Windows Smart Card Minidriver Specification

Version 7.07

February 25, 2016

Abstract

Smart card vendors can write card minidrivers to present a consistent interface to their smart card type to the Microsoft Smart Card Base Cryptographic Service Provider (CSP) or Cryptography API: Next Generation (CNG) Key Storage Provider (KSP) and to the smart card management interface. These card minidrivers plug in to Windows operating system code. The functionality in a card minidriver is narrowly scoped and carefully defined so that the card-dependent code is simple to implement and easy to verify functionally.

This specification provides implementation guidelines for Smart Card Base CSP/KSP card minidrivers.

This information applies to the following operating systems:

Windows 8 and later operating systems

The current version of this paper is maintained on the Web at:   
 [Windows Smart Card Minidriver Specification](http://msdn.microsoft.com/en-us/library/windows/hardware/gg487500.aspx)

**Disclaimer**: This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet website references, may change without notice. Some information relates to pre-released product, which may be substantially modified before it is commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here. You bear the risk of using it.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2016 Microsoft. All rights reserved.

C:\Users\jenlin\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.Outlook\KN5ONHWU\dep_MicrosoftLogotype.png

Document History

|  |  |  |
| --- | --- | --- |
| **Date** | **Version** | **Change** |
| February 25, 2016 | 7.07 | Updated table in section 4.2.1.4 (**PIN\_CACHE\_POLICY\_TYPE** – enumeration) |
| February 18, 2016 | 7.07 | Updated **General Design Guidelines** bullets: Replaced WHQL reference with information about Hardware Compatibility Program. Updated name of API kit to show CPDK. |
| May, 21, 2015 | 7.07 | Updated with a list of requirements to run in LSA Protected Process Plugins or Drivers, correct permissions for CardReadFile in section 5 and guidance on using S\_CARD\_E\_FILE\_NOT\_FOUND as an umbrella for CardReadFile. |
| October 28, 2014 | 7.07 | Updated with description about how to implement a minidriver when a card only supports one session pin |
| October 19, 2012 | 7.07 | Updated with information about the generic minidriver that supports the Generic Identity Device Specification (GIDS) Card Edge with Microsoft Electrical profile |
| September 25, 2009 | 7.07 | Changed MS PnP AID to SC PnP AID |
| July 9, 2009 | 7.06 | First publication |

Contents

[1 Introduction 6](#_Toc338403393)

[1.1 What’s New in this version 8](#_Toc338403394)

[1.1.1 Generic Inbox minidriver that supports the GIDS card edge 8](#_Toc338403395)

[2 Card-Specific Minidriver Details 9](#_Toc338403396)

[2.1 Overview 9](#_Toc338403397)

[2.2 Related Document 9](#_Toc338403398)

[2.3 General Design Guidelines 9](#_Toc338403399)

[2.3.1 Transaction Management 9](#_Toc338403400)

[2.3.2 Conventions 10](#_Toc338403401)

[2.3.3 Authentication and Authorization 10](#_Toc338403402)

[2.3.4 Handling Memory Allocations 10](#_Toc338403403)

[2.4 Caching 11](#_Toc338403404)

[2.5 Mandatory Version Checking 11](#_Toc338403405)

[2.5.1 CARD\_DATA Version Checks 11](#_Toc338403406)

[2.5.2 Other Structure Version Checks 12](#_Toc338403407)

[3 Registration and General Import Mechanisms 12](#_Toc338403408)

[3.1 **DllMain** and Registration Mechanisms 12](#_Toc338403409)

[3.1.1 **DllMain** 12](#_Toc338403410)

[3.1.2 **DllRegisterServer** and **DllUnregisterServer** 13](#_Toc338403411)

[3.1.3 Registration Mechanisms 13](#_Toc338403412)

[3.1.4 Smart Card INF File Requirements 13](#_Toc338403413)

[3.2 API Imports from the Smart Card Base CSP/KSP 14](#_Toc338403414)

[3.2.1 Memory Management Functions 14](#_Toc338403415)

[3.2.2 Cache Functions 15](#_Toc338403416)

[3.2.3 Cryptographic Utilities 17](#_Toc338403417)

[3.3 API Imports from the Smart Card Resource Manager 19](#_Toc338403418)

[4 Card Minidriver API Reference 19](#_Toc338403419)

[4.1 Initialization and Deconstruct 19](#_Toc338403420)

[4.1.1 **CardAcquireContext** 19](#_Toc338403421)

[4.1.2 **CardDeleteContext** 23](#_Toc338403422)

[4.2 Card PIN Operations 23](#_Toc338403423)

[4.2.1 Data Structures and Enumerations 24](#_Toc338403424)

[4.2.2 **CardAuthenticatePin** 29](#_Toc338403425)

[4.2.3 **CardGetChallenge** 30](#_Toc338403426)

[4.2.4 **CardAuthenticateChallenge** 31](#_Toc338403427)

[4.2.5 **CardDeauthenticate** 32](#_Toc338403428)

[4.2.6 **CardAuthenticateEx** 33](#_Toc338403429)

[4.2.7 **CardGetChallengeEx** 36](#_Toc338403430)

[4.2.8 **CardDeauthenticateEx** 37](#_Toc338403431)

[4.2.9 **CardChangeAuthenticatorEx** 38](#_Toc338403432)

[4.2.10 **CardUnblockPin** 39](#_Toc338403433)

[4.2.11 **CardChangeAuthenticator** 41](#_Toc338403434)

[4.3 Public Data Operations 42](#_Toc338403435)

[4.3.1 **CardCreateDirectory** 43](#_Toc338403436)

[4.3.2 **CardDeleteDirectory** 44](#_Toc338403437)

[4.3.3 **CardReadFile** 44](#_Toc338403438)

[4.3.4 **CardCreateFile** 45](#_Toc338403439)

[4.3.5 **CardGetFileInfo** 47](#_Toc338403440)

[4.3.6 **CardWriteFile** 47](#_Toc338403441)

[4.3.7 **CardDeleteFile** 49](#_Toc338403442)

[4.3.8 **CardEnumFiles** 49](#_Toc338403443)

[4.3.9 **CardQueryFreeSpace** 50](#_Toc338403444)

[4.4 Card Capabilities (Minidriver Version 5 and Earlier) 51](#_Toc338403445)

[4.4.1 Defines and Data Structures 51](#_Toc338403446)

[4.4.2 **CardQueryCapabilities** 52](#_Toc338403447)

[4.5 Card and Container Properties 52](#_Toc338403448)

[4.5.1 Defines and Data Structures 52](#_Toc338403449)

[4.5.2 **CardGetContainerProperty** 53](#_Toc338403450)

[4.5.3 **CardSetContainerProperty** 54](#_Toc338403451)

[4.5.4 **CardGetProperty** 56](#_Toc338403452)

[4.5.5 **CardSetProperty** 60](#_Toc338403453)

[4.6 Key Container 63](#_Toc338403454)

[4.6.1 **CardCreateContainer** 63](#_Toc338403455)

[4.6.2 **CardCreateContainerEx** 65](#_Toc338403456)

[4.6.3 **CardDeleteContainer** 67](#_Toc338403457)

[4.6.4 **CardGetContainerInfo** 67](#_Toc338403458)

[4.7 Cryptographic Operations 69](#_Toc338403459)

[4.7.1 **CardRSADecrypt** 69](#_Toc338403460)

[4.7.2 **CardConstructDHAgreement** 70](#_Toc338403461)

[4.7.3 **CardDeriveKey** 71](#_Toc338403462)

[4.7.4 **CardDestroyDHAgreement** 74](#_Toc338403463)

[4.7.5 **CardSignData** 74](#_Toc338403464)

[4.7.6 **CardQueryKeySizes** 78](#_Toc338403465)

[4.8 Secure Key Injection 78](#_Toc338403466)

[4.8.1 Defines and Structures 80](#_Toc338403467)

[4.8.2 Server Functions 86](#_Toc338403468)

[4.8.3 Shared Functions 89](#_Toc338403469)

[4.8.4 Client functions 97](#_Toc338403470)

[5 File System Requirements 99](#_Toc338403471)

[5.1 File Naming Requirements 99](#_Toc338403472)

[5.2 File System Virtualization 99](#_Toc338403473)

[5.3 Physical Card Data Layout 99](#_Toc338403474)

[5.4 Logical Data Layout 100](#_Toc338403475)

[5.4.1 Card Identifier 100](#_Toc338403476)

[5.4.2 Application Directory 100](#_Toc338403477)

[5.4.3 Cache File 101](#_Toc338403478)

[5.4.4 Container Map File 101](#_Toc338403479)

[5.5 Data Layout Summary 103](#_Toc338403480)

[5.6 File Access Control 104](#_Toc338403481)

[5.6.1 Known Principals 104](#_Toc338403482)

[5.6.2 Directory Access Conditions 104](#_Toc338403483)

[5.6.3 File Access Operations 105](#_Toc338403484)

[5.6.4 File Access Conditions 105](#_Toc338403485)

[5.6.5 Notes on the Directory and File Access Conditions 106](#_Toc338403486)

[6 Card Requirements 107](#_Toc338403487)

[6.1 What a “Blank Card” Is 107](#_Toc338403488)

[6.2 Card “Creation” 107](#_Toc338403489)

[7 Developer Notes and Guidelines 108](#_Toc338403490)

[7.1 Challenge/Response Method of Unblocking Smart Card PIN 108](#_Toc338403491)

[7.2 Enhanced PIN Support 108](#_Toc338403492)

[7.3 Session PINs and Secure PIN Channel 109](#_Toc338403493)

[7.4 Read-Only Cards 110](#_Toc338403494)

[7.5 Cache Modes 112](#_Toc338403495)

[7.6 Challenge/Response Mechanism 113](#_Toc338403496)

[7.7 Interoperability with msroots 115](#_Toc338403497)

[7.8 Group Policy Settings for Microsoft Base Smart Card CSP 116](#_Toc338403498)

[7.9 Group Policy Settings for Microsoft CNG Smart Card KSP 117](#_Toc338403499)

[7.10 Known Issues 117](#_Toc338403500)

[Appendix A. Smart Card Plug and Play 118](#_Toc338403501)

[A.1 Pairing Process 118](#_Toc338403502)

[A.2 Sample INF for x86 and amd64 119](#_Toc338403503)

[Appendix B. Use Case Scenario for Secure Key Injection 122](#_Toc338403504)

[Appendix C. Overview of the Windows Inbox Smart Card Minidriver 125](#_Toc338403505)

[C.1 Electrical Profile for GIDS cards with the Microsoft Generic Profile 125](#_Toc338403506)

[C.1.1 GIDS Application Metadata 125](#_Toc338403507)

[C.1.2 PIN Creation 126](#_Toc338403508)

[C.1.3 Pin Unblock Key (PUK) Creation 127](#_Toc338403509)

[C.1.4 ACL Creation 127](#_Toc338403510)

[C.1.5 Create EF for Admin Key 128](#_Toc338403511)

[C.1.6 Inject Admin Key 128](#_Toc338403512)

[C.1.7 Set Operational State 129](#_Toc338403513)

[C.1.8 Data objects on a GIDS card after the filesystem is created 129](#_Toc338403514)

[C.2 INF Sample to re-brand inbox class minidriver 130](#_Toc338403515)

[Appendix D. Smart Card Discovery Process 134](#_Toc338403516)

[D.1 Smart Card Plug and Play Process 135](#_Toc338403517)

[D.2 Winscard Discovery Process 136](#_Toc338403518)

[D.3 Windows Smart Card Class Minidriver Discovery Process 137](#_Toc338403519)

[D.4 Selection Mechanisms 138](#_Toc338403520)

[D.4.1 Applications that Contain Microsoft identifiers 138](#_Toc338403521)

[D.4.2 GET DATA 138](#_Toc338403522)

[D.4.3 SELECT PIV AID Command 138](#_Toc338403523)

[D.4.4 SELECT MS GIDS AID Command 138](#_Toc338403524)

[D.4.5 Use of the ATR Historical Bytes 139](#_Toc338403525)

[D.4.6 Windows Smart Card Framework Card Identifier 139](#_Toc338403526)

[Appendix E. Acronyms 140](#_Toc338403527)

# Introduction

This document is the specification for smart card minidrivers for the Microsoft® Cryptographic API (CAPI)-based Windows® smart card framework. The smart card minidriver provides a simpler alternative to developing a legacy cryptographic service provider (CSP) by encapsulating most of the complex cryptographic operations from the card minidriver developer.

Beginning with Windows Vista®, applications can use the Microsoft Cryptography API: Next Generation (CNG) for smart card–based cryptographic services. As part of the elliptic curve cryptography (ECC) effort that was introduced in Windows Vista, ECC smart cards are supported in the new cryptographic framework. Applications and interfaces that interact with existing Rivest-Shamir-Adleman (RSA) card minidrivers through the legacy CAPI subsystem continue to work without modification.

RSA smart card minidrivers can also be registered with the smart card key storage provider (KSP) so that they can be called through the CNG interface. Dual-mode ECC/RSA + ECC-only requests are routed to the KSP and, through it, to the appropriate card minidrivers. For Windows Vista–based clients, ECC-only and ECC/RSA dual-mode cards are supported by using the Windows smart card framework. Dual-mode cards can also be accessed through CAPI primarily to expose RSA-only features.

Applications use CAPI for smart card–based cryptographic services. CAPI, in turn, routes these requests to the appropriate CSP to handle the cryptographic requirements.

The Microsoft Smart Card Base CSP and KSP is a refinement of the architecture that separates commonly needed CAPI-based CSP and CNG-based KSP functionality, respectively, from the implementation details that must change for every card vendor.

Although Base CSP can use the RSA capabilities of a smart card only by using the minidriver, the CNG-based KSP supports ECC-only as well as ECC/RSA dual-mode smart cards in Windows Vista and later versions of Windows.

Ultimately, the intention is for the new architecture to support all new smart cards—RSA, ECC, and whatever follows. It splits the implementation of the CSP into two parts:

* The Base CSP/KSP (the common part), which includes functionality for hashing, symmetric, and public key cryptographic operations in addition to personal identification number (PIN) entry and caching.
* A series of plug-ins, which are known as “card minidrivers,” that translate the characteristics of particular smart cards into a uniform interface that is the same for all smart cards. Card minidrivers then communicate with their cards by using the services of the smart card resource manager (SCRM) that similarly abstracts the characteristics of a variety of smart card readers.

The remaining portion for smart card vendors is to implement a card minidriver, a reasonably limited interface layer that provides an abstraction of the card to the Base CSP/KSP and that is organized as a file system, and a set of primitive capabilities. Higher order functionality, such as caching (ensuring that different files on the card have consistent content) or handling naming collisions, is handled at a higher level, outside the card minidriver.

Figures 1 and 2 show the architecture of the system and the position of the card minidriver within it.



Figure 1. Interfaces between card minidrivers and CAPI-based applications



Figure 2. Interfaces between card minidrivers and CAPI2-based applications

We recommend that developers take advantage of the rich set of libraries that Microsoft provides for cryptographic operations that the minidriver performs. This lets developers benefit from the Microsoft Windows Update infrastructure for the distribution of critical security updates.

## What’s New in this version

### Generic Inbox minidriver that supports the GIDS[[1]](#footnote-1) card edge

Windows 8 ships with a generic minidriver that supports the GIDS (Generic Identity Device Specification) Card Edge with Microsoft Electrical profile. Refer to [Appendix C](#_Appendix_C_–) for more information on the process that Windows follows to identify and pair a GIDS card with the inbox minidriver. The generic minidriver is also available on Windows 7 SP1.

# Card-Specific Minidriver Details

## Overview

The card-specific minidriver is the lowest logical interface layer in the Base CSP/KSP. This minidriver lets the Base CSP/KSP and applications interact directly with a specific type of card by using SCRM.

The card minidriver is a DLL that exports a specific set of APIs as defined in this specification. Each call to the card minidriver includes a pointer to a CARD\_DATA structure that provides context information. This context information provides some state information in addition to a table of function pointers that is used to facilitate communication between the upper layer and the card minidriver.

For more information about this context structure, see [**CardAcquireContext**](#_CardAcquireContext)**”** later in this specification.

## Related Document

The *Cardmod.h* C header file provides additional information that is relevant to this specification. This file contains the function prototypes and structures that Microsoft smart card minidriver API specifies. This API is available through the Microsoft Cryptographic Provider Development Kit (CPDK).

## General Design Guidelines

* The card minidriver should be distributed as a DLL.
* Each card-specific operation should implement a single, atomic transaction except as otherwise noted.
* A standardized set of macro-level operations should be implemented.
* The logical card file-system objects should be mapped to their physical locations.
* Cards based on this new model should be able to, dynamically, grow any files that are stored on the card. For cards that are read-only and cannot follow this guideline, the minidriver should follow the specific guidelines for read-only cards that were detailed in this specification.
* The minidriver imports definitions from the CPDK. The minidriver header file (*Cardmod.h*) includes *Bcrypt.h* for this purpose. Implementations must resolve this dependency through Microsoft Visual Studio® project settings for compiling minidrivers.
* Protected Process Requirements for Plug-ins or Drivers
  + For an LSA plug-in, or a driver to successfully load as a protected process, it must meet the following criteria:
    - Signature Verification
      * Protected mode requires that any plug-in loaded into the LSA is digitally signed with a Microsoft signature. Therefore, any unsigned or third party signed plug-ins will fail to load in LSA. Examples of such plug-ins are smart card drivers, crypto plug-ins, password filters, etc.
      * LSA plug-ins that are drivers (such as smart card drivers) need to be digitally signed.
      * **NOTE**: The Windows Hardware Compatibility Program offers the only method for digitally signing drivers for Windows. Therefore, it is important to refer to this web site for more information: <https://msdn.microsoft.com/en-us/library/windows/hardware/dn939961(v=vs.85).aspx>.
  + Adherence to Microsoft Security Development Lifecycle (SDL) Process Guidance
    - All the plug-ins also need to conform to the guidance in the relevant portions of the *Microsoft Security Development Lifecycle (SDL) – Process Guidance* topic. That topic can be found here: <http://msdn.microsoft.com/en-us/library/windows/desktop/cc307891.aspx> . For example, see the 'No Shared Sections' content in Appendix G of the topic.
    - Even if the plug-ins are properly signed with a Microsoft signature, non-compliance with the SDL Process might result in a failure to load the plug-ins.

### Transaction Management

* A card minidriver should assume that transactions are handled by the caller, if it uses SCRM to access the card.
* The card minidriver can assume that all entry points except **CardDeleteContext** are called by holding the card transaction. This cannot be assumed in **CardDeleteContext** because the card might have been removed or it is being called as part of a cleanup procedure.
* Multiple contexts can exist in a single process. Calling **CardDeleteContext** on one process should not prevent the other context from functioning.
* Handling the authentication state of the card is also the responsibility of the caller, not the card minidriver.

### Conventions

#### Strings: UNICODE and ANSI

At the application level, strings are generally encountered as elements of the user interface, either directly or indirectly. Therefore, they usually must be localized (translated into the user’s language) so that they can be understood. For this reason, the string type that most applications use is double-byte (that is, UNICODE) to accommodate different character sets.

However, smart cards operate with minimal resources and with very few options on what to name directories, files, users, and so on. The character set for strings is single-byte ANSI, which provides a more compact representation of string data.

Accordingly, string buffers to and from the card minidriver are expected to be single-byte ANSI, and conversions to and from this character type as required must be performed outside the card minidriver.

#### Error Handling

To ensure consistent error handling, response to failure, and consistent behavior for card minidrivers, the following conventions should be followed:

* All NULL and invalid parameters, including bad flags return SCARD\_E\_INVALID\_PARAMETER.
* All incorrect PIN or attempts with the wrong key return SCARD\_W\_WRONG\_CHV.
* If a generic failure happens, the APIs return SCARD\_E\_UNEXPECTED.

In addition, the errors returned by the functions that are described in the following sections should be from the SCARD\_\* category (*winerror.h*). For example, we recommend that you use SCARD\_E\_INVALID\_PARAMETER (0x80100004) instead of ERROR\_INVALID\_PARAMETER (0x00000057).

### Authentication and Authorization

Beginning with Version 6, the minidriver interface expands the concept of a PIN to beyond just a traditional alphanumeric string.

For more information, see “[SECRET\_TYPE (enumeration)](#_SECRET_TYPE_(enumeration))” later in this specification.

### Handling Memory Allocations

All API elements in this specification that allocate memory buffers internally do so by calling PFN\_CSP\_ALLOC. Because of this, any such memory buffers must be freed by calling PFN\_CSP\_FREE.

Any allocation of memory that the card minidriver performs should be done by using PFN\_CSP\_ALLOC or PFN\_CSP\_REALLOC.

## Caching

The Card Interface layer in the Base CSP/KSP implements a data cache to minimize the amount of data that must be written to or read from the card. The data cache is also made available for the card minidriver to use through function pointers in the CARD\_DATA structure, and the card minidriver should use these pointers to enhance performance by caching its internal data files that are stored on the card.

Data caching requires write access to the card to persist cache freshness counters to the card. The minidriver can control data caching if writing data to the card is not feasible.

For more information on how to control data caching, see the definition of the CP\_CARD\_CACHE\_MODE property in “[**CardGetProperty**](#_CardGetPropertyCardGetProperty)” later in this specification.

## Mandatory Version Checking

All card minidrivers must implement version checks. The version of the CARD\_DATA structure is a negotiation between the version that the caller wants to support and the version that the card minidriver can actually support.

### CARD\_DATA Version Checks

Define minimum version as the minimum version of the card minidriver context structure (that is, CARD\_DATA structure) that is supported, and define the current version as the level for which this card minidriver was designed and for which all card-minidriver-set structure items are guaranteed to be valid on a successful return from **CardAcquireContext**. The current version must be greater than or equal to the minimum version and less than or equal to CARD\_DATA\_CURRENT\_VERSION, which is defined in *Cardmod.h*.

When the calling application calls **CardAcquireContext**, it specifies the desired version that it wants to load. This requested version is set in the **dwVersion** member in the CARD\_DATA structure.

If the requested version is less than the minimum version that the card minidriver supports, **CardAcquireContext** must return a revision mismatch error (see the following sample code).

If the requested version is at least as great as the minimum version, the card minidriver should set the **dwVersion** member to the highest version that it can support that is less than or equal to the requested version.

The following sample code shows the expected card minidriver behavior when checking the version. This is assumed to be in the body of the **CardAcquireContext** function. pCardData is a pointer to the CARD\_DATA structure passed into this call.

#define MINIMUM\_VERSION\_SUPPORTED (4)

#define CURRENT\_VERSION\_SUPPORTED (7)

    // The lowest supported version is 4.

    If (pCardData->dwVersion < MINIMUM\_VERSION\_SUPPORTED)

    {

        dwError = (DWORD) ERROR\_REVISION\_MISMATCH;

        goto Ret;

    }

// Set the version to what we support, but don’t exceed the

// requested version

    pCardData->dwVersion =

min(pCardData->dwVersion, CURRENT\_VERSION\_SUPPORTED);

**Note:** If the version that the card minidriver returns is not suitable for the purposes of the calling application, it is the responsibility of the calling application to handle this appropriately.

After dwVersion is set in the call to **CardAcquireContext**, assume that it will not be changed by either the caller or the card minidriver while it is in the same context.

### Other Structure Version Checks

For other versioned structures and other card minidriver API methods, version handling is the same as for the CARD\_DATA structure, with one exception. If the API method is called with a structure that contains a **dwVersion** member that is set to 0, this must be treated as a **dwVersion** value of 1.

CardRSADecrypt and CardSignData have special handling for version numbers for the data structures that are passed to the functions.

# Registration and General Import Mechanisms

These operations are general in scope and are called by the card management applications and by the Base CSP/KSP. These operations manipulate data of general interest to any application on the card. This includes personalization details, the PIN, and the card file system.

## **DllMain** and Registration Mechanisms

### **DllMain**

Description:

This function provides handling for load/unload and attach/detach notifications to allow the DLL to manage its state and allocated resources. For more information, see “[DllMain Callback Function](http://msdn.microsoft.com/en-us/library/ms682583(VS.85).aspx)” on MSDN®.

BOOL WINAPI DllMain(

\_\_in HANDLE *hinstDLL*,

\_\_in DWORD  *dwReason*,

\_\_in LPVOID  *lpvReserved*

);

Input:

*hinstDLL* A handle to this DLL instance that the caller supplies.

*dwReason* The reason code that indicates why the DLL entry-point function is being called. This parameter can be one of the following values, which are defined in *Winbase.h*:

* DLL\_PROCESS\_ATTACH
* DLL\_PROCESS\_DETACH
* DLL\_THREAD\_ATTACH
* DLL\_THREAD\_DETACH

*lpvReserved* For more information, see “[DllMain Callback Function](http://msdn.microsoft.com/en-us/library/ms682583(VS.85).aspx)” on MSDN.

Output:

Return value TRUE on DLL\_PROCESS\_ATTACH if initialization of the DLL was successful; otherwise, FALSE. Value ignored at other times by caller.

**Note: A call to DllMain with DLL\_PROCESS\_DETACH can be followed by a call to** [**CardDeleteContext**](#_CardDeleteContext)**.**

### **DllRegisterServer** and **DllUnregisterServer**

**DllRegisterServer** and **DllUnregisterServer** are no longer called in Windows Vista. The registration of the card minidriver is performed through an INF-based update to the system registry.

### Registration Mechanisms

An INF-based approach should be used for the registration of a smart card minidriver. The INF file allows for the creation of the necessary registry entries as well as the copy of files from the driver package to the appropriate directories.

For an example of a smart card INF file, see “[Appendix A](#_Toc151285663)” later in this specification.

### Smart Card INF File Requirements

The smart card INF file should contain directives that create the following registry entries for each card.

[HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Cryptography\Calais\SmartCards\VENDORCARDNAME]

"80000001"="VENDOR.dll"

"ATR"=hex:01,23,45,67,89,01,23,45,67,89,01,23,45,67,89,01,23,45

"ATRMask"=hex:ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff,ff

If the minidriver supports loading under CAPI, the following line should be included in the registry file:

"Crypto Provider"="Microsoft Base Smart Card Crypto Provider"

If the minidriver supports loading under CNG, the following line should be included in the registry file:

"Smart Card Key Storage Provider"="Microsoft Smart Card Key Storage Provider"

## API Imports from the Smart Card Base CSP/KSP

When the Base CSP/KSP calls **CardAcquireContext** (see the following for more information), it passes to the card minidriver a structure that contains, among other things, function pointers for the card minidriver to use. These pointers provide support for memory management operations and cryptographic operations (such as padding and unpadding) so that all card minidrivers can use an optimized implementation that the Base CSP/KSP provides instead of duplicating these complex functions in each minidriver.

This section describes these functions by using text that is drawn from their definitions in *Cardmod.h*.

### Memory Management Functions

The following functions should be used for memory management needs in the card minidriver because they offer security enhancements that the Base CSP/KSP provides.

#### PFN\_CSP\_ALLOC

Description:

The **PFN\_CSP\_ALLOC** function is called through a pointer in the CARD\_DATA structure when the card minidriver wants to allocate a block of memory. The returned memory is aligned and size-adjusted according to the requirements of the platform.

typedef LPVOID (WINAPI \*PFN\_CSP\_ALLOC)(

\_\_in SIZE\_T *Size*

);

In:

*Size* Size, in bytes, of the memory block to be created by this operation.

Out:

Return value Nonzero on a successful allocation; otherwise, NULL.

Comments:

A return of NULL implies an out-of-memory condition and should be treated as if a call to HeapAlloc failed. Subsequently, the card minidriver should return ERROR\_NOT\_ENOUGH\_MEMORY.

#### PFN\_CSP\_REALLOC

Description:

The **PFN\_CSP\_REALLOC** function is called through a pointer in the CARD\_DATA structure when the card minidriver wants to change the size of a block of memory. The existing contents of the memory block are copied to the reallocated block to the extent that they fit.

typedef LPVOID (WINAPI \*PFN\_CSP\_REALLOC)(

\_\_in LPVOID *Address*,

\_\_in SIZE\_T *Size*

);

In:

*Address* Pointer to existing memory block.

*Size* Size, in bytes, of the memory block following this operation.

Out:

Return value Nonzero if the reallocation was successful; otherwise, NULL.

#### PFN\_CSP\_FREE

Description:

The **PFN\_CSP\_FREE** function is called through a pointer in the CARD\_DATA structure when the card minidriver wants to free a block of memory.

typedef void (WINAPI \*PFN\_CSP\_FREE)(

\_\_in LPVOID *Address*

);

In:

*Address* Pointer to existing memory block to be freed.

Out:

<none>

Comments:

There is no return value from this function.

### Cache Functions

The cache management functions are set by the Base CSP/KSP after the call to **CardAcquireContext**. The minidriver can use these functions to maintain its own internal cache that is best managed in the physical memory of the system to which the smart card is connected.

The minidriver implementation of **CardAcquireContext** should not use these function pointers for operations that are related to the cache files that the Base CSP/KSP manages. For more information about these cache files, see ”[CacheFile](#_Cache_File)” later in this specification.

#### PFN\_CSP\_CACHE\_ADD\_FILE

Description:

The **PFN\_CSP\_CACHE\_ADD\_FILE** function is called through a pointer in the CARD\_DATA structure when the card minidriver wants to add a file to the set of data that resides in the cache of the Base CSP/KSP. Generally, this would be a cache of a file that exists on the card.

typedef DWORD (WINAPI \*PFN\_CSP\_CACHE\_ADD\_FILE)(

\_\_in PVOID *pvCacheContext*,

\_\_in LPWSTR *wszTag*,

\_\_in DWORD *dwFlags*,

\_\_in\_bcount(cbData) PBYTE *pbData*,

\_\_in DWORD *cbData*

);

In:

*wszTag* The name of the file to add.

*dwFlags* Reserved—must be zero.

*pbData* Pointer to the buffer that contains the data. The buffer is allocated and freed by the card minidriver.

*cbData* Byte count of the data to which *pbData* points.

*pvCacheContext* The cache context value that is supplied by the Base CSP/KSP and taken from the CARD\_DATA structure.

Out:

Return value Zero on success; otherwise, nonzero.

Comments:

Files on the card that the card minidriver uses only internally can take advantage of the Base CSP/KSP-provided caching implementation to improve performance by avoiding redundant read/write activity to the card.

This function is used to initially create or to update the contents of the associated file cache.

#### PFN\_CSP\_CACHE\_LOOKUP\_FILE

Description:

The **PFN\_CSP\_CACHE\_LOOKUP\_FILE** function is called through a pointer in the CARD\_DATA structure when the card minidriver wants to read the contents of a file from the cache.

typedef DWORD (WINAPI \*PFN\_CSP\_CACHE\_LOOKUP\_FILE)(

\_\_in PVOID *pvCacheContext*,

\_\_in LPWSTR wszTag,

\_\_in DWORD *dwFlags*,

\_\_deref\_out\_bcount(\*pcbData) PBYTE \**ppbData*,

\_\_out PDWORD *pcbData*)

);

In:

*pvCacheContext* The cache context value that is supplied by the Base CSP/KSP.

*wszTag* The name of the file to read.

*dwFlags* Reserved—must be zero.

Out:

*ppbData* Address of a byte pointer that receives the address of the returned data buffer. The data buffer is freed by the card minidriver. The value of this pointer is not examined or used before it is overwritten by the returned buffer address.

*pcbData* Address of a DWORD that receives the byte count of the data pointed to by *ppbData*.

Return value Zero on success; otherwise, nonzero.

#### PFN\_CSP\_CACHE\_DELETE\_FILE

Description:

The **PFN\_CSP\_CACHE\_DELETE\_FILE** function is called through a pointer in the CARD\_DATA structure when the card minidriver wants to delete a file from the cache of the Base CSP/KSP.

typedef DWORD (WINAPI \*PFN\_CSP\_CACHE\_DELETE\_FILE)(

\_\_in PVOID pvCacheContext,

\_\_in LPWSTR wszTag,

\_\_in DWORD *dwFlags*

);

In:

*wszTag* The name of the file to delete.

*dwFlags* Reserved—must be zero.

*pvCacheContext* The cache context value that is supplied by the Base CSP/KSP and taken from the CARD\_DATA structure.

Out:

Return value Zero on success; otherwise, nonzero.

### Cryptographic Utilities

#### PFN\_CSP\_PAD\_DATA

Description:

The **PFN\_CSP\_PAD\_DATA** function can be optionally called by the card to perform cryptographic padding when the card cannot do that itself. For best security, padding should occur on the card. However, it is recognized that not all cards support this feature.

typedef DWORD (WINAPI \*PFN\_CSP\_PAD\_DATA)(

\_\_in PCARD\_SIGNING\_INFO *pSigningInfo*,

\_\_in DWORD *cbMaxWidth*,

\_\_out DWORD \**pcbPaddedBuffer*,

\_\_deref\_out\_bcount(\**pcbPaddedBuffer*) PBYTE \**ppbPaddedBuffer*

);

In:

*pSigningInfo* Contains buffer to pad plus needed algorithm information.

*cbMaxWidth* Block size, in bytes, of the padded buffer.

Out:

*pcbPaddedBuffer* Populated with the count of bytes in padded buffer.

*ppbPaddedBuffer* New buffer that contains original data plus padding.

Return value Zero on success; otherwise, nonzero.

Comments:

The padded buffer must be released ultimately with a call to PFN\_CSP\_FREE. The currently supported padding methods are PKCS #1 v1.1 with the Base CSP/KSP-supplied call backs and PSS padding with KSP-supplied callbacks.

#### PFN\_CSP\_UNPAD\_DATA

Description:

The **PFN\_CSP\_UNPAD\_DATA** function can be optionally called by the card to remove cryptographic padding when the card cannot do that itself. This function is optional.

To provide the best security, removal of cryptographic padding should occur on the card. However, it is recognized that not all cards support this feature.

typedef DWORD (WINAPI \*PFN\_CSP\_UNPAD\_DATA)(

\_\_in PCARD\_RSA\_DECRYPT\_INFO *pRSADecryptInfo*,

\_\_out DWORD \**pcbUnpaddedData*,

\_\_deref\_out\_bcount(\**pcbUnpaddedData*) PBYTE \**ppbUnpaddedData*

);

**In:**

*pRSADecryptInfo* Contains the buffer to unpad plus needed algorithm information.

**Out:**

*pcbUnpaddedData* Populated with count of bytes in unpadded buffer dereferenced by the *ppbUnpaddedData* parameter.

*ppbUnpaddedData* Buffer with padding removed.

**Return Value:**

Zero on success; otherwise, nonzero.

Comments:

The minidriver must free the unpadded buffer with a call to PFN\_CSP\_FREE.

The currently supported padding methods are as follows:

* PKCS#1 v1.1 with the Base CSP/KSP supplied call backs.
* OAEP padding with KSP supplied call backs.

If the **dwVersion** member of the *pRSADecryptInfo* parameter has a value that is less than CARD\_RSA\_KEY\_DECRYPT\_INFO\_CURRENT\_VERSION, this function fails with ERROR\_REVISION\_MISMATCH.

#### PFN\_CSP\_GET\_DH\_AGREEMENT

Description:

This callback function is set by the KSP before calling **CardAcquireContext**.

This function is used when one of the parameters in the CARD\_DERIVE\_KEY structure (for *pfnCardDeriveKey*) is of KDF\_NCRYPT\_SECRET\_HANDLE type. Call this function to retrieve the on-card handle. If KDF\_NCRYPT\_SECRET\_HANDLE corresponds to a non-card secret agreement, this function returns failure.

typedef DWORD (WINAPI \*PFN\_CSP\_GET\_DH\_AGREEMENT)(

\_\_in PCARD\_DATA *pCardData*,

\_\_in PVOID *hSecretAgreement*,

\_\_out BYTE \**pbSecretAgreementIndex*,

\_\_in DWORD *dwFlags*

);

*pCardData* This should be the same structure that is passed into the pfn**CardDeriveKey** function.

*hSecretAgreement* This should be the KDF\_VALUE\_SECRET parameter that is passed into pfn**CardDeriveKey** through the **pParameterList** member of the CARD\_DERIVE\_KEY structure.

*pbSecretAgreement* This is returned by this callback and is the on-card handle maintained by the card minidriver itself. This secret agreement should not be destroyed during the call to pfn**CardDeriveKey**.

*dwFlags* This is reserved and must be 0.

Ephemeral nature of secret agreements on the card:

The lifetime of *pbSecretAgreement* is limited by the length of time of the card minidriver context associated with the card or by a call to **CardDestroyDHAgreement**. All smart card DH agreements are ephemeral and are not retrievable after a card reset. Card minidrivers should not rely on calls to **CardDestroyDHAgreement**. They can keep them in volatile memory or clean them up during power-up.

## API Imports from the Smart Card Resource Manager

The Smart Card Resource Manager (SCRM) provides the mechanism to communicate with smart cards. It provides for arbitration of access and other functions to manage the availability of smart cards to applications. The interface with SCRM is accomplished by linking to *Winscard.dll*.

SCRM is the Microsoft Implementation of the “ICC Resource Manager” as described in *Interoperability Specification for ICCs and Personal Computer Systems: Part 5. ICC Resource Manager Definition*, a document that can be downloaded at <http://www.pcscworkgroup.com/specifications/>. The Microsoft implementations of the functions in this specification are prefixed with “Scard” and can be found in the platform SDK under “Security” in the “Authentication Functions” section.

For more information, see “[Smart Card Resource Manager API](http://msdn.microsoft.com/en-us/library/aa380149(VS.85).aspx)” on MSDN.

# Card Minidriver API Reference

These functions handle the creation and destruction of the communication interface and state information between the Base CSP/KSP and the card minidriver.

## Initialization and Deconstruct

### **CardAcquireContext**

Description:

The **CardAcquireContext** function is used to initialize communication between the Base CSP/KSP and the card minidriver. Upon return, *pCardData* points to a structure that is used to provide the context for most subsequent operations with the card minidriver.

DWORD WINAPI CardAcquireContext(

\_\_inout PCARD\_DATA *pCardData*,

\_\_in DWORD *dwFlags*

);

In:

*pCardData* Address of CARD\_DATA structure, initialized by the Base CSP/KSP.

*dwFlags* A set of flags that modify the behavior of this function. Currently, only one flag is allowed. For more information, see the following “Comments.”

Out:

Return value Zero on success; otherwise, nonzero. A list of expected error return values is shown later in this specification (in addition to standard behavior described in “[Error Handling](#_Error_Handling)” later in this specification).

Comments:

The Base CSP/KSP initializes the structure and writes the Base CSP/KSP state and function table information before calling **CardAcquireContext**. The card minidriver adds its function table and state information to the structure and returns a status of zero if successful or a nonzero value if an error was encountered.

The Base CSP/KSP provides seven important function exports for the card minidriver to use:

* Three for using the data caching mechanisms that the Base CSP/KSP provides.
* Three to manage memory allocation.
* One for removal of padding during decryption.

Many card minidriver functions transfer buffers between the Base CSP/KSP and the card minidriver by using a structure that contains a byte pointer. Memory is allocated by the party that generates the data and freed by the consumer of that data unless the description in this specification dictates otherwise. The card minidriver allocates and frees memory by calling the three memory management functions that the Base CSP/KSP defines.

The **dwVersion** member of the CARD\_DATA structure is taken as an input when **CardAcquireContext** is called and is the desired version structure to be returned. To eventually support the loading of existing RSA card minidrivers, older versions may have to be loaded and recognized as such.

**CardAcquireContext** must accept 0 for *dwFlags* when Secure Key Injection is not supported or requested.

**CardAcquireContext** can be called with CARD\_SECURE\_KEY\_INJECTION\_NO  
\_CARD\_MODE set in *dwFlags* if context must be initialized for secure key Injection calls in server mode.

#define CARD\_SECURE\_KEY\_INJECTION\_NO\_CARD\_MODE 0x1

This flag instructs the minidriver not to expect any card in the reader. This implies that the answer-to-Reset (ATR) fields in PCARD\_DATA is not filled. The **hSCard** and **hSCardCtx** members of the CARD\_DATA structure is NULL.

When this flag is set, the minidriver can accept only the following function calls:

* **MDImportSessionKey**
* **MDEncryptData**
* **CardGetSharedKeyHandle**
* **CardDestroyKey**
* **CardGetAlgorithmProperty**
* **CardGetKeyProperty**
* **CardSetKeyProperty**
* **CardProcessEncryptedData**
* **CardDeleteContext**

All other functions must return SCARD\_E\_INVALID\_PARAMETER when **CardAcquireContext** is called with \_SECURE\_KEY\_INJECTION\_NO\_CARD\_MODE set in *dwFlags*.

For more information about Secure Key Injection, see “[Secure Key Injection](#_Secure_Key_Injection)” later in this specification.

**Note:** The smart card minidriver must not assume a single context to manage communications with the card.

The following is the structure of the CARD\_DATA, taken from *Cardmod.h*.

#define CARD\_DATA\_VERSION\_SEVEN 7

#define CARD\_DATA\_CURRENT\_VERSION CARD\_DATA\_VERSION\_SEVEN

typedef struct \_CARD\_DATA

{

// These members must be initialized by the CSP/KSP before

// calling CardAcquireContext.

DWORD dwVersion;

PBYTE pbAtr;

DWORD cbAtr;

LPWSTR pwszCardName;

PFN\_CSP\_ALLOC pfnCspAlloc;

PFN\_CSP\_REALLOC pfnCspReAlloc;

PFN\_CSP\_FREE pfnCspFree;

PFN\_CSP\_CACHE\_ADD\_FILE pfnCspCacheAddFile;

PFN\_CSP\_CACHE\_LOOKUP\_FILE pfnCspCacheLookupFile;

PFN\_CSP\_CACHE\_DELETE\_FILE pfnCspCacheDeleteFile;

PVOID pvCacheContext;

PFN\_CSP\_PAD\_DATA pfnCspPadData;

SCARDCONTEXT hSCardCtx;

SCARDHANDLE hScard;

// pointer to vendor specific information

PVOID pvVendorSpecific;

// These members are initialized by the card module

PFN\_CARD\_DELETE\_CONTEXT pfnCardDeleteContext;

PFN\_CARD\_QUERY\_CAPABILITIES pfnCardQueryCapabilities;

PFN\_CARD\_DELETE\_CONTAINER pfnCardDeleteContainer;

PFN\_CARD\_CREATE\_CONTAINER pfnCardCreateContainer;

PFN\_CARD\_GET\_CONTAINER\_INFO pfnCardGetContainerInfo;

PFN\_CARD\_AUTHENTICATE\_PIN pfnCardAuthenticatePin;

PFN\_CARD\_GET\_CHALLENGE pfnCardGetChallenge;

PFN\_CARD\_AUTHENTICATE\_CHALLENGE pfnCardAuthenticateChallenge;

PFN\_CARD\_UNBLOCK\_PIN pfnCardUnblockPin;

PFN\_CARD\_CHANGE\_AUTHENTICATOR pfnCardChangeAuthenticator;

PFN\_CARD\_DEAUTHENTICATE pfnCardDeauthenticate;

PFN\_CARD\_CREATE\_DIRECTORY pfnCardCreateDirectory;

PFN\_CARD\_DELETE\_DIRECTORY pfnCardDeleteDirectory;

LPVOID pvUnused3;

LPVOID pvUnused4;

PFN\_CARD\_CREATE\_FILE pfnCardCreateFile;

PFN\_CARD\_READ\_FILE pfnCardReadFile;

PFN\_CARD\_WRITE\_FILE pfnCardWriteFile;

PFN\_CARD\_DELETE\_FILE pfnCardDeleteFile;

PFN\_CARD\_ENUM\_FILES pfnCardEnumFiles;

PFN\_CARD\_GET\_FILE\_INFO pfnCardGetFileInfo;

PFN\_CARD\_QUERY\_FREE\_SPACE pfnCardQueryFreeSpace;

PFN\_CARD\_QUERY\_KEY\_SIZES pfnCardQueryKeySizes;

PFN\_CARD\_SIGN\_DATA pfnCardSignData;

PFN\_CARD\_RSA\_DECRYPT pfnCardRSADecrypt;

PFN\_CARD\_CONSTRUCT\_DH\_AGREEMENT pfnCardConstructDHAgreement;

// New functions in version five.

PFN\_CARD\_DERIVE\_KEY pfnCardDeriveKey;

PFN\_CARD\_DESTROY\_DH\_AGREEMENT pfnCardDestroyDHAgreement;

PFN\_CSP\_GET\_DH\_AGREEMENT pfnCspGetDHAgreement;

// version 6 additions below here

PFN\_CARD\_GET\_CHALLENGE\_EX pfnCardGetChallengeEx;

PFN\_CARD\_AUTHENTICATE\_EX pfnCardAuthenticateEx;

PFN\_CARD\_CHANGE\_AUTHENTICATOR\_EX pfnCardChangeAuthenticatorEx;

PFN\_CARD\_DEAUTHENTICATE\_EX pfnCardDeauthenticateEx;

PFN\_CARD\_GET\_CONTAINER\_PROPERTY pfnCardGetContainerProperty;

PFN\_CARD\_SET\_CONTAINER\_PROPERTY pfnCardSetContainerProperty;

PFN\_CARD\_GET\_PROPERTY pfnCardGetProperty;

PFN\_CARD\_SET\_PROPERTY pfnCardSetProperty;

// version 7 additions below here

PFN\_CSP\_UNPAD\_DATA pfnCspUnpadData;

PFN\_MD\_IMPORT\_SESSION\_KEY pfnMDImportSessionKey;

PFN\_MD\_ENCRYPT\_DATA pfnMDEncryptData;

PFN\_CARD\_IMPORT\_SESSION\_KEY pfnCardImportSessionKey;

PFN\_CARD\_GET\_SHARED\_KEY\_HANDLE pfnCardGetSharedKeyHandle;

PFN\_CARD\_GET\_ALGORITHM\_PROPERTY pfnCardGetAlgorithmProperty;

PFN\_CARD\_GET\_KEY\_PROPERTY pfnCardGetKeyProperty;

PFN\_CARD\_SET\_KEY\_PROPERTY pfnCardSetKeyProperty;

PFN\_CARD\_DESTROY\_KEY pfnCardDestroyKey;

PFN\_CARD\_PROCESS\_ENCRYPTED\_DATA pfnCardProcessEncryptedData;

PFN\_CARD\_CREATE\_CONTAINER\_EX pfnCardCreateContainerEx;

} CARD\_DATA, \*PCARD\_DATA;

**Note:** Smart card vendors must ensure that the ATR value (referenced through the **pbAtr** member) is unique for each card to avoid the erroneous pairing of a minidriver with a smart card. Multiple cards with different card-edges cannot have the same ATR value in the same system. Also, the ATR value must be sufficiently unique for Windows to load the appropriate card-specific minidriver.

After applying the ATR mask, a card must be identifiable through a unique ATR. Windows expects the card to have the same ATR between resets. If the card changes the ATR value between warm or cold resets, the masked ATR values before and after the reset must match.

The **pfnCardDeriveKey**, **pfnCardDestroyDHAgreement**, and **pfnCspGetDHAgreement** members of the CARD\_DATA structure are described in later sections. Starting with Version 5 of this specification, the necessary modifications to the **pfnCardConstructDHAgreement** function are handled through versioning the structure that is associated with that function.

If **pfnCspPadData** is NULL, the minidriver can expect that the calling application will not invoke a decrypt functionality.

We recommend that the minidriver issue a SELECT command for the desired application identifier (AID) on the card to ensure that the correct application is selected on the card if the card has multiple applications.

Errors:

When passing an invalid **pbAtr** member to **CardAcquireContext**, the minidriver should return SCARD\_E\_UNKNOWN\_CARD.

The minidriver should return the following errors if *dwFlags* is 0:

* When passing NULL as **pbAtr** in the CARD\_DATA structure, it should return SCARD\_E\_INVALID\_PARAMETER.
* When passing NULL for the card name (**pwszCardName**) in the CARD\_DATA structure, it should return SCARD\_E\_INVALID\_PARAMETER.
* When passing invalid value (any correct length) in the **cbAtr** member in the CARD\_DATA structure, it should return SCARD\_E\_INVALID\_PARAMETER. If the value of the cbAtr field parameter is valid but is not of a known card, SCARD\_E\_UNKNOWN\_CARD must be returned.

If CARD\_SECURE\_KEY\_INJECTION\_NO\_CARD\_MODE is set in *dwFlags* and the minidriver does not support this flag, the minidriver should return SCARD\_E\_UNSUPPORTED\_FEATURE.

When passing a valid pbAtr to the wrong card minidriver, it should return SCARD\_E\_UNKNOWN\_CARD.

When passing NULL for **pfnCspAlloc**, **pfnCspReAlloc**, or **pfnCspFree** callbacks, SCARD\_E\_INVALID\_PARAMETER should be returned.

If the **hSCardCtx** and **hSCard handles** are set to NULL, the minidriver should return SCARD\_E\_INVALID\_HANDLE unless CARD\_SECURE\_KEY\_INJECTION\_NO\_CARD\_MODE is set in *dwFlags*.

In addition, the conventions for smart card minidrivers must be followed. For more information, see “[Conventions](#_Conventions)” earlier in this specification.

### **CardDeleteContext**

Description:

The **CardDeleteContext** function reverses the effect of **CardAcquireContext** and severs the communication between the Base CSP/KSP and the card minidriver. This function also performs any needed deallocations and cleanup.

DWORD WINAPI CardDeleteContext(

\_\_inout PCARD\_DATA *pCardData*

);

In:

*pCardData* Address of CARD\_DATA structure.

Out:

Return value Zero on success; otherwise, nonzero.

Comments:

**Note: CardDeleteContext can be called after DllMain was called with the DLL\_PROCESS\_DETACH parameter.**

## Card PIN Operations

The term PIN was inherited from the banking industry because of its first use on the numeric keypad of ATM machines. Some other industry documentation use the term *card holder verification*  (CHV). It is understood that the data format is not just numeric but can be anything that the user can provide given the means at his or her disposal. The value that is passed as PIN data is constrained by interoperability considerations to the ANSI single-byte character set.

Authentication of the user differs greatly from authentication of the administrator in that the user is normally not privileged to possess the administrative authentication secret. This has many implications about what kind of data can be used for this and how it is to be handled. If the administrative secret is used on the client computer to do something like unblock a user’s card with assistance from a central authority, this data must be either securely transmitted to the card without any possibility of disclosure or else be completely ephemeral so that it has no value outside the current transaction. The difficulty of arranging secure transmission to the card is why use of a PIN to authenticate the administrator is discouraged.

An authentication is valid only within a transaction, to prevent another application from hijacking an authenticated session. Deauthentication occurs automatically upon ending a transaction.

Changing the PIN *must* invalidate secure token.

### Data Structures and Enumerations

#### General Defines

We define two new data types: one for describing individual PINs that are associated with roles and PIN\_SET that is used for a bit-mask with PIN identifiers. Also, we discontinued having strings for user names and introduce role numbers that translate to PIN identifiers. We also define two flags for the PIN change operation that are explained later in this specification.

typedef DWORD PIN\_ID, \*PPIN\_ID;

typedef DWORD PIN\_SET, \*PPIN\_SET;

#define MAX\_PINS 8

#define ROLE\_EVERYONE 0

#define ROLE\_USER 1

#define ROLE\_ADMIN 2

#define PIN\_SET\_ALL\_ROLES 0xFF

#define CREATE\_PIN\_SET(PinId) (1 << PinId)

#define SET\_PIN(PinSet, PinId) PinSet |= CREATE\_PIN\_SET(PinId)

#define IS\_PIN\_SET(PinSet, PinId) (0 != (PinSet & CREATE\_PIN\_SET(PinId)))

#define CLEAR\_PIN(PinSet, PinId) PinSet &= ~CREATE\_PIN\_SET(PinId)

#define PIN\_CHANGE\_FLAG\_UNBLOCK 0x01

#define PIN\_CHANGE\_FLAG\_CHANGEPIN 0x02

To be functionally equivalent to current card minidriver cards, all cards must be provisioned with at least three roles: ROLE\_EVERYONE, ROLE\_USER, and ROLE\_ADMIN. Each role is equivalent to one PIN\_ID on the card. There is only one true administrator role for a card, but there can be multiple roles that can unblock other roles. However, only one role should control access to perform administrator-level operations such as deleting the file system, and this is ROLE\_ADMIN. Additionally, ROLE\_ADMIN must be able to unblock ROLE\_USER. There is also only one user role that gives access to the file system for a card. The additional roles 3 through 7 are optional and can be associated only with key containers.

For special considerations that can apply to read only-cards, see “[Read-Only Cards](#_Read-Only_Cards)” later in this specification.

#### SECRET\_TYPE (enumeration)

The following enumeration describes the type of PIN.

typedef enum

{

AlphaNumericPinType = 0, // Regular PIN

ExternalPinType, // External PIN

ChallengeResponsePinType, // Challenge/Response PIN

EmptyPinType // No PIN

} SECRET\_TYPE;

**Note: When encountering PIN SECRET\_TYPE EmptyPinType, Windows does not prompt for PIN nor does it call CardAuthenticatePin or CardAuthenticatePinEx. This setting is useful when an unconditional access to material on the card is desired.**

#### SECRET\_PURPOSE (enumeration)

**The following enumeration is used by the PIN\_INFO data structure to describe the purpose of the PIN for user information purpose.**

typedef enum

{

AuthenticationPin, // Authentication PIN

DigitalSignaturePin, // Digital Signature PIN

EncryptionPin, // Encryption PIN

NonRepudiationPin, // Non Repudiation PIN

AdministratorPin, // Administrator PIN

PrimaryCardPin,

UnblockOnlyPin // Unblocking other PINs

} SECRET\_PURPOSE;

Windows uses the enumeration value to display an appropriate message to the user that describes which card PIN is currently requested. The minidriver completely controls which SECRET\_TYPE to use. Figure 3 is an illustration of a PIN prompt dialog box that includes sample context strings.

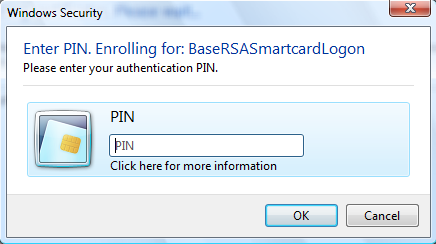


Figure 3. PIN dialog box in Windows Vista

The first string in Figure 3 (“Enter PIN. Enrolling for: BaseRSASmartcardLogon”) is provided by the calling application to provide application context. If no application context string exists, the dialog box displays a standard text.

The second string (“Please enter your authentication PIN”) is driven by SECRET\_PURPOSE in one of the following ways:

* Default context strings

By default, the Base CSP displays the following predefined strings, which are localized appropriately.

|  |  |
| --- | --- |
| AuthenticationPin | “Please enter your authentication PIN.” |
| DigitalSignaturePin | “Please enter your digital signature PIN.” |
| EncryptionPin | “Please enter your encryption PIN.” |
| NonRepudiationPin | “Please enter your non repudiation PIN.” |
| AdministratorPin | “Please enter your administrator PIN.” |
| PrimaryCardPin | “Please enter your PIN.” |
| UnblockOnlyPin | “Please enter your PIN to unblock the user PIN.” |

* Custom strings

Developers can override the default context strings by setting custom strings in the following registry values of the minidriver’s registry key (HKLM\Software\SOFTWARE\Microsoft\Cryptography\Calais\SmartCards\XYZ, where XYZ is the name of the card minidriver).

To override a predefined context string, add a registry string value to the minidriver’s registry key with the custom string. The name of the key sets which SECRET\_PURPOSE predefined context string is being overridden, with 80000100 that corresponds to the first member of SECRET\_TYPE and onward. It is not possible to override just one string, some, or all context strings.

The value of the string should follow the following format:

“LangID,xxxx;LangID,xxxxx”

**Note: Quotation marks around the custom string are not handled properly and should not be relied on to prevent parsing special characters within the string.**

**Note: Including two different custom strings for the same locale results in the first custom string being picked up.**

The third string in the dialog box (“Digital Signature PIN”) is a predefined string that is determined by the SECRET\_PURPOSE value in the PIN\_INFO data structure.

For UnblockOnlyPin, the intended purpose is to unblock the user PIN. This PIN must not be used for any other purpose.

#### PIN\_CACHE\_POLICY\_TYPE (Enumeration)

The following enumeration describes the PIN caching policy that is to be associated with this PIN.

typedef enum

{

PinCacheNormal = 0,

PinCacheTimed,

PinCacheNone,

PinCacheAlwaysPrompt

} PIN\_CACHE\_POLICY\_TYPE;

The following table describes how the Base CSP acts upon the three different cache modes.

|  |  |
| --- | --- |
| Cache mode | Description |

|  |  |
| --- | --- |
| **PinCacheNormal** | For this mode, the PIN is cached by the Base CSP per process per logon ID. The entire PIN cache structure is encrypted in memory to keep it protected. |
| **PinCacheTimed** | For this mode, the PIN is invalidated after an indicated period of time (value is given in seconds). This was implemented by recording the timestamp when the PIN is added to the cache and then verifying this timestamp versus the time when the PIN is accessed. This means that the PIN potentially lives in the cache longer than the specified timestamp, but is not used after it has expired. The PIN is encrypted in memory to keep it protected. |
| **PinCacheNone** | When the PIN cannot be cached, Base CSP never adds the PIN to the cache. When the Base CSP/KSP is called with **CryptSetProvParam** to set a PIN, the PIN is submitted to the card for verification but not cached. This means that any subsequent operations must occur before the Base CSP transaction time-out expires. |
| **PinCacheAlwaysPrompt** | Unlike **PinCacheNone**, when this cache mode is set, the Base CSP transaction time-out is not applicable. The PIN is collected from the user and then submitted to the card for verification before each call that requires authentication. Calls to CryptSetProvParam and NcryptSetProperty for setting the PIN return ERROR\_SUCCESS without verifying and caching the PIN. This implies that calls from applications that use silent contexts will fail if the call requires authentication. |

**Note: Windows logon may not work properly if a PIN is not cached. This behavior is by design. Therefore, careful consideration should be given when setting a PIN cache mode to any value other than PinCacheNormal.**

#### PIN\_CACHE\_POLICY (structure)

The PIN cache policy structure contains information that describes the PIN cache policy. It describes the PIN cache type, in addition to associated information with this PIN cache policy. An example of this associated information would be a time-out value for the PIN cache when the policy indicates **PinCacheTimed.**

#define PIN\_CACHE\_POLICY\_CURRENT\_VERSION 6

typedef struct \_PIN\_CACHE\_POLICY

{

DWORD dwVersion;

PIN\_CACHE\_POLICY\_TYPE PinCachePolicyType;

DWORD dwPinCachePolicyInfo;

} PIN\_CACHE\_POLICY, \*PPIN\_CACHE\_POLICY;

#### PIN\_INFO (structure)

The PIN object structure contains information that describes the PIN. It describes the PIN type, which PIN is allowed to unblock this target PIN, and the PIN caching policy. After a PIN information structure is obtained by the Base CSP/KSP, it should be cached in the data cache similar to how data files are cached.

#define PIN\_INFO\_CURRENT\_VERSION 6

#define      PIN\_INFO\_REQUIRE\_SECURE\_ENTRY       1

typedef struct \_PIN\_INFO

{

DWORD dwVersion;

SECRET\_TYPE PinType;

SECRET\_PURPOSE PinPurpose;

PIN\_SET dwChangePermission;

PIN\_SET dwUnblockPermission;

PIN\_CACHE\_POLICY PinCachePolicy;

DWORD dwFlags;

} PIN\_INFO, \*PPIN\_INFO;

The **dwUnblockPermission** member is a bit-mask that describes which PINs have permission to unblock the PIN. The permission is based on a bitwise ‘or’ of the specified PINs. For an unblock operation, the card minidriver should ignore any self-reference. The ROLE\_USER would have an update permission bitmask of 0x00000100. This means that it can be unblocked by ROLE\_ADMIN. ROLE\_ADMIN, which has an update permission of 0x00000000. This means that it cannot be unblocked.

The **dwChangePermission** member is an analog to **dwUnblockPermission** that describes which PINs have access to change another PIN. For example, ROLE\_USER has a change permission bitmask of 0x00000010 and ROLE\_ADMIN has 0x00000100.

The **dwFlags** member contains PIN flags. Currently, only one flag is defined: PIN\_INFO\_REQUIRE\_SECURE\_ENTRY. This flag indicates to the Base CSP/KSP whether a secure desktop is required for PIN entry.

**Note:** It is possible by using this structure to give ROLE\_EVERYONE permission to change or unblock a PIN. We do not recommend this, and no mechanism is provided in the minidriver API to allow ROLE\_EVERYONE to change or unblock a PIN.

### **CardAuthenticatePin**

Description:

The **CardAuthenticatePin** function submits a PIN value as a string to the card to establish the user’s identity and to satisfy access conditions for an operation to be undertaken on the user’s behalf. Submission of a PIN to the card may involve some processing by the card minidriver to render the PIN information to a card-specific form.

DWORD WINAPI CardAuthenticatePin(

\_\_in PCARD\_DATA *pCardData*,

\_\_in LPWSTR *pwszUserId*,

\_\_in\_bcount(*cbPin*) PBYTE *pbPin*,

\_\_in DWORD *cbPin*,

\_\_out\_opt PDWORD *pcAttemptsRemaining*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pwszUserId* String that specifies the card principal that is associated with the PIN.

*pbPin* Pointer to a buffer that contains the PIN information.

*cbPin* Byte count of the data in the PIN information buffer.

*pdwcAttemptsRemaining* Count of times that an incorrect PIN can be presented to the card before the card is locked. The card minidriver tests this value for NULL before attempting to use it.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

A card principal is the user type (or role) that is associated with the authentication process that the smart card uses. For more information about the various types of card principals, see ”[Known Principals](#_Known_Principals)” later in this specification.

The allowed values for the *pwszUserId* are wszCARD\_USER\_USER or wszCARD\_USER\_ADMINISTRATOR as defined in *Cardmod.h*.

For any other *pwszUserId* value, **CardAuthenticatePin** should return SCARD\_E\_INVALID\_PARAMETER.

This function returns SCARD\_E\_INVALID\_PARAMETER for external PINs or empty PINs. This function is deprecated by **CardAuthenticateEx**, which is the recommended function for all PIN types.

**Note:** Challenge/response is the preferred mechanism for administrator authentication to the card and the only authentication mode that Windows uses to authenticate an administrator.

If an incorrect PIN is presented, this function returns SCARD\_W\_WRONG\_CHV. If the *pdwcAttemptsRemaining* parameter is non-NULL, it returns the number of remaining attempts.

On the last allowed attempt, the function returns SCARD\_W\_WRONG\_CHV and the *pdwcAttemptsRemaining* parameter returns zero. For all attempts beyond the allowed number, the function returns SCARD\_W\_CHV\_BLOCKED and the *pdwcAttemptsRemaining* parameter returns zero.

Implementations that do not support returning the count of remaining authentication attempts should return -1 for this value if *pdwcAttemptsRemaining* is non-NULL.

If the *pbPin* is NULL, the call fails with the SCARD\_E\_INVALID\_PARAMETER error code.

### **CardGetChallenge**

Description:

A card principal can be authenticated by using either a PIN or a challenge/response protocol in which the card generates a block of challenge data by using its administrative key. The authenticating caller must compute the response to the challenge by using shared knowledge of that key and submit the response back to the card. If the response is correct, the principal is authenticated to the card.

DWORD WINAPI CardGetChallenge(

\_\_in PCARD\_DATA *pCardData*,

\_\_deref\_out\_bcount(\**pcbChallengeData*) PBYTE \**ppbChallengeData*,

\_\_out PDWORD *pcbChallengeData*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*ppbChallengeData* Pointer to byte pointer to receive the challenge data from the card.

*pcbChallengeData* Byte count of the challenge data.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

A card principal is the user type (or role) that is associated with the authentication process that the smart card uses. For more information about the various types of card principals, see [”Known Principals](#_Known_Principals)” later in this specification.

This challenge/response method is generally used to establish the context for privileged operations such as unblocking a user’s PIN. For security reasons, we recommend that implementers of card minidrivers produce a design in which the challenge and response values are not invariant so that these values cannot be replayed.

The caller can decide not to use the challenge value. It is significant only if an authentication tries to use it. It is discarded if the next command to the card is not an authentication attempt that uses it (for more information, see the following section, “[**CardAuthenticateChallenge**](#_CardAuthenticateChallenge)**”**). The smart card’s internal operating system should be designed to enforce this behavior.

The challenge buffer is allocated by the card minidriver and freed by the caller by using PFN\_CSP\_FREE.

Errors:

The conventions that are specified in “[Error Handling](#_Error_Handling)” earlier in this specification should be followed.

### **CardAuthenticateChallenge**

Description:

The **CardAuthenticateChallenge** function performs authentication of a card principal by using a challenge/response protocol. The caller of this function must have previously called **CardGetChallenge** to retrieve the challenge data from the card and computed the correct response data to submit with this call.

DWORD WINAPI CardAuthenticateChallenge(

\_\_in PCARD\_DATA *pCardData*,

\_\_in\_bcount(cbResponseData) *pbResponseData*,

\_\_in DWORD *cbResponseData*,

\_\_out\_opt PDWORD *pcAttemptsRemaining*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pbResponseData* Pointer to a buffer that contains the response data that corresponds to the challenge.

*cbResponseData* Byte count of the response data.

*pcAttemptsRemaining* Count of times that authentications to the card can fail before the card is locked. The card minidriver tests this pointer for NULL before attempting to use it.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

A card principal is the user type (or role) that is associated with the authentication process that the smart card uses. For more information about the various types of card principals, see [”Known Principals](#_Known_Principals)” later in this specification.

This challenge/response method is generally used to establish the context for privileged operations such as unblocking a user’s PIN.

If challenge/response authentication fails, the card minidriver returns SCARD\_W\_WRONG\_CHV. In addition, if the *pdwcAttemptsRemaining* parameter is non-NULL, it returns the number of remaining attempts. On the last allowed attempt, the function returns SCARD\_W\_WRONG\_CHV and the pdwcAttemptsRemaining parameter returns zero. For all attempts beyond the allowed number, the function returns SCARD\_W\_CHV\_BLOCKED and the *pdwcAttemptsRemaining* parameter returns zero.

If **CardGetChallenge** was not called before calling **CardAuthenticateChallenge**, the count of remaining authentication attempts is not decremented.

Implementations that do not support returning the count of remaining authentication attempts should always return -1 for this value if *pdwcAttemptsRemaining* is non-NULL, even when the card is blocked.

The minidriver must use the following general rules:

* Failed authentication attempts should always leave the card in a deauthenticated state.
* Successful authentication attempts should leave the card authenticated to the authenticated principal.

### **CardDeauthenticate**

Description:

The **CardDeauthenticate** function is an *optional* export that should be provided if it is possible within the card minidriver to efficiently reverse the effect of authenticating a user or administrator without resetting the card. If this function is not implemented, the card minidriver should put NULL in the CARD\_DATA structure pointer for this function.

The Base CSP/KSP tests this pointer for NULL value before calling it. If it is found NULL, the Base CSP/KSP deauthenticates a user by resetting the card. Because a card reset is a time-consuming operation, the card minidriver should implement this function if it can be done.

DWORD WINAPI CardDeauthenticate(

\_\_in PCARD\_DATA *pCardData*,

\_\_in LPWSTR *pwszUserId*,

\_\_in DWORD *dwFlags*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pwszUserId* String that indicates the card principal to be deauthenticated.

*dwFlags* Reserved—must be zero.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

A card principal is the user type (or role) that is associated with the authentication process that the smart card uses. For more information about the various types of card principals, see [”Known Principals](#_Known_Principals)” later in this specification.

If the card minidriver returns a nonzero value from this function, the Base CSP/KSP resets the card.

After successfully de-authenticating the user, the minidriver should clear any cache content that it may have created during the duration of the authenticated session using the call back caching functions set by Smart card Base CSP/SC KSP.

### **CardAuthenticateEx**

Description:

The **CardAuthenticateEx** function handles PIN authentication operations to the card.

This function replaces the **CardAuthenticate** function of earlier versions of these specifications and adds support for the following PIN types:

* External PINs, which are PINs that are accessed from a device that is connected to the computer.
* Challenge/response PINs.
* Secure PIN channels.
* Session PINs.

DWORD WINAPI CardAuthenticateEx(

\_\_in PCARD\_DATA *pCardData*,

\_\_in PIN\_ID *PinId*,

\_\_in DWORD *dwFlags*,

\_\_in\_bcount(*cbPinData*) PBYTE *pbPinData*,

\_\_in DWORD *cbPinData*,

\_\_deref\_opt\_out\_bcount(\**pcbSessionPin*) PBYTE \**ppbSessionPin*,

\_\_out\_opt PDWORD *pcbSessionPin*,

\_\_out\_opt PDWORD *pcAttemptsRemaining*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*PinId* PIN identifier to be authenticated, such as ROLE\_USER.

*dwFlags* Flags, which are discussed later in “Comments.”

*pbPinData* Pointer to a buffer that contains the PIN information.

*cbPinData* Byte count of the data in the PIN information buffer.

*ppbSessionPin* Optional pointer to a byte buffer to receive a session PIN.

*pcbSessionPin* Optional pointer to a byte count of the session PIN data.

*pcAttemptsRemaining* Count of times that an incorrect PIN may be presented to the card before the PIN is locked. The card minidriver tests this value for NULL before attempting to use it.

Output:

Return value Zero on success: otherwise, nonzero.

Comments:

**Expected card behavior:**

On success, the user can perform any action that requires *PinId* to be authenticated. This state persists until one of the following occurs:

* Either **CardDeauthenticate** or **CardDeauthenticateEx** is called.
* The card is reset through the Winscard API
* The card loses power.

**Note:** This does not apply if CARD\_AUTHENTICATE\_GENERATE\_SESSION\_PIN is specified in *dwFlags*.

The allowed values for *PinId* are ROLE\_USER, ROLE\_ADMIN or 3 through 7. For any other *PinId* value, **CardAuthenticatePinEx** should return SCARD\_E\_INVALID\_PARAMETER.

For UnblockOnly PINS, it is acceptable for **CardAuthenticateEx** to return SCARD\_E\_INVALID\_PARAMETER or SCARD\_E\_UNSUPPORTED\_FEATURE. An UnblockOnly PIN is specified by setting the **SECRET\_PURPOSE** member of *pbPinData* to **UnblockOnlyPin**.

**Failed authentication attempts:**

If an incorrect PIN is presented, this function returns SCARD\_W\_WRONG\_CHV. If the *pcAttemptsRemaining* parameter is non-NULL, this function returns the number of remaining attempts. On the last allowed attempt, the function returns SCARD\_W\_WRONG\_CHV and the *pcAttemptsRemaining* parameter returns zero. For all attempts beyond the allowed number, the function returns SCARD\_W\_CHV\_BLOCKED and the *pcAttemptsRemaining* parameter returns zero.

If the minidriver does not support returning the count of remaining authentication attempts, it should return -1 for the *pcAttemptsRemaining* value if *pcAttemptsRemaining* is non-NULL.

Presenting an incorrect PIN for a particular role (as specified through *PinId*) should result in that role being deauthenticated on the card, but should not affect other roles that are already authenticated to the card.

If the pointer to *pbPinData* is NULL, the call fails with the error code SCARD\_E\_INVALID\_PARAMETER.

**Session PIN:**

A session PIN is defined as a temporary PIN. This PIN type is generated by the card and expires upon termination of the session.

Cards that support session PINs can return the generated session PIN. If the *ppbSessionPin* parameter is non-NULL and the card can generate a session PIN, the call should allocate *ppbSessionPin* to hold the session PIN. In this situation, *pcbSessionPin* should contain the length of the session PIN. Windows caches the session PIN (according to the PIN caching policy in the PIN\_CACHE\_POLICY structure in *pbPinData*) and presents it to the card for the next call to **CardAuthenticateEx**.

After a successful generation of a session, Windows calls **CardAuthenticateEx** and sets CARD\_AUTHENTICATE\_SESSION\_PIN in *dwFlags*. In this situation, the actual PIN is not passed and the minidriver must use the session PIN in *pbPinData* and *cbPinData* to authenticate the card. If the CARD\_AUTHENTICATE\_GENERATE\_SESSION\_PIN flag is set in *dwFlags* and either *ppbSessionPin* or *pcbSessionPin* is set to NULL, the call must fail with SCARD\_E\_INVALID\_PARAMETER.

If this session PIN is invalid, SCARD\_WRONG\_CHV should be returned, *pcAttemptsRemaining* is not expected to hold valid data, and the retry count of the original PIN should not be decremented. The retry counter for the session PIN should be decremented.

**External PIN:**

An external PIN (ExternalPinType returned in PIN\_INFO) is defined as a PIN that is stored on a device that is connected to the computer. For example, this might be a BIO match-on-card PIN. In this situation, Windows does not prompt the user for a PIN but calls **CardAuthenticateEx** with an NULL value for *pbPinData* for the PIN.

An external PIN is specified by setting the **SECRET\_PURPOSE** member of *PIN\_INFO* structure to **ExternalPinType**.

The minidriver must always return a session PIN when processing an external PIN. This allows applications to perform PIN caching.

In this call, a minidriver can display its own UI windows, as long as CARD\_PIN\_SILENT\_CONTEXT was not set in *dwFlags*.

If CARD\_PIN\_SILENT\_CONTEXT is set and the PIN information is passed in through *pbPinData*, the minidriver must perform pin verification silently. If CARD\_PIN\_SILENT\_CONTEXT is set and no PIN information is passed in *pbPinData*, the minidriver must return SCARD\_E\_INVALID\_PARAMETER.

If the PIN is a biometric PIN, **CardAuthenticateEx** should expect a session pin as input in the *pbPinData* parameter. This session pin may be generated by a prior call to **CardAuthenticateEx** with CARD\_AUTHENTICATE\_GENERATE\_SESSION\_PIN set in *dwFlags*.

The handle of the parent window and a context string are passed to the minidriver before calling **CardAuthenticateEx** by calling **CardSetProperty** with the following parameters:

* **CardSetProperty** (CP\_PARENT\_WINDOW, YYYYY) where YYYYY is a HWND.
* **CardSetProperty** (CP\_PIN\_CONTEXT\_STRING, YYYYY) where YYYYY is a string.

**Important:** Displaying a UI when CARD\_PIN\_SILENT\_CONTEXT was set results in operating system instability. The minidriver must always provide an option to close the UI windows to let users use alternative credential providers for authentication.

**Secure PIN channel:**

A secure PIN channel is enabled if one of the following conditions are true:

* The Common Criteria Group Policy is enabled.
* The card requests a secure PIN channel. For more information, see “[**CardGetProperty**](#_CardGetPropertyCardGetProperty),” and “[Card and Container Properties](#_Card_and_Container)” later in this specification.
* The **dwFlags** member of the PIN\_INFO structure (which is pointed to by *pbPinData*) contains PIN\_INFO\_REQUIRE\_SECURE\_ENTRY.

In secure PIN channel mode, the PIN prompt is presented to the user on a secure desktop after the user presses the CTRL+ ALT+DEL keyboard shortcut.

When in secure PIN channel mode, the operating system calls **CardAuthenticateEx** from a trusted process and sets CARD\_AUTHENTICATE\_GENERATE\_SESSION\_PIN in *dwFlags*. The PIN is specified in clear text. It is expected that the minidriver establishes a secure connection with the card and return a session PIN to the system. The system then passes the session PIN to the nonsecure context process for authentication to the card.

**Note:** It is imperative that the clear text PIN is handled securely when a CARD\_AUTHENTICATE\_GENERATE\_SESSION\_PIN flag is set. This PIN must never be transmitted to the card.

If either *ppbSessionPin* or *pcbSessionPin* are NULL, the function should return SCARD\_E\_INVALID\_PARAMETER.

### **CardGetChallengeEx**

Description:

Besides authentication by using a PIN, a card principal can be authenticated by using a challenge/response protocol in which the card generates a block of challenge data.

The authenticating caller must compute the response to the challenge by using shared knowledge of a key and submit the response back to the card by calling **CardGetChallengeEx**. If the response is correct, the principal is authenticated to the card.

DWORD WINAPI CardGetChallengeEx(

\_\_in PCARD\_DATA *pCardData*,

\_\_in PIN\_ID *PinId*,

\_\_deref\_out\_bcount(\**pcbChallengeData*) PBYTE \**ppbChallengeData*,

\_\_out PDWORD *pcbChallengeData*,

\_\_in DWORD *dwFlags*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*PinId* PIN identifier to be authenticated.

*ppbChallengeData* Pointer to a byte pointer to receive the challenge data from the card.

*pcbChallengeData* Byte count of the challenge data.

*dwFlags* Flags, reserved for future use. Must be 0.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

A card principal is the user type (or role) that is associated with the authentication process that the smart card uses. For more information about the various types of card principals, see [”Known Principals](#_Known_Principals)” later in this specification.

This challenge/response method is generally used to establish the context for privileged operations such as unblocking a user’s PIN. For security reasons, implementers of card minidrivers are advised to produce a design in which the challenge and response values are not invariant so that these values cannot be replayed.

The caller may choose to not use the challenge value. It is significant only if an authentication is attempted by using it. It is discarded if the next command to the card is not an authentication attempt to use it. For more information, see “[**CardAuthenticateChallenge**](#_CardDeauthenticate)” earlier in this specification. The smart card’s internal operating system should be designed to enforce this behavior.

The challenge buffer is allocated by the card minidriver and freed by the caller by using PFN\_CSP\_FREE.

Errors:

If *PinId* is not set to ChallengeResponsePinType, the function should return SCARD\_E\_INVALID\_PARAMETER.

The conventions in “[Error Handling](#_Error_Handling)” earlier in this specification should be followed.

### **CardDeauthenticateEx**

Description:

The **CardDeauthenticateEx** function must always be provided. If it is not possible within the card minidriver to efficiently reverse the effect of an authentication operation without resetting the card, the call must return SCARD\_E\_UNSUPPORTED\_FEATURE. In this situation, the Base CSP/KSP performs deauthentication by resetting the card. Because a card reset is a time-consuming operation, the card minidriver must implement this function if it can be done.

DWORD WINAPI CardDeauthenticateEx(

\_\_in PCARD\_DATA *pCardData*,

\_\_in PIN\_SET *PinId*,

\_\_in DWORD *dwFlags*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*PinId* Set of PINs to be deauthenticated.

*dwFlags* Reserved—must be zero.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

The *PinId* parameter can contain multiple PINs for deauthentication. For example, a *PinId* that contains the value 0x06 means to deauthenticate ROLE\_USER and ROLE\_ADMIN. A value of 0xFF means to deauthenticate all PINs that are currently authenticated. If the ROLE\_EVERYONE bit is set in the *PinId* parameter, it should be ignored.

If the card minidriver returns a nonzero value from this function, the Base CSP/KSP resets the card.

If the function returns 0 (success), all specified PINs have been deauthenticated.

After successfully de-authenticating the user, the minidriver should clear any cache content that it may have created during the duration of the authenticated session using the call back caching functions set by Smart card Base CSP/SC KSP.

### **CardChangeAuthenticatorEx**

Description:

This function changes the authenticator for the affected card principal. It can be used to change a PIN or unblock a PIN. The usages are distinguished by use of a flag value.

DWORD WINAPI CardChangeAuthenticatorEx(

\_\_in PCARD\_DATA *pCardData*,

\_\_in DWORD dwFlags,

\_\_in PIN\_ID *dwAuthenticatingPinId*,

\_\_in\_bcount(*cbAuthenticatingPinData*)   
 PBYTE *pbAuthenticatingPinData*,

\_\_in DWORD *cbAuthenticatingPinData*,

\_\_in PIN\_ID *dwTargetPinId*,

\_\_in\_bcount(*cbTargetData*) PBYTE *pbTargetData*,

\_\_in DWORD *cbTargetData*,

\_\_in DWORD *cRetryCount*,

\_\_out\_opt PDWORD *pcAttemptsRemaining*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*dwFlags* Indication of whether this is a PIN change or unblock operation.

*dwAuthenticatingPinId* PIN identifier to be authenticated.

*pbAuthenticatingPinData* Pointer to a byte buffer that contains PIN data.

*cbAuthenticatingPinData* Byte count of the PIN data.

dwTargetPinId PIN identifier to be updated.

*pbTargetData* Pointer to a byte buffer that contains the new PIN.

*cbTargetData* Byte count of the new PIN data.

*cRetrycount* The count of times that a wrong PIN does not result in a blocked card.

*pcAttemptsRemaining* Pointer to the count of remaining times that a wrong PIN does not result in a blocked card.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

This function is used for all situations in which the authenticator is to be set.

A card principal is the user type (or role) that is associated with the authentication process that the smart card uses. For more information about the various types of card principals, see “[Known Principals](#_Known_Principals)” see later in this specification.

The allowed values for the *PinId* parameter are ROLE\_USER, ROLE\_ADMIN or 3 through 7. For any other *PinId* value, this function should return SCARD\_E\_INVALID\_PARAMETER.

The interpretation of the Authenticator buffers is dictated by the value of *dwFlags*. Currently supported values are PIN\_CHANGE\_FLAG\_UNBLOCK and PIN\_CHANGE\_FLAG\_CHANGEPIN. If *dwFlags* indicates PIN\_CHANGE\_FLAG\_UNBLOCK, the card minidriver performs an unblock operation. In this scenario, *dwAuthenticatingPinId* indicates the authenticator being verified and *dwTargetPinId* indicates the PIN identifier for the authenticator to be changed (the value should be different in the unblock scenario). If the authenticating PIN is a challenge response PIN, the caller must have previously obtained a challenge value from the card through CardGetChallenge.

For a description of the usage of *pdwcAttemptsRemaining*, see the comments for **CardAuthenticatePin** earlier in this specification.

If zero is passed for *cRetryCount*, the PIN retry maximum value is unchanged. Implementations that do not support setting the retry count should return an invalid parameter error if a retry value other than 0 is passed.

Implementations that enforce policies about the authenticator (such as PIN policies) should return SCARD\_E\_INVALID\_PARAMETER if changing the authenticator or the form of the new authenticator do not comply with policy.

When **CardChangeAuthenticatorEx** is used to change a PIN, successful completion should leave the card in an authenticated state. If **CardChangeAuthenticatorEx** is used to unblock a PIN, the successful completion should leave the card in a deauthenticated state for both the unblocked PIN and the authenticating PIN.

### **CardUnblockPin**

Description:

The **CardUnblockPin** function is used to unblock a card that has become blocked by too many incorrect PIN entry attempts. The unblock function is atomic in that authentication and unblocking the card must occur as a single operation. Therefore, authentication information and the new user PIN must be presented when the call is made.

DWORD WINAPI CardUnblockPin(

\_\_in PCARD\_DATA *pCardData*,

\_\_in LPWSTR *pwszUserId*,

\_\_in\_bcount(*cbAuthenticationData*) PBYTE *pbAuthenticationData*,

\_\_in DWORD *cbAuthenticationData*,

\_\_in\_bcount(*cbNewPinData*) PBYTE *pbNewPinData*,

\_\_in DWORD *cbNewPinData*,

\_\_in DWORD *cRetryCount*,

\_\_in DWORD *dwFlags*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pwszUserId* String that specifies the card principal that is associated with the PIN.

*pbAuthenticationData* Pointer to the **CardGetChallengeEx** response data or the PIN unblock key (PUK) value for cards that support only PUK for unblocking.

*cbAuthenticationData* Byte count of the authentication data.

*pbNewPinData* Pointer to a buffer that contains the new PIN to be set.

*cbNewPinData* Byte count of the data to which *pbNewPinData* points.

*cRetryCount* Count of times that a wrong PIN does not result in a blocked card.

*dwFlags* CARD\_AUTHENTICATE\_PIN\_CHALLENGE\_RESPONSE.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

A card principal is the user type (or role) that is associated with the authentication process that the smart card uses. For more information about the various types of card principals, see “[Known Principals](#_Known_Principals)” later in this specification.

The authentication data for the operation is a response that corresponds to the challenge that is acquired by a call to CardGetChallenge. This is distinguished by the CARD\_ AUTHENTICATE \_PIN\_CHALLENGE\_RESPONSE flag value that identifies the buffer because that contains a response to a challenge.

For example, a typical scenario is unblocking the user PIN by using administrator challenge/response.

In addition, for general conventions and guidelines for using PIN and challenge/response authenticators, see [Card PIN Operations](#_Card_PIN_Operations) earlier in this specification. For administrators, challenge/response support is mandatory. For users, challenge/response support is not supported.

If zero is passed for *cRetryCount*, the PIN retry maximum value is unchanged. Implementations that do not support setting the retry count should return an invalid parameter error if a retry value other than zero is passed. In that situation, the challenge should be considered invalid and a fresh one be requested.

A successful call to **CardUnblockPin** should leave the card in a deauthenticated state.

Errors:

If **CardUnblockPin** is called with a NULL value for *pbAuthenticationData*, the expected error code is SCARD\_E\_INVALID\_PARAMETER.

### **CardChangeAuthenticator**

Description:

This function changes the authenticator for the affected card principal. It can be used to change a user’s PIN or to change the challenge/response key. The two usages are distinguished by use of a flag value.

DWORD WINAPI CardChangeAuthenticator(

\_\_in PCARD\_DATA *pCardData*,

\_\_in LPWSTR *pwszUserId*,

\_\_in\_bcount(*cbCurrentAuthenticator*)   
 PBYTE *pbCurrentAuthenticator*,

\_\_in DWORD *cbCurrentAuthenticator*,

\_\_in\_bcount(*cbNewAuthenticator*) PBYTE *pbNewAuthenticator*,

\_\_in DWORD *cbNewAuthenticator*,

\_\_in DWORD *cRetryCount*,

\_\_in DWORD *dwFlags*,

\_\_out\_opt PDWORD *pcAttemptsRemaining*);

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pwszUserId* String that specifies the card principal that is associated with the PIN.

*pbCurrentAuthenticator* Pointer to a buffer that contains the current PIN information or a response to a previously issued challenge. For more information, see the following “Comments.”

*dwcbCurrentAuthenticator* Byte count of the current PIN/response.

*pbNewAuthenticator* Pointer to a buffer that contains the new PIN/key to be set.

*cbNewAuthenticator* Byte count of the new PIN/key.

*cRetryCount* Count of the times that a wrong PIN does not result in a blocked card.

*dwFlags* For more information, see the following “Comments.”

*pcAttemptsRemaining* Count of the remaining times that a wrong PIN does not result in a blocked card.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

A card principal is the user type (or role) that is associated with the authentication process that the smart card uses. For more information about the various types of card principals, see “[Known Principals](#_Known_Principals)” later in this specification .

This function is used when the authenticator is to be set and the value on the card is known. Generally, new cards are set up with common values. Therefore, this function is the logical choice to use when individualizing a card.

The allowed values for *pwszUserId* are **wszCARD\_USER\_USER** or **wszCARD\_USER\_ADMIN** as defined in *Cardmod.h*.

The interpretation of the Authenticator buffers is dictated by the value of *dwFlags*. Currently, the only supported values are **CARD\_AUTHENTICATE\_PIN\_PIN** and **CARD\_AUTHENTICATE\_PIN\_CHALLENGE\_RESPONSE**. In the latter situation, the caller must have previously obtained a challenge value from the card through **CardGetChallenge** and this response is placed in the **pbCurrentAuthenticator** member of the context information to which *pCardData* points.

For a description of the usage of *pdwcAttemptsRemaining*, see “Comments” under “[**CardAuthenticatePin**](#_CardAuthenticatePin)**”** earlier in this specification.

If zero is passed for *cRetryCount*, the PIN retry maximum value is unchanged. Implementations that do not support setting the retry count should return an invalid parameter error if a retry value other than zero is passed.

Implementations that enforce policies about the authenticator (such as PIN policies) should return SCARD\_E\_INVALID\_PARAMETER if changing the authenticator or the form of the new authenticator do not comply with policy.

A successful call to **CardChangeAuthenticator** should leave the card in an authenticated state.

## Public Data Operations

Data storage is organized by directories on the card. There are a few globally significant well-known files (cache file, card unique ID, and the application map). However, the remaining files are organized by reference to their application-associated directory.

Directory and file names must be composed of ANSI characters (8 bit), excluding characters that the Windows file and directory-naming conventions do not allow (namely: “, \*, /, :, <, >, ?, \, |, and character codes 1 through 31). Also, they must be 8 or fewer characters in length, excluding the terminating null.

Setting up an application on the card involves the following steps:

1. Creating the application’s storage subdirectory.
2. Creating a DWORD entry for the application in the cache file.
3. Adding the new application to the application directory.

These steps are performed above the card minidriver so that the card minidriver must expose only primitive functions that are required to create directories, create files, and write files.

Note that all file operations are atomic and self-contained. There is no concept of a handle being acquired and being used for successive operations. When a file is written by using **CardWriteFile**, for example, it is opened or created, the data is written, and the file is closed, all being implicit operations in the call.

### **CardCreateDirectory**

Description:

This function creates a subdirectory from the root in the file system of the card and applies the provided access condition. Directories are generally created for segregating the files that belong to a single application on the card. As an example, the files that belong to the Microsoft cryptographic application are in the “mscp” directory.

DWORD WINAPI CardCreateDirectory(

\_\_in PCARD\_DATA *pCardData*,

\_\_in LPSTR *pszDirectoryName*,

\_\_in CARD\_DIRECTORY\_ACCESS\_CONDITION *AccessCondition*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pszDirectory* Name of the directory.

*AccessCondition* Access control permissions to be applied to the directory.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

Directory names must be 8 or fewer ANSI characters.

Currently-defined directory access control permissions are taken from the CARD\_DIRECTORY\_ACCESS\_CONDITION from *Cardmod.h*. These access control permissions are:

* **InvalidAc**
* **UserCreateDeleteDirAc**
* **AdminCreateDeleteDirAc**

**Note:** The **AdminCreateDeleteDirAc** access control list (ACL) is optional. It may be removed in future revisions of the minidriver specification.

For more information about these access control permission values, see “[File System Requirements](#_File_System_Requirements)” later in this specification.

Errors:

The function fails if the subdirectory already exists (ERROR\_FILE\_EXISTS) or insufficient space exists to create the new directory on the card (SCARD\_E\_NO\_MEMORY).

**Note**The amount of free space can be retrieved by using **CardQueryFreeSpace**.

If calling **CardCreateDirectory** with a NULL *pszDirectoryName*, SCARD\_E\_INVALID\_PARAMETER should be returned.

If the *pszDirectoryName* directory already exists or if there is no such directory but there is a file that is named the same, ERROR\_FILE\_EXISTS should be returned.

If calling **CardCreateDirectory** without a previous card authentication, the function is expected to fail with an SCARD\_W\_SECURITY\_VIOLATION error.

If calling **CardCreateDirectory** with invalid access conditions, the function is expected to fail with an SCARD\_E\_INVALID\_PARAMETER error.

If the name that was specified by *pszDirectoryName* is longer than the maximum length that is defined for file/directory names, SCARD\_E\_INVALID\_PARAMETER must be returned.

### **CardDeleteDirectory**

Description:

This function deletes a directory from the card. This operation fails if it violates permissions on the directory or if the directory is not empty.

DWORD WINAPI CardDeleteDirectory(

\_\_in CARD\_DATA \**pCardData*,

\_\_in LPSTR *pszDirectoryName*,

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pszDirectoryName* Name of the directory.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

Returns invalid argument error if NULL is passed as the directory name.

If **CardDeleteDirectory** is called without previous authentication, the call should return SCARD\_W\_SECURITY\_VIOLATION.

If **CardDeleteDirectory** is called for a directory that is not empty (because it contains at least one file), ERROR\_DIR\_NOT\_EMPTY should be returned.

If **CardDeleteDirectory** is called on a nonexistent directory, an SCARD\_E\_DIR\_NOT\_FOUND error code must be returned.

If the name that *pszDirectoryName* specified is longer than the maximum length that is defined for file/directory names, SCARD\_E\_INVALID\_PARAMETER must be returned.

### **CardReadFile**

Description:

The **CardReadFile** function reads the entire file at the specified location into the user-supplied buffer.

DWORD WINAPI CardReadFile(

\_\_in PCARD\_DATA *pCardData*,

\_\_in\_opt LPSTR *pszDirectoryName*,

\_\_in LPSTR *pszFileName*,

\_\_in DWORD *dwFlags*,

\_\_deref\_out\_bcount\_opt(\*pcbData) PBYTE \*p*pbData*,

\_\_out PDWORD pcbData

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pszDirectoryName* Name of the directory that contains the file; NULL for root.

*pszFileName* File name for the file of interest.

*dwFlags* Reserved—must be zero.

*ppbData* Address of a byte pointer to receive the address of a buffer that contains the file contents.

*pcbData* Address of a DWORD to receive the byte count of the file contents. On input, the contents of the pointer’s destination should be ignored.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

The buffer that contains the returned data is allocated by the card minidriver and freed by the Base CSP/KSP.

For more information, see the comments regarding file sizes in “[**CardWriteFile**](#_CardWriteFile)” later in this specification.

If *pszFileName* specifies a nonexistent file, **CardReadFile** should fail with SCARD\_E\_FILE\_NOT\_FOUND.

The BaseCSP/SCKSP will cache SCARD\_E\_FILE\_NOT\_FOUND return values from **CardReadFile** to improve performance. Therefore mini-dirvers should not use SCARD\_E\_FILE\_NOT\_FOUND as an umbrella when returning errors from **CardReadFile**.

If **CardReadFile** is called on a nonexistent directory, an SCARD\_E\_DIR\_NOT\_FOUND error code must be returned.

If the name that was specified by *pszFileName* or *pszDirectoryName* is longer than the maximum length for file/directory names, SCARD\_E\_INVALID\_PARAMETER must be returned.

When this function is called for the card identifier (*cardid*) file or cache (*cardcf*) file, the cache functions in the CARD\_DATA structure should not be called. Otherwise, the attempt results in an endless loop. For more information about the cache functions, see “[Cache Functions](#_Cache_Functions)” earlier in this specification.

### **CardCreateFile**

Description:

The **CardCreateFile** function creates a file on the card with a specified name and access permission. This function cannot be used to create directories. If the directory that is named by *pszDirectoryName* does not exist, the function fails with SCARD\_E\_DIR\_NOT\_FOUND.

DWORD WINAPI CardCreateFile(

\_\_in PCARD\_DATA *pCardData*,

\_\_in\_opt LPSTR *pszDirectoryName*,

\_\_in LPSTR *pszFileName*,

\_\_in DWORD *cbInitialCreationSize*,

\_\_in CARD\_FILE\_ACCESS\_CONDITION *AccessCondition*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pszDirectoryName* Name of the directory that is to contain the file; NULL for root.

*pszFileName* Logical File Name for the file to be created.

*cbInitialCreationSize* Initial size of the file at creation time.

*AccessCondition* Access control permissions to be applied to the file.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

The initial size parameter can be used to avoid the situation in which a later file write fails for lack of space after the file was successfully created. Rules for file name are as defined in ”[File System Requirements](#_Toc232242982)” later in this specification.

If *pszFileName* is NULL or an empty string, an SCARD\_E\_INVALID\_PARAMETER error must be returned. If *cbInitialCreationSize* is greater than the free space on the card, an SCARD\_E\_INVALID\_PARAMETER error must be returned.

Currently-defined file access control permissions are taken from the CARD\_FILE\_ACCESS\_CONDITION from *Cardmod.h*. The following are the file access control permissions:

* InvalidAc
* EveryoneReadUserWriteAc
* UserWriteExecuteAc
* EveryoneReadAdminWriteAc.
* UserReadWriteAc
* AdminReadWriteAc

For more information about these control permissions for file access, see “[File System Requirements](#_File_System_Requirements)” later in this specification.

Errors:

If CardCreateFile receives as a parameter the name of an existing file or directory (when creating a file in the root dir), it should fail with an ERROR\_FILE\_EXISTS error code.

If **CardCreateFile** is called on a nonexistent directory, an SCARD\_E\_DIR\_NOT\_FOUND error code must be returned.

If **CardCreateFile** is called on a file in a directory where the caller has no permissions to write, an SCARD\_W\_SECURITY\_VIOLATION error code must be returned.

If the name that *pszFileName* or *pszDirectoryName* specified is longer than the maximum length for file/directory names, SCARD\_E\_INVALID\_PARAMETER must be returned.

### **CardGetFileInfo**

Description:

This function retrieves information about a file, specifically its size and ACL information.

DWORD WINAPI CardGetFileInfo(

\_\_in PCARD\_DATA *pCardData*,

\_\_in\_opt LPSTR *pszDirectoryName*,

\_\_in LPSTR *pszFileName*,

\_\_inout PCARD\_FILE\_INFO *pCardFileInfo*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pszDirectoryName* Name of the directory that contains the file; NULL for root.

*pszFileName* Logical File Name for the file of interest.

*pCardFileInfo* Address of a CARD\_FILE\_INFO structure.

Output:

*pCardFileInfo* Caller’s CARD\_FILE\_INFO structure is filled in.

Return value Zero on success; otherwise, nonzero.

Comments:

**CardGetFileInfo** fails if the specified file does not exist.

For more information about file sizes, see the next section, “[**CardWriteFile**](#_CardWriteFile),” in this specification.

The file information that is returned is in the following structure.

typedef struct \_CARD\_FILE\_INFO {

DWORD dwVersion;

DWORD cbFileSize;

CARD\_FILE\_ACCESS\_CONDITION AccessCondition;

} CARD\_FILE\_INFO, \*PCARD\_FILE\_INFO;

The file size that is returned is the size of the data in its uncompressed form. It is not the “size of the file on the card.” Therefore, the reported size of a newly created file may be zero, even if that file was created with a nonzero file size. Alternatively, it can be the size used at file creation.

If **CardGetFileInfo** is called on a nonexistent directory, an SCARD\_E\_DIR\_NOT\_FOUND error code must be returned.

If CardGetFileInfo is called on an unreadable file, an SCARD\_W\_SECURITY\_VIOLATION error code must be returned

If the name that *pszFileName* or *pszDirectoryName* specified is longer than the maximum length for file/directory names, SCARD\_E\_INVALID\_PARAMETER must be returned.

### **CardWriteFile**

Description:

The **CardWriteFile** function writes the entire contents of a data buffer to a file. The file contents are replaced, starting at the beginning of the file. The file must exist, or **CardWriteFile** fails.

DWORD WINAPI CardWriteFile(

\_\_in PCARD\_DATA *pCardData*,

\_\_in\_opt LPSTR *pszDirectoryName*,

\_\_in LPSTR *pszFileName*,

\_\_in DWORD *dwFlags*,

\_\_in\_bcount(cbData) PBYTE *pbData*,

\_\_in DWORD *cbData*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pszDirectoryName* Name of the directory that contains the file; NULL for root.

*pszFileName* Logical File Name for the file of interest.

*dwFlags* Reserved—must be zero.

*pbData* Address of byte buffer that contains data to write to the file.

*cbData* Byte count of data to write to file.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

Rewriting the contents of an existing file presents special problems. If files change their allocated size, the available storage of the smart card can become fragmented. This results in significant loss of useful space. This is because it is usually infeasible to implement a reorganizing memory manager for card storage. Therefore, implementations can choose not to “shrink” a file if its size has been decreased.

**Note:** Users of this function should be careful not to rely on exact sizing of the file to its contents. The file size may exceed the data size.

Errors:

If the size (*cbData*) that is specified through **CardWriteFile** is larger than the current file size that is specified through **CardCreateFile**, it should succeed, unless the card is out of space. If this is true, SCARD\_E\_WRITE\_TOO\_MANY should be returned.

Card minidriver-based cards must be able to dynamically grow files.

If incorrect flags are passed into *dwFlags*, the **CardWriteFile** call is expected to fail with the SCARD\_E\_INVALID\_PARAMETER error code.

If *pszFileName* specifies a nonexistent file, **CardWriteFile** should fail with SCARD\_E\_FILE\_NOT\_FOUND.

If **CardWriteFile** is called on a nonexistent directory, an SCARD\_E\_DIR\_NOT\_FOUND error code must be returned.

If the name that *pszFileName* or *pszDirectoryName* specified is longer than the maximum length for file/directory names, SCARD\_E\_INVALID\_PARAMETER must be returned.

### **CardDeleteFile**

Description:

The **CardDeleteFile** function deletes the specified file. If the file does not exist, the returned Status value should indicate that the file did not exist.

DWORD WINAPI CardDeleteFile(

\_\_in PCARD\_DATA *pCardData*,

\_\_in\_opt LPSTR *pszDirectoryName*,

\_\_in LPSTR *pszFileName*,

\_\_in DWORD *dwFlags*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pszDirectoryName* Name of the directory that contains the file; NULL for root.

*pszFileName* Logical File Name for the file to be deleted.

*dwFlags* Must be zero.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

**CardDeleteFile** should check the flags parameter and return an SCARD\_E\_INVALID\_PARAMETER error code if incorrect flags are passed into *dwFlags*.

If authentication has not been done correctly before calling **CardDeleteFile**, an SCARD\_W\_SECURITY\_VIOLATION error code must be returned.

If **CardDeleteFile** is called on a nonexistent file, it should fail and return an SCARD\_E\_FILE\_NOT\_FOUND error.

If **CardDeleteFile** is called on an existing file that was created in a directory in which the caller has no permission to delete, an SCARD\_W\_SECURITY\_VIOLATION error code must be returned.

If **CardDeleteFile** is called on a nonexistent directory, an SCARD\_E\_DIR\_NOT\_FOUND error code must be returned.

If the name that *pszFileName* or *pszDirectoryName* specified is longer than the maximum length for file/directory names, SCARD\_E\_INVALID\_PARAMETER must be returned.

### **CardEnumFiles**

Description:

The **CardEnumFiles** function returns name information about available files in a directory as a multistring list.

DWORD WINAPI CardEnumFiles(

\_\_in PCARD\_DATA *pCardData*,

\_\_in\_opt LPSTR *pszDirectoryName*,

\_\_deref\_out\_ecount(\**pdwcbFileName*) LPSTR \**pmszFileNames*,

\_\_out LPDWORD *pdwcbFileName*,

\_\_in DWORD *dwFlags*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pszDirectoryName* Name of the directory; NULL for root.

*pmszFileNames* Pointer to byte pointer to receive returned multistring.

*pdwcbFileName* Size of allocation pointed to by *pmszFileNames*.

*dwFlags* Reserved—must be zero.

Output:

Return value Zero on success; otherwise, nonzero.

pmszFileNames File names of files in the named directory or for root if the passed directory name was NULL. If the directory does not contain files, an SCARD\_E\_FILE\_NOT\_FOUND error code should be returned.

Comments:

**CardEnumFiles** should check the *dwFlags* value to ensure that it is zero. If not, it should return SCARD\_E\_INVALID\_PARAMETER.

The multistring is allocated by the card minidriver and must be freed by the caller by using PFN\_CSP\_FREE. It is returned as a contiguous buffer and must require exactly one call to free. The format of this string is a multistring. It is a contiguous block of data. Individual strings are separated by “\0” characters. The block is terminated by two “\0” characters in a row (one for the final string and another to indicate that the multistring is finished).

If **CardEnumFiles** is called on a nonexistent directory, an SCARD\_E\_DIR\_NOT\_FOUND error code must be returned.

If the name that *pszDirectoryName* specified is longer than the maximum length for file/directory names, SCARD\_E\_INVALID\_PARAMETER must be returned.

### **CardQueryFreeSpace**

Description:

The **CardQueryFreeSpace** function determines the amount of available card storage space.

DWORD WINAPI CardQueryFreeSpace(

\_\_in PCARD\_DATA *pCardData*,

\_\_in DWORD *dwFlags*,

\_\_inout PCARD\_FREE\_SPACE\_INFO *pCardFreeSpaceInfo*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*dwFlags* Reserved—must be zero.

*pCardFreeSpaceInfo* Pointer to an uninitialized CARD\_FREE\_SPACE\_INFO structure.

Output:

*pCardFreeSpaceInfo* Card space information (such as the number of remaining bytes or the number of available key containers).

Return value Zero on success; otherwise, nonzero.

Comments:

Free space information is returned in the following structure.

typedef struct \_CARD\_FREE\_SPACE\_INFO

{

IN OUT DWORD dwVersion;

OUT DWORD dwBytesAvailable;

OUT DWORD dwKeyContainersAvailable;

OUT DWORD dwMaxKeyContainers;

} CARD\_FREE\_SPACE\_INFO, \*PCARD\_FREE\_SPACE\_INFO;

Sometimes these may be approximate values. Examples of the use of this information are determining if a new key container can be created and determining if the card has sufficient storage for a given certificate.

**CardQueryFreeSpace** should check the *dwFlags* value. If this is nonzero, it should fail and return SCARD\_E\_INVALID\_PARAMETER.

**Important** In the CARD\_FREE\_SPACE\_INFO structure, the caller must set the dwVersion member. The following are the currently defined values.

#define CARD\_FREE\_SPACE\_INFO\_CURRENT\_VERSION 1

In the CARD\_FREE\_SPACE\_INFO structure that was discussed earlier, values that are unknown should be set to CARD\_DATA\_VALUE\_UNKNOWN for each of the three fields that are used (dwBytesAvailable, dwKeyContainersAvailable, and/or dwMaxKeyContainers).

## Card Capabilities (Minidriver Version 5 and Earlier)

(The following section details implementation that is required for backward compatibility with Base CSP/KSP versions earlier than Version 6).

The card Base CSP/KSP must support multiple variations of specific cards and card minidrivers. To best take advantage of the capabilities of a specific card, the card specific minidriver provides an API that the Base CSP/KSP can use to query the full set of functionality that the card provides. If the Base CSP/KSP provides any functionality that the card provides, such as compression, the Base CSP/KSP should always rely on the card implementation. Otherwise, the Base CSP/KSP falls back to its own implementation of this functionality.

### Defines and Data Structures

#define CARD\_CAPABILITIES\_CURRENT\_VERSION 1

typedef struct \_CARD\_CAPABILITIES

{

IN OUT DWORD dwVersion;

IN BOOL fCertificateCompression;

IN BOOL fKeyGen;

} CARD\_CAPABILITIES, \*PCARD\_CAPABILITIES;

Members:

**dwVersion** The version of the structure that is being used.

**fCertificateCompression** Set TRUE to indicate that the card minidriver implements its own compression of certificates.

**fKeyGen** Set TRUE to indicate that the card can generate keys.

### **CardQueryCapabilities**

Description:

This function queries the card and card-specific minidriver combination for the functionality that is provided at this level, such as certificate or file compression.

DWORD WINAPI CardQueryCapabilities(

\_\_in PCARD\_DATA *pCardData*,

\_\_inout PCARD\_CAPABILITIES *pCardCapabilities*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pCardCapabilities* Card capabilities structure with version number set.

Output:

*pCardCapabilities* Filled-in PCARD\_CAPABILITIES structure.

Return value Zero on success; otherwise, nonzero.

Comments:

Certificate files should be written to the card in a compressed state and with accompanying error-checking information. The Base CSP/KSP performs these operations if the card minidriver cannot do so. The information that this function returns indicates whether the card minidriver or card can do so.

## Card and Container Properties

### Defines and Data Structures

The following defines functions that are specific to querying capabilities of a smart card and functions that are specific to discovering attributes of a container of key material on the card. This section applies beginning with Version 6 of the *Windows Smart Card Minidriver Specification*. However, for backward compatibility reasons, functions such as **CardQueryFreeSpace**, **CardQueryKeySizes**, and **CardQueryCapabilities** must be implemented as well.

The following is a list of card properties constants.

#define CP\_CARD\_FREE\_SPACE L"Free Space" #define CP\_CARD\_CAPABILITIES L"Capabilities" #define CP\_CARD\_KEYSIZES L"Key Sizes"

#define CP\_CARD\_READ\_ONLY L"Read Only Mode"

#define CP\_CARD\_CACHE\_MODE L"Cache Mode"

#define CP\_SUPPORTS\_WIN\_X509\_ENROLLMENT L"Supports Windows x.509 Enrollment"

#define CP\_CARD\_GUID L"Card Identifier"

#define CP\_CARD\_SERIAL\_NO L"Card Serial Number"

#define CP\_CARD\_PIN\_INFO L"PIN Information"

#define CP\_CARD\_LIST\_PINS L"PIN List"

#define CP\_CARD\_AUTHENTICATED\_STATE L"Authenticated State"

#define CP\_CARD\_PIN\_STRENGTH\_VERIFY L"PIN Strength Verify"

#define CP\_CARD\_PIN\_STRENGTH\_CHANGE L"PIN Strength Change"

#define CP\_CARD\_PIN\_STRENGTH\_UNBLOCK L"PIN Strength Unblock"

#define CP\_PARENT\_WINDOW L"Parent Window"

#define CP\_PIN\_CONTEXT\_STRING L"PIN Context String"

The following is a list of container properties constants.

#define CCP\_CONTAINER\_INFO L”Container Info”

#define CCP\_PIN\_IDENTIFIER L”PIN Identifier”

**Note:** CP\_CARD\_PIN\_STRENGTH\_CHANGE and CP\_CARD\_PIN\_STRENGTH\_UNBLOCK are currently not used by the Base CSP/KSP and should not be used by the minidriver.

### **CardGetContainerProperty**

Description:

The **CardGetContainerProperty** function is modeled after the query functions of CAPI for keys. It takes a LPWSTR that indicates which parameter is being requested. Then it returns data written into the *pbData* parameter.

DWORD WINAPI CardGetContainerProperty(

\_\_in PCARD\_DATA *pCardData*,

\_\_in BYTE *bContainerIndex*,

\_\_in LPWSTR *wszProperty*,

\_\_out\_bcount\_part\_opt(*cbData*, \**pdwDataLen*) PBYTE *pbData*,

\_\_in DWORD *cbData*,

\_\_out PDWORD *pdwDataLen*,

\_\_in DWORD *dwFlags*

);

Input:

*pCardData* Address of CARD\_DATA structure.

*bContainerIndex* Index to a key container on the card.

*wszProperty* LPWSTR that indicates which property is requested.

*pdData* Byte pointer to data buffer to receive the data.

*cbData* Length of input buffer.

*pdwDataLen* Pointer to a DWORD receiving the actual data length returned.

*dwFlags* Flags, currently reserved for future use.

Output:

Return value Zero on success; nonzero on failure.

Comments:

**CardGetContainerProperty** should check the *dwFlags* value. If this is nonzero, it should fail and return SCARD\_E\_INVALID\_PARAMETER.

If an unsupported *wszProperty* is passed, the call should fail and return SCARD\_E\_INVALID\_PARAMETER. Any minidriver can choose to define and support optional custom properties that are not defined in this specification.

If *cbData* is less than the length of the buffer to be returned, **CardGetContainerProperty** should return ERROR\_INSUFFICIENT\_BUFFER.

If **CardGetContainerProperty** is called with a nonexistent *bContainerIndex* parameter, it should return the SCARD\_E\_NO\_KEY\_CONTAINER error.

The format of *pbData* is different depending on the *wszProperty* parameter that is passed to the function. The following table is a list of the different types that *pbData* takes depending on *wszProperty* (the structures are serialized as byte arrays).

| ***wszProperty*** | ***pbData*** |
| --- | --- |
| CCP\_CONTAINER\_INFO | typedef struct \_CONTAINER\_INFO  {  DWORD dwVersion;  DWORD dwReserved;  DWORD cbSigPublicKey;  PBYTE pbSigPublicKey;  DWORD cbKeyExPublicKey;  PBYTE pbKeyExPublicKey;  )  CONTAINER\_INFO, \*PCONTAINER\_INFO;  **CardGetContainerProperty** allocates memory for **pbKeyExPublicKey** and **pbSigPublicKey** that the caller must free by calling PFN\_CSP\_FREE. |
| CCP\_PIN\_IDENTIFIER | In this situation, *pbData* contains a PIN\_ID that describes the PIN identifier of the PIN that is associated with this container. |
| CCP\_ASSOCIATED\_ECDH\_KEY | This property requests the return of the corresponding ECDH key container index for an ECDSA key that was used during logon. The return value in *pbData* is a container index of the associated ECDH key. |

### **CardSetContainerProperty**

Description:

This function sets the properties on containers. Only two container properties are supported:

* CCP\_PIN\_IDENTIFIER
* CCP\_ASSOCIATED\_ECDH\_KEY

DWORD WINAPI CardSetContainerProperty(

\_\_in PCARD\_DATA *pCardData*,

\_\_in BYTE *bContainerIndex*,

\_\_in LPWSTR *wszProperty*,

\_\_in\_bcount(cbDataLen) PBYTE *pbData*,

\_\_in DWORD *cbDataLen*,

\_\_in DWORD *dwFlags*

);

Input:

*pCardData* Address of CARD\_DATA structure.

*bContainerIndex* Index to a key container on the card.

*wszProperty* LPWSTR that indicates which property is requested.

*pdData* Byte pointer to data buffer that contains the data.

*cbDataLen* DWORD that indicates the data buffer length.

*dwFlags* Reserved—must be zero.

Output:

Return value Zero on success; nonzero on failure.

Comments:

**CardSetContainerProperty** should check the *dwFlags* value. If this is nonzero, it should fail and return SCARD\_E\_INVALID\_PARAMETER.

If an unsupported *wszProperty* is passed to **CardSetContainerProperty**, it should fail and return SCARD\_E\_INVALID\_PARAMETER or SCARD\_E\_UNSUPPORTED\_FEATURE.

**Note:** Any minidriver can choose to define and support optional custom properties that are not defined in this specification.

If **CardSetContainerProperty** is called with a nonexistent bContainerIndex parameter, it should return an SCARD\_E\_NO\_KEY\_CONTAINER error.

The format of *pbData* is different depending on the *wszProperty* parameter that is passed to the function. The following table is a list of the different types that *pbData* takes depending on *wszProperty* (the structures are serialized as byte arrays).

| ***wszProperty*** | **pdData** |
| --- | --- |
| CCP\_PIN\_IDENTIFIER | In this situation, *pbData* contains a DWORD that describes the PIN identifier to the PIN that is associated with this container.  Although this function is not consumed by the Base CSP/KSP, the following are some guidelines for this function:   * When a new key is created on the card, the user PIN must be authenticated and the new key container is associated with ROLE\_USER. This function is used to update the PIN property if needed. * The PIN identifier can be updated only by using the user PIN or the administrator PIN. * The administrator PIN cannot be associated with a key container. * If the user PIN is currently authenticated and this function is called to associate the key container with, for example, PIN #3, PIN #3 must be authenticated to use this key.   If the key container already has a PIN associated with it, ROLE\_USER or the associated object PIN can be used to change the associated PIN. |
| CCP\_ASSOCIATED\_ECDH\_KEY | In this situation, *pbData* points to a container index for the ECDH key of an ECDSA key container. This creates an association between an ECDSA key and an ECDH key for logon. |

### **CardGetProperty**

Description:

The **CardGetProperty** function is modeled after the query functions of CAPI for keys. It takes a LPWSTR that indicates which parameter is being requested. The function returns data in the *pbData* parameter.

DWORD WINAPI CardGetProperty(

\_\_in PCARD\_DATA *pCardData*,

\_\_in LPWSTR *wszProperty*,

\_\_out\_bcount\_part\_opt(*cbData*, \**pdwDataLen*) PBYTE *pbData*,

\_\_in DWORD *cbData*,

\_\_out PDWORD *pdwDataLen*,

\_\_in DWORD *dwFlags*

);

Input:

*pCardData* Address of CARD\_DATA structure.

*wszProperty* LPWSTR that indicates which property is requested.

*pbData* Byte pointer to data buffer to receive the data.

*cbData* Length of input buffer.

*pdwDataLen* Pointer to a DWORD receiving the actual data length returned.

*dwFlags* Flags.

Output:

Return value Zero on success; nonzero on failure.

Comments:

**CardGetProperty** should check the *dwFlags* value. Unless *dwFlags* is specified for the property and the value is nonzero, it should fail and return SCARD\_E\_INVALID\_PARAMETER.

If an unsupported *wszProperty* is passed to **CardGetProperty**, it should fail and return SCARD\_E\_INVALID\_PARAMETER or SCARD\_E\_UNSUPPORTED\_FEATURE. Implementing all the following properties is mandatory unless explicitly stated otherwise. Any minidriver can choose to define and support optional custom properties that are not defined in this specification.

If *cbData* is less than the length of the buffer that is to be returned, **CardGetProperty** should return ERROR\_INSUFFICIENT\_BUFFER.

**Important**: Careful attention must be taken when returning CP\_READ\_ONLY\_CARD as true. When this property is returned as true, all write operations to the card are blocked at the Base CSP layer.

The format of *pbData* is different depending on the *wszProperty* parameter that is passed to the function.

The following table is a list of the different types that *pbData* takes depending on *wszProperty* (the structures are serialized as byte arrays).

| ***wszProperty*** | ***pbData* type** | ***pbData* value** |
| --- | --- | --- |
| CP\_CARD\_FREE\_SPACE |  | typedef struct \_CARD\_FREE\_SPACE\_INFO  {  DWORD dwVersion;  DWORD dwBytesAvailable;  DWORD dwKeyContainersAvailable;  DWORD dwMaxKeyContainers;  ) CARD\_FREE\_SPACE\_INFO, \*PCARD\_FREE\_SPACE\_INFO; |
| CP\_CARD\_CAPABILITIES |  | typedef struct \_CARD\_CAPABILITIES  {  DWORD dwVersion;  BOOL fCertificateCompression;  BOOL fKeyGen;  )  CARD\_CAPABILITIES, \*PCARD\_CAPABILITIES; |
| CP\_CARD\_KEYSIZES |  | *dwFlags* indicates key type to be queried. This is one of the AT\_\* defined values, such as AT\_SIGNATURE or AT\_ECDSA\_P256.  typedef struct \_CARD\_KEY\_SIZES  {  DWORD dwVersion;  DWORD dwMinimumBitlen;  DWORD dwDefaultBitlen;  DWORD dwMaximumBitlen;  DWORD dwIncrementalBitlen;  )  CARD\_KEY\_SIZES, \*PCARD\_KEY\_SIZES;  A card minidriver that supports read-only cards may support more key types than what the specific read only card has been provisioned with. In this case the call may succeed and return a corresponding CARD\_KEY\_SIZES structure for the supported key spec. |
| CP\_CARD\_READ\_ONLY | BOOL | If True, all write operations are blocked at the Base CSP layer.  This flag also affects the data cache. If the card indicates that it is read-only, the Base CSP/KSP does not write to the cardcf file. |
| CP\_CARD\_CACHE\_MODE | DWORD | #define CP\_CACHE\_MODE\_GLOBAL\_CACHE 1  #define CP\_CACHE\_MODE\_SESSION\_ONLY 2  #define CP\_CACHE\_MODE\_NO\_CACHE 3 |
| CP\_SUPPORTS\_WIN\_X509 \_ENROLLMENT | BOOL | Indicates whether Windows PKI should be allowed to write or renew certificates on the card. This should be used to avoid unexpected results because of a lack of support for multiple PINs in Windows PKI enrollment client. |
| CP\_CARD\_GUID | BYTE[] | In this situation, *pbData* is a buffer that contains a unique GUID for the card. This value must exactly match the GUID in the “cardid” file. |
| CP\_CARD\_SERIAL\_NO | BYTE[] | In this situation, *pbData* is a buffer that contains a serial number for the card. The format of the serial number is opaque to the Base CSP and is intended for other applications that query the card minidriver directly.  This is an optional property that may be supported by the card. |
| CP\_CARD\_PIN\_INFO | PIN\_INFO | In this situation, *pbData* is a PIN\_INFO structure that contains information about the PIN. The *dwFlags* parameter contains the identifier of the PIN to return. |
| CP\_CARD\_LIST\_PINS | PIN\_SET | In this situation, *pbData* contains a PIN\_SET that indicates by a bit-mask what entities the card currently uses. |
| CP\_CARD\_AUTHENTICATED \_STATE | PIN\_SET | In this situation, *pbData* contains a PIN\_SET that indicates by a bit-mask what entities the card currently authenticates.  This is an optional property that may be supported by the card. |
| CP\_CARD\_PIN\_STRENGTH \_VERIFY |  | In this situation, *pbData* contains a bitmask of one or more of the following values:   * CARD\_PIN\_STRENGTH\_PLAINTEXT – Card can accept a plaintext PIN for authentication. * CARD\_PIN\_STRENGTH\_SESSION\_PIN – Card can generate a session PIN that can be used for subsequent authentications. * The *dwFlags* parameter contains the identifier of the PIN to return.   The following points apply to PIN strength:   * Currently the PIN strength is ignored for EmptyPinType and ChallengeResponsePinType. * Even if CARD\_PIN\_STRENGTH\_SESSION\_PIN is set for a PIN, the plaintext PIN must also be accepted for authentication. This is because trusted processes in Windows may use the plaintext PIN. |
| CP\_KEY\_IMPORT\_SUPPORT | DWORD | In this situation, *pbData* is a DWORD value. This value is a bitmask that describes the types of key import which the card supports.  #define CARD\_KEY\_IMPORT\_PLAIN\_TEXT 0x1  #define CARD\_KEY\_IMPORT\_RSA\_KEYEST 0x2  #define CARD\_KEY\_IMPORT\_ECC\_KEYEST 0x4  #define CARD\_KEY\_IMPORT\_SHARED\_SYMMETRIC 0x8  This property is defined in version 7 and later versions of this specification.  **Note:** This property is read-only. The flags can be combined with the ‘or’ operation to indicate whether secure key injection is supported with either clear text or asymmetric key establishment. A value of zero indicates that key import is not supported on this smart card. |
| CP\_ENUM\_ALGORITHMS | LPWSTR | In this situation, *pbData* contains a multistring value that describes the algorithms that the minidriver supports. The *dwFlags* parameter for this operation should contain one of the following values that define the type of algorithm enumeration to perform:  #define CARD\_CIPHER\_OPERATION 0x1 // Symmetric operations  #define CARD\_ASYMMETRIC\_OPERATION 0x2 // Asymmetric operations  The returned data can contain any of the following strings that identify the algorithms:  #define CARD\_3DES\_ALGORITHM L“3DES”  #define CARD\_3DES\_112\_ALGORITHM L“3DES\_112”  #define CARD\_AES\_ALGORITHM L“AES”  This property is defined in version 7 and later versions of this specification.  **Note:** This property is read-only. 3DES\_112 is for the situation in which the first and third keys are the same. |
| CP\_PADDING\_SCHEMES | DWORD | In this situation, *pbData* contains a DWORD value. This value is a bitmask that describes the types of padding that the card supports for RSA cryptographic operations.  The *dwFlags* parameter controls the cipher operation to which the query pertains. For more information, see the CP\_ENUM\_ALGORITHMS property value that was previously described.  For asymmetric keys, the return type contains the bitmask with the following padding schemes:  #define CARD\_PADDING\_PKCS1 0x00000002  #define CARD\_PADDING\_PSS 0x00000004  #define CARD\_PADDING\_OAEP 0x00000008  For symmetric keys, only one flag is defined that indicates whether the message was padded up to the next block size. If the message is a multiple of the block size, no padding is needed:  #define CARD\_BLOCK\_PADDING 0x1  This property is defined in version 7 and later versions of this specification.  **Note:** This property is read-only. For symmetric key padding, only support for PKCS#5 padding will exist. |
| CP\_CHAINING\_MODES | LPWSTR | In this situation, *pbData* contains a multistring value that describes the types of chaining modes that the card supports for symmetric key operations. Currently, the only supported chaining mode is cipher block chaining (CBC):  #define CARD\_CHAIN\_MODE\_CBC L”ChainingModeCBC”  This property is defined in version 7 and later versions of this specification.  **Note:** This property is read-only. |

### **CardSetProperty**

Description:

This function can be used to set properties on the card.

DWORD WINAPI CardSetProperty(

\_\_in PCARD\_DATA *pCardData*,

\_\_in LPWSTR *wszProperty*,

\_\_in\_bcount(*cbDataLen*) PBYTE *pbData*,

\_\_in DWORD *cbDataLen*,

\_\_in DWORD *dwFlags*

);

Input:

*pCardData* Address of CARD\_DATA structure.

*wszProperty* LPWSTR that indicates which property is being set.

*pbData* Byte pointer to data buffer that contains the data.

*cbDataLen* DWORD that indicates the data buffer length.

*dwFlags* Flags.

Output:

Return value Zero on success; nonzero on failure.

Comments:

For read-only cards, setting properties through **CardSetProperty** is optional.

**CardSetProperty** should check the *dwFlags* value. Unless *dwFlags* is specified for the property and the value is nonzero, it should fail and return SCARD\_E\_INVALID\_PARAMETER.

If an unsupported *wszProperty* is passed to **CardSetProperty**, it should fail and return SCARD\_E\_UNSUPPORTED\_FEATURE. Any minidriver can choose to define and support optional custom properties that are not defined in this specification.

The format of *pbData* is different depending on the *wszProperty* parameter that is passed to the function. For a list of the different types that *pbData* takes depending on *wszProperty*, see ”[**CardGetProperty**](#_CardGetPropertyCardGetProperty)” earlier in this specification.

The following properties are read-only and are not supported by the **CardSetProperty** function:

* CP\_CARD\_FREE\_SPACE
* CP\_CARD\_CAPABILITIES
* CP\_CARD\_KEYSIZES
* CP\_CARD\_LIST\_PINS
* CP\_CARD\_AUTHENTICATED\_STATE
* CP\_KEY\_IMPORT\_SUPPORT
* CP\_ENUM\_ALGORITHMS
* CP\_PADDING\_SCHEMES
* CP\_CHAINING\_MODES

**CardSetProperty** must return SCARD\_E\_UNSUPPORTED\_FEATURE or SCARD\_W\_SECURITY\_VIOLATION for the read-only properties on the preceding functions.

**CardSetProperty** must return SCARD\_E\_UNSUPPORTED\_FEATURE or SCARD\_W\_SECURITY\_VIOLATION for read-only cards for the following properties:

* CP\_CARD\_CACHE\_MODE
* CP\_SUPPORTS\_WIN\_X509\_ENROLLMENT
* CP\_CARD\_GUID
* CP\_CARD\_SERIAL\_NO
* CP\_CARD\_PIN\_INFO
* CP\_PARENT\_WINDOW
* CP\_PIN\_CONTEXT\_STRING
* CP\_CARD\_PIN\_STRENGTH\_VERIFY
* CP\_CARD\_PIN\_STRENGTH\_CHANGE
* CP\_CARD\_PIN\_STRENGTH\_UNBLOCK

CP\_CARD\_READ\_ONLY is writable if the appropriate level of authentication to the card is successful. SCARD\_W\_SECURITY\_VIOLATION should be returned if it is supported. However, the appropriate principal (ROLE\_ADMIN) is not authenticated.

To set card properties, the specified permission in the following table must be satisfied.

| ***wszProperty*** | ***pbData*** | **Permission** |
| --- | --- | --- |
| CP\_CARD\_CACHE\_MODE | In this situation *pbData* is a DWORD value. Three flags indicate which cache mode to use:  #define CP\_CACHE\_MODE\_GLOBAL\_CACHE 1  #define CP\_CACHE\_MODE\_SESSION\_ONLY 2  #define CP\_CACHE\_MODE\_NO\_CACHE 3  Minidrivers that do not support the modification of this property can return SCARD\_E\_UNSUPPORTED\_FEATURE. | Administrator |
| CP\_SUPPORTS\_WIN\_X509 \_ENROLLMENT | If False, enrollment operations is blocked at the Base CSP layer.  Minidrivers that do not support the modification of this property can return SCARD\_E\_UNSUPPORTED\_FEATURE. | Administrator |
| CP\_CARD\_GUID | In this situation, *pbData* is a buffer that contains a unique GUID for the card. Whether updating the GUID by property or through the “cardid” file, retrieving the GUID by either means should always return the same value. | Administrator |
| CP\_CARD\_SERIAL\_NO | In this situation, *pbData* is a buffer that contains a serial number for the card.  This is an optional property that the card may choose to support. | Administrator |
| CP\_CARD\_PIN\_INFO | In this situation, *pbData* is a PIN\_INFO structure that contains information about the PIN. The *dwFlags* parameter contains the identifier of the PIN to return.  If the PIN\_INFO structure contains information that the card minidriver does not support, such as the PIN\_INFO\_REQUIRE\_SECURE\_ENTRY flag, the card minidriver should return SCARD\_E\_UNSUPPORTED.  Minidrivers that do not support the modification of this property can return SCARD\_E\_UNSUPPORTED\_FEATURE. | Administrator |
| CP\_PARENT\_WINDOW | In this situation, *pbData* is a HANDLE to the parent window. If the card minidriver wants to show UI to collect an external PIN, this property should be used to tie the UI to the parent window.  **Note:** This property is required only for cards that support external PINs. | Everyone |
| CP\_PIN\_CONTEXT\_STRING | In this situation, *pbData* is a LPWSTR that contains context information from the application. If the card minidriver wants to show UI to collect an external PIN, this property should be used to display the context string from the calling application.  **Note:** *pbData* may be NULL if an application has not set a context string.  **Note:** This property is only required for cards that support external PINs. | Everyone |
| CP\_CARD\_PIN\_STRENGTH \_VERIFY | In this situation, *pbData* contains a bitmask of one or more of the following values:   * CARD\_PIN\_STRENGTH\_PLAINTEXT – Card can accept a plaintext PIN for authentication. * CARD\_PIN\_STRENGTH\_SESSION\_PIN – Card can generate a session PIN that should be used for subsequent authentications.   The *dwFlags* parameter contains the identifier of the PIN to return.  Minidrivers that do not support the modification of this property can return SCARD\_E\_UNSUPPORTED\_FEATURE. | Administrator |

## Key Container

CAPI handles key information in association with “containers.” The following functions support the creation, enumeration, and deletion of containers.

### **CardCreateContainer**

Description:

The **CardCreateContainer** function creates a new key container that is identified by the container index that the *bContainerIndex* argument specifies. For applications in which the card does not support on-card key generation or if it is desired to archive the keys, the key material can be supplied with the call by specifying in flags that the card is to import the supplied key material.

DWORD WINAPI CardCreateContainer(

\_\_in PCARD\_DATA *pCardData*,

\_\_in BYTE *bContainerIndex*,

\_\_in DWORD *dwFlags*,

\_\_in DWORD *dwKeySpec*,

\_\_in DWORD *dwKeySize*,

\_\_in PBYTE *pbKeyData*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*bContainerIndex* Index number for this container.

*dwFlags* CARD\_CREATE\_CONTAINER\_KEY\_GEN or CARD\_CREATE\_CONTAINER\_KEY\_IMPORT.

*dwKeySpec* AT\_ECDHE\_P256, AT\_ECDHE\_P384, AT\_ECDHE\_P521, AT\_ECDSA\_P256, AT\_ECDSA\_P384, or AT\_ECDSA\_P521 specify ECC keys.

AT\_SIGNATURE or AT\_KEYEXCHANGE specify RSA keys and are usable on dual-mode cards.

*dwKeySize* The size, in bits, of the key material. This value must be zero for ECC keys when the key is generated in the card. For RSA keys, this must specify the key bit length.

*pbKeyData* If *dwFlags* is set to CARD\_CREATE\_CONTAINER\_KEY\_IMPORT, a pointer to the passed key material, or else ignored for CARD\_CREATE\_CONTAINER\_KEY\_GEN.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

Containers are referenced in the interface between the Base CSP/KSP and the card minidriver by index number. These index numbers are assigned by the Base CSP/KSP. Therefore, the Base CSP/KSP maintains a map file that is named *mscp\Map* on the card. This file lists the CAPI/CNG GUIDs for the containers that were used to this point on the card.

For a new container, the Base CSP/KSP selects the next container or a previously vacated one. A container can be vacated by setting the GUID information in the Map file to zero for that index.

The card minidriver *can* support both the CARD\_CREATE\_CONTAINER\_KEY\_GEN and CARD\_CREATE\_CONTAINER\_KEY\_IMPORT parameters, but *must* support at least one of these parameters.

If CARD\_CREATE\_CONTAINER\_KEY\_GEN or CARD\_CREATE\_CONTAINER\_KEY\_IMPORT is passed and the card does not support that feature, the call should return SCARD\_E\_UNSUPPORTED\_FEATURE.

If the target container already exists, it is overwritten by the new one. The new container always contains a valid key if the call succeeds. The two methods of creating a new container are through random key generation and importing existing key data. If a wrong value for bContainerIndex is passed (invalid or nonexistent), an SCARD\_E\_NO\_KEY\_CONTAINER return value is expected.

Imported key material is passed in a “private key BLOB,” which is typically returned from **CryptExportKey**. For more information, see “[CryptExportKey Function](http://msdn.microsoft.com/en-us/library/aa379931(VS.85).aspx)“ and “[Base Provider Key BLOBs](http://msdn.microsoft.com/en-us/library/aa375601(VS.85).aspx)” on MSDN.

RSA keys comply with the CAPI key BLOB format. If the card supports the key type that *dwKeySpec* specifies but *dwKeySize* is invalid or unsupported, the card minidriver should reject the operation and return either SCARD\_E\_INVALID\_PARAMETER or SCARD\_E\_UNSUPPORTED\_FEATURE.

If *dwKeySpec* is invalid or undefined, the function should return a value of SCARD\_E\_INVALID\_PARAMETER. If the *dwKeySpec* value is defined but not supported, the function should return a value of SCARD\_E\_UNSUPPORTED\_FEATURE.

Only users can create containers. Both administrators and users should be able to obtain information and delete containers. If an administrator tries to create a container, the SCARD\_W\_SECURITY\_VIOLATION error should be returned.

Error checking is performed based on the order of cost. The minidriver should perform all parameter verification first without communicating to the card. This would include the validation of the *pbKeyData* and *dwKeySize* parameters. Error conditions that would result in a return of SCARD\_E\_UNSUPPORTED\_FEATURE must be checked first.

### **CardCreateContainerEx**

Description:

The **CardCreateContainerEx** function creates a new key container that the container index identifies and the *bContainerIndex* parameter specifies. The function associates the key container with the PIN that the *PinId* parameter specified.

This function is useful if the card-edge does not allow for changing the key attributes after the key container is created. This function replaces the need to call **CardSetContainerProperty** to set the CCP\_PIN\_IDENTIFIER property **CardCreateContainer** is called.

The caller of this function can provide the key material that the card imports. This is useful in those situations in which the card either does not support internal key generation or the caller requests that the key be archived in the card.

DWORD WINAPI CardCreateContainerEx(

\_\_in PCARD\_DATA *pCardData*,

\_\_in BYTE *bContainerIndex*,

\_\_in DWORD *dwFlags*,

\_\_in DWORD *dwKeySpec*,

\_\_in DWORD *dwKeySize*,

\_\_in PBYTE *pbKeyData*,

\_\_in PIN\_ID *PinId*

);

Input:

*pCardData* Context information for the call. For more information, see ”[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*bContainerIndex* Index number for this container.

*dwFlags* CARD\_CREATE\_CONTAINER\_KEY\_GEN or CARD\_CREATE\_CONTAINER\_KEY\_IMPORT.

*dwKeySpec* AT\_ECDHE\_P256, AT\_ECDHE\_P384, AT\_ECDHE\_P521, AT\_ECDSA\_P256, AT\_ECDSA\_P384, or AT\_ECDSA\_P521 specifies ECC keys.

AT\_SIGNATURE or AT\_KEYEXCHANGE specifies RSA keys and can be used on dual-mode cards.

*dwKeySize* The size, in bits, of the key material. This value must be zero for ECC keys when the key is generated in the card. For RSA keys, this must specify the key bit length.

*pbKeyData* If *dwFlags* is set to CARD\_CREATE\_CONTAINER\_KEY\_IMPORT, this is a pointer to the passed key material. Otherwise, this parameter is ignored.

*PinId* PIN Identifier for the container.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

Containers are referenced in the interface between the Base CSP/KSP and the card minidriver by index number. The Base CSP/KSP assigns these index numbers. Therefore, the Base CSP/KSP maintains a map file that is named *mscp\Map* on the card. This file lists the CAPI/CNG GUIDs for the containers that were used to this point on the card.

For a new container, the Base CSP/KSP selects the next container or a previously vacated one. A container can be vacated by setting the GUID information in the *mscp\Map* file to zero for that index.

The card minidriver can support both the CARD\_CREATE\_CONTAINER\_KEY\_GEN and CARD\_CREATE\_CONTAINER\_KEY\_IMPORT parameters, but must support at least one of these parameters.

If CARD\_CREATE\_CONTAINER\_KEY\_GEN or CARD\_CREATE\_CONTAINER\_KEY\_IMPORT is passed and the card does not support that feature, the function should return SCARD\_E\_UNSUPPORTED\_FEATURE.

If the target container already exists, it is overwritten by the new one. The new container always contains a valid key if the call succeeds. The two methods of creating a new container are through random key generation and importing existing key data. If an invalid or nonexistent value for *bContainerIndex* is passed in the call, the function should return SCARD\_E\_NO\_KEY\_CONTAINER.

Imported key material is passed in a “private key BLOB” that is typically returned from **CryptExportKey**. For a description of this format, see the documentation in the Platform SDK for **CryptExportKey**. For more information, see “[CryptExportKey Function](http://msdn.microsoft.com/en-us/library/aa379931(VS.85).aspx)“ and “[Base Provider Key BLOBs](http://msdn.microsoft.com/en-us/library/aa375601(VS.85).aspx)” on MSDN. RSA keys comply with the CAPI key BLOB format.

If the card supports the key type that *dwKeySpec* specified but *dwKeySize* is invalid or unsupported, the card minidriver should reject the operation and return either SCARD\_E\_INVALID\_PARAMETER or SCARD\_E\_UNSUPPORTED\_FEATURE.

If *dwKeySpec* is invalid or undefined, the function should return a value of SCARD\_E\_INVALID\_PARAMETER. If the *dwKeySpec* value is defined but not supported, the function should return a value of SCARD\_E\_UNSUPPORTED\_FEATURE.

Only users can create containers. Both administrators and users should be able to obtain information and delete containers. If an administrator attempts to create a container, the SCARD\_W\_SECURITY\_VIOLATION error should be returned.

Error checking is performed based on the order of cost. The minidriver should perform all parameter verification first without communicating to the card. This would include the validation of the *pbKeyData* and *dwKeySize* parameters. Error conditions that would result in a return of SCARD\_E\_UNSUPPORTED\_FEATURE must be checked first.

### **CardDeleteContainer**

Description:

The **CardDeleteContainer** function deletes the key container specified by its index value. This is done by deleting all key material (public and private) that is associated with that index value.

DWORD WINAPI CardDeleteContainer(

\_\_in PCARD\_DATA *pCardData*,

\_\_in BYTE *bContainerIndex*,

\_\_in DWORD *dwReserved*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

bContainerIndex KSP-assigned index for the CAPI container that is to be deleted.

dwReserved Must be zero.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

This function deletes the key material that is associated with the indexed container. Certificates are deleted separately by the Base CSP/KSP through calls to **CardDeleteFile** for the files that contain the affected certificates. **CardDeleteContainer** removes key material that is not reachable through the file system. Note also that update of the ContainerMapFile is completely the responsibility of the Base CSP/KSP, which it does through the file system calls.

Status should indicate success if the container existed and was successfully deleted. If **CardDeleteContainer** is called with an invalid or nonexistent *bContainerIndex* parameter, it should succeed.

### **CardGetContainerInfo**

Description:

The **CardGetContainerInfo** function queries the specified key container for more information about which keys are present, such as its key specification (such as AT\_ECDSA\_P384).

DWORD WINAPI CardGetContainerInfo(

\_\_in PCARD\_DATA *pCardData*,

\_\_in BYTE *bContainerIndex*,

\_\_in DWORD *dwFlags*,

\_\_inout PCONTAINER\_INFO *pContainerInfo*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*bContainerIndex* The index for the container, which the Base CSP/KSP assigns.

*dwFlags* Reserved—must be zero.

*pContainerInfo* Pointer to a CONTAINER\_INFO structure that the caller supplies and that the card minidriver fills.

Output:

pContainerInfo Information, which may include public key material.

Return value Zero on success; otherwise, nonzero.

Comments:

**CardGetContainerInfo** allocates memory that the caller must free by calling PFN\_CSP\_FREE.

The container information is returned in the following structure.

#define CONTAINER\_INFO\_CURRENT\_VERSION 1

typedef struct \_CONTAINER\_INFO

{

IN OUT DWORD dwVersion;

IN DWORD dwReserved;

OUT DWORD cbSigPublicKey;

OUT PBYTE pbSigPublicKey;

OUT DWORD cbKeyExPublicKey;

OUT PBYTE pbKeyExPublicKey;

} CONTAINER\_INFO, \*PCONTAINER\_INFO;

If the **cbSigPublicKey** and **pbSigPublicKey** members are not set In the CONTAINER\_INFO structure, it implies that the Signature key is not present.

If the **cbKeyExPublicKey** and **pbKeyExPublicKey** members are not set In the CONTAINER\_INFO structure, it implies that the Encryption (Key Exchange) key is not present.

The **dwVersion** member must be set by the caller.

If **CardGetContainerInfo** is called with an invalid or nonexistent *bContainerIndex* parameter, it should return the SCARD\_E\_NO\_KEY\_CONTAINER error.

It is not necessary for the caller to be authenticated to the card for **CardGetContainerInfo** to succeed.

## Cryptographic Operations

### **CardRSADecrypt**

Description:

This function performs an RSA decryption operation on the passed buffer by using the private key that a container index refers to. Note that for ECC-only smart cards, this entry point is not defined and is set to NULL in the returned CARD\_DATA structure from **CardAcquireContext**. This operation is restricted to a single buffer of a size equal to the key modulus.

DWORD WINAPI CardRSADecrypt(

\_\_in PCARD\_DATA *pCardData*,

\_\_inout PCARD\_RSA\_DECRYPT\_INFO *pInfo*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pInfo* Structure that contains data to be decrypted, which the Base CSP/KSP allocates.

Output:

pInfo Returned buffer that contains data that the card processed.

Return value Zero on success; otherwise, nonzero.

Comments:

Minidrivers that do not support OnCard padding removal to the card should call PFN\_CSP\_UNPAD\_DATA to perform padding removal. Information about the padding can be retrieved from the **pPaddingInfo** and **dwPaddingType** members of the CARD\_RSA\_DECYPT\_INFO structure to which *pInfo* points.

If the **dwVersion** member of the CARD\_RSA\_DECRYPT\_INFO structure has a value that is less than CARD\_RSA\_KEY\_DECRYPT\_INFO\_CURRENT\_VERSION and the **dwVersion** member of *pCardData* is set to CARD\_DATA\_CURRENT\_VERSION, this function should return ERROR\_REVISION\_MISMATCH.

Data to be processed by the card is passed in and received back in the following structure that is defined in *Cardmod.h*.

#define CARD\_RSA\_KEY\_DECRYPT\_INFO\_VERSION\_TWO 2

#define CARD\_RSA\_KEY\_DECRYPT\_INFO\_CURRENT\_VERSION CARD\_RSA\_KEY\_DECRYPT\_INFO\_VERSION\_TWO

typedef struct \_CARD\_RSA\_DECRYPT\_INFO

{

DWORD dwVersion;

BYTE bContainerIndex;

// For RSA operations, this should be AT\_SIGNATURE or AT\_KEYEXCHANGE.

DWORD dwKeySpec;

// This is the buffer and length that the caller expects to be decrypted.

// For RSA operations, cbData is redundant since the length of the buffer

// should always be equal to the length of the key modulus.

PBYTE pbData;

DWORD cbData;

// The following parameters are new in version 2 of the

// CARD\_RSA\_DECRYPT\_INFO structure.

// Currently supported values for dwPaddingType are

// CARD\_PADDING\_PKCS1, CARD\_PADDING\_OAEP, and CARD\_PADDING\_NONE.

// If dwPaddingType is set to CARD\_PADDING\_OAEP, then pPaddingInfo

// will point to a BCRYPT\_OAEP\_PADDING\_INFO structure.

LPVOID pPaddingInfo;

DWORD dwPaddingType;

} CARD\_RSA\_DECRYPT\_INFO, \*PCARD\_RSA\_DECRYPT\_INFO;

The **dwKeySpec** member indicates the usage type for the key. For the allowed values, see **CardCreateContainer**.

The **dwVersion** member should be set by the caller.

The input data should be padded by the Base CSP/KSP to meet the requirements of the algorithm that the caller requested. For RSA decryption, the buffer size is always equal in length to the public modulus. This frees the card-specific layer from having to implement various padding schemes.

If the card does not support OnCard padding removal, the Base CSP/KSP validates the padding in the plain text. Therefore, this API should succeed except in a hardware error. If the card minidriver finds that the buffer size is insufficient, it should return SCARD\_E\_INSUFFICIENT\_BUFFER.

The input data is passed in little-endian format.

In an RSA decrypt operation if *bContainerIndex* parameter is invalid or nonexistent, it should return the SCARD\_E\_NO\_KEY\_CONTAINER error.

### **CardConstructDHAgreement**

Description:

The **CardConstructDHAgreement** function performs a secret agreement calculation for Diffie Hellman (DH) key exchange by using a private key that is present on the card. For RSA-only card minidrivers, this entry point is not defined and is set to NULL in the CARD\_DATA structure that is returned from **CardAcquireContext**. The CARD\_DH\_AGREEMENT structure changes to allow for return of a handle to the agreed secret. This raises a point about how to index the DH agreement on the card in an opaque manner. Maintaining a map file is unnecessary because Ncrypt makes no provision for persistent DH agreements and there is no way to retrieve one after a provider is closed. DH agreements are addressable on card through an opaque BYTE that the card minidriver maintains. This BYTE should be associated with a handle to a card-side agreement.

DWORD WINAPI CardConstructDHAgreement(

\_\_in PCARD\_DATA *pCardData*,

\_\_inout PCARD\_DH\_AGREEMENT\_INFO *pSecretInfo*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pSecretInfo* Information that needs necessary context to calculate the secret agreement. This structure is also used to return the results.

Output:

*pSecretInfo* The **bSecretAgreementIndex** member is updated within the passed-in structure to which *pSecretInfo* points.

Return value Zero on success; otherwise, nonzero.

Comments:

Like **CardRSADecrypt**, the information is passed to this routine through the following structure.

#define CARD\_DH\_AGREEMENT\_INFO\_VERSION 2

typedef struct \_CARD\_DH\_AGREEMENT\_INFO

{

DWORD dwVersion;

BYTE bContainerIndex;

DWORD *dwFlags*;

DWORD dwPublicKey;

PBYTE pbPublicKey;

PBYTE pbReserved;

DWORD cbReserved;

OUT BYTE bSecretAgreementIndex;

} CARD\_DH\_AGREEMENT\_INFO, \*PCARD\_DH\_AGREEMENT\_INFO;

Version 1 of the structure is not supported on any card minidriver that is intended to be certified for FIPS 140-2. If the **dwVersion** member has a value of 1, the function should return ERROR\_REVISION\_MISMATCH.

One can support as many agreements in parallel. If no space to store an agreement exists, the function should return SCARD\_E\_NO\_MEMORY.

**Note:** You can implement **bSecretAgreementIndex** as a persistent counter on the card. We expect that a secret agreement is ephemeral in nature and not usable after the card has been removed. This index is also not designed to be used across processes.

### **CardDeriveKey**

Description:

The key derivation structure represents the majority of the required changes for FIPS 140-2 compliance for smart cards. It holds the requested key derivation function (KDF) and the associated input. The KDFs are defined in the “[CNG Reference](http://msdn.microsoft.com/en-us/library/aa376214(VS.85).aspx)” documentation on MSDN. For RSA-only card minidrivers, this entry point is not defined and is set to NULL in the CARD\_DATA structure that is returned from **CardAcquireContext**.

The **CardDeriveKey** function is defined as follows.

DWORD WINAPI CardDeriveKey(

\_\_in PCARD\_DATA *pCardData*,

\_\_inout PCARD\_DERIVE\_KEY *pAgreementInfo*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pAgreementInfo* Information that is related to the request for a derived key.

Output:

*pAgreementInfo* Information that is related to the response for a derived key.

Return value Zero on success; otherwise, nonzero.

The pAgreementInfo parameter is formatted as a CARD\_DERIVE\_KEY structure.

#define CARD\_DERIVE\_KEY\_VERSION\_2 2

#define CARD\_RETURN\_KEY\_HANDLE 0x1000000

typedef struct \_CARD\_DERIVE\_KEY

{

DWORD dwVersion;

// If CARD\_BUFFER\_SIZE\_ONLY is passed then the card module

// should return only the size of the resulting key in

// cbDerivedKey

DWORD dwFlags;

LPWSTR pwszKDF;

BYTE bSecretAgreementIndex;

PVOID pParameterList;

PBYTE pbDerivedKey;

DWORD cbDerivedKey;

// The following parameter can be used by the card to determine

// key derivation material and to pass back a symmetric key

// handle

// as a result of the key derivation algorithm

LPWSTR pwszAlgId;

DWORD dwKeyLen;

CARD\_KEY\_HANDLE hKey;

} CARD\_DERIVE\_KEY, \*PCARD\_DERIVE\_KEY;

If the **dwVersion** member is set to CARD\_DERIVE\_KEY\_VERSION\_2 and the CARD\_RETURN\_KEY\_HANDLE flag is set in *dwFlags*, the minidriver should return a key handle in the **hKey** member instead of returning the key material through the **pbDerivedKey** and **cbDerivedKey** members.

Input: (as supplied by KSP)

The following members must be set in the *pAgreementInfo* parameter and verified by the **CardDeriveKey** function.

**dwVersion** Represents the revision of the **CardDeriveKey** functionality. The current version is 2. The current version is defined by CARD\_DERIVE\_KEY\_VERSION.

**dwFlags** Required to be zero, KDF\_USE\_SECRET\_AS\_HMAC  
\_KEY\_FLAG, CARD\_RETURN\_KEY\_HANDLE ,or CARD\_BUFFER\_SIZE\_ONLY.

**pwszKDF** A string that indicates the KDF to be used. This is set to the KDF that the client requests. These KDFs are defined in *Bcrypt.h*. The following is a list of possible KDFs:

BCRYPT\_KDF\_HASH

BCRYPT\_KDF\_HMAC

BCRYPT\_KDF\_TLS\_PRF

BCRYPT\_KDF\_SP80056A\_CONCAT

For detailed information, see the “[CNG Reference](http://msdn.microsoft.com/en-us/library/aa376214(VS.85).aspx)” documentation on MSDN. If a card minidriver does not implement the requested KDF, SCARD\_E\_INVALID\_PARAMETER should be returned.

**pParameterList** Contains the optional list of parameters to the key derivation algorithm. Type and number of parameters are determined and must be compatible by the key derivation function that the pwszKDF parameter selects. For information about acceptable parameters for a KDF, see the “[CNG Reference](http://msdn.microsoft.com/en-us/library/aa376214(VS.85).aspx)” documentation on MSDN.

If a card minidriver does not recognize one of the parameters or that parameter is invalid for the specified KDF, SCARD\_E\_INVALID\_PARAMETER should be returned.

For hash-based KDFs, such as BCRYPT\_KDF\_HASH or BCRYPT\_KDF\_HMAC, the KDF\_HASH\_ALGORITHM may be NULL. In this situation, the minidriver must use a default hash algorithm. We recommend that the minidriver use the SHA-1 algorithm as documented in the CNG documentation for the **NCryptDerivekey** function.

**dwAlgId** A value that identifies the algorithm to be used to derive the key. Possible values are as follows:

* CARD\_3DES\_112\_ALGORITHM
* CARD\_3DES\_ALGORITHM
* CARD\_AES\_ALGORITHM

**dwkeyLen** Length, in bits, of the derived key. The possible values should not differ from the values that the CARD\_KEY\_SIZES structure returned when **CardGetAlgorithmProperty** is called for CP\_CARD\_KEYSIZES of the desired algorithm.

Output:

The following members of the *pAgreementInfo* parameter must be set on a successful call.

**pbDerivedKey** This is the buffer that contains the binary data of the derived key. The caller is responsible for calling the appropriate memory management function to allocate and deallocate this buffer.

To obtain the size of the required buffer for key derivation, the caller calls **CardDeriveKey** with CARD\_BUFFER\_SIZE\_ONLY set in *dwFlags*. In this situation, the minidriver must return the size of the buffer in *cbDerivedKey*.

**cbDerivedKey** Specifies the maximum length, in bytes, of the **pbDerivedKey** buffer.

**hKey** When the **dwVersion** member is set to CARD\_DERIVE\_KEY\_VERSION\_2 and the CARD\_RETURN\_KEY\_HANDLE flag is set in the **dwFlags** member, the minidriver should return a key handle in the **hKey** member instead of returning the key material in the **pbDerivedKey** and **cbDerivedKey** members.

Comments:

If the **dwFlags** member is set to CARD\_BUFFER\_SIZE\_ONLY, the minidriver must return the required size of the buffer in the **cbDerivedKey** member. The caller must use this data to allocate a buffer for the derived key, which is passed in through the **pbDerivedKey** member on subsequent calls to **CardDeriveKey**.

### **CardDestroyDHAgreement**

Description:

The **CardDestroyDHAgreement** function removes an agreed secret from the card. For RSA-only card minidrivers, this entry point is not defined and is set to NULL in the CARD\_DATA structure that was returned from **CardAcquireContext**.

DWORD WINAPI CardDestroyDHAgreement(

\_\_in PCARD\_DATA *pCardData*,

\_\_in BYTE *bSecretAgreementIndex*,

\_\_in DWORD *dwFlags*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*bSecretAgreementIndex* The index of the agreement to destroy.

*dwFlags* Reserved (must be zero).

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

SCARD\_E\_INVALID\_PARAMETER should be returned if *bSecretAgreementIndex* does not contain a valid agreement.

SCARD\_E\_INVALID\_PARAMETER should be returned when a nonzero *dwFlags* parameter is passed.

SCARD\_W\_SECURITY\_VIOLATION should be returned if **CardDestroyDHAgreement** is called without authenticating to the card first.

### **CardSignData**

Description:

The **CardSignData** function signs a block of unpadded data. This entry either performs padding on the card or pads the data by using the PFN\_CSP\_PAD\_DATA callback. All card minidrivers must support this entry point.

DWORD WINAPI CardSignData(

\_\_in PCARD\_DATA *pCardData*,

\_\_in PCARD\_SIGNING\_INFO *pInfo*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pInfo* Structure that contains data to be signed, which is allocated by the Base CSP/KSP.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

The Base CSP/KSP performs the hashing operation on the data before passing it to **CardSignData** for signature.

The *pInfo* parameter is formatted as a CARD\_SIGNING\_INFO structure, which is defined in *Cardmod.h*.

#define CARD\_PADDING\_INFO\_PRESENT 0x40000000

#define CARD\_BUFFER\_SIZE\_ONLY 0x20000000

#define CARD\_PADDING\_NONE 0x00000001

#define CARD\_PADDING\_PKCS1 0x00000002

#define CARD\_PADDING\_PSS 0x00000004

// CARD\_SIGNING\_INFO\_BASIC\_VERSION is provided for those

// applications do not intend to support passing in the

// pPaddingInfo structure

#define CARD\_SIGNING\_INFO\_BASIC\_VERSION 1

// Function: CardSignData

//

// Purpose: Sign input data using a specified key

//

#define CARD\_SIGNING\_INFO\_CURRENT\_VERSION 2

typedef struct \_CARD\_SIGNING\_INFO

{

DWORD dwVersion;

BYTE bContainerIndex;

// See dwKeySpec constants

DWORD dwKeySpec;

// If CARD\_BUFFER\_SIZE\_ONLY flag is present then the card

// module should return only the size of the resulting

// key in cbSignedData

DWORD dwSigningFlags;

// If the aiHashAlg is non zero, then it specifies the algorithm

// to use when padding the data using PKCS

ALG\_ID aiHashAlg;

// This is the buffer and length that the caller expects to be signed.

// Signed version is allocated a buffer and put in cb/pbSignedData.

// That should be freed using PFN\_CSP\_FREE callback.

PBYTE pbData;

DWORD cbData;

PBYTE pbSignedData;

DWORD cbSignedData;

// The following parameters are new in version 2 of the

// CARD\_SIGNING\_INFO structure.

// If CARD\_PADDING\_INFO\_PRESENT is set in dwSigningFlags then

// pPaddingInfo will point to the BCRYPT\_PADDING\_INFO structure

// defined by dwPaddingType. Currently supported values are

// CARD\_PADDING\_PKCS1, CARD\_PADDING\_PSS and CARD\_PADDING\_NONE

LPVOID pPaddingInfo;

DWORD dwPaddingType;

} CARD\_SIGNING\_INFO, \*PCARD\_SIGNING\_INFO;

The **dwSigningFlags** member takes the same flag values as CryptSignHash, for example, CRYPT\_NOHASHOID.

When CARD\_PADDING\_INFO\_PRESENT is not set in dwSigningFlags, this is the basic version of the signing structure. (If this is not the basic verison of the signing structure, the minidriver should return ERROR\_REVISION\_MISMATCH.) The minidriver should only do PKCS1 padding and use the value in aiHashAlg.

When CARD\_PADDING\_INFO\_PRESENT is set in dwSigningFlags, this is the current version of the signing structure. (If this is not the current verison of the signing structure, the minidriver should return ERROR\_REVISION\_MISMATCH.) The minidriver should get the padding algorithm from dwPaddingType, get padding parameters from pPaddingInfo, and ignore the value set in aiHashAlg.

If **dwPaddingType** is CARD\_PADDING\_PKCS1, **pPaddingInfo** should point to a BCRYPT\_PKCS1\_PADDING\_INFO structure. If **dwPaddingType** is set to CARD\_PADDING\_PSS, **pPaddingInfo** should point to a BCRYPT\_PSS\_PADDING\_INFO structure.

The **aiHashAlg** member takes those values allowed by ALG\_ID from the HASH algorithm class. For a list of algorithm ID, see “[ALG\_ID](http://msdn.microsoft.com/en-us/library/aa375549.aspx)” on MSDN.

For maximum interoperability with applications, we recommend that the following algorithm identifiers be supported for the **aihashAlg** member:

* CALG\_TLS1PRF
* CALG\_MAC
* CALG\_SHA\_256
* CALG\_SHA\_384
* CALG\_SHA\_512
* CALG\_HASH\_REPLACE\_OWF
* CALG\_MD2, CALG\_MD4
* CALG\_MD5, CALG\_SHA
* CALG\_SHA1, CALG\_HUGHES\_MD5
* CALG\_HMAC
* CALG\_SSL3\_SHAMD5

If the **aiHashAlg** member is nonzero, it specifies the hash algorithm’s object identifier (OID) that is encoded in the PKCS padding. This padding is added to the hashed data to which the *pbData* parameter pointed. The card itself can add this padding, or the minidriver can request this padding to be added by using the PFN\_CSP\_PAD\_DATA function.

The algorithm identifier that the **pszAlgId** member specified in *pPaddingInfo* takes those values that are allowed by CNG for hash algorithm identifier. For a complete list of algorithm identifiers, see “[CNG Algorithm Identifiers](http://msdn.microsoft.com/en-us/library/aa375534(VS.85).aspx)” on MSDN.

For maximum interoperability with applications, we recommend that only the following algorithm identifiers be supported for **pszAlgId**:

* BCRYPT\_MD2\_ALGORITHM
* BCRYPT\_MD4\_ALGORITHM
* BCRYPT\_MD5\_ALGORITHM
* BCRYPT\_SHA1\_ALGORITHM
* BCRYPT\_SHA256\_ALGORITHM
* BCRYPT\_SHA384\_ALGORITHM
* BCRYPT\_SHA512\_ALGORITHM

Algorithms that the card does not support should result in **CardSignData** returning SCARD\_E\_UNSUPPORED\_FEATURE.

When an invalid or nonexistent *bContainerIndex* is passed in the CARD\_SIGNING\_INFO structure, an SCARD\_E\_NO\_KEY\_CONTAINER error code should be returned.

When an invalid value for *dwKeySpec* is passed (see either “[**CardCreateContainer**](#_CardCreateContainerCardCreateContai)” or “[**CardCreateContainerEx**](#_CardCreateContainerEx)**”** earlier in this specification), SCARD\_E\_INVALID\_PARAMETER should be returned. When the value for *dwKeySpec* is valid but not supported, SCARD\_E\_UNSUPPORTED\_FEATURE must be returned.

**Note:** If the card does not support on-card padding, the card minidrivers are not required to inspect the parameters. It is expected that they call into padding callback function (**pfnCspPadData**) under normal operating conditions.

We recommend supporting the CARD\_BUFFER\_SIZE\_ONLY flag, but this is optional. If supported, it helps reduce the amount of traffic to the card.

Card minidrivers that advertise that they are compatible with Version 5 must support both CARD\_SIGNING\_INFO\_BASIC\_VERSION and CARD\_SIGNING\_INFO\_CURRENT\_VERSION versions.

If the **dwVersion** member of the CARD\_SIGNING\_INFO structure has a value that is less than CARD\_SIGNING\_INFO\_CURRENT\_VERSION and the **dwVersion** member of *pCardData* is set to CARD\_DATA\_CURRENT\_VERSION, this function should return ERROR\_REVISION\_MISMATCH. In other words, if the minidriver is loaded for the latest version, CARD\_SIGNING\_INFO must have the latest version of the structure as well.

The input data to be signed is passed in little-endian format.

### **CardQueryKeySizes**

Description:

This function returns the public key sizes that are supported by the card in use.

DWORD WINAPI CardQueryKeySizes(

\_\_in PCARD\_DATA *pCardData*,

\_\_in DWORD *dwKeySpec*,

\_\_in DWORD *dwFlags*,

\_\_inout PCARD\_KEY\_SIZES *pKeySizes*

);

#define CARD\_KEY\_SIZES\_CURRENT\_VERSION 1

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*dwKeySpec* Type of key of interest: For allowed values see **CardCreateContainer**.

*dwFlags* Reserved—must be zero.

*pKeySizes* Pointer to CARD\_KEY\_SIZES structure.

Output:

*pKeySizes* Supported key sizes for the specified algorithm type.

Return value Zero on success; otherwise, nonzero.

Comments:

Key size information is returned in the following structure. For ECC, minimum, default, and maximum are a specific value. Increment is 1.

typedef struct \_CARD\_KEY\_SIZES

{

DWORD dwVersion; // version should be set by the caller

DWORD dwMinimumBitlen;

DWORD dwDefaultBitlen;

DWORD dwMaximumBitlen;

DWORD dwIncrementalBitlen;

} CARD\_KEY\_SIZES, \*PCARD\_KEY\_SIZES;

If *dwKeySpec* is undefined, the function should return SCARD\_E\_INVALID\_PARAMETER.

If *dwKeySpec* is defined but not supported by the card, the function should return SCARD\_E\_UNSUPPORTED\_FEATURE.

A card minidriver that supports read-only cards may support more key types than what the specific read only card has been provisioned with. In this case CardQueryKeySizes should succeed and return a corresponding CARD\_KEY\_SIZES structure for the supported key spec.

## Secure Key Injection

Secure Key Injection provides support for the encrypted transfer of sensitive material from a server application to a smart card through an untrusted client.

For Secure Key Injection to work properly, the following steps must occur:

1. Establishment of encryption keys:

* 1. Use shared symmetric keys between the server and the smart card on the client.
  2. Generate a temporary symmetric session key on the server and import it to the smart card. The session key must be encrypted by a public key that has the corresponding private key generated on the smart card[[2]](#footnote-2).
  3. Derive a session key from a shared symmetric key. For more information, see “[**GetSharedKeyHandle**](#_CardGetSharedKeyHandle)” later in this specification.
  4. Use DH key derivation.

2. Encryption of data on the server:

1. Data could be authentication data such as a PIN.
2. Data could be an asymmetric key pair such as RSA/ECC.

3. Decryption of data in the smart card on the client.

Figure 4 shows a server application that generates a key and then securely transfers the key across a trust boundary to the client. After the key is received, the client imports it to the smart card. As the final step, the key is imported into the CA for archival. An encrypted channel should exist between the server application and the smart card, and the client application/minidriver should be unable to access the encrypted data.

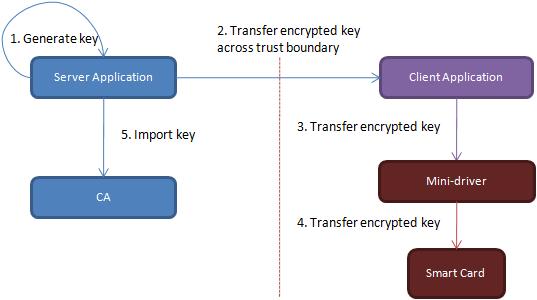


Figure 4: Overview of server-client interaction during a secure key injection with smart cards

To encrypt the key in step 2, the server and the smart card require a shared symmetric key.

To accommodate existing cards that use a proprietary format when they perform secure key injection, the minidriver can be loaded on the server-side without the card being present. The minidriver formats the message and then finally encrypts it, which allows the same minidriver that runs on the client to decrypt the message.

Figure 5 provides an overview of server/client key archival with minidrivers,

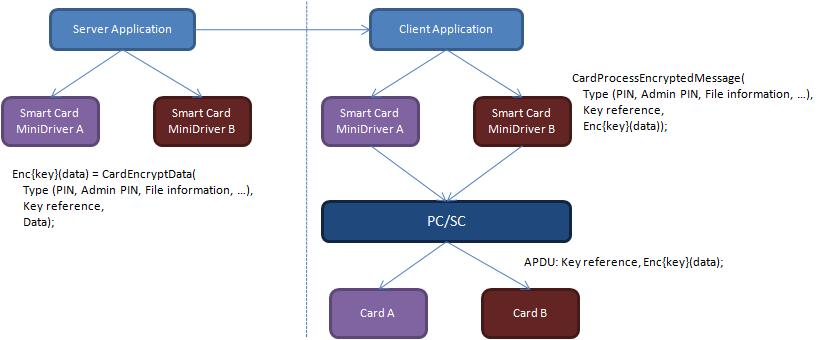


Figure 5: Overview of server/client key archival with minidrivers

“[Appendix B](#_Appendix_B._Use)” contains a use case scenario that uses API calls to perform a secure key injection.

### Defines and Structures

In addition to new card properties, new defines, structures and functions are introduced in version 7 of the smart card minidriver API.

For more information about the new properties that have been added, see “[**CardGetProperty**](#_CardGetProperty)” earlier in this specification.

#### Card Key Handle

When dealing with symmetric keys, CARD\_KEY\_HANDLE should be used to pass around the key handle.

typedef ULONG\_PTR CARD\_KEY\_HANDLE;

#### No Card Mode

To facilitate server applications that format and encrypt data by using the same minidriver that is installed on the untrusted client, **CardAcquireContext** can be called in a mode that does not require the card to be present. This mode is enabled by setting the following flag in the *dwFlags* parameter of **CardAcquireContext** .

#define CARD\_SECURE\_KEY\_INJECTION\_NO\_CARD\_MODE 0x1

This setting instructs **CardAcquireContext** not to expect any card to be in the reader. This means that the ATR fields in the PCARD\_DATA are not filled and **hSCard** and **hSCardCtx** are set to zero.

When this flag is set, the minidriver can accept only the following function calls:

* **MDImportSessionKey**
* **MDEncryptData**
* **CardGetSharedKeyHandle**
* **CardGetAlgorithmProperty**
* **CardDestroyKey**
* **CardGetKeyProperty**
* **CardSetKeyProperty**
* **CardProcessEncryptedData**

#### CARD\_ENCRYPTED\_DATA

Description:

This structure is used by the minidriver to return encrypted data to the calling application during the **MDEncryptData** function call.

**Structure:**

typedef struct \_CARD\_ENCRYPTED\_DATA {

PBYTE pbEncryptedData;

DWORD cbEncryptedData;

} CARD\_ENCRYPTED\_DATA, \*PCARD\_ENCRYPTED\_DATA;

Members:

**cbEncryptedData** The size, in bytes, of the encrypted data in the *pbEncryptedData* buffer.

**pbEncryptedData** The address of a buffer that contains the encrypted data. *cbEncryptedData* contains the size of this buffer.

Comments:

For more information on how to encrypt data, see ”[**MDEncryptData**](#_MDEncryptData)” later in this specification.

#### CARD\_IMPORT\_KEYPAIR

Description:

This structure is used by the minidriver to define the attributes of the key BLOB to be processed by secure key injection calls.

Structure:

#define CARD\_IMPORT\_KEYPAIR\_CURRENT\_VERSION 7

typedef struct \_CARD\_IMPORT\_KEYPAIR

{

DWORD dwVersion;

BYTE bContainerIndex;

PIN\_ID PinId;

DWORD dwKeySpec;

DWORD dwKeySize;

DWORD cbInput;

BYTE pbInput[0];

} CARD\_IMPORT\_KEYPAIR, \*PCARD\_IMPORT\_KEYPAIR;

Members:

**dwVersion** The version of the structure. The current version is 7.

**bContainerIndex** Index number for the container to be created.

**PinID** PIN Identifier for the key to be imported.

**dwKeySpec** AT\_ECDH\_P256, AT\_ECDH\_P384, AT\_ECDH\_P521, AT\_ECDSA\_P256, AT\_ECDSA\_P384, or AT\_ECDSA\_P521, specify ECC keys.

AT\_SIGNATURE or AT\_KEYEXCHANGE specify RSA keys that can be used on dual-mode cards.

**dwKeySize** The size, in bits, of the key material.

**cbInput** The size, in bytes, of the key BLOB in the **pbInput** buffer.

**pbInput** Zero-sized byte array that points to the end of the structure. The key BLOB that is to be imported should be appended at this address. The **cbInput** member contains the size of this key BLOB byte array.

Comments:

See **CardCreateContainer** for general information on key containers.

If the target container already exists, it is overwritten by the new one. The new container always contains a valid key if the call succeeds. If a wrong value for **bContainerIndex** is passed (invalid or nonexistent), a SCARD\_E\_NO\_KEY\_CONTAINER return value should be returned.

Imported key material is passed in “private key BLOB,” which is typically returned from **CryptExportKey**. For more information, see “[CryptExportKey Function](http://msdn.microsoft.com/en-us/library/aa379931(VS.85).aspx)“ and “[Base Provider Key BLOBs](http://msdn.microsoft.com/en-us/library/aa375601(VS.85).aspx)” on MSDN. RSA keys comply with CAPI key BLOB format.

If a **dwKeySpec** member is invalid or undefined, a return value of SCARD\_E\_INVALID\_PARAMETER should be returned. If the **dwKeySpec** value is defined but not supported, a return value of SCARD\_E\_UNSUPPORTED\_FEATURE should be returned.

Only users can create containers. Both administrators and users should be able to obtain information and delete containers. If an administrator attempts to create a container, the function should return SCARD\_W\_SECURITY\_VIOLATION.

When this data structure is passed to any of the secure key injection functions as input buffer, the size of the buffer that the function call specifies should include both structure and key BLOB buffer.

#### CARD\_CHANGE\_AUTHENTICATOR

Description:

This structure is used by the minidriver to facilitate changing the PIN by using secure key injection function calls.

Structure:

#define CARD\_CHANGE\_AUTHENTICATOR\_CURRENT\_VERSION 7

typedef struct \_CARD\_CHANGE\_AUTHENTICATOR

{

DWORD dwVersion;

DWORD dwFlags;

PIN\_ID dwAuthenticatingPinId;

DWORD cbAuthenticatingPinData;

PIN\_ID dwTargetPinId;

DWORD cbTargetData;

DWORD cRetryCount;

BYTE pbData[0];

} CARD\_CHANGE\_AUTHENTICATOR, \*PCARD\_CHANGE\_AUTHENTICATOR;

Members:

**dwVersion** The version of the structure. The current version is 7.

**dwFlags** A set of flags that specify the operation to be performed with the data. Current flags settings indicate whether this data is used for a PIN change or unblock operation.

**dwAuthenticatingPinId** PIN identifier to be authenticated.

**cbAuthenticatingPinData** Byte count of the PIN data.

**dwTargetPinId** PIN identifier to be updated.

**cbTargetData** Byte count of the new PIN data.

**cRetrycount** The count of times that a wrong PIN does not result in a blocked card.

**pbData** Both PIN data and new PIN data are appended at this address. PIN data is saved at *pbDatapbData*, size is specified by *cbAuthenticatingPinData*. New PIN data is saved at (*pbDatapbData* + *cbAuthenticatingPinData)*, and size is specified by *cbTargetData*.

Comments:

This structure must be used in all situations in which the authenticator is to be changed securely by using the secure key injection API.

The current PIN data is accessed within the **pbData** buffer at offset 0, and its size is specified by the **cbAuthenticatingPinData** member.

The new PIN data is accessed within the **pbData** buffer at offset **cbAuthenticatingPinData**, and its size is specified by the **cbTargetData** member.

The allowed values for **dwAuthenticatingPinId**are ROLE\_USER, ROLE\_ADMIN or 3 through 7. For any other **dwAuthenticatingPinId** value, the function should return SCARD\_E\_INVALID\_PARAMETER.

For an explanation of **dwFlags**, see “[**CardChangeAuthenticatorEx**](#_CardChangeAuthenticatorEx)” earlier in this specification.

If changing the authenticator or the form of the new authenticator does not comply with policy, implementations that enforce policies about the authenticator (such as, PIN policies) should return SCARD\_E\_INVALID\_PARAMETER.

When the call is used to change a PIN, the successful completion should leave the card in an authenticated state. If the call is used to unblock a PIN, the successful completion should leave the card in a de-authenticated state for both the unblocked PIN and the authenticating PIN.

When this data structure is passed to any of the secure key injection functions as an input buffer, the size of the buffer that is specified in the function call should include the following:

* The size of the CARD\_CHANGE\_AUTHENTICATOR structure.
* The current PIN data as specified by the **cbAuthenticatingPinData** member.
* The new PIN data as specified by the **cbTargetData** member.

#### CARD\_CHANGE\_AUTHENTICATOR\_RESPONSE

Description:

This structure is used by the minidriver when it returns a response for a secure PIN change operation by using the secure key injection function calls.

Structure:

#define CARD\_CHANGE\_AUTHENTICATOR\_\_RESPONSE\_CURRENT\_VERSION 7

typedef struct \_CARD\_CHANGE\_AUTHENTICATOR\_RESPONSE

{

DWORD dwVersion;

DWORD cAttemptsRemaining;

} CARD\_CHANGE\_AUTHENTICATOR\_RESPONSE, \*PCARD\_CHANGE\_AUTHENTICATOR\_RESPONSE;

Members:

**dwVersion** The version of the structure. The current version is 7.

**cAttemptsRemaining** When the function returns, this should contain the count of remaining times that a wrong PIN does not result in a blocked card.

Comments:

This structure is used to transfer data back from the card for all situations in which the authenticator is to be set securely. The only data that is currently returned is the number of authentication attempts that remain for the authentication PIN.

#### CARD\_AUTHENTICATE

Description:

This structure is used by the minidriver to allow for a remote secure key injection application to securely authenticate to the card that is connected to an un-trusted client.

Structure:

#define CARD\_AUTHENTICATE\_CURRENT\_VERSION 7

typedef struct \_CARD\_AUTHENTICATE

{

DWORD dwVersion;

DWORD *dwFlags*;

PIN\_ID PinId;

DWORD cbPinData;

BYTE pbPinData[0];

} CARD\_AUTHENTICATE, \*PCARD\_AUTHENTICATE;

Members:

**dwVersion** The version of the structure. The current version is 7.

**dwFlags** A set of flags that specify the meaning of the data within the structure. For more information, see “Comments.”

**PinId** The PIN identifier to be authenticated.

**cbPinData** Byte count of the data in the **pbPinData** buffer.

**pbPinData** A zero-sized byte array that which points to the end of the structure. A byte array that contains PIN information should be attached at this address. *cbPinData* contains the length of this byte array.

Comments:

For the expected behavior and explanation of **dwFlags,** see “[**CardAuthenticatePin**](#_CardAuthenticatePin)” earlier in this specification.

The secure version of PIN authentication is not valid for the external PIN type, which are PINs that are stored on a device that is connected to the computer.

If the function returns 0 (success), the user can perform any action that requires **PinId** to be authenticated. This state persists until one of the following occurs:

* Either **CardDeauthenticate** or **CardDeauthenticateEx** is called.
* The card is reset through the Winscard API.
* The card loses power.

**Note:** This does not apply if CARD\_AUTHENTICATE\_GENERATE\_SESSION\_PIN is specified in **dwFlags**.

If this data structure is passed to any of the secure key injection functions as an input buffer, the size of the buffer that is specified in the function call should include both the size of the structure and the length of the PIN data.

In addition, **pbPinData** should point to location of the PIN data within the input buffer. This data should immediately follow the CARD\_AUTHENTICATE structure within the input buffer.

#### CARD\_AUTHENTICATE\_RESPONSE

Description:

This structure is used by the minidriver to return status information to a secure key injection application that uses CARD\_AUTHENTICATE to perform remote authentication to the card.

Structure:

#define CARD\_AUTHENTICATE\_RESPONSE\_CURRENT\_VERSION 7

typedef struct \_CARD\_AUTHENTICATE\_RESPONSE

{

DWORD dwVersion;

DWORD cbSessionPin;

DWORD cAttemptsRemaining;

BYTE pbSessionPin[0];

} CARD\_AUTHENTICATE\_RESPONSE, \*PCARD\_AUTHENTICATE\_RESPONSE;

Members:

**dwVersion** The version of the structure. The current version is 7.

**cbSessionPin** Byte count of the session PIN data. This member is set by the minidriver if a session PIN is returned.

**cAttemptsRemaining** A count of the times that an incorrect PIN was presented to the card. If this count becomes zero, the PIN is locked.

**pbSessionPin** A zero-sized byte array that points to the end of the structure. If the minidriver wants to return a session PIN, a buffer that contains a session PIN should be appended here. *cbSessionPin* contains the length of this byte array.

Comments:

This structure is used to transfer data back from a secure authentication operation. Currently, the card can pass back a session PIN in addition to the number of authentication attempts that remain in the card following a failed authentication operation.

If a session PIN is returned, it may be encrypted. The decryption of the encrypted session pin can be performed by calling **CardProcessEncryptedData**.

If this data structure is passed to any of the secure key injection functions as input buffer, the size of the buffer that is specified in the function call should include both the size of the structure and the length of the session PIN data.

In addition, if a session PIN is returned, **pbPinData** should point to location of the session PIN data within the input buffer. This data should immediately follow the CARD\_AUTHENTICATE\_RESPONSE structure within the input buffer. Also, **cbSessionPin** must be the nonzero value of the length of the PIN data.

**Note:** If a session key is not returned, the minidriver must set **cbSessionPin** to zero.

### Server Functions

#### MDImportSessionKey

Description:

The **MDImportSessionKey** function imports a temporary session key to the card minidriver and returns a key handle to the caller.

DWORD WINAPI MDImportSessionKey(

\_\_in PCARD\_DATA *pCardData*,

\_\_in LPCWSTR *pwszBlobType*,

\_\_in LPCWSTR *pwszAlgId*,

\_\_out PCARD\_KEY\_HANDLE *phKey*,

\_\_in\_bcount(*cbInput*) PBYTE *pbInput*,

\_\_in DWORD *cbInput*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pwszBlobType* A pointer to a null-terminated Unicode string. This string identifies the type of BLOB that is contained in the *pbInput* buffer. For more information, see the following “Comments.”

*pwszAlgId* A pointer to a null-terminated Unicode string. This string identifies the algorithm to be used to encrypt the key. For more information, see the description of CP\_ENUM\_ALGORITHMS in “[**CardGetProperty**](#_CardGetPropertyCardGetProperty)” earlier in this specification.

*phKey* A pointer to a CARD\_KEY\_HANDLE that receives the handle of the imported key. This handle is used in subsequent function calls that require the key, such as **CardProcessEncryptedData**. The caller must release the handle when it is no longer needed by calling **CardDestroyKey**.

*pbInput* The address of a buffer that contains the key BLOB to be imported. The *cbInput* contains the size of this buffer. The *pwszBlobType* parameter specifies the type of key BLOB that this buffer contains. *cbInput* The size, in bytes, of the key BLOB in the *pbInput* buffer.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

If the card minidriver does not support the import of temporary symmetric keys, it should return SCARD\_E\_UNSUPPORTED\_FEATURE.

The *pwszBlobType* parameter can currently have only one value as described in the following table.

| **String value** | **Description** |
| --- | --- |
| “KeyDataBlob” | The *pbInput* parameter is a pointer to a buffer that contains a BCRYPT\_KEY\_DATA\_BLOB\_HEADER structure. The key BLOB data immediately follows the BCRYPT\_KEY\_DATA\_BLOB\_HEADER structure in the buffer.  For more information, see “[BCRYPT\_KEY\_DATA\_BLOB\_HEADER Structure](http://msdn.microsoft.com/en-us/library/aa375524(VS.85).aspx)” on MSDN. |

If *pwszBlobType* contains an invalid or undefined value, the function should return SCARD\_E\_INVALID\_PARAMETER. For legacy applications and cards that use the minidriver interface for secure key injection, the minidriver can accept proprietary BLOB types.

If *pwszAlgId* contains an invalid or undefined value, the function should return SCARD\_E\_INVALID\_PARAMETER. If the *pwszAlgId* value is defined but not supported, the function should return SCARD\_E\_UNSUPPORTED\_FEATURE. For legacy cards, the minidriver can support other algorithms.

When the function returns, the *phKey* parameter should contain a handle to the imported key. If the parameter does not contain a valid pointer, the function should return SCARD\_E\_INVALID\_PARAMETER. The key is expected to be valid until either **CardDestroyKey** is called or the card is removed from the reader.

#### MDEncryptData

Description:

The **MDEncryptData** function uses a key handle to encrypt data with a symmetric key. The data is encrypted in a format that the smart card supports.

DWORD WINAPI MDEncryptData(

\_\_in PCARD\_DATA *pCardData*,

\_\_in CARD\_KEY\_HANDLE *hKey*,

\_\_in LPCWSTR *pwszSecureFunction*,

\_\_in\_bcount(cbInput) PBYTE *pbInput*,

\_\_in DWORD *cbInput*,

\_\_in DWORD *dwFlags*,

\_\_deref\_out\_ecount(\**pcEncryptedData*)   
 PCARD\_ENCRYPTED\_DATA \**ppEncryptedData*,

\_\_out PDWORD *pcEncryptedData*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*hKey* The handle of the cryptographic key that is used to encrypt the data.

*pwszSecureFunction* A pointer to a null-terminated Unicode string that contains the name of the data structure to be encrypted. For more information, see the following “Comments.”

*pbInput* A byte pointer to the buffer that contains the data.

*cbInput* The length, in bytes, of the data buffer.

*dwFlags* A set of flags that specify options for the encryption operation. Currently, only one flag is supported. For more information, see the following “Comments.”

*ppEncryptedData* A pointer to an array of CARD\_ENCRYPTED\_DATA structures. The buffer that contains the array is allocated by the minidriver and returned to the calling application. The application is responsible for freeing the buffer.

*pcEncryptedData* A pointer to a DWORD value that contains the number of returned encrypted data BLOBs.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

If the card minidriver does not support encrypting data for secure transmission, the function should return SCARD\_E\_UNSUPPORTED\_FEATURE.

The *dwFlags* parameter is used to specify flag settings for optional parameters for the encryption operation. Currently, the only allowed flag is CARD\_BLOCK\_PADDING, which specifies that the encrypted data should be padded by using PKCS #5. For more information, see the description of CP\_PADDING\_SCHEMES in “[**CardGetProperty**](#_CardGetPropertyCardGetProperty)” earlier in this specification.

If *dwFlags* contains an invalid or undefined value, the function should return SCARD\_E\_INVALID\_PARAMETER.

If an unsupported *pwszSecureFunction* value is passed to **MDEncryptData**, the function should return SCARD\_E\_INVALID\_PARAMETER.

**Note:** The minidriver may choose to define and support optional custom secure functions that are not defined in the specification.

The format of *pbInput* depends on the value of the *pwszSecureFunction* parameter. The following table describes the different supported values for *pwszSecureFunction* along with the corresponding format for *pbInput*:

| ***pwszSecureFunction* value** | ***pbInput* value** |
| --- | --- |
| CSF\_IMPORT\_KEYPAIR | The data contains a structure of type [CARD\_IMPORT\_KEYPAIR](#_CARD_IMPORT_KEYPAIR). |
| CSF\_CHANGE\_AUTHENTICATOR | The data contains a structure of type [CARD\_CHANGE\_AUTHENTICATOR](#_CARD_CHANGE_AUTHENTICATOR). |
| CSF\_AUTHENTICATE | The data contains a structure of type [CARD\_AUTHENTICATE](#_CARD_AUTHENTICATE). |

The function should allocate an array of CARD\_ENCRYPTED\_DATA structures and return them in the *ppEncryptedData* pointer.

This function can be called only when CARD\_SECURE\_KEY\_INJECTION  
\_NO\_CARD\_MODE is passed to **CardAcquireContext**.

If the appropriate properties are not set on the *hKey* key handle before the call to **MDEncryptData**, the function should return SCARD\_E\_INVALID\_PARAMETER.

### Shared Functions

#### CardGetSharedKeyHandle

Description:

The **CardGetSharedKeyHandle** function returns a session key handle to the caller.

**Note:** The manner in which this session key has been established is outside the scope of this specification. For example, the session key could be established by either a permanent shared key or a key derivation algorithm that has occurred before the call to CardGetSharedKeyHandle.

DWORD WINAPI CardGetSharedKeyHandle(

\_\_in PCARD\_DATA *pCardData*,

\_\_in\_bcount(cbInput) PBYTE *pbInput*,

\_\_in DWORD *cbInput*,

\_\_deref\_opt\_out\_bcount(\**pcbOutput*)

PBYTE \**ppbOutput*,

\_\_out\_opt PDWORD *pcbOutput*,

\_\_out PCARD\_KEY\_HANDLE *phKey*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pbInput* A byte pointer to the optional data buffer that contains the input data.

*cbInput* The length, in bytes, of the optional data buffer.

*ppbOutput* A byte pointer to the optional data buffer that receives the potential data that is returned from the minidriver. The data buffer is allocated by the minidriver and freed by the caller.

*pcbOutput* An optional pointer to a DWORD value that receives the actual data length that is returned in *ppbOutput*.

*phKey* A pointer to a CARD\_KEY\_HANDLE. This handle is used in subsequent functions that require the key, such as **CardProcessEncryptedData**. The handle must be released when it is no longer needed by calling **CardDestroyKey**.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

If the card minidriver does not support the return of shared key handles, the function should return SCARD\_E\_UNSUPPORTED\_FEATURE.

The optional input (*pbInput*) and output (*ppbOutput*) data buffers can be used to provide key derivation data to and from the minidriver. For example, this would work well with a key establishment protocol, such as the protocol that is specified by GlobalPlatform.

If the *phKey* parameter is NULL or does not contain a valid pointer, the function should return SCARD\_E\_INVALID\_PARAMETER.

If this function succeeds, the *phKey* parameter should hold one of the following values:

* A NULL value, which indicates that the key establishment protocol has not completed. In this situation, the minidriver requires another round trip of data before it can establish the shared key. For example, this could be used with the optional input/output buffers as part of a key derivation algorithm.
* A handle to the shared key after the key establishment protocol is complete.

The key is expected to be valid until **CardDestroyKey** is called or the card is removed from the reader.

The context of this key handle is determined by whether the CARD\_SECURE\_KEY\_INJECTION\_NO\_CARD\_MODE flag was set in the *dwFlags* parameter when **CardAcquireContext** was called. If this flag was not set, the key handle translates to a key on the card. If this flag was set, the key handle might be a handle to a predefined key on the server.

#### CardDestroyKey

Description:

The **CardDestroyKey** function releases a temporary key on the card. The card should delete all of the key material that is associated with that key handle.

DWORD WINAPI CardDestroyKey(

\_\_in PCARD\_DATA *pCardData*,

\_\_in CARD\_KEY\_HANDLE *hKey*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*hKey* Key handle that is returned by **CardImportSessionKey**, **MDImportSessionKey** or **CardGetSharedKeyHandle**

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

This function deletes the key material that is associated with the key handle.

The returned status code should indicate success if the key handle was valid and the key material was successfully deleted.

If **CardDestroyKey** is called with an invalid key handle, the function should return SCARD\_E\_INVALID\_HANDLE.

If **CardDestroyKey** cannot delete the key material, the function should return SCARD\_E\_INVALID\_HANDLE.

#### CardGetAlgorithmProperty

Description:

This function can be used to get properties for a cryptographic algorithm.

DWORD WINAPI CardGetAlgorithmProperty (

\_\_in PCARD\_DATA *pCardData*,

\_\_in LPCWSTR *pwszAlgId*,

\_\_in LPCWSTR *pwszProperty*,

\_\_out\_bcount\_part\_opt(*cbData*, \**pdwDataLen*)

PBYTE *pbData*,

\_\_in DWORD *cbData*,

\_\_out PDWORD *pdwDataLen*,

\_\_in DWORD *dwFlags*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*pwszAlgId* A pointer to a null-terminated Unicode string. This string identifies the algorithm whose property is queried. For more information, see the description of CP\_ENUM\_ALGORITHMS in “[**CardGetProperty**](#_CardGetPropertyCardGetProperty)” earlier in this specification.

*pwszProperty* A pointer to a null-terminated Unicode string. This string identifies the name of the property to be retrieved. For more information, see the following “Comments.”

*pbData* A byte pointer to a data buffer that receives the property data.

*cbData* The maximum length, in bytes, of the buffer to which *pbData* points.

*pdwDataLen* A pointer to a DWORD variable that receives the actual returned data length.

*dwFlags* A set of flags that specify options for the operation. Currently, no flags are defined for this function and the value must be zero.

Output:

Return value Zero on success; nonzero on failure.

Comments:

If *dwFlags* has a nonzero value, the function should return SCARD\_E\_INVALID\_PARAMETER.

If *pwszAlgId* contains an invalid or undefined value, the function should return SCARD\_E\_INVALID\_PARAMETER. If the *pwszAlgId* value is defined but not supported, the function should value SCARD\_E\_UNSUPPORTED\_FEATURE. For legacy cards the minidriver can support other algorithms.

If p*wszProperty* contains an unsupported value, the function should return SCARD\_E\_INVALID\_PARAMETER.

**Note:** Any minidriver may choose to define and support optional custom properties that are not defined in this specification.

The format of *pbData* depends on the value of the p*wszProperty* parameter. The following table describes the supported values for p*wszProperty* along with the corresponding format for *pbData*.

| **p*wszProperty* value** | ***pbData* type** | ***pbData* value** |
| --- | --- | --- |
| CP\_CARD\_KEYSIZES | DWORD | The return data contains a structure of the following format. This format describes the different key length values that are available for the cryptographic algorithm as specified by p*wszAlgId*:  typedef struct \_CARD\_KEY\_SIZES  {  DWORD dwVersion;  DWORD dwMinimumBitlen;  DWORD dwDefaultBitlen;  DWORD dwMaximumBitlen;  DWORD dwIncrementalBitlen;  ) CARD\_KEY\_SIZES, \*PCARD\_KEY\_SIZES;  A card minidriver that supports read-only cards may support more key types than what the specific read only card has been provisioned with. In this case the returned value should include all the supported key spec.  **Note:** This property is read only. |

#### CardGetKeyProperty

Description:

This function is used to query the properties of a key.

DWORD WINAPI CardGetKeyProperty(

\_\_in PCARD\_DATA *pCardData*,

\_\_in CARD\_KEY\_HANDLE *hKey*,

\_\_in LPCWSTR *pwszProperty*,

\_\_out\_bcount\_part\_opt(*cbData*, \**pdwDataLen*) PBYTE *pbData*,

\_\_in DWORD *cbData*,

\_\_out PDWORD *pdwDataLen*,

\_\_in DWORD *dwFlags*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*hKey* The handle of the cryptographic key whose property is to be queried.

*pwszProperty* A pointer to a null-terminated Unicode string that contains the name of the property to be queried. For more information, see the following “Comments.”

*pbData* A byte pointer to that data buffer that receives the property data.

*cbData* The maximum length, in bytes, of the buffer pointed to by *pbData*.

*pdwDataLen* A pointer to a DWORD variable that receives the returned length, in bytes, of the data in the buffer to which by *pbData* points.

*dwFlags* A set of flags that specify options for the operation. Currently, no flags are defined for this function and the value must be zero.

Output:

Return value Zero on success; nonzero on failure.

Comments:

If *dwFlags* has a nonzero value, the function should return SCARD\_E\_INVALID\_PARAMETER.

If p*wszProperty* contains an unsupported value, the function should return SCARD\_E\_INVALID\_PARAMETER.

**Note:** Any minidriver may choose to define and support optional custom properties that are not defined in this specification.

The format of *pbData* depends on the value of the p*wszProperty* parameter. The following table describes the supported values for p*wszProperty* along with the corresponding format for *pbData*.

| **p*wszProperty* value** | ***pbData* type** | ***pbData* value** |
| --- | --- | --- |
| CKP\_BLOCK\_LENGTH | DWORD | The returned data contains a DWORD value that contains the block length of the cipher.  **Note:** This property is read only. |

#### CardSetKeyProperty

Description:

This function is used to set the properties of a key.

DWORD WINAPI CardSetKeyProperty(

\_\_in PCARD\_DATA *pCardData*,

\_\_in CARD\_KEY\_HANDLE *hKey*,

\_\_in LPCWSTR *pwszProperty*,

\_\_in\_bcount(*cbInput*) PBYTE *pbInput*,

\_\_in DWORD *cbInput*,

\_\_in DWORD *dwFlags*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*hKey* The handle of the cryptographic key whose property is to be set.

*pwszProperty* A pointer to a null-terminated Unicode string that contains the name of the property to be set. For more information, see the following “Comments.”

*pbInput* A byte pointer to data buffer that contains the property data.

*cbInput* The length, in bytes, of the data in the buffer to which *pbInput* points*.*

*dwFlags* A set of flags that specify options for the operation. Currently, no flags are defined for this function and the value must be zero.

Output:

Return value Zero on success; nonzero on failure.

Comments:

If *dwFlags* has a nonzero value, the function should return SCARD\_E\_INVALID\_PARAMETER.

If p*wszProperty* contains an unsupported value, the function should return SCARD\_E\_INVALID\_PARAMETER.

**Note:** Any minidriver may choose to define and support optional custom properties that are not defined in this specification.

The format of *pbData* depends on the value of the p*wszProperty* parameter. The following table describes the supported values for p*wszProperty* along with the corresponding format for *pbData*.

| ***pwszProperty* value** | ***pbInput* type** | ***pbInput* value** |
| --- | --- | --- |
| CKP\_CHAINING\_MODE | LPWSTR | The data contains a string that describes the type of chaining mode that the card should use for decryption.  For more information, see the description of CP\_CHAINING\_MODES in “[**CardGetProperty**](#_CardGetPropertyCardGetProperty)” earlier in this specification. |
| CKP\_INITIALIZATION\_VECTOR | PBYTE | The data contains an initialization vector to be used for decryption. |

#### CardProcessEncryptedData

Description:

**CardProcessEncryptedData** processes a set of encrypted data BLOBs by sending them to the card where the data BLOBs are decrypted.

DWORD WINAPI CardProcessEncryptedData(

\_\_in PCARD\_DATA *pCardData*,

\_\_in CARD\_KEY\_HANDLE *hKey*,

\_\_in LPCWSTR *pwszSecureFunction*,

\_\_in\_ecount(*cEncryptedData*)

PCARD\_ENCRYPTED\_DATA *pEncryptedData*,

\_\_in DWORD *cEncryptedData*,

\_\_out\_bcount\_part\_opt(*cbOutput*, \**pdwOutputLen*)

PBYTE *pbOutput*,

\_\_in DWORD *cbOutput*,

\_\_out\_opt PDWORD *pdwOutputLen*,

\_\_in DWORD *dwFlags*

);

Input:

*pCardData* Context information for the call. For more information, see “[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*hKey* The handle of the cryptographic key to be used to decrypt the data.

*pwszSecureFunction* A pointer to a null-terminated Unicode string that contains the name of the function to be executed. For more information, see the following “Comments.”

*pEncryptedData* Pointer to an array of CARD\_ENCRYPTED\_DATA structures that contain encrypted data BLOBs.

*cEncryptedData* The number of CARD\_ENCRYPTED\_DATA structures in the array pointed to by *pEncryptedData*.

*pbOutput* A byte pointer to the data buffer that receives the decrypted data that is returned from the card minidriver.

*cbOutput* The maximum length, in bytes, of the buffer to which *pbOutput* points.

*pdwOutputLen* A pointer to a DWORD variable that receives the returned length, in bytes, of the data in the buffer to which *pbOutput* points.

*dwFlags* A set of flags that specify options for the operation. For more information, see the following “Comments.”

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

If the card minidriver does not support processing secure data BLOBs, the function should return SCARD\_E\_UNSUPPORTED\_FEATURE.

The *dwFlags* parameter is used to specify flag settings for optional parameters for the decryption operation. Currently, the only valid flag is CARD\_BLOCK\_PADDING, which specifies that the encrypted data was padded by using PKCS #5. For more information, see the description of CP\_PADDING\_SCHEMES in “[**CardGetProperty**](#_CardGetPropertyCardGetProperty)” earlier in this specification.

If *dwFlags* contains an invalid or undefined value, the function should return SCARD\_E\_INVALID\_PARAMETER.

If an unsupported *pwszSecureFunction* value is passed to **CardProcessEncryptedData**, the function should return SCARD\_E\_INVALID\_PARAMETER.

**Note:** The minidriver may choose to define and support optional custom secure functions that are not defined in the specification.

The format of the data to which *pEncryptedData* points is card-dependent. The value of the *pwszSecureFunction* parameter specifies the context to which the decryption operation is performed.

This function can be called when **CardAcquireContext** is called in No\_Card mode. This mode allows the application that receives data that the card encrypts to decrypt the data. In this situation, the *hKey* parameter is set to the value of a handle for a key that the minidriver manages.

The format of *pbOutput* depends on the value of the *pwszSecureFunction* parameter. The following table describes the different supported values for *pwszSecureFunction* along with the corresponding format for *pbOutput*.

| ***pwszSecureFunction* value** | **pbOutput type** | **pbOutput value** |
| --- | --- | --- |
| CSF\_CHANGE\_AUTHENTICATOR | STRUCT | The data contains a structure of type [CARD\_CHANGE\_AUTHENTICATOR\_RESPONSE](#_CARD_CHANGE_AUTHENTICATOR_RESPONSE). |
| CSF\_AUTHENTICATE | STRUCT | The data contains a structure of type [CARD\_AUTHENTICATE\_RESPONSE](#_CARD_AUTHENTICATE_RESPONSE). |
| CSF\_IMPORT\_KEY PAIR | None | None |

To determine the length of the buffer to allocate for *pbOutput*, a caller can first call **CardProcessEncryptedData** with *pbOutput* set to NULL. The required buffer length is returned in *pdwOutputLen*.

### Client functions

#### CardImportSessionKey

Description:

The **CardImportSessionKey** function imports a temporary session key to the card. The session key is encrypted with a key exchange key, and the function returns a handle of the imported session key to the caller.

DWORD WINAPI CardImportSessionKey(

\_\_in PCARD\_DATA *pCardData*,

\_\_in BYTE *bContainerIndex*,

\_\_in VOID \**pPaddingInfo*,

\_\_in LPCWSTR *pwszBlobType*,

\_\_in LPCWSTR *pwszAlgId*,

\_\_out CARD\_KEY\_HANDLE \**phKey*,

\_\_in\_bcount(*cbInput*) PBYTE *pbInput*,

\_\_in DWORD *cbInput*,

\_\_in DWORD *dwFlags*

);

Input:

*pCardData* Context information for the call. For more information, see ”[**CardAcquireContext**](#_CardAcquireContext)” earlier in this specification.

*bContainerIndex* Index number for the container that is used to decrypt the key material in *pbInput*. This index value must identify an RSA key container. ECC keys cannot be used for this decryption operation.

*pPaddingInfo* A pointer to a structure that contains padding information. The type of structure to which this parameter points depends on the value of the *dwFlags* parameter.

*pwszBlobType* A pointer to a null-terminated Unicode string. This string identifies the type of BLOB in the *pbInput* buffer. For more information, see the following “Comments.”

*pwszAlgId* A pointer to a null-terminated Unicode string. This string identifies algorithm of the key inside encrypted data. For more information, see the description of CP\_ENUM\_ALGORITHMS in “[**CardGetProperty**](#_CardGetPropertyCardGetProperty)” earlier in this specification.

*phKey* A pointer to a CARD\_KEY\_HANDLE that receives the handle of the imported key. This handle is used in subsequent functions that require the key, such as **CardProcessEncryptedData**. The handle must be released when it is no longer needed by calling **CardDestroyKey**.

*pbInput* A byte pointer to the buffer that contains the key BLOB to be imported. The *cbInput* contains the size of this buffer. The *pwszBlobType* parameter specifies the type of key BLOB that this buffer contains. Everything should be encrypted with the exception of the BLOB header, if it is present.

*cbInput* The size, in bytes, of the key BLOB in the *pbInput* buffer.

*dwFlags* A set of flags that specify options for the import operation. For more information, see the following “Comments.”

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

If the card minidriver does not support the import of temporary symmetric keys, the function should return SCARD\_E\_UNSUPPORTED\_FEATURE.

The following table describes the string values that are valid for the *pwszBlobType* parameter .

| **String value** | **Description** |
| --- | --- |
| “KeyDataBlob” | The *pbInput* parameter is a pointer to a buffer that contains a BCRYPT\_KEY\_DATA\_BLOB\_HEADER structure. The key BLOB data immediately follows the BCRYPT\_KEY\_DATA\_BLOB\_HEADER structure in the buffer.  For more information, see “[BCRYPT\_KEY\_DATA\_BLOB\_HEADER Structure](http://msdn.microsoft.com/en-us/library/aa375524(VS.85).aspx)” on MSDN. |

If *pwszBlobType* contains an invalid or undefined value, the function should return SCARD\_E\_INVALID\_PARAMETER. For legacy applications and cards that use the minidriver interface for secure key injection, the minidriver can accept proprietary BLOB types.

If *pwszAlgId* contains an invalid or undefined value, the function should return SCARD\_E\_INVALID\_PARAMETER. If the *pwszAlgId* value is defined but not supported, the function should return SCARD\_E\_UNSUPPORTED\_FEATURE. For legacy cards, the minidriver can support other algorithms.

When the function returns, the *phKey* parameter should contain a handle to the imported key. If the parameter does not contain a valid pointer, the function should return SCARD\_E\_INVALID\_PARAMETER. The key is expected to be valid until either **CardDestroyKey** is called or the card is removed from the reader.

The flags in the following table can be set in the *dwFlags* parameter:.

| **Flag value** | **Description** |
| --- | --- |
| CARD\_PADDING\_NONE | No padding was used. The *pPaddingInfo* parameter is not used. |
| CARD\_PADDING\_PKCS1 | The data was padded with a random number when the data was encrypted. The *pPaddingInfo* parameter is not used. |
| CARD\_PADDING\_OAEP | The OAEP scheme was used when the data was encrypted. The pPaddingInfo parameter is a pointer to a BCRYPT\_OAEP\_PADDING\_INFO structure.  For more information, see “[BCRYPT\_OAEP\_PADDING\_INFO Structure](http://msdn.microsoft.com/en-us/library/aa375526(VS.85).aspx)” on MSDN. |

**Note:** Only one flag can be set in the *dwFlags* parameter.

If *dwFlags* contains an invalid or undefined value, the function should return SCARD\_E\_INVALID\_PARAMETER. If *dwFlags* contains a value that is defined but not supported, the function should return SCARD\_E\_UNSUPPORTED\_FEATURE.

Only users can create symmetric keys. If an administrator attempts to import a symmetric key, the function should return SCARD\_W\_SECURITY\_VIOLATION.

Error checking is performed based on the order of cost. The minidriver should perform all parameter verification first without communicating to the card. Error conditions that would result in a return of SCARD\_E\_UNSUPPORTED\_FEATURE must be checked first.

# File System Requirements

The “logical” layout is the data layout that was presented to the Base CSP/KSP. This layout uses more human-readable names, and the files may not correspond one-to-one with files in the physical layout that the card employs.

## File Naming Requirements

File names are composed of up to eight ANSI characters (8 bit), excluding characters that the Windows file and directory naming conventions do not allow. The directory structure consists of two levels: the root directory and directories that applications use. Directory names are composed of up to eight ANSI characters. To produce file names and directory names that are not case-sensitive, card minidriver implementations should convert strings to lowercase.

## File System Virtualization

It is permissible to implement a virtual file system in the card minidriver that maps directories and files to appropriate locations on the card. Cards that do not allow write operations during normal operations (such as National ID cards) may simulate the writing operations but must maintain any files that are “written” for the duration of the insertion of the card and must be able to return these files when they are read.

## Physical Card Data Layout

The following information about files on the card is an overview of how the card and file system are used. It is not intended that the card minidriver should be designed with knowledge of these files or their contents. The card minidriver should be written as a generalized interface layer.

## Logical Data Layout

### Card Identifier

The card identifier is a unique identifier for a card. It may be represented in some form to the user in the UI, but otherwise is used only for comparison to a reference value to establish the identity of a card. This value is assigned when the card is prepared for the user. It is organized as a byte array.

File Name

The logical name for this file is “CardId”. It is in the root directory.

Access Conditions

The access conditions for this file are E(R), U(R), and A(RW).

Contents

The file is organized as a 16-byte array. It should be treated as opaque binary data.

Remarks

This value is assigned by Microsoft software to assure that a unique value is generated for the card. It is unrelated to the serial number that may or may not be assigned to the card during manufacture.

### Application Directory

The Application directory file consists of a list of fixed-length application name entries. The application directory name is the name of the logical subdirectory that contains all of the application’s files. For an application that uses CAPI2, the name is “mscp”, for which the index value is zero.

Logical Name

The logical name for this file is “cardapps”. It is in the root directory.

Access Conditions

The access conditions for this file areE(R), U(RW), and A(RW).

Contents

The file is organized as a series of records that contain a byte index followed by a zero-terminated application name string (ANSI).

Remarks

The implementation of applications requires that application names map to a unique directory on the card and also to a unique index for the application’s data in the card cache file. The card application directory allows an application to find its index value in the cache file by finding its name in the application directory and noting the index of the position where this occurs. The file consists of an 8‑byte records that contain the application name, zero filled at the end. The application name can use all 8 bytes so that there is no requirement that the resulting string be zero-terminated. Thus, the contents of the file for a “created” card are the following 8 bytes:

{‘mscp’,0,0,0,0}

### Cache File

To improve performance and reduce communication with the card, the Base CSP/KSP can cache card data in various ways. The cache file is used to control operation of the caching subsystem within the Base CSP/KSP by indicating the version number of data on the card. When data is changed, this value is incremented. Comparing its internal copy of the cache file with the version that was read from the card allows the Base CSP/KSP to determine whether cached data can be used or must be refreshed. The need to make this determination can occur for many reasons, including withdrawing and reinserting the card.

Reading the card identifier and the cache file from the card should be entirely sufficient to permit using information that was cached for an indeterminate period of time on the host.

Logical Name

The logical name for this file is “CardCF”. It is in the root directory.

Access Conditions

The access conditions for this file areE(R)U(RW)A(RW).

Contents

The file is organized global data in the form of 2‑byte values followed by a succession of 32-bit cache values that applications maintain and interpret. The first of these is reserved for the Base CSP/KSP to use. Thereafter, each application is allocated a single DWORD.

typedef struct \_CARD\_CACHE\_FILE\_FORMAT

{

BYTE bVersion; // Cache version

BYTE bPinsFreshness; // Card PIN

WORD wContainersFreshness;

    WORD wFilesFreshness;

} CARD\_CACHE\_FILE\_FORMAT, \*PCARD\_CACHE\_FILE\_FORMAT;

Remarks

An application’s internal cache is refreshed if the cache data copy that is internal to the application indicates a different version number for the data of interest than the file read from the card. The cache is generally checked at the beginning of each transaction with the card.

The array of application cache data DWORDs, one for each caching application, is indexed by the application index from the application directory file. As applications are added, the file grows by 4-byte increments.

### Container Map File

The container map file is owned by the Base CSP/KSP and consists of a number of records of CONTAINERMAPRECORD type. These records associate a container identifier, which is typically a GUID that was assigned by CAPI to an index that can be used to access keys and certificates for that container.

The position (index) of the record in the file corresponds to the index of the certificate and key information that are associated with that container. Thus, the second record in such a file would see zero-based index 1.

The certificate that is associated with this container and the signing and/or key exchange keys for the container all share this index (UserCerts\SignatureCert1, SignatureKey1, and so on). The records contain the container GUID and size information for keys that are associated with that index.

Logical Name

The logical name for this file is “CMapFile”. It is in the “mscp” directory.

Access Conditions

The access conditions for this file are E(R), U(RW), and A(RW).

Contents

The file is organized as a series of fixed length records. For a description of the record format, see the following “Remarks.”

Remarks

This file is created and its content maintained by the Base CSP/KSP. Information about the internal structure of this file is provided for reference only. The records in the file have the following format:

**CONTAINERMAPRECORD**

These records contain the CAPI-assigned container GUID and the key sizes for the associated key exchange or signing keys that are associated with that container. All WORD members are little-Endean byte order.

//

// Type: CONTAINER\_MAP\_RECORD

//

// This structure describes the format of the Base CSP's   
// container map file, stored on the card. This is well-known   
// logical file wszCONTAINER\_MAP\_FILE. The file consists of   
// zero or more of these records.

//

#define MAX\_CONTAINER\_NAME\_LEN                  39

// This flag is set in the CONTAINER\_MAP\_RECORD bFlags   
// member if the corresponding container is valid and currently   
// exists on the card. // If the container is deleted, its   
// bFlags field must be cleared.

#define CONTAINER\_MAP\_VALID\_CONTAINER           1

// This flag is set in the CONTAINER\_MAP\_RECORD bFlags

// member if the corresponding container is the default

// container on the card.

define CONTAINER\_MAP\_DEFAULT\_CONTAINER         2

typedef struct \_CONTAINER\_MAP\_RECORD

{

    WCHAR wszGuid [MAX\_CONTAINER\_NAME\_LEN + 1];

    BYTE bFlags;

    BYTE bReserved;

    WORD wSigKeySizeBits;

    WORD wKeyExchangeKeySizeBits;

} CONTAINER\_MAP\_RECORD, \*PCONTAINER\_MAP\_RECORD;

The **wszGuid** member consists of a UNICODE character string representation of an identifier that CAPI assigned to the container. This is usually, but not always, a GUID string. Identifier names cannot contain the special character “\”. When read-only cards are provisioned, the provisioning process must follow the same guidelines for identifier names.

Container names must be null-terminated and must not be greater than (MAX\_CONTAINER\_NAME\_LEN + 1) characters in length including the NULL terminator.

If a record must be removed from this table, the entry is invalidated by writing zeroes to the record. Such a record can later be overwritten by new data. The table is not “packed” to remove inactive entries.

The following bits are valid for the Flags byte:

* Bit 0 is set when the container record is valid.
* Bit 1 is set when the container is default. Only one record in the container map can have this bit set at any time. This bit can be set only if Bit 0 is also set. In other words, you cannot have a default container that is not valid. All other bits are currently reserved for future revisions of the card minidriver.
* For the default container, this translates to the byte 0x03. For a valid container that is not the default, this value is 0x01.
* Bits 2-7 are reserved for future use.

## Data Layout Summary

The following table summarizes the organization of the data at the interface between the card minidriver and the Base CSP/KSP for a typical implementation. The “Logical Name” is the string that the Base CSP/KSP uses to communicate with the card minidriver; it may or may not directly map to a corresponding element on the card.

Note that certificates and keys are logically grouped by the Base CSP/KSP into subdirectories according to their purpose, by using only an index for the actual file name. Any certificates or keys that are added to the card are named according to their index number in their directory. Some example certificates and keys are shown in the following table for the purpose of illustration.

| **Directory name** | **File name** | **Type** | **Access conditions** | **Comments** |
| --- | --- | --- | --- | --- |
| <root> | cardid | File | E(R) U(R) A(RW) | Card identifier |
| <root> | cardcf | File | E(R) U(RW) A(RW) | Cache file |
| <root> | cardapps | File | E(R) U(RW) A(RW) | Directory index by application name. For more information, see “[Application Directory](#_Application_Directory)” earlier in this specification. |
| mscp |  | Dir | E(R) U(RW) A(RW) | Base CSP/KSP App Directory |
| mscp | cmapfile | File | E(R) U(RW) A(RW) | CAPI GUID to index |
| mscp | kxc00 | File | E(R) U(RW) A(RW) | (example) key exchange cert 0 |
| mscp | ksc00 | File | E(R) U(RW) A(RW) | (example) key signature cert 0 |
| mscp | ksc01 | File | E(R) U(RW) A(RW) | (example) key signature cert 1 |
| mscp | msroots | File | E(R) U(RW) A(RW) | Enterprise trusted roots |

**Note:** Interoperability with msroots: mscp\msroots file is a PKCS #7 formatted certificate store.

## File Access Control

### Known Principals

Known principals are identifiers for the various types of users that can attempt to access card data in some way. The following table shows valid principals, with a single letter abbreviation that can be used together with a data access operation identifier to define an access condition. Although there can be more identifiable principals, the listing is restricted to those that have meaning to the communication between the Base CSP/KSP and the card minidriver.

| **Name** | **Description** | **Mnemonic** | **PIN\_ID mapping** |
| --- | --- | --- | --- |
| Everyone | Any requestor, including unauthenticated (or anonymous) users. | E | ROLE\_EVERYONE (0) |
| User | A user client of the card, who proves his identity to the card by use of a PIN. | U | ROLE\_USER (1) |
| Administrator | Card issuer or other party with an administrative relationship to the card or data on the card. Uses a special PIN or KEY (that may or may not be unique to the card or user) to perform administrative tasks that the user cannot perform without using this data, such as PIN unblocking. | A | ROLE\_ADMIN (2) |

When “everyone” is used in the following discussion, it typically means any user of the card, whether authenticated or not. “Everyone can read a file,” for example, means that the user or administrator can automatically read the file.

For file system access, the administrator is generally regarded as a “super-user” and has all the same privileges as the user (with the exception of execute privilege).

### Directory Access Conditions

Principals can create directories in the card file system with two sets of permissions. The following table summarizes the effect of each of the permissions.

| **Directory access condition** | **What this means** |
| --- | --- |
| UserCreateDeleteDirAc | The user and administrator can create files in the directory by using **CardCreateFile**.  The user and administrator can delete the Directory (if it is not empty) by calling **CardDeleteDirectory**.  Everyone can list the contents of the directory by using **CardEnumFiles**. |
| AdminCreateDeleteDirAc | The administrator can create files in the directory by using **CardCreateFile**,  The administrator can delete the Directory by using CardDeleteDirectory.  Everyone can list the contents of the directory by using **CardEnumFiles**.  This ACL is optional. It may be removed from future revisions of the smart card minidriver specification. |

**Note:** When creating a directory, everyone automatically has permissions to list the files in the directory. There are no separate “list” permissions for directories.

### File Access Operations

Principals can use the contents of files in various ways. Valid operations are listed in the following table, with a single letter abbreviation that can be used, together with a principal designator to define an access condition. In particular, note that Execute (X) has no logical relationship to other file access operations—it is an independent operation.

| **Operations/privileges** | **Description** | **Mnemonic** |
| --- | --- | --- |
| Read | Receive the contents of the file either directly or in a formatted or processed form. | R |
| Write | Change the contents of a file, possibly creating the file, or removing, replacing, or altering existing data. | W |
| Execute | Use the file contents for an operation that is conducted by the card on the requestor’s behalf, without being able to receive the data so used or feasibly derive it. | X |

### File Access Conditions

Access conditions are similar to ACLs. Access conditions control which principals can access a given file and what operations they can perform. Each file on the card has an access condition that can be described by a list of principals and their access privileges. If a principal or a privilege is not included in a description, it is assumed to be denied. Generally speaking, access conditions are enforced on the card.

The following table lists the access conditions that are available through **CardCreateFile** and maps them to the appropriate access condition mnemonic.

| **File access condition** | **What this actually means** | **Access condition mnemonic** |
| --- | --- | --- |
| InvalidAc | There was an error retrieving the ACL. |  |
| EveryoneReadUserWriteAc | This means that everyone can read the file or get the file information (**CardReadFile** or **CardGetFileInfo**), respectively, and that the user and administrator can read the file, write the file, and delete the file. | E(R), U(RW), A(RW) |
| UserWriteExecuteAc | The user can write the file, can “execute” the file, and can delete the file. No one, including the user, can read the contents of the file. The administrator can also write, but not execute, the contents of this file and can delete the file. | U(WX) A(W) |
| EveryoneReadAdminWriteAc | This means that everyone can read the file or get the file information (**CardReadFile** or **CardGetFileInfo**), respectively, but that only the administrator can write the file and delete the file. | E(R), U(R), A(RW) |
| UnknownAc | The file is protected by an access condition (AC) on the card that is not one of the predefined AC types. |  |
| UserReadWriteAc | Everyone No Access  // User Read Write  //  // Example: A password wallet file | U(RW), A(RW) |
| AdminReadWriteAc | Everyone/User No Access  // Admin Read Write  //  // Example: Administration data. | A(RW) |

The following table lists some sample access conditions for common items.

| **Access condition** | **Description** |
| --- | --- |
| E(X) U(W) A(W) | This would be the access condition for the user PIN. A user is unidentified when an operation that requires the PIN begins. The PIN must be “executed” to establish the user’s identity. After entry of the PIN, the user’s identity is promoted from E to U. Both the user and the administrator may write a PIN. |
| U(WX) A(W) | The user’s private key file may never be read from the card, and only the user may use its contents for cryptographic operations. This data may be changed by either the user or administrator. |
| E(R) U(R) A(RW) | Card identifier. |

### Notes on the Directory and File Access Conditions

* The principal needs Read access on the file for GetFileInfo to succeed.
* There are no separate list permissions for listing the contents of a directory.
* “Create access on a directory” means having the privilege to create files in the directory, whereas “delete access on the directory” means having the privilege to delete the directory itself. To delete a file, the card principal must have write access to the file itself.
* It is not possible through the smart card minidriver interface to create directories with E(W) permissions.
* It is not possible through the smart card minidriver interface to change file or directory permissions without deleting and re-creating the file or directory.
* It is not possible through the smart card minidriver interface to create a private key file that is owned by either the administrator or by a non-authenticated user.
* It is not possible through the smart card minidriver interface to create a PIN file on the card (E(X), U(W), and A(W)).
* It is not possible through the smart card minidriver interface to query directory access conditions.
* It is only possible through the smart card minidriver interface to create files with a subset of the access condition combinations that are available.

# Card Requirements

To provide some context for the other requirements, this section gives some information about how the card is provisioned and used.

## What a “Blank Card” Is

A “blank card,” which can be “created” and then used by the Microsoft Smart Card Base CSP/KSP, is a card that :

* Contains the card operating system.
* Contains or can virtualize necessary files and data to implement the file system.
* Has default values for administrative and/or user PINs or keys.
* Does not yet have the files that are discussed under “Card Creation” (the following section).
* Is ready for card creation with *no further preparation*.
* For future purposes, can provide an AID as defined in ISO 7816-4 part 8.

## Card “Creation”

For a card to be useful for cryptographic operations, it must have an identity that allows it to be recognized for purposes of deployment and management and it must be usable by the Base CSP/KSP. This requires a card ID file and certain files that the Base CSP/KSP requires to be stored on the card. The operation of creating these necessary files on the card is called “creating” the card. This is done by a deployment tool and consists of the following steps:

1. Create the card ID file, “cardid”, in the root directory of the card with everyone having Read and the administrator having Write permissions. This file contains a unique 16-byte binary identifier for the card. It is never updated or overwritten unless the card is entirely recycled.

2. Create the cache file, “cardcf”, in the root directory, with everyone having Read/Write permission. Initial contents are 6 bytes with values of zero.

3. Create the application map, “cardapps”, in the root directory, with everyone having Read and users having Write permissions. Initial contents are an 8‑byte record that consists of the string “mscp” followed by 4 zero bytes.

4. Create the Base CSP/CNG KSP application by a call to **CardCreateDirectory**, referring to application “mscp”, with everyone having Read and the users having Write permissions.

5. Create the certificate map file, “cmapfile”, in the “mscp” directory with everyone having Read and users having Write permissions. It is initially empty.

Technically, a card is “created” after step 2, but we define that all cards shall reserve the Microsoft “mscp” application, whether it is actually used. This explains the unusual facts that the “mscp” application is always created and that a file is created within the “mscp” application. As card creation is expected to be implemented by functions within the card management DLL that Microsoft supplies, this information is provided as reference information for card minidriver authors to be able to properly support these operations in that context.

# Developer Notes and Guidelines

## Challenge/Response Method of Unblocking Smart Card PIN

For an administrator to successfully use this mechanism to unblock a user’s card, administrators must be able to identify and use the administrator key that is stored on the card so that they can correctly generate the response data to the challenge that was issued.

One way to do this is to use the card identifier to uniquely identify the card. (The card identifier is a unique identifier for a card.) This can be represented in some form to users in the UI, but otherwise a program could be written to send appropriate APDU commands to the card to read this information.

This information can then allow the administrator to identify the secret key on the card and calculate the appropriate response to the challenge data that is issued to users.

It is assumed that the administrator secret key stored on a card is held by using some secure mechanism that is accessible only to valid and trusted administrators (preferably as few as possible). However, this is beyond the scope of this specification.

For more information, see “[Challenge/Response Mechanism](#_Toc150855260)” later in this specification.

## Enhanced PIN Support

Version 6.0 supported a flexible architecture for multiple PIN support. This architecture introduced a new concept of roles in which each role corresponds to a PIN identifier. The PIN identifiers are used to extract PIN information from the card, as well as to associate a PIN with a key container.

The identifier consists of a number, currently limited to 0 through7. We also introduced the notion of a PIN\_SET, which is a bitmask that can be generated from the PIN identifier. Currently only the lower 8 bits are used for the PIN set. We can also choose to use the remaining bits to indicate conditions such as ‘and’, ‘or’, or other information that we might find useful in the future. We chose this approach so that the bit mask is easy for the card to enforce.

Assume that the user authenticates with role 3, corresponding to PIN #3. This translates to the bit mask 0000 0100 (base 2). The card can record this as the currently authenticated ID and can easily verify access control rules on keys and PINs by doing a bit-wise AND operation. The design allows having multiple authenticated identities on the card simultaneously, and this is a requirement for cards that support v6 card minidrivers. As an example, if PIN #1 is authenticated and then subsequently PIN #2 is authenticated, operations that *any* of these PINs control should be allowed.

## Session PINs and Secure PIN Channel

When Windows must establish a secure PIN channel for PIN authentication, the following sequence of operations is performed with the minidriver. To comply, a minidriver and the card must be compatible with the following sequence. In particular, session PINs should be transferable between processes and last for only a certain length of time. (We recommend that any session PIN be valid until the cold reset of the card by using the CARD\_AUTHENTICATE\_ SESSION\_PIN flag even if **CardAuthenticateEx** is called with the GENERATE\_SESSION\_PIN flag set.)

The following behavior should be supported:

1. Application A, a trusted system process, acquires a handle to the smart card and collects a PIN.

2. Application A then calls the card **CardAuthenticateEx** minidriver function, and passes the PIN that was collected and sets the CARD\_AUTHENTICATE\_GENERATE\_SESSION\_PIN flag. This does *not* cause the card to be unlocked.

3. Application A stores the session PIN that was generated and releases the handle to the card and card minidriver. The card is not cold reset.

4. Application A sends the session PIN and the name of the reader that has the card that was acquired in step 1 to Application B

5. Application B acquires the same card as in 1.

6. Application B calls **CardAuthenticateEx** and passes in the session PIN and sets the CARD\_AUTHENTICATE\_SESSION\_PIN flag. If the session PIN is still valid, the card should be authenticated and valid for use.

7. When Application B is finished using the card, it calls **CardDeauthenticateEx** to deauthorize the card.

This behavior has the following practical limitations:

* Cards must declare their ability to work with session PINs by returning the appropriate value for CP\_CARD\_PIN\_STRENGTH\_VERIFY.
* Cards that rely on having the PIN for each verification are not compatible with this system.
* Several applications can have what they determine to be valid session PINs at any one time. If only one session PIN is possible for each PIN, the following implementation is advised:
* "The card should remember the most recent session PIN that was generated and continue to return that session PIN for future calls to CardAuthenticateEx with the GENERATE\_SESSION\_PIN flag set until that session PIN has been invalidated for another reason (e.g. card is cold reset, PIN retry counter exhausted, etc.)."
* If an invalid session PIN is presented, the card should fail the authentication and, if supported, decrement the retry counter for the session PIN. If the retry count reaches 0 and the next authentication attempt is invalid, the session PIN should be invalidated.
* Subsequent session PIN presentations should fail until a new session PIN is negotiated.
* The session PIN must be able to be used from different applications on the system.
* The session PIN must not simply be an encoding of the PIN.
* The security of this system is limited to the strength of the session PIN and the negotiation protocol that is used to generate it. The actual session PIN negotiation is outside the scope of this specification. We make no requirements on the design except that it works as described in this section.
* The session PIN is still considered valuable and should be treated as a secret.
* The card should be able to detect an invalid session PIN.

## Read-Only Cards

To address cards that are personalized outside the Base CSP/KSP environment and are inherently read-only, we have introduced a new concept of read-only cards. If a card is read-only, it must advertise this through the [**CardGetProperty**](#_CardGetPropertyCardGetProperty) function (see this section earlier in this specification).

Read-only cards must support only a subset of the Version 7 card minidriver interface and are not required to support an administrator PIN.

The following table lists the functions that a read-only card must support.

| **Function name** | **Required** | **Notes** |
| --- | --- | --- |
| **CardAcquireContext** | Yes |  |
| **CardDeleteContext** | Yes |  |
| **CardAuthenticatePin** | Yes |  |
| **CardGetChallenge** | No (Optional) |  |
| **CardAuthenticateChallenge** | No (Optional) |  |
| **CardDeauthenticate** | Yes (Optional) |  |
| **CardUnblockPin** | No (Optional) |  |
| **CardChangeAuthenticator** | No (Optional) |  |
| **CardCreateDirectory** | No |  |
| **CardDeleteDirectory** | No |  |
| **CardReadFile** | Yes | Card minidriver must emulate a file system. |
| **CardCreateFile** | No |  |
| **CardGetFileInfo** | Yes | Card minidriver must emulate a file system. |
| **CardWriteFile** | No |  |
| **CardDeleteFile** | No |  |
| **CardEnumFiles** | Yes | Card minidriver must emulate a file system. |
| **CardQueryFreeSpace** | Yes | Card minidriver must emulate a file system. |
| **CardQueryCapabilities** | Yes | Card minidriver must emulate a file system. |
| **CardCreateContainer** | No |  |
| **CardCreateContainerEx** | No (Optional) |  |
| **CardDeleteContainer** | No |  |
| **CardGetContainerInfo** | Yes |  |
| **CardRSADecrypt** | Yes (Optional) |  |
| **CardConstructDHAgreement** | Yes (Optional) |  |
| **CardDeriveKey** | Yes (Optional) |  |
| **CardDestroyDHAgreement** | Yes (Optional) |  |
| **CardSignData** | Yes |  |
| **CardQueryKeySizes** | Yes |  |
| **CardAuthenticateEx** | Yes |  |
| **CardChangeAuthenticatorEx** | No (Optional) |  |
| **CardDeauthenticateEx** | Yes |  |
| **CardGetChallengeEx** | No (Optional) |  |
| **CardGetContainerProperty** | Yes |  |
| **CardSetContainerProperty** | No |  |
| **CardGetProperty** | Yes |  |
| **CardSetProperty** | Yes |  |
| **MDImportSessionKey** | No (Optional) |  |
| **MDEncryptData** | No (Optional) |  |
| **CardImportSessionKey** | No (Optional) |  |
| **CardGetSharedKeyHandle** | No (Optional) |  |
| **CardGetAlgorithmProperty** | No (Optional) |  |
| **CardGetKeyProperty** | No (Optional) |  |
| **CardSetKeyProperty** | No (Optional) |  |
| **CardDestroyKey** | No (Optional) |  |
| **CardProcessEncryptedData** | No (Optional) |  |

| **Legend** | |
| --- | --- |
| Yes | This function must be implemented. |
| No | Entry point must exist and must return SCARD\_E\_UNSUPPORTED\_FEATURE. |
| No (Optional) | The operation is not required to be supported for a read-only card, but may be implemented if the card supports the operation. If not supported, the entry point must return SCARD\_E\_UNSUPPORTED\_FEATURE. |
| Yes (Optional) | This function should be implemented according to its definition in this specification, regardless of whether the card is read-only. |

The following requirements should be considered when developing a minidriver for a read-only card:

* All expected Base CSP/KSP files, with the exception of the ‘msroots’ file (such as ‘cardcf’ and ‘cardid’) must exist on the read-only card (or must be virtualized through the minidriver interface).
* A read-only card *must* contain at least one key on the card that is protected by the primary card (that is, ROLE\_USER) PIN.
* A read-only card is allowed to not contain an admin key. If this is the situation, it is expected that the minidriver will not support **CardGetChallenge**, **CardAuthenticateChallenge**, and **CardUnblockPin**.
* When queried, a read-only card should return 0 bytes available and 0 containers available.
* Only the CP\_PARENT\_WINDOW and CP\_PIN\_CONTEXT\_STRING properties should be allowed to be set on a read-only card.
* For a read-only card, the CP\_SUPPORTS\_WIN\_X509\_ENROLLMENT property should be false.

## Cache Modes

The Base CSP/KSP supports three different modes of caching depending on the cache mode that was returned by the **CardGetProperty** called with the parameter CP\_CARD\_CACHE\_MODE:

* If the returned flag is CP\_CACHE\_MODE\_GLOBAL\_CACHE and the card reported the CP\_READ\_ONLY\_CARD property as TRUE, the Base CSP/KSP data cache is a global cache. If the card is read-only, the Base CSP/KSP does not write to the cardcf file. If the card can be written to the Base CSP/KSP, it will operate as today.
* For more information about CP\_CARD\_CACHE\_MODE and CP\_CACHE\_MODE\_GLOBAL\_CACHE, see “[**CardGetProperty**](#_CardGetPropertyCardGetProperty)” later in this specification.
* When the returned flag is CP\_CACHE\_MODE\_SESSION\_ONLY, the Base CSP/KSP operates so that the data cache is cleared when it detects that the card has been removed or reinserted. In other words, we have defined a session to be the span between card insertion and removal.
* The cache is also implemented for each process and is not global. This mode is designed for read-only cards that do not change on a user’s PC, but rather at some government station or other external site. (This mode is supported for read/write cards, but we recommend the global cache for these cards.)
* If the card is read-only and there is a chance that the card will change on the user’s PC (by means other than Base CSP/KSP), the application should use the no‑cache mode that is described later in this specification to avoid the situation in which the cache could contain stale data.
* When the flag is CP\_CACHE\_MODE\_NO\_CACHE, the Base CSP/KSP does not implement any data caching. This mode is designed for card minidrivers that do not support writing the cardcf file, but where the card state can change. The card minidriver decides whether it wants to do any caching in its layer.

## Challenge/Response Mechanism

The card minidriver interface supports a challenge/response authentication mechanism. The card must generate a challenge of one or more 8‑byte blocks. The authenticating entity calculates the response by encrypting the challenge by using Triple DES (3DES) that operates operating in CBC mode with a 168-bit key (and ignoring the parity bits).

The card verifies the response by using one of the following methods:

* Repeating the encryption operation on the previously issued challenge and comparing the results.
* Decrypting the response and comparing the result to the challenge.

If the resulting values are the same, the authentication is successful.

Both the card and the authenticating entity must use the same symmetric key.

The following sample code details how the authenticating entity could calculate the response. This code does not cover any associated warranties and is provided merely as an example and guidance.

/\* © Microsoft Corporation

\* Created 08/17/05

\*/

#include <windows.h>

#include <wincrypt.h>

#include <winscard.h>

#include <stdlib.h>

#include <stdio.h>

#include <memory.h>

int \_\_cdecl wmain(int argc, \_\_in\_ecount(argc) WCHAR \*\*wargv)

{

//Acquire the context Use CryptAcquireContext

HCRYPTPROV hProv= 0;

DWORD dwMode=CRYPT\_MODE\_ECB;

BYTE \*pbLocData = NULL,tempbyte;

DWORD cbLocData = 8, count = 0;

HCRYPTKEY hKey = 0;

BYTE rgEncByte [] = {0xA8,0x92,0xD7,0x56,0x01,0x61,0x7C,0x5D };

BYTE DesKeyBlob [] = {

0x08, 0x02, 0x00, 0x00, 0x03, 0x66, 0x00, 0x00,

0x18, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00

};

pbLocData = (BYTE \*) malloc (sizeof(BYTE)\*cbLocData);

memcpy(pbLocData,rgEncByte,cbLocData);

if(!CryptAcquireContext(

&hProv,

NULL,

L"Microsoft Enhanced Cryptographic Provider V1.0",

PROV\_RSA\_FULL,

CRYPT\_VERIFYCONTEXT))

{

printf(

"Acquire context failed with 0x%08x \n",

GetLastError());

goto Cleanup;

}

if (!CryptImportKey(

hProv,

DesKeyBlob,

sizeof(DesKeyBlob),

0,

0,

&hKey ) )

{

printf("Error 0x%08x in importing the 3Des key \n",

GetLastError());

goto Cleanup;

}

if(!CryptSetKeyParam(

hKey,

KP\_MODE,

(BYTE \*)&dwMode,

0))

{

printf("Error 0x%08x in CryptSetKeyParam \n",

GetLastError());

goto Cleanup;

}

if(!CryptEncrypt(

hKey,

0,

FALSE,

0,

pbLocData,

&cbLocData,

cbLocData))

{

printf("Error 0x%08x in CryptEncrypt call \n",

GetLastError());

goto Cleanup;

}

for(count=0; count < cbLocData; ++count)

{

printf("0x%02x",pbLocData[count]);

}

printf("\n");

Cleanup:

if(hKey)

{

CryptDestroyKey(hKey);

hKey = 0;

}

if(pbLocData)

{

free(pbLocData);

pbLocData = NULL;

}

if(hProv)

CryptReleaseContext(hProv,0);

return 0;

}

## Interoperability with msroots

The msroots file is a PKCS #7 formatted certificate store for enterprise trusted roots. (The file is a bag of certificates with empty content and an empty signature and is written and read by the Base CSP.) Card minidriver developers are not required to write any special code in the card minidriver to handle this file. When storing certificates in msroots file, properties such as CODE\_SIGNING EKU are not propagated to the smart card because the msroots file stores certificates in a format different from the machine stores. Developers who want to read or write this file from other applications can use the following sample code snippets to access the data.

**Read operations:**

if (FALSE == CryptQueryObject( CERT\_QUERY\_OBJECT\_BLOB,

        &dbStore,

        CERT\_QUERY\_CONTENT\_FLAG\_PKCS7\_SIGNED,

CERT\_QUERY\_FORMAT\_FLAG\_BINARY,

        0,

NULL,

NULL,

NULL,

phCertStore,

NULL,

NULL))

    {

dwSts = GetLastError();

    }

**Write operations:**

// Serialize the store

if (FALSE == CertSaveStore( hCertStore,

PKCS\_7\_ASN\_ENCODING | X509\_ASN\_ENCODING,

CERT\_STORE\_SAVE\_AS\_PKCS7,

CERT\_STORE\_SAVE\_TO\_MEMORY,

&dbStore,

0))

    {

dwSts = GetLastError();

goto Ret;

}

dbStore.pbData = CspAllocH(dbStore.cbData);

if (NULL == dbStore.pbData)

    {

dwSts = ERROR\_NOT\_ENOUGH\_MEMORY;

goto Ret;

}

    if (FALSE == CertSaveStore( hCertStore,

PKCS\_7\_ASN\_ENCODING | X509\_ASN\_ENCODING,

  CERT\_STORE\_SAVE\_AS\_PKCS7,

CERT\_STORE\_SAVE\_TO\_MEMORY,

&dbStore,

0))

{

dwSts = GetLastError();

goto Ret;

}

## Group Policy Settings for Microsoft Base Smart Card CSP

Group Policy settings for the Microsoft Base Smart Card Crypto Service Provider are located in [HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Cryptography\Defaults  
\Provider\Microsoft Base Smart Card Crypto Provider].

| **Key** | **Description** |
| --- | --- |
| DefaultPrivateKeyLenBits | dword:00000400  Default key generation parameter—1024-bit key. |
| RequireOnCardPrivateKeyGen | dword:00000000  This sets the flag for requiring on-card private key generation (default).  If this value is set, the key that is generated on a host can be imported into the card. This is used for cards that do not support on-card key generation or where key escrow is required. |
| TransactionTimeoutMilliseconds | dword:000005dc  1500, 1.5 seconds is the default time-out for holding transactions to the card. |
| AllowPrivateSignatureKeyImport | dword:00000000  Allows importing signature keys, that is, key archival scenarios. |
| AllowPrivateExchangeKeyImport | dword:00000000  Allows importing exchange keys, that is, key archival scenarios. |

## Group Policy Settings for Microsoft CNG Smart Card KSP

Group Policy Settings for Microsoft CNG Smart Card Key Storage Provider are located in [HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\Cryptography  
\Providers\Microsoft Smart Card Key Storage Provider].

| **Key** | **Description** |
| --- | --- |
| DefaultPrivateKeyLenBits | dword:00000400  Default key generation parameter—1024-bit key. |
| RequireOnCardPrivateKeyGen | dword:00000000  This sets the flag for requiring on-card private key generation (default).  If this value is set, a key that is generated on a host can be imported into the card. This is used for cards that do not support on-card key generation or where key escrow is required. |
| TransactionTimeoutMilliseconds | dword:000005dc  1500, 1.5 seconds is the default time-out for holding transactions to the card. |
| AllowPrivateSignatureKeyImport | dword:00000000  Allows importing signature keys, that is, key archival scenarios. |
| AllowPrivateExchangeKeyImport | dword:00000000  Allows importing exchange keys, that is, key archival scenarios. |
| AllocPrivateECDHEKeyImport | Dword:00000000  Allows importing ECDH keys, that is, key archival scenarios |
| AllowPrivateECDSAKeyImport | Dword:00000000  Allows importing ECDSA keys, that is, key archival scenarios |

## Known Issues

* In Windows Vista SP1, while the operating system is running in safe mode, no PIN-required smart card operations are possible, other than Windows logon.
* Calling **CryptAcquireContext** with one of the following flags prompts for PIN authentication with USER\_PIN regardless of the actual PIN that is assigned to the container:
* CRYPT\_NEWKEYSET
* CRYPT\_DEFAULT\_CONTAINER\_OPTIONAL
* CRYPT\_DELETEKEYSET
* CRYPT\_VERIFYCONTEXT
* **CardDeleteContext** can be called even after **DllMain** was called with DLL\_PROCESS\_DETACH.

# Appendix A. Smart Card Plug and Play

## Pairing Process

The operating system follows these steps to pair a smart card with an already installed minidriver:

* Get the ATR from the smart card.
* Iterate through entries in the **HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Cryptography\Calais\SmartCards** registry key and do the following:
* Apply **ATRMask** subkey value that is stored in the registry to the ATR that was acquired from the smart card.
* Compare the masked ATR value to the **ATR** subkey value that is stored in the registry.
* If the two ATR values match, stop processing and pair the corresponding minidriver with the smart card.

Smart card ATR and ATRMask values must be carefully chosen to avoid the erroneous pairing of a minidriver with a smart card. The smart card ATR value that is stored in the registry should be the expected value after the ATRMask has been applied to an ATR read from a smart card. Otherwise, the masked ATR values from the card and the registry do not match and the pairing fails.

Starting with Windows 7, the first time a smart card is inserted into a card reader triggers Plug and Play events that result in a search for an appropriate minidriver on the Windows Update site. The device ID that Windows generates to locate the driver on Windows Update depends upon the following factors:

* Historical bytes from the ATR. For more information about ATR historical bytes, see section 8 of the ISO/IEC 7816-4:2005(E) standard.
* Presence of the Microsoft Plug and Play AID application with a list of GUIDS in tag 0x7F68.
* Presence of a PIV application on the card which will be paired with an inbox driver.
* Presence of a GIDS (Generic Identity Device Specification) application with [Microsoft Generic Profile](#_Electrical_Profile_for_1) on the card which will be paired with an inbox driver.

For more detailed information on the smart card discovery process for Plug and Play and Winscard, see “[Appendix D](#_Appendix_D_Smart).” These processes result in the generation of a unique device ID for the smart card.

**Note:** To determine the device ID that Windows generates for a smart card, the recommended approach is to insert the smart card in a smart card reader that is attached to a computer that is running Windows 7 or later versions of Windows. The device ID can then be found by looking at the “Hardware Ids” property of the smart card device in Device Manager.

## Sample INF for x86 and amd64

The following is a sample INF file for smart card installation in Windows 8 and earlier versions of Windows. This INF file is decorated for installation in X86 and AMD64 CPU platforms. **To avoid problems with deployments, it is strongly advised to test your driver package on clean installations of all targeted operating systems prior to submitting the driver package to Winqual.**

;

;FabrikamVendor Smartcard Minidriver for an x86 and x64 based package.

;

[Version]

Signature="$Windows NT$"

Class=SmartCard

ClassGuid={990A2BD7-E738-46c7-B26F-1CF8FB9F1391}

Provider=%FABRIKAMVENDOR%

CatalogFile=delta.cat

DriverVer=10/03/2008,7.0.0.4

[Manufacturer]

%FABRIKAMVENDOR%=FabrikamVendor,NTamd64,NTamd64.6.1,NTx86,NTx86.6.1

[FabrikamVendor.NTamd64]

%FabrikamCardDeviceName%=FabrikamVendor64\_Install,SCFILTER\CID\_51FF0800

[FabrikamVendor.NTx86]

%FabrikamCardDeviceName%=FabrikamVendor32\_Install,SCFILTER\CID\_51FF0800

[FabrikamVendor.NTamd64.6.1]

%FabrikamCardDeviceName%=FabrikamVendor64\_61\_Install,SCFILTER\CID\_51FF0800

[FabrikamVendor.NTx86.6.1]

%FabrikamCardDeviceName%=FabrikamVendor32\_61\_Install,SCFILTER\CID\_51FF0800

[DefaultInstall]

CopyFiles=x86\_CopyFiles

AddReg=AddRegDefault

[DefaultInstall.ntamd64]

CopyFiles=amd64\_CopyFiles

CopyFiles=wow64\_CopyFiles

AddReg=AddRegWOW64

AddReg=AddRegDefault

[DefaultInstall.NTx86]

CopyFiles=x86\_CopyFiles

AddReg=AddRegDefault

[SourceDisksFiles]

Fabrikamcm64.dll=1

Fabrikamcm.dll=1

[SourceDisksNames]

1 = %MediaDescription%

[FabrikamVendor64\_Install.NT]

CopyFiles=amd64\_CopyFiles

CopyFiles=wow64\_CopyFiles

AddReg=AddRegWOW64

AddReg=AddRegDefault

[FabrikamVendor64\_61\_Install.NT]

CopyFiles=amd64\_CopyFiles

CopyFiles=wow64\_CopyFiles

AddReg=AddRegWOW64

AddReg=AddRegDefault

Include=umpass.inf

Needs=UmPass

[FabrikamVendor32\_Install.NT]

CopyFiles=x86\_CopyFiles

AddReg=AddRegDefault

[FabrikamVendor32\_61\_Install.NT]

CopyFiles=x86\_CopyFiles

AddReg=AddRegDefault

Include=umpass.inf

Needs=UmPass

[FabrikamVendor64\_61\_Install.NT.Services]

Include=umpass.inf

Needs=UmPass.Services

[FabrikamVendor32\_61\_Install.NT.Services]

Include=umpass.inf

Needs=UmPass.Services

[FabrikamVendor64\_61\_Install.NT.HW]

Include=umpass.inf

Needs=UmPass.HW

[FabrikamVendor64\_61\_Install.NT.CoInstallers]

Include=umpass.inf

Needs=UmPass.CoInstallers

[FabrikamVendor64\_61\_Install.NT.Interfaces]

Include=umpass.inf

Needs=UmPass.Interfaces

[FabrikamVendor32\_61\_Install.NT.HW]

Include=umpass.inf

Needs=UmPass.HW

[FabrikamVendor32\_61\_Install.NT.CoInstallers]

Include=umpass.inf

Needs=UmPass.CoInstallers

[FabrikamVendor32\_61\_Install.NT.Interfaces]

Include=umpass.inf

Needs=UmPass.Interfaces

[amd64\_CopyFiles]

Fabrikamcm.dll,Fabrikamcm64.dll

[x86\_CopyFiles]

Fabrikamcm.dll

[wow64\_CopyFiles]

Fabrikamcm.dll

[AddRegWOW64]

HKLM, %SmartCardNameWOW64%,"ATR",0x00000001,3b,04,51,ff,08,00

HKLM, %SmartCardNameWOW64%,"ATRMask",0x00000001,ff,ff,ff,ff,ff,ff

HKLM, %SmartCardNameWOW64%,"Crypto Provider",0x00000000,"Microsoft Base Smart Card Crypto Provider"

HKLM, %SmartCardNameWOW64%,"Smart Card Key Storage Provider",0x00000000,"Microsoft Smart Card Key Storage Provider"

HKLM, %SmartCardNameWOW64%,"80000001",0x00000000,%SmartCardCardModule%

[AddRegDefault]

HKLM, %SmartCardName%,"ATR",0x00000001,3b,04,51,ff,08,00

HKLM, %SmartCardName%,"ATRMask",0x00000001,ff,ff,ff,ff,ff,ff

HKLM, %SmartCardName%,"Crypto Provider",0x00000000,"Microsoft Base Smart Card Crypto Provider"

HKLM, %SmartCardName%,"Smart Card Key Storage Provider",0x00000000,"Microsoft Smart Card Key Storage Provider"

HKLM, %SmartCardName%,"80000001",0x00000000,%SmartCardCardModule%

[DestinationDirs]

amd64\_CopyFiles=10,system32

x86\_CopyFiles=10,system32

wow64\_CopyFiles=10,syswow64

; =================== Generic ==================================

[Strings]

FABRIKAMVENDOR ="FabrikamVendor"

MediaDescription="FabrikamVendor Smart Card Minidriver Installation Disk"

FabrikamCardDeviceName="FabrikamVendor Minidriver for Smart Card"

SmartCardName="SOFTWARE\Microsoft\Cryptography\Calais\SmartCards\Fabrikam"

SmartCardNameWOW64="SOFTWARE\Wow6432Node\Microsoft\Cryptography\Calais\SmartCards\Fabrikam"

SmartCardCardModule="Fabrikamcm.dll"

**Notes:**

1. The hardware ID that is specified by the %FabrikamCardDeviceName% string must either be the ATR historical bytes of the device or the decoded value of the device’s smart card framework identifier. For more information about this identifier, see “[Appendix D.4.6](#_Windows_Smartcard_Framework).”
2. The **DefaultInstall** section is mandatory in INF files for smart card minidriver packages.
3. For more information on INF files and syntax, see “[Device and Driver Installation](http://msdn.microsoft.com/en-us/library/aa972910.aspx)“ in the Windows Driver Kit (WDK).

# Appendix B. Use Case Scenario for Secure Key Injection

In this example scenario, a client application requests that a certificate be issued from the CA application that is running on a server on behalf of the smart card owner. The CA also requires key archival. Please refer to the footnote in section Secure Key Injection for guidance on using asymmetric keypair to establish temporary symmetric session keys.

The user key is generated on the server-side, archived and then injected into the user’s smart card by using [Secure Key Injection](#_Secure_Key_Injection) APIs. The following figure illustrates this process.

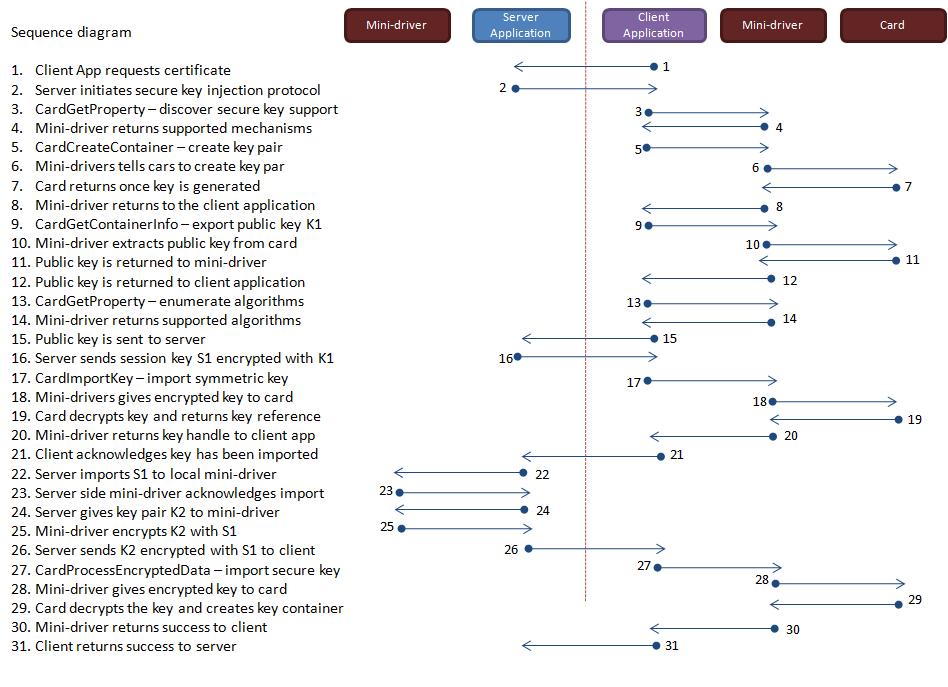


Figure B1. Process for key generation and insertion

This scenario is based on importing a symmetric session key that is encrypted with an asymmetric key, and then using this symmetric key for subsequent key wrapping.

The following describes the steps of this process as shown in Figure B1:

1. The client applications request a new certificate from a CA application that is running on the server

2. When it receives the client’s request, the server application detects that the certificate template has been configured for key recovery. As a result, the server application initiates the secure key injection protocol.

3. The client application calls [**CardGetProperty**](#_CardGetPropertyCardGetProperty) for CP\_KEY\_IMPORT\_SUPPORT to discover the following:

* Whether the card supports secure key injection.
* Which method of symmetric key import is supported.
* What algorithms are supported.

4. The minidriver indicates to the client application that it supports key injection through the asymmetric mechanism (CARD\_KEY\_IMPORT\_ASYMMETRIC\_KEYEST).

5. The client application looks through the container map file of the smart card to see if any containers are useful for key import. If none is found, the client application calls [**CardCreateContainer**](#_CardCreateContainerCardCreateContai) to generate a new key pair.

6. The minidriver instructs the smart card to create a key pair.

7. The smart card returns the key to the minidriver after the key is created.

8. The minidriver returns an indication to the client application that the key was generated.

9. The client application now calls [**CardGetContainerInfo**](#_CardGetContainerInfoCardGetContaine) to export the public key of the key pair that was created in step 6.

10. The card minidriver instructs the card to return the public key.

11. The card extracts the public key (*K1*)from the card and returns it to the minidriver.

12. The minidriver returns K1 to the client application.

13. The client application calls [**CardGetProperty**](#_CardGetPropertyCardGetProperty) to enumerate the symmetric algorithms that the card supports, as well as enumerate the padding schemes that can be used with K1.

14. The minidriver returns the algorithms and padding modes that are supported.

15. The client application sends K1 back to the server application, along with the information that describes the symmetric key algorithms and padding modes that the card supports.

16. By using one of the algorithms that the card supports, the server application generates a symmetric key (*S1*). The symmetric key S1 is encrypted with K1 and returned to the client application. The server application also returns information about the encryption algorithm and the type of padding that was used to encrypt S1.

17. The client application calls **CardImportSessionKey** with an encrypted key data BLOB along with the reference to K1 and any padding information to be used to decrypt the BLOB.

For more information about key data BLOBs, see “[BCRYPT\_KEY\_DATA\_BLOB\_HEADER Structure](http://msdn.microsoft.com/en-us/library/aa375524(VS.85).aspx)” on MSDN.

18. The minidriver passes the encrypted BLOB data to the smart card for decryption.

19. After the symmetric key is decrypted, the smart card returns a reference to the symmetric key to the minidriver.

20. The minidriver returns a key handle to the client application for the symmetric key.

21. The client application sends an acknowledgment to the server application that the symmetric key has been imported.

22. The server application imports S1 to the server-side minidriver by calling [**MDImportSessionKey**](#_MDImportSessionKey).

23. The server-side minidriver returns success to indicate that S1 was successfully imported.

24. The server application generates the asymmetric key pair (*K2*). K2 is sent to the server-side minidriver by calling [**MDEncryptData**](#_MDEncryptData). The server application generates the IV and Chaining mode, and set this info to the server-side minidriver by calling the CardSetKeyProperty.

25. The server-side minidriver encrypts K2 by using S1, and returns the encrypted K2 to the server application.

26. The server application sends the encryptedK2 to the client application, along with any information that pertains to the encryption. This includes the IV and Chaining mode information.

27. The client application calls [**CardSetKeyProperty**](#_CardGetKeyProperty) to instruct the minidriver what IV and chaining mode to use with the S1. The client application then calls [**CardProcessEncryptedData**](#_CardProcessEncryptedData) with the following data:

* The encrypted key data BLOB that contains K2.
* The key reference to S1 so that the card can decrypt the data and create the key.

28. The minidriver performs the necessary steps to prepare a new key container and gives the encrypted key data BLOB to the smart card.

29. The smart card decrypts K2 using S1 and generates a new key container for K2. The card returns success to indicate that the key has been imported.

30. The minidriver returns success from [**CardProcessEncryptedData**](#_CardProcessEncryptedData).

31. The client application returns success and the process is complete.

# Appendix C. Overview of the Windows Inbox Smart Card Minidriver

Beginning with Windows 7, an inbox generic class minidriver is provided that supports PIV-compliant smart cards and cards that implement the GIDS card edge.

For more information about PIV, see the “[About Personal Identity Verification (PIV) of Federal Employees and Contractors](http://csrc.nist.gov/groups/SNS/piv/index.html)” Web page.

For more information about GIDS, see the “[Generic Identity Device Specification](http://msdn.microsoft.com/en-us/windows/hardware/gg487496)” web page.

When a smart card is inserted into the reader and the Base CSP/KSP calls [**CardAcquireContext**](#_CardAcquireContextCardAcquireContex), the class minidriver performs the following discovery process to mark the associated card as either PIV- or GIDS-compliant:

1. A SELECT command is issued to locate the PIV AID. If the command succeeds, Windows considers the card to be a PIV device and the discovery process stops.
2. If the command fails, a SELECT command is issued to locate the GIDS AID. If the command succeeds, Windows considers the card to be an GIDS device and the discovery process stops.
3. If the command fails with a status code that indicates neither AID exists on the smart card, Windows still proceeds as if the card is an GIDS device. If the command fails with any other error, Windows considers the card to be an unknown device.

## Electrical Profile for GIDS cards with the Microsoft Generic Profile

For Smart Cards that implement the GIDS card edge, they must be pre-provisioned with an electrical profile that enables them for provisioning with the inbox class minidriver. The information in this section requires deep understanding of APDUs, data model and the card edge as specified in the GIDS specification.

Sections C.1.1 thru C.1.8 must be followed in the order listed before the card can be used for personalization. Please refer to section 11 of the GIDS specification for more information on APDUs referenced in this section.

### GIDS Application Metadata

The DOs described in this section are managed by GIDS and can be retrieved only in the response data field of the SELECT command. This metadata can only be created when the application is in the “creation” state. Please refer to section 6 of the GIDS specification for more information on the GIDS Life Cycle Management.

Note that the metadata provided below only include what is required to be present exactly as described (unless otherwise noted). There are other fields that may be optional, or are customizable by the card application vendor.

#### File Control Information (DF FCI)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tag** | **Len** | **Value** | | |
| 61 | Var. | **Application Template Data Object** | | |
|  | | **Tag** | **Len** | **Value** |
| 4F | Var. | Application AID =  **A0 00 00 03 97 42 54 46 59 xx yy**   * **XX** = GIDS specification revision number that is either 01 or 02. * **YY** = Reserved for the card application |

#### File Management Data (DF FMD)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tag** | **Len** | **Value** | | |
| 64 | Var. | **FMD Template** | | |
|  | | **Tag** | **Len** | **Value** |
| 5F2F | Var. | PIN usage policy (see “PIN Usage Policy”) =  Either **40** or **60**   * **40** – Application PIN is present and may be used to satisfy CHV. * **60** – Application and Global PINs are both present and may be used to satisfy CHV. |

#### File Control Parameters (DF FCP)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tag** | **Len** | **Value** | | |
| 62 | Var. | **FCP Template** | | |
|  | | **Tag** | **Len** | **Value** |
| 82 | 01 | File descriptor byte: 38 (“not shareable-DF”) |
| 8C | 03 | Security attribute in compact format =  **03 30 30**   * **03** – Following bytes specify requirements for CREATE FILE for EFs and DELETE FILE for EFs (and in that order). * **30 –** User Authentication OR External Authentication satisfy requirements to create EFs. * **30** – User Authentication OR External Authentication satisfy requirements to delete EFs.   **Note:** The security attribute does not have to exactly match this, but allowing User Authentication OR External Authentication to both create and delete EFs is required. |

Once the DF FCP has been created, the card shall transition to the “initialization” state, which is the state required for creating the objects listed in section C.1.2 to C.1.6.

### PIN Creation

To create a PIN, a CHANGE REFERENCE DATA APDU for the application password must be sent to the card:

|  |  |
| --- | --- |
| **CLA** | 00 |
| **INS** | 24 |
| **P1** | 01 |
| **P2** | 80 |
| **Lc** | Length of command data field |
| **Data Field** | <password> |
| **Le** | Absent |

For example, to set the PIN to 12345678, the following APDU must be sent to the card:

00 24 01 80 08 31 32 33 34 35 36 37 38

### Pin Unblock Key (PUK) Creation

A PUK is used to unblock and/or reset the PIN in the cases where the card becomes blocked or the PIN is forgotten. If [admin key challenge/response](#_Challenge/Response_Method_of) is to be used instead, DO NOT create a PUK.

To create a PUK, a CHANGE REFERENCE DATA APDU for the application resetting password must be sent to the card:

|  |  |
| --- | --- |
| **CLA** | 00 |
| **INS** | 24 |
| **P1** | 01 |
| **P2** | 81 |
| **Lc** | Length of command data field |
| **Data Field** | <password> |
| **Le** | Absent |

For example, to set the PUK to 12345678, the following APDU must be sent to the card:

00 24 01 81 08 31 32 33 34 35 36 37 38

### ACL Creation

ACLs must be created using the CREATE FILE APDU:

|  |  |
| --- | --- |
| **CLA** | 00 |
| **INS** | E0 |
| **P1-P2** | 00 00 |
| **Lc** | Length of data field |
| **Data Field** | FCP template for EF |
| **Le** | Absent |

The ACLs mentioned in the table below must be created. Each ACL creation APDU must be followed by ActivateFile APDU (00 44 00 00 00)

|  |  |
| --- | --- |
| ACL | APDU |
| UserCreateDeleteDirAc | 00 E0 00 00 0E 62 0C 82 01 39 83 02 A0 00 8C 03 03 30 00 |
| EveryoneReadUserWriteAc | 00 E0 00 00 0E 62 0C 82 01 39 83 02 A0 10 8C 03 03 30 00 |
| UserWriteExecuteAc | 00 E0 00 00 0E 62 0C 82 01 39 83 02 A0 11 8C 03 03 30 FF |
| EveryoneReadAdminWriteAc | 00 E0 00 00 0E 62 0C 82 01 39 83 02 A0 12 8C 03 03 20 00 |
| UserReadWriteAc | 00 E0 00 00 0E 62 0C 82 01 39 83 02 A0 13 8C 03 03 30 30 |
| AdminReadWriteAc | 00 E0 00 00 0E 62 0C 82 01 39 83 02 A0 14 8C 03 03 20 20 |

### Create EF for Admin Key

EF for Admin key must be created using the CREATE FILE APDU.

|  |  |
| --- | --- |
| **CLA** | 00 |
| **INS** | E0 |
| **P1-P2** | 00 00 |
| **Lc** | Length of data field |
| **Data Field** | FCP template for EF (EFID = B080 and KeyID=80) |
| **Le** | Absent |

The following APDU must be sent to the card to create the EF for a Triple-DES three-key Admin Key:

00 E0 00 00 1C 62 1A 82 01 18 83 02 B0 80 8C 04 87 00 20 FF A5 0B A4 09 80 01 02 83 01 80 95 01 C0

The command mentioned above must be followed by an ActivateFile APDU:

00 44 00 00 00

### Inject Admin Key

The Admin Key must be injected onto the card using the PUT KEY APDU:

|  |  |
| --- | --- |
| **CLA** | 00 |
| **INS** | DB |
| **P1-P2** | 3F FF |
| **Lc** | Length of data field |
| **Data Field** | Key Usage Template |
| **Le** | Absent |

The following APDU must be sent to the card to inject the Admin key into KeyID 80:

00 DB 3F FF 26 70 24 84 01 80 A5 1F 87 18 01 02 03 04 05 06 07 08 01 02 03 04 05 06 07 08 01 02 03 04 05 06 07 08 88 03 B0 73 DC

In the example mentioned above injects the admin key with the following value:

01 02 03 04 05 06 07 08 01 02 03 04 05 06 07 08 01 02 03 04 05 06 07 08

### Set Operational State

To transition the card from the “initialization” state to the “operational” state, a SELECT DF with EFID followed an ACTIVATE FILE command needs to be sent to the card.

First, send a SELECT APDU for the DF:

|  |  |
| --- | --- |
| **CLA** | 00 |
| **INS** | A4 |
| **P1-P2** | 00 0C |
| **Lc** | 02 |
| **Data Field** | 3F FF |
| **Le** | Absent |

Secondly, use the ACTIVATE FILE APDU to change the state of the DF to “operational”:

|  |  |
| --- | --- |
| **CLA** | 00 |
| **INS** | 44 |
| **P1-P2** | 00 00 |
| **Lc** | 00 |
| **Data Field** | Absent |
| **Le** | Absent |

The following APDU must be sent to the card to bring it to operational state:

00 A4 00 0C 02 3F FF

00 44 00 00 00

After this step, the card is ready for placing the file system as described [in file system specification section](#_File_System_Requirements) and is considered a [“blank card”](http://windows/content/Shared%20Documents/Hardware%20Ecosystem%20Team/Content%20Projects/GA_white_papers/GA%20white%20paper%20drafts/Blank#_What_a_). Follow the steps for [card “creation”](http://windows/content/Shared%20Documents/Hardware%20Ecosystem%20Team/Content%20Projects/GA_white_papers/GA%20white%20paper%20drafts/Creation#_Card_) to place the filesystem on the card using the minidriver API. Alternatively, follow the steps in the next section to place the filesystem on the card using APDUs.

### Data objects on a GIDS card after the filesystem is created

For cards compliant with GIDS specification with Microsoft Generic Profile, the following table describes the data objects and their corresponding EFIDs after the mandatory objects are created as per the section on [card “creation”](http://windows/content/Shared%20Documents/Hardware%20Ecosystem%20Team/Content%20Projects/GA_white_papers/GA%20white%20paper%20drafts/Creation#_Card_). Place each of the data objects from the table below onto the card using the PUT DATA APDU as specified in the GIDS specification if the minidriver API is not being used for creating the filesystem.

|  |  |  |  |
| --- | --- | --- | --- |
| EFID | DO Tag | Contents | Friendly Name |
| **A000** | DF1F | 01 6d 73 63 70 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 a0 00 00 00 00 00 00 00 00 00 00 00 63 61 72 64 69 64 00 00 00 00 00 20 df 00 00 12 a0 00 00 00 00 00 00 00 00 00 00 00 63 61 72 64 61 70 70 73 00 00 00 21 df 00 00 10 a0 00 00 00 00 00 00 00 00 00 00 00 63 61 72 64 63 66 00 00 00 00 00 22 df 00 00 10 a0 00 00 6d 73 63 70 00 00 00 00 00 63 6d 61 70 66 69 6c 65 00 00 00 23 df 00 00 10 a0 00 00 | Master file system table |
| **A010** | DF21 | 6d 73 63 70 00 00 00 00 | \cardapps |
| **A010** | DF22 | 00 00 00 00 00 00 | \cardcf |
| **A010** | DF23 | <empty 0-byte data object> | mscp\cmapfile |
| **A012** | DF20 | <random 16-byte value> | \cardid |

## INF Sample to re-brand inbox class minidriver

Smart card vendors can use the inbox minidriver without the need to ship a driver package. To add branding information to the Plug and Play experience for such cards, vendors can provide INF files that override various strings to provide branding information. These strings include the following:

* ProviderName
* CardDeviceName
* SmartCardName

The following is a sample INF file that can be used with the inbox minidriver. This INF file is decorated for installation in x86 and amd64 CPU platforms.

;

;FabrikamVendor Smartcard Minidriver for an x86 and x64 based package.

;

[Version]

Signature="$Windows NT$"

Class=SmartCard

ClassGuid={990A2BD7-E738-46c7-B26F-1CF8FB9F1391}

Provider=%ProviderName%

CatalogFile=delta.cat

DriverVer=10/03/2009,10.0.0.1

[Manufacturer]

%ProviderName%=Minidriver,NTamd64,NTamd64.6.1,NTx86,NTx86.6.1

[Minidriver.NTamd64]

%CardDeviceName%=Minidriver64\_Install,SCFILTER\CID\_51FF0800

[Minidriver.NTx86]

%CardDeviceName%=Minidriver32\_Install,SCFILTER\CID\_51FF0800

[Minidriver.NTamd64.6.1]

%CardDeviceName%=Minidriver64\_61\_Install,SCFILTER\CID\_51FF0800

[Minidriver.NTx86.6.1]

%CardDeviceName%=Minidriver32\_61\_Install,SCFILTER\CID\_51FF0800

[DefaultInstall]

CopyFiles=x86\_CopyFiles

AddReg=AddRegDefault

[DefaultInstall.ntamd64]

CopyFiles=amd64\_CopyFiles

CopyFiles=wow64\_CopyFiles

AddReg=AddRegWOW64

AddReg=AddRegDefault

[DefaultInstall.NTx86]

CopyFiles=x86\_CopyFiles

AddReg=AddRegDefault

[DefaultInstall.ntamd64.6.1]

AddReg=AddRegWOW64

AddReg=AddRegDefault

[DefaultInstall.NTx86.6.1]

AddReg=AddRegDefault

[SourceDisksFiles]

msclmd64.dll=1

msclmd.dll=1

[SourceDisksNames]

1 = %MediaDescription%

[Minidriver64\_Install.NT]

CopyFiles=amd64\_CopyFiles

CopyFiles=wow64\_CopyFiles

AddReg=AddRegWOW64

AddReg=AddRegDefault

[Minidriver64\_61\_Install.NT]

AddReg=AddRegWOW64

AddReg=AddRegDefault

Include=umpass.inf

Needs=UmPass

[Minidriver32\_Install.NT]

CopyFiles=x86\_CopyFiles

AddReg=AddRegDefault

[Minidriver32\_61\_Install.NT]

AddReg=AddRegDefault

Include=umpass.inf

Needs=UmPass

[Minidriver64\_61\_Install.NT.Services]

Include=umpass.inf

Needs=UmPass.Services

[Minidriver32\_61\_Install.NT.Services]

Include=umpass.inf

Needs=UmPass.Services

[Minidriver64\_61\_Install.NT.HW]

Include=umpass.inf

Needs=UmPass.HW

[Minidriver64\_61\_Install.NT.CoInstallers]

Include=umpass.inf

Needs=UmPass.CoInstallers

[Minidriver64\_61\_Install.NT.Interfaces]

Include=umpass.inf

Needs=UmPass.Interfaces

[Minidriver32\_61\_Install.NT.HW]

Include=umpass.inf

Needs=UmPass.HW

[Minidriver32\_61\_Install.NT.CoInstallers]

Include=umpass.inf

Needs=UmPass.CoInstallers

[Minidriver32\_61\_Install.NT.Interfaces]

Include=umpass.inf

Needs=UmPass.Interfaces

[amd64\_CopyFiles]

msclmd.dll,msclmd64.dll

[x86\_CopyFiles]

msclmd.dll

[wow64\_CopyFiles]

msclmd.dll

[AddRegWOW64]

HKLM, %SmartCardNameWOW64%,"ATR",0x00000001,3b,04,51,ff,08,00

HKLM, %SmartCardNameWOW64%,"ATRMask",0x00000001,ff,ff,ff,ff,ff,ff

HKLM, %SmartCardNameWOW64%,"Crypto Provider",0x00000000,"Microsoft Base Smart Card Crypto Provider"

HKLM, %SmartCardNameWOW64%,"Smart Card Key Storage Provider",0x00000000,"Microsoft Smart Card Key Storage Provider"

HKLM, %SmartCardNameWOW64%,"80000001",0x00000000,%SmartCardCardModule%

[AddRegDefault]

HKLM, %SmartCardName%,"ATR",0x00000001,3b,04,51,ff,08,00

HKLM, %SmartCardName%,"ATRMask",0x00000001,ff,ff,ff,ff,ff,ff

HKLM, %SmartCardName%,"Crypto Provider",0x00000000,"Microsoft Base Smart Card Crypto Provider"

HKLM, %SmartCardName%,"Smart Card Key Storage Provider",0x00000000,"Microsoft Smart Card Key Storage Provider"

HKLM, %SmartCardName%,"80000001",0x00000000,%SmartCardCardModule%

[DestinationDirs]

amd64\_CopyFiles=10,system32

x86\_CopyFiles=10,system32

wow64\_CopyFiles=10,syswow64

; =================== Generic ==================================

[Strings]

ProviderName ="FabrikamVendor"

MediaDescription="FabrikamVendor Smart Card Minidriver Installation Disk"

CardDeviceName="FabrikamVendor Minidriver for Smart Card"

SmartCardName="SOFTWARE\Microsoft\Cryptography\Calais\SmartCards\Fabrikam"

SmartCardNameWOW64="SOFTWARE\Wow6432Node\Microsoft\Cryptography\Calais\SmartCards\Fabrikam"

SmartCardCardModule="msclmd.dll"

**Notes:**

1. The hardware ID that is specified by the %FabrikamCardDeviceName% string must either be the ATR historical bytes of the device or the decoded value of the device’s smart card framework identifier. For more information about this identifier, see “[Appendix D.4.6](#_Windows_Smartcard_Framework).”
2. The **DefaultInstall** section is mandatory in INF files for smart card minidriver packages.
3. The **DriverVer** directive in the INF file must have a value that is greater than the version and timestamp value in the inbox driver’s INF file. Otherwise, the system does not install the device by using the vendor’s INF file.

The DriverVer directive has the following syntax.

**DriverVer=***mm/dd/yyyy*[**,***w*.*x.y.z*]

We recommend that you follow these guidelines when setting the value for the DriverVer directive:

* Specify a date value that is far enough into the future so as to avoid conflicts with Windows service pack updates.
* Although the 4-digit version number is optional, you must specify a version that is significantly higher than the current version that is specified in the inbox driver’s INF file.

1. For more info about INF files and syntax, see “[Device and Driver Installation](http://msdn.microsoft.com/en-us/library/aa972910.aspx)“ in the WDK.

# Appendix D. Smart Card Discovery Process

Starting with Windows 7, smart card minidrivers that are logo-certified through the Windows Logo Program (WLP) are automatically downloaded and installed by the Windows Plug and Play components. Windows 7 also introduces a class minidriver for PIV-compatible cards and cards that support the GIDS card edge.

When a smart card is inserted into the reader, Windows performs the following discovery processes:

* Smart Card Plug and Play Process:

This process requests and download a logo-certified minidriver from Windows Update through Plug and Play.

* Winscard Discovery Process :

This process associates a compatible smart card with a PIV- or GIDS-compatible class minidriver.

* Windows Smart Card Class Minidriver Discovery Process :

This process associates an installed minidriver with a smart card.

The following table lists the AID values that the different discovery processes use.

|  |  |  |
| --- | --- | --- |
| AID name | AID value | Description |

|  |  |  |
| --- | --- | --- |
| PIV AID | A0 00 00 03 08 00 00 10 00 xx yy | PIV AID, which does not include version information. The Microsoft smart card framework ignores the least-significant 2‑bytes. |
| MS GIDS AID | A0 00 00 03 97 42 54 46 59 xx yy | Microsoft (MS) GIDS AID, which does not include version information.  The least-significant 2 bytes are not sent to the card, but are reserved by the host as follows:   * The first of these bytes (xx) is used by the Windows smart card framework for the GIDS version number. This byte must be set to the GIDS specification revision number which is either 0x01 or 0x02. * The second byte (yy) is reserved for use by the card application. |
| SC PNP AID | A0 00 00 03 97 43 49 44 5F 01 00 | Smart card Plug and Play AID. |

The following table lists the files used by the discovery process:

|  |  |
| --- | --- |
| **Command** | **Instruction (INS) value** |
| MF | 0x3F00 |
| EF.ATR | 0x2F01 |

The following table lists the commands that the different discovery processes use.

|  |  |
| --- | --- |
| Command | Instruction (INS) value |

|  |  |
| --- | --- |
| SELECT | 0xA4 |
| GET DATA | 0xCA |
| GET RESPONSE | 0xC0 |

## Smart Card Plug and Play Process

Plug and Play installs a smart card minidriver if no compatible inbox minidriver is available. Plug and Play also updates the installed smart card minidrivers though Windows Update.

To do either of these tasks, Plug and Play must be able to derive a unique ID for the smart card. Beginning with Windows 7, the following describes the smart card discovery process that Plug and Play uses to derive a unique ID for the card:

1. Plug and Play gets the historical bytes from the ATR. These bytes are used later in this discovery process.
2. Plug and Play issues a SELECT command to locate the SC PNP AID.Plug and Play issues a GET DATA command to locate the Windows proprietary tag 0x7F68 (ASN.1 DER encoded). For more information, see “[Appendix D.4.6](#_Windows_Smartcard_Framework).” If this command is successful, a list of unique identifiers is returned. Plug and Play uses the first identifier in the list as the smart card’s device ID and uses that value for the card’s unique ID. For more information, see “[Device IDs](http://msdn.microsoft.com/en-us/library/dd567931.aspx)” in the WDK.
3. If Plug and Play derives a unique ID for the smart card, it proceeds to step 12.
4. If Windows fails to obtain a device ID in the step above it will issue a SELECT of the MF and EF.ATR followed by a READ BINARY command, if Windows succeeds in obtaining a unique identifier that it can use as a device ID for WU go to step 12.
5. If Plug and Play fails to obtain a unique identifier in the step above, it issues a SELECT command for the PIV AID. If Plug and Play succeeds, it considers the smart card to be a PIV-compatible device. Plug and Play uses the following as the card’s unique ID:
   1. The PIV-compatible device ID as the device’s compatible ID. For more information, see “[Compatible IDs](http://msdn.microsoft.com/en-us/library/dd567980.aspx)” in the WDK.
   2. The card’s ATR historical bytes as the device ID. If there are no historical ATR bytes, Windows uses the PIV-compatible device id as the device ID.
6. If Plug and Play derives a unique ID for the smart card, it proceeds to step 12.
7. If the SELECT command in step 4 is unsuccessful, Windows issues a SELECT command for the MS GIDS AID.If Plug and Play succeeds in selecting the MS GIDS AID, it considers the smart card to be a GIDS-compatible device. Plug and Play uses the following as the card’s unique ID:
   1. The GIDS-compatible device ID as the compatible ID.
   2. The card’s ATR historical bytes as the device ID. If there are no historical ATR bytes, Plug and Play uses the GIDS-compatible device ID as the device ID.
8. If Plug and Play derives a unique ID for the smart card, it proceeds to step 12.
9. If Plug and Play fails to select the PIV AID or the MS GIDS AID, it uses the card’s ATR historical bytes (if any) as the device ID for the smart card’s unique ID.
10. If Plug and Play does not have the ATR historical bytes, it does not have enough information for Windows Update. Plug and Play fails the discovery process with SCARD\_E\_UNEXPECTED.
11. If Plug and Play derives a unique ID for the smart card, it proceeds to step 12.
12. Plug and Play stops the discovery process and uses the unique identifier.

Starting from Windows 8, if Plug and Play is unable to find a driver for the card, the card is paired with an inbox NULL driver. Additional software specific to the card is then required for the card to function when connected to a smart card reader connected to the PC.

## Winscard Discovery Process

The Winscard (*Winscard.dll*) discovery process is used to associate a card in the system with an installed minidriver. The process is started when **SCardListCards** or **SCardLocateCards** is called.

For more information about **SCardListCards**, see the “[SCardListCards Function](http://msdn.microsoft.com/en-us/library/aa379789(VS.85).aspx)**”** on MSDN.

For more information about **SCardLocateCards**, see “[SCardLocateCards Function](http://msdn.microsoft.com/en-us/library/aa379794(VS.85).aspx)**”** on MSDN.

Beginning with Windows 7, the following describes the Winscard discovery process:

1. Winscard looks in the registry under the **Calais** key for various subkeys that represent smart cards that are installed in the computer. These subkeys are located at:

**HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Cryptography\Calais\SmartCards**

2. Winscard searches each subkey under the **SmartCards** subkey for a match between the subkey’s **ATR** value and an ATR value that is obtained from the smart card. If a match is found, go to step 6.

3. Winscard looks for a match between a **SmartCards** subkey value for a minidriver and a value within either the **PIV Device ATR Cache (for PIV cards)** or **IDMP ATR Cache (for Microsoft GIDS-compatible cards)** subkeys. If a match is found go to step 6.

4. Winscard issues a SELECT command for the MS GIDS AID. If this command is successful, go to step 6.

5. If step 4 fails, Winscard issues a SELECT command for the PIV AID. If this command is successful, go to step 6.

6. Winscard returns the name of the card, which corresponds to the minidriver registry key that matches the card.

**Note:** The following table lists the various registry keys that the Winscard discovery process uses.

|  |  |
| --- | --- |
| Registry key | Use |

|  |  |
| --- | --- |
| **HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft \Cryptography\Calais\SmartCards** | Winscard uses this key as the **Calais\SmartCards** key in step 1. |
| **HKEY\_LOCAL\_MACHINE\ SOFTWARE\Microsoft**  **\Cryptography\Calais\PIV Device ATR Cache** | If a match is found in step 4, the full ATR of the matched card is stored in this registry key as a binary value. The name of the entry is randomly selected.  After this entry is cached, it is used in step 3 to improve performance. |
| **HKEY\_LOCAL\_MACHINE\ SOFTWARE\Microsoft**  **\Cryptography\Calais\IDMP ATR Cache** | If a match is found in step 5, the full ATR of the matched card is stored in under this registry key as a binary value. The name of the entry is randomly selected.  After this entry is cached, it is used in step 3 to improve performance. |

## Windows Smart Card Class Minidriver Discovery Process

The Windows smart card class minidriver performs the following discovery process when [**CardAcquireContext**](#_CardAcquireContext) is called. The minidriver performs this discovery process to mark the associated card as PIV- or Microsoft GIDS-compatible:

1. The minidriver issues a SELECT command for the PIV AID. If the command succeeds, the card is marked as PIV-compatible and the discovery process stops.

2. Otherwise, the minidriver issues a SELECT command for the MS GIDS AID. If the command succeeds or fails to locate the AID, the minidriver marks the card as MS GIDS.

**Notes:**

1. If the smart card was previously discovered through the Winscard discovery process with the class minidriver, it might not respond to the SELECT command for either the PIV or GIDS AID. In this situation, it must be a card from a vendor that implements the GIDS card-edge with a custom AID. Such cards could extend the Microsoft smart card data model with additional data objects.
2. PIV and GIDS smart card vendors can use the Windows smart card class minidriver and add branding by providing an INF-only installation package. For more information about using the class minidriver for compatible cards, see “[INF sample](#_INF_sample)” earlier in this specification. Only historical bytes are used for Plug and Play matching in the INF.

The INF file that the vendor provi9des creates entries under the **Calais\SmartCards** registry subkey with the following information.

|  |  |  |
| --- | --- | --- |
| Entry name | Type | Value |

|  |  |  |
| --- | --- | --- |
| **80000001** | String | Msclmd.dll |
| **ATR** | Binary | Card’s ATR |
| **ATRMask** | Binary | Card’s ATR Mask |
| **Crypto Provider** | String | Microsoft Base Smart Card Crypto Provider |
| **Smart Card Key Storage Provider** | String | Microsoft Smart Card Key Storage Provider |

## Selection Mechanisms

### Applications that Contain Microsoft identifiers

The Windows smart card framework tries to select an application by using the Microsoft Plug and Play AID. If the card does not support the specified AID, it should return an error after the SELECT command. If the SELECT command completes successfully, the framework attempts to identify the card and corresponding smart card minidriver by issuing a GET DATA command.

The GET DATA commands take place regardless of whether the SC Plug and Play AID is supported. This allows applications, which are either associated with other AIDs or are not associated with any AIDs, to implement the card selection mechanisms in this specification.

### GET DATA

After it selects the Plug and Play MS AID on the card, the smart card framework issues a GET DATA command with the Windows proprietary tag of 0x7F68. If the card supports the GET DATA command and the proprietary tag, it responds by returning a list of one or more unique identifiers. The unique identifiers must be structured as defined in “[Windows Smart Card Framework Card Identifier](#_Toc231720632).”

The Windows smart card framework uses only the first unique identifier in the list to locate and install the appropriate smart card minidriver. The other identifiers may be used in the future.

### SELECT PIV AID Command

To identify a PIV application, Windows issues the SELECT PIV AID command. If this command succeeds, a PIV application is present on the card and is now selected. In this situation, the Windows smart card framework can now associate a PIV-compliant minidriver with the card.

### SELECT MS GIDS AID Command

To identify an MS GIDS application, a SELECT MS GIDS AID command is used. If this command succeeds, an MS GIDS application is present on the card and is now selected. The Windows smart card framework can now associate an MS GIDS–compliant minidriver with the card.

### Use of the ATR Historical Bytes

Under the following conditions, the Windows smart card framework reverts to using the ATR historical bytes ATR to determine the minidriver to load:

* The smart card does not support the GET DATA command.
* The smart card does not support the AID selection methods in this specification.

The use of the ATR historical bytes is the legacy method that is used to identify the inserted card. The framework uses all historical bytes in its search for a minidriver.

### Windows Smart Card Framework Card Identifier

If the smart card supports the GET DATA command, the Windows smart card framework expects the card to return a DER-TLV encoded byte array that is formatted in the following ASN.1 Structure.

CardID ::= SEQUENCE {

  version Version DEFAULT v1,

  vendor VENDOR,

                   guids GUIDS }

**Version** ::= INTEGER {v1(0), v2(1), v3(2)}

**VENDOR** ::= IA5STRING(SIZE(0..8))

**GUID** ::= OCTET STRING(SIZE(16))

**GUIDS** ::= SEQUENCE OF GUID

The **Version** member must be set to 0 (v1).

The **VENDOR** member must be set to “MSFT”.

The **GUID** member is a 16-byte GUID that uniquely identifies the card/application combination. This value is used to detect and load the appropriate smart card minidriver.

**Note:** The IHV or ISV that issues the application must create a unique GUID for its card/application combination.

# Appendix E. Acronyms

|  |  |  |
| --- | --- | --- |
| Acronym | **Meaning** | |
| ACL | | Access Control List |
| AID | | Application ID |
| ASN.1 | | Abstract Syntax Notation One. For more information, see the [ISO/IEC 8825-1](http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=35688) standard. |
| ATR | | Answer-To-Reset |
| CA | | Certification Authority |
| CAPI | | Cryptography API |
| CBC | | Cipher Block Chaining |
| CHV | | Card Holder Verification |
| CNG | | Cryptography API: Next Generation |
| CPDK | | Cryptographic Provider Development Kit |
| CSP | | Cryptographic Service Provider |
| DER | | Distinguished Encoding Rules Of ASN.1. |
| ECC | | Elliptic Curve Cryptography |
| ECDH | | Elliptic Curve Diffie-Hellman |
| ECDSA | | Elliptic Curve Digital Signature Algorithm |
| GIDS | | Generic Identity Device Specification. Generic Identity Device Specification (GIDS) is a specification published by Microsoft under the Microsoft Community Promise. This specification is available from this location: <http://www.microsoft.com/whdc/device/input/smartcard/GIDS.mspx> |
| IV | | Initialization Vector |
| KDF | | Key Derivation Function |
| KSP | | Key Storage Provider |
| NIST | | National Institute of Standards and Technology |
| OID | | Object Identifier |
| PIN | | Personal Identification Number |
| PIV | | Personal Identity Verification |
| PUK | | PIN Unblock Key |
| RSA | | Rivest-Shamir-Adleman |
| SCRM | | Smart Card Resource Manager |
| SDK | | Software Development Kit |
| TLV | | Tag-Length-Value |
| WDK | | Windows Driver Kit |
| WLP | | Windows Logo Program |

1. Generic Identity Device Specification (GIDS) is a specification published by Microsoft under the Microsoft Community Promise. This specification is available from this location: <http://www.microsoft.com/whdc/device/input/smartcard/GIDS.mspx> [↑](#footnote-ref-1)
2. This mode of establishing temporary symmetric sessions keys require that the public key be trusted by the server application out-of-band. Moreover, the smart card must not accept decryption requests for the private key used in this context in order to mitigate against a threat where an attacker records the protocol and later on requests the card to decrypt the data that contains the symmetric session key. [↑](#footnote-ref-2)