with SCardFreeMemory.

Return Value:

This function returns different values depending on whether it succeeds or fails:

Success: SCARD_S_SUCCESS.

Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

To return all smart cards introduced to the subsystem, set pbAtr and rgguidInterfaces to NULL.

The SCardListCards function is a database query function. For more information on other database query functions, see Smart Card Database Query Functions.

Examples:

The following example shows listing of the smart cards.

```
LPTSTR pmszCards = NULL;
LPTSTR pCard;
LONG lReturn;
DWORD cch = SCARD_AUTOALLOCATE;
// Retrieve the list of cards.
lReturn = SCardListCards(NULL,
                         NULL,
                         NULL,
                         NULL,
                         (LPTSTR)&pmszCards,
                         &cch );
if ( SCARD_S_SUCCESS != lReturn )
{
   printf("Failed SCardListCards\n");
   exit(1); // Or other appropriate error action
}
// Do something with the multi string of cards.
// Output the values.
// A double-null terminates the list of values.
pCard = pmszCards;
while ( '\0' != *pCard )
{
   // Display the value.
   printf("%S\n", pCard );
   // Advance to the next value.
   pCard = pCard + wcslen(pCard) + 1;
}
// Remember to free pmszCards (by calling SCardFreeMemory).
// ...
```

SCardListInterfaces Function

The SCardListInterfaces function provides a list of interfaces supplied by a given card.

The caller supplies the name of a smart card previously introduced to the subsystem, and receives the list of interfaces supported by the card.

Syntax:

```
LONG WINAPI SCardListInterfaces(
  __in     SCARDCONTEXT hContext,
  __in     LPCTSTR szCard,
  __out    LPGUID pguidInterfaces,
  __inout  LPDWORD pcguidInterfaces
);
```

Parameters:

hContext [in]

> Handle that identifies the resource manager context for the query. The resource manager context can be set by a previous call to SCardEstablishContext. This parameter cannot be NULL.

szCard [in]

> Name of the smart card already introduced to the smart card subsystem.

pguidInterfaces [out]

> Array of interface identifiers (GUIDs) that indicate the interfaces supported by the smart card. If this value is NULL, SCardListInterfaces ignores the array length supplied in pcguidInterfaces, returning the size of the array that would have been returned if this parameter had not been NULL to pcguidInterfaces and a success code.

pcguidInterfaces [in, out]

> Size of the pcguidInterfaces array, and receives the actual size of the returned array. If the array size is specified as SCARD_AUTOALLOCATE, then pcguidInterfaces is converted to a pointer to a GUID pointer, and receives the address of a block of memory containing the array. This block of memory must be deallocated with SCardFreeMemory.

Return Value:

This function returns different values depending on whether it succeeds or fails:

> Success: SCARD_S_SUCCESS.
>
> Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

The SCardListInterfaces function is a database query function. For more information on other database query functions, see Smart Card Database Query Functions.

Examples:

The following example shows listing the interfaces for a smart card.

```
LPGUID       pGuids = NULL;
LONG         lReturn;
```

```
DWORD          cGuid = SCARD_AUTOALLOCATE;

// Retrieve the list of interfaces.
lReturn = SCardListInterfaces(NULL,
                  (LPCSTR) "MyCard",
                  (LPGUID)&pGuids,
                  &cGuid );
if ( SCARD_S_SUCCESS != lReturn )
{
  printf("Failed SCardListInterfaces\n");
  exit(1);  // Or other appropriate action
}

if ( 0 != cGuid )
{
  // Do something with the array of Guids.
  // Remember to free pGuids when done (by SCardFreeMemory).
  // ...
}
```

SCardListReaderGroups Function

The SCardListReaderGroups function provides the list of reader groups that have previously been introduced to the system.

Syntax:

```
LONG WINAPI SCardListReaderGroups(
  __in     SCARDCONTEXT hContext,
  __out    LPTSTR mszGroups,
  __inout  LPDWORD pcchGroups
);
```

Parameters:

hContext [in]

> Handle that identifies the resource manager context for the query. The resource manager context can be set by a previous call to SCardEstablishContext. This parameter cannot be NULL.

mszGroups [out]

> Multi-string that lists the reader groups defined to the system and available to the current user on the current terminal. If this value is NULL, SCardListReaderGroups ignores the buffer length supplied in pcchGroups, writes the length of the buffer that would have been returned if this parameter had not been NULL to pcchGroups, and returns a success code.

pcchGroups [in, out]

> Length of the mszGroups buffer in characters, and receives the actual length of the multi-string structure, including all trailing null characters. If the buffer length is specified as SCARD_AUTOALLOCATE, then mszGroups is converted to a pointer to a byte pointer, and receives the address of a block of memory containing the multi-string structure. This block of memory must be deallocated with SCardFreeMemory.

Return Value:

This function returns different values depending on whether it succeeds or fails:

> Success: SCARD_S_SUCCESS.
>
> Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

A group is returned only if it contains at least one reader. This includes the group SCard$DefaultReaders. The group SCard$AllReaders cannot be returned, since it only exists implicitly.

The SCardListReaderGroups function is a database query function. For more information on other database query functions, see Smart Card Database Query Functions.

Examples:

The following example shows listing the reader groups.

```
LPTSTR        pmszReaderGroups = NULL;
LPTSTR        pReaderGroup;
LONG          lReturn;
```

```
DWORD          cch = SCARD_AUTOALLOCATE;
// Retrieve the list the reader groups.
// hSC was set by a previous call to SCardEstablishContext.
lReturn = SCardListReaderGroups(hSC,
                                (LPTSTR)&pmszReaderGroups,
                                &cch );
if ( SCARD_S_SUCCESS != lReturn )
    printf("Failed SCardListReaderGroups\n");
else
{
    // Do something with the multi string of reader groups.
    // Output the values.
    // A double-null terminates the list of values.
    pReaderGroup = pmszReaderGroups;
    while ( '\0' != *pReaderGroup )
    {
        // Display the value.
        printf("%S\n", pReaderGroup );
        // Advance to the next value.
        pReaderGroup = pReaderGroup + wcslen((wchar_t *) pReaderGroup) + 1;
    }
    // Remember to free pmszReaderGroups by a call to SCardFreeMemory.
    // ...
}
```

SCardListReaders Function

The SCardListReaders function provides the list of readers within a set of named reader groups, eliminating duplicates.

The caller supplies a list of reader groups, and receives the list of readers within the named groups. Unrecognized group names are ignored.

Syntax:

```
LONG WINAPI SCardListReaders(
  __in      SCARDCONTEXT hContext,
  __in_opt  LPCTSTR mszGroups,
  __out     LPTSTR mszReaders,
  __inout   LPDWORD pcchReaders
);
```

Parameters:

hContext [in]

> Handle that identifies the resource manager context for the query. The resource manager context can be set by a previous call to SCardEstablishContext. This parameter cannot be NULL.

mszGroups [in, optional]

> Names of the reader groups defined to the system, as a multi-string. Use a NULL value to list all readers in the system (that is, the SCard$AllReaders group).

mszReaders [out]

> Multi-string that lists the card readers within the supplied reader groups. If this value is NULL, SCardListReaders ignores the buffer length supplied in pcchReaders, writes the length of the buffer that would have been returned if this parameter had not been NULL to pcchReaders, and returns a success code.

pcchReaders [in, out]

> Length of the mszReaders buffer in characters. This parameter receives the actual length of the multi-string structure, including all trailing null characters. If the buffer length is specified as SCARD_AUTOALLOCATE, then mszReaders is converted to a pointer to a byte pointer, and receives the address of a block of memory containing the multi-string structure. This block of memory must be deallocated with SCardFreeMemory.

Return Value:

This function returns different values depending on whether it succeeds or fails:

> Success: SCARD_S_SUCCESS.
>
> Group contains no readers: SCARD_E_NO_READERS_AVAILABLE
>
> Other: An error code. For more information, see Smart Card Return Values.

Remarks:

The SCardListReaders function is a database query function. For more information on other database query functions, see Smart Card Database Query Functions.

Examples:

The following example shows listing the readers.

```
LPTSTR        pmszReaders = NULL;
LPTSTR        pReader;
LONG         lReturn, lReturn2;
DWORD         cch = SCARD_AUTOALLOCATE;
// Retrieve the list the readers.
// hSC was set by a previous call to SCardEstablishContext.
lReturn = SCardListReaders(hSC,
                           NULL,
                           (LPTSTR)&pmszReaders,
                           &cch );
switch( lReturn )
{
    case SCARD_E_NO_READERS_AVAILABLE:
        printf("Reader is not in groups.\n");
        // Take appropriate action.
        // ...
        break;
    case SCARD_S_SUCCESS:
        // Do something with the multi string of readers.
        // Output the values.
        // A double-null terminates the list of values.
        pReader = pmszReaders;
        while ( '\0' != *pReader )
        {
            // Display the value.
            printf("Reader: %S\n", pReader );
            // Advance to the next value.
            pReader = pReader + wcslen((wchar_t *)pReader) + 1;
        }
        // Free the memory.
        lReturn2 = SCardFreeMemory( hSC,
                                   pmszReaders );
        if ( SCARD_S_SUCCESS != lReturn2 )
            printf("Failed SCardFreeMemory\n");
        break;
default:
        printf("Failed SCardListReaders\n");
        // Take appropriate action.
        // ...
        break;
}
```

SCardAddReaderToGroup Function

The SCardAddReaderToGroup function adds a reader to a reader group.

Syntax:

```
LONG WINAPI SCardAddReaderToGroup(
  __in  SCARDCONTEXT hContext,
  __in  LPCTSTR szReaderName,
  __in  LPCTSTR szGroupName
);
```

Parameters:

hContext [in]

> Handle that identifies the resource manager context. The resource manager context is set by a previous call to SCardEstablishContext. This parameter cannot be NULL.

szReaderName [in]

> Display name of the reader that you are adding.

szGroupName [in]

> Display name of the group to which you are adding the reader.

Return Value:

This function returns different values depending on whether it succeeds or fails:

> Success: SCARD_S_SUCCESS.
>
> Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

SCardAddReaderToGroup automatically creates the reader group specified if it does not already exist.

The SCardAddReaderToGroup function is a database management function. For more information on other database management functions, see Smart Card Database Management Functions.

Examples:

The following example demonstrates how to add a smart card reader to a group. The example assumes that lReturn is an existing variable of type LONG, that hContext is a valid handle received from a previous call to the SCardEstablishContext function, and that "MyReader" and "MyReaderGroup" are known by the system through previous calls to the SCardIntroduceReader and SCardIntroduceReaderGroup functions, respectively.

```
lReturn = SCardAddReaderToGroup( hContext,
                                 L"MyReader",
                                 L"MyReaderGroup");
if ( SCARD_S_SUCCESS != lReturn )
   printf("Failed SCardAddReaderToGroup\n");
```

SCardForgetCardType Function

The SCardForgetCardType function removes an introduced smart card from the smart card subsystem.

Syntax:

```
LONG WINAPI SCardForgetCardType(
  __in  SCARDCONTEXT hContext,
  __in  LPCTSTR szCardName
);
```

Parameters:

hContext [in]

Handle that identifies the resource manager context. The resource manager context is set by a previous call to SCardEstablishContext. This parameter cannot be NULL.

szCardName [in]

Display name of the card to be removed from the smart card database.

Return Value:

This function returns different values depending on whether it succeeds or fails:

Success: SCARD_S_SUCCESS.

Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

The SCardForgetCardType function is a database management function. For more information on other database management functions, see Smart Card Database Management Functions.

Examples:

The following example removes the specified card type from the system. The example assumes that lReturn is a valid variable of type LONG, that hContext is a valid handle received from a previous call to the SCardEstablishContext function, and that "MyCardName" was previously introduced by a call to the SCardIntroduceCardType function.

```
lReturn = SCardForgetCardType(hContext,
                              L"MyCardName");
if ( SCARD_S_SUCCESS != lReturn )
    printf("Failed SCardForgetCardType\n");
```

SCardForgetReader Function

The SCardForgetReader function removes a previously introduced reader from control by the smart card subsystem. It is removed from the smart card database, including from any reader group that it may have been added to.

Syntax:

```
LONG WINAPI SCardForgetReader(
  __in  SCARDCONTEXT hContext,
  __in  LPCTSTR szReaderName
);
```

Parameters:

hContext [in]

Handle that identifies the resource manager context. The resource manager context is set by a previous call to SCardEstablishContext. This parameter cannot be NULL.

szReaderName [in]

Display name of the reader to be removed from the smart card database.

Return Value:

This function returns different values depending on whether it succeeds or fails:

Success: SCARD_S_SUCCESS.

Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

If the specified reader is the last member of a reader group, the reader group is automatically removed as well.

The SCardForgetReader function is a database management function. For more information on other database management functions, see Smart Card Database Management Functions.

Examples:

The following example removes the display name of the specified card reader from the system. The example assumes that lReturn is a valid variable of type LONG and that hContext is a valid handle received from a previous call to the SCardEstablishContext function.

```
lReturn = SCardForgetReader(hContext,
                            TEXT("MyReader"));
if ( SCARD_S_SUCCESS != lReturn )
    printf("Failed SCardForgetReader\n");
```

SCardForgetReaderGroup Function

The SCardForgetReaderGroup function removes a previously introduced smart card reader group from the smart card subsystem. Although this function automatically clears all readers from the group, it does not affect the existence of the individual readers in the database.

Syntax:

```
LONG WINAPI SCardForgetReaderGroup(
  __in  SCARDCONTEXT hContext,
  __in  LPCTSTR szGroupName
);
```

Parameters:

hContext [in]

Handle that identifies the resource manager context. The resource manager context is set by a previous call to SCardEstablishContext. This parameter cannot be NULL.

szGroupName [in]

Display name of the reader group to be removed. System-defined reader groups cannot be removed from the database.

Return Value:

This function returns different values depending on whether it succeeds or fails:

Success: SCARD_S_SUCCESS.

Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

The SCardForgetReaderGroup function is a database management function. For more information on other database management functions, see Smart Card Database Management Functions.

Examples:

The following example shows how to remove a reader group from the system. The example assumes that lReturn is an existing variable of type LONG, and that hContext is a valid handle to a resource manager context previously obtained from a call to the SCardEstablishContext function.

```
lReturn = SCardForgetReaderGroup(hContext,
                                 L"MyReaderGroup");
if ( SCARD_S_SUCCESS != lReturn )
    printf("Failed SCardForgetReaderGroup\n");
```

SCardIntroduceCardType Function

The SCardIntroduceCardType function introduces a smart card to the smart card subsystem (for the active user) by adding it to the smart card database.

Syntax:

```
LONG WINAPI SCardIntroduceCardType(
  __in      SCARDCONTEXT hContext,
  __in      LPCTSTR szCardName,
  __in_opt  LPCGUID pguidPrimaryProvider,
  __in_opt  LPCGUID rgguidInterfaces,
  __in      DWORD dwInterfaceCount,
  __in      LPCBYTE pbAtr,
  __in      LPCBYTE pbAtrMask,
  __in      DWORD cbAtrLen
);
```

Parameters:

hContext [in]

Handle that identifies the resource manager context. The resource manager context is set by a previous call to SCardEstablishContext. This parameter cannot be NULL.

szCardName [in]

Name by which the user can recognize the card.

pguidPrimaryProvider [in, optional]

Pointer to the identifier (GUID) for the smart card's primary service provider.

rgguidInterfaces [in, optional]

Array of identifiers (GUIDs) that identify the interfaces supported by the smart card.

dwInterfaceCount [in]

Number of identifiers in the rgguidInterfaces array.

pbAtr [in]

ATR string that can be used for matching purposes when querying the smart card database (for more information, see SCardListCards). The length of this string is determined by normal ATR parsing.

pbAtrMask [in]

Optional bitmask to use when comparing the ATRs of smart cards to the ATR supplied in pbAtr. If this value is non-NULL, it must point to a string of bytes the same length as the ATR string supplied in pbAtr. When a given ATR string A is compared to the ATR supplied in pbAtr, it matches if and only if A & M = pbAtr, where M is the supplied mask, and & represents bitwise AND.

cbAtrLen [in]

Length of the ATR and optional ATR mask. If this value is zero, then the length of the ATR is determined by normal ATR parsing. This value cannot be zero if a pbAtr value is supplied.

Return Value:

This function returns different values depending on whether it succeeds or fails:

Success: SCARD_S_SUCCESS.

Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

The SCardIntroduceCardType function is a database management function. For more information on other database management functions, see Smart Card Database Management Functions.

To remove a smart card, use SCardForgetCardType.

Examples:

The following example shows how to introduce a card type. The example assumes that hContext is a valid handle obtained from a previous call to the SCardEstablishContext function.

```
GUID  MyGuid = { 0xABCDEF00,
            0xABCD,
            0xABCD,
            0xAA, 0xBB, 0xCC, 0xDD,
            0xAA, 0xBB, 0xCC, 0xDD };
static const BYTE MyATR[] =     { 0xaa, 0xbb, 0xcc, 0x00, 0xdd };
static const BYTE MyATRMask[] = { 0xff, 0xff, 0xff, 0x00, 0xff};
LONG         lReturn;
lReturn = SCardIntroduceCardType(hContext,
                        L"MyCardName",
                        &MyGuid,
                        NULL,    // No interface array
                        0,       // Interface count = 0
                        MyATR,
                        MyATRMask,
                        sizeof(MyATR));
if ( SCARD_S_SUCCESS != lReturn )
   printf("Failed SCardIntroduceCardType\n");
```

SCardIntroduceReader Function

The SCardIntroduceReader function introduces a new name for an existing smart card reader.

Note Smart card readers are automatically introduced to the system; a smart card reader vendor's setup program can also introduce a smart card reader to the system.

Syntax:

```
LONG WINAPI SCardIntroduceReader(
  __in  SCARDCONTEXT hContext,
  __in  LPCTSTR szReaderName,
  __in  LPCTSTR szDeviceName
);
```

Parameters:

hContext [in]

Handle that identifies the resource manager context. The resource manager context is set by a previous call to SCardEstablishContext. This parameter cannot be NULL.

szReaderName [in]

Display name to be assigned to the reader.

szDeviceName [in]

System name of the smart card reader, for example, "MyReader 01".

Return Value:

This function returns different values depending on whether it succeeds or fails:

Success: SCARD_S_SUCCESS.

Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

All readers installed on the system are automatically introduced by their system name. Typically, SCardIntroduceReader is called only to change the name of an existing reader.

The SCardIntroduceReader function is a database management function. For more information on other database management functions, see Smart Card Database Management Functions.

To remove a reader, use SCardForgetReader.

Examples:

The following example shows introducing a smart card reader.

```
// This example renames the reader name.
// This is a two-step process (first add the new
// name, then forget the old name).
LPBYTE    pbAttr = NULL;
DWORD     cByte = SCARD_AUTOALLOCATE;
LONG      lReturn;
// Step 1: Add the new reader name.
// The device name attribute is a necessary value.
// hCardHandle was set by a previous call to SCardConnect.
lReturn = SCardGetAttrib(hCardHandle,
                         SCARD_ATTR_DEVICE_SYSTEM_NAME,
                         (LPBYTE)&pbAttr,
                         &cByte);
if ( SCARD_S_SUCCESS != lReturn )
{
    printf("Failed SCardGetAttrib\n");
    exit(1);  // Or other error action
}
// Add the reader name.
// hContext was set earlier by SCardEstablishContext.
lReturn = SCardIntroduceReader(hContext,
                               TEXT("My New Reader Name"),
                               (LPCTSTR)pbAttr );
if ( SCARD_S_SUCCESS != lReturn )
{
    printf("Failed SCardIntroduceReader\n");
    exit(1);  // Or other error action
}
// Step 2: Forget the old reader name.
lReturn = SCardForgetReader(hContext,
                            (LPCTSTR)pbAttr );
if ( SCARD_S_SUCCESS != lReturn )
{
    printf("Failed SCardForgetReader\n");
    exit(1);  // Or other error action
}
// Free the memory when done.
lReturn = SCardFreeMemory( hContext, pbAttr );
```

SCardIntroduceReaderGroup Function

The SCardIntroduceReaderGroup function introduces a reader group to the smart card subsystem. However, the reader group is not created until the group is specified when adding a reader to the smart card database.

Syntax:

```
LONG WINAPI SCardIntroduceReaderGroup(
  __in  SCARDCONTEXT hContext,
  __in  LPCTSTR szGroupName
);
```

Parameters:

hContext [in]

Supplies the handle that identifies the resource manager context. The resource manager context is set by a previous call to the SCardEstablishContext function. If this parameter is NULL, the scope of the resource manager is SCARD_SCOPE_SYSTEM.

szGroupName [in]

Supplies the display name to be assigned to the new reader group.

Return Value:

This function returns different values depending on whether it succeeds or fails:

Success: SCARD_S_SUCCESS.

Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

The SCardIntroduceReaderGroup function is provided for PC/SC specification compatibility. Reader groups are not stored until a reader is added to the group.

The SCardIntroduceReaderGroup function is a database management function. For a description of other database management functions, see Smart Card Database Management Functions.

To remove a reader group, use SCardForgetReaderGroup.

Examples:

The following example shows introducing a smart card reader group.

```
// Introduce the reader group.
// lReturn is of type LONG.
// hContext was set by a previous call to SCardEstablishContext.
lReturn = SCardIntroduceReaderGroup(hContext,
                                    L"MyReaderGroup");
if ( SCARD_S_SUCCESS != lReturn )
    printf("Failed SCardIntroduceReaderGroup\n");
```

SCardRemoveReaderFromGroup Function

The SCardRemoveReaderFromGroup function removes a reader from an existing reader group. This function has no effect on the reader.

Syntax:

```
LONG WINAPI SCardRemoveReaderFromGroup(
  __in  SCARDCONTEXT hContext,
  __in  LPCTSTR szReaderName,
  __in  LPCTSTR szGroupName
);
```

Parameters:

hContext [in]

Handle that identifies the resource manager context. The resource manager context is set by a previous call to SCardEstablishContext. This parameter cannot be NULL.

szReaderName [in]

Display name of the reader to be removed.

szGroupName [in]

Display name of the group from which the reader should be removed.

Return Value:

This function returns different values depending on whether it succeeds or fails:

Success: SCARD_S_SUCCESS.

Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

When the last reader is removed from a group, the group is automatically forgotten.

The SCardRemoveReaderFromGroup function is a database management function. For information about other database management functions, see Smart Card Database Management Functions.

To add a reader to a reader group, use SCardAddReaderToGroup.

Examples:

The following example shows how to remove a reader from the group.

```
// Remove a reader from the group.
// lReturn is of type LONG.
// hContext was set by a previous call to SCardEstablishContext.
// The group is automatically forgotten if no readers remain in it.
lReturn = SCardRemoveReaderFromGroup(hContext,
                              L"MyReader",
                              L"MyReaderGroup");
if ( SCARD_S_SUCCESS != lReturn )
   printf("Failed SCardRemoveReaderFromGroup\n");
```

SCardEstablishContext Function

The SCardEstablishContext function establishes the resource manager context (the scope) within which database operations are performed.

Syntax:

```
LONG WINAPI SCardEstablishContext(
  __in   DWORD dwScope,
  __in   LPCVOID pvReserved1,
  __in   LPCVOID pvReserved2,
  __out  LPSCARDCONTEXT phContext
);
```

Parameters:

dwScope [in]

Scope of the resource manager context. This parameter can be one of the following values:

SCARD_SCOPE_USER: Database operations are performed within the domain of the user.

SCARD_SCOPE_SYSTEM: Database operations are performed within the domain of the system. The calling application must have appropriate access permissions for any database actions.

pvReserved1 [in]

Reserved for future use and must be NULL. This parameter will allow a suitably privileged management application to act on behalf of another user.

pvReserved2 [in]

Reserved for future use and must be NULL.

phContext [out]

A handle to the established resource manager context. This handle can now be supplied to other functions attempting to do work within this context.

Return Value:

If the function succeeds, the function returns SCARD_S_SUCCESS.

If the function fails, it returns an error code. For more information, see Smart Card Return Values.

Remarks:

The context handle returned by SCardEstablishContext can be used by database query and management functions. For more information, see Smart Card Database Query Functions and Smart Card Database Management Functions.

To release an established resource manager context, use SCardReleaseContext.

If the client attempts a smart card operation in a remote session, such as a client session running on a terminal server, and the operating system in use does not support smart card redirection, this function returns ERROR_BROKEN_PIPE.

Examples:

The following example establishes a resource manager context.

```
SCARDCONTEXT    hSC;
LONG            lReturn;
// Establish the context.
lReturn = SCardEstablishContext(SCARD_SCOPE_USER,
                                NULL,
                                NULL,
                                &hSC);
if ( SCARD_S_SUCCESS != lReturn )
    printf("Failed SCardEstablishContext\n");
else
{
    // Use the context as needed. When done,
    // free the context by calling SCardReleaseContext.
    // ...
}
```

SCardReleaseContext Function

The SCardReleaseContext function closes an established resource manager context, freeing any resources allocated under that context, including SCARDHANDLE objects and memory allocated using the SCARD_AUTOALLOCATE length designator.

Syntax:

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

Parameters:

hContext [in]

Handle that identifies the resource manager context. The resource manager context is set by a previous call to SCardEstablishContext.

Return Value:

This function returns different values depending on whether it succeeds or fails:

Success: SCARD_S_SUCCESS.

Failure: An error code. For more information, see Smart Card Return Values.

Examples:

The following example shows releasing a context.

```
// Free the context.
// lReturn is of type LONG.
// hSC was set by an earlier call to SCardEstablishContext.
lReturn = SCardReleaseContext(hSC);
if ( SCARD_S_SUCCESS != lReturn )
        printf("Failed SCardReleaseContext\n")
```

SCardFreeMemory Function

The SCardFreeMemory function releases memory that has been returned from the resource manager using the SCARD_AUTOALLOCATE length designator.

Syntax:

```
LONG WINAPI SCardFreeMemory(
  __in  SCARDCONTEXT hContext,
  __in  LPCVOID pvMem
);
```

Parameters:

hContext [in]

Handle that identifies the resource manager context returned from SCardEstablishContext, or NULL if the creating function also specified NULL for its hContext parameter. For more information, see Smart Card Database Query Functions.

pvMem [in]

Memory block to be released.

Return Value:

This function returns different values depending on whether it succeeds or fails:

Success: SCARD_S_SUCCESS.

Failure: An error code. For more information, see Smart Card Return Values.

Examples:

The following example shows how to free memory allocated by the resource manager. The example assumes that lReturn is an existing variable of type LONG, that hSC is a valid handle to a resource manager context obtained from a previous call to the SCardEstablishContext function, and that pmszReaders is a string initialized in a previous call to the SCardListReaders function.

```
lReturn = SCardFreeMemory(hSC,
                          pmszReaders );
if ( SCARD_S_SUCCESS != lReturn )
    printf("Failed SCardFreeMemory\n");
```

SCardLocateCards Function

The SCardLocateCards function searches the readers listed in the rgReaderStates parameter for a card with an ATR string that matches one of the card names specified in mszCards, returning immediately with the result.

Syntax:

```
LONG WINAPI SCardLocateCards(
  __in     SCARDCONTEXT hContext,
  __in     LPCTSTR mszCards,
  __inout  LPSCARD_READERSTATE rgReaderStates,
  __in     DWORD cReaders
);
```

Parameters:

hContext [in]

A handle that identifies the resource manager context. The resource manager context is set by a previous call to SCardEstablishContext.

mszCards [in]

A multiple string that contains the names of the cards to search for.

rgReaderStates [in, out]

An array of SCARD_READERSTATE structures that, on input, specify the readers to search and that, on output, receives the result.

cReaders [in]

The number of elements in the rgReaderStates array.

Return Value:

This function returns different values depending on whether it succeeds or fails:

Success: SCARD_S_SUCCESS.

Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

This service is especially useful when used in conjunction with SCardGetStatusChange. If no matching cards are found by means of SCardLocateCards, the calling application may use SCardGetStatusChange to wait for card availability changes.

The SCardLocateCards function is a smart card tracking function. For more information on other tracking functions, see Smart Card Tracking Functions.

Examples:

The following example shows locating smart cards.

```
// Copyright (c) Microsoft Corporation. All rights reserved.
#include <stdio.h>
#include <winscard.h>
#include <tchar.h>
#pragma comment(lib, "winscard.lib")
HRESULT __cdecl main()
```

```
{
HRESULT         hr = S_OK;
LPTSTR          szReaders, szRdr;
DWORD           cchReaders = SCARD_AUTOALLOCATE;
DWORD           dwI, dwRdrCount;
SCARD_READERSTATE rgscState[MAXIMUM_SMARTCARD_READERS];
TCHAR           szCard[MAX_PATH];
SCARDCONTEXT    hSC;
LONG            lReturn;
// Establish the card to watch for.
// Multiple cards can be looked for, but
// this sample looks for only one card.
_tcscat_s ( szCard, MAX_PATH * sizeof(TCHAR), TEXT("GemSAFE"));
szCard[lstrlen(szCard) + 1] = 0;  // Double trailing zero.
// Establish a context.
lReturn = SCardEstablishContext(SCARD_SCOPE_USER,
                        NULL,
                        NULL,
                        &hSC );
if ( SCARD_S_SUCCESS != lReturn )
{
   printf("Failed SCardEstablishContext\n");
   exit(1);
}
// Determine which readers are available.
lReturn = SCardListReaders(hSC,
                      NULL,
                      (LPTSTR)&szReaders,
                      &cchReaders );
if ( SCARD_S_SUCCESS != lReturn )
{
   printf("Failed SCardListReaders\n");
   exit(1);
}
// Place the readers into the state array.
szRdr = szReaders;
for ( dwI = 0; dwI < MAXIMUM_SMARTCARD_READERS; dwI++ )
{
   if ( 0 == *szRdr )
```

```
    break;
  rgscState[dwI].szReader = szRdr;
  rgscState[dwI].dwCurrentState = SCARD_STATE_UNAWARE;
  szRdr += lstrlen(szRdr) + 1;
}
dwRdrCount = dwI;
// If any readers are available, proceed.
if ( 0 != dwRdrCount )
{
 for (;;)
 {
  // Look for the card.
  lReturn = SCardLocateCards(hSC,
                szCard,
                rgscState,
                dwRdrCount );
  if ( SCARD_S_SUCCESS != lReturn )
  {
    printf("Failed SCardLocateCards\n");
    exit(1);
  }
  // Look through the array of readers.
  for ( dwI=0; dwI < dwRdrCount; dwI++)
  {
    if ( 0 != ( SCARD_STATE_ATRMATCH &
          rgscState[dwI].dwEventState))
    {
     _tprintf( TEXT("Card '%s' found in reader '%s'.\n"),
           szCard,
           rgscState[dwI].szReader );
      SCardFreeMemory( hSC,
               szReaders );
      return 0;  // Context will be release automatically.
    }
    // Update the state.
    rgscState[dwI].dwCurrentState = rgscState[dwI].dwEventState;
  }
 // Card not found yet; wait until there is a change.
 lReturn = SCardGetStatusChange(hSC,
                         INFINITE, // infinite wait
                         rgscState,
                         dwRdrCount );
  if ( SCARD_S_SUCCESS != lReturn )
  {
   printf("Failed SCardGetStatusChange\n");
   exit(1);
  }
 }  // for (;;)
}
else
   printf("No readers available\n");
// Release the context.
lReturn = SCardReleaseContext(hSC);
if ( SCARD_S_SUCCESS != lReturn )
{
   printf("Failed SCardReleaseContext\n");
   exit(1);
}
SCardFreeMemory( hSC,
             szReaders );
return hr;
}
```

SCardGetStatusChange Function

The SCardGetStatusChange function blocks execution until the current availability of the cards in a specific set of readers changes.

The caller supplies a list of readers to be monitored by an SCARD_READERSTATE array and the maximum amount of time (in milliseconds) that it is willing to wait for an action to occur on one of the listed readers. Note that SCardGetStatusChange uses the user-supplied value in the dwCurrentState members of the rgReaderStates SCARD_READERSTATE array as the definition of the current state of the readers. The function returns when there is a change in availability, having filled in the dwEventState members of rgReaderStates appropriately.

Syntax:

```
LONG WINAPI SCardGetStatusChange(
  __in     SCARDCONTEXT hContext,
  __in     DWORD dwTimeout,
  __inout  LPSCARD_READERSTATE rgReaderStates,
  __in     DWORD cReaders
);
```

Parameters:

hContext [in]

> A handle that identifies the resource manager context. The resource manager context is set by a previous call to the SCardEstablishContext function.

dwTimeout [in]

> The maximum amount of time, in milliseconds, to wait for an action. A value of zero causes the function to return immediately. A value of INFINITE causes this function never to time out.

rgReaderStates [in, out]

> An array of SCARD_READERSTATE structures that specify the readers to watch, and that receives the result.
>
> To be notified of the arrival of a new smart card reader, set the szReader member of a SCARD_READERSTATE structure to "\\\\?PnP?\\Notification", and set all of the other members of that structure to zero.
>
> Important: Each member of each structure in this array must be initialized to zero and then set to specific values as necessary. If this is not done, the function will fail in situations that involve remote card readers.

cReaders [in]

> The number of elements in the rgReaderStates array.

Return Value:

This function returns different values depending on whether it succeeds or fails:

> Success: SCARD_S_SUCCESS.
>
> Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

The SCardGetStatusChange function is a smart card tracking function. For more information about other tracking functions, see Smart Card Tracking Functions.

Examples:

For information about how to call this function, see the example in SCardLocateCards.

SCardCancel Function

The SCardCancel function terminates all outstanding actions within a specific resource manager context.

The only requests that you can cancel are those that require waiting for external action by the smart card or user. Any such outstanding action requests will terminate with a status indication that the action was canceled. This is especially useful to force outstanding SCardGetStatusChange calls to terminate.

Syntax:

```
LONG WINAPI SCardCancel(
  __in  SCARDCONTEXT hContext
);
```

Parameters:

hContext [in]

Handle that identifies the resource manager context. The resource manager context is set by a previous call to SCardEstablishContext.

Return Value:

This function returns different values depending on whether it succeeds or fails:

Success: SCARD_S_SUCCESS.

Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

The SCardCancel function is a smart card tracking function. For a description of other tracking functions, see Smart Card Tracking Functions.

Examples:

The following example cancels all outstanding actions in the specified context. The example assumes that lReturn is an existing variable of type LONG and that hContext is a valid handle received from a previous call to SCardEstablishContext.

```
lReturn = SCardCancel( hContext );
if ( SCARD_S_SUCCESS != lReturn )
   printf("Failed SCardCancel\n");
```

SCardConnect Function

The SCardConnect function establishes a connection (using a specific resource manager context) between the calling application and a smart card contained by a specific reader. If no card exists in the specified reader, an error is returned.

Syntax:

```
LONG WINAPI SCardConnect(
  __in   SCARDCONTEXT hContext,
  __in   LPCTSTR szReader,
  __in   DWORD dwShareMode,
  __in   DWORD dwPreferredProtocols,
  __out  LPSCARDHANDLE phCard,
  __out  LPDWORD pdwActiveProtocol
);
```

Parameters:

hContext [in]

A handle that identifies the resource manager context. The resource manager context is set by a previous call to SCardEstablishContext.

szReader [in]

The name of the reader that contains the target card.

dwShareMode [in]

A flag that indicates whether other applications may form connections to the card.

SCARD_SHARE_SHARED: This application is willing to share the card with other applications.

SCARD_SHARE_EXCLUSIVE: This application is not willing to share the card with other applications.

SCARD_SHARE_DIRECT: This application is allocating the reader for its private use, and will be controlling it directly. No other applications are allowed access to it.

dwPreferredProtocols [in]

A bitmask of acceptable protocols for the connection. Possible values may be combined with the OR operation.

SCARD_PROTOCOL_T0: T=0 is an acceptable protocol.

SCARD_PROTOCOL_T1: T=1 is an acceptable protocol.

0: This parameter may be zero only if dwShareMode is set to SCARD_SHARE_DIRECT. In this case, no protocol negotiation will be performed by the drivers until an IOCTL_SMARTCARD_SET_PROTOCOL control directive is sent with SCardControl.

phCard [out]

A handle that identifies the connection to the smart card in the designated reader.

pdwActiveProtocol [out]

A flag that indicates the established active protocol.

SCARD_PROTOCOL_T0: T=0 is the active protocol.

SCARD_PROTOCOL_T1: T=1 is the active protocol.

SCARD_PROTOCOL_UNDEFINED: SCARD_SHARE_DIRECT has been specified, so that no protocol negotiation has occurred. It is possible that there is no card in the reader.

Return Value:

This function returns different values depending on whether it succeeds or fails:

Success: SCARD_S_SUCCESS.

Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

The SCardConnect function is a smart card and reader access function. For more information about other access functions, see Smart Card and Reader Access Functions.

Examples:

The following example creates a connection to a reader. The example assumes that hContext is a valid handle of type SCARDCONTEXT received from a previous call to SCardEstablishContext.

```
SCARDHANDLE     hCardHandle;
LONG            lReturn;
DWORD           dwAP;
lReturn = SCardConnect( hContext,
                        (LPCTSTR)"Rainbow Technologies SCR3531 0",
                        SCARD_SHARE_SHARED,
                        SCARD_PROTOCOL_T0 | SCARD_PROTOCOL_T1,
                        &hCardHandle,
                        &dwAP );
if ( SCARD_S_SUCCESS != lReturn )
{
    printf("Failed SCardConnect\n");
    exit(1);  // Or other appropriate action.
}
// Use the connection.
// Display the active protocol.
switch ( dwAP )
{
    case SCARD_PROTOCOL_T0:
        printf("Active protocol T0\n");
        break;
    case SCARD_PROTOCOL_T1:
        printf("Active protocol T1\n");
        break;
    case SCARD_PROTOCOL_UNDEFINED:
    default:
        printf("Active protocol unnegotiated or unknown\n");
        break;
}
// Remember to disconnect (by calling SCardDisconnect).
// ...
```

SCardReconnect Function

The SCardReconnect function reestablishes an existing connection between the calling application and a smart card. This function moves a card handle from direct access to general access, or acknowledges and clears an error condition that is preventing further access to the card.

Syntax:

```
LONG WINAPI SCardReconnect(
  __in      SCARDHANDLE hCard,
  __in      DWORD dwShareMode,
  __in      DWORD dwPreferredProtocols,
  __in      DWORD dwInitialization,
  __out_opt  LPDWORD pdwActiveProtocol
);
```

Parameters:

hCard [in]

Reference value obtained from a previous call to SCardConnect.

dwShareMode [in]

Flag that indicates whether other applications may form connections to this card:

SCARD_SHARE_SHARED: This application will share this card with other applications.

SCARD_SHARE_EXCLUSIVE: This application will not share this card with other applications.

dwPreferredProtocols [in]

Bitmask of acceptable protocols for this connection. Possible values may be combined with the OR operation. The value of this parameter should include the current protocol. Attempting to reconnect with a protocol other than the current protocol will result in an error.

SCARD_PROTOCOL_T0: T=0 is an acceptable protocol.

SCARD_PROTOCOL_T1: T=1 is an acceptable protocol.

dwInitialization [in]

Type of initialization that should be performed on the card:

SCARD_LEAVE_CARD: Do not do anything special on reconnect.

SCARD_RESET_CARD: Reset the card (Warm Reset).

SCARD_UNPOWER_CARD: Power down the card and reset it (Cold Reset).

pdwActiveProtocol [out, optional]

Flag that indicates the established active protocol:

SCARD_PROTOCOL_T0: T=0 is the active protocol.

SCARD_PROTOCOL_T1: T=1 is the active protocol.

Return Value:

This function returns different values depending on whether it succeeds or fails.

Success: SCARD_S_SUCCESS.

Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

SCardReconnect is a smart card and reader access function. For information about other access functions, see Smart Card and Reader Access Functions.

Examples:

The following example shows reestablishing a connection.

```
DWORD    dwAP;
LONG     lReturn;

// Reconnect.
// hCardHandle was set by a previous call to SCardConnect.
lReturn = SCardReconnect(hCardHandle,
                         SCARD_SHARE_SHARED,
                         SCARD_PROTOCOL_T0 | SCARD_PROTOCOL_T1,
                         SCARD_LEAVE_CARD,
                         &dwAP );
if ( SCARD_S_SUCCESS != lReturn )
            printf("Failed SCardReconnect\n")
```

SCardDisconnect Function

The SCardDisconnect function terminates a connection previously opened between the calling application and a smart card in the target reader.

Syntax:

```
LONG WINAPI SCardDisconnect(
  __in  SCARDHANDLE hCard,
  __in  DWORD dwDisposition
);
```

Parameters:

hCard [in]

Reference value obtained from a previous call to SCardConnect.

dwDisposition [in]

Action to take on the card in the connected reader on close:

SCARD_LEAVE_CARD: Do not do anything special.

SCARD_RESET_CARD: Reset the card.

SCARD_UNPOWER_CARD: Power down the card.

SCARD_EJECT_CARD: Eject the card.

Return Value:

This function returns different values depending on whether it succeeds or fails:

Success: SCARD_S_SUCCESS.

Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

If an application (which previously called SCardConnect) exits without calling SCardDisconnect, the card is automatically reset.

The SCardDisconnect function is a smart card and reader access function. For more information on other access functions, see Smart Card and Reader Access Functions.

Examples:

The following example terminates the specified smart card connection. The example assumes that lReturn is a variable of type LONG, and that hCardHandle is a valid handle received from a previous call to SCardConnect.

```
lReturn = SCardDisconnect(hCardHandle,
                          SCARD_LEAVE_CARD);
if ( SCARD_S_SUCCESS != lReturn )
{
   printf("Failed SCardDisconnect\n");
   exit(1);  // Or other appropriate action.
}
```

SCardBeginTransaction Function

The SCardBeginTransaction function starts a transaction.

The function waits for the completion of all other transactions before it begins. After the transaction starts, all other applications are blocked from accessing the smart card while the transaction is in progress.

Syntax:

```
LONG WINAPI SCardBeginTransaction(
  __in  SCARDHANDLE hCard
);
```

Parameters:

hCard [in]

A reference value obtained from a previous call to SCardConnect.

Return Value:

If the function succeeds, it returns SCARD_S_SUCCESS.

If the function fails, it returns an error code. For more information, see Smart Card Return Values.

Note: This function returns SCARD_S_SUCCESS even if another process or thread has reset the card. To determine whether the card has been reset, call the SCardStatus function immediately after calling this function.

Remarks:

The SCardBeginTransaction function is a smart card and reader access function. For more information on other access functions, see Smart Card and Reader Access Functions.

Examples:

The following example demonstrates how to begin a smart card transaction. The example assumes that lReturn is an existing variable of type LONG and that hCard is a valid handle received from a previous call to SCardConnect.

```
lReturn = SCardBeginTransaction( hCard );
if ( SCARD_S_SUCCESS != lReturn )
 printf("Failed SCardBeginTransaction\n");
```

SCardEndTransaction Function

The SCardEndTransaction function completes a previously declared transaction, allowing other applications to resume interactions with the card.

Syntax:

```
LONG WINAPI SCardEndTransaction(
  __in  SCARDHANDLE hCard,
  __in  DWORD dwDisposition
);
```

Parameters:

hCard [in]

Reference value obtained from a previous call to SCardConnect. This value would also have been used in an earlier call to SCardBeginTransaction.

dwDisposition [in]

Action to take on the card in the connected reader on close:

SCARD_EJECT_CARD: Eject the card.

SCARD_LEAVE_CARD: Do not do anything special.

SCARD_RESET_CARD: Reset the card.

SCARD_UNPOWER_CARD: Power down the card.

Return Value:

If the function succeeds, the function returns SCARD_S_SUCCESS.

If the function fails, it returns an error code. For more information, see Smart Card Return Values. Possible error codes follow:

SCARD_W_RESET_CARD (0x80100068L)

The transaction was released. Any future communication with the card requires a call to the SCardReconnect function.

Windows Server 2008, Windows Vista, Windows Server 2003, Windows XP, and Windows 2000: The transaction was not released. The application must immediately call the SCardDisconnect, SCardReconnect, or SCardReleaseContext function to avoid an existing transaction blocking other threads and processes from communicating with the smart card.

Remarks:

The SCardEndTransaction function is a smart card and reader access function. For more information on other access functions, see Smart Card and Reader Access Functions.

Examples:

The following example ends a smart card transaction. The example assumes that lReturn is a valid variable of type LONG, that hCard is a valid handle received from a previous call to the SCardConnect function, and that hCard was passed to a previous call to the SCardBeginTransaction function.

```
lReturn = SCardEndTransaction(hCard,
                              SCARD_LEAVE_CARD);
if ( SCARD_S_SUCCESS != lReturn )
```

```
    printf("Failed SCardEndTransaction\n");
```

SCardStatus Function

The SCardStatus function provides the current status of a smart card in a reader. You can call it any time after a successful call to SCardConnect and before a successful call to SCardDisconnect. It does not affect the state of the reader or reader driver.

Syntax:

```
LONG WINAPI SCardStatus(
  __in         SCARDHANDLE hCard,
  __out        LPTSTR szReaderName,
  __inout_opt  LPDWORD pcchReaderLen,
  __out_opt    LPDWORD pdwState,
  __out_opt    LPDWORD pdwProtocol,
  __out        LPBYTE pbAtr,
  __inout_opt  LPDWORD pcbAtrLen
);
```

Parameters:

hCard [in]

Reference value returned from SCardConnect.

szReaderName [out]

List of display names (multiple string) by which the currently connected reader is known.

pcchReaderLen [in, out, optional]

On input, supplies the length of the szReaderName buffer.

On output, receives the actual length (in characters) of the reader name list, including the trailing NULL character. If this buffer length is specified as SCARD_AUTOALLOCATE, then szReaderName is converted to a pointer to a byte pointer, and it receives the address of a block of memory that contains the multiple-string structure.

pdwState [out, optional]

Current state of the smart card in the reader. Upon success, it receives one of the following state indicators:

SCARD_ABSENT: There is no card in the reader.

SCARD_PRESENT: There is a card in the reader, but it has not been moved into position for use.

SCARD_SWALLOWED: There is a card in the reader in position for use. The card is not powered.

SCARD_POWERED: Power is being provided to the card, but the reader driver is unaware of the mode of the card.

SCARD_NEGOTIABLE: The card has been reset and is awaiting PTS negotiation.

SCARD_SPECIFIC: The card has been reset and specific communication protocols have been established.

pdwProtocol [out, optional]

Current protocol, if any. The returned value is meaningful only if the returned value of pdwState is SCARD_SPECIFICMODE:

SCARD_PROTOCOL_RAW: The Raw Transfer protocol is in use.

SCARD_PROTOCOL_T0: The ISO 7816/3 T=0 protocol is in use.

SCARD_PROTOCOL_T1: The ISO 7816/3 T=1 protocol is in use.

pbAtr [out]

Pointer to a 32-byte buffer that receives the ATR string from the currently inserted card, if available.

pcbAtrLen [in, out, optional]

On input, supplies the length of the pbAtr buffer.

On output, receives the number of bytes in the ATR string (32 bytes maximum). If this buffer length is specified as SCARD_AUTOALLOCATE, then pbAtr is converted to a pointer to a byte pointer, and it receives the address of a block of memory that contains the multiple-string structure.

Return Value:

If the function successfully provides the current status of a smart card in a reader, the return value is SCARD_S_SUCCESS.

If the function fails, it returns an error code. For more information, see Smart Card Return Values.

Remarks:

The szReaderName function is a smart card and reader access function. For information about other access functions, see Smart Card and Reader Access Functions.

Examples:

The following example shows how to determine the state of the smart card.

```
WCHAR       szReader[200];
DWORD       cch = 200;
BYTE        bAttr[32];
DWORD       cByte = 32;
DWORD       dwState, dwProtocol;
LONG        lReturn;
// Determine the status.
// hCardHandle was set by an earlier call to SCardConnect.
lReturn = SCardStatus(hCardHandle,
                      szReader,
                      &cch,
                      &dwState,
                      &dwProtocol,
                      (LPBYTE)&bAttr,
                      &cByte);
if ( SCARD_S_SUCCESS != lReturn )
{
   printf("Failed SCardStatus\n");
```

```
    exit(1);     // or other appropriate action
}
// Examine retrieved status elements.
// Look at the reader name and card state.
printf("%S\n", szReader );
switch ( dwState )
{
    case SCARD_ABSENT:
        printf("Card absent.\n");
        break;
    case SCARD_PRESENT:
        printf("Card present.\n");
        break;
    case SCARD_SWALLOWED:
        printf("Card swallowed.\n");
        break;
    case SCARD_POWERED:
        printf("Card has power.\n");
        break;
    case SCARD_NEGOTIABLE:
        printf("Card reset and waiting PTS negotiation.\n");
        break;
    case SCARD_SPECIFIC:
        printf("Card has specific communication protocols set.\n");
        break;
    default:
        printf("Unknown or unexpected card state.\n");
        break;
}
```

SCardTransmit Function

The SCardTransmit function sends a service request to the smart card and expects to receive data back from the card.

Syntax:

```
LONG WINAPI SCardTransmit(
  __in        SCARDHANDLE hCard,
  __in        LPCSCARD_IO_REQUEST pioSendPci,
  __in        LPCBYTE pbSendBuffer,
  __in        DWORD cbSendLength,
  __inout_opt  LPSCARD_IO_REQUEST pioRecvPci,
  __out        LPBYTE pbRecvBuffer,
  __inout      LPDWORD pcbRecvLength
);
```

Parameters:

hCard [in]

A reference value returned from the SCardConnect function.

pioSendPci [in]

A pointer to the protocol header structure for the instruction. This buffer is in the format of an SCARD_IO_REQUEST structure, followed by the specific protocol control information (PCI).

For the T=0, T=1, and Raw protocols, the PCI structure is constant. The smart card subsystem supplies a global T=0, T=1, or Raw PCI structure, which you can reference by using the symbols SCARD_PCI_T0, SCARD_PCI_T1, and SCARD_PCI_RAW respectively.

pbSendBuffer [in]

A pointer to the actual data to be written to the card.

For T=0, the data parameters are placed into the address pointed to by pbSendBuffer according to the following structure:

```
struct {
   BYTE
      bCla,   // the instruction class
      bIns,   // the instruction code
      bP1,    // parameter to the instruction
      bP2,    // parameter to the instruction
      bP3;    // size of I/O transfer
} CmdBytes;
```

The data sent to the card should immediately follow the send buffer. In the special case where no data is sent to the card and no data is expected in return, bP3 is not sent.

Member Meaning:

bCla The T=0 instruction class.

bIns An instruction code in the T=0 instruction class.

bP1, bP2 Reference codes that complete the instruction code.

bP3 The number of data bytes to be transmitted during the command, per ISO 7816-4, Section 8.2.1.

cbSendLength [in]

The length, in bytes, of the pbSendBuffer parameter.

For T=0, in the special case where no data is sent to the card and no data expected in return, this length must reflect that the bP3 member is not being sent; the length should be sizeof(CmdBytes) - sizeof(BYTE).

pioRecvPci [in, out, optional]

Pointer to the protocol header structure for the instruction, followed by a buffer in which to receive any returned protocol control information (PCI) specific to the protocol in use. This parameter can be NULL if no PCI is returned.

pbRecvBuffer [out]

Pointer to any data returned from the card.

For T=0, the data is immediately followed by the SW1 and SW2 status bytes. If no data is returned from the card, then this buffer will only contain the SW1 and SW2 status bytes.

pcbRecvLength [in, out]

Supplies the length, in bytes, of the pbRecvBuffer parameter and receives the actual number of bytes received from the smart card. This value cannot be SCARD_AUTOALLOCATE because SCardTransmit does not support SCARD_AUTOALLOCATE.

For T=0, the receive buffer must be at least two bytes long to receive the SW1 and SW2 status bytes.

Return Value:

If the function successfully sends a service request to the smart card, the return value is SCARD_S_SUCCESS.

If the function fails, it returns an error code. For more information, see Smart Card Return Values.

Remarks:

The SCardTransmit function is a smart card and reader access function. For information about other access functions, see Smart Card and Reader Access Functions.

For the T=0 protocol, the data received back are the SW1 and SW2 status codes, possibly preceded by response data. The following paragraphs provide information about the send and receive buffers used to transfer data and issue a command.

Sending data to the card :

To send n bytes of data to the card, where n>0, the send and receive buffers must be formatted as follows.

The first four bytes of the pbSendBuffer buffer contain the CLA, INS, P1, and P2 values for the T=0 operation. The fifth byte must be set to n: the size, in bytes, of the data to be transferred to the card. The next n bytes must contain the data to be sent to the card.

The cbSendLength parameter must be set to the size of the T=0 header information (CLA, INS, P1, and P2) plus a byte that contains the length of the data to be transferred (n), plus the size of data to be sent. In this example, this is n+5.

The pbRecvBuffer will receive the SW1 and SW2 status codes from the operation.

The pcbRecvLength should be at least two and will be set to two upon return.

Retrieving data from the card:

To receive n>0 bytes of data from the card, the send and receive buffers must be formatted as follows.

The first four bytes of the pbSendBuffer buffer contain the CLA, INS, P1, and P2 values for the T=0 operation. The fifth byte must be set to n: the size, in bytes, of the data to be transferred from the card. If 256 bytes are to be transferred from the card, then this byte must be set to zero.

The cbSendLength parameter must be set to five, the size of the T=0 header information.

The pbRecvBuffer will receive the data returned from the card, immediately followed by the SW1 and SW2 status codes from the operation.

The pcbRecvLength should be at least n+2 and will be set to n+2 upon return.

Issuing a command without exchanging data:

To issue a command to the card that does not involve the exchange of data (either sent or received), the send and receive buffers must be formatted as follows.

The pbSendBuffer buffer must contain the CLA, INS, P1, and P2 values for the T=0 operation. The P3 value is not sent. (This is to differentiate the header from the case where 256 bytes are expected to be returned.)

The cbSendLength parameter must be set to four, the size of the T=0 header information (CLA, INS, P1, and P2).

The pbRecvBuffer will receive the SW1 and SW2 status codes from the operation.

The pcbRecvLength should be at least two and will be set to two upon return.

Examples:

The following example shows sending a service request to the smart card.

```
//  Transmit the request.
//  lReturn is of type LONG.
//  hCardHandle was set by a previous call to SCardConnect.
//  pbSend points to the buffer of bytes to send.
//  dwSend is the DWORD value for the number of bytes to send.
//  pbRecv points to the buffer for returned bytes.
//  dwRecv is the DWORD value for the number of returned bytes.
lReturn = SCardTransmit(hCardHandle,
                SCARD_PCI_T0,
                pbSend,
                dwSend,
                NULL,
                pbRecv,
                &dwRecv );
if ( SCARD_S_SUCCESS != lReturn )
```

```
{
    printf("Failed SCardTransmit\n");
    exit(1);    // or other appropriate error action
}
```

SCardControl Function

The SCardControl function gives you direct control of the reader. You can call it any time after a successful call to SCardConnect and before a successful call to SCardDisconnect. The effect on the state of the reader depends on the control code.

Syntax:

```
LONG WINAPI SCardControl(
  __in   SCARDHANDLE hCard,
  __in   DWORD dwControlCode,
  __in   LPCVOID lpInBuffer,
  __in   DWORD nInBufferSize,
  __out  LPVOID lpOutBuffer,
  __in   DWORD nOutBufferSize,
  __out  LPDWORD lpBytesReturned
);
```

Parameters:

hCard [in]

Reference value returned from SCardConnect.

dwControlCode [in]

Control code for the operation. This value identifies the specific operation to be performed.

lpInBuffer [in]

Pointer to a buffer that contains the data required to perform the operation. This parameter can be NULL if the dwControlCode parameter specifies an operation that does not require input data.

nInBufferSize [in]

Size, in bytes, of the buffer pointed to by lpInBuffer.

lpOutBuffer [out]

Pointer to a buffer that receives the operation's output data. This parameter can be NULL if the dwControlCode parameter specifies an operation that does not produce output data.

nOutBufferSize [in]

Size, in bytes, of the buffer pointed to by lpOutBuffer.

lpBytesReturned [out]

Pointer to a DWORD that receives the size, in bytes, of the data stored into the buffer pointed to by lpOutBuffer.

Return Value:

This function returns different values depending on whether it succeeds or fails:

Success: SCARD_S_SUCCESS.

Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

The SCardControl function is a direct card access function. For more information on other direct access functions, see Direct Card Access Functions.

Examples:

The following example issues a control code. The example assumes that hCardHandle is a valid handle received from a previous call to SCardConnect and that dwControlCode is a variable of type DWORD previously initialized to a valid control code. This particular control code requires no input data and expects no output data.

```
lReturn = SCardControl( hCardHandle,
                        dwControlCode,
                        NULL,
                        0,
                        NULL,
                        0,
                        0 );
if ( SCARD_S_SUCCESS != lReturn )
    printf("Failed SCardControl\n");
```

SCardGetAttrib Function

The SCardGetAttrib function retrieves the current reader attributes for the given handle. It does not affect the state of the reader, driver, or card.

Syntax:

```
LONG WINAPI SCardGetAttrib(
  __in     SCARDHANDLE hCard,
  __in     DWORD dwAttrId,
  __out    LPBYTE pbAttr,
  __inout  LPDWORD pcbAttrLen
);
```

Parameters:

hCard [in]

Reference value returned from SCardConnect.

dwAttrId [in]

Identifier for the attribute to get. The following table lists possible values for dwAttrId. These values are read-only. Note that vendors may not support all attributes:

SCARD_ATTR_ATR_STRING: Answer to reset (ATR) string.

SCARD_ATTR_CHANNEL_ID: DWORD encoded as 0xDDDDCCCC, where DDDD = data channel type and CCCC = channel number:

The following encodings are defined for DDDD:

0x01 serial I/O; CCCC is a port number.

0x02 parallel I/O; CCCC is a port number.

0x04 PS/2 keyboard port; CCCC is zero.

0x08 SCSI; CCCC is SCSI ID number.

0x10 IDE; CCCC is device number.

0x20 USB; CCCC is device number.

0xFy vendor-defined interface with y in the range zero through 15; CCCC is vendor defined.

SCARD_ATTR_CHARACTERISTICS: DWORD indicating which mechanical characteristics are supported. If zero, no special characteristics are supported. Note that multiple bits can be set:

0x00000001 Card swallowing mechanism

0x00000002 Card ejection mechanism

0x00000004 Card capture mechanism

All other values are reserved for future use (RFU).

SCARD_ATTR_CURRENT_BWT: Current block waiting time.

SCARD_ATTR_CURRENT_CLK: Current clock rate, in kHz.

SCARD_ATTR_CURRENT_CWT: Current character waiting time.

SCARD_ATTR_CURRENT_D: Bit rate conversion factor.

SCARD_ATTR_CURRENT_EBC_ENCODING: Current error block control encoding.

0 = longitudinal redundancy check (LRC)

1 = cyclical redundancy check (CRC)

SCARD_ATTR_CURRENT_F: Clock conversion factor.

SCARD_ATTR_CURRENT_IFSC: Current byte size for information field size card.

SCARD_ATTR_CURRENT_IFSD: Current byte size for information field size device.

SCARD_ATTR_CURRENT_N: Current guard time.

SCARD_ATTR_CURRENT_PROTOCOL_TYPE: DWORD encoded as 0x0rrrpppp where rrr is RFU and should be 0x000. pppp encodes the current protocol type. Whichever bit has been set indicates which ISO protocol is currently in use. (For example, if bit zero is set, T=0 protocol is in effect.)

SCARD_ATTR_CURRENT_W: Current work waiting time.

SCARD_ATTR_DEFAULT_CLK: Default clock rate, in kHz.

SCARD_ATTR_DEFAULT_DATA_RATE: Default data rate, in bps.

SCARD_ATTR_DEVICE_FRIENDLY_NAME: Reader's display name.

SCARD_ATTR_DEVICE_IN_USE: Reserved for future use.

SCARD_ATTR_DEVICE_SYSTEM_NAME: Reader's system name.

SCARD_ATTR_DEVICE_UNIT: Instance of this vendor's reader attached to the computer. The first instance will be device unit 0, the next will be unit 1 (if it is the same brand of reader) and so on. Two different brands of readers will both have zero for this value.

SCARD_ATTR_ICC_INTERFACE_STATUS: Single byte. Zero if smart card electrical contact is not active; nonzero if contact is active.

SCARD_ATTR_ICC_PRESENCE: Single byte indicating smart card presence:

0 = not present

1 = card present but not swallowed (applies only if reader supports smart card swallowing)

2 = card present (and swallowed if reader supports smart card swallowing)

4 = card confiscated.

SCARD_ATTR_ICC_TYPE_PER_ATR: Single byte indicating smart card type:

0 = unknown type

1 = 7816 Asynchronous

2 = 7816 Synchronous

Other values RFU.

SCARD_ATTR_MAX_CLK: Maximum clock rate, in kHz.

SCARD_ATTR_MAX_DATA_RATE: Maximum data rate, in bps.

SCARD_ATTR_MAX_IFSD: Maximum bytes for information file size device.

SCARD_ATTR_POWER_MGMT_SUPPORT: Zero if device does not support power down while smart card is inserted. Nonzero otherwise.

SCARD_ATTR_PROTOCOL_TYPES: DWORD encoded as 0x0rrrpppp where rrr is RFU and should be 0x000. pppp encodes the supported protocol types. A '1' in a given bit position indicates support for the associated ISO protocol, so if bits zero and one are set, both T=0 and T=1 protocols are supported.

SCARD_ATTR_VENDOR_IFD_SERIAL_NO: Vendor-supplied interface device serial number.

SCARD_ATTR_VENDOR_IFD_TYPE: Vendor-supplied interface device type (model designation of reader).

SCARD_ATTR_VENDOR_IFD_VERSION: Vendor-supplied interface device version (DWORD in the form 0xMMmmbbbb where MM = major version, mm = minor version, and bbbb = build number).

SCARD_ATTR_VENDOR_NAME: Vendor name.

pbAttr [out]

Pointer to a buffer that receives the attribute whose ID is supplied in dwAttrId. If this value is NULL, SCardGetAttrib ignores the buffer length supplied in pcbAttrLen, writes the length of the buffer that would have been returned if this parameter had not been NULL to pcbAttrLen, and returns a success code.

pcbAttrLen [in, out]

Length of the pbAttr buffer in bytes, and receives the actual length of the received attribute If the buffer length is specified as SCARD_AUTOALLOCATE, then pbAttr is converted to a pointer to a byte pointer, and receives the address of a block of memory containing the attribute. This block of memory must be deallocated with SCardFreeMemory.

Return Value:

This function returns different values depending on whether it succeeds or fails:

Success: SCARD_S_SUCCESS.

Attribute value not supported: ERROR_NOT_SUPPORTED.

Other Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

The SCardGetAttrib function is a direct card access function. For more information on other direct access functions, see Direct Card Access Functions.

Examples:

The following example shows how to retrieve an attribute for a card reader. The example assumes that hCardHandle is a valid handle obtained from a previous call to the SCardConnect function.

```
LPBYTE   pbAttr = NULL;
DWORD    cByte = SCARD_AUTOALLOCATE;
DWORD    i;
LONG     lReturn;
lReturn = SCardGetAttrib(hCardHandle,
                 SCARD_ATTR_VENDOR_NAME,
                 (LPBYTE)&pbAttr,
                 &cByte);
if ( SCARD_S_SUCCESS != lReturn )
```

```
{
    if ( ERROR_NOT_SUPPORTED == lReturn )
        printf("Value not supported\n");
    else
    {
        // Some other error occurred.
        printf("Failed SCardGetAttrib - %x\n", lReturn);
        exit(1);  // Or other appropriate action
    }
}
else
{
    // Output the bytes.
    for (i = 0; i < cByte; i++)
        printf("%c", *(pbAttr+i));
    printf("\n");
    // Free the memory when done.
    // hContext was set earlier by SCardEstablishContext
    lReturn = SCardFreeMemory( hContext, pbAttr );
}
```

SCardSetAttrib Function

The SCardSetAttrib function sets the given reader attribute for the given handle. It does not affect the state of the reader, reader driver, or smart card. Not all attributes are supported by all readers (nor can they be set at all times) as many of the attributes are under direct control of the transport protocol.

Syntax:

```
LONG WINAPI SCardSetAttrib(
  __in  SCARDHANDLE hCard,
  __in  DWORD dwAttrId,
  __in  LPCBYTE pbAttr,
  __in  DWORD cbAttrLen
);
```

Parameters:

hCard [in]

Reference value returned from SCardConnect.

dwAttrId [in]

Identifier for the attribute to set. The values are write-only. Note that vendors may not support all attributes:

SCARD_ATTR_SUPRESS_T1_IFS_REQUEST: Suppress sending of T=1 IFSD packet from the reader to the card. (Can be used if the currently inserted card does not support an IFSD request.)

pbAttr [in]

Pointer to a buffer that supplies the attribute whose ID is supplied in dwAttrId.

cbAttrLen [in]

Length (in bytes) of the attribute value in the pbAttr buffer.

Return Value:

This function returns different values depending on whether it succeeds or fails:

Success: SCARD_S_SUCCESS.

Failure: An error code. For more information, see Smart Card Return Values.

Remarks:

The SCardSetAttrib function is a direct card access function. For information about other direct access functions, see Direct Card Access Functions.

Examples:

The following example shows how to set an attribute.

```
// Set the attribute.
// hCardHandle was set by a previous call to SCardConnect.
// dwAttrID is a DWORD value, specifying the attribute ID.
// pbAttr points to the buffer of the new value.
// cByte is the count of bytes in the buffer.
lReturn = SCardSetAttrib(hCardHandle,
                dwAttrID,
                (LPBYTE)pbAttr,
                cByte);
if ( SCARD_S_SUCCESS != lReturn )
   printf("Failed SCardSetAttrib\n");
```

SCard Return Values

Smart Card Functions return the following return values. These return values are defined in Scarderr.h.

Note: Some return values may have the same value as existing Windows return values that signify a similar condition.

Error Code	Hexadecimal value	Description
SCARD_F_INTERNAL_ERROR	0x80100001	An internal consistency check failed
SCARD_E_CANCELLED	0x80100002	The action was cancelled by a SCardCancel request
SCARD_E_INVALID_HANDLE	0x80100003	The supplied handle was invalid
SCARD_E_INVALID_PARAMETER	0x80100004	One or more of the supplied parameters could not be properly interpreted
SCARD_E_INVALID_TARGET	0x80100005	Registry startup information is missing or invalid
SCARD_E_NO_MEMORY	0x80100006	Not enough memory available to complete this command
SCARD_F_WAITED_TOO_LONG	0x80100007	An internal consistency timer has expired
SCARD_E_INSUFFICIENT_BUFFER	0x80100008	The data buffer to receive returned data is too small for the returned data
SCARD_E_UNKNOWN_READER	0x80100009	The specified reader name is not recognized
SCARD_E_TIMEOUT	0x8010000A	The user-specified timeout value has expired
SCARD_E_SHARING_VIOLATION	0x8010000B	The smart card cannot be accessed because of other connections outstanding
SCARD_E_NO_SMARTCARD	0x8010000C	The operation requires a smart card, but no smart card is currently in the device
SCARD_E_UNKNOWN_CARD	0x8010000D	The specified smart card name is not recognized

SCARD_E_CANT_DISPOSE	0x8010000E	The system could not dispose of the media in the requested manner
SCARD_E_PROTO_MISMATCH	0x8010000F	The requested protocols are incompatible with the protocol currently in use with the smart card
SCARD_E_NOT_READY	0x80100010	The reader or smart card is not ready to accept commands
SCARD_E_INVALID_VALUE	0x80100011	One or more of the supplied parameters values could not be properly interpreted
SCARD_E_SYSTEM_CANCELLED	0x80100012	The action was cancelled by the system, presumably to log off or shut down
SCARD_F_COMM_ERROR	0x80100013	An internal communications error has been detected
SCARD_F_UNKNOWN_ERROR	0x80100014	An internal error has been detected, but the source is unknown
SCARD_E_INVALID_ATR	0x80100015	An ATR obtained from the registry is not a valid ATR string
SCARD_E_NOT_TRANSACTED	0x80100016	An attempt was made to end a non-existent transaction
SCARD_E_READER_UNAVAILABLE	0x80100017	The specified reader is not currently available for use
SCARD_P_SHUTDOWN	0x80100018	The operation has been aborted to allow the server application to exit
SCARD_E_PCI_TOO_SMALL	0x80100019	The PCI Receive buffer was too small
SCARD_E_READER_UNSUPPORTED	0x8010001A	The reader driver does not meet minimal requirements for support
SCARD_E_DUPLICATE_READER	0x8010001B	The reader driver did not produce a unique reader name
SCARD_E_CARD_UNSUPPORTED	0x8010001C	The smart card does not meet

		minimal requirements for support
SCARD_E_NO_SERVICE	0x8010001D	The Smart Card Resource Manager is not running
SCARD_E_SERVICE_STOPPED	0x8010001E	The Smart Card Resource Manager has shut down
SCARD_E_UNEXPECTED	0x8010001F	An unexpected card error has occurred
SCARD_E_ICC_INSTALLATION	0x80100020	No primary provider can be found for the smart card
SCARD_E_ICC_CREATEORDER	0x80100021	The requested order of object creation is not supported.
SCARD_E_UNSUPPORTED_FEATURE	0x80100022	This smart card does not support the requested feature
SCARD_E_DIR_NOT_FOUND	0x80100023	The identified directory does not exist in the smart card
SCARD_E_FILE_NOT_FOUND	0x80100024	The identified file does not exist in the smart card
SCARD_E_NO_DIR	0x80100025	The supplied path does not represent a smart card directory
SCARD_E_NO_FILE	0x80100026	The supplied path does not represent a smart card file
SCARD_E_NO_ACCESS	0x80100027	Access is denied to this file
SCARD_E_WRITE_TOO_MANY	0x80100028	The smart card does not have enough memory to store the information
SCARD_E_BAD_SEEK	0x80100029	There was an error trying to set the smart card file object pointer
SCARD_E_INVALID_CHV	0x8010002A	The supplied PIN is incorrect
SCARD_E_UNKNOWN_RES_MNG	0x8010002B	An unrecognized error code was returned from a layered component
SCARD_E_NO_SUCH_CERTIFICATE	0x8010002C	The requested certificate does not exist
SCARD_E_CERTIFICATE_UNAVAILABLE	0x8010002D	The requested certificate could

		not be obtained
SCARD_E_NO_READERS_AVAILABLE	0x8010002E	Cannot find a smart card reader
SCARD_E_COMM_DATA_LOST	0x8010002F	A communications error with the smart card has been detected. Retry the operation
SCARD_E_NO_KEY_CONTAINER	0x80100030L	The requested key container does not exist on the smart card
SCARD_E_SERVER_TOO_BUSY	0x80100031	The Smart Card Resource Manager is too busy to complete this operation
SCARD_W_UNSUPPORTED_CARD	0x80100065	The reader cannot communicate with the card, due to ATR string configuration conflicts
SCARD_W_UNRESPONSIVE_CARD	0x80100066	The smart card is not responding to a reset
SCARD_W_UNPOWERED_CARD	0x80100067L	Power has been removed from the smart card, so that further communication is not possible
SCARD_W_RESET_CARD	0x80100068	The smart card has been reset, so any shared state information is invalid
SCARD_W_REMOVED_CARD	0x80100069	The smart card has been removed, so further communication is not possible
SCARD_W_SECURITY_VIOLATION	0x8010006A	Access was denied because of a security violation
SCARD_W_WRONG_CHV	0x8010006B	The card cannot be accessed because the wrong PIN was presented
SCARD_W_CHV_BLOCKED	0x8010006C	The card cannot be accessed because the maximum number of PIN entry attempts has been reached
SCARD_W_EOF	0x8010006D	The end of the smart card file has been reached
SCARD_W_CANCELLED_BY_USER	0x8010006E	The action was cancelled by the user

SCARD_W_CARD_NOT_AUTHENTICATED	0x8010006F	No PIN was presented to the smart card

Table 9 – SCard return values

Linux WinSCard API

The Linux WinSCard API is similar to the Microsoft Windows API with the following exception:

SCardStatus()

SCardStatus() returns a bit field on pcsc-lite but a enumeration on Windows.

This difference may be resolved in a future version of pcsc-lite. The bit-fields would then only contain one bit set.

You can have a portable code using:

```
if (dwState & SCARD_PRESENT)
{
  // card is present
}
```

SCARD_E_UNSUPPORTED_FEATURE

Windows may return ERROR_NOT_SUPPORTED instead of SCARD_E_UNSUPPORTED_FEATURE

This difference will not be corrected. pcsc-lite only uses SCARD_E_* error codes.

SCARD_E_UNSUPPORTED_FEATURE

For historical reasons the value of SCARD_E_UNSUPPORTED_FEATURE is 0x8010001F in pcsc-lite but 0x80100022 in Windows WinSCard. You should not have any problem if you always use the symbolic name.

The value 0x8010001F is also used by SCARD_E_UNEXPECTED on pcsc-lite but SCARD_E_UNEXPECTED is never returned by pcsc-lite. So 0x8010001F does always means SCARD_E_UNSUPPORTED_FEATURE.

Applications like rdekstop that allow a Windows application to talk to pcsc-lite should take care of this difference and convert the value between the two worlds.

SCardConnect()

If SCARD_SHARE_DIRECT is used the reader is accessed in shared mode (like with SCARD_SHARE_SHARED) and not in exclusive mode (like with SCARD_SHARE_EXCLUSIVE) as on Windows.

SCardEstablishContext()

Each thread of an application shall use its own SCARDCONTEXT. SCardCancel() is the only exception to the rule. On Windows the same SCARDCONTEXT can be shared by different threads of same application.

The next functions are available:

SCardEstablishContext	Creates an Application Context for a client.
SCardReleaseContext	Destroys a communication context to the PC/SC Resource Manager.
SCardIsValidContext	Check if a SCARDCONTEXT is valid.
SCardConnect	Establishes a connection to the reader specified in * szReader.
SCardReconnect	Reestablishes a connection to a reader that was previously

	connected to using SCardConnect().
SCardDisconnect	Terminates a connection made through SCardConnect().
SCardBeginTransaction	Establishes a temporary exclusive access mode for doing a serie of commands in a transaction.
SCardEndTransaction	Ends a previously begun transaction.
SCardStatus	Returns the current status of the reader connected to by hCard.
SCardGetStatusChange	
SCardControl	Sends a command directly to the IFD Handler (reader driver) to be processed by the reader.
SCardTransmit	Sends an APDU to the smart card contained in the reader connected to by SCardConnect().
SCardListReaderGroups	Returns a list of currently available reader groups on the system.
SCardListReaders	This function returns a list of currently available readers on the system.
SCardFreeMemory	Releases memory that has been returned from the resource manager using the SCARD_AUTOALLOCATE length designator.
SCardCancel	This function cancels all pending blocking requests on the SCardGetStatusChange() function.
SCardGetAttrib	Get an attribute from the IFD Handler (reader driver).
SCardSetAttrib	Set an attribute of the IFD Handler.

Table 10 – Linux WinSCard Functions

For more information, please refer to the pcsc-lite website: http://pcsclite.alioth.debian.org.

Prox–DU and Prox–SU reader name

The Prox–DU and the Prox–SU devices will be recognized using their PC/SC name.

The string name depends on the operating system.

Windows operating systems

The name will comply with the following string format:

- “Gemalto Prox-DU Contactless_xxxxxxxx N1” for the Prox–DU contactless interface
- “Gemalto Prox-DU Contact_xxxxxxxx N2” for the Prox–DU contact interface
- “Gemalto Prox-SU Contactless_yyyyyyyy N3” for the Prox–SU contactless interface
- “Gemalto Prox-SU Contact_yyyyyyyy N4” for the Prox–SU contact interface

N1, N2, N3, N4 are numbers delivered by the computer. xxxxxxxx or yyyyyyyy is the 8-byte reader/writer’s serial number printed on the label located on the rear cabinet.

The next figure displays the name for one Prox–DU connected to the computer:

```
Gemalto Prox-DU Contactless_xxxxxxxx 0
Gemalto Prox-DU Contact_xxxxxxxx 1
```

Figure 2 – Prox–DU PC/SC name (Windows)

The next figure displays the name for one Prox–SU connected to the computer:

```
Gemalto Prox-SU Contactless_xxxxxxxx 0
Gemalto Prox-SU Contact_xxxxxxxx 1
```

Figure 3 – Prox–SU PC/SC name (Windows)

The next figure displays the name for one Prox–DU and one Prox–SU both connected to the computer:

```
Gemalto Prox-DU Contactless_xxxxxxxx 0
Gemalto Prox-DU Contact_xxxxxxxx 1
Gemalto Prox-SU Contactless_yyyyyyyy 2
Gemalto Prox-SU Contact_yyyyyyyy 3
```

Figure 4 – Prox–DU and Prox–SU PC/SC names (Windows)

The next figure displays the name for two Prox–DU devices both connected to the computer:

```
Gemalto Prox-DU Contactless_xxxxxxxx 0
Gemalto Prox-DU Contact_xxxxxxxx 1
Gemalto Prox-DU Contactless_yyyyyyyy 2
Gemalto Prox-DU Contact_yyyyyyyy 3
```

Figure 5 – Two Prox–DU PC/SC names (Windows)

The two first names belong to the first Prox–DU device. The two next names belong to the second Prox–DU device.

Note: The application should use the name of the device for connecting the appropriate smart card interface.

Linux and Mac OS X operating systems

The name will comply with the following string format:

- “Gemalto Prox-DU (xxxxxxxx) N1 00” for the Prox–DU contactless interface
- “Gemalto Prox-DU (xxxxxxxx) N1 01” for the Prox–DU contact interface
- “Gemalto Prox-SU (yyyyyyyy) N2 00” for the Prox–SU contactless interface
- “Gemalto Prox-SU (yyyyyyyy) N2 01” for the Prox–SU contact interface

N1, N2 are numbers delivered by the computer. xxxxxxxx or yyyyyyyy is the 8-byte reader/writer’s serial number printed on the label located on the rear cabinet.

The next figure displays the name for one Prox–DU connected to the computer:

```
Gemalto Prox-DU (xxxxxxxx) 00 00
Gemalto Prox-DU (xxxxxxxx) 00 01
```

Figure 6 – Prox–DU PC/SC name (Linux)

The next figure displays the name for one Prox–SU connected to the computer:

```
Gemalto Prox-SU (xxxxxxxx) 00 00
Gemalto Prox-SU (xxxxxxxx) 00 01
```

Figure 7 – Prox–SU PC/SC name (Linux)

The next figure displays the name for one Prox–DU and one Prox–SU both connected to the computer:

```
Gemalto Prox-DU (xxxxxxxx) 00 00
Gemalto Prox-DU (xxxxxxxx) 00 01
Gemalto Prox-SU (yyyyyyyy) 01 00
Gemalto Prox-SU (yyyyyyyy) 01 01
```

Figure 8 – Prox–DU and Prox–SU PC/SC names (Linux)

The next figure displays the name for two Prox–DU devices both connected to the computer:

```
Gemalto Prox-DU (xxxxxxxx) 00 00
Gemalto Prox-DU (xxxxxxxx) 00 01
Gemalto Prox-DU (yyyyyyyy) 01 00
Gemalto Prox-DU (yyyyyyyy) 01 01
```

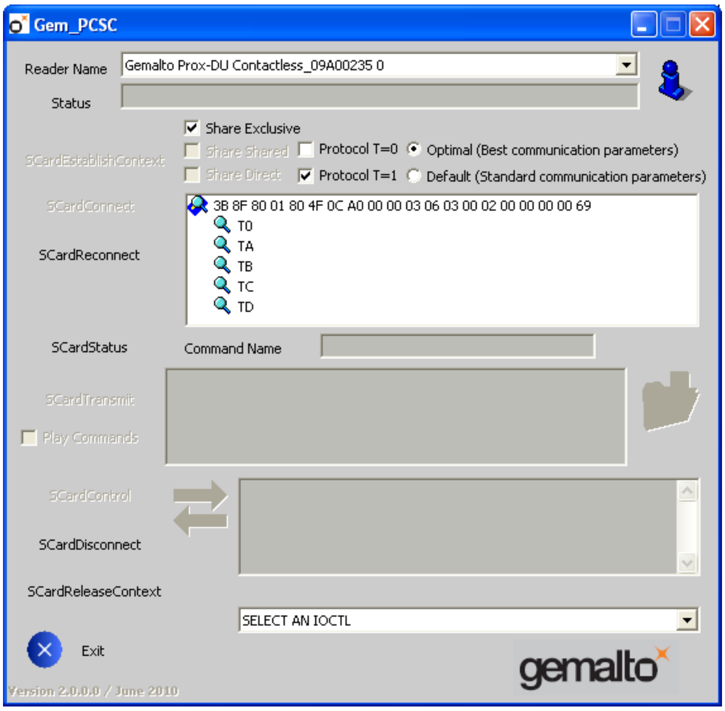
Figure 9 – Two Prox–DU PC/SC names (Linux)

The two first names belong to the first Prox–DU device. The two next names belong to the second Prox–DU device.

Note: The application should use the name of the device for connecting the appropriate smart card interface.

Gem_PC/SC software tool

The Gemalto Gem_PCSC tool may help to become familiar with the PC/SC environment:



Reader Name: Gemalto Prox-DU Contactless_09A00235 0

Status:

☑ Share Exclusive

□ Share Shared □ Protocol T=0 ● Optimal (Best communication parameters)

□ Share Direct ☑ Protocol T=1 ○ Default (Standard communication parameters)

3B 8F 80 01 80 4F 0C A0 00 00 03 06 03 00 02 00 00 00 00 69
- T0
- TA
- TB
- TC
- TD

Command Name:

□ Play Commands

SELECT AN IOCTL

Figure 10 – Gem_PCSC window

The Gemalto Gem_PCSC tool can be used to easily evaluate the Prox–DU and the Prox–SU in the PC/SC environment with a Windows based operating system.

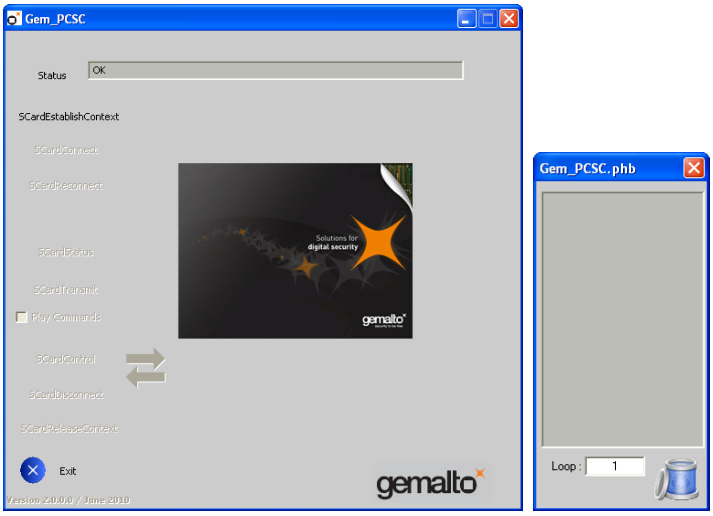
In the example below the read of the memory block number 0 of a Mifare® 1K smart card will be performed:

1. The contactless reader is selected
2. The Mifare® 1K is connected
3. Then the block number 0 is authenticated and read
4. The smart card is disconnected
5. The reader is deselected

The Gem_PCSC tool is available for download in the Gemalto support website: http://support.gemalto.com.

Playing with PC/SC

After the Gem_PCSC installation run the Gem_PCSC tool. The next window will be displayed:

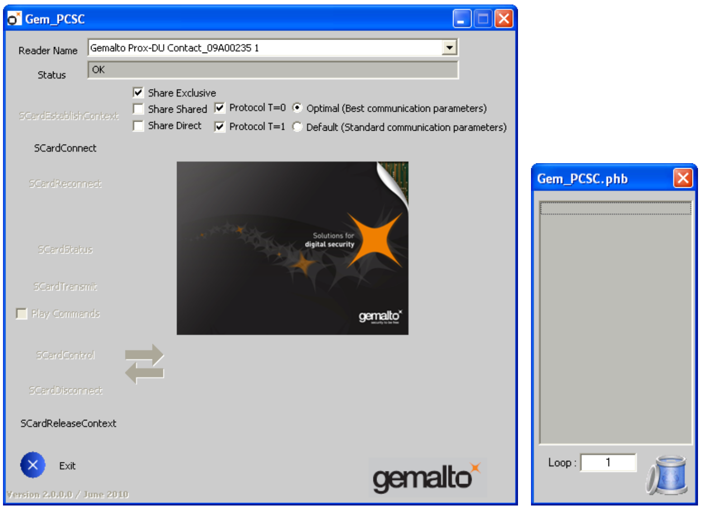


Status: OK

Figure 11 – Gem_PCSC window at start up

The main window displays the available PC/SC commands. The Gem_PCSC.phb small window is used to display the current commands stored by the user.

1- Click the “**SCardEstablishContext**” button to have the list of all the available PC/SC devices.
The next window will be displayed:



Reader Name: Gemalto Prox-DU Contact_09A00235 1

Status: OK

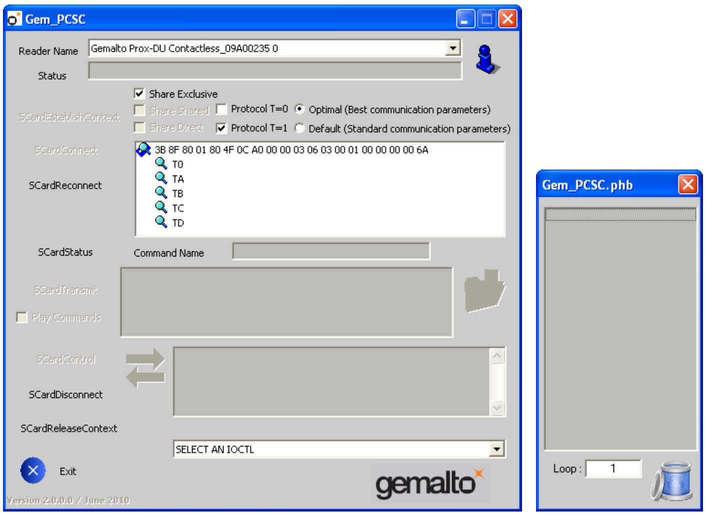
☑ Share Exclusive
□ Share Shared
□ Share Direct
☑ Protocol T=0
☑ Protocol T=1
● Optimal (Best communication parameters)
○ Default (Standard communication parameters)

Figure 12 – Gem_PCSC window after “SCardEstablishContext”

2- Perform the following operations to start a communication with the smart card:

- Put a Mifare$^{®}$ 1K contactless smart card in front of the landing zone of the reader
- In the “**Reader Name**” box select the contactless interface of the Prox-DU named “**Gemalto Prox-DU Contactless_xxxxxxxx 0**” (xxxxxxxx is the reader serial number printed on the label located in the rear of the casing)
- Click the “**SCardConnect**” button

The next window will be displayed:



Gem_PCSC

Reader Name: Gemalto Prox-DU Contactless_09A00235 0

Status:

☑ Share Exclusive

□ Share Shared □ Protocol T=0 ● Optimal (Best communication parameters)

□ Share Direct ☑ Protocol T=1 ○ Default (Standard communication parameters)

3B 8F 80 01 80 4F 0C A0 00 00 03 06 03 00 01 00 00 00 00 6A
- T0
- TA
- TB
- TC
- TD

Command Name:

SELECT AN IOCTL

Version 2.0.0.0 / June 2010

Gem_PCSC.phb

Loop : 1

Figure 13 – Gem_PCSC window after “SCardConnect”

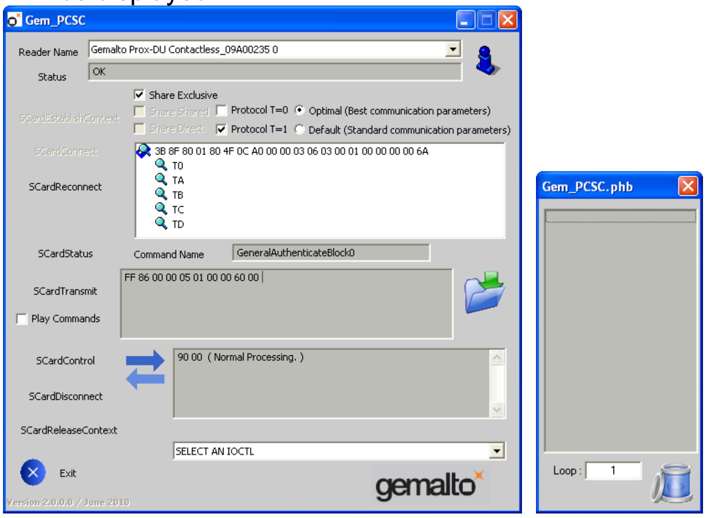
The ATR of the smart card will be displayed: **3B 8F 80 01 80 4F 0C A0 00 00 03 06 03 00 01 00 00 00 00 6A** corresponding to a Mifare® 1K smart card.

3- Define the Mifare® commands to be used:

3-a First authenticate the block number 0:

- In the “**Command Name**” box type the following smart card command:
  - “**GeneralAuthenticateBlock0**”
- In the field below type the corresponding APDU command (according to PC/SC V2 specification)
  - “**FF 86 00 00 05 01 00 00 60 00**“
  - Meaning “Authenticate block number 0 using the Key A number 0”
- Click the “**SCardTransmit**” button

The next window will be displayed:



Gem_PCSC

Reader Name: Gemalto Prox-DU Contactless_09A00235 0

Status: OK

☑ Share Exclusive

□ Share Shared □ Protocol T=0 ● Optimal (Best communication parameters)

□ Share Direct ☑ Protocol T=1 ○ Default (Standard communication parameters)

3B 8F 80 01 80 4F 0C A0 00 00 03 06 03 00 01 00 00 00 00 6A
- T0
- TA
- TB
- TC
- TD

Command Name: GeneralAuthenticateBlock0

```
FF 86 00 00 05 01 00 00 60 00
```

```
90 00  ( Normal Processing. )
```

SELECT AN IOCTL

Version 2.0.0.0 / June 2010

Gem_PCSC.phb

Loop : 1

Figure 14 – Gem_PCSC window after “SCardTransmit” the Authentication command

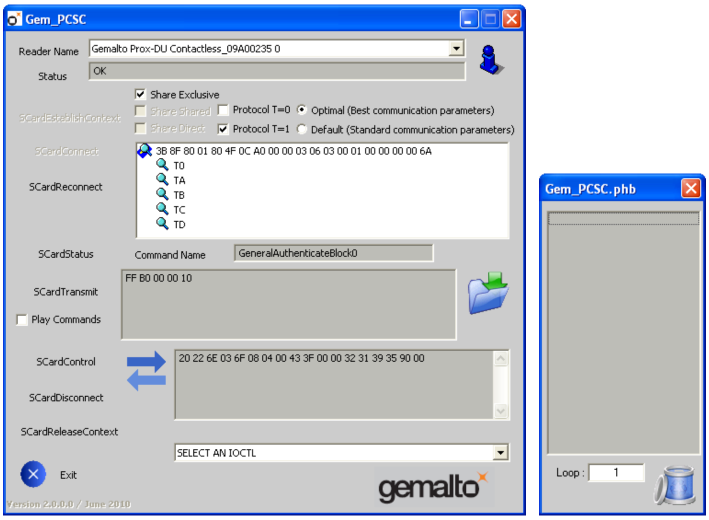
The smart card response is displayed:

- **90 00** if the command was correctly processed
- Else an error status code

3-b Then read the block number 0:

- In the “**Command Name**” box type the following smart card command:
    - “**ReadBlock0**”
- In the field below type the corresponding APDU command (according to PC/SC V2 specification)
    - “**FF B0 00 00 10**“
    - Meaning “Read block number 0 – 16 bytes”
- Click the “**SCardTransmit**” button

The next window will be displayed:



Gem_PCSC

Reader Name: Gemalto Prox-DU Contactless_09A00235 0

Status: OK

☑ Share Exclusive
□ Share Shared □ Protocol T=0 ● Optimal (Best communication parameters)
□ Share Direct ☑ Protocol T=1 ○ Default (Standard communication parameters)

```
3B 8F 80 01 80 4F 0C A0 00 00 03 06 03 00 01 00 00 00 00 6A
    T0
    TA
    TB
    TC
    TD
```

Command Name: GeneralAuthenticateBlock0

```
FF B0 00 00 10
```

□ Play Commands

```
20 22 6E 03 6F 08 04 00 43 3F 00 00 32 31 39 35 90 00
```

SELECT AN IOCTL

Version 2.0.0.0 / June 2010

Gem_PCSC.phb

Loop : 1

Figure 15 – Gem_PCSC window after “SCardTransmit” the Read command

The smart card response is displayed:

- **20 22 6E 03 6F 08 04 00 43 3F 00 00 32 31 39 35** are the data bytes
- Followed by the status code **90 00** if the command was correctly processed
- Else an error status code will only be displayed

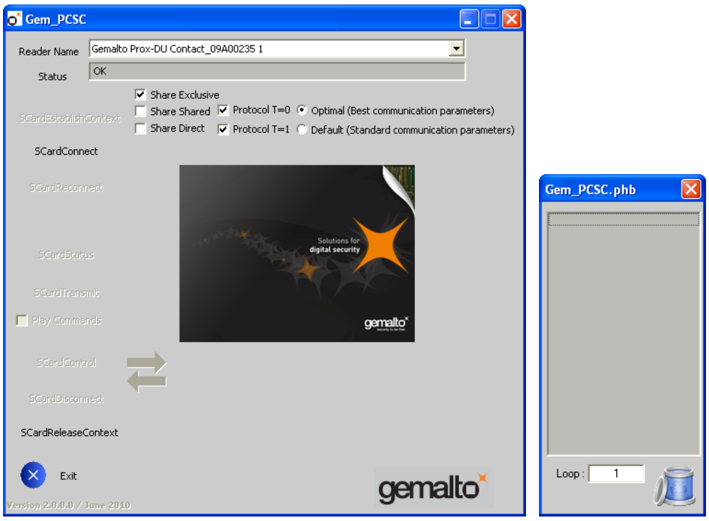
20 22 6E 03 is the serial number of the smart card (03 is the MSB and 20 the LSB).

Note: the data bytes displayed are depending on the smart card content. The Key A into the smart card should be the same as the Key A into the reader for a proper authentication.

4- Perform the following operations to close the communication with the smart card:

- Remove the contactless smart card from the landing zone
- Click the “**SCardDisconnect**” button

The next window will be displayed:



Reader Name: Gemalto Prox-DU Contact_09A00235 1

Status: OK

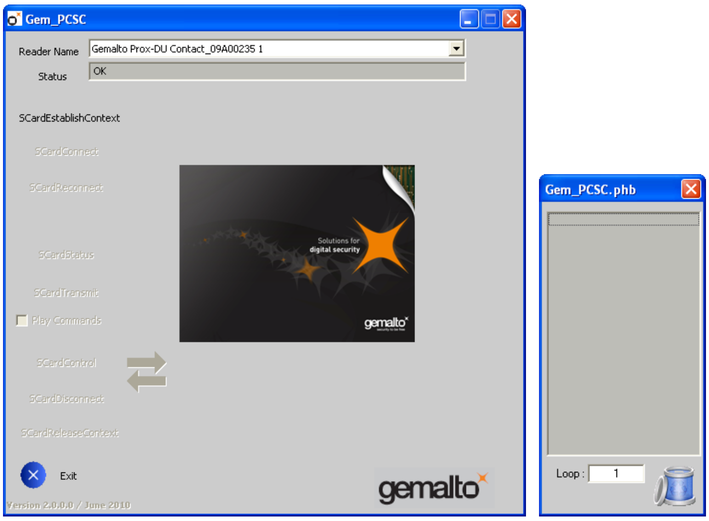
☑ Share Exclusive
□ Share Shared ☑ Protocol T=0 ● Optimal (Best communication parameters)
□ Share Direct ☑ Protocol T=1 ○ Default (Standard communication parameters)

□ Play Commands

Loop : 1

Figure 16 – Gem_PCSC window after "SCardDisconnect"

5- Click the "**SCardReleaseContext**" button to close the connection with all the available PC/SC devices.
The next window will be displayed:



Reader Name: Gemalto Prox-DU Contact_09A00235 1

Status: OK

□ Play Commands

Loop : 1

Figure 17 – Gem_PCSC window after "SCardReleaseContext"

You can now close the Gem_PCSC tool.

Known issues and limitations with all the operating systems

The Prox–DU and the Prox–SU devices have the following limitations:

- The contactless interface will only support the T=1 protocol.

Consequently any connection requiring the T=0 protocol will not be accepted by the contactless interface.

- Multi-activation of contactless smart cards is not supported.

Consequently the first smart card detected in front of the reader/writer will be activated. The remaining smart cards will be ignored.

- The communication with the contactless interface and the contact interface shall be exclusive.

Consequently the application shall not use the two interfaces simultaneously. Else communication errors can occur.

Known issues and limitations with Windows operating systems

The Prox–DU and the Prox–SU devices have the following limitations when operating with Windows operating systems:

Operating System	Known issue	Workaround
Windows XP Vista Seven	Prox–DU & Prox–SU: Simultaneous use of the contact and the contactless interface may freeze the device	Prox–DU: Do not use the the dual interface card protection in off mode Prox–SU: Do not use the two interfaces simulatneously When the device is frozen, unplug and replug the USB cable to recover a proper operation
Windows XP	Prox–SU: When the computer is restarting it may happen the internal SIM/SAM card is not detected by the device	Unplug and replug the USB cable to recover a proper operation

Windows XP Vista Seven	Prox–SU: Enabling/Disabling the USB smart card reader , Installing/Uninstalling the USB smart card reader, from the Device Manager window may cause an issue	These operations should be done from the composite device and not the USB smart card reader
Windows Vista	Prox–SU: When the computer is restarting or rebooting, the contactless card notification event may not be detected	Unplug and replug the USB cable to recover a proper operation
Windows Seven	Prox–DU: Some “exotic“ contacless cards may not be recognized by the device Some “exotic” dual cards may not be recognized by the device	Unplug and replug the USB cable to recover a proper operation
Windows XP	Prox–DU & Prox–SU: High speed contact smart cards (supporting TA1=97h ISO7816 parameter) are not recognized	The Microsoft CCID driver does not support high speed contact smart cards (with TA1=97h parameter). Please contact the Gemalto support website http://support.gemalto.com/.

Table 11 – Known issues and limitations (Windows OS)

Known issues and limitations with Linux operating systems

The Prox–DU and the Prox–SU devices have the following limitations when operating with Linux operating systems:

Operating System	Known issue	Workaround
Linux	Prox-DU & Prox-SU: The reader can be frozen when both interfaces (contact & contactless) are used simultaneously.	Use libccid driver version 1.4.0 minimum. The libccid source code is available on the following web site: http://pcsclite.alioth.debian.org/ccid.html

Table 12 – Known issues and limitations (Linux OS)

Known issues and limitations with Mac operating systems

Please note the following issues and limitations related to **PCSC-Lite** included into the MAC OS X operating systems:

Operating System	Known issue For PCSC-Lite	Workaround
Mac OS X Tiger (10.4)	SCardControl() issue: PC/SC SCardControl() command requires two arguments at least.	To compile and use a C source code using SCardControl() command, please use the reader.h file delivered with the "Secure Pin Entry sample code" available in the website http://support.gemalto.com/?id=63
Mac OS X Tiger (10.4)	Dual protocol card issue: For dual protocol cards (T=0 and T=1) the connection will be used in T=0 (default protocol). pcsc-lite will not switch the card to T=1 as it is required for contactless cards.	If the application needs to use the contactless interface the connection should be made using the T=1 protocol only.

Mac OS X Leopard (10.5) Snow Leopard (10.6)	"Ghost" reader: After the connection 3 PC/SC readers will be displayed instead of 2: The second reader (contact) will be displayed twice (A "ghost" contact reader will be displayed)	Both contact readers are the same.
Mac OS X Leopard (10.5) Snow Leopard (10.6)	"Ghost" reader: After the device is disconnected one "ghost" PC/SC reader will still be displayed.	This PC/SC reader is no more available.
Mac OS X Leopard (10.5) Snow Leopard (10.6)	"Ghost" reader: After a Suspend / Wakeup cycle only the PC/SC contactless reader will be displayed correctly. The PC/SC contact reader is displayed but is no more useable. The "ghost" PC/SC contact reader is still present.	Unplug and Replug the reader.
Mac OS X Tiger (10.4) Leopard (10.5) Snow Leopard (10.6)	SCardStatus() issue: The pdwProtocol parameter is not correct with the contact reader. Returned value is 0 (SCARD_PROTOCOL_UNDEFINED)	No workaround

Table 13 – PCSC-Lite known issues and limitations (Mac OS X)

Interfacing with Contactless Cards

As defined in the PC/SC V2.0 specifications, the Prox–DU and the Prox–SU devices handle all the ISO7816-4 Inter Industry commands to interface ISO14443 contactless smart cards.

The Prox–DU and the Prox–SU devices support both type ISO14443-A and ISO14443-B cards.

In addition the Prox–DU and the Prox–SU devices will poll the field for the following smart card events:

- Insertion
- Removal

Detecting an insertion

The contactless reader/writer periodically sends out commands to poll the RF field. If a smart card comes within the range of the RF field, the contactless reader/writer detects the card and activates it.

When the card is activated, its properties are recorded and an insertion event is generated.

The ISO14443 contactless smart card will be activated using the reader parameters stored into the device’s configuration EEPROM.

ISO14443-A and ISO14443-B cards are polled with a default periodic rate of 100 ms.

Note: Multi-activation of contactless smart cards is not supported by the Prox–DU and the Prox–SU devices. The first smart card detected in front of the reader/writer will be activated.

When a smart card insertion is detected, a CCID insertion notification message will be generated and the blue LED of the contactless reader/writer will be set to an enlightened steady state.

Detecting a removal

A smart card being removed from the field is detected by the contactless reader/writer.

The contactless reader/writer polls for an ISO14443-3 (MIFARE®) smart card by periodically accessing the smart card during periods when there is no communication between the reader/writer and the card.

The contactless reader/writer polls for an ISO14443-4 (T=CL) smart card by periodically sending negative acknowledge frames to the smart card expecting, either a positive acknowledge or the last I-block to be repeated (according to the ISO14443-4 standards).

When a smart card removal is detected, a CCID removal notification message will be generated and the blue LED of the contactless reader/writer will blink slowly.

ATR for contactless smart cards

The Answer To Request (ATR) returned by an ISO14443-A or ISO14443-B smart card is compliant with the PC/SC V2.0 Part 3 Revision 2.01.09 specifications.

The ATR is as follows:

Byte Number	Value	Designation	Description
0	3Bh	Initial header	
1	8Nh	T0	• Higher nibble 8 means no TA1, TB1, TC1 only TD1 is following. • Lower nibble N is the number of historical bytes (HistByte 0 to HistByte N-1).
2	80h	TD1	• Higher nibble 8 means no TA2, TB2, TC2 only TD2 is following. • Lower nibble 0 means T = 0.
3	01h	TD2	• Higher nibble 0 means no TA3, TB3, TC3, TD3 following. • Lower nibble 1 means T = 1.
4 to 3+N	XX XX XX	T1 … .. Tk	Historical bytes: • ISO14443A: The historical bytes from ATS response. Refer to the ISO14443-4 specification • ISO14443B: Byte 1-4: Application Data from ATQB Byte 5-7: Protocol Info Byte from ATQB Byte 8: Higher nibble = MBLI from ATTRIB command. Lower nibble (RFU) = 0 Refer to the ISO14443-3 specification
4 + N	UU	TCK	Exclusive-OR of bytes T0 to Tk

Table 14 – ATR for contactless Smart cards

The contactless smart card exposes its ATS or information bytes not directly, but via a specific ATR mapping. For those cards that provide such information, optionally with Historical Bytes (or Application Information respectively), the mapping in the table above applies.

- The ATR returned by a DESFire smart card is:

  3Bh 8Fh 80h 01h 80h 80h 65h B0h 07h 02h 02h 89h 83h 00h 90h 00h 00h 00h 00h 46h

  With:

  n = Fh (15 historical bytes)

  Historical bytes from the ATS response = 80h 80h 65h B0h 07h 02h 02h 89h 83h

  00h 90h 00h 00h 00h 00h

UU = 46h (TCK)

- The ATR returned by a GemCombi Xpresso Lite R2 STD smart card will be:

  3Bh 8Bh 80h 01h 80h 31h 80h 65h B0h 07h 02h 02h 89h 83h 00h E3h

  With:
  n = Bh (11 historical bytes)
  Historical bytes from the ATS response = 80h 31h 80h 65h B0h 07h 02h 02h 89h 83h 00h
  UU = E3h (TCK)

- The ATR returned by a GemCombi CDLite smart card will be:

  3Bh 80h 80h 01h 01h

  With:
  n = 0h (no historical byte)
  UU = 01h (TCK)

Interfacing with MIFARE® DESFire Cards

The MIFARE® DESFire smart card is based on open global standards for both air interface and cryptographic methods. It is compliant to all 4 levels of ISO14443-A and uses optional ISO 7816-4 commands.

The native MIFARE® DESFire commands are non ISO7816-4 commands.

A proprietary APDU command is implemented into the Prox–DU and Prox–SU reader/writer in order to send and receive these native commands.

Warning: Note that the EEPROM configuration parameter byte "T=CL card presence check behavior" should be set to value "01h" to avoid the reader to send a dummy APDU command during the selection process, because when the MIFARE® DESFire smart card receives an APDU command after the selection process the native commands are no more available. Refer to the "*Reference Manual*" document for more information.

The command is formatted as follows:

CLA	INS	P1	P2	Lc	Data In	
FFh	DEh	00h	00h	N	DESFire Command	
1 byte	1 byte	1 byte	1 byte	1 byte	N bytes	

The response is formatted as follows:

Data Out	[SW1	SW2]
M bytes	1 byte	1 byte

Where:

N	Length of the Data In field	Length of the native command
Data In	DESFire native command	Refer to the DESFire datasheet
Data Out	DESFire native response	Refer to the DESFire datasheet
[SW1-SW2]	Command execution status	Optional field: SW1 SW2 = 90h 00h is added by the reader when the DESFire smart card response is only one byte Status.
	Command executed successfully	90h 00h
	Others	
	67h 00h	Wrong length
	6Ah 81h	Function not supported
	6Bh 00h	Wrong P1 or P2

As an example, to get the version of the DESFire smart card, the following native command should be send: 60h

The proprietary command to consider is the following:

FFh DEh 00h 00h 01h 60h

The response will be:

AFh 04h 01h 01h 00h 02h 18h 05h (example)

Refer to the DESFire datasheet for more information about the response.

Requesting contactless smart card information

This proprietary APDU command is used to retrieve the contactless smart card parameters returned by the smart card during the contactless selection process.

The command is formatted as follows:

CLA	INS	P1	P2	Lc	Data In	Le
FFh	FCh	Param	00h	-	-	00h
1 byte	1 byte	1 byte	1 byte	-	-	1 byte

The response is formatted as follows:

Data Out	SW1	SW2
M bytes	1 byte	1 byte

Where:

Param	ISO14443-A or ISO14443-B requested information	00h : ATQA + SN + SAK 01h : complete ATS 02h : ATQB 03h :complete ATTRIB response
Data Out	Requested information	Refer to the ISO14443 standard
SW1-SW2	Command execution status	added by the reader
	Command executed successfully	90h 00h
	Others	
	67h 00h	Wrong length
	6Bh 00h	Wrong P1 or P2
	6Ch xxh	Wrong length (XX is required)
	62h 82h	End of data reach before Le bytes

Note: When the requested information does not correspond to the current smart card type (ISO14443-A or ISO14443-B) an error is reported.
For ISO14443-A3 smart cards, the ATS field is empty.

As an example, to get the ATQA + SN + SAK of the DESFire smart card, the proprietary command to consider is the following:

FFh FCh 01h 00h 00h

The response will be:

44h 03h 04h 26h 47h 09h 48h E8h 10h 20h 90h 00h (example)

ATQA = 44h 03h 04h

SN = 26h 47h 09h 48h E8h 10h (7 bytes)

SAK = 20h

Refer to the DESFire datasheet for more information about the response.

Interfacing with MIFARE® Classic Cards

As defined in PC/SC V2.0 Part 3 Revision 2.01.09 specifications, the Prox–DU and the Prox–SU devices perform the appropriate mapping for memory smart card commands that consist of Inter Industry commands (and the exposed data structures) to memory card commands (and the associated data structures defined for the MIFARE® contactless memory smart cards).

The Prox–DU and the Prox–SU devices will handle the following ISO7816-4 Inter Industry commands to interface with MIFARE® 1K, MIFARE® 4K, MIFARE® Ultralight and MIFARE® Mini memory smart cards:

- **Get Data**: retrieves the UID or the historical bytes of the ATS of the inserted smart card.
- **Load Keys**: Load MIFARE® secret into the contactless reader/writer.
- **General Authenticate**: Perform an authentication between the contactless reader/writer and the MIFARE® memory smart cards.
- **Read Binary**: Read data from the MIFARE® memory smart cards.
- **Update Binary**: Write data to the MIFARE® memory smart cards.

The MIFARE® 1K is a 8-Kbit (1 Kbyte) MIFARE® memory contactless smart card arranged as 64 memory blocks as shown in the appendix “MIFARE® cards mapping”.

The MIFARE® 4K is a 32-Kbit (4 Kbytes) MIFARE® memory contactless smart card arranged as 256 memory blocks as shown in the appendix “MIFARE® cards mapping”.

The MIFARE® Ultralight is a 512-bit (64 bytes) MIFARE® memory contactless smart card arranged as 16 memory pages as shown in the appendix “MIFARE® cards mapping”.

The MIFARE® Mini is a 2.5-Kbit (320 bytes) MIFARE® memory contactless smart card arranged as 20 memory blocks as shown in the appendix “MIFARE® cards mapping”.

Important note regarding contactless smart cards including both MIFARE® and ISO14443-A4 (T=CL) modes:

When the smart card is connected, the ISO14443-A4 (T=CL) mode will be selected. The corresponding ATR will be returned.

When a MIFARE® command is send to the smart card an automatic switch to the MIFARE® mode is done and the command will be processed accordingly.

When an ISO14443-A4 (T=CL) command is send to the smart card an automatic switch to the ISO14443-A4 (T=CL) mode is done and the command will be processed accordingly.

When the smart card is in the MIFARE® mode, the only way to retrieve the MIFARE® type (1K-4K-UL-Mini) is to reconnect the smart card. The appropriate MIFARE® ATR will then be returned.

ATR for MIFARE® cards

The Answer To Request (ATR) returned by a MIFARE® card is compliant with PC/SC V2.0 Part 3 Revision 2.01.09 specifications.

The ATR will be as follows:

Byte Number	Value	Designation	Description
0	3Bh	Initial header	
1	8Nh	T0	• Higher nibble 8 means no TA1, TB1, TC1 only TD1 is following. • Lower nibble N is the number of historical bytes (HistByte 0 to HistByte N-1).
2	80h	TD1	• Higher nibble 8 means no TA2, TB2, TC2 only TD2 is following. • Lower nibble 0 means T = 0.
3	01h	TD2	• Higher nibble 0 means no TA3, TB3, TC3, TD3 following. • Lower nibble 1 means T = 1.
4 to 2+n	80h	T1 … .. Tk	Category indicator byte, 80h means a status indicator may be present in an optional COMPACT-TLV data object.
	4Fh		Application identifier presence indicator
	LL		Length
	A0h 00h 00h 03h 06h		5 bytes for registered application provider identifier (RID)
	SS		1 byte for Standard
	NN NN		2 bytes for Card Name
	00h 00h 00h 00h		RFU: Shall be set to zero. Assigned by PC/SC for future extensions.
3 + n	UU	TCK	Exclusive-OR of bytes T0 to Tk

Table 15 – ATR for MIFARE® cards

The ATR of a contactless storage card is structured in the manner described in the table above. In order to allow the application to identify a storage card type properly, its Standard and Card Name bytes must be interpreted according to the following tables:

b7	b6	b5	b4	b3	b2	b1	b0	Description
0	0	0	0	0	0	0	0	No information given
0	0	0	0	0	0	1	1	ISO14443-A, part 3
0	0	0	0	0	1	0	0	ISO14443-A, part 4
…								RFU
1	1	1	1	1	1	1	1	RFU

Table 16 – SS Byte for Standard

Card Name	Two bytes identifier
MIFARE® Standard 1K	00h 01h
MIFARE® Standard 4K	00h 02h
MIFARE® Ultralight	00h 03h
MIFARE® Mini	00h 26h

Table 17 – NN Bytes for Card Name

- The ATR returned by a MIFARE® Standard 1K will be:

  3Bh 8Fh 80h 01h 80h 4Fh 0Ch A0h 00h 00h 03h 06h 03h 00h 01h 00h 00h 00h 00h 6Ah

  With:

  LL = 0Ch (12 bytes)

  SS = 03h (ISO14443-A, part 3)

  NN NN = 00h 01h (MIFARE® Standard 1K)

  UU = 6Ah (TCK)

- The ATR returned by a MIFARE® Standard 4K will be:

  3Bh 8Fh 80h 01h 80h 4Fh 0Ch A0h 00h 00h 03h 06h 03h 00h 02h 00h 00h 00h 00h 69h

  With:

  LL = 0Ch (12 bytes)

  SS = 03h (ISO14443-A, part 3)

  NN NN = 00h 02h (MIFARE® Standard 4K)

  UU = 69h (TCK)

- The ATR returned by a MIFARE® Ultralight will be:

3Bh 8Fh 80h 01h 80h 4Fh 0Ch A0h 00h 00h 03h 06h 03h 00h 03h 00h 00h 00h 00h 68h

With:

LL = 0Ch (12 bytes)

SS = 03h (ISO14443-A, part 3)

NN NN = 00h 03h (MIFARE® Ultralight)

UU = 68h (TCK)

- The ATR returned by a MIFARE® Mini will be:

  3Bh 8Fh 80h 01h 80h 4Fh 0Ch A0h 00h 00h 03h 06h 03h 00h 26h 00h 00h 00h 00h 4Dh

  With:

  LL = 0Ch (12 bytes)

  SS = 03h (ISO14443-A, part 3)

  NN NN = 00h 26h (MIFARE® Mini)

  UU = 4Dh (TCK)

Get Data command

This command is used to retrieve information about the inserted smart card. This command can be used for all kinds of contactless cards.

The command is formatted as follows:

CLA	INS	P1	P2	Lc	Data	Le
FFh	CAh	INF	00h	-	-	NN
1 byte	1 byte	1 byte	1 byte	-	-	1 byte

The response is formatted as follows:

Data	SW1	SW2
NN bytes	1 byte	1 byte

Where:

INF	Info type	INF = 00h means: Card serial number (UID or PUPI) is returned INF = 01h means: All historical bytes from the ATS of a ISO14443 A card without CRC are returned
NN	Expected length of the data	

		NN = 00h means: Return full length of the UID (e.g. for ISO14443-A single 4 bytes, double 7 bytes, triple 10 bytes, for ISO14443-B 4 bytes PUPI)
Data	Serial Number	
	For ISO14443-A smart cards: UID0-UID1-UID2-UID3 or UID0-UID1-UID2-UID3-UID4-UID5-UID6 or UID0-UID1-UID2-UID3-UID4-UID5-UID6-UID7-UID8-UID9 For ISO14443-B smart cards: PUPI3-PUPI2-PUPI1-PUPI0	The UID is exposed as a string of the expected length. If the expected length is greater than the actual length the rest of the string is filled with zero-value padding bytes. No cast must be done over the UID or parts of it. For example, casting four bytes of the UID to a 32-bit Integer is illegal. The order of the bytes within the string matches the order of bytes received from the card during the anti-collision process. Consequently, the first byte received will be at index zero. The bit order of the string bytes must be such that the LSB (MSB) matches with the LSB (MSB) of the card-defined UID. Refer to ISO14443-A standard.
	Historical bytes of the ATS	For a MIFARE® or ISO14443-B card that command is not supported.
SW1-SW2	Command execution status	
	Command executed successfully	90h 00h
	Others	Refer to the error codes table below

Load Keys command

This command is used to load the MIFARE® secret keys into the contactless reader/writer.

Up to 160 keys can be loaded to support all the keys pairs needed for the Mifare 4K cards (2 keys for each sector):

- 80 keys stored in the reader/writer's EEPROM
- 80 keys stored in the reader/writer's RAM

The command is formatted as follows:

CLA	INS	P1	P2	Lc	Data
FFh	82h	KS	KN	KL	Key
1 byte	1 byte	1 byte	1 byte	1 byte	6 bytes

If the Load MIFARE® key security bit is set to one in the configuration EEPROM, a Transport secret key should be added to the MIFARE® key:

The command is formatted as follows:

CLA	INS	P1	P2	Lc	Data
FFh	82h	KS	KN	KL	Key
1 byte	1 byte	1 byte	1 byte	1 byte	12 bytes

The response is formatted as follows:

SW1	SW2
1 byte	1 byte

Where:

KS	Key Structure	00h for key location storage in RAM 20h for key location storage in EEPROM
KN	Key Number	160 keys are available
	MIFARE® Key Number	0 to 159 (00h to 9Fh) The key number which will be used for the authentication The key number 0 to 79 (00h to 4Fh) are reserved for the non volatile key stored in EEPROM The key number 80 to 159 (50h to 9Fh) are reserved for the volatile key stored in RAM
KL	Key Length	KL = 06h means: 6 bytes long KL = 0Ch means: 12 bytes long if the Load MIFARE® key security bit is set on.
Key	MIFARE® Secret Key	The MIFARE® key value Should be followed by the Gemalto Transport key if the Load MIFARE® key security bit is set on. The byte order must be the same as the byte order in the card sector trailer (A0h first for the key A0h A1h A2h A3h A4h A5h)
	Gemalto default Key A	A0h A1h A2h A3h A4h A5h
	Gemalto default Key B	B0h B1h B2h B3h B4h B5h
	Gemalto Transport Key	T0 T1 T2 T3 T4 T5
	Other values	User Key

SW1-SW2	Command execution status	
	Command executed successfully	90h 00h
	Others	Refer to the error codes table below

Warning: If the Load MIFARE® key security bit is set to one, a 6 bytes transport key must be added to the Data field and the total key length must be equal to 12.

The Transport keys are secret and are available upon request.

To load the secret key A0h-A1h-A2h-A3h-A4h-A5h into the key number KN using location storage in RAM the following APDU command should be used:

CLA	INS	P1	P2	Lc	Data
FFh	82h	00h	KN	06h	A0h A1h A2h A3h A4h A5h

With KN = 80 to 159 (50h to 9Fh)

To load the secret key A0h-A1h-A2h-A3h-A4h-A5h into the key number KN using location storage in EEPROM the following APDU command should be used:

CLA	INS	P1	P2	Lc	Data
FFh	82h	20h	KN	06h	A0h A1h A2h A3h A4h A5h

With KN = 0 to 79 (00h to 4Fh)

Note: Loading key number 0 to 79 in RAM is forbidden. Loading key number 80 to 159 in EEPROM is forbidden. The error code SW1-SW2 69h 88h will be returned (Key number not valid)

Note: After delivery the non volatile keys stored in EEPROM (number 0 to 79) are initialized to a default value:

The keys number 00 to 39 are initialized with value A0h A1h A2h A3h A4h A5h

The keys number 40 to 79 are initialized with value B0h B1h B2h B3h B4h B5h

Note: Each time the Prox–DU and the Prox–SU is powered, the volatile keys stored in RAM (number 80 to 159) are initialized to a default value:

The keys number 80 to 119 are initialized with value A0h A1h A2h A3h A4h A5h

The keys number 120 to 159 are initialized with value B0h B1h B2h B3h B4h B5h

General Authenticate command

The General Authenticate command is used to perform an authentication between the contactless reader/writer and a MIFARE® memory block.

For MIFARE® 1K, MIFARE® 4K and MIFARE® Mini it is mandatory to perform the General Authenticate command before each read or write memory block operation. Otherwise, an authentication error will occur.

For MIFARE® Ultralight the General Authenticate operation is not required.

This command is formatted as follows:

CLA	INS	P1	P2	Lc	Data In				
FFh	86h	00h	00h	05h	VER	ABLM	ABLL	KT	KN
1 byte	1 byte	1 byte	1 byte	1 byte	1 byte	1 byte	1 byte	1 byte	1byte

The response is formatted as follows:

SW1	SW2
1 byte	1 byte

Where:

VER	Version		01h VER is used in the future to differentiate different version of this command.
ABLM	Address Block MSB		00h
ABLL	Address Block LSB		
		MIFARE® 1K	00h – 3Fh
		MIFARE® 4K	00h – FFh
		MIFARE® Mini	00h – 13h
KT	Key Type		
		Key A	60h
		Key B	61h
KN	Key Number		
		MIFARE® Key Number	0 to 159 (00h to 9Fh) The key number 0 to 79 are reserved for the non volatile key stored in EEPROM The key number 80 to 159 are reserved for the volatile key stored in RAM
SW1-SW2	Command execution status		
		Command executed successfully	90h 00h
		Others	Refer to the error codes table below

The authentication is performed for a memory sector. As each memory sector is composed of four memory blocks, the authentication will be done for all the four memory blocks.

The authentication operation is not required for a MIFARE® Ultralight chip.

If an authentication fails, the smart card is automatically deselected (MIFARE® specification). However the reader/writer automatically recovers the communication with the smart card.

Read Binary command

The Read Binary command is used to read data from a MIFARE® memory area.

Data consist of a memory block (16 bytes) or a memory page (4 bytes).

This command is formatted as follows:

CLA	INS	P1	P2	Lc
FFh	B0h	ABLM	ABLL	Size
1 byte	1 byte	1 byte	1 byte	1 byte

The response is formatted as follows:

Data	SW1	SW2
16 bytes	1 byte	1 byte

Where:

ABLM	Address Block MSB		00h
ABLL	Address Block LSB		
		MIFARE® 1K	00h – 3Fh
		MIFARE® 4K	00h – FFh
		MIFARE® Mini	00h – 13h
		MIFARE® Ultralight	00h – 0Fh
Size		Size of the memory area	
		MIFARE® 1K, 4K, Mini	10h (size of the memory block)
		MIFARE® Ultralight	04h (size of the memory page)
			If this parameters is 00h then all the bytes of the block will be returned (16 bytes or 4 bytes)
Data			
		MIFARE® 1K, 4K, Mini	16-byte of data
		MIFARE® Ultralight	4-byte of data
			The first byte of the block is byte 0
			Present only when there is no error in the status report.
SW1-SW2	Command execution status		

Command executed successfully	90 00h
Others	Refer to the error codes table below

Note:

For MIFARE® 1K, MIFARE® 4K and MIFARE® Mini, it is mandatory to perform the General Authenticate command before each read memory block operation. Otherwise, an authentication error will occur.

For MIFARE® Ultralight the General Authenticate operation is not required. Refer to the appendix for the MIFARE® Ultralight read operation.

Update Binary command

The Update Binary command is used to write data into a MIFARE® memory area.

Data consist of a memory block (16 bytes) or a memory page (4 bytes).

This command is formatted as follows:

CLA	INS	P1	P2	Lc	DATA
FFh	D6h	ABLM	ABLL	Size	Data
1 byte	1 byte	1 byte	1 byte	1 byte	16 bytes

The response is formatted as follows:

SW1	SW2
1 byte	1 byte

Where:

ABLM	Address Block MSB	00h
ABLL	Address Block LSB	
	MIFARE® 1K	00h – 3Fh
	MIFARE® 4K	00h – FFh
	MIFARE® Mini	00h – 13h
	MIFARE® Ultralight	00h – 0Fh (*)
Size	Size of the memory area	
	MIFARE® 1K, 4K, Mini	10h (size of the memory block)
	MIFARE® Ultralight	04h (size of the memory page)
Data		
	MIFARE® 1K, 4K, Mini	16-byte of data
	MIFARE® Ultralight	4-byte of data
		The first byte of the block is byte 0
		Present only when there is no error

			in the status report.
SW1-SW2	Command execution status		
	Command executed successfully		90 00h
		Others	Refer to the error codes table below

Note:

For MIFARE® 1K, MIFARE® 4K and MIFARE® Mini, it is mandatory to perform the General Authenticate command before each write memory block operation. Otherwise, an authentication error will occur.

For MIFARE® Ultralight the General Authenticate operation is not required. Refer to the appendix for the MIFARE® Ultralight write operation.

Error code list summary

The error codes returned by the commands listed above are defined in the following table:

SW1	SW2	Meaning
Get Data error codes		
62h	82h	End of data reach before Le bytes (Le is greater than data length)
67h	00h	Wrong length
6Ah	81h	Function not supported
6Bh	00h	Wrong parameter P1-P2
6Ch	XXh	Wrong length (wrong number Le; XX is the exact number) if Le is less than the available data length
6Dh	00h	Instruction code not supported
Load Keys error codes		
65h	81h	Memory failure
67h	00h	Wrong length
69h	83h	Reader key not supported
69h	85h	Secure transmission not supported
69h	88h	Key number not valid
69h	89h	Key length is not correct
General Authenticate error codes		
67h	00h	Wrong length
69h	82h	Security status not satisfied
69h	83h	Authentication cannot be done
69h	85h	Secure transmission not supported
69h	86h	Key type not known
69h	88h	Key number not valid
6Ah	81h	Function not supported
6Bh	00h	Wrong parameter P1-P2
6Dh	00h	Instruction code not supported
Read Binary error codes		
62h	82h	End of data reach before Le bytes (Le is greater than data length)

67h	00h	Wrong length
68h	00h	Class byte is not correct
69h	82h	Security not satisfied
69h	85h	Address out of range
6Ah	81h	Function not supported
6Ch	XX	Wrong length (wrong number Le; XX is the exact number) if Le is less than the available data length
Update Binary error codes		
67h	00h	Wrong length
69h	82h	Security not satisfied
69h	85h	Address out of range
6Ah	81h	Function not supported

Table 18 – Memory card error codes

Interfacing with Contact Cards

ISO7816 asynchronous smart cards are accessible via standard PC/SC using Microsoft's library "winscard.dll". This type of cards supports at least one of the asynchronous protocols T=0 or T=1. No additional libraries or third-party software components are necessary to integrate ISO7816 smart cards.

As defined in the PC/SC specifications, the Prox–DU and the Prox–SU devices handle all the ISO7816-4 Inter Industry commands to interface ISO7816 asynchronous contact smart cards.

In addition the Prox–DU device will support the following smart card events:

- Insertion
- Removal

As the Prox–SU has no capability to detect a smart card insertion or removal, the SIM/SAM card will always be considered as inserted when the SIM/SAM card is into its connector.

Detecting an Insertion

The contact reader/writer will check if a smart card is inserted into the slot.

When a smart card insertion is detected, its properties are recorded and a CCID insertion notification message will be generated.

Detecting a Removal

A smart card being removed from the slot is detected by the contact reader/writer.

When a smart card removal is detected, a CCID removal notification message will be generated.

ATR for Contact Smart Cards

The Answer To Request (ATR) returned by a contact smart card is compliant with the ISO7816-3 specifications.

The Prox–DU and the Prox–SU will return the smart card ATR after a smart card power up.

The ATR is as follows:

Byte Number	Value	Designation	Description
0	3Bh or 3Fh	TS	Initial header (Mandatory) Direct or inverse convention

1	Y1-K	T0	Format character (Mandatory) Encodes Y1 and K Y1 indicator for the presence of the interface characters TA1-TB1-TC1-TD1 K=number of historical bytes
2	Fi-Di	TA1	Interface characters (Optional) Global, encodes Fi and Di
3	XX	TB1	Interface characters (Optional) Global, deprecated
4	N	TC1	Interface characters (Optional) Global, encodes N
5	Y2-T	TD1	Interface characters (Optional) Strutctural, encodes Y2 and T
6	XX	TA2	Interface characters (Optional) Global, specific mode byte
7	XX	TB2	Interface characters (Optional) Global, deprecated
8	XX	TC2	Interface characters (Optional) Specific to T=0
9	Y3-T	TD2	Interface characters (Optional) Structural, encodes Y3 and T
		-	For i > 2
	Yi-T	TDi-1	Interface characters (Optional) Structural, encodes Yi and T
	XX	TAi	Interface characters (Optional) Specific to T after T from 0 to 14 in TDi–1 Global after T=15 in TDi–1
	XX	TBi	
	XX	TCi	
	Yi+1-T	TDi	Interface characters (Optional) Interface characters (Optional) Structural, encodes Yi+1 and T
		-	
	XX	T1	Historical characters (Optional): max 15 bytes

	XX XX	… .. Tk	
N	UU	TCK	Check character (Conditional): Exclusive-OR of bytes T0 to Tk

Table 19 – ATR for contact smart cards

Structures and content

A reset operation results in the answer from the smart card consisting of the initial character TS followed by at most 32 characters in the following order:

• T0	Format character	(Mandatory)
• TAi, TBi, TCi, TDi	Interface characters	(Optional)
• T1, T2, ... ,TK	Historical characters	(Optional)
• TCK	Check character	(Conditional)

The interface characters specify physical parameters of the integrated circuit in the smart card and logical characteristics of the subsequent exchange protocol.

The historical characters designate general information, for example, the smart card manufacturer, the chip inserted in the smart card, the masked ROM in the chip, the state of the life of the smart card. The specification of the historical characters falls outside the scope of this part of ISO7816.

For simplicity, T0, TAi, ... ,TCK will designate the bytes as well as the characters in which they are contained.

Structure of the subsequent characters in the ATR

The initial character TS is followed by a variable number of subsequent characters in the following order: The format character T0 and, optionally the interface characters TAi, TBi, TCi, TDi and the historical characters T1, T2, ... , TK and conditionally, the check character TCK.

The presence of the interface characters is indicated by a bit map technique explained below.

The presence of the historical characters is indicated by the number of bytes as specified in the format character defined below.

The presence of the check character TCK depends on the protocol type(s) as defined as below.

Format character T0

The T0 character contains two parts:

- The most significant half byte (b4, b5, b5, b7) is named Y1 and indicates with a logic level ONE the presence of subsequent characters TA1, TB1, TC1, TD1 respectively.
- The least significant half byte (b3 to b0) is named K and indicates the number (0 to 15) of historical characters.

b7	b6	b5	b4	b3	b2	b2	b0

Y1	K

Figure 18 – Information provided by T0

Y1: indicator for the presence of the interface characters

- TA1 is transmitted when b4=1
- TB1 is transmitted when b5=1
- TC1 is transmitted when b6=1
- TD1 is transmitted when b7=1
- K: number of historical characters

Interface characters TAi, TBi, TCi, TDi

TAi, TBi, TCi (i=1, 2, 3, ...) indicate the protocol parameters.

Each interface byte TA, TB or TC is either global or specific:

- Global interface bytes refer to parameters of the integrated circuit within the smart card,
- Specific interface bytes refer to parameters of a transmission protocol offered by the smart card.

TDi indicates the protocol type T and the presence of subsequent characters.

Bits b4, b5, b6, b7 of the byte containing Yi (T0 contains Y1; TDi contains Yi+1) state whether character TAi for b4, character TBi for b5, character TCi for b6, character TDi for b7 are or are not (depending on whether the relevant bit is 1 or 0) transmitted subsequently in this order after the character containing Yi.

When needed, the interface device shall attribute a default value to information corresponding to a non transmitted interface character.

When TDi is not transmitted, the default value of Yi+1 is null, indicating that no further interface characters TAi+j, TBi+j, TCi+j, TDi+j will be transmitted.

b7	b6	b5	b4	b3	b2	b2	b0
Yi+1				T			

Figure 19 – Information provided by TDi

Yi+1: indicator for the presence of the interface characters

- TAi+1 is transmitted when b4=1
- TBi+1 is transmitted when b5=1
- TCi+1 is transmitted when b6=1
- TDi+1 is transmitted when b7=1

T: Protocol type for subsequent transmission.

If TD1, TD2 and so on are present, the encoded types T shall be in ascending numerical order. If present, T=0 shall be first, T=15 shall be last. T=15 is invalid in TD1.

Historical characters T1, T2, ... ,TK

When K is not null, the answer to reset is continued by transmitting K historical characters T1, T2, ... , TK.

Check character TCK

The value of TCK shall be such that the exclusive-oring of all bytes from T0 to TCK included is null.

Protocol type T

The four least significant bits of any interface character TDi indicate a protocol type T, specifying rules to be used to process transmission protocols. When TDi is not transmitted, T=0 is used.

- T=0 is the asynchronous half duplex character transmission protocol.
- T=1 is the asynchronous half duplex block transmission protocol.
- T=2 and T=3 are reserved for future full duplex operations.
- T=4 is reserved for an enhanced asynchronous half duplex character transmission protocol.
- T=5 to T=13 are reserved for future use.
- T=14 is reserved for protocols not standardized by ISO.
- T=15 does not refer to a transmission protocol, but only qualifies global interface bytes.

Note: If only T=0 is indicated, TCK shall not be sent. In all other cases TCK shall be sent.

Specifications of the global interface bytes

Among the interface bytes possibly transmitted by the smart card in answering to reset, this subclaus defines only the global interface bytes TA1, TB1, TC1, TA2, TB2, the first TA for T=15 and the first TB for T=15.

These global interface bytes convey information to determine parameters which the interface device shall take into account.

TA1

TA1 encodes the indicated value of the clock rate conversion integer (Fi), the indicated value of the baud rate adjustment integer (Di) and the maximum value of the frequency supported by the smart card (f (max.)). The default values are Fi = 372, Di = 1 and f (max.) = 5 MHz.

Fi	0000	0001	0010	0011	0100	0101	0110	0111
F	372	372	558	744	1116	1488	1860	RFU
Fs (max) MHz	4	5	6	8	12	16	20	-

Fi	1000	1001	1010	1011	1100	1101	1110	1111
F	RFU	512	768	1024	1536	2048	RFU	RFU
Fs (max) MHz	-	5	7.5	10	15	20	-	-

Table 20 – Clock rate conversion factor F

Di	0000	0001	0010	0011	0100	0101	0110	0111
D	RFU	1	2	4	8	16	32	64

Di	1000	1001	1010	1011	1100	1101	1110	1111
D	12	20	RFU	RFU	RFU	RFU	RFU	RFU

Table 21 – Bit rate adjustment factor D

TB1 and TB2

TB1 and TB2 are deprecated. The smart card should not transmit them. The interface device shall ignore them.

Note: The first two editions of ISO 7816-3 specified TB1 and TB2 to fix electrical parameters of the integrated circuit for the deprecated use of contact C6.

TC1

TC1 encodes the extra guard time integer (N) from 0 to 255 over the eight bits. The default value is N = 0.

If N = 0 to 254, then before being ready to receive the next character, the smart card requires the following delay from the leading edge of the previous character (transmitted by the smart card or the interface device):

$$GT = 12 \text{ etu} + R \times N/f$$

- If T=15 is absent in the Answer-to-Reset, then R = F / D, i.e., the integers used for computing the etu.
- If T=15 is present in the Answer-to-Reset, then R = Fi / Di, i.e., the integers defined above by TA1.

No extra guard time is used to transmit characters from the card: GT = 12 etu.

The use of N = 255 is protocol dependent: GT = 12 etu in PPS and in T=0. For the use of N = 255 in T=1, refer to the ISO7816-3 standard.

TA2

TA2 is the specific mode byte. For the use of TA2 refer to the ISO7816-3 standard.

Bit 7 indicates the ability for changing the negotiable/specific mode:

- capable to change if bit 7 is set to 0,
- unable to change if bit 7 is set to 1.

Bits 6 and 5 are reserved for future use (set to 0 when not used).

Bit 4 indicates the definition of the parameters F and D.

- If bit 4 is set to 0, then the integers Fi and Di defined above by TA1 shall apply.
- If bit 4 is set to 1, then implicit values (not defined by the interface bytes) shall apply.

Bits 3 to 0 encode a type T.

The first TA1 for T=15

The first TA for T=15 encodes the clock stop indicator (X) and the class indicator (Y). The default values are X = "clock stop not supported" and Y = "only class A supported". For the use of clock stop and for the use of the classes of operating conditions refer to the ISO7816-3 standard.

- According to the next table, bits 7 and 6 indicate whether the smart card supports clock stop (≠ 00) or not (= 00) and, when supported, which state is preferred on the electrical circuit CLK when the clock is stopped.

Bits 7 and 6	00	01	10	11
X	Clock stop not supported	State L	State H	No preference

Table 22 – clock stop indicator X

- According to the next table 10, bits 5 to 1 indicate the classes of operating conditions accepted by the smart card. Each bit represents a class: bit 1 for class A, bit 2 for class B and bit 3 for class C.

Bits 5 to 0	00 0001	00 0010	00 0100	00 0011
Y	A only (5V)	B only (3V)	C only (1.8V)	A and B
Bits 5 to 0	00 0110	00 0111	Any other value	
Y	B and C	A, B and C	RFU	

Table 23 – class indicator Y

The first TB for T=15

The first TB for T=15 indicates the use of standard or proprietary use contact (SPU) by the smart card. The default value is "SPU not used".

Coded over bits 6 to 0, the use is either standard (bit 7 set to 0), or proprietary (bit 7 set to 1). The value '00' indicates that the smart card does not use SPU. Any other value where bit 7 is set to 0 are reserved for future use.

For additional information about the ATR contents please refer to the ISO7816-3 standard.

MIFARE® Cards Mapping

MIFARE® 1K Memory Mapping

This is an 8-Kbit (1 Kbyte) MIFARE® memory contactless smart card arranged as 16 four-block sectors as shown in the following table:

		Bytes																
Sector	Block	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Description
0	0																	Manufacturer Block
	1																	Data
	2																	Data
	3	Key A						Access Bits				Key B						Sector Trailer 0
1	4																	Data
	5																	Data
	6																	Data
	7	Key A						Access Bits				Key B						Sector Trailer 1
2	8																	Data
	9																	Data
	10																	Data
	11	Key A						Access Bits				Key B						Sector Trailer 2
– – –	– – –	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–		– – –
15	60																	Data
	61																	Data
	62																	Data
	63	Key A						Access Bits				Key B						Sector Trailer 15

Table 24 – Memory Sectors of MIFARE® 1K

Each contactless smart card consists of a 16-byte memory block assembled in sectors.

The first block of the first sector contains manufacturing information.

The last block of each sector is the sector trailer containing the keys and the access conditions of the blocks.

MIFARE® Mini Memory Mapping

This is a 2.5-Kbit (320 bytes) MIFARE® memory contactless smart card arranged as 5 four-block sectors as shown in the following table:

		Bytes																
Sector	Block	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Description
0	0																	Manufacturer Block
	1																	Data
	2																	Data
	3	Key A						Access Bits				Key B						Sector Trailer 0
1	4																	Data
	5																	Data
	6																	Data
	7	Key A						Access Bits				Key B						Sector Trailer 1
2	8																	Data
	9																	Data
	10																	Data
	11	Key A						Access Bits				Key B						Sector Trailer 2
– – –	– – –	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–		– – –
4	16																	Data
	17																	Data
	18																	Data
	19	Key A						Access Bits				Key B						Sector Trailer 4

Table 25 – Memory Sectors of MIFARE® Mini

Each contactless smart card consists of a 16-byte memory block assembled in sectors.

The first block of the first sector contains manufacturing information.

The last block of each sector is the sector trailer containing the keys and the access conditions of the blocks.

MIFARE® 4K Memory Mapping

This is a 32-Kbit (4 Kbytes) MIFARE® memory contactless smart card arranged as 32 four-block sectors and 8 sixteen-block sectors as shown in the following table:

		Bytes																
Sector	Block	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Description
0	0																	Manufacturer Block
	1																	Data
	2																	Data
	3	Key A						Access Bits				Key B						Sector Trailer 0
1	4																	Data
	5																	Data
	6																	Data
	7	Key A						Access Bits				Key B						Sector Trailer 1
2	8																	Data
	9																	Data
	10																	Data
	11	Key A						Access Bits				Key B						Sector Trailer 2
– – –	– – –	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	– – –
31	124																	Data
	125																	Data
	126																	Data
	127	Key A						Access Bits				Key B						Sector Trailer 31
32	128																	Data
	129																	Data
	130																	Data
	131																	Data
	132																	Data
	133																	Data
	134																	Data

	135																	Data
	136																	Data
	137																	Data
	138																	Data
	139																	Data
	140																	Data
	141																	Data
	142																	Data
	143	Key A						Access Bits				Key B						Sector Trailer 32
– – –	– – –	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	– – –
39	240																	Data
	241																	Data
	242																	Data
	243																	Data
	244																	Data
	245																	Data
	246																	Data
	247																	Data
	248																	Data
	249																	Data
	250																	Data
	251																	Data
	252																	Data
	253																	Data
	254																	Data
	255	Key A						Access Bits				Key B						Sector Trailer 39

Table 26 – Memory Sectors of MIFARE® 4K

Each contactless smart card consists of a 16-byte memory block assembled in sectors.

The first block of the first sector contains manufacturing information.

The last block of each sector is the sector trailer containing the keys and the access conditions of the blocks.

MIFARE® UL Memory Mapping

The MIFARE® Ultralight chip is a 512-bit EEPROM memory card.

The MIFARE® UL memory is organized in 16 pages with 4 bytes each as depicted in the following table:

Byte Number	0	1	2	3	Page
Serial Number	SN0	SN1	SN2	BCC0	0
Serial Number	SN3	SN4	SN5	SN6	1
Internal/Lock	BCC1	Internal	Lock0	Lock1	2
OTP	OTP0	OTP1	OTP2	OTP3	3
Data Read-Write	Data0	Data1	Data2	Data3	4
Data Read-Write	Data4	Data5	Data6	Data7	5
Data Read-Write	Data8	Data9	Data10	Data11	6
Data Read-Write	Data12	Data13	Data14	Data15	7
Data Read-Write	Data16	Data17	Data18	Data19	8
Data Read-Write	Data20	Data21	Data22	Data23	9
Data Read-Write	Data24	Data25	Data26	Data27	10
Data Read-Write	Data28	Data29	Data30	Data31	11
Data Read-Write	Data32	Data33	Data34	Data35	12
Data Read-Write	Data36	Data37	Data38	Data39	13
Data Read-Write	Data40	Data41	Data42	Data43	14
Data Read-Write	Data44	Data45	Data46	Data47	15

Bold frame indicates user area.

Table 27 – Memory mapping of MIFARE® UL

Serial Number Area

SN0-SN7 is the 7 bytes serial number according to ISO14443-3.

BCC0 and BCC1 are the check bytes according to ISO14443-3.

Internal byte is reserved for internal data.

These 10 bytes are write-protected after having been programmed by the chip manufacturer after production.

Lock Bytes Area

Lock0 and Lock1 represent the field-programmable read-only locking mechanism. Each Page x from 3 (OTP) to 15 may be locked individually to prevent further write access by setting the corresponding locking bit Lx to 1. After locking the page is read-only memory.

The 3 least significant bits of lock byte 0 are the block-locking bits. Bit 2 handles pages 15 to 10, bit 1 pages 9 to 4 and bit 0 page 3 (OTP). Once the blocking-locking bits are set the locking configuration for the corresponding memory area is frozen - for example if BL15-10 is set to "1", L15 to L10 (bit 7 to bit 2 of lock byte 2) can no longer be changed.

Lock0 byte

MSB							LSB
L7	L6	L5	L4	L OTP	BL 15-10	BL 9-4	BL OTP

Lock1 byte

MSB							LSB
L15	L14	L13	L12	L11	L10	L9	L8

Lx locks Page x to read-only

BLx blocks further locking for the memory area x

The locking and block-locking bits are set via standard write command to Page 2.

Bytes 2 and 3 of the write command and the actual contents of the lock bytes are bite-wise "OR-ed" and the result then becomes the new contents of the lock bytes.

This process is irreversible. If a bit is set to "1", it cannot be changed back to "0" again.

Note: The content of bytes 0 and 1 of Page 2 is not affected by the corresponding data bytes of the write command.

Warning: To activate the new locking configuration after a write to the lock bit area, a new smart card selection has to be carried out.

OTP Bytes Area

Page 3 is the OTP page. It is pre-set to all "0" (zeros) after production. These bytes may be bit-wise modified by a write command.

The bytes of the write command of the current contents of the OTP bytes are bit-wise "OR-ed" and the result becomes the new contents of the OTP bytes.

This process is irreversible. If a bit is set to "1", it cannot be changed back to "0" again.

Note: This memory area may be used as a 32 ticks one-time counter.

Data Bytes Area

Pages 4 to 15 constitute the user read/write area. After production the data pages are initialized to all "0" (zeroes).

MIFARE® UL Read/Write Operation

The MIFARE® Ultralight chip does not embed the MIFARE® Classic security.

So no authentication operation is required before any read/write operation.

MIFARE® Memory Organization

Sector Trailer

The last block of every sector is the sector trailer. It contains the individual secret authentication Key A, optional Key B and the access condition bits for the blocks of the particular sector.

Sector Trailer	Byte	bit 8 MSB	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1 LSB
Secret Key	0	Authentication Key A							
	1								
	2								
	3								
	4								
	5								
Access Bits	6	$\overline{C2_3}$	$\overline{C2_2}$	$\overline{C2_1}$	$\overline{C2_0}$	$\overline{C1_3}$	$\overline{C1_2}$	$\overline{C1_1}$	$\overline{C1_0}$
	7	$C1_3$	$C1_2$	$C1_1$	$C1_0$	$\overline{C3_3}$	$\overline{C3_2}$	$\overline{C3_1}$	$\overline{C3_0}$
	8	$C3_3$	$C3_2$	$C3_1$	$C3_0$	$C2_3$	$C2_2$	$C2_1$	$C2_0$
Data	9								
Secret Key	10	Authentication Key B							
	11								
	12								
	13								
	14								
	15								
CXy: Access bit x for block y $\overline{CXy}$: Complement of CXy									

Authentication Keys

Each sector contains a six-byte authentication Key A and a six-byte optional Key B. All sectors are assigned to the different applications determined by different system providers.

The mutual authentication procedure is performed between the reader/writer and the contactless card and is driven by the reader/writer. Access to the data stored in a sector is only possible after a successful authentication.

The secret authentication keys are always read as logical "0". In applications using only one authentication key, Key A, user can set the access bits where the memory space of the optional authentication Key B can be used for data storage.

In this case when the authentication key, Key B can no longer be used for authentication, the card will not allow memory access using Key B.

Access Bits

The access conditions for the specified operations are defined for each block. The sector trailer and the data blocks are controlled independently.

In sectors consisting of four blocks, the access conditions for each individual block are programmable.

In sectors consisting of sixteen blocks, the 15 data blocks are arranged into three groups of five blocks, with the access conditions are defined independently for each group.

Refer to “Access to Data Blocks” Table and “Access to Sector Trailer” Table for the values of these bytes.

The access bits determine the access rights to the memory using the authentication keys A and B. The access conditions may be altered, provided that the relevant key is known and the actual access condition allows this operation.

The following table describes only access bits in the non-inverted mode (although they can be stored in both non-inverted and inverted mode):

Access Bits			Valid Commands	Block	Description
$C1_0$	$C2_0$	$C3_0$	Read, Write, Increment, Decrement, Transfer, Restore	0	Data Block
$C1_1$	$C2_1$	$C3_1$	Read, Write, Increment, Decrement, Transfer, Restore	1	Data Block
$C1_2$	$C2_2$	$C3_2$	Read, Write, Increment, Decrement, Transfer, Restore	2	Data Block
$C1_3$	$C2_3$	$C3_3$	Read, Write	3	Sector Trailer

Table 19 – Access Bits and the Valid Commands

The internal logic of the MIFARE® circuit ensures that the commands are executed only after an authentication using either Key A or Key B has been successfully performed.

Note: the “Increment”, “Decrement”, “Transfer”, “Restore” commands are not available using the PC/SC V2 MIFARE® commands.

Data Block Access Conditions

The access bits for the data blocks are specified as Never, Key A or Key B.

The setting of the relevant access bits defines the application and the resulting applicable commands. Key A | B indicates that access is possible only after an authentication using Key A or Key B of this sector.

The access condition for every block is dependant on the sector number as explained in the following table:

Sector	Block	Description
N (0 – 31)	0	$C3_0$ - $C2_0$ - $C1_0$
	1	$C3_1$ - $C2_1$ - $C1_1$
	2	$C3_2$ - $C2_2$ - $C1_2$

	3	$C3_3$ - $C2_3$ - $C1_3$
N (32 – 39)	0	$C3_0$ - $C2_0$ - $C1_0$
	1	
	2	
	3	
	4	
	5	$C3_1$ - $C2_1$ - $C1_1$
	6	
	7	
	8	
	9	
	10	$C3_2$ - $C2_2$ - $C1_2$
	11	
	12	
	13	
	14	
	15	$C3_3$ - $C2_3$ - $C1_3$

Table 20 – Access Condition for Data Blocks

The MIFARE® system regards authentication Key B as the primary key for access control to the data memory. Operations which are performed with authentication Key A can also be done with authentication Key B. But, only some sensitive operations can be performed with Key B.

The previous Table “Access Bits and the Valid Commands” shows the types of access conditions associated with their bit values and the access granted by authentication with Key A and Key B.

Access Bits			Access Condition Data Block or Superior Block Group b = 0, 1, 2				
$C1_b$	$C2_b$	$C3_b$	Read	Write	Increment	Decrement/ Transfer/ Restore	Comments
0	0	0	Key A | B[1]	Key A | B[1]	Key A | B[1]	Key A | B[1]	A or B All function memory block
0	0	1	Key A | B[1]	Never	Never	Key A | B[1]	A or B Read /Subtract Value block
0	1	0	Key A | B[1]	Never	Never	Never	A or B Read only memory block
0	1	1	Key B[1]	Key B[1]	Never	Never	B Read /Write memory block

1	0	0	Key A | B[1]	Key B[1]	Never	Never	A or B Read and B Write memory block
1	0	1	Key B[1]	Never	Never	Never	B Read only memory Block
1	1	0	Key A | B[1]	Key B[1]	Key B[1]	Key A | B[1]	A Read/Subtract B Write/Add Value block
1	1	1	Never	Never	Never	Never	Locked block, Access never allowed

Transport Configuration: When the card is delivered, the access conditions for the sector trailer and the authentication Keys A and B are already containing a particular transport configuration.

[1] When Key B can be read in the corresponding Sector trailer, it cannot be used for authentication. If the reader/writer tries to authenticate any block of a sector with Key B using the shaded access conditions, the card will reject subsequent memory access after authentication.

Table 28 – Access to Data Blocks

Note: the “Increment”, “Decrement”, “Transfer”, “Restore” commands are not available using the PC/SC V2 MIFARE® commands.

The following describes the functions of the blocks in previous Table “Access Condition for Data Blocks”:

Read/Write Block	The operation read and write are allowed,
Value Block	Allows the additional value operations such as Increment, Decrement, Transfer and Restore. In the case ('001') only Read and Decrement are possible for a non-rechargeable card. In the other case ('110') recharging is possible using Key B.
Manufacturer Block	The read-only condition is not affected by the setting of the access bits.
Key Management	In transport configuration, the use of Key A for authentication is mandatory.

Sector Trailer Access Conditions

The access bits for the sector trailer shown in the following table determine the access condition to either of the authentication keys or the access bits themselves to be Never, Key B, or Key A | B.

Key A | B indicates the access for this sector is only possible after an authentication using either Key A or Key B.

Access Bits			Access Condition			Comments
			Authentication Key A	Access Bits	Authentication Key B	

$C1_b$	$C2_b$	$C3_b$	Read	Write	Read	Write	Read	Write	
0	0	0	Never	Key A	Key A	Never	Key A	Key A	Key B may be read
0	0	1	Never	Key A	Key A	Key A	Key A	Key A	Key B may be read. (Transport configuration)
0	1	0	Never	Never	Key A	Never	Key A	Never	Key B may be read
0	1	1	Never	Key B	Key A | B	Key B	Never	Key B	
1	0	0	Never	Key B	Key A | B	Never	Never	Key B	
1	0	1	Never	Never	Key A | B	Key B	Never	Never	
1	1	0	Never	Never	Key A | B	Never	Never	Never	
1	1	1	Never	Never	Key A | B	Never	Never	Never	

The shaded areas are access conditions where Key B is readable and may be used for data.

Table 29 – Access to Sector Trailer

The access conditions for the sector trailer and Key A are predefined as transport configuration upon card delivery.

As Key B is read in transport configuration, new cards are authenticated with Key A.

Note:

The access bits can also be blocked by the user to prohibit any further changes to the access conditions.

As the access bits can be altered by the user, special care should be taken during personalization phase.

For More Information

Standards and Specifications

- PC/SC V2 specifications: Part 3. Requirements for PC-Connected Interface Devices - Revision 2.01.09

End of Document