Windows Smart Card Minidriver Specification

Version 6.02

March 7, 2008

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 Crypto 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 Base CSP and KSP card minidrivers.

This information applies to the Windows Vista® operating system SP1 and later.

The current version of this paper is maintained on the Web at:   
 http://www.microsoft.com/whdc/device/input/smartcard/sc-minidriver.mspx

Disclaimer: This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

© 2007–2008 Microsoft Corporation. All rights reserved.

Microsoft, Windows, and Windows Vista are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Author** | **Date** | **Ver-sion** | **Comments** |
| Microsoft | March 2008 | 6.02 | * Section 3.14: Clarify registry requirement. * Section 3.2.3.1: Clarify supported padding algorithm. * Section 4.2.2: Clarify CardAuthenticatePin behavior w.r.t. EXTERNAL PIN and empty PIN. * Section 4.2.8: Clarify CardDeAuthentcateEx behavior. * Allow custom card properties and container properties. * Correct incorrect card property names so that they are consistent with cardmod.h. * Section 7.3: Clarify the refcount behavior of session PIN. |
| Microsoft | October 2007 | 6.01 | * Updated version to 6.02 Beta. * Clarified that the card administrator, where present, must be able to unblock the user PIN. * Modified the expected error handling behavior of CardAuthenticate for cards that support empty or external PINs. * Deprecated CP\_CARD\_PIN\_STRENGTH\_CHANGE and CP\_CARD\_PIN\_STRENGTH\_UNBLOCK properties. * Added some points of clarification on the CP\_CARD\_PIN\_STRENGTH\_VERIFY property. * Updated the definition of read-only cards. |
| Microsoft | March 2007 | 6.0 |  |

Contents

[1 Introduction 6](#_Toc192484369)

[1.1 What’s New in Version 6 8](#_Toc192484370)

[1.1.1 Enhanced Support for PINs 8](#_Toc192484371)

[1.1.2 Support for Read-Only Cards 9](#_Toc192484372)

[1.1.3 Secure PIN Channel 9](#_Toc192484373)

[1.1.4 External PIN Support 10](#_Toc192484374)

[2 Card-Specific Minidriver Details 11](#_Toc192484375)

[2.1 Overview 11](#_Toc192484376)

[2.2 Related Document 11](#_Toc192484377)

[2.3 General Design Guidelines 11](#_Toc192484378)

[2.3.1 Transaction Management 11](#_Toc192484379)

[2.3.2 Conventions 12](#_Toc192484380)

[2.3.3 Authentication and Authorization 12](#_Toc192484381)

[2.3.4 Handling Memory Allocations 12](#_Toc192484382)

[2.4 Caching 12](#_Toc192484383)

[2.5 Mandatory Version Checking 13](#_Toc192484384)

[2.5.1 CARD\_DATA Version Checks 13](#_Toc192484385)

[2.5.2 Other Structure Version Checks 14](#_Toc192484386)

[3 Manifest and General Imports 15](#_Toc192484387)

[3.1 DllMain and Registration Mechanisms 15](#_Toc192484388)

[3.1.1 DllMain 15](#_Toc192484389)

[3.1.2 DllRegisterServer and DllUnregisterServer 15](#_Toc192484390)

[3.1.3 Registration Mechanisms 15](#_Toc192484391)

[3.1.4 Registry File 16](#_Toc192484392)

[3.2 Imports from the Smart Card Base CSP 16](#_Toc192484393)

[3.2.1 Memory Management Functions 17](#_Toc192484394)

[3.2.2 Cache Functions 18](#_Toc192484395)

[3.2.3 Cryptographic Utilities 20](#_Toc192484396)

[3.3 Imports from Smart Card Resource Manager 21](#_Toc192484397)

[4 Card Minidriver API Reference 22](#_Toc192484398)

[4.1 Initialization and Deconstruct 22](#_Toc192484399)

[4.1.1 CardAcquireContext 22](#_Toc192484400)

[4.1.2 CardDeleteContext 25](#_Toc192484401)

[4.2 Card PIN Operations 25](#_Toc192484402)

[4.2.1 Data Structures and Enumerations 25](#_Toc192484403)

[4.2.2 CardAuthenticatePin 30](#_Toc192484404)

[4.2.3 CardGetChallenge 31](#_Toc192484405)

[4.2.4 CardAuthenticateChallenge 32](#_Toc192484406)

[4.2.5 CardDeauthenticate 33](#_Toc192484407)

[4.2.6 CardAuthenticateEx 34](#_Toc192484408)

[4.2.7 CardGetChallengeEx 36](#_Toc192484409)

[4.2.8 CardDeauthenticateEx 37](#_Toc192484410)

[4.2.9 CardChangeAuthenticatorEx 38](#_Toc192484411)

[4.2.10 CardUnblockPin 39](#_Toc192484412)

[4.2.11 CardChangeAuthenticator 40](#_Toc192484413)

[4.3 Public Data Operations 41](#_Toc192484414)

[4.3.1 CardCreateDirectory 42](#_Toc192484415)

[4.3.2 CardDeleteDirectory 43](#_Toc192484416)

[4.3.3 CardReadFile 44](#_Toc192484417)

[4.3.4 CardCreateFile 45](#_Toc192484418)

[4.3.5 CardGetFileInfo 46](#_Toc192484419)

[4.3.6 CardWriteFile 47](#_Toc192484420)

[4.3.7 CardDeleteFile 48](#_Toc192484421)

[4.3.8 CardEnumFiles 49](#_Toc192484422)

[4.3.9 CardQueryFreeSpace 50](#_Toc192484423)

[4.4 Card Capabilities (Minidriver Version 5 and Earlier) 50](#_Toc192484424)

[4.4.1 Defines and Data Structures 51](#_Toc192484425)

[4.4.2 CardQueryCapabilities 51](#_Toc192484426)

[4.5 Card and Container Properties 52](#_Toc192484427)

[4.5.1 Defines and Data Structures 52](#_Toc192484428)

[4.5.2 CardGetContainerProperty 53](#_Toc192484429)

[4.5.3 CardSetContainerProperty 54](#_Toc192484430)

[4.5.4 CardGetProperty 55](#_Toc192484431)

[4.5.5 CardSetProperty 58](#_Toc192484432)

[4.6 Key Container 59](#_Toc192484433)

[4.6.1 CardCreateContainer 59](#_Toc192484434)

[4.6.2 CardDeleteContainer 61](#_Toc192484435)

[4.6.3 CardGetContainerInfo 62](#_Toc192484436)

[4.7 Cryptographic Operations 63](#_Toc192484437)

[4.7.1 CardRSADecrypt 63](#_Toc192484438)

[4.7.2 CardConstructDHAgreement 64](#_Toc192484439)

[4.7.3 CardDeriveKey 65](#_Toc192484440)

[4.7.4 CardDestroyDHAgreement 67](#_Toc192484441)

[4.7.5 CardSignData 68](#_Toc192484442)

[4.7.6 CardQueryKeySizes 70](#_Toc192484443)

[5 File System Requirements 71](#_Toc192484444)

[5.1 File Naming Requirements 71](#_Toc192484445)

[5.2 File System Virtualization 71](#_Toc192484446)

[5.3 Physical Card Data Layout 71](#_Toc192484447)

[5.4 Logical Data Layout 71](#_Toc192484448)

[5.4.1 Card Identifier 71](#_Toc192484449)

[5.4.2 Application Directory 72](#_Toc192484450)

[5.4.3 Cache File 72](#_Toc192484451)

[5.4.4 Container Map File 73](#_Toc192484452)

[5.5 Data Layout Summary 75](#_Toc192484453)

[5.6 File Access Control 75](#_Toc192484454)

[5.6.1 Known Principals 75](#_Toc192484455)

[5.6.2 Directory Access Conditions 76](#_Toc192484456)

[5.6.3 File Access Operations 76](#_Toc192484457)

[5.6.4 File Access Conditions 77](#_Toc192484458)

[5.6.5 Some Clarifying Notes on the Directory and File Access Conditions 78](#_Toc192484459)

[6 Card Requirements 79](#_Toc192484460)

[6.1 What a “Blank Card” Is 79](#_Toc192484461)

[6.2 Card “Creation” 79](#_Toc192484462)

[7 Developer Notes and Guidelines 80](#_Toc192484463)

[7.1 Challenge/Response Method of Unblocking Smart Card PIN 80](#_Toc192484464)

[7.2 Enhanced PIN Support 80](#_Toc192484465)

[7.3 Session PINs and Secure PIN Channel 80](#_Toc192484466)

[7.4 Read-Only Cards 82](#_Toc192484467)

[7.5 Cache Modes 83](#_Toc192484468)

[7.6 Challenge/Response Mechanism 84](#_Toc192484469)

[7.7 Interoperability with msroots 86](#_Toc192484470)

[7.8 Group Policy Settings for Microsoft Base Smart Card Crypto Service Provider 88](#_Toc192484471)

[7.9 Group Policy Settings for Microsoft CNG Smart Card Key Storage Provider 89](#_Toc192484472)

[7.10 Known Issues 89](#_Toc192484473)

# 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 CSP by encapsulating most of the complex cryptographic operations from the card minidriver developer.

In Windows Vista®, applications have the option of using the next-generation Microsoft Cryptographic API (CNG) for smart card–based cryptographic services. As part of the elliptic curve cryptography effort for Windows Vista, ECC smart cards are supported in the new cryptographic framework. Applications and interfaces that consume existing RSA card minidrivers through the legacy CAPI subsystem continue to work the same way.

Alternatively, RSA card minidrivers can be registered under the new Smart Card Key Storage Provider (KSP) so that they can be called via 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, though solely to expose RSA-only features.

Applications use CAPI for smart card–based cryptographic services. CAPI, in turn, routes these requests to the appropriate Cryptographic Service Provider (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. The CNG-based smart card KSP is a separate consumer, similar to the Base CSP (which is RSA only), for supporting ECC-only and ECC/RSA dual-mode smart cards in Windows Vista. Ultimately, the intention is for all new smart cards—RSA, ECC, and whatever follows—to be supported by the new architecture. It splits the implementation of the CSP into two parts:

* The Microsoft Smart Card Base CSP/KSP (the common part), which includes functionality for hashing, symmetric, and public key cryptographic operations as well as PIN entry and caching.
* A series of plug-ins, called “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 Smart Card Resource Manager (SCRM) that similarly abstracts the characteristics of a variety of smart card readers.

The portion left to smart card vendors is to implement a card minidriver, a reasonably limited interface layer that provides an abstraction of the card to the Microsoft Smart Card Base CSP/KSP 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. The following diagrams 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

## What’s New in Version 6

### Enhanced Support for PINs

Version 6 of the smart card minidriver specification enhances the support for PINs. This version introduces a new concept of a logical PIN object. A developer can use the PIN object architecture to control PIN prompting in Windows and enable a flexible and wide set of scenarios. The PIN object may or may not correspond to an actual physical PIN on the card and should be viewed as a way for a card minidriver to control PIN-related behavior in Windows.

Through a new set of APIs, a card minidriver developer can now:

* Support cards that use more than 2 PINs (up to 8 PINs in total).
* Control how each PIN is cached in Windows (cached, not cached, or cached for a specific length of time).
* Return a session PIN to Windows to cache instead of the actual PIN.
* Control what strings appear to the user during a PIN prompt.
* Indicate to Windows to prompt for a PIN when a specific key is requested.
* Allow access to a card or container without a PIN prompt (empty PIN).

For more information, see section , “.”

For reference information, see section , “.”

New APIs added in this version include:



**Important Note:**

Not all provisioning systems support multiple PINs; consequently, care must be taken when applying PINs on keys that can be updated in the field by a card provisioning system.

### Support for Read-Only Cards

Smart cards are considered read-only when Windows cannot write specific cache data to the card. The definition of a read-only card is included in section , “.”

Version 6 adds new modes of data caching and enables a card minidriver to control those cache modes. For more details, see section , “API .”

With these new data cache modes, along with proper virtualization of other system files (see section , “”), it is now possible to develop a card minidriver that is fully compatible with a read-only smart card.

For more information, see section , “,” and section , “.”

### Secure PIN Channel

Secure PIN channel is a feature in Windows Vista SP1 that enables a secure PIN prompt followed by establishment of a secure channel between Windows and the smart card for PIN authentication. Secure PIN channel protects the card PIN against eavesdropping while traveling through operating system component and while transmitted to the card.

Secure PIN prompt means that the user is requested to press ALT+CTL+DEL before prompted for the PIN following a visual experience identical to Windows logon. Secure PIN prompt reduces the risk of PIN harvesting by a spoof PIN prompt dialog box.

Secure PIN channel can be controlled and triggered by Common Criteria group policy setting and also by a specific attribute on the PIN object.

For more information on secure PIN channel, see section , “.”

The new API related to this feature includes section , “.”

### External PIN Support

An external PIN support is a PIN that is collected off the PC from the user. Examples of an external PIN scenario include:

* A PIN is collected on a PIN PAD reader.
* A smart card has a fingerprint reader attached to it and performs a match on a card with a fingerprint template as an alternative to a PIN.

In an external PIN mode, whenever PIN authentication to a smart card is required, Windows does not prompt the user for a PIN but rather calls the minidriver's authentication API immediately without any notification to the user. It is expected that the actual authentication and PIN collection occur without operating system involvement.

Optionally, and subject to specific restrictions, the minidriver is allowed to display its own user interface (UI) to instruct the user to perform specific actions in relationship to PIN collection. It is not expected that such UI will be used to actually collect a PIN from the user, but rather to direct the user that Windows is waiting for a PIN to be collected externally. A minidriver is not allowed to display UI when the context is silent mode and is expected to use a specific window handle to create UI elements. More information can be found in section , “,” and section , “.”

Cards that can return a temporary session PIN may return such a PIN to Windows for subsequent caching. In such a case, Windows presents the session PIN for any further card authentication until the card invalidates the session PIN. For more information, see section , “.”

# Card-Specific Minidriver Details

## Overview

The card-specific minidriver is the lowest logical interface layer in the Microsoft Smart Card Base CSP/CNG KSP. This minidriver allows the Card Interface layer to speak directly with a specific type of card by using SCRM.

The card minidriver is a DLL library 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 as well as a table of function pointers that is used to facilitate communication between the upper layer and the card minidriver. For details concerning this context structure, see the description of CardAcquireContext in the API reference chapter of this specification.

## Related Document

The following document provides additional information relevant to this specification:

* CARDMOD.H – C language header file, Microsoft card minidriver API (available within the Microsoft CNG SDK package).

## 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 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 detailed in this specification.

### Transaction Management

* A card minidriver should assume that transactions are handled by the caller, provided that it is using SCRM to access the card.
* The card minidriver can assume that all entry points except CardDeleteContext are called 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 DeleteContext 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. As such, 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 the naming of 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 a consistent error handling, response to failure, and consistent behavior for card minidrivers, these conventions should be followed:

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

In addition, the errors returned by the functions described in the following sections should be from SCARD\_\* category (winerror.h). For example, we recommend using 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 (see section 4.2.1.2, “SECRET\_TYPE (enumeration)”).

### 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 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 by means of 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 for cases in which writing data to the card is not feasible. For more on how to control data caching, see section , “,” for the CP\_CARD\_CACHE\_MODE property.

## 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) supported, and define the current version as the level for which this card minidriver was designed and for which all of the card-minidriver-set structure items are guaranteed to be valid on a successful return from CardAcquireContext. The current version must be >= Minimum version.

When the calling application calls CardAcquireContext(), it specifies the desired version that it wants to load. This requested version is set in the dwVersion element 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, then the card minidriver should set the dwVersion 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 (4)

    // 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, then it is the responsibility of the calling application to handle this appropriately.

After the dwVersion is set in the call to CardAcquireContext(), it can be assumed that it will not be changed by either the caller or the card minidriver while 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 containing a dwVersion of 0, then this must be treated as a dwVersion of 1.

# Manifest and General Imports

These operations are general in scope, being called by the card management applications and by the Microsoft Smart Card Base CSP/CNG KSP. These operations manipulate data of general interest to any application on the card, including 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 detailed information, see “DllMain” in the Platform SDK:

BOOL WINAPI DllMain(

IN HANDLEhinstDLL,

IN DWORDdwReason,

IN LPVOIDlpvReserved

);

Input:

hinstDLL A handle to this DLL instance supplied by the caller.

dwReason DLL\_PROCESS\_ATTACH,DLL\_PROCESS\_DETACH, DLL\_THREAD\_ATTACH, or DLL\_THREAD\_DETACH.

(winbase.h).

lpvReserved See SDK documentation for details.

Output:

Return value TRUE on 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.**

### DllRegisterServer and DllUnregisterServer

DllRegisterServer and DllUnregisterServer are no longer called on Windows Vista. On Windows XP, registration of the card minidriver is accomplished by Registry File.

### Registration Mechanisms

For card minidrivers that are written specifically for Base CSP to work on Windows releases earlier than Windows Vista, only Registry file must be modified. This information is detailed in section 3.1.4, “Registry File.”

Windows Vista introduced a mechanism for DLL registration through the use of manifests. DLLs on the system are to be registered by using this mechanism, so DllRegisterServer and DllUnregister are deprecated for Windows Vista. Refer to the manifest specification for details about their structure and creation. Card minidrivers, however, can also be used on down-level systems. Additionally, card minidrivers can be loaded by both CAPI Base-CSP and the CNG Smart Card Key Storage Provider, and so must register one or both of the associated provider keys. The following table, together with the details in subsequent subsections, details the necessary registration information.

|  | **Windows XP and Windows 2003** | **Windows Vista** |
| --- | --- | --- |
| CAPI-1 | Registry file  Must set “Crypto Provider” registry key. | Manifest  Must set “Crypto Provider” registry key. |
| CNG | Not supported | Manifest  Must set “Smart Card Key Storage Provider” key. |

### Registry File

The registry file should have the following format for each card and be named VENDOR.reg:

Windows Registry Editor Version 5.00

[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"

## Imports from the Smart Card Base CSP

When the CPS/KSP calls CardAcquireContext (see the following for details), it passes to the card minidriver a structure that contains among other things six function pointers for the card minidriver to use. These pointers provide support for data caching and memory management operations, so that all card minidrivers can use an optimized implementation that the CSP/KSP provides instead of duplicating these complex functions in each minidriver.

This section describes these functions, using text 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 Microsoft Smart Card Base CSP/CNG KSP provides.

#### PFN\_CSP\_ALLOC

Description:

The alloc function is called via 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 realloc function is called via 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 free function is called via 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

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

Description:

The cache add file function is called via a pointer in the CARD\_DATA structure when the card minidriver wants to add a file to the set of data that is the CSP/KSP caches. 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 PBYTE pbData,

IN DWORD cbData

);

In:

wszTag The name of the file to add.

dwFlags Reserved—must be zero.

pbData Pointer to the buffer containing the data. The buffer is allocated and freed by the card minidriver.

cbData Byte count of the data pointed to by pbFileData.

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

Out:

Return value Zero on success; otherwise, nonzero.

Comments:

Files on the card that are used only internally by the card minidriver can take advantage of the 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 cache lookup file function is called via 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,

OUT PBYTE \*ppbData,

OUT PDWORD pcbData

);

In:

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

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 pbFileData.

Return value Zero on success; otherwise, nonzero.

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

Description:

The cache delete function is called via a pointer in the CARD\_DATA structure when the card minidriver wants to delete a file from the CSP/KSP’s cache:

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 supplied by the 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 padding 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, but 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,

OUT PBYTE\* ppbPaddedBuffer);

In:

pSigningInfo Contains buffer to pad plus needed algorithm information.

cbMaxWidth Maximum width of pad-able buffer (in bytes).

Out:

pcbPaddedBuffer Populated with the count of bytes in padded buffer.

ppbPaddedBuffer New buffer containing 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 SC Base CSP and KSP-supplied call backs and PSS padding with KSP-supplied callbacks.

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

Description:

This callback is set by the KSP for Version 5 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, then this function returns failure. The function signature is as follows:

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

IN PCARD\_DATA pCardData,

IN PVOID hSecretAgreement,

OUT BYTE\* pbSecretAgreementIndex,

IN DWORD dwFlags);

DWORD WINAPI CspGetDHAgreement(

\_\_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 pfnCardDeriveKey function.

hSecretAgreement This should be the KDF\_VALUE\_SECRET parameter passed into pfnCardDeriveKey via the pParameterList field 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 pfnCardDeriveKey.

dwFlags This is reserved and must be 0.

Ephemeral nature of secret agreements on the card:

The lifetime of hAgreedSecret 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.

## Imports from Smart Card Resource Manager

Smart Card Resource Manager (SCRM) provides the preferred mechanism of communicating with smart cards. It provides for arbitration of access and other functions to manage the availability of smart cards to applications. 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.

To locate this topic in the Platform SDK, search the SDK for “Smart Card Resource Manager API.”

# Card Minidriver API Reference

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

## Initialization and Deconstruct

### CardAcquireContext

Description:

This function is used to initialize communication between the 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(

IN OUT PCARD\_DATA pCardData,

IN DWORD dwFlags

);

In:

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

dwFlags Reserved—must be zero.

Out:

Return value Zero on success; otherwise, nonzero. List of expected error return values is shown later in this specification (in addition to standard behavior described in section 2.3.2.2).

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 with status 0 if successful or a nonzero value if an error was encountered.

The Base CSP/KSP provides six important function exports for the card minidriver to use: three for using the data caching mechanisms provided by the Base CSP/KSP and three to manage memory allocation. Many of the 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 defined by the Base CSP/KSP.

dwVersion 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 need to be loaded and recognized as such.

**Note:** When implementing, card minidriver vendors must not assume single context to manage communications with the card.

The following is the structure of the CARD\_DATA, taken from cardmod.h:

#define CARD\_DATA\_CURRENT\_VERSION 6

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; // memory mgmt

PFN\_CSP\_REALLOC pfnCspReAlloc; // memory mgmt

PFN\_CSP\_FREE pfnCspFree; // memory mgmt

PFN\_CSP\_CACHE\_ADD\_FILE pfnCspCacheAddFile; // cache

PFN\_CSP\_CACHE\_LOOKUP\_FILE pfnCspCacheLookupFile; // cache

PFN\_CSP\_CACHE\_DELETE\_FILE pfnCspCacheDeleteFile; // cache

PVOID pvCacheContext; // cache

PFN\_CSP\_PAD\_DATA pfnCspPadData; // padding

SCARDCONTEXT hSCardCtx;

SCARDHANDLE hSCard;

PVOID pvVendorSpecific; // pvoid for card minidriver use

// These members are initialized by the card minidriver

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\_SUBMIT\_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;

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;

// version 5 additions below here

PFN\_CARD\_DERIVE\_KEY pfnCardDeriveKey;

PFN\_CARD\_DELETE\_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;

} CARD\_DATA, \*PCARD\_DATA;

Note that the ATR value is assumed to be unique per card. The ATR value must be sufficient for loading the appropriate card-specific minidriver that works with the card.

The pfnCardDeriveKey, pfnCardDestroyDHAgreement, and pfnCspGetDHAgreement fields are called out in later sections. As of Version 5 of this specification, the necessary modifications to the pfnCardConstructDHAgreement function are handled via versioning the structure that is associated with that function.

If pfnCspPadData is NULL, the minidriver can expect that no decrypt functionality will be invoked by the calling application.

Errors:

When passing an invalid pbAtr field to CardAcquireContext, it should return SCARD\_E\_UNKNOWN\_CARD.

When passing NULL as pbAtr to CardAcquireContext, 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 proper length) in the cbAtr field 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, then SCARD\_E\_UNKNOWN\_CARD must be returned.

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.

hSCardCtx and hSCard handles should be checked against 0 and should return SCARD\_E\_INVALID\_HANDLE in such case.

In addition, the conventions specified in section 2.3.2.2 must be respected.

### CardDeleteContext

Description:

CardDeleteContext() reverses the effect of CardAcquireContext(), severing communication between the CSP/KSP and the card minidriver and performing any needed deallocations and cleanup:

DWORD WINAPI CardDeleteContext(

IN 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, meaning “personal identification number,” due to its first use on the numeric keypad of ATM machines. Some other industry documentation generalizes this term to CHV, meaning “card holder verification,” and it is understood that its form is not just numeric, but anything that the user can supply given the means at his or her disposal. The value 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 either be securely transmitted to the card without any possibility of disclosure or else be entirely ephemeral so that it has no value outside of 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 associated with roles and PIN\_SET that is used for a bit-mask with PIN identifiers. Also, we discontinue 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 will be explained later in the 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 section 7.4.

#### SECRET\_TYPE (enumeration)

The following enumeration describes which type the PIN is:

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)

This 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

} SECRET\_PURPOSE;

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

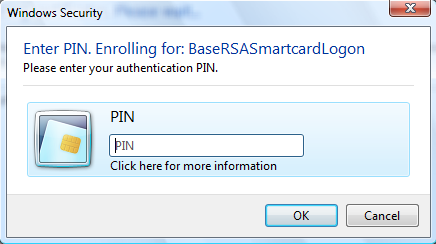


Figure 3. PIN Dialog Box in Windows Vista

The first string in Figure 3 (“Enter PIN. Enrolling for: BaseRSASmartcardLogon”) is provided by the calling application to provide application context. In the absence of an application context string, the dialog box displays a standard text.

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

* Default context strings

By default, smart card base CSP displays the following predefined strings, 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.” |

* Custom strings

It is possible to 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 corresponding 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 the parsing of special characters within the string.**

**Note: Including two different custom strings for the same locale results in the last custom string to be picked up.**

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

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

This enumeration describes the PIN caching policy that is to be associated with this PIN:

typedef enum

{

PinCacheNormal = 0,

PinCacheTimed,

PinCacheNone

} PIN\_CACHE\_POLICY\_TYPE;

The following table is a description of how the BaseCSP acts upon the three different cache modes.

|  |  |
| --- | --- |
| **PinCacheNormal** | For this mode, the PIN is cached by the BaseCSP per process per logon ID. |
| **PinCacheTimed** | For this mode, the PIN is invalidated after an indicated period of time (value given in seconds.) This has been 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 won’t be used after it has expired. The PIN is encrypted in memory to keep it protected. |
| **PinCacheNone** | When the PIN cannot be cached, BaseCSP never adds the PIN to the cache. When the BaseCSP or 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 take place before the BaseCSP transaction time-out expires. |

**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 anything 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, as well as 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 info structure is obtained by the BaseCSP, it should be cached in the data cache similar to the way in which 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;

dwUnblockPermission is a bit-mask that describes which PINs have permission to unblock the PIN. The permission is based on a bit-wise ‘or’ of the specified PINs. For an unblock operation, any self-reference should be ignored by the card minidriver. The ROLE\_USER would have an update permission bit mask of 0x00000100, meaning it can be unblocked by ROLE\_ADMIN. ROLE\_ADMIN has an update permission of 0x00000000, meaning it cannot be unblocked. dwChangePermission is an analog to dwUpdatePermission that describes which PINs have access to change another PIN. For example, ROLE\_USER has a change permission bit mask of 0x00000010, and ROLE\_ADMIN has 0x00000100.

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

Note that 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:

CardAuthenticatePin 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 behalf of the user. 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 PBYTE pbPinData,

IN DWORD cbPinData,

OUT OPTIONAL PDWORD pdwcAttemptsRemaining

);

Input:

pCardData Context information for the call. See CardAcquireContext().

pwszUserId String indicating the card principal associated with the PIN.

pbPinData Pointer to a buffer containing the PIN information.

cbPinData 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:

The allowable 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 API returns SCARD\_E\_INVALID\_PARAMETER for external PINs or empty PINs. This API is deprecated by CardAuthenticateEx, which is the recommended API 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, and 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 pointer to pbPinData is NULL, the call fails with the SCARD\_E\_INVALID\_PARAMETER error code.

The card minidriver should implement all card data validation tests on pbPinData and cbPinData as follows:

* If pbPinData is NULL, it should return SCARD\_E\_INVALID\_PARAMETER.
* If inconsistencies are found (such as the actual PIN doesn’t match the PIN length (PIN length is 0xFFFFFFFF and so on), then it should return SCARD\_W\_WRONG\_CHV without presenting a PIN to the card and without decrementing the PIN counter.

### 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,

OUT PBYTE \*ppbChallengeData,

OUT PDWORD pcbChallengeData

);

Input:

pCardData Context information for the call. See CardAcquireContext().

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:

This authentication technique is normally 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 elect not to use the challenge value. It is only significant if an authentication has attempted to use it. It is discarded if the next command to the card is not an authentication attempt using it (see 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:

Conventions specified in section 2.3.2.2 for error handling should be followed.

### CardAuthenticateChallenge

Description:

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

DWORD WINAPI CardAuthenticateChallenge(

IN PCARD\_DATA pCardData,

IN PBYTE pbResponseData,

IN DWORD cbResponseData,

OUT OPTIONAL PDWORD pcAttemptsRemaining

);

Input:

pCardData Context information for the call. See CardAcquireContext().

pbResponseData Pointer to a buffer containing the response data corresponding 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:

This authentication technique is normally used for establishing 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, then 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.

In general:

* 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:

CardDeauthenticate 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 place NULL in the CARD\_DATA structure pointer for this function.

The CSP/KSP tests this pointer for NULL value before calling it. If it is found NULL, the 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. See CardAcquireContext().

pwszUserId String indicating the card principal to be deauthenticated.

dwFlags Reserved—must be zero.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

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

### CardAuthenticateEx

Description:

CardAuthenticateEx handles PIN authentication operations to the card. This function replaces CardAuthenticate of previous versions of these specifications and adds support for external PINs (a PIN collected on a device), challenge/response PIN, secure PIN channel, and a session PIN:

DWORD WINAPI CardAuthenticateEx(

IN PCARD\_DATA pCardData,

IN PIN\_ID PinId,

IN DWORD dwFlags,

IN PBYTE pbPinData,

IN DWORD cbPinData,

OUT OPTIONAL PBYTE \*ppbSessionPin,

OUT OPTIONAL PDWORD pcbSessionPin,

OUT OPTIONAL PDWORD pdwcAttemptsRemaining

);

Input:

pCardData Context information for the call. See CardAcquireContext().

PinId PIN identifier to be authenticated, for example, PIN\_4 or PIN\_USER.

dwFlags Flags. See remarks.

pbPinData Pointer to a buffer containing 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.

pdwcAttemptsRemaining 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 CardDeauthenticate or CardDeauthenticateEx is called or the card is reset (through the winscard API or the card losing power). Note that this does not apply if CARD\_AUTHENTICATE\_GENERATE\_SESSION\_PIN is specified.

The allowable values for PinId are ROLE\_USER, ROLE\_ADMIN or 3 through 7.

For any other PinId value, CardAuthenticatePinEx should return SCARD\_E\_INVALID\_PARAMETER.

**Failed authentication attempts:**

If an incorrect PIN is presented, this function returns SCARD\_W\_WRONG\_CHV, and 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.

Additionally, presenting an incorrect PIN for a particular role should result in that role being deauthenticated on the card, but should have no impact on other roles that are already authenticated to the card.

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

**Session PIN:**

Session PIN is defined as a temporary PIN, generated by the card, that expires upon termination of the session. Cards that support session PINs can now return the generated session PIN. If the pointer to ppbSessionPin is non-NULL and the card can generate a session PIN, the call should allocate ppbSessionPin to hold the session PIN. In such a case, pcbSessionPin should contain the length of the session PIN. Windows caches the session PIN (according to the PIN caching policy set in PIN\_INFO) and presents it to the card for the next call to CardAuthenticateEx.

Following a successful generation of a session, Windows calls CardAuthenticateEx and passes in CARD\_AUTHENTICATE\_SESSION\_PIN. In this case, the actual PIN is not passed and it is expected that the minidriver use the session PIN, passed in pbPinData and cbPinData, to authenticate to the card. If the CARD\_AUTHENTICATE\_GENERATE\_SESSION\_PIN flag is specified and either ppbSessionPin or pcbSessionPin is NULL, the call must fail with SCARD\_E\_INVALID\_PARAMETER.

If this session PIN is invalid, SCARD\_WRONG\_CHV should be returned, pdwcAttemptsRemaining 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:**

External PIN (ExternalPinType returned in PIN\_INFO) is defined as a PIN collected on a device other than the computer running Windows. For example, this could be a BIO match-on-card PIN. In this case, Windows does not prompt the user for a PIN but calls CardAuthenticateEx with an NULL value for pbData for the PIN. If a session PIN has been returned by the call, then that session PIN is passed in for subsequent calls.

In this call, a minidriver is allowed to display its own UI windows, as long as CARD\_PIN\_SILENT\_CONTEXT was not passed in dwFlags. The parent windows handle 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 UI when CARD\_SILENT\_CONTEXT was passed in results in operating system instability.

**Secure PIN channel:**

Secure PIN channel is enabled if one of the following conditions exists:

* Common Criteria group policy flag is on.
* If the card is requesting a secure PIN channel (see section , “,” and CP\_CARD\_PIN\_STRENGTH\_VERIFY property).
* dwFlags of PIN\_INFO structure 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 ALT+CTRL+DEL key combination.

When in secure PIN channel mode, the system calls CardAuthenticateEx from a trusted process and passes in CARD\_AUTHENTICATE\_GENERATE\_SESSION\_PIN in dwFlags along with the PIN in clear text. It is expected that the minidriver will establish a secure connection with the card and return a session PIN to the system. The system then passes the session PIN to the non-secure context process for authentication to the card.

It is imperative that the clear text PIN is handled securely when a CARD\_AUTHENTICATE\_GENERATE\_SESSION\_PIN flag is detected and never transmitted to the card.

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

### CardGetChallengeEx

Description:

A card principal can be authenticated by using either a PIN or 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. If the response is correct, the principal is authenticated to the card:

DWORD WINAPI CardGetChallengeEx(

IN PCARD\_DATA pCardData,

IN PIN\_ID PinId,

OUT PBYTE \*ppbChallengeData,

OUT PDWORD pcbChallengeData,

IN DWORD dwFlags

);

Input:

pCardData Context information for the call. See CardAcquireContext().

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.

If PIN\_ID is not ChallengeResponsePinType, return SCARD\_E\_INVALID\_PARAMETER.

Comments:

This authentication technique is normally 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 elect not to 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 using it (see 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:

Conventions specified in section 2.3.2.2 for error handling should be followed.

### CardDeauthenticateEx

Description:

CardDeauthenticateEx 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 which case the 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 PinSet,

IN DWORD dwFlags

);

Input:

pCardData Context information for the call. See CardAcquireContext().

PinSet Set of PINs to be deauthenticated.

dwFlags Reserved—must be zero.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

The PinSet can contain multiple PINs for deauthentication. For example, a PinSet containing the value 0x03 means to deauthenticate ROLE\_USER and ROLE\_ADMIN. A value of oxFF means to deauthenticate all PINs that are currently authenticated. If the ROLE\_EVERYONE bit is set in the PinSet, it should be ignored.

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

Upon success, all specified PINs must be deauthenticated.

### 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 PBYTE pbAuthenticatingPinData,

IN DWORD cbAuthenticatingPinData,

IN PIN\_ID dwTargetPinId,

IN PBYTE pbTargetData,

IN DWORD cbTargetData,

IN DWORD cRetryCount,

OUT OPTIONAL PDWORD pcAttemptsRemaining

);

Input:

pCardData Context information for the call. See CardAcquireContext().

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

dwAuthenticatingPinId PIN identifier to be authenticated.

pbAuthenticatingPinData Pointer to a byte buffer containing PIN data.

cbAuthenticatingPinData Byte count of the PIN data.

dwTargetPinId PIN identifier to be updated.

pbTargetData Pointer to a byte buffer containing 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 cases where the authenticator is to be set.

The allowable values for the PinId 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 by means of CardGetChallenge.

For a description of the usage of pdwcAttemptsRemaining, see comments under CardAuthenticatePin, earlier in this specification. If 0 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 regarding 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, then successful completion should leave the card in a deauthenticated state for both the unblocked PIN and the authenticating PIN.

### CardUnblockPin

Description:

CardUnblockPin is used to unblock a card that has become blocked by an excessive number of incorrect PIN entry attempts. The unblock function is atomic in that authentication and the unblocking of 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 PBYTE pbAuthenticationData,

IN DWORD cbAuthenticationData,

IN PBYTE pbNewPinData,

IN DWORD cbNewPinData,

IN DWORD cRetryCount,

IN DWORD dwFlags

);

Input:

pCardData Context information for the call. See CardAcquireContext().

pwszUserId Card principal associated with the PIN.

pbAuthenticationData Pointer to CardGetChallenge() response data.

cbAuthenticationData Byte count of the authentication data.

pbNewPinData Pointer to buffer containing the new PIN to be set.

cbNewPinData Byte count of the data pointed to by pbNewPinData.

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:

The authentication data for the operation is a response that corresponds to the challenge acquired by a call to CardGetChallenge(). This is distinguished by the CARD\_ AUTHENTICATE \_PIN\_CHALLENGE\_RESPONSE flag value identifying the buffer as containing a response to a challenge.

Example: a typical scenario is unblocking the user PIN by using administrator challenge/response.

In addition, refer to section 3.3.3 for general conventions and guidelines for using PIN and challenge/response authenticators. For administrators, challenge/response support is mandatory. For users, challenge/response support is not supported.

If 0 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. In that case, 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 PBYTE pbCurrentAuthenticator,

IN DWORD cbCurrentAuthenticator,

IN PBYTE pbNewAuthenticator,

IN DWORD cbNewAuthenticator,

IN DWORD cRetryCount,

IN DWORD dwFlags,

OUT OPTIONAL PDWORD pcAttemptsRemaining);

);

Input:

pCardData Context information for the call. See CardAcquireContext().

pwszUserId Card principal associated with the PIN (administrator or user).

pbCurrentAuthenticator Pointer to a buffer containing the current PIN information or a response to a previously issued challenge (see dwFlags later in this specification).

dwcbCurrentAuthenticator Byte count of the current PIN/response.

pbNewAuthenticator Pointer to a buffer containing 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 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:

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, so this function is the logical choice to use when individualizing a card.

The allowable 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 supported values are CARD\_ AUTHENTICATE\_PIN\_PIN and CARD\_AUTHENTICATE\_PIN\_CHALLENGE\_RESPONSE. In the latter case, the caller must have previously obtained a challenge value from the card by means of CardGetChallengeand this response is put into the pbCurrentAuthenticator field.

For a description of the usage of pdwcAttemptsRemaining, see comments under CardAuthenticatePin, earlier in this specification. If 0 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 regarding 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), but the remainder of 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 are not allowed by Windows file and directory-naming convention (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 entails the following steps:

* Creating the application’s storage subdirectory.
* Creating a DWORD entry for the application in the cache file.
* 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 needed to create directories, create files, and write files.

Note that all of the 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 normally created for segregating the files belonging to a single application on the card. As an example, the files belonging 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. See CardAcquireContext().

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 are:

* InvalidAc
* UserCreateDeleteDirAc
* AdminCreateDeleteDirAc

For a detailed description of what each of these means for file access, see section 5.

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 named the same, ERROR\_FILE\_EXISTS should be returned.

If calling CardCreateDirectory without authenticating to the card first, it is expected to fail with an SCARD\_W\_SECURITY\_VIOLATION error.

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

If the name specified by pszDirectoryName is longer than the maximum length defined for file/directory names, then 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 pszDirectory,

);

Input:

pCardData Context information for the call. See CardAcquireContext().

pszDirectory Name of the directory.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

Returns invalid argument error for NULL passed as directory name.

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

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

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

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

### CardReadFile

Description:

CardReadFile reads the entire file at the specified location into the user-supplied buffer:

DWORD WINAPI CardReadFile(

IN PCARD\_DATA pCardData,

\_\_in LPSTR pszDirectoryName,

\_\_in LPSTR pszFileName,

IN DWORD dwFlags,

OUT PBYTE \*ppbData,

OUT PDWORD pcbData);

);

Input:

pCardData Context information for the call. See CardAcquireContext().

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 containing 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 containing the returned data is allocated by the card minidriver and freed by the CSP/KSP.

See comments regarding file sizes under CardWriteFile() later in this specification.

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

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

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

### CardCreateFile

Description:

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

DWORD WINAPI CardCreateFile(  
 IN PCARD\_DATA pCardData,

\_\_in LPSTR pszDirectoryName,

\_\_in LPSTR pszFileName,

IN DWORD cbInitialCreationSize,

IN CARD\_FILE\_ACCESS\_CONDITION AccessCondition);

);

Input:

pCardData Context information for the call. See CardAcquireContext().

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 where a later file write fails for lack of space after the file was successfully created. Rules for file name are as defined in Section 0. If pszFileName is NULL or an empty string, then an SCARD\_E\_INVALID\_PARAMETER error must be returned. If cbInitialCreationSize is greater than the free space on the card, then 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. These are:

* InvalidAc
* EveryoneReadUserWriteAc
* UserWriteExecuteAc
* EveryoneReadAdminWriteAc.

For a detailed description of what each of these means for file access, see section 5, “File System Requirements.”

Errors:

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

If CardCreateFile is called on a nonexisting directory, then 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, then an SCARD\_W\_SECURITY\_VIOLATION error code must be returned.

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

### CardGetFileInfo

Description:

This function is used to retrieve information about a file, specifically its size and access control list (ACL) information:

DWORD WINAPI CardGetFileInfo(

IN PCARD\_DATA pCardData,

\_\_in LPSTR pszDirectoryName,

\_\_in LPSTR pszFileName,

OUT PCARD\_FILE\_INFO pCardFileInfo);

);

Input:

pCardData Context information for the call. See CardAcquireContext().

pszDirectoryName Name of 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:

Return value Zero on success; otherwise, nonzero.

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

Comments:

CardGetFileInfo fails if the specified file doesn’t exist.

See comments regarding file sizes under CardWriteFile() later in this specification.

The file information returned is contained in the following structure:

typedef struct \_CARD\_FILE\_INFO

{

DWORD dwVersion; // version of this structure

DWORD cbFileSize;

CARD\_FILE\_ACCESS\_CONDITION AccessCondition;

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

The file size 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 is 0, even if that file was created with a nonzero file size).

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

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

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

### CardWriteFile

Description:

CardWriteFile 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 LPSTR pszDirectoryName,

\_\_in LPSTR pszFileName,

IN DWORD dwFlags,

IN PBYTE pbData,

IN DWORD cbData);

);

Input:

pCardData Context information for the call. See CardAcquireContext().

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 containing 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, resulting in significant loss of useful space. This is because it is usually infeasible to implement a reorganizing memory manager for card storage. For this reason, implementations can choose not to “shrink” a file if its size has been decreased. *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) specified through CardWriteFile is larger than the current file size specified through CardCreateFile, it should succeed, unless the card is out of space, in which case 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 nonexisting file, then CardWriteFile should fail with SCARD\_E\_FILE\_NOT\_FOUND.

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

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

### CardDeleteFile

Description:

CardDeleteFile deletes the specified file. If the file doesn’t exist, the returned Status value should indicate that the file did not exist:

DWORD WINAPI CardDeleteFile(

IN PCARD\_DATA pCardData,

\_\_in LPSTR pszDirectoryName,

\_\_in LPSTR pszFileName,

IN DWORD dwFlags);

);

Input:

pCardData Context information for the call. See CardAcquireContext().

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, then an SCARD\_W\_SECURITY\_VIOLATION error code must be returned.

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

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

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

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

### CardEnumFiles

Description:

CardEnumFiles() returns name information about available files in a directory as a multistring list:

DWORD WINAPI CardEnumFiles(

IN PCARD\_DATA pCardData,

\_\_in LPSTR pszDirectoryName,

\_\_out\_ecount(\*pdwcbFileName)

LPSTR \*pmszFileNames,

\_\_out LPDWORD pdwcbFileName,

IN DWORD dwFlags

);

Input:

pCardData Context information for the call. See CardAcquireContext().

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, then an SCARD\_E\_FILE\_NOT\_FOUND error code should be returned.

Comments:

CardEnumFiles should check the dwFlags value to ensure that it is 0. 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 nonexisting directory, then an SCARD\_E\_DIR\_NOT\_FOUND error code must be returned.

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

### CardQueryFreeSpace

Description:

CardQueryFreeSpece determines the amount of available card storage space:

DWORD WINAPI CardQueryFreeSpace(

IN PCARD\_DATA pCardData,

IN DWORD dwFlags,

OUT PCARD\_FREE\_SPACE\_INFO pCardFreeSpaceInfo

);

Input:

pCardData Context information for the call. See CardAcquireContext().

dwFlags Reserved—must be zero.

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

Output:

Return value Zero on success; otherwise, nonzero.

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

Comments:

Free space information is returned in the following structure:

typedef struct \_CARD\_FREE\_SPACE\_INFO

{

DWORD dwVersion;

DWORD dwBytesAvailable;

DWORD dwKeyContainersAvailable;

DWORD dwMaxKeyContainers;

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

These may be approximate values in some cases. 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 dwVersion must be set by the caller. Current defined values are:

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

In the CARD\_FREE\_SPACE\_INFO structure discussed earlier, values that are not known should be set to CARD\_DATA\_VALUE\_UNKNOWN for each of the three fields 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 BaseCSP versions older than Version 6).

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

{

DWORD dwVersion;

BOOL fCertificateCompression;

BOOL fKeyGen;

} CARD\_CAPABILITIES, \*PCARD\_CAPABILITIES;

Members:

dwVersion Version of the structure in use.

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:

Queries the card and card-specific minidriver combination for the functionality provided at this level, such as certificate or file compression:

DWORD WINAPI CardQueryCapabilities(

IN PCARD\_DATA pCardData,

IN OUT PCARD\_CAPABILITIES pCardCapabilities

);

Input:

pCardData Context information for the call. See CardAcquireContext().

pCardCapabilities Card capabilities structure with version number set.

Output:

Return value Zero on success; otherwise, nonzero.

pCardCapabilities Filled-in pCardCapabilities struct.

Comments:

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

## Card and Container Properties

### Defines and Data Structures

The following defines functions specific to querying capabilities of a smart card and functions specific to discovering attributes of a container of key material on the card. This section is applicable beginning with Version 6 of the card minidriver specification. However, for backward compatibility reasons, functions such as CardQueryFreeSpace, CardQueryKeySizes, and CardQueryCapabilities must be implemented as well.

Card properties constants include:

#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"

Container properties constants include:

#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 BaseCSP/KSP and should not be used by the minidriver.

### CardGetContainerProperty

Description:

The function is modeled after the query functions of CAPI for keys. It takes a LPWSTR indicating which parameter is being requested. It then returns data written into pbData:

DWORD WINAPI CardGetContainerProperty(

IN PCARD\_DATA pCardData,

IN BYTE bContainerIndex,

IN LPWSTR wszProperty,

OUT 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 indicating 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 RFU.

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 must be freed by the caller using PFN\_CSP\_FREE. |
| CCP\_PIN\_IDENTIFIER | In this case, pbData contains a PIN\_ID describing the PIN identifier of the PIN associated with this container. |

### CardSetContainerProperty

Description:

The function is intended to set properties on containers. Only one container property is supported: CCP\_PIN\_IDENTIFIER:

DWORD WINAPI CardSetContainerProperty(

IN PCARD\_DATA pCardData,

IN BYTE bContainerIndex,

IN LPWSTR wszProperty,

IN PBYTE pbData,

IN DWORD dwDataLen,

IN DWORD dwFlags

);

Input:

pCardData Address of CARD\_DATA structure.

bContainerIndex Index to a key container on the card.

wszProperty LPWSTR indicating which property is requested.

pdData Byte pointer to data buffer containing the data.

dwDataLen DWORD indicating the data buffer length.

dwFlags Flags, currently RFU.

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. 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 case, pbData contains a DWORD describing the PIN identifier to the PIN that is associated with this container.  Although this function is not consumed by the BaseCSP/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 the user PIN. 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. |

### CardGetProperty

Description:

The function is modeled after the query functions of CAPI for keys. It takes a LPWSTR indicating which parameter is being requested. It then returns data written into pbData:

DWORD WINAPI CardGetProperty(

IN PCARD\_DATA pCardData,

IN LPWSTR wszProperty,

OUT PBYTE pbData,

IN DWORD cbData,

OUT PDWORD pdwDataLen,

IN DWORD dwFlags

);

Input:

pCardData Address of CARD\_DATA structure.

wszProperty LPWSTR indicating 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.

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. Implementing all of 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 to be returned, CardGetProperty should return ERROR\_INSUFFICIENT\_BUFFER.

**Important note**:   
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 BaseCSP 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, for example, 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; |
| CP\_CARD\_READ\_ONLY | BOOL | If True, all write operations are blocked at the CSP layer.  This flag also affects the data cache. If the card indicates that it is read only, the BaseCSP/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 due to lack of support for multiple PINs in Windows PKI enrollment client. |
| CP\_CARD\_GUID | BYTE[] | In this case, pbData is a buffer containing a unique GUID for the card. This value must exactly match the GUID contained in the “cardid” file. |
| CP\_CARD\_SERIAL\_NO | BYTE[] | In this case, pbData is a buffer containing a serial number for the card. The format of the serial number is opaque to the BaseCSP and is intended for other applications that query the card minidriver directly.  This is an optional property that may or may not be supported by the card. |
| CP\_CARD\_PIN\_INFO | PIN\_INFO | In this case, pbData is a PIN\_INFO structure containing information about the PIN. The dwFlags parameter contains the identifier of the PIN to return. |
| CP\_CARD\_LIST\_PINS | PIN\_SET | In this case, pbData contains a PIN\_SET indicating by a bit-mask what entities the card currently uses. |
| CP\_CARD\_AUTHENTICATED \_STATE | PIN\_SET | In this case, pbData contains a PIN\_SET indicating by a bit-mask what entities the card currently authenticates.  This is an optional property that may or may not be supported by the card. |
| CP\_CARD\_PIN\_STRENGTH \_VERIFY |  | In this case, pbData contains a bit mask 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. |

### CardSetProperty

Description:

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

DWORD WINAPI CardSetProperty(

IN PCARD\_DATA pCardData,

IN LPWSTR wszProperty,

IN PBYTE pbData,

IN DWORD dwDataLen,

IN DWORD dwFlags

);

Input:

pCardData Address of CARD\_DATA structure.

wszProperty LPWSTR indicating which property is being set.

pdData Byte pointer to data buffer containing the data.

dwDataLen DWORD indicating the data buffer length.

dwFlags Flags.

Output:

Return value Zero on success; nonzero on failure.

Comments:

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\_INVALID\_PARAMETER. 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 the previous section on CardGetProperty.

The following properties are read-only and are not supported by CardSetPropertyFunction:

* CP\_CARD\_FREE\_SPACE  
  CP\_CARD\_CAPABILITIES  
  CP\_CARD\_KEYSIZES  
  CP\_CARD\_LIST\_PINS  
  CP\_CARD\_AUTHENTICATED\_STATE

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, but the appropriate principal (ROLE\_ADMIN) is not authenticated.

To set card properties, the permission in the right columns must be satisfied.

| **wszProperty** | **pbData** | **Permission** |
| --- | --- | --- |
| CP\_CARD\_CACHE\_MODE | In this case pbData is a dword. There are three flags indicating 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 | Administrator |
| CP\_SUPPORTS\_WIN\_X509 \_ENROLLMENT | If False, enrollment operations is blocked at the CSP layer. | Administrator |
| CP\_CARD\_GUID | In this case, pbData is a buffer containing a unique GUID for the card. Whether updating the GUID by property or via the “cardid” file, retrieving the GUID by either means should always return the same value. | Administrator |
| CP\_CARD\_SERIAL\_NO | In this case, pbData is a buffer containing a serial number for the card.  This is an optional property that the card may or may not choose to support. | Administrator |
| CP\_CARD\_PIN\_INFO | In this case, pbData is a PIN\_INFO structure containing information about the PIN. The dwFlags parameter contains the identifier of the PIN to return. If the PIN\_INFO structure contains information that is not supported by the card minidriver, such as the PIN\_INFO\_REQUIRE\_SECURE\_ENTRY flag, the card minidriver should return SCARD\_E\_UNSUPPORTED. | Administrator |
| CP\_PARENT\_WINDOW | In this case, 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. | Everyone |
| CP\_PIN\_CONTEXT\_STRING | In this case, pbData is a LPWSTR containing 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. | Everyone |
| CP\_CARD\_PIN\_STRENGTH \_VERIFY | In this case, pbData contains a bit mask 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. | Administrator |

## Key Container

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

### CardCreateContainer

Description:

CardCreateContainer creates a new key container named by using a GUID generated by CAPI. For applications where 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. See CardAcquireContext().

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. Must be 0 for ECC keys when key is being generated on card. For RSA keys, this must specify the key bit length.

pbKeyData If KeyImport, a pointer to the passed key material, or else ignored for KeyGen.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

Containers are referred to in communication between the CSP/KSP and the card minidriver by index number. These index numbers are assigned by the CSP/KSP. To this end, the CSP/KSP maintains a map file (mscp\Map) on the card that lists the CAPI/CNG GUIDs for the containers used so far on the card. For a new container, the 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 doesn’t 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 via random key generation and import of 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,” typically returned from CryptExportKey. For a description of this format, see the documentation in the platform SDK for CryptExportKey and in “Base Provider Key BLOBs” and “Elliptic Curve Support.” RSA keys conform to CAPI key blob format. If the card supports the key type specified by dwKeySpec 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 passed is invalid or undefined, then a return value of SCARD\_E\_INVALID\_PARAMETER is expected. If the dwKeySpec value is defined but not supported, then a return value of SCARD\_E\_UNSUPPORTED\_FEATURE is expected.

Containers can be created only by users. Both administrators and users should be able to get information and delete containers. If an administrator attempts to create a container, the SCARD\_W\_WRONG\_CHV error should be returned.

Error checking is done based on the order of cost, that is, checks that can be done without communicating to the card first. This would include validating pbKeyData and dwKeySize parameters. SCARD\_E\_UNSUPPORTED\_FEATURE is checked first.

### CardDeleteContainer

Description:

CardDeleteContainer deletes the key container specified by its index value. This is done by deleting all of the 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. See CardAcquireContext().

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

dwReserved Must be zero.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

Deletes the key material associated with the indexed container. Certificates are deleted separately by the CSP/KSP by means of calls to CardDeleteFile() for the files that contain the affected certificates. CardDeleteContainer removes key material that is not reachable via the file system. Note also that update of the ContainerMapFile is solely the responsibility of the CSP/KSP, which it does via the file system calls.

Status should indicate success if the container existed and was successfully deleted. Otherwise, status should indicate that the container didn’t exist, failed for a specific reason, and so on.

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

### CardGetContainerInfo

Description:

CardGetContainerInfo queries the specified key container for additional 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,

IN OUT PCONTAINER\_INFO pContainerInfo

);

Input:

pCardData Context information for the call. See CardAcquireContext().

bContainerIndex CSP/KSP-assigned index for the container.

dwFlags Reserved—must be zero.

pContainerInfo Pointer to a CONTAINER\_INFO structure supplied by the caller to be filled by the card minidriver.

Output:

Return value Zero on success; otherwise, nonzero.

pContainerInfo Information, which may include public key material.

Comments:

CardGetContainerInfo allocates memory that must be freed by the caller using PFN\_CSP\_FREE.

The container information is returned in the following structure:

#define CONTAINER\_INFO\_CURRENT\_VERSION 1

typedef struct \_CONTAINER\_INFO

{

DWORD dwVersion;

DWORD dwReserved;

DWORD cbSigPublicKey;

PBYTE pbSigPublicKey;

DWORD cbKeyExPublicKey;

PBYTE pbKeyExPublicKey;

} CONTAINER\_INFO, \*PCONTAINER\_INFO;

In the \_CONTAINER\_INFO structure if cbSigPublicKey and pbSigPublicKey fields are not set, it implies that the Signature key is not present. The same is true for the fields corresponding to the Encryption (Key Exchange) key.

Version must be set by the caller.

If CardGetContainerInfo is called with an invalid or nonexisting 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 referred to by a container index. 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,

IN OUT PCARD\_RSA\_DECRYPT\_INFO pInfo

);

Input:

pCardData Context information for the call. See CardAcquireContext().

pInfo Structure containing data to be decrypted, allocated by the Base CSP/CNG KSP.

Output:

Return value Zero on success; otherwise, nonzero.

pInfo Returned buffer containing data processed by the card.

Comments:

Data to be processed by the card is passed in and received back in the following structure (see cardmod.h):

typedef struct \_CARD\_RSA\_DECRYPT\_INFO

{

DWORD dwVersion; // IN

BYTE bContainerIndex; // IN

// See CardCreateContainer to see available values for dwKeySpec

DWORD dwKeySpec; // IN

// 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; // IN | OUT

DWORD cbData; // IN | OUT

}

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

#define CARD\_RSA\_DECRYPT\_INFO\_CURRENT\_VERSION 1

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

Version should be set by the caller.

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

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

### CardConstructDHAgreement

Description:

CardConstructDHAgreement performs a secret agreement calculation for Diffie Hellman 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 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 indexing 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 via an opaque BYTE that the card minidriver maintains. This BYTE should be concerned with a handle to a card-side agreement:

DWORD WINAPI CardConstructDHAgreement(

IN PCARD\_DATA pCardData,

IN OUT PCARD\_DH\_AGREEMENT\_INFO pSecretInfo);

Input:

pCardData Context information for the call. See CardAcquireContext().

pSecretInfo Information needing necessary context for calculating the secret agreement. This structure is also used to return the results.

Output:

Return value Zero on success; otherwise, nonzero.

pSecretInfo bSecretAgreementIndex updated within the passed-in structure pointed to by pSecretInfo.

Comments:

Like CardRSADecrypt, the information is passed to this routine via a structure:

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

typedef struct \_CARD\_DH\_AGREEMENT\_INFO

{

IN DWORD dwVersion;

IN BYTE bContainerIndex;

IN DWORD dwFlags;

IN DWORD cbPublicKey;

IN PBYTE pbPublicKey;

IN PBYTE pbReserved; //ignored

IN DWORD cbReserved; //ignored

// Version 2

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 FIPS 140-2 certified. If dwVersion is passed is 1, the card should return the following error:

ERROR\_REVISION\_MISMATCH

One can support as many agreements in parallel. If there is no space to store an agreement, the card should return the following error:

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 the FIPS 140-2 compliance for smart cards. It holds the requested KDF and the associated input. The KDFs are defined in the CNG documentation. For RSA-only card minidrivers, this entry point is not defined and is set to NULL in the CARD\_DATA structure returned from CardAcquireContext.

The structure is defined as follows:

#define CARD\_DERIVE\_KEY\_VERSION 1

typedef struct \_CARD\_DERIVE\_KEY

{

IN DWORD dwVersion;

IN DWORD dwFlags;

IN LPWSTR pwszKDF;

IN BYTE bSecretAgreementIndex;

// IN Optional

PVOID \*pParameterList;

// OUT PARAMETERS

OUT PBYTE pbDerivedKey;

OUT DWORD cbDerivedKey;

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

typedef DWORD (WINAPI \*PFN\_CARD\_DERIVE\_KEY)(

\_\_in PCARD\_DATA pCardData,

\_\_in\_out PCARD\_DERIVE\_KEY pAgreementInfo);

DWORD WINAPI CardDeriveKey(

\_\_in PCARD\_DATA pCardData,

\_\_in\_out PCARD\_DERIVE\_KEY pAgreementInfo);

NCcryptBufferDesc is defined in ncrypt.h:

typedef struct \_NcryptBuffer {

ULONG cbBuffer; // Length of buffer, in bytes

ULONG BufferType; // Buffer type

PVOID pvBuffer; // Pointer to buffer

} NcryptBuffer, \* PNCryptBuffer;

typedef struct \_NcryptBufferDesc {

ULONG ulVersion; // Version number

ULONG cBuffers; // Number of buffers

PNCryptBuffer pBuffers; // Pointer to array of buffers

} NcryptBufferDesc, \* PNCryptBufferDesc;

Input: (as supplied by KSP)

dwVersion Represents the revision of the CardDeriveKey functionality. The current version is 1. The current version is defined by CARD\_DERIVE\_KEY\_VERSION.

dwFlags Required to be zero or KDF\_USE\_SECRET\_AS\_HMAC\_KEY\_FLAG.

pwszKDF A string that indicates the KDF to be used. This is set to the KDF requested by the client. 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

For detailed information, refer to the CNG documentation. If a card minidriver does not implement the requested KDF, SCARD\_E\_INVALID\_PARAMETER should be returned.

pParameterListContains 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 selected by the pwszKDF parameter. For information about acceptable parameters for a KDF, see the CNG documentation. If a card minidriver does not recognize one of the parameters or that parameter is invalid for the KDF specified, SCARD\_E\_INVALID\_PARAMETER should be returned.

Output:

The following parameters must be set on a successful call.

pbDerivedKey The result of the requested KDF allocated via the pfnCspAlloc callback and deleted appropriately by the KSP. The output format is the binary data that is the result of the KDF.

cbDerivedKey Indicates how large pbDerivedKey is.

### CardDestroyDHAgreement

Description:

CardDestroyDHAgreement 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 returned from CardAcquireContext. The function signature is as follows:

typedef DWORD (WINAPI \*PFN\_CARD\_DESTROY\_DH\_AGREEMENT)(

IN PCARD\_DATA pCardData,

IN BYTE bSecretAgreementIndex,

IN DWORD dwFlags);

DWORD WINAPI CardDestroyDHAgreement(

IN PCARD\_DATA pCardData,

IN BYTE bSecretAgreementIndex,

IN DWORD dwFlags);

Input:

pCardData Context information for the call. See CardAcquireContext().

bSecretAgreementIndex The index of the agreement to destroy.

dwFlags Reserved (must be 0).

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:

CardSignData 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. See CardAcquireContext().

pInfo Structure containing data to be signed, allocated by the Base CSP/CNG KSP.

Output:

Return value Zero on success; otherwise, nonzero.

Comments:

Data to be processed by the card is passed in and received back in the following structure (see 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\_ALL CARDS\_VERSION is provided for those

// applications that do not intend to support passing in the

// pPaddingInfo structure

#define CARD\_SIGNING\_INFO\_ALL CARDS\_VERSION1

//

// Function: CardSignData

//

// Purpose: Sign input data using a specified key

//

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

typedef struct \_CARD\_SIGNING\_INFO

{

IN DWORD dwVersion;

IN BYTE bContainerIndex;

// See dwKeySpec constants

IN DWORD dwKeySpec;

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

// minidriver should return only the size of the resulting

// key in cbSignedData

IN DWORD dwSigningFlags;

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

// to use when padding the data using PKCS

IN 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.

IN PBYTE pbData;

IN DWORD cbData;

OUT PBYTE pbSignedData;

OUT 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

IN LPVOID pPaddingInfo;

IN DWORD dwPaddingType;

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

typedef DWORD (WINAPI \*PFN\_CARD\_SIGN\_DATA)(

\_\_in PCARD\_DATA pCardData,

\_\_in PCARD\_SIGNING\_INFO pInfo);

dwSigningFlags takes the same flag values as CryptSignHash, for example, CRYPT\_NOHASHOID.

aiHashAlg takes those values permissible by ALG\_ID from the HASH algorithm class (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, and CALG\_HMAC).

Algorithms unsupported by the card should be 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 CardCreateContainer), 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 CspPadData under normal operating conditions.

We recommend supporting the CARD\_BUFFER\_SIZE\_ONLY flag, but this is not mandatory. If supported, it helps to 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\_ALL CARDS\_VERSION and CARD\_SIGNING\_INFO\_CURRENT\_VERSION versions.

The input data to be signed is passed in in little-endian format.

### CardQueryKeySizes

Description:

Returns the public key sizes supported by the card in use:

DWORD WINAPI CardQueryKeySizes(

IN PCARD\_DATA pCardData,

IN DWORD dwKeySpec,

IN DWORD dwFlags,

OUT PCARD\_KEY\_SIZES pKeySizes

);

#define CARD\_KEY\_SIZES\_CURRENT\_VERSION 1

Input:

pCardData Context information for the call. See CardAcquireContext().

dwKeySpec Type of key of interest: For allowed values see CardCreateContainer().

dwFlags Reserved—must be zero.

pKeySizes Pointer to CARD\_KEY\_SIZES structure.

Output:

Return value Zero on success; otherwise, nonzero.

pKeySizes Supported key sizes for the specified algorithm type.

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, then the function should return SCARD\_E\_INVALID\_PARAMETER.

If dwKeySpec is defined but not supported by the card, then the function should return SCARD\_E\_UNSUPPORTED\_FEATURE.

# File System Requirements

The “logical” layout is the data layout as presented to the 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 8 ANSI characters (8 bit), excluding characters that are not allowed by Windows file and directory naming convention. The directory structure consists of two levels: the root directory and directories that applications use. Directory names are composed of up to 8 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 being “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) 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 the application using 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)A(RW).

Contents

The file is organized as a series of records containing 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 containing 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 CSP/KSP can cache card data in various ways. The cache file is used to control operation of the caching subsystem within the 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 read from the card allows the 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 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 use by the Microsoft Smart Card Base CSP/CNG KSP. 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 when it is found that the cache data copy that is internal to the application indicates a different version number for data of interest than the file read from the card. The cache is generally checked at the beginning of every transaction with the card.

The array of application cache data DWORDs, one per 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 is owned by the “mscp” application. It consists of a number of records of CONTAINERMAPRECORD type. These records associate a container GUID 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 associated with that container. Thus, the second record in such a file would refer to zero-based index 1. The certificate 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 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) A(RW).

Contents

The file is organized as a series of fixed length records. For a description of the record format, see “Remarks.”

Remarks

This file is created and its content maintained by the Base CSP/CNG 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 associated with that container. All WORD members are little Endian 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;

**GuidInfo** consists of a UNICODE character string representation of an identifier assigned to the container by CAPI. This is usually, but not always, a GUID string.

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 CSP/KSP for a typical implementation. The “Logical Name” is the string used by the CSP/KSP in communication 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 CSP/KSP into subdirectories according to their purpose, using only an index for the actual file name. Any certificates or keys 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(R) A(RW) | Directory index by application name (see section ) |
| **mscp** |  | **Dir** | E(R) U(RW) A(RW) | Microsoft CSP/KSP App Dir |
| 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 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 (which may or may not be unique to the card or user) to perform administrative tasks that the user cannot perform without use of 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 the purposes of file system access, the administrator is generally regarded as a “super-user” and has all of 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 using 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(). |

**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 conducted by the card on the behalf of the requestor, without being able to receive the data so used or feasibly derive it. | X |

### File Access Conditions

Access conditions are similar to access control lists. 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 ACL on the card but it is not one of the defined |  |
| UserReadWriteAc | Everyone No Access  // User Read Write  //  // Example: A password wallet file | E(), U(RW), A(RW) |
| AdminReadWriteAc | Everyone/User No Access  // Admin Read Write  //  // Example: Administration data. | E(), U(), 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 requiring 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. |

### Some Clarifying 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 Card Minidriver interface to Create Directories with E(W) permissions.
* It is not possible through the Card Minidriver interface to change file or directory permissions without deleting and re-creating the file or directory.
* It is not possible through the Card Minidriver interface to create a private key file owned by either the administrator or by a nonauthenticated user.
* It is not possible through the Card Minidriver interface to create a PIN file on the card (E(X), U(W), A(W)).
* It is not possible through the Card Minidriver interface to query directory access conditions.
* It is only possible through the Card Minidriver interface to create files with a subset of the access condition combinations 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/CNG KSP, is a card with the following characteristics:

* 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 discussed under “Card Creation” below.
* 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 Microsoft Smart Card Base CSP/CNG KSP. This requires a card ID file and certain files required by the Base CSP/CNG KSP 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 user having Write permissions. Initial contents are an 8‑byte record consisting of the string “mscp” followed by 4 zero bytes.

4. Create the Microsoft Smart Card Base CSP/CNG KSP application by a call to CardCreateDirectory, referring to application “mscp”, with everyone having Read and 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 of these cards shall reserve the Microsoft “mscp” application, whether or not it is actually used. This explains the unusual facts that the “mscp” application is always created and that there is a file created within the “mscp” application. As card creation is expected to be implemented by functions within the card management DLL supplied by Microsoft, 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

Section 7.6 has information on the challenge/response mechanism.

For an administrator to successfully use this mechanism to unblock a user’s card, it is necessary that administrators can identify and use the administrator key that is stored on the card so that they can correctly generate the response data to the challenge 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 the user 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 issued to the user.

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.

## Enhanced PIN Support

Version 6.0 supports a flexible architecture for multiple PIN support. This architecture introduced a new concept of roles where 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 through 7. We also introduce 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 it can easily verify access control rules on keys and PINs by doing a bit-wise AND operation. The design allows having multiple identities authenticated on the card simultaneously, and this is a requirement for cards supporting v6 card minidrivers. As an example, if PIN #1 is authenticated and then subsequently PIN #2 is authenticated, operations controlled by *any* these PINs should be allowed.

## Session PINs and Secure PIN Channel

When Windows is required to 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:

* Application A, a trusted system process, acquires a handle to the smart card and collects a PIN.
* Application A then calls the card CardAuthenticateEx minidriver function, passing the PIN that was collected and sets the CARD\_AUTHENTICATE\_GENERATE\_SESSION\_PIN flag. This does *not* cause the card to be unlocked.
* Application A stores the session PIN generated and releases the handle to the card and card minidriver. The card is not cold reset.
* Application A sends the session PIN and the name of the reader that has the card acquired in step 1 to Application B
* Application B acquires the same card as in 1.
* Application B calls CardAuthenticateEx, passing in the session PIN and setting the CARD\_AUTHENTICATE\_SESSION\_PIN flag. If the session PIN is still valid, the card should be authenticated and valid for use.
* When Application B is done using the card, it calls CardDeauthenticateEx to de‑authorize the card.

This has several practical limitations to be aware of:

* 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 think to be valid session PINs at any one time. If only one session PIN is possible per PIN, the following implementation is advised.
* The card should remember the most recent session PIN generated.
* 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, this should result in the session PIN being 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 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 BaseCSP/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 function (see section 4.5.4). A read-only card only needs to support a subset of the Version 6 card minidriver interface and is not required to support an administrator PIN. The following table lists the functions that must be supported by a read-only card.

| **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 |  |
| 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 |  |

| **Legend** | |
| --- | --- |
| Yes | 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, then the entry point must return SCARD\_E\_UNSUPPORTED\_FEATURE. |
| Yes (Optional) | These functions should be implemented according to their definition in this specification, regardless of whether the card is read-only. |

The following considerations should also be taken into account when developing a minidriver for a read-only card:

* All expected BaseCSP/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 case, then it is expected that CardGetChallenge, CardAuthenticateChallenge, and CardUnblockPin are not supported by the minidriver.
* 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 BaseCSP/KSP supports three different modes of caching depending on the cache mode 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 (see section , “”), the BaseCSP/KSP data cache is a global cache. If the card is read-only, the BaseCSP/KSP does not write to the cardcf file. If the card can be written to the BaseCSP/KSP, it will operate as today.
* When the returned flag is CP\_CACHE\_MODE\_SESSION\_ONLY, the BaseCSP/KSP operates such that the data cache is cleared when it detects that the card has been removed/reinserted. In other words, we have defined a session to be the span between card insertion and removal. The cache is also implemented per 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 BaseCSP/KSP), the application should use the no-cache mode described later to avoid the situation in which the cache could contain stale data.
* When the flag is CP\_CACHE\_MODE\_NO\_CACHE, the BaseCSP/KSP does not implement any data caching. This mode is designed for card minidrivers that do not support writing of 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 64-bit (8-byte) challenge. The authenticating entity calculates the response by encrypting the challenge using Triple DES (3DES) operating in ECB mode with a 168-bit key (ignoring the parity bits). The card verifies the response by either:

* 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 BaseCSP.) Card minidriver developers are not required to write any special code in the card minidriver to handle this file. 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 operation:

// 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 Crypto Service Provider

Group Policy Settings for 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, then key generated on a host can be imported into the card. This is used for cards that don’t 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  Allow importing of signature keys, that is, key archival scenarios. |
| AllowPrivateExchangeKeyImport | dword:00000000  Allow import of exchange keys, that is, key archival scenarios. |

## Group Policy Settings for Microsoft CNG Smart Card Key Storage Provider

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, then a key generated on a host can be imported into the card. This is used for cards that don’t 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  Allow importing of signature keys, that is, key archival scenarios. |
| AllowPrivateExchangeKeyImport | dword:00000000  Allow import of exchange keys, that is, key archival scenarios. |
| AllocPrivateECDHEKeyImport | Dword:00000000  Allow import of ECDH keys, that is, key archival scenarios |
| AllowPrivateECDSAKeyImport | Dword:00000000  Allow import of 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 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.