Interoperability Specification for ICCs and Personal Computer Systems

Part 6. ICC Service Provider Interface Definition

Apple Computer, Inc.

Axalto

Gemplus SA

Infineon Technologies AG

Ingenico SA

Microsoft Corporation

Philips Semiconductors

Toshiba Corporation

Revision 2.01.01

September 2005

Copyright © 1996–2005 Apple, Axalto, Gemplus, Hewlett-Packard, IBM, Infineon, Ingenico, Microsoft, Philips, Siemens, Sun Microsystems, Toshiba and VeriFone.
All rights reserved.

INTELLECTUAL PROPERTY DISCLAIMER
THIS SPECIFICATION IS PROVIDED “AS IS” WITH NO WARRANTIES WHATSOEVER INCLUDING ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION, OR SAMPLE.
NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED OR INTENDED HEREBY.
AXALTO, BULL CP8, GEMPLUS, HEWLETT-PACKARD, IBM, MICROSOFT, SIEMENS NIXDORF, SUN MICROSYSTEMS, TOSHIBA AND VERIFONE DISCLAIM ALL LIABILITY, INCLUDING LIABILITY FOR INFRINGEMENT OF PROPRIETARY RIGHTS, RELATING TO IMPLEMENTATION OF INFORMATION IN THIS SPECIFICATION. AXALTO, BULL CP8, GEMPLUS, HEWLETT-PACKARD, IBM, MICROSOFT, SIEMENS NIXDORF, SUN MICROSYSTEMS, TOSHIBA AND VERIFONE DO NOT WARRANT OR REPRESENT THAT SUCH IMPLEMENTATION(S) WILL NOT INFRINGE SUCH RIGHTS.

Windows, Windows NT, Windows 2000 and Windows XP are trademarks and Microsoft and Win32 are registered trademarks of Microsoft Corporation.
PS/2 is a registered trademark of IBM Corp. JAVA is a registered trademark of Sun Microsystems, Inc. All other product names are trademarks, registered trademarks, or servicemarks of their respective owners.

Revision History

Revision	Issue Date	Comments
2.00.4	May 28, 2004	Spec 2.0 Final Draft
2.01.00	June 23, 2005	Final Release
2.01.01	September 29, 2005	Changed Schlumberger to Axalto

Contents

1 SYSTEM ARCHITECTURE 1

2 THEORY OF OPERATION 2

2.1 Functional Overview 2

2.2 Implementation Considerations 2

2.3 Installation Considerations 3

2.4 The ICC Service Provider 3

2.5 Cryptographic Service Provider 4

2.6 User Interface Elements 4

2.7 Run time Considerations 5

3 FUNCTIONAL DEFINITION 6
3.1.1 Syntax 6
3.1.2 Data types 6
3.1.3 Calling Conventions 6
3.1.4 Data structures 7
3.1.4.1 Tagged Length Value 7
3.1.4.2 File Specification 7
3.1.4.3 File Path 8
3.1.5 Defined constants 8
3.1.6 Error codes 12

3.2 Required and Optional Interfaces 14

3.3 Required Interfaces 15
3.3.1 Class SCARD 15
3.3.1.1 Properties 15
3.3.1.2 Methods 15

3.4 Optional Interfaces 17
3.4.1 Class FILEACCESS 17
3.4.1.1 Properties 18
3.4.2 Class CHVERIFICATION 24
3.4.2.1 Properties 25
3.4.3 Class CARDAUTH 26
3.4.3.1 Properties 26
3.4.4 Class CRYPTPROV 28
3.4.4.1 Properties 28
3.4.4.2 Methods 28

© 1996–2005 PC/SC Workgroup. All rights reserved.

3.4.5 Class CRYPTHASH 33
3.4.5.1 Properties 33
3.4.5.2 Methods 33
3.4.6 Class CRYPTKEY 36
3.4.6.1 Properties 36
3.4.6.2 Methods 37

4 APPENDIX A - GUID ASSIGNMENTS 43

© 1996–2005 PC/SC Workgroup. All rights reserved.

1 System Architecture

The general architecture of the Interoperability Specification is described in detail in Part 1 of this document and is summarized following. This part deals with one specific element of this architecture, the Service Provider (SP), indicated by the shaded area of the figure. The SP is further subdivided into an ICC Service Provider (ICCSP) and Cryptographic Service Provider (CSP) component. The CSP exposes cryptographic services provided by the ICC that are accessible to external applications, while the ICCSP exposes non-cryptographic functionality. This distinction is necessary because of existing import/export restrictions on cryptographic functionality.

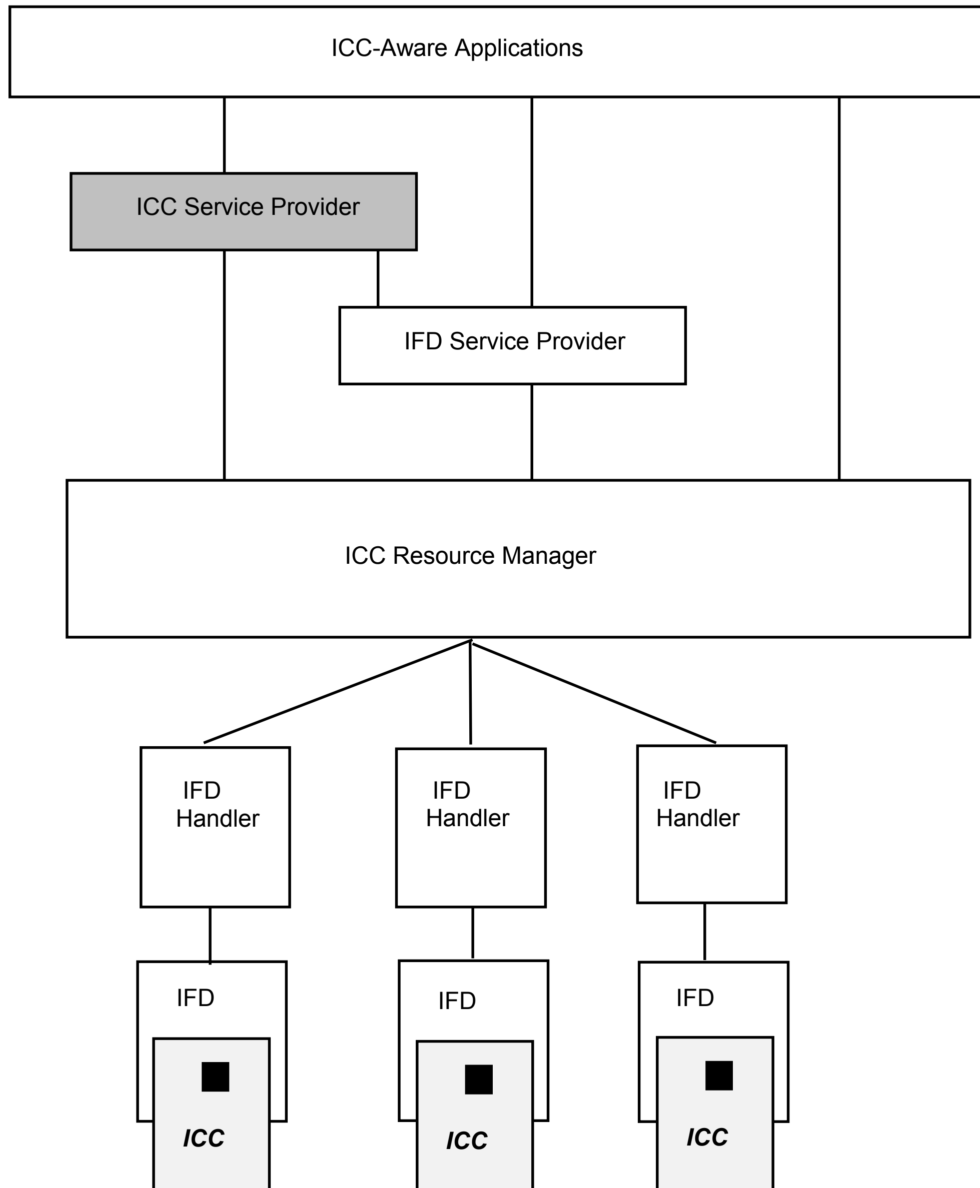

Figure 1-1: PC/SC Architecture

© 1996–2005 PC/SC Workgroup. All rights reserved.

2 Theory of Operation

2.1 Functional Overview

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

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

- File services
- Authentication Services
- Cryptographic Services

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

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

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

- The ICC supplier who wants to enable ICC use within the PC environment. Providing an SP makes accessing the ICC an application software development effort that can be pursued by application developers with no specific expertise in ICC technology (either ICCs or IFDs).

- The ICC issuer, who might layer a "personalized" SP on top of the SP provided by its ICC supplier.

- An ICC-Aware Application supplier who wishes to define the level of functionality required of an ICC to adequately support the application. In defining the API, the application supplier enables one or more ICC suppliers to provide an ICC and the SP that implements the API defined by the application supplier.

- One or more parties interested in a specific domain, who wish to enable the development of both applications and ICCs to support those applications within a domain of interest.

2.2 Implementation Considerations

The role of the SP is to abstract implementation details at the ICC level and expose them in a standard way that application software can easily access. In particular, these details eliminate the need for the application developer to have an intimate knowledge of

© 1996–2005 PC/SC Workgroup. All rights reserved.

specific command mappings and parameter encodings associated with the ISO T=0 or T=1 protocol. The interfaces exposed by the SP should be developed within the context of a given platform. That is, the interfaces should be consistent with API standards within that platform. It is possible to implement interfaces defined herein in a manner suitable for use with procedural programming languages (C, Pascal, and so on) as well as object-oriented languages (C++, Java, VB, and so on). This may entail some modification of the naming conventions, parameter types, an so on. used here. As long as the implementation is functionally equivalent, it is consistent with the intent of this specification.

SPs are built upon the services exposed by the ICC Resource Manager. This provides SP developers with a known set of low-level interfaces they can use to simplify development and maintenance of their software. In addition, developers can use the device sharing and transaction primitives exposed by the ICC Resource Manager to insure correct operation of the ICC. SPs may also make use of existing SPs, if desired, to further simplify development and maximize code reuse.

Another important point is that SPs can be implemented in several different ways and still comply with the intent of this specification. Typically, SPs will be implemented as shared libraries on the end-user system. However, this is not the only option. To meet specific application requirements, it may be best to implement SPs as a client-server sub-system. This could be done, for example, to incorporate application-specific secure messaging operations within an application server's security perimeter, while still conforming to this architecture.

2.3 Installation Considerations

Before an SP can be used within this architecture, it must be "introduced" to the ICC Resource Manager. Typically, this is done through an ICC setup utility that the ICC vendor provides. This utility must provide the following pieces of information related to the card:

- The executable code corresponding to the implementation of the SP.
- Its ATR string and a mask to use as an aid in identifying the ICC.
- An identifier for the SP(s) that support the ICC.
- A list of ICC interfaces that the ICC supports.
- A "friendly name" for the ICC, to identify the ICC to the user (in most cases, the user will supply this name to the setup utility).

2.4 The ICC Service Provider

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

All ICCSPs shall implement the interface for managing connections to an ICC as defined herein (see Section 3). This interface provides mechanisms for connecting and disconnecting to an ICC.

© 1996–2005 PC/SC Workgroup. All rights reserved.

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

The file access interface defines mechanisms for the following tasks:

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

The authentication interface defines mechanisms for the following tasks:

- Cardholder verification
- ICC authentication
- Application authentication to the ICC

2.5 Cryptographic Service Provider

The Cryptographic Service Provider (CSP) is a sub-component of the SP. In contrast to the ICCSP, the CSP isolates cryptographic services because existing regulations imposed by various governments affect import and export. The CSP allows applications to make use of cryptographic services in a manner that compartmentalizes the sensitive elements of cryptographic support into a well-defined and independently installable software package.

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

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

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

See Section 3.4 for a functional definition of this interface.

2.6 User Interface Elements

The ICC and Cryptographic Service Providers are not required to provide any user interface (UI) elements. However, certain recommended UI elements should be

© 1996–2005 PC/SC Workgroup. All rights reserved.

implemented. These should follow published guidelines for the UI within the target environment.

In particular, the SP should implement the UI for managing CHV when supported. The SP is the best place to implement CHV management, because the SP encapsulates knowledge about authentication requirements, password and PIN lengths, administrative commands, and so on. This UI should provide the following mechanisms:

- Cardholder authentication to the ICC
- Password and PIN management
- Administrative access to reset or disable CHV functionality

2.7 Run-time Considerations

The SP is intended to be developed as a user-mode application module that runs on behalf of the current user. The SP will typically exist as a shared library or DLL, though it could be implemented as a system service, depending on guidelines that the operating system vendor establishes.

It is recommended that the SP run with the same privileges as the application(s) making use of it, though this is platform-dependent. Also, SPs should be designed to not consume processing resources unless an active application is using them.

An ICC-Aware Application that uses the services exposed by a specific SP can either directly connect to the SP, or use the services of the ICC Resource Manager to determine which SPs are available to provide the desired services.

The precise mechanisms by which an application can “connect” to an SP are likely to differ depending on the operating environment and programming language being used. As noted in Part 5, the RESOURCEQUERY object provides methods for retrieving a “Provider ID.” This object must encode sufficient information to acquire the context for a specific SP. This information could take the form of a reference to a specific shared library, a COM interface, a URL, and so on. The intent is that based on this knowledge, they can instantiate an SCARD object (see Section 3.3.1) implementation associated with the desired SP.

© 1996–2005 PC/SC Workgroup. All rights reserved.

3 Functional Definition

This section describes the functional interface of the SP. The interface is described in terms of object classes, and methods on object instances, along with required parameters and expected return values. Implementations may alter the naming conventions and parameters as required to adapt to specific environments, but shall conform to the functional interfaces defined herein.

3.1.1 Syntax

The syntax used in describing the SP is based on common procedural language constructs. Data types are described in terms of common C-language due to its widespread use. The following table lists specific conventions and pre-defined values used in this document.

3.1.2 Data types

Type	Characteristic
BYTE	unsigned char, a 8-bit value
USHORT	unsigned short, a 16-bit value
BOOL	short, a 16-bit value
DWORD	unsigned long, a 32-bit value
STR	char array (string)
GUID	unsigned char[16], a 128-bit unique identifier
RESPONSECODE	long, signed 32-bit value
HANDLE	unsigned long, a 32-bit quantity
VOID	unspecified data type whose interpretation is context-specific.
REFTYPE	enumeration type. Authorized values are defined in « Defined constant » section.
FILETYPE	enumeration type. Authorized values are defined in « Defined constant » section.

Arrays of these basic data types are indicated by [ ]. For example BYTE[ ] indicates an array of BYTE values of unspecified length. BYTE[4] indicates an array of BYTE values with four elements.

Data structures are indicated using C-language “struct” type definitions. The following example defines a data structure consisting of a BYTE and DWORD value that is referenced using the SAMPLE_STRUCT identifier.

```
typedef struct {
      BYTE ByteValue
      DWORD DwordValue
} SAMPLE_STRUCT ;
```

3.1.3 Calling Conventions

The interface to the ICCSP is defined in terms of methods associated with one-high level object. Methods are invoked by referencing a named method within the context of an

© 1996–2005 PC/SC Workgroup. All rights reserved.

object instance. How the object is referenced is not specified, because this may vary by implementation. Methods require zero or more parameters and return information using a simple data type and optional output parameters.

For example,

```
RESPONSECODE MethodA(
       IN      DWORD  DwordValue
       IN OUT  BYTE   ByteValue
       OUT     BYTE   OutValue
}
```

defines a method with 3 parameters which returns a RESPONSECODE value. It has two input parameters (DwordValue and ByteValue) and returns additional information in two output parameters (ByteValue and OutValue).

3.1.4 Data structures

3.1.4.1 Tagged Length Value

Tagged Length Value (TLV) encoding is used to encode certain parameter information within this specification and is also used extensively within ISO/IEC 7816 for standardized data encoding. The TLV structure definition is provided following and is compatible with all relevant TLV types including tag values consisting of either 1 or 2 bytes.

Length could be one or two

```
typedef struct {
       DWORD  Tag ;
       DWORD  Length ;
       BYTE[] Value ;
       BOOL   Valid;         // used to differentiate between
                             // TLV with valid, zero-length data,
                             // and an unknown value
} TLV ;
```

3.1.4.2 File Specification

The purpose of this structure is to handle the name associated with an ICC-based file. All ICC files are required to have a 2-byte FileID. DF's may optionally have a name.

A name is a string value less than or equal to MAX_NAME_LEN in size. For ISO/IEC 7816 compliant ICCs, MAX_NAME_LENGTH is equal to 16.

```
typedef struct {
       WORD                 FileID ;
       BYTE                 FileShortID ;
       BYTE[MAX_NAME_LEN]   FileName ;
       BYTE                 FileType ;
}      FILESPEC ;
```

3.1.4.3 File Path

ICC files are referenced by a FILESPEC structure as described in the previous section (3.1.4.2). To uniquely identify an ICC file, however, it may be necessary to refer to it by a fully qualified path specification. This reflects common conventions within the PC environment and is mappable onto common file conventions within the ICC environment.

Within a file path specification, we define the following special symbols :

- / or \ represents the root directory or MF within the ICC
- . represents the current directory
- .. represents the parent directory

A file path specification may be absolute (that is, it explicitly includes the 'root' or MF) or relative. If DF names are supported, the path may optionally use either Names or FileID values for the DFs. If the path includes a final EF, it will always be encoded as a FileID. Flags are provided in the appropriate interfaces to specify whether an application is using Names or FileIDs.

Examples of legal file paths include:

- \File ID1\File ID2..\File IDn - absolute path to the file identified by "File IDn"
- ..\File IDx - relative path to the file identified by "File IDx"
- ..\name1 - relative path to EF identified by "name1"

A legal file path may be up to MAX_PATH_LEN in size. By default, this length will be 256 characters. This may be redefined to meet the needs of specific target environments.

3.1.5 Defined constants

Non-cryptographic defined constants and symbols: The TAG that can be used as follows:

Parameter	Symbol	Comments
REFTYPE		
	SC_TYPE_BY_NAME	An EF or DF is referenced by a name.
	SC_TYPE_BY_ID	An EF or DF is referenced by an ID.
	SC_TYPE_BY_SHORT_ID	An EF or DF is referenced by a short ID.
FILETYPE		
	SC_TYPE_DIRECTORIES	Reference DF only.
	SC_TYPE_FILES	Reference EF only.
	SC_TYPE_ALL_FILES	Reference both EF and DF.
	SC_TYPE_DIRECTORY_FILE	Directory file (DF).
	SC_TYPE_TRANSPARENT_EF	Transparent elementary file.

© 1996–2005 PC/SC Workgroup. All rights reserved.

Parameter	Symbol	Comments
	SC_TYPE_FIXED_EF	Linear fixed elementary file.
	SC_TYPE_CYCLIC_EF	Cyclic fixed elementary file.
	SC_TYPE_VARIABLE_EF	Linear variable elementary file.
	SC_FILETYPES_SUPPORTED	Value is the OR of : SC_ TRANSPARENT_SUPPORTED SC_FIXED_SUPPORTED SC_ CIRCULAR_SUPPORTED SC_VARIABLE_SUPPORTED
	SC_SHORTID_SUPPORTED	
Path specification		
	MAX_PATH_LEN	256
	MAX_NAME_LEN	16
TLV validity		
	SC_TLV_VALID	TLV is valid.
	SC_TLV_NOT_VALID	TLV is not valid.
Flags		
	SC_FL_REPLACE	Replace data with input data.
	SC_FL_ERASE	Erase data.
	SC_FL_OR	Or data with input data.
	SC_FL_AND	And data with input data.
	SC_FL_RECURSIVE	Recursive delete.
	SC_FL_NON_RECURSIVE	Non-recursive delete.
	SC_FL_IHV_GLOBAL	Global ICC holder verification.
	SC_FL_IHV_LOCAL	Local ICC holder verification.
	SC_FL_IHV_ENABLE	Enable ICC holder verification.
	SC_FL_IHV_DISABLE	Disable ICC holder verification.
	SC_FL_IHV_CHANGE	Change ICC holder code (PIN).
	SC_FL_PREALLOCATE	Pre-allocate file during creation process.

© 1996–2005 PC/SC Workgroup. All rights reserved.

Parameter	Symbol	Comments
	SC_FL_GET_ALL_PROPERTIES	Reference to all properties of a file.
	SC_ TRANSPARENT_SUPPORTED	
	SC_FIXED_SUPPORTED	
	SC_ CIRCULAR_SUPPORTED	
	SC_VARIABLE_SUPPORTED	
Seek type		
	SC_SEEK_FROM_BEGINNING	Search forward from the beginning.
	SC_SEEK_FROM_END	Search backward from the end.
	SC_SEEK_RELATIVE	Search in relative mode.

© 1996–2005 PC/SC Workgroup. All rights reserved.

Cryptographic defined constants and symbols:

Parameter	Symbol	Comments
PARAMTYPE		
	HP_ALGID	Hash algorithm.
	HP_HASHSIZE	The hash value size.
	HP_HASHVAL	The hash value.
	KP_ALGID	Key algorithm.
	KP_BLOCKLEN	Block length of cipher or decipher data.
	KP_SALT	The salt value.
	KP_PERMISSIONS	Key permissions.
	KP_IV	The initialization vector.
	KP_PADDING	The padding mode.
	KP_MODE	The cipher mode.
	KP_MODE_BITS	The number of bits to feed back.
	PP_CLIENT_HWND	A window handle is contained in the blob.
	PP_KEYEXCHANGE_KEYSIZE	A new exchange key size is contained in the blob.
	PP_SIGNATURE_KEYSIZE	A new signature key size is contained in the blob.
Cipher mode values		
	CRYPT_MODE_ECB	Electronic codebook.
	CRYPT_MODE_CBC	Cipher block chaining.
	CRYPT_MODE_OFB	Output feedback mode.
	CRYPT_MODE_CFB	Cipher feedback mode.
Permissions values		
	CRYPT_ENCRYPT	Allow encryption.
	CRYPT_DECRYPT	Allow decryption.
	CRYPT_EXPORT	Allow key to be exported.
	CRYPT_READ	Allow parameters to be read.
	CRYPT_WRITE	Allow parameters to be set.
	CRYPT_MAC	Allow MACs to be used with a

© 1996–2005 PC/SC Workgroup. All rights reserved.

Parameter	Symbol	Comments
		key.
KEYSPEC		
	AT_KEYEXCHANGE	Exchange private key.
	AT_SIGNATURE	Signature private key.
FLAGS		
	CRYPT_USERDATA	Prompt for user data.
	CRYPT_EXPORTABLE	Exportable key.
	CRYPT_CREATE_SALT	Assign a salt value.
	CRYPT_USER_PROTECTED	Notification to the user requested.
	CRYPT_UPDATE_KEY	Multiple calls to the same key function.
BLOBTYPE		
	SIMPLEBLOB	Simple type blob.
	PUBLICKEYBLOB	Public holder.
	PRIVATEKEYBLOB	Private key holder.

3.1.6 Error codes

Error, Warning, and Failure codes.

Symbol	Meaning
SCARD_S_SUCCESS	No error was encountered.
SCARD_E_CANCELLED	The action was cancelled by an SCardCancel request.
SCARD_E_CANT_DISPOSE	The system could not dispose of the media in the requested manner.
SCARD_E_INSUFFICIENT_BUFFER	The data buffer to receive returned data is too small for the returned data.
SCARD_E_INVALID_ATR	An ATR obtained from the configuration store is not a valid ATR string.
SCARD_E_INVALID_HANDLE	The supplied handle was invalid.
SCARD_E_INVALID_PARAMETER	One or more of the supplied parameters' could not be properly interpreted.

© 1996–2005 PC/SC Workgroup. All rights reserved.

Symbol	Meaning
SCARD_E_INVALID_TARGET	Configuration startup information is missing or invalid.
SCARD_E_INVALID_VALUE	One or more of the supplied parameters’ values is not valid.
SCARD_E_NO_MEMORY	Not enough memory available to complete this command.
SCARD_E_NO_SMARTCARD	The operation requires an ICC, but no ICC is currently in the device.
SCARD_E_NOT_TRANSACTED	An attempt was made to end a non-existent transaction.
SCARD_E_READER_UNAVAILABLE	The specified IFD is not currently available for use.
SCARD_E_SHARING_VIOLATION	The ICC cannot be accessed because of other connections outstanding.
SCARD_E_SYSTEM_CANCELLED	The action was cancelled by the system, presumably to log off or shut down.
SCARD_E_TIMEOUT	The user-specified timeout value has expired.
SCARD_E_UNKNOWN_CARD	The specified ICC name is not recognized.
SCARD_E_UNKNOWN_READER	The specified IFD name is not recognized.
SCARD_F_COMM_ERROR	An internal communications error has been detected.
SCARD_F_INTERNAL_ERROR	An internal consistency check failed.
SCARD_F_UNKNOWN_ERROR	An internal error has been detected, but the source is unknown.
SCARD_F_WAITED_TOO_LONG	An internal consistency timer has expired. You'll probably have to restart the Resource Manager.
SCARD_W_REMOVED_CARD	The card has been removed, so further communication is not possible. This error may be cleared by the SCardReconnect service.
SCARD_W_RESET_CARD	The card has been reset, so any shared state information is invalid. This error may be cleared by the SCardReconnect service.
SCARD_W_UNPOWERED_CARD	Power has been removed from the card, so further communication is not possible. This error may be cleared by the ScardReconnect service.
SCARD_W_UNRESPONSIVE_CARD	The card is not responding to a reset. This error may be cleared by the SCardReconnect service.
SCARD_W_UNSUPPORTED_CARD	The reader cannot communicate with the card, due to ATR configuration conflicts. This error may be cleared by the SCardReconnect service.
SCARD_W_SECURITY_VIOLATION	Access was denied because of a security violation.
SCARD_W_WRONG_CHV	A Verify failed because the wrong PIN was presented.
SCARD_W_CHV_BLOCKED	A Verify is blocked because the maximum number of PIN submission attempts has been reached.

© 1996–2005 PC/SC Workgroup. All rights reserved.

Symbol	Meaning
SCARD_E_UNEXPECTED	Unexpected ICC error
SCARD_E_ICC_INSTALLATION	No Primary Provider entry of the current ICCSSP found for any ICC.
SCARD_E_ICC_CREATEORDER	Wrong order of creating the Scard objects.
SCARD_E_UNSUPPORTED_FEATURE	Feature not supported by this card.
SCARD_E_DIR_NOT_FOUND	The supplied directory does not exist in the ICC.
SCARD_E_FILE_NOT_FOUND	The supplied file does not exist in the ICC.
SCARD_E_NO_DIR	The supplied path does not represent a directory.
SCARD_E_NO_FILE	The supplied path does not represent a file.
SCARD_E_NO_ACCESS	Access is denied to this file.
SCARD_E_WRITE_TOO_MANY	The data of a write operation exceeds the object's size.
SCARD_E_BAD_SEEK	There was an error trying to set the object pointer.
SCARD_E_INVALID_CHV	A Verify failed because the PIN was invalid.
SCARD_E_UNKNOWN_RES_MNG	The ICC Resource Manager returned an error code not defined at its interface.
SCARD_W_WRONG_CHV	A Verify failed because the wrong PIN was presented.
SCARD_W_EOF	The end of the file has been reached.
SCARD_W_CANCELLED_BY_USER	An action was cancelled by the user.
E_NOTIMPL	Function not implemented.

3.2 Required and Optional Interfaces

The SCARD class is required. The FILEACCESS, CHVERIFICATION, CARDAUTH, CRYPTPROV, CRYPTHASH, and CRYPTKEY classes are optional. It is recommended that objects of the optional classes be created through methods exposed by an object of the SCARD class.

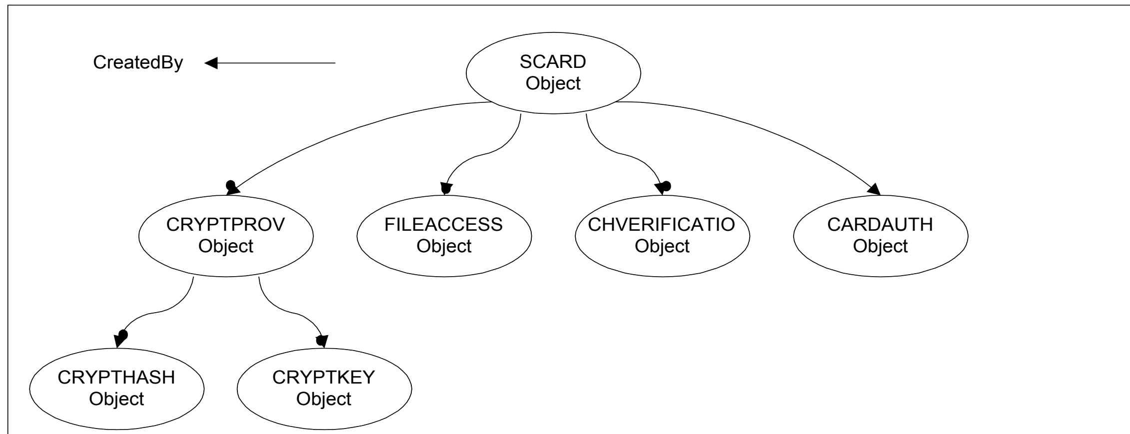

Figure 3-1 Object Creation

© 1996–2005 PC/SC Workgroup. All rights reserved.

3.3 Required Interfaces

The following is a generic description of the standard interfaces that must be supported by compliant ICCs. This description is suitable for implementation in a variety of languages on a variety of systems.

A reference design, created by Microsoft Corporation for the Windows operating system, is provided in Appendix A. This design makes specific assumptions that are consistent with existing Win32 APIs and COM interfaces. For other environments, implementers may use this reference design, or create a functionally equivalent alternative for their specific needs.

3.3.1 Class SCARD

The SCARD class defines methods to support application interfaces to the ICC. It is also responsible for maintaining a well-defined context within which an application may communicate with a supported ICC.

In general, this class is expected to be implemented as part of an ICCSP. An associated CSP may then take advantage of the context information maintained by an application instantiation of the SCARD object to communicate with the desired card. The way this context is passed is system-specific and may be implemented by the developer in any convenient manner. If vendors choose to implement only a CSP, they should implement this class as part of that provider.

3.3.1.1 Properties

HANDLE hContext   // Handle to an ICC communication context. This is equivalent to a context available through the ICC Resource Manager **SCARDCOMM.hCard** property. Set to NULL at object creation.

3.3.1.2 Methods

SCARD()

The SCARD() method creates an instance of the SCARD class and returns a reference to the calling application. The type of this object reference is implementation-specific.

~SCARD()

The ~SCARD() method deletes an instance of the SCARD object. If hContext is valid, this method calls the **Detach()** method before destroying the object.

RESPONSECODE CreateFileAccess(
OUT FileAccess   aFileAccess
)

RESPONSECODE CreateCHVerification(

© 1996–2005 PC/SC Workgroup. All rights reserved.

OUT CHVerification aCHVerification
)

RESPONSECODE CreateCardAuth(
OUT CardAuth aCardAuth
)

RESPONSECODE CreateCryptProv(
OUT CryptProv aCryptProv
)

These four methods either return a SCARD_S_SUCCESS, meaning that the requested object was successfully created or a SCARD_E_UNSUPPORTED_INTERFACE, meaning that the requested interface is not supported by that Service Provider.

RESPONSECODE AttachByHandle(
IN HANDLE hCard
)

The AttachByHandle() method allows an application to pass in a handle to an existing communication context retrieved from the ICC Resource Manager **SCARDCOMM.hCard** property. This allows an application to make use of the ICC Resource Manager to locate an ICC of interest and open a connection before invoking the associated SP.

In using this method, the calling application is effectively passing control of this context to the SP. The application should never pass a valid context to more than one SP and should use the **SCARD::Detach()** method to release the context.

If hCard is an invalid context, this method returns an error.

RESPONSECODE AttachByIFD(
IN STR ReaderName,
IN DWORD Flags // desired access mode

)

The AttachByIFD() method allows an application to create a communication context to an ICC based on the friendly name of a specific reader device. The Flags value can be either SCARD_SHARE_SHARED or SCARD_SHARE_EXCLUSIVE as defined in Part 5 of this specification. It is assumed the ICC SP will set the other connection options to optimize communications with the card.

If the ReaderName isn't known to the ICC Resource Manager, an error will be returned.

RESPONSECODE Detach() ;

© 1996–2005 PC/SC Workgroup. All rights reserved.

The Detach() method releases the communication context associated with the hContext property and resets hContext to NULL.

RESPONSECODE Reconnect(

```
IN DWORD    Flags          // desired access mode
)
```

The Reconnect() method reopens a connection to the card associated with a valid context referenced by the hContext property, which must have been created by calling one of the Attach methods. If hContext is invalid, then an error is returned. The Flags value may be either SCARD_SHARE_SHARED or SCARD_SHARE_EXCLUSIVE.

This method calls the ICC Resource Manager **SCARDCOMM::Reconnect()** method. See Part 5 of this specification for additional information.

RESPONSECODE Lock();

RESPONSECODE Unlock();

These methods provide a mechanism through which an ICC SSP can issue an uninterrupted sequence of accesses to SCARD methods without fear of being interrupted by other applications.

3.4 Optional Interfaces

The following is a generic description of the optional interfaces defined for compliant SPs. If an ICC exposes file access services, authentication services, or cryptographic services, they shall be exposed through these interfaces. If they do not support this functionality, then these interfaces need not be supported.

3.4.1 Class FILEACCESS

The FILEACCESS class implements a high level interface to a card-based file system and should be implemented as part of a compliant ICCSP when access to file-like entities is provided. It is assumed that the underlying card file system is based on the structure defined in ISO/IEC 7816-4. Other implementations are possible, but this is expected to continue to be the most common.

This interface exposes file system entities in a manner very familiar to application developers in the PC environment. It provides mechanisms for locating specific files and performing common operations such as selecting, reading, writing, creating and deleting. It encapsulates and masks much of the low level detail involved in performing these operations at the card level.

In describing this interface, we use the term “file” to denote either an EF or DF within the card. DFs are analogous to directories in a typical PC-based file system, the MF is analogous to a root directory, and EFs are analogous to data files.

© 1996–2005 PC/SC Workgroup. All rights reserved.

Note that three of the methods in the FILEACCESS class are mandatory:

- **FILEACCESS ::Directory()**
- **FILEACCESS ::ChangeDirectory()**
- **FILEACCESS ::GetCurrentDirectory()**

They are mandatory because they are required by the CHVERIFICATION class. It should also be noted that only one instance of this class can be instantiated from each instance of the SCARD object; otherwise, the CHVERIFICATION class won't work correctly.

3.4.1.1 Properties

```
private SCARD scard          // References to an SCARD object.
private STR CurrentPath      // Maintains the path for the selected DF for this
                             // instantiation of the FILEACCESS class.
```

In addition, the following private information should be maintained for each open EF. This information is private to the SP, and no specific implementation requirements are imposed.

- **File Reference.** Mapping between a file handle (hFile) and a specific file on the card
- **File Type.** Type of the file such as transparent, or linear with fixed record length
- **Seek Position.** Current position in the file to use for the next read or write operation
- **Security State.** Security state associated with the file to track whether authentication is required before invoking specific operations

FILEACCESS(
IN SCARD scard
)

This creates an instance of the FILEACCESS class and returns a reference to the calling application. The type of this object reference is implementation-spedific. An object instance will be created only if a valid reference to an SCARD object is supplied.

~FILEACCESS()

This deletes an instance of the FILEACCESS object. The object referenced by the scard property is unaffected.

RESPONSECODE ChangeDir(

```
IN REFTYPE      Ref,        // type of references in PathSpec
IN STR          PathSpec    // relative or absolute path
```

© 1996–2005 PC/SC Workgroup. All rights reserved.

)

Mandatory.

Changes the currently selected ICC DF to that specified by the PathSpec.

The references in the PathSpec are specified in the ref parameter and may be any of the following:

- SC_TYPE_BY_NAME
- SC_TYPE_BY_ID

```
RESPONSECODE GetCurrentDir(
        OUT STR        PathSpec        // absolute path
        )
```

Mandatory.

This method returns the absolute path specification to the currently selected DF as maintained by this instance of the FILEACCESS object. Note that it is possible that this is not the currently selected DF within the ICC.

```
RESPONSECODE Directory (
        IN FILETYPE        Type,        // type of files
        OUT FILESPEC[]     FileList,    // list of FilesSpecs
        OUT WORD           Length       // number of FileList entries
    )
```

Mandatory.

This method returns a list of FileSpecs, of the type specified by the Type parameter, which are the immediate children of the currently selected DF as maintained by this instance of the FILEACCESS object.

At object creation, the current DF is set to the root or the MF of the card.

The type of files to be listed shall be one of the following values:

- SC_TYPE_DIRECTORIES
- SC_TYPE_FILES
- SC_TYPE_ALL_FILES.

```
RESPONSECODE GetProperties(
        IN REFTYPE         Ref,          // type of references
        IN STR             PathSpec,     // relative or an absolute path
        IN OUT TLV[]       Properties,   // list of TLV_TABLE structures
        IN OUT WORD        Length,       // number entries in Properties list
        IN DWORD           Flags
        )
```

© 1996–2005 PC/SC Workgroup. All rights reserved.

This method may be used to retrieve primitive TLV data objects for the file (DF or EF) specified by the PathSpec parameter. The PathSpec parameter can contain file references by name or IDs based on the value of the Ref parameter. Ref can include one of the following values:

- SC_TYPE_BY_NAME
- SC_TYPE_BY_ID
- SC_TYPE_BY_SHORT_ID

The data objects to be retrieved are indicated by a list of TLV structures that contain the tag-values of the objects desired. The tag values need not be set if the Flag includes the value SC_FL_GET_ALL_PROPERTIES. On return, these structures will contain the requested data, if available. Primitive data that is not defined, or cannot be retrieved, is marked SC_TLV_NOT_VALID. Defined TLVs are marked SC_TLV_VALID. Note that a valid TLV may have data of length zero.

RESPONSECODE SetProperties(

```
IN REFTYPE      Ref,          // type of the references
IN STR          PathSpec,     // relative or absolute path
IN TLV[]        Properties,   // list of TLV_TABLE structures
IN WORD         Length,       // number of entries in Properties list
IN DWORD        Flags
)
```

This method can be used to set primitive TLV data objects for the file (DF or EF) specified by the PathSpec parameter. The values to be set are provided in a list of TLV structures. The remaining parameters are identical to those in the **GetProperties()** method.

RESPONSECODE GetFileCapabilities(

```
IN HANDLE       hFile,        // handle to a file
IN OUT TLV[]    Properties,   // list of TLV_TABLE structures
IN OUT WORD     Length,       // number entries in Properties list
IN DWORD        Flags
)
```

The data objects to be retrieved are indicated by a list of TLV structures which contain the tag-values of the objects desired. The tag values need not be set if the Flag includes the value SC_FL_GET_ALL_PROPERTIES. On return, these structures will contain the requested data, if available. Primitive data that are not defined, or cannot be retrieved, are marked SC_TLV_NOT_VALID. Defined TLVs are marked SC_TLV_VALID. Note that a valid TLV may have data of length zero. It should be further noted that this method works with the file referenced by the arameter hFile..

RESPONSECODE Open(

```
IN REFTYPE      Ref,          // type of the references
IN STR          PathSpec,     // relative or absolute path
```

© 1996–2005 PC/SC Workgroup. All rights reserved.

```
        OUT HANDLE          hFile           // handle to a file
)
```

This method opens the specified EF for further use. If successful a non null HANDLE is returned to the caller. This handle is to be used to access the file in subsequent operations. If the PathSpec paremeter references a non-existent file or a DF, an error is returned.

File references are of the following types:

- SC_TYPE_BY_NAME
- SC_TYPE_BY_ID
- SC_TYPE_BY_SHORT_ID

RESPONSECODE Close (

```
        IN HANDLE           hFile           // handle to a file
)
```

This method closes the specified EF. After being closed, any further attempts to use the file handle (hFile) will result in an error.

RESPONSECODE Seek(

```
        IN HANDLE           hFile,          // handle to a file
        IN OUT DWORD        Offset,         // relative offset value
        IN DWORD            SeekType        // starting location
        )
```

This method selects the object from which Read or Write access will be done. A valid hFile file handle must be supplied to this function. The type of data object and the meaning of the Offset parameter depend on the type of file, as described in the table.

Open methods set the current position (for the Seek method) to the beginning of the file.

The following seek modes are allowed:

- **SC_SEEK_FROM_BEGINNING.** Offset is relative to the beginning of the file. The first data object of the file is the reference object for the Offset parameter.
- **SC_SEEK_FROM_END.** Offset is relative to the end of the file. The last data object of the file is the reference object for the Offset parameter.
- **SC_SEEK_RELATIVE.** Offset is relative to the current position. The current reference data object of the file is the reference object for the Offset parameter.

© 1996–2005 PC/SC Workgroup. All rights reserved.

Interpretation of the Offset parameter depends on the type of the file, as indicated in the following table.

File type	Offset	Data
Transparent EF	Byte # from beginning of the file	BYTE[]
Linear or cyclic fixed EF	Record number	Fixed-length data structure
Linear variable EF	Record number	Variable-length data structure

RESPONSECODE Write(

```
        IN HANDLE    hFile,        // handle to a file
        IN DWORD     Length,       // length of the data object to write
        IN BYTE[]    Buffer,       // data object to be written
        IN DWORD     Flags
        )
```

This method writes data, provided in the Buffer parameter, to the specified file. This method can also be used to erase data from a file by specifying the SC_FL_ERASE value in the Flags parameter. A valid hFile file handle must be supplied or an error results.

The data is written into the file at the current file position location (see **Seek()**). The data can be written in one of several modes as encoded in the Flags parameter. These values can be one of the following:

- SC_FL_REPLACE : Replaces the current data with the specified data
- SC_FL_ERASE : Erase the specified field of data (Buffer shall be ignored)
- SC_FL_OR : The specified buffer is ORed with the current data
- SC_FL_AND : The specified buffer is ANDed with the current data

RESPONSECODE Read(

```
        IN HANDLE        hFile,        // handle to a file
        IN OUT DWORD     Length,       // length of the data object
        OUT BYTE[]       Buffer,       // data read
        IN DWORD         Flags
        )
```

This method reads the data from the specified file and returns it in Buffer. Data is read starting at the current file position (see **Seek()**). A valid hFile must be supplied to this function. The type of data object depends upon the type of file as described in the table given for the Seek method.

© 1996–2005 PC/SC Workgroup. All rights reserved.

RESPONSECODE Create(

```
IN REFTYPE       Ref,          // type of reference in  PathSpec
IN STR           PathSpec,     // path specification for new file
IN OUT TLV[]     TLVs,         // list of TLV_TABLE structures,
                               // includes file properties

IN OUT WORD      Length,       // number entries in TLVs list
IN WORD          Flags,
IN VOID          DataBuffer    // data to be pre-allocated
)
```

This method creates a file, of the type specified by the Ref parameter, at the location specified by the PathSpec parametet; i.e. in the currently specified directory.

The following file types are allowed for creation :

- SC_TYPE_DIRECTORY_FILE Creates a DF.
- SC_TYPE_TRANSPARENT_EF Creates a binary EF.
- SC_TYPE_FIXED_EF Creates a record oriented EF with a fixed record size.
- SC_TYPE_CIRCULAR_EF Creates a record oriented, circular, EF.
- SC_TYPE_VARIABLE_EF Creates a record oriented EF with a variable record size.

It is the responsibility of the calling application to know which file types the card supports.

The following flag values are allowed :

- SC_FL_PREALLOCATE - Indicates that EF data space should be pre-allocated within the card.

RESPONSECODE Delete (

```
IN REFTYPE   Ref,
IN STR       PathSpec,     // relative path
IN WORD      Flags
)
```

This method deletes the specified file. The file and its contents are no longer accessible after a successful execution of the command. If there are any open handles to the file, they are invalidated as a result of this command and subsequent attempts to use them will return an error.

© 1996–2005 PC/SC Workgroup. All rights reserved.

The Flags parameter allows the calling application to request a recursive or non-recursive deletion. This is ignored unless the PathSpec parameter references a DF. The following values are allowable:

- SC_FL_RECURSIVE - All children of the specified DF are deleted
- SC_FL_NON_RECURSIVE - The specified DF will be deleted only if it has no children, otherwise an error will be returned.

Attempts to delete a DF that has children will fail unless the “recursive” delete flag is specified.

Deleting a file can affect the CurrentPath property. If the DF associated with the current path is deleted, then the current path will be set to the parent of the deleted DF. Note that attempts to delete the root (corresponding to the MF) will always fail and return and error.

```
RESPONSECODE Invalidate(
        IN REFTYPE  Ref,         // type of references
        IN STR      PathSpec,    // relative path
        IN WORD     Flag
        )
```

This method will mark the file (EF or DF) as invalid. An invalidated file can be selected for use only within a PathSpec parameter, or deleted. All other attempts to access the file will fail. An invalidated file can be restored to full access using the Rehabilitate command.

```
RESPONSECODE Rehabilitate(
        IN REFTYPE  Ref,         // type of references
        IN STR      PathSpec,    // relative path
        IN WORD     Flag
        )
```

This method restores full access to a file (EF or DF), which has been previously invalidated using **Invalidate()**. A Rehabilitate call on an already valid file (EF or DF) returns an error indicating that the file is already valid.

3.4.2 Class CHVERIFICATION

This class is defined for those applications that have detailed knowledge of the ICC’s internal implementation and that implement application-specific CHV policy. This class provides an application with the ability to force a CHV verification or allow the user to change a CHV code. This class should be implemented as part of ICCSPs that expose CHV functionality.

It is expected that applications will allow the SP to determine when CHV is required and prompt the user. Similarly, it is expected that CHV administration will be under user control and will be performed using an SP-implemented user interface.

© 1996–2005 PC/SC Workgroup. All rights reserved.

3.4.2.1 Properties

```
private SCARD scard // Reference to an SCARD object.
```

CHVERIFICATION(

```
IN SCARD      scard
)
```

This creates an instance of the CHVERIFICATION class and returns a reference to the calling application. The type of this object reference is implementation-specific. An object instance will be created only if a valid reference to an SCARD object is supplied.

~CHVERIFICATION()

This deletes an instance of the CHVERIFICATION object. The object referenced by the scard property is unaffected.

RESPONSECODE Verify(

```
IN BYTE[]          Code,  // CHV code, may be NULL
IN DWORD           Flags,
IN OUT DWORD       Ref    // ICC specific reference.
)
```

This method forces a CHV verification against the card. If a code is supplied, then it is used to verify to the card. If Code is NULL, then the SP will prompt the user for a code. Note that if the CHV requirements for the currently selected path are satisfied (by a prior CHV) then the SP may simply return success.

The Flags parameter is used to indicate whether the CHV is to be made against a local (SC_FL_IHV_LOCAL) or global (SC_FL_IHV_GLOBAL) code. The Ref parameter is provided to allow specification of a specific code value within a specified CHV file. Its use is ICC implementation-specific. The value SC_FL_IHV_CHECKONLY is used to indicate to the ICCSP that it should not pop up a dialogue requesting input of CHV information; rather, only the CHV state should be returned.

RESPONSECODE ChangeCode(

```
IN BYTE[]    OldCode,     // old CHV code, may be NULL
IN BYTE[]    NewCode,     // new CHV code, may be NULL
IN DWORD     Flags,
IN DWORD     Ref          // ICC specific reference
)
```

This method forces a CHV code change. This is done by setting the SC_FL_IHV_LOCAL, SC_FL_IHV_ENABLE, or SC_FL_IHV_DISABLE flags. If an application supplys either the current CHV code, a new CHV code, or both, the SP will attempt to do the UI to prompt the user for any data not supplied and request confirmation for the change. If neither the current CHV code or a new CHV code are supplied, the UI will be done by the CSP.

© 1996–2005 PC/SC Workgroup. All rights reserved.

The Flags and Ref parameters of the**Verify()** method are also used.

RESPONSECODE Unblock(

```
IN VOID      Data,        //vendor-specified data to be used for unblocking
IN DWORD     Flags,
IN DWORD     Ref          // ICC specific reference
)
```

This method enables the application to unblock a blocked CHV. This is a security-critical operation and the mechanisms for doing this are vendor defined. The Flags and Ref parameters are as defined for the **Verify()** method.

RESPONSECODE ResetSecurityState(

```
IN DWORD     Flags,
)
```

This method requests that either the global or the local security state of the card be reset, depending on the value of the Flags parameter; SC_FL_IHV_GLOBAL or SC_FL_IHV_LOCAL. Implementation is vendor-specific.

3.4.3 Class CARDAUTH

This class exposes the interfaces to the authentication services that may be supported by an ICC. This class should be implemented by compliant ICCSPs if these services are supported.

3.4.3.1 Properties

```
private SCARD        scard   // Reference to an SCARD object
```

CARDAUTH (

```
IN SCARD scard
)
```

This creates an instance of the CARDAUTH class and returns a reference to the calling application. The type of this object reference is implementation-specific. An object instance will be created only if a valid reference to an SCARD object is supplied.

~CARDAUTH ()

This deletes an instance of the CARDAUTH object. The object referenced by the scard property is unaffected.

RESPONSECODE GetChallenge(

```
IN DWORD           AlgID,        // algorithm identifier that challenge is used
                                 // with.
                                 // may be NULL
```

© 1996–2005 PC/SC Workgroup. All rights reserved.

```
VOID              Param,      // vendor-specific parameters
IN OUT DWORD      Length,     // length of the challenge

OUT BYTE[]        Buffer      // challenge data
)
```

This method returns random challenge data from the ICC to be used in subsequent authentication of an entity external to the card. The AlgID parameter is optional, but if it is supplied it can be used to verify the length of the challenge required or to provide hints for the challenge generation. The Param parameter provides a way to input arbitrary vendor defined parameters and may be NULL. On input, the Length parameter indicates the length of the challenge data request, and on output it indicates the length of the challenge data actually returned in Buffer.

RESPONSECODE ICC_Auth(

```
IN LONG           AlgID,      // algorithm identifier
VOID              Param,      // vendor-specific parameters
IN OUT DWORD      Length,     // length of data in Buffer
IN OUT BYTE[]     Buffer      // challenge data on input, authentication
data on output
)
```

This method provides a means for an application to authenticate the ICC. The supported authentication method(s) are vendor-specific but will typically map onto the ISO/IEC 7816-4 Internal Authenticate command.

The calling application indicates the desired algorithm to use with the AlgID parameter and supplies any vendor-defined data in Param. On input, a challenge of length Length is provided in Buffer. On output, a response is returned in Buffer, and its length is indicated by Length.

RESPONSECODE APP_Auth(

```
IN LONG           AlgID,      // algorithm identifier
VOID              Param,      // vendor-specific parameters
IN BYTE[]         Buffer      // authentication data
)
```

This method provides a means for an application to authenticate to the ICC. The authentication method(s) supported are vendor-specific but will typically map onto the ISO/IEC 7816-4 External Authenticate command.

The calling application indicates the algorithm in use with the AlgID parameter and supplies any vendor-defined data in Param. The Buffer contains the authentication data, typically a cryptogram that includes card-originated data retrieved using the **GetChallenge()** method. The length of the valid authentication data is assumed to be available from the Buffer object.

RESPONSECODE User_Auth(

```
IN LONG           AlgID,      // algorithm identifier
VOID              Param,      // vendor-specific parameters
```

© 1996–2005 PC/SC Workgroup. All rights reserved.

```
IN OUT DWORD      Length,     // length of data in Buffer
IN OUT BYTE[]     Buffer      // challenge data on input, authentication
                              // data on output
)
```

This method provides an interface to user authentication algorithms implemented with an ICC. This is intended to allow a card to be used as part of the authentication process to a remote entity, typically based on shared secret information. Some possible algorithms are described in Part 8 of this specification. The specific algorithms available are vendor-specific.

The calling application indicates the algorithm to use and provides any vendor-defined parameters. Buffer will contain challenge data on input and authentication data on output. The length of the valid data is provided in Length.

3.4.4 Class CRYPTPROV

The CRYPTPROV class exposes the primary methods for accessing cryptographic services and provides mechanisms to create the related objects of the CRYPTKEY and CRYPTHASH class. All CSPs shall implement this class.

3.4.4.1 Properties

(none exposed).

3.4.4.2 Methods

CRYPTPROV (

```
IN SCARD    scard
)
```

This creates an instance of the CRYPTPROV class and returns a reference to the calling application. The type of this object reference is implementation-dependent. An object instance will be created only if a valid reference to an SCARD object is supplied.

~CRYPTPROV ()

This deletes an instance of the CRYPTPROV object. The object referenced by the scard property is unaffected.

RESPONSECODE CreateHash(

```
IN DWORD          AlgId,      // algorithm identifier of the hash algorithm
IN CRYPTKEY       CryptKey,   // key object, if keyed hash is used, else
                              // NULL
IN DWORD          Flags,      // flag values
OUT CRYPTHASH     CryptHash   // CRYPTHASH object reference
)
```

© 1996–2005 PC/SC Workgroup. All rights reserved.

The CreateHash method is used to instantiate a CRYPTHASH object and initialize the hash. The returned object can be used in subsequent calls to HashData and HashSessionKey (both in the CRYPTHASH object) to hash streams of data and session keys.

The computation of the actual hash is done with the HashData and HashSessionKey methods (both in the CRYPTHASH object). After all the data has been added to the object, exactly one of the following operations can be performed:

- The hash value can be retrieved using the GetParam method in the CRYPTHASH object.
- A session key can be derived using DeriveKey.
- The hash can be signed using the SignHash method in the CRYPTHASH object.
- A signature can be verified using VerifySignature in the CRYPTHASH object.

After one of the methods from this list has been called, no other hashing method can be used on this object instance.

The Flags parameter is not defined in this version of the specification and should be NULL.

RESPONSECODE DeriveKey(

```
IN DWORD AlgId,              // algorithm identifier
IN CRYPTHASH CryptHash,      // reference to a CRYPTHASH object
IN DWORD Flags,              // flags specifying the type of key generated
OUT CRYPTKEY CryptKey        // CryptKey object references
)
```

The DeriveKey method generates cryptographic keys derived from base data contained in the CRYPTHASH object. This method guarantees that all keys generated from the same base data will be identical, provided the same CSP and algorithms are used. The base data can be a password or any other user data.

This method is the same as GenKey, except that the generated session keys are derived from base data instead of being random. DeriveKey cannot be used to generate public/private key pairs.

When keys are generated for symmetric block ciphers, the key by default will be set up in cipher block chaining (CBC) mode with an initialization vector of zero. This cipher mode provides a good default method for bulk encryption of data. To change these parameters, use the **SetParam()** method in the CryptKey object.

After the DeriveKey method has been called, no more data can be added to the hash object.

The Flags parameter can be zero, or you can specify one or more of the following flags, using the binary OR operator to combine them:

- CRYPT_EXPORTABLE: If this flag is set, then the session key can be

© 1996–2005 PC/SC Workgroup. All rights reserved.

transferred out of the CSP into a key blob through the Export method of CryptKey object. Because keys generally must be exportable, this flag should usually be set.

If this flag is not set, then the session key will *not* be exportable. This means the key will be available only within the current session and only the application that created it will be able to use it.

This flag does not apply to public/private key pairs.

- CRYPT_CREATE_SALT: Typically, when a session key is made from a hash value, there are a number of leftover bits. For example, if the hash value is 128 bits and the session key is 40 bits, there will be 88 bits left over.

  If this flag is set, then the key will be assigned a salt value based on the unused hash value bits. You can retrieve this salt value using the GetParam method of CryptKey object with the ParamType parameter set to KP_SALT.

  If this flag is not set, then the key will be given a salt value of zero.
  When keys with nonzero salt values are exported (using Export method of CryptKey object), the salt value must also be obtained and kept with the key blob.

- CRYPT_USER_PROTECTED: If this flag is set, then the user will be notified through a dialog box or another method when certain actions are attempted using this key. The precise behavior is specified by the CSP being used.

- CRYPT_UPDATE_KEY: Some CSPs use session keys that are derived from multiple hash values. When this is the case, DeriveKey must be called multiple times.

RESPONSECODE GenKey(

```
IN DWORD        AlgId,      // algorithm identifier
IN DWORD        Flags,      // flags
OUT CRYPTKEY    CryptKey    // CryptKey object reference
)
```

The GenKey method generates random cryptographic keys for use with the CSP module.
The calling application is required to specify the algorithm when calling this method. Because this algorithm type is kept bundled with the key, the application does not need to specify the algorithm later when the actual cryptographic operations are performed.

When keys are generated for symmetric block ciphers, the key by default will be set up in cipher block chaining (CBC) mode with an initialization vector of zero. This cipher mode provides a good default method for bulk encrypting data. To change these parameters, use the SetParam method in the CryptKey object.

In addition to generating keys for symmetric algorithms, the GenKey method can also generate keys for public-key algorithms. The use of public-key algorithms is

© 1996–2005 PC/SC Workgroup. All rights reserved.

restricted to key exchange and digital signatures. To generate one of these key pairs, set the AlgId parameter to one of the following values:

- AT_KEYEXCHANGE Exchange public key.
- AT_SIGNATURE Signature public key.

The Flags parameter can be zero, or you can specify one or more of the following flags, using the binary OR operator to combine them:

- CRYPT_EXPORTABLE : If this flag is set, the key can be transferred out of the CSP into a key blob using the Export method. Because session keys generally must be exportable, this flag should usually be set when they are created.

  If this flag is not set, then the key will *not* be exportable. For a session key, this means that the key will be available only within the current session and only the application that created it will be able to use it. For a public/private key pair, this means that the private key cannot be transported or backed up.

  This flag only applies to session key and private key blobs. It does not apply to public keys, which are always exportable.

- CRYPT_CREATE_SALT: If this flag is set, then the key will be assigned a random salt value automatically. You can retrieve this salt value using the GetParam method in the CryptKey object with the ParamType parameter set to KP_SALT.

  If this flag is not set, then the key will be given a salt value of zero.

  When keys with non zero salt values are exported (through Export), then the salt value must also be obtained and kept with the key blob.

- CRYPT_USER_PROTECTED: If this flag is set, then the user will be notified through a dialog box or another method when certain actions are attempted using this key. The precise behavior is specified by the CSP being used.

RESPONSECODE GenRandom(

```
IN DWORD     Length,      // number of bytes of random data generated
OUT BYTE[]   DataBlob     // random data, of length Length.
)
```

The GenRandom method returns a buffer with random bytes.

The data produced by this method shall be “cryptographically random.” It must meet more stringent requirements than generally provided by typical random number generator such as the ones shipped with common PC industry compilers.

This method is often used to generate random initialization vectors and salt values.

© 1996–2005 PC/SC Workgroup. All rights reserved.

RESPONSECODE GetParam(

```
IN DWORD     ParamType,   // parameter number
IN DWORD     Flags,       // flag values
OUT BYTE[]   DataBlob,    // retrieved parameter
OUT DWORD    Length       // length of DataBlob
)
```

The GetParam method retrieves parameters that govern the operations of a CSP.

The Flags parameter is not defined in this version of the specification and should be NULL.

RESPONSECODE GetUserKey(

```
IN DWORD          KeySpec,     // the specification of the key to retrieve
OUT CRYPTKEY      CryptKey     // the CryptKey object
)
```

The **GetUserKey()** method creates a CryptKey object for user-specific key pairs, such as the user's signature key pair.

The following values are allowed for the KeySpec parameter:

- AT_KEYEXCHANGE   Exchange public key.
- AT_SIGNATURE   Signature public key.

RESPONSECODE ImportKey(

```
IN BYTE[]         DataBlob,       // key blob
IN DWORD          Length,         // length of DataBlob
IN CRYPTKEY       CryptKeyPub,    // key object reference
IN DWORD          Flags,          // flag values
OUT CRYPTKEY      CryptKey        // the CryptKey object
)
```

The ImportKey method is used to transfer a cryptographic key from a key blob to the CSP.

The CryptKeyPub parameter indicates the following:

- If the key blob is not encrypted (for example, a PUBLICKEYBLOB) or if the key blob is encrypted with the key exchange key pair (for example, a SIMPLEBLOB), then this parameter is not used, and must be NULL.
- If a signed key blob is being imported, this key is used to validate the signature of the key blob. In this case, this parameter should contain a reference to the CryptKey object of the process that created the key blob.
- If the key blob is encrypted with a session key (for example, an encrypted PRIVATEKEYBLOB), then this parameter should contain a reference to the CryptKey object for this session key.

© 1996–2005 PC/SC Workgroup. All rights reserved.

The Flags parameter is used only when a public/private key pair is being imported into the CSP (in the form of a PRIVATEKEYBLOB). In this case, if the CRYPT_EXPORTABLE flag is set then subsequent applications will be permitted to export the private key back out of the CSP.

RESPONSECODE SetParam(

```
IN DWORD      ParamType,   // parameter number
IN DWORD      Flags,       // flag values
IN BYTE[]     DataBlob,    // parameter data
IN DWORD      Length       // length of DataBlob
)
```

The SetParam method customizes the operations of a CSP.

The currently defined ParamType values are as follows:

- PP_CLIENT_HWND: Specifies that a window handle is contained in DataBlob.
- PP_KEYEXCHANGE_KEYSIZE: Specifies that a new exchange key size is contained in DataBlob.
- PP_SIGNATURE_KEYSIZE: Specifies that a new signature key size is contained in DataBlob.

The Flags parameter is not defined in this version of the specification and should be NULL.

3.4.5 Class CRYPTHASH

The CRYPTHASH class exposes methods associated with hashing (or message digests) used as part of numerous cryptographic protocols, such as digital signatures. If these services are exposed, they shall be implemented within a CSP.

A CRYPTHASH object should only be instantiated by calling **CRYPTPROV::CreateHash()**.

3.4.5.1 Properties

(none defined)

3.4.5.2 Methods

protected CRYPTHASH ()

This creates an instance of the CRYPTHASH class and returns a reference to the calling application. The type of this object reference is implementation-dependent.

~CRYPTHASH ()

This deletes an instance of the CRYPTHASH object.

© 1996–2005 PC/SC Workgroup. All rights reserved.

RESPONSECODE GetParam(

```
IN DWORD        ParamType,   // The parameter number
IN DWORD        Flags,       // The flag values
OUT BYTE[]      DataBlob,    // The retrieved parameter
OUT DWORD       Length       // length of DataBlob
)
```

The GetParam method retrieves data that governs the operations of a hash object. The actual hash value can also be retrieved using this method.

The ParamType value can be set to one of the following hash parameter types:

- HP_ALGID: The hash algorithm identifier. The returned DataBlob will contain a long value indicating the algorithm that was specified when the hash object was created. See the CreateHash method of the CRYPTPROV class for a list of hash algorithms.
- HP_HASHSIZE: The hash value size. The returned DataBlob will contain a long value indicating the number of bytes in the hash value. This value will usually be 16 or 20, depending on the hash algorithm. Applications should retrieve this parameter just before the HP_HASHVAL parameter so the correct amount of memory can be allocated.
- HP_HASHVAL: The hash value. The returned DataBlob will contain the hash value or message digest. This value is generated based on the data supplied earlier through the HashData and HashSessionKey methods from CrypHash object. After this parameter has been retrieved, this object is marked “finished” and no more data can be added to it.

The Flags parameter is not defined in this version of the specification and should be NULL.

RESPONSECODE HashData(

```
IN BYTE[]       DataBlob,    // data to be hashed
OUT DWORD       Length,      // length of DataBlob
IN DWORD        Flags        // the flag values
)
```

The **HashData()** method is used to compute the cryptographic hash on a stream of data. This method, and **HashSessionKey()**, can be called multiple times to compute the hash on long streams or on discontinuous streams.

The Flags value currently defined is:

- CRYPT_USERDATA: When this flag is set, the CSP should prompt the user to input some data directly. This is then added to the hash. The calling application is not allowed access to the data. For example, this flag can be used to allow the user to enter a PIN into the system.

RESPONSECODE HashSessionKey(

```
IN CRYPTKEY     CryptKey,    // CRYPTKEY object to be hashed
IN DWORD        Flags        // the flag values
)
```

© 1996–2005 PC/SC Workgroup. All rights reserved.

The **HashSessionKey()** method is used to compute the cryptographic hash on a CRYPTKEY object. This method can be called multiple times to compute the hash on multiple keys. Calls to **HashData()** and **HashSessionKey()** may be interspersed.

The Flags parameter is not defined in this version of the specification and should be NULL.

RESPONSECODE SetParam(

```
    IN DWORD        ParamType,   // parameter number
    IN DWORD        Flags,       // flag values
    IN BYTE[]       DataBlob,    // parameter data.
    IN DWORD        Length       // length of DataBlob
)
```

This method is provided to allow customization of the hash object. Currently, only a single parameter is defined for this method:

- HP_HASHVAL: Hash value. The DataBlob contains a byte array containing a hash value to place directly into the hash object. Before setting this parameter, the size of the hash value should be determined by reading the HP_HASHSIZE parameter with the **GetParam()** method.

Normal applications should never set this parameter and compliant implementations may simply return an error if they choose not to support this capability.

The Flags parameter is not defined in this version of the specification and should be NULL.

RESPONSECODE SignHash(

```
    IN DWORD        KeySpec,      // key pair used to sign the hash
    IN STR          Description,  // string describing the data to sign
    IN DWORD        Flags,        // flag values
    OUT DWORD       Length,       // length of Signature Buffer
    OUT BYTE[]      Signature     // signature data
)
```

The SignHash method is used to sign data. Because all signature algorithms are asymmetric, and computationally expensive, you generally hash the data to be signed and then use SignHash to sign the hash value.

The following keys can be specified for the KeySpec parameter:

- AT_KEYEXCHANGE   Exchange private key
- AT_SIGNATURE   Signature private key

The signature algorithm used is implied by the key pair and is set when the keys are created.

© 1996–2005 PC/SC Workgroup. All rights reserved.

The Flags parameter is not defined in this version of the specification and should be NULL.

The description string (generally human-readable text) specified in the Description parameter is added to the hash object before the signature is generated. Whenever the signature is authenticated (with VerifySignature), the exact same description string must be supplied. This ensures that both the signer and the authenticator agree on what is being signed or authenticated.

CSPs may display this description string to the user. This lets the user confirm what they are signing. This gives some protection to the user from ill behaved or malicious applications.

The description parameter can be NULL if no description string is to be included in the signature. Usually, this is the case only when the signature is performed using a signature key that is not legally bound to the user. For example, when a signature operation is performed with the key exchange private key as part of a key exchange protocol, no description string is typically specified.

```
RESPONSECODE VerifySignature(
    OUT DWORD       Length,       // length of Signature Buffer
    IN BYTE[]       Signature,    // signature data to be verified
    IN CRYPTKEY     CryptKeyPub,  // CRYPTKEY object for the verification
                                  // public key
    IN DWORD        Flags         // the flag values
    )
```

The VerifySignature method is used to verify a signature against a hash object. Before calling this method, the HashData and/or HashSessionKey methods are called to add the data and/or session keys to the hash.

After this method has been completed, attempts to add additional data to the current hash will fail.

The Flags parameter is not defined in this version of the specification and should be NULL.

3.4.6 Class CRYPTKEY

The CRYPTKEY class exposes methods for using and managing encryption keys used in the encryption/decryption process. If these services are exposed, they shall be implemented within a CSP.

A CRYPTKEY object should only be instantiated by calling one of the CRYPTPROV key generation methods.

3.4.6.1 Properties

(none exposed).

© 1996–2005 PC/SC Workgroup. All rights reserved.

3.4.6.2 Methods

protected CRYPTKEY ()

This create an instance of the CRYPTKEY class and returns a reference to the calling application. The type of this object reference is implementation-dependent.

~CRYPTKEY()

This deletes an instance of the CRYPTKEY object.

RESPONSECODE Decrypt(

```
IN CRYPTHASH    CryptHash,      // references to a CRYPTHASH object,
                                // may be NULL
IN BOOL         Final,          // true if last block to decrypt
IN DWORD        Flags,          // the flag values
IN BYTE[]       Dencrypted,     // data to be decrypted
 IN DWORD       DecryptLength,  // length of Buffer to be decrypted
OUT BYTE[]      DataBlob        // plain-text after decryption
OUT DWORD       BlobLength,     // length of DataBlob Buffer
)
```

The Decrypt method is used to decrypt data that was previously encrypted via the **Encrypt()** method.

If data is to be decrypted and hashed simultaneously, a CRYPTHASH object can be passed in the CryptHash parameter. The hash value will be updated with the decrypted plain-text. This option is useful when simultaneously decrypting and verifying a signature. Prior to calling **Decrypt()**, the application should obtain a CRYPTHASH object by calling **CRYTPROV::CreatHash()**. After the decryption is complete, the hash value can be:

- Obtained (through the GetParam method in the CRYPTHASH object)
- Signed (through the SignHash method in the CRYPTHASH object)
- Used to verify a digital signature (through the VerifySignature method in the CRYPTHASH object)

When a large amount of data needs to be decrypted, it can be done in sections. This is done by calling Decrypt multiple times. The Final parameter should be set to True only on the last invocation of Decrypt, so the decryption engine can properly finish the decryption process. The following extra actions are performed when Final is True:

- If the key is a block cipher key, the data will be padded to a multiple of the block size of the cipher. To find the block size of a cipher, use GetParam to get the KP_BLOCKLEN parameter of the key.
- If the cipher is operating in a chaining mode, the next Decrypt operation will reset the cipher's feedback register to the KP_IV value of the key.

© 1996–2005 PC/SC Workgroup. All rights reserved.

- If the cipher is a stream cipher, the next Decrypt call will reset the cipher to its initial state.

The Flags parameter is not defined in this version of the specification and should be NULL.

RESPONSECODE Encrypt(

```
IN CRYPTHASH     CryptHash,    // reference to a CRYPTHASH object,
                               // may be NULL
IN BOOL          Final,        // true if last block to encrypt
IN DWORD         Flags,        // flag values
IN BYTE[]        DataBlob,     // data to be encrypted.
IN DWORD         BlobLength,   // length of DataBlob Buffer
OUT DWORD        EncLength,    // length of Encrypted Buffer
OUT BYTE[]       Encrypted     // encrypted data
)
```

The Encrypt method is used to encrypt data. The algorithm used to encrypt the data is the one designated when this CryptKey object was created.

If data is to be hashed and encrypted simultaneously, a CRYPTHASH object can be passed in the CryptHash parameter. The hash value will be updated with the plain-text passed in. This option is useful when generating signed and encrypted text. Prior to calling Encrypt, the application should obtain a CRYPTHASH object by calling the CreateHash method in the CRYPTPROV object. After the encryption is complete, the hash value can be obtained through the GetParam method or the hash can be signed using the SignHash method in the CRYPTHASH object.

When a large amount of data needs to be encrypted, it can be done in sections. This is done by calling Encrypt multiple times. The Final parameter should be set to True only on the last invocation of Encrypt, so the encryption engine can properly finish the encryption process. The following extra actions are performed when Final is True:

- If the key is a block cipher key, the data will be padded to a multiple of the block size of the cipher. To find the block size of a cipher, use GetParam to get the KP_BLOCKLEN parameter of the key.
- If the cipher is operating in a chaining mode, the next Encrypt operation will reset the cipher's feedback register to the KP_IV value of the key.
- If the cipher is a stream cipher, the next Encrypt will reset the cipher to its initial state.

The Flags parameter is not defined in this version of the specification and should be NULL.

RESPONSECODE Export(

```
IN CRYPTKEY      CryptKeyExp,  // references to another CryptKey object
IN DWORD         BlobType,     // key blob type to be exported
IN DWORD         Flags,        // flag values
```

© 1996–2005 PC/SC Workgroup. All rights reserved.

```
OUT BYTE[]      DataBlob,     // exported key blob
OUT DWORD       BlobLength    // length of DataBlob Buffer
)
```

The Export method is used to export cryptographic keys out of a CSP in a secure manner.

The key blob associated with this CryptKey object is returned to the caller. This key blob can be sent over a non secure transport or stored in non-secure storage. The key blob is useless until the intended recipient uses the ImportKey method from CryptKey object on it, which will import the key into the recipient's CSP.

Most often, CryptKeyExp will be the key exchange public key of the destination user. However, certain protocols require that a session key belonging to the destination user be used for this purpose.

If the key blob type specified by BlobType is PUBLICKEYBLOB, then this parameter is unused and should be set to NULL. If the key blob specified by BlobType is PRIVATEKEYBLOB, then this is typically a CryptKey object for a session key that is to be used to encrypt the key blob.

BlobType must currently be one of the following constants:

- SIMPLEBLOB
- PUBLICKEYBLOB
- PRIVATEKEYBLOB

The Flags parameter is not defined in this version of the specification and should be NULL.

RESPONSECODE GetParam(

```
IN DWORD        ParamType,    // parameter number
IN DWORD        Flags,        // flag values
OUT BYTE[]      DataBlob,     // retrieved parameter
OUT DWORD       BlobLength    // length of DataBlob Buffer
)
```

The **GetParam()** method retrieves data that governs the operations of a key. Note that the base keying material is not obtainable by this method or any other method.

For all key types, the ParamType value can be set to one of the following key parameter types:

- KP_ALGID: Key algorithm identifier. The returned BLOB will contain a long value indicating what algorithm was specified when the key was created.

- KP_BLOCKLEN: If a session key was specified for this CryptKey object, this parameter returns the block length, in bits, of the cipher. The returned BLOB contains a long value indicating the block length. For stream ciphers, this

© 1996–2005 PC/SC Workgroup. All rights reserved.

value will always be zero.

If a public/private key pair was specified for this CryptKey object, this parameter returns the key pair's encryption granularity in bits. For example, the Microsoft RSA Base Provider generates 512-bit RSA key pairs, so a value of 512 is returned for these keys. If the public-key algorithm does not support encryption, the value returned by this parameter is undefined.

- KP_SALT: The salt value. The returned BLOB will contain a BYTE array indicating the current salt value. The size of the salt value will vary depending on the CSP and algorithm being used.

  Salt values do not apply to public/private key pairs.

- KP_PERMISSIONS: Key permissions. The returned BLOB will contain a long value with zero or more permission flags set. Refer to the table at the end of this section for a description of each of these flags.

If a block cipher session key was specified for this CryptKey object, the ParamType value can also be set to one of the following parameter types :

- KP_IV: The initialization vector. The returned DataBlob will contain a BYTE array indicating the current initialization vector. This array contains block_length/8 elements. For example, if the block length is 64 bits, the initialization vector will consist of eight bytes.
- KP_PADDING: The padding mode. The returned DataBlob will contain a long value indicating the padding method used by the cipher. Following are the padding modes currently defined:
  - PKCS5_PADDING—PKCS 5 (sec 6.2) padding method.
- KP_MODE: The cipher mode. The returned DataBlob will contain a long value indicating the mode of the cipher. Refer to the following table for a list of valid cipher modes.
- KP_MODE_BITS: The number of bits to feed back. The returned DataBlob will contain a long value indicating the number of bits that are processed per cycle when the OFB or CFB cipher modes are used.

The following table lists the possible cipher mode values that can be returned when ParamType is KP_MODE.

KP_MODE cipher mode values	
CRYPT_MODE_ECB	Electronic codebook
CRYPT_MODE_CBC	Cipher block chaining
CRYPT_MODE_OFB	Output feedback mode
CRYPT_MODE_CFB	Cipher feedback mode

The following table lists the flags in the bit field that is returned when ParamType is KP_PERMISSIONS. Custom CSPs can use these flags to restrict operations on keys.

© 1996–2005 PC/SC Workgroup. All rights reserved.

KP_PERMISSIONS flags	
CRYPT_ENCRYPT	Allow encryption.
CRYPT_DECRYPT	Allow decryption.
CRYPT_EXPORT	Allow key to be exported.
CRYPT_READ	Allow parameters to be read.
CRYPT_WRITE	Allow parameters to be set.
CRYPT_MAC	Allow MACs to be used with key.

The Flags parameter is not defined in this version of the specification and should be NULL.

```
RESPONSECODE SetParam(
    IN DWORD        ParamType,   // parameter number
    IN DWORD        Flags,       // flag values
    IN BYTE[]       DataBlob,    // parameter data
    IN DWORD        BlobLength   // length of DataBlob Buffer
    )
```

The SetParam method customizes various aspects of a key's operations. Generally, this method is used to set session-specific parameters on symmetric keys. Note that the base keying material is not accessible by this method. CSP developers may define parameters that can be set on these keys.

For all session key types, the ParamType value can be set to one of the following key parameter types:

- KP_SALT: The salt value. The DataBlob should contain a BYTE array specifying a new salt value. This value is made part of the session key. The size of the salt value will vary depending on the CSP being used, so before setting this parameter, it should be read using GetParam method of CryptKey object in order to determine its size.

  When it is suspected that the base data used for derived keys is less than ideal, salt values are often used to make the session keys more random. This makes dictionary attacks more difficult.

- KP_PERMISSIONS: The key permissions flags. The DataBlob should contain a long value specifying zero or more permission flags. Refer to the GetParam method of CryptKey object for a description of these flags.

If a block cipher session key is specified for this CryptKey object, the ParamType value can also be set to one of the following parameter types:

- KP_IV: The initialization vector. The DataBlob should contain a BYTE array specifying the initialization vector. This array should contain block_length/8 elements. For example, if the block length is 64 bits, the initialization vector will consist of eight bytes.
- KP_PADDING: The padding mode. The DataBlob should contain a long

© 1996–2005 PC/SC Workgroup. All rights reserved.

value specifying the padding method to be used by the cipher. Following are the padding modes currently defined:

  - PKCS5_PADDING—PKCS 5 (sec 6.2) padding method.

- KP_MODE: The cipher mode. The DataBlob should contain a long value specifying the cipher mode to be used. Refer to the GetParam method of CryptKey object for a list of the defined cipher modes.
- KP_MODE_BITS: The number of bits to feed back. The DataBlob contains a long value indicating the number of bits that are processed per cycle when the OFB or CFB cipher mode is used.

The Flags parameter is not defined in this version of the specification and should be NULL.

© 1996–2005 PC/SC Workgroup. All rights reserved.

4 Appendix A - GUID Assignments

Defined GUIDs

Parameter	Symbol	Comments
ISCardManage	5E586211-5A09-11D0-B84C-00C04FD424B9	This interface used for Management methods (i.e. AttachByHandle, AttachByIFD, etc.)
ISCardFileAccess	4029DD8A-5902-11D0-B84C-00C04FD424B9	File Access methods
ISCardVerify	4029DD85-5902-11D0-B84C-00C04FD424B9	Verification methods
ISCardAuth	7B063D61-6E40-11D0-B858-00C04FD424B9	Authentication methods

© 1996–2005 PC/SC Workgroup. All rights reserved.