15370 Barranca Parkway
Irvine, CA 92618

OMNIKEY®

5326 DFR

SOFTWARE DEVELOPER GUIDE

5326-903, Rev A.0

March 2012

Contents

List of Figures

List of Tables

Copyright

Trademarks

Revision History

Date	Author	Description	Document Version
3/27/2012	Jacqueline Maatuq	Initial Version	A.0

Contacts

North America		Europe, Middle East and Africa	
15370 Barranca Parkway Irvine, CA 92618 USA		Phoenix Road Haverhill, Suffolk CB9 7AE England	
Phone:	800 237 7769	Phone:	+44 1440 714 850
Fax:	949 732 2120	Fax:	+44 1440 714 840

Asia Pacific

19/F 625 King's Road
North Point, Island East
Hong Kong
Phone: 852 3160 9800
Fax: 852 3160 4809

support.hidglobal.com

About this Guide

Purpose

This Developer Guide is for developers integrating contactless storage or CPU cards using the OMNIKEY 5326 DFR.

How you should read this guide

Beginners should read this guide chapter by chapter.

Developers familiar with OMNIKEY 5x2x should read Chapter 3 and 4 for migration purposes.

How this guide is organized

After a brief overview in Chapter 0 and a PC/SC introduction in Chapter 2, you can start building up your first “hello card” program.

Chapter 3 discusses the OMNIKEY 5326 DFR use of the HID SIO processor technology.

Chapter 4 describes migration scenarios.

Finally, Chapter 5 shows how to retrieve reader information.

Overview

Product Description

HID Global’s OMNIKEY 5326 DFR opens new market opportunities for system integrators seeking simple reader integration and development using standard interfaces, such as CCID (Circuit Card Interface Device). This reader works without installing or maintaining device drivers; only an operating system driver, for example, Microsoft CCID driver is necessary.

The OMNIKEY 5326 DFR features include supporting the common low and high frequency card technologies. This includes iCLASS, HID Prox and facilitating the credential migration from low frequency (PROX) to high frequency (iCLASS) cards.

The OMNIKEY 5326 DRF provides a TIP enabled boot loader for secure firmware upgrades. No special driver is necessary for firmware upgrades. It is possible to upgrade the reader with firmware add-ons.

See www.hidglobal.com/omnikey for new firmware versions.

Features

- CCID Support – Removes the requirement to install drivers on standard operating systems to fully support capabilities of the reader board
- Dual Frequency – Allows straightforward migration scenarios by simultaneously supporting Low and High Frequency credentials, including HID PROX and iCLASS®
- Rapid and Easy Integration – No special driver installation is required
- TIP Enabled Boot Loader – Allows a secure firmware upgrade in the field
- SIO Enabled – The integrated SIO processor enables the reader to process PAC bits and all future Secure objects

1.1 Getting Started

1.1.1 Driver installation

As stated previously, no extra driver installation is necessary and every CCID compliant driver should work with the reader. However, Microsoft's CCID driver prevents the execution of CCID Escape commands. If an application uses those commands, apply the following procedure. See also 2.2.

In order to send or receive an Escape command to a reader, add the DWORD registry value EscapeCommandEnable and set to a non-zero value under the HKLM\SYSTEM\CCS\Enum\USB\Vid*Pid**\Device Parameters key.

The VID and PID of the reader are 076B and 5326 so create the DWORD under:

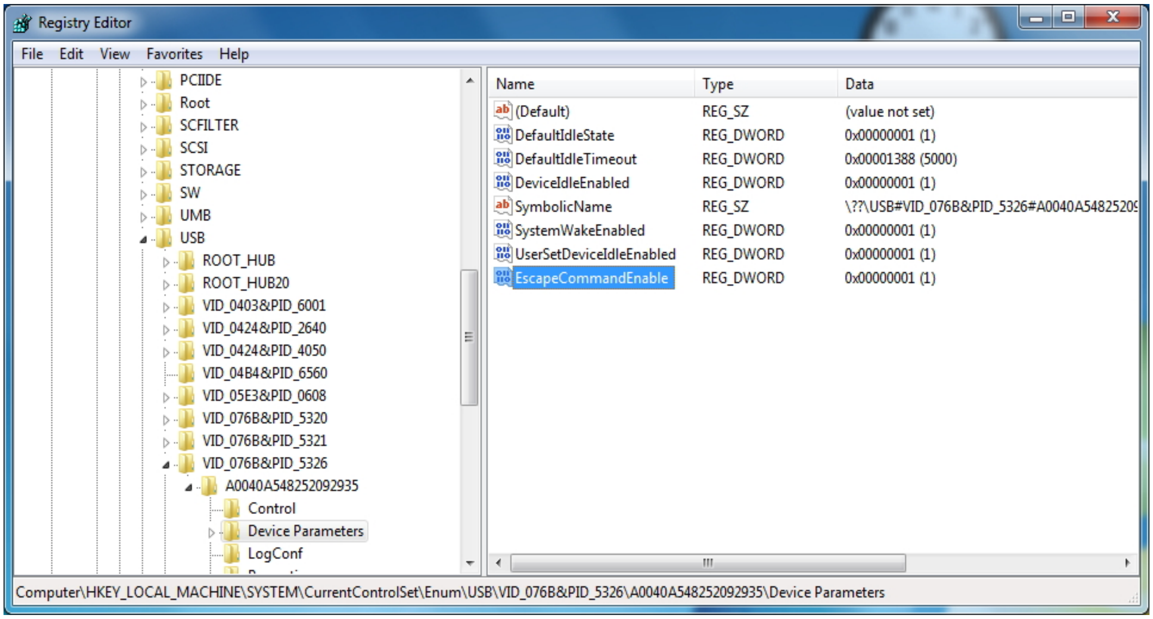
HKLM\SYSTEM\CCS\Enum\USB\VID_076B&PID_5326\xxxxxxxx\Device Parameters

And set the value to “1”.

Then the vendor IOCTL for the Escape command is defined as follows:

#define IOCTL_CCID_ESCAPE SCARD_CTL_CODE(3500).

For details see http://msdn.microsoft.com/en-us/windows/hardware/gg487509.aspx .



Name	Type	Data
(Default)	REG_SZ	(value not set)
DefaultIdleState	REG_DWORD	0x00000001 (1)
DefaultIdleTimeout	REG_DWORD	0x00001388 (5000)
DeviceIdleEnabled	REG_DWORD	0x00000001 (1)
SymbolicName	REG_SZ	\??\USB#VID_076B&PID_5326#A0040A54825209
SystemWakeEnabled	REG_DWORD	0x00000001 (1)
UserSetDeviceIdleEnabled	REG_DWORD	0x00000001 (1)
EscapeCommandEnable	REG_DWORD	0x00000001 (1)

Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USB\VID_076B&PID_5326\A0040A548252092935\Device Parameters

Figure 1 - Registry Editor

2 PC/SC 2.02

2.1 Overview

With the OMNIKEY 5326, access contactless cards through the same framework as ISO7816 contact cards. This makes card integration a snap for any developer who is already familiar with PC/SC. Even valuable PC/SC resource manager functions, such as card tracking, are available for contactless card integration.

The Microsoft® Developer Network (MSDN®) Library contains valuable information and a complete documentation of the SCard API within the MSDN Platform SDK.

See http://msdn.microsoft.com/en-us/library/windows/desktop/aa380149(v=vs.85).aspx

You can directly access contactless CPU cards through the PC/SC driver.

2.2 How to access Contactless Cards or the reader through PC/SC

The following steps provide a guideline to create your first contactless smart card application using industry standard, PC/SC compliant API function calls. The function definitions provided are taken verbatim from the MSDN Library [MSDNLIB]. For additional descriptions of these and other PC/SC functions provided by the Microsoft Windows PC/SC smart card components, refer directly to the MSDN Library.
See http://msdn.microsoft.com/en-us/library/ms953432.aspx.

1. Establish Context

   This step initializes the PC/SC API and allocates all resources necessary for a smart card session. The **SCardEstablishContext** function establishes the resource manager context (scope) within which database operations is performed.

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

2. Get Status Change

   Check the status of the reader for card insertion, removal, or availability of the reader. This **SCardGetStatusChange** function blocks execution until the current availability of the cards in a specific set of readers change. The caller supplies a list of monitored readers and the maximum wait time (in milliseconds) for an action to occur on one of the listed readers.

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

3. List Readers

   Gets a list of all PC/SC readers using the **SCardListReaders** function. Look for **OMNIKEY CardMan 5326** in the returned list. If multiple OMNIKEY Contactless Smart Card readers are connected to your system, they will be enumerated.

   Example: OMNIKEY CardMan 5326 1, and OMNIKEY CardMan 5x21-CL 2.

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

4. Connect

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

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

5. Exchange Data and Commands with the Card or the reader

   Exchange command and data through APDUs. The **SCardTransmit** function sends a service request to the smart card, expecting to receive data back from the card.

```
LONG SCardTransmit( IN SCARDHANDLE hCard,
                    IN LPCSCARD_I0_REQUEST pioSendPci,
                    IN LPCBYTE pbSendBuffer,
                    IN DWORD cbSendLength,
                    IN OUT LPSCARD_IO_REQUEST pioRecvPci,
                    OUT LPBYTE pbRecvBuffer,
                    IN OUT LPDWORD pcbRecvLength);
```

Note: In environments not allowing SCardTransmit() without an ICC or caused by any other reasons or developers preferences the application can communicate via Control().

The application should retrieve the control code corresponding to FEATURE_CCID_ESC_COMMAND (see part 10, rev.2.02.07). In case this feature is not returned, the application may try SCARD_CTL_CODE(3500) as control code to use.

```
LONG SCardControl( IN SCARDHANDLE hCard,
                   IN DWORD dwControlCode,
                   IN LPCVOID lpInBuffer,
                   IN DWORD nInBufferSize,
                   OUT LPVOID lpOutBuffer,
                   IN DWORD nOutBufferSize,
                   OUT LPDWORD lpBytesReturned);
```

6. Disconnect

   It is not absolutely necessary to disconnect the card after the completion of all transactions, but it is recommended. The **SCardDisconnect** function terminates a connection previously opened between the calling application and a smart card in the target reader.

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

7. Release

   This step ensures all system resources are released. The **SCardReleaseContext** function closes an established resource manager context, freeing any resources allocated under that context.

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

2.3 Contactless specific PC/SC commands

The PC/SC command set for contactless cards is defined in section 3.2 of the document “Interoperability Specification for ICCs and Personal Computer Systems - Part 3. Requirements for PC-Connected Interface Devices”, and is available from the PC/SC Workgroup website http://www.pcscworkgroup.com. The commands use standard APDU syntax and standard SCardTransmit API, but use the reserved value of the CLA byte of ‘FF’.

Supported Reader Commands

Instruction	Description	Comments
0xCA	Get Data	Partially supported (only UID)
0x70	Vendor Specific	Fully support for all vendor specific generic commands
0x82	Load Keys	Partially supported (only Reader key)

Common SW1SW2 return codes

SW1SW2	Meaning
0x9000	Operation successful
0x6700	Wrong length (Lc or Le)
0x6A81	Function not supported
0x6B00	Wrong parameter (P1 or P2)
0xC0XX	Wrong length (wrong number Le; 'XX' encodes the exact number) if Le is less than the available UID length)
0x6F00	Operation failed

2.3.1 Get Data

This Get Data command will retrieve the UID of an inserted card.

This command can be used with or without an established secure channel. See Chapter 4.1 for a code example.

Command APDU

CLA	INS	P1	P2	Lc	Data In	Le
0xFF	0xCA	0x00	0x00	-	-	XX

Response APDU

P1	Card type	Data Out	SW1SW2	
0x00	iCLASS 15693	8-byte CSN	0x9000	Operation successful
	Other	-	0x6A81	Function not supported

2.3.2 Vendor specific generic command

This command allows applications to control OMNIKEY specific features provided by the reader and can only be used in secure mode. For an example see Chapter 4.1.

Command APDU

CLA	INS	P1	P2	Lc	Data Field	Le
FF	70	07	6B	xx	DER TLV coded PDU (Vendor Payload)	xx

Vendor Command	Tag	Vendor Payload Branch
FF 70 07 6C Lc	00h	sioApi [A0h]
	01h	manageSecureSession [A1h]
	02h	readerInformationApi [A2h]
	0Dh	response [BDh] or [9Dh] (primitive)
	0Eh	errorResponse [BE] or [9Eh] (primitive)

Response APDU

Data field	SW1 SW2
DER TLV Response PDU	See ISO 7816-4

The response APDU is encapsulated in the response TAG or error response TAG. In cases of internal errors, the IFD returns SW1SW2 = 9000 and the data field is encapsulated in the error response tag. In cases of an ISO 7816 violation, the return code is according to ISO 7816-4 and the data field is empty.

Error Response:

The DFR error response can be caused by two processes

SIO Processor exception
If the error response is caused by the SIO processor then the error response TAG is BEh
(Class Context Specific) + (Constructed) + (0Eh). For details, see the following table, **Error Response Message**.

Error Response Message

BE 07 80 01 err 81 02 sw1 sw2		
Value	Description	ASN.1 Encoding Notes
0xBE	Tag = ErrorResponse (0x1E)	Constructed Type => 0xBE
0x07	Len = 7	
0x80	Tag = ErrorCode (0x00)	Primitive Type => 0x80
0x01	Len = 1	
Err	VALUE = see table **Error Codes**	ENUMERATED
0x81	Tag = Data (0x01)	Primitive Type => 0x81
0x02	Len = 2	
sw1 sw2	VALUE = Status Word specific to the Error Code	OCTET STRING

Error Codes

Name	Value	Notes
erCommunicationError	0x00	A communications error was detected.
erCardNotFound	0x01	A card was not found by the performAnti-collision command.
erNotSupported	0x03	Command not supported in current version of the SIO Processor.
erTlvNotFound	0x04	Message TLV not found in current version of the SIO Processor.
erTlvMalformed	0x05	The message TLV is not constructed properly.
erIso7816Exception	0x06	An unrecoverable violation of ISO7816 occurred
erSIOError	0x22	Exception from SIO processing of a card
erSIOProcessorException	0x3C	Exception from SIO Processor, likely due to an invalid parameter

A) DFR PC/SC handler exception

If the error response is caused by the DFR Firmware core then the error response TAG is 9Eh (Class Context Specific) + (Primitive) + (0Eh). Length is 2 byte. First byte is the cycle in which the error is occurred and the second byte is the exception type.

9E 02 xx yy 90 00	
Value	**Description**
9Eh	Tag = Error Response (0Eh) + (Class Context Specific) + (Primitive)
02h	Len = 2
cycle	Value byte 1: Cycle in which the error is occurred, see Error Cycle
error	Value byte 2: Error code, see Error Code
SW1	90
SW2	00

Error Cycle

First value byte	
Cycle	**Description**
0	HID Proprietary Command APDU
1	HID Proprietary Response APDU
2	HID Read or Write DFR EEPROM Structure
3	RFU
4	SIO Processor Process Command APDU
5	SIO Processor Process Response APDU

Error Code

Second value byte		
Exception		**Description**
3	03h	NOT_SUPPORTED
4	04h	TLV_NOT_FOUND
5	05h	TLV_MALFORMED
6	06	ISO_EXCEPTION
13	1Dh	OUT_OF_PERSISTENT_MEMORY
17	11h	INVALID_STORE_OPERATION
19	13h	TLV_INVALID_SETLENGTH
20	14h	TLV_INSUFFICIENT_BUFFER
21	15h	DATA_OBJECT_READONLY
31	1F	APPLICATION_EXCEPTION (Destination Node ID mismatch)
42	2Ah	MEDIA_TRANSMIT_EXCEPTION (Destination Node ID mismatch)
43	2Bh	SAM_INSUFFICIENT_MSGHEADER (Secure Channel ID not allowed)
47	2Fh	TLV_INVALID_INDEX

2.3.3 Load Keys

For an update of the reader key a secure channel is mandatory.

See 4.3 for a code example.

Command	Class	Ins	P1	P2	Lc	Data In	Le
Load Keys	0xFF	0x82	0x20	Key Number	0x10	Key	---

Type	SW1	SW2	Description
Normal	0x90	0x00	Successful
Execution Error	0x64	0x00	No Response from media (Time Out)
	0x65	0x81	Not usable block number in the memory area (Memory failure)
Checking Error	0x67	0x00	Wrong APDU length
	0x69	0x82	Block not authenticated (Security status not satisfied)

3 Objects and Items

3.1 Overview

The reader presents smart card information as well as reader information as ASN.1 objects/items, whereby every object is identified by a unique Object Identifier (OID). A special kind of object is a Secure Identity Object (SIO).

3.2 SIO Processor

The HID SIO (Secure Identity Object™) is a data model for storing and transporting identity information in a single object. SIOs consist of a number of independent but associated data objects for such items as physical access control (for example, card numbers), finger print templates, and cash on card. The collection of this information in an SIO ensures the proper coupling of related data (that is, guaranteeing that one individual's card is not associated with another individual's fingerprint. Deploy SIOs in any number of form factors, including contactless and contact smart cards, smart phones, and USB tokens. When combined with an SIO interpreter on the authentication (or reader) side an SIO based system functions the same as a traditional card and reader systems with enhanced levels of **Security, Portability and Flexibility.**

3.3 The OID Tree

OIDs are organized as a tree under an “invisible” root node. The following table shows the first root nodes.

Object sub tree	Tag Value (hex)	Description
sioApi	0xA0	SIO API, equivalent to SAMCommand
manageSecureSession	0xA1	Establish and manage a secure session
readerInformationApi	0xA2	Reader information API
hidMediaPdu	0xA3	HID media specific PDU
nativeCardCommand	0xA4	Native Smart Card commands
humanInterfaceCommand	0xA5	Control of human interfaces
contactSmartCardCtrl	0xA6	Control of contact smart card parameters
deviceSpecificCommand	0xBC	Device specific command set
response	0xBD	Response
errorResponse	0xBE	Error Response

3.4 Secure Channel

3.4.1 Overview

OK5326 DFR provides a Secure Channel for a secure communication between Host application and reader. Benefits of using the Secure Channel are:

- Protect the data communication on the USB channel from eavesdropping
- Protect the host application from replay attacks

For certain operations (for example, reading of PAC bits from HID iCLASS cards) a secure channel is mandatory.

For a secure channel transmission the SCardConnect should be used with ShareMode = SCARD_SHARE_EXCLUSIVE. If a secure channel as established successful, then IFD do not execute polling activities. The Client (host application) must ensure the correct termination of the secure cannel after the last transaction.

The procedure to establish a secured channel is achieved in two phases, AUTH1 and AUTH2.

Afterwards “Client” is the host application and “Server” is the IFD.

Note: In principle, manage the Secure Session by processing the SIO API (see 3.3).

Independent of the method the SIO Processor informs the dispatcher if the secure cannel is established and terminated.

SAM sends this message “Core Command” with Node ID router (reader core).

// sam informs dispatcher about established SC

0A010A000081 a502 8800 9000

// dispatcher ACK

A0DA02630000nn 010a00000081 bd820002 8200 0000

// sam terminates the SC and informs the dispatcher about it

0A010A000081 a502 8900 9000

// dispatcher ACK - here SC UID doesn't have to be set anymore

A0DA02630000nn 010a00000000 bd820002 8200 0000

Note: The key for establishing a secure channel is available under NDA. For a key update see chapter 4.3.

3.4.2 Initialize Secure Channel (AUTH1)

For initialize the secured channel the client must send an 8 byte RND.A and the key number. By means of the Key Number the Client can establish a secured read only session or a secured read / write session.

DER TLV PDU:

```
A1 12                                  // CHOICE ManageSECS
  A0 10                                // CHOICE EstablishAUTH1
    80 01 00                           // VersionSECCH (Currently SAM
                                       ignores this value)
       81 01 yy                        // Key Number (OID)
       82 08 xx xx xx xx xx xx xx xx   // RND.A
```

Note: Currently, the SIO processor ignores the value of version Tag (RFU). Code RFU as 0.

Response APDU:

```
Data field       SW1 SW2
9D 20
uu uu uu uu uu uu uu uu        // 8 byte UID
rr rr rr rr rr rr rr rr        // 8 byte RND.B
xx xx xx xx xx xx xx xx        // 16 byte Reader Cryptogram
xx xx xx xx xx xx xx xx        See Table 1
```

The complete APDU is:

FF 70 07 6B 14 A1 12 A0 10 80 01 00 81 01 yy 82 08 xx xx xx xx xx xx xx xx 00

3.4.3 Initialize Secure Channel (AUTH2)

With the second authentication phase is the establishment of the secured channel finished.

DER TLV PDU:

```
A1 26                                              // CHOICE ManageSECS
  A1 24                                            // CHOICE EstablishAUTH2
    80 10 xx xx xx xx xx xx xx xx      // xx = ClientCryptogram
        xx xx xx xx xx xx xx xx
    81 10 yy yy yy yy yy yy yy yy              // yy = C-MAC
        yy yy yy yy yy yy yy yy
```

Response APDU:

```
Data field      SW1 SW2
9D 10
yy yy yy yy yy yy yy yy           // 16 byte R-MAC
yy yy yy yy yy yy yy yy See Table 1 - Secure Channel Return codes.
```

The complete APDU is:

FF 70 07 6B 28 A1 26 A1 24 80 10 xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx 81 10 yy yy yy yy yy yy yy yy yy yy yy yy yy yy yy yy 00.

3.4.4 Terminate Secure Channel

The Session is terminated if an error occurred (bad client cryptogram) or if the Client terminates the session. In both cases the IFD deletes the session keys S-MAC1, S-MAC2 and S-ENC. The IFD must ensure that the card loses the security state.

DER TLV PDU:

```
A1 02                                       // CHOICE ManageSECS
A2 00                                   // CHOICE terminateSecuredSession
```

This message is always encrypted in the secure channel and is never send plain.

```
Plain APDU   PADDING
FF 70 07 6B 04 A1 02 A2 00   80 00 00 00 00 00 00
```

Encrypted message:

```
FF 70 07 6B
20
xx xx xx xx xx xx xx xx       // xx = Enc(APDU+PADDING, S-ENC)
xx xx xx xx xx xx xx xx
yy yy yy yy yy yy yy yy       // yy = C-MAC
yy yy yy yy yy yy yy yy
00
```

3.4.5 Security Engine Selection

According to clause 3.1 the IFD can support the native firmware security or (and) the SIO Processor. Both methods use the same unified key numbering schema. The Host application has the opportunity to choose the default security engine, if the IFD supports both. The physical key storage is managed by the IFD firmware.

DER TLV PDU:

```
A1 02                                       // CHOICE ManageSECS
  A3 00                                     // CHOICE SecureElementEngine


A1 02                                       // CHOICE ManageSECS
  A4 00                                     // CHOICE NativeSecurityEngine
```

Response APDU:

```
Data field     SW1 SW2
9D 00 (See Table 1 - Secure Channel Return codes)
```

Table 1 - Secure Channel Return codes

Type	SW1	SW2	Description
Normal	90	00	Successful
Execution Error	64	00	No Response from endpoint
Checking Error	67	00	Wrong APDU length
	69	82	Security status not satisfied

4 Migration Scenarios

4.1 Get CSN

The CSN of a smart card can be read using the PC/SC command Get DATA (see chapter 2.3.1). Send the following command with the function ScardTransmit after a shared connection to the card has been established (see chapter 2.2).

Example: Reading iCLASS card CSN

```
Command:
FF CA                      // Get Data
  00 00                    // Get UID
    08                     // Le
Response:
EF 8B AF 00 FB FF 12 E0    // CSN
  90 00                    // SW1 SW2
```

Example: Reading HID PROX card CSN

```
FF CA                      // Get Data
  00 00                    // Get UID
    00                     // Le
Response:
6A 81                      // SW1 SW2 -> Function not supported
```

4.2 Get PAC Bits

OMNIKEY 5x21 or 5x25 generates an ATS which contains the PACS bits. The OMNIKEY 5326 introduces a new method to retrieve those bytes in a card independent way.

The command "Get PAC Bits" returns the Physical Access Control bits of the inserted media. The "Get PAC Bits" command is internal mapped to "GetContent Element". If the inserted media is not supported by the SIO processor, the GET PAC Bits command is processed by the IFD firmware. This is the use case for an IFD which uses the SDR LF controller to read the HID PROX media. The GET PAC Bits command is used in the following coding

DER TLV PDU:

```
A0 05                    // CHOICE SioAPI
  A1 03                  // CHOICE SamCommandGetContentElement
    80 01                // Sequence ContentElementTag
      04                 // Value = implicitFormatPhysicalAccessBits
```

The complete APDU for LF (HID PROX) media is:

FF 70 07 6B 07 A0 05 A1 03 80 01 04 00

For all media which are supported by the SIO processor, a secure channel is mandatory to perform the GET PAC Bits command. In a secure channel, the GET PAC Bits command must comprise the root OID of the Secure Object or the virtual OID for legacy cards.

```
A0 13                    // CHOICE SioAPI
  A1 11                  // CHOICE SamCommandGetContentElement
    80 01 04             // ContentElementTag = implicitFormatPhysicalAccessBits
    84 0C 2B 06 01 04    // SoRootOID (virtual OID)
        01 81 E4 38
        01 01 02 04
```

The plain command APDU for HF media (iCLASS), supported by the SIO Processor is:

FF 70 07 6B 15 A0 13 A1 11 80 01 04 84 0C 2B 06 01 04 01 81 E4 38 01 01 02 04 00

FF 70 07 6B 30 { Enc(plain APDU + PADDING, S-ENC) + C-MAC } 00

The command to send is encrypted according to clause 4.2.2.3.

Response:

```
9D                    // Tag = Response (1D)
  xx                  // length of PAC Bits
    PAC BIT STRING 1) // UNIVERSAL BIT_STRING TAG (1st byte) = 03
                      // Length of UNIVERSAL BIT_STRING (2nd byte) = nn
                      // Unused Bits (3rd byte) is the number of trailing 0s in the last
                      byte
                      // PAC bits (ex 4th byte, nn-1 bytes) including required trailing
                      zeros
```

Example response for 35bit PAC bit string 1):

```
9D
  08
    03 06 05 81 ED BE 15 60
```

1) Note: In use case of a secure channel is the PAC BIT STRING a server cryptogram (see 3.4).

4.3 Update Reader Key

Update the reader key with the Load Keys (see chapter **Error! Reference source not found.**) command. Establish a secure channel and transmit the following command:

Example: Updating the reader key

```
FF 82                                              // Load Keys
 20 80                                             // Reader key slot number 80
  10                                               // Le
   11 22 33 44 55 66 77 88 99 00 AA BB CC DD EE FF // New reader key
```

Response:

```
90 00                                              // SW1 SW2
```

5 Reader Configuration

5.1 Overview

All OMNIKEY 5326 DFR configuration items are identified by a unique ASN.1 leaf. The root is defined as Reader Information API and is encapsulated in a vendor specific generic command see 2.3.2. For a READ command the Le byte must be present. The IFD reply is encapsulated in the Tag BDh and each leaf is encapsulated in the leaf Tag.

Under this root are a number of branches, organized as follows:

Reader Information Structure

Vendor Command	Reader Information API	Request	Branch
FF 70 07 6B Lc	Tag = A2h	Get [A0h] Set [A1h]	readerCapabilities [A0h]
			readerHostInterface [A1h]
			readerInformationVersion [88h]
			readerConfigurationControl [A9h]

Appendix 63 lists all available objects.

5.2 Example Get Product Name

Command:

```
FF 70 07 6B 08       // Vendor Specific APDU with 8 bytes object
  A2 06              // Reader Information API
    A0 04            // Get Request
      A0 02          // Reader Capabilities
        82 00        // Product Name
        00           // Le
```

Reply:

```
BD 0F                                      // Response
  82 0D                                    // Product Name
    4F 4D 4E 49 4B 45 59 20 35 33 32 36 00 // OMNIKEY 5326
    90 00                                  // SW1 SW2
```

6 Code Examples

6.1 Initialize Secure Channel

```
bool CReaders::EstablishSecureChannel(CString strKey, unsigned int uiVID, CString
strKeyRef)
{
        bool    fRet = true;
        CString strChallenge;
        CString strServerAuthentication;
        CString strClientAuthData;
        CString strFirstRMAC;
        CString strIN;
        CString strOUT;

        do
        {
                // initialize the secure channel and get the client challange back
                if ( strKey.GetLength() != 32 )
                {
                        fRet = false;
                        break;
                }
                if ( !m_secChannel.Init(strKey, &strChallenge) )
                {
                        fRet = false;
                        break;
                }

                // send init command to the reader
                if ( strChallenge.GetLength() != 16 )
                {
                        fRet = false;
                        break;
                }
                if ( strKeyRef.GetLength() != 2 )
                {
                        fRet = false;
                        break;
                }
                strIN.Format("%s%s%s%s", m_cstrInitAuthD.Left(18), strKeyRef,
m_cstrInitAuthD.Right(4), strChallenge);
                if ( !TransmitIFDspecific( uiVID, strIN, &strOUT) )
                {
                        fRet = false;
                        break;
                }

                // get the server authentication data from the received message
                if (
strOUT.Left(m_cstrInitAuthDReply.GetLength()).CompareNoCase(m_cstrInitAuthDReply) !=
0)
                {
                        if (
strOUT.Left(m_cstrInitAuthDReplyS.GetLength()).CompareNoCase(m_cstrInitAuthDReplyS)
!= 0)
                        {
                            fRet = false;
                            break;
                        }
                }
                strServerAuthentication = strOUT.Mid(strOUT.GetLength() - 68, 64);

                // check server authentication
                if ( !m_secChannel.CheckServerAutentication(strServerAuthentication,
&strClientAuthData) )
                {
                        fRet = false;
                        break;
                }
```

```
            // send cont auth to the reader
            strIN.Format("%s%s", m_cstrContAuthD, strClientAuthData.Right(0x20));
            strIN.Replace("x", strClientAuthData.Left(0x20));
            if ( !TransmitIFDspecific( uiVID, strIN, &strOUT) )
            {
                fRet = false;
                break;
            }
            // check reply to cont BD8200128A10
            if ( strOUT.Left(m_cstrContAuthDReply.GetLength()).CompareNoCase(m_cstrContAuthDReply) != 0 )
            {
                if ( strOUT.Left(m_cstrContAuthDReplyS.GetLength()).CompareNoCase(m_cstrContAuthDReplyS) != 0 )
                {
                    fRet = false;
                    break;
                }
            }
            strFirstRMAC = strOUT.Mid(strOUT.GetLength() - 36, 32);
            if ( !m_secChannel.Unwrap(strFirstRMAC, &strOUT) )
            {
                fRet = false;
                break;
            }
            // secure channel is now established and ready to use
            m_SecChannelIs = true;
        } while (false);

        return fRet;
}
```

6.2 Terminate Secure Channel

```
bool CReaders::TerminateSecureChannel(unsigned int uiVID)
{
        bool      fRet = true;
        CString strOUT;
        CString strIN;
        CString strReceive;

        do
        {
                if ( !m_secChannel.WrapInput(m_cstrTerminateSecCh, &strIN) )
                {
                        fRet = false;
                        break;
                }
                if ( !TransmitIFDspecific(uiVID, strIN, &strOUT) )
                {
                        fRet = false;
                        break;
                }

                if ( strOUT.GetLength() >= 72 )
                {
                        // error response
                        if (strOUT.Left(2).CompareNoCase("BE") == 0)
                                strOUT.Delete(0, 4);
                        else
                                // short coded length
                                if (strOUT.Mid(3, 2).CompareNoCase("82") != 0)
                                        strOUT.Delete(0, 4);
                                // long coded length
                                else
                                        strOUT.Delete(0, 12);
                        // reove sw1sw2
                strOUT.Delete(strOUT.GetLength() - 4, 4);
                if ( !m_secChannel.Unwrap(strOUT, &strReceive) )
                {
                    fRet = false;
                    break;
                }
            }

            if ( (strOUT.CompareNoCase(m_cstrChannelTerminated) != 0) &&
(strOUT.CompareNoCase(m_cstrChannelTerminatedS) != 0) )
            {
                fRet = false;
                break;
            }
        } while (false);
        m_SecChannelIs = false;
        return fRet;
}
```

6.3 Transmit IFD Specific

```
bool CReaders::TransmitIFDspecific(unsigned int uiVID, CString strSend, CString
*strReceive)
{
    DWORD wReturnCode = SCARD_E_CANCELLED;
    CString strData;
    char acIN[512];
    char acOUT[512];
    DWORD dwINsize                = sizeof(acIN);
    DWORD dwOUTsize               = sizeof(acOUT);
    if (!m_fIsConnected)
         return false;
    // basic check of the input string
    if ( (strSend.GetLength() % 2) != 0)
         return false;

    strData.Format("%s%04X%02X%s00", m_cstrIFDspecificCLAINS, uiVID,
(strSend.GetLength() / 2), strSend);
    ConvertStringToHex(strData, acIN, &dwINsize);
    if (m_dwConnectionMode == SCARD_SHARE_SHARED)
         wReturnCode = SCardTransmit(m_hCard, SCARD_PCI_T1, (LPCBYTE)acIN,
dwINsize, NULL, (LPBYTE)acOUT, &dwOUTsize);
    else
         wReturnCode = SCardTransmit(m_hCard, SCARD_PCI_RAW,
(LPCBYTE)acIN, dwINsize, NULL, (LPBYTE)acOUT, &dwOUTsize);
    if ( wReturnCode == SCARD_S_SUCCESS )
    {
         ConvertHexToString(acOUT, dwOUTsize, strReceive);
    }
    else
    {
         *strReceive = "";
         return false;
    }
    return true;
}
```

6.4 Transmit PCSC

```
bool CReaders::TransmitPCSC(CString strSend, CString *strReceive)
{
    DWORD wReturnCode = SCARD_E_CANCELLED;
    CString strData;
    char acIN[512];
    char acOUT[512];
    DWORD dwINsize                    = sizeof(acIN);
    DWORD dwOUTsize                   = sizeof(acOUT);

    ConvertStringToHex(strSend, acIN, &dwINsize);

    if (m_dwConnectionMode == SCARD_SHARE_SHARED)
        wReturnCode = SCardTransmit(m_hCard, SCARD_PCI_T1, (LPCBYTE)acIN,
dwINsize, NULL, (LPBYTE)acOUT, &dwOUTsize);
    else
        wReturnCode = SCardTransmit(m_hCard, SCARD_PCI_RAW,
(LPCBYTE)acIN, dwINsize, NULL, (LPBYTE)acOUT, &dwOUTsize);

    if ( wReturnCode == SCARD_S_SUCCESS )
    {
        ConvertHexToString(acOUT, dwOUTsize, strReceive);
    }
    else
    {
        strReceive->Format("returncode 0X%08X", wReturnCode);
        return false;
    }

    return true;
}
```

6.5 Transmit PCSC UID

```
bool CReaders::TransmitPCSC(CString strSend, CString *strReceive, unsigned int uiVID)
{
        bool fRet = true;
        CString strIN;
        CString strOUT;
        CString strTemp;

        do
        {
                // check if there is a secure channel
                if ( !m_SecChannelIs )
                {
                        fRet = TransmitPCSC(strSend, strReceive);
                        break;
                }
                // wrap the input
                if ( !m_secChannel.WrapInput(strSend, &strIN) )
                {
                        fRet = false;
                        break;
                }
      // WW begin
      // strIN.SetAt(strIN.GetLength()-1,'F');
      // strIN.SetAt(strIN.GetLength()-2,'F');
      // WW end

                // send via IFD specific
                if ( !TransmitIFDspecific(uiVID, strIN, &strOUT) )
                {
                        fRet = false;
                        break;
                }
                if ( strOUT.GetLength() >= 72 )
                {
                        // error response
                        if (strOUT.Left(2).CompareNoCase("BE") == 0)
                            strOUT.Delete(0, 4);
                        else
                            // short coded length
                            if (strOUT.Mid(3, 2).CompareNoCase("82") != 0)
                                strOUT.Delete(0, 4);
                            // long coded length
                            else
                                strOUT.Delete(0, 12);
                        // reove sw1sw2
                        strOUT.Delete(strOUT.GetLength() - 4, 4);
                        if ( !m_secChannel.Unwrap(strOUT, strReceive) )
                        {
                            fRet = false;
                            break;
                        }
                }else
                {
                        strReceive->Format("%s", strOUT);
                }
        } while (false);

        return fRet;
}
```

6.6 Get UID

```
void CTestApp_PCportDlg::OnBnClickedButtonGetuid()
{
    CString strData;

    UpdateData(true);

    unsigned int uiVID = 0;
    if (sscanf_s(m_strVID, "%x", &uiVID) != 1)
    {
        uiVID = 0;
    }
    if (m_cReaders.TransmitPCSC(m_cstrGetsDataUID, &strData, uiVID))
    {
        if (strData.GetLength() >= 4)
        {
            m_strPCSCSW1SW2 = strData.Right(4);
            m_strPCSCGetDataResult = strData.Left(strData.GetLength() - 4);
        }
    }
    else
    {
        MessageBox("problems in Transmit", "Hint", MB_OK);
        m_strPCSCGetDataResult = strData;
    }

    UpdateData(false);
}
```

6.7 Transmit IFD Specific

```
bool CReaders::TransmitIFDspecific(unsigned int uiVID, CString strSend, CString *strReceive)
{
    DWORD wReturnCode = SCARD_E_CANCELLED;
    CString strData;
    char acIN[512];
    char acOUT[512];
    DWORD dwINsize                      = sizeof(acIN);
    DWORD dwOUTsize                     = sizeof(acOUT);
    if (!m_fIsConnected)
        return false;
    // basic check of the input string
    if ( (strSend.GetLength() % 2) != 0)
        return false;
    strData.Format("%s%04X%02X%s00", m_cstrIFDspecificCLAINS, uiVID,
(strSend.GetLength() / 2), strSend);
    ConvertStringToHex(strData, acIN, &dwINsize);
    if (m_dwConnectionMode == SCARD_SHARE_SHARED)
        wReturnCode = SCardTransmit(m_hCard, SCARD_PCI_T1, (LPCBYTE)acIN,
dwINsize, NULL, (LPBYTE)acOUT, &dwOUTsize);
    else
        wReturnCode = SCardTransmit(m_hCard, SCARD_PCI_RAW, (LPCBYTE)acIN,
dwINsize, NULL, (LPBYTE)acOUT, &dwOUTsize);
    if ( wReturnCode == SCARD_S_SUCCESS )
    {
        ConvertHexToString(acOUT, dwOUTsize, strReceive);
    }
    else
    {
        *strReceive = "";
        return false;
    }

    return true;
}
```

6.8 Get PAC Bits

```
void CTestApp_PCportDlg::OnBnClickedButtonGetpacbits()
{
    unsigned int uiVID = 0;
    CString     strReceive, strSend;
    UpdateData(true);
    if (sscanf_s(m_strVID, "%x", &uiVID) == 1)
    {
        if ( m_cReaders.SecureChannelIsEstablished() )
        {
            strSend.Format("%s%04X%02X%s00", m_cstrIFDspecificCLAINS, uiVID,
(m_cstrGetPACBitswOID.GetLength() / 2), m_cstrGetPACBitswOID);
//  strSend.Format("%s", m_cstrGetPACBitswOID);
//          strSend.Format("FF70076B07A005A10380010400");
//          strSend.Format("A103800104");
            if (m_cReaders.TransmitPCSC(strSend, &strReceive, uiVID))
            {
                m_strPACBits = strReceive;
            }
            else
            {
                MessageBox("problems in Transmit", "Hint", MB_OK);
            }
        }
        else if (m_cReaders.TransmitIFDspecific(uiVID, m_cstrGetPACBits, &strReceive))
        {
            m_strPACBits = strReceive;
        }
        else
        {
            MessageBox("problems in Transmit", "Hint", MB_OK);
        }
    }
    else
        MessageBox("please enter a valid VID", "Hint", MB_OK);
    UpdateData(false);
}
```

7 Secure Channel Sample Class Implementation

7.1 Constructor

```
CSecureChannel::CSecureChannel()
{
        m_hProv                 = NULL;
        m_hMK           = NULL;
        m_hSCBK                 = NULL;
        m_hSMAC1        = NULL;
        m_hSMAC2        = NULL;
        m_hSENC                 = NULL;
        memset((void*)m_abICV, 0x00, sizeof(m_abICV));
}
```

7.2 Destructor

```
CSecureChannel::~CSecureChannel(void)
{
        // release the resources
        if(m_hProv)
  {
                CryptReleaseContext(m_hProv, 0);
  }
        if(m_hSCBK)
        {
                CryptDestroyKey(m_hSCBK);
        }
        if(m_hSMAC1)
        {
                CryptDestroyKey(m_hSMAC1);
        }
        if(m_hSMAC2)
        {
                CryptDestroyKey(m_hSMAC2);
        }
        if(m_hSENC)
        {
                CryptDestroyKey(m_hSENC);
        }
        if(m_hMK)
        {
                CryptDestroyKey(m_hMK);
        }
}
```

7.3 Initialize 1

```
bool CSecureChannel::Init(BYTE* abKey, DWORD* dwLength, BYTE* abChallenge, DWORD*
dwLengthCh)
{
        BYTE  pbData[100];
        DWORD dwSize = sizeof(pbData);
        HCRYPTKEY phKey;

        struct {
                BLOBHEADER hdr;
                DWORD                  cbKeySize;
                BYTE           rgbKeyData [16];
        } myBlob;

        if (*dwLength != 16)
                return false;

        if(!CryptAcquireContext(   &m_hProv,
                                                  0,
                                                  NULL,//MS_ENH_RSA_AES_PROV,
                                                  PROV_RSA_AES,
                                                  CRYPT_NEWKEYSET)) // 0))
        {
          if(!CryptAcquireContext(   &m_hProv,
                             0,
                             NULL,//MS_ENH_RSA_AES_PROV,
                             PROV_RSA_AES,
                             0))
        {
          return false;
        }
    }

    memcpy(m_abMasterKey, abKey, 16);

    // generate the client challange (8 byte random number)
    if (CryptGenKey(m_hProv, CALG_AES_128, CRYPT_EXPORTABLE, &phKey))
    {
        if (!CryptExportKey(phKey, NULL, PLAINTEXTKEYBLOB, 0, pbData, &dwSize))
            return false;

        if (dwSize == sizeof(myBlob))
        {
                BYTE clientChallenge[] =
{0x46,0x90,0x2B,0x57,0xC6,0x8D,0x14,0x90};
                memcpy(m_abClientChallenge, clientChallenge,
sizeof(clientChallenge));

            //memcpy(m_abClientChallenge, &pbData[dwSize-16], 8);
        }
        else
            return false;
        CryptDestroyKey(phKey);

        // import the master key
        myBlob.hdr.bType         = PLAINTEXTKEYBLOB;
        myBlob.hdr.bVersion      = CUR_BLOB_VERSION;
        myBlob.hdr.reserved      = 0;
        myBlob.hdr.aiKeyAlg = CALG_AES_128;
        myBlob.cbKeySize    = 16;
        memcpy(myBlob.rgbKeyData, m_abMasterKey, 16);
        if (!CryptImportKey(m_hProv, (BYTE*)&myBlob, sizeof(myBlob), NULL, 0, &m_hMK))
        {
            return false;
        }
    }
    if (*dwLengthCh < 8)
        return false;
    memcpy(abChallenge, m_abClientChallenge, 8);
    *dwLengthCh = 8;

    return true;
}
```

7.4 Initialize 2

```
bool CSecureChannel::Init(CString strKey, CString* strChallenge)
{
	bool fRet = true;
	BYTE  abKey[32];
	DWORD dwKSize = sizeof(abKey);
	BYTE  abChallenge[32];
	DWORD dwCSize = sizeof(abChallenge);

	do
	{
		if ( !ConvertStringToHex(strKey, abKey, &dwKSize) )
		{
			fRet = false;
			break;
		}
		if ( !Init(abKey, &dwKSize, abChallenge, &dwCSize) )
		{
			fRet= false;
			break;
		}
		if ( !ConvertHexToString(abChallenge, dwCSize, strChallenge) )
		{
			fRet = false;
			break;
		}
	} while (false);

	return fRet;
}
```

7.5 Check Server Authentication 1

```
bool CSecureChannel::CheckServerAutentication(BYTE* abServerAuthData, DWORD*
dwLengthAuthData, BYTE* abClientAuthData, DWORD* dwLengthCl)
{
    BYTE abKeyInput[16];
    BYTE abClientCryptogram[16];
    BYTE abServerCryptogram[16];
    BYTE abIV[16];
    BYTE pbData[100];
    DWORD dwSizeCMAC = sizeof(m_abCMAC);

    memset(abClientCryptogram, 0x00, sizeof(abClientCryptogram));
    memset(abServerCryptogram, 0x00, sizeof(abServerCryptogram));
    memset(m_abCMAC, 0x00, sizeof(m_abCMAC));
    memset(abIV, 0x00, sizeof(abIV));

    DWORD dwSize = sizeof(abKeyInput);

    if (*dwLengthAuthData != 32)
        return false;

    if (*dwLengthCl != 32)
        return false;

    memcpy(m_abUID, abServerAuthData, 8);
    for (int ii = 0; ii < 8; ++ii)
    {
        abKeyInput[8+ii] = ~m_abUID[ii];
        abKeyInput[ii] = m_abUID[ii];
    }

    // generate the basekey
    DWORD dwData = CRYPT_MODE_ECB;
```

```
        if (!CryptSetKeyParam(m_hMK, KP_MODE, (BYTE*)&dwData, 0))
        {
                return false;
        }
        if (!ImportKey(&m_hSCBK, abKeyInput, &dwSize, m_hMK))
        {
                return false;
        }
        dwSize = sizeof(pbData);
        if (CryptExportKey(m_hSCBK, NULL, PLAINTEXTKEYBLOB, 0, pbData, &dwSize))
        {
                memcpy(m_abSCBK, &pbData[dwSize-16], 16);
        }

        // save the server challange
        memcpy(m_abServerChallenge, &abServerAuthData[8], 8);

        // compute session keys
        if (!DeriveKeys( (unsigned short)(m_abServerChallenge[0] << 8) + m_abServerChallenge[1]
))
                return false;

        // compute the client cryptogram
        memcpy(abClientCryptogram, m_abServerChallenge, sizeof(m_abServerChallenge));
        memcpy(&abClientCryptogram[8], m_abClientChallenge, sizeof(m_abClientChallenge));
        dwSize = sizeof(abClientCryptogram);
        if (!ComputeCryptogram(abClientCryptogram, &dwSize))
                return false;

        // compute the server cryptogram
        memcpy(abServerCryptogram, m_abClientChallenge, sizeof(m_abClientChallenge));
        memcpy(&abServerCryptogram[8], m_abServerChallenge, sizeof(m_abServerChallenge));
        dwSize = sizeof(abServerCryptogram);
        if (!ComputeCryptogram(abServerCryptogram, &dwSize))
```

```
                return false;


        // check server authentication data
        if (memcmp(&abServerAuthData[16], abServerCryptogram, sizeof(abServerCryptogram)) !=
0)
                return false;


        // check length of the buffer
        if (*dwLengthCl < 32)
                return false;
        // copy client auth data to the output buffer
        memcpy(abClientAuthData, abClientCryptogram, sizeof(abClientCryptogram));
        // calculate the C-MAC
        if (!ComputeMAC(abIV, abClientCryptogram, 0, sizeof(abClientCryptogram), m_abCMAC,
&dwSizeCMAC))
                return false;
        memcpy(&abClientAuthData[16], m_abCMAC, 16);


        return true;
}
```

7.6 Check Server Authentication 2

```
bool CSecureChannel::CheckServerAutentication(CString strServerAuthData, CString* strClientAuthData)
{
        bool fRet = true;
        BYTE pbData1[100];
        BYTE pbData2[32];
        DWORD dwSize1;
        DWORD dwSize2;

        do{
                dwSize1 = sizeof(pbData1);
                if ( !ConvertStringToHex(strServerAuthData, pbData1, &dwSize1) )
                {
                        fRet = false;
                        break;
                }
                dwSize2 = sizeof(pbData2);
                if ( !CheckServerAutentication(pbData1, &dwSize1, pbData2, &dwSize2) )
                {
                        fRet = false;
                        break;
                }
                if ( !ConvertHexToString(pbData2, dwSize2, strClientAuthData) )
                {
                        fRet = false;
                        break;
                }
        } while (false);

        return fRet;
}
```

7.7 Derive Keys

```
bool CSecureChannel::DeriveKeys(unsigned short usSeqCounter)
{
        BYTE abInput[16];
        DWORD dwSize = sizeof(abInput);
        DWORD dwData;
        BYTE  pbData[100];
        memset((void*)abInput, 0x00, sizeof(abInput));

        dwData = CRYPT_MODE_ECB;
        if (!CryptSetKeyParam(m_hSCBK, KP_MODE, (BYTE*)&dwData, 0))
        {
                return false;
        }

        // generate the SMAC1
        abInput[0] = 0x01;
        abInput[1] = 0x01;
        abInput[2] = (BYTE)(usSeqCounter >> 8);
        abInput[3] = (BYTE) usSeqCounter;
        dwSize = sizeof(abInput);
        if (!ImportKey(&m_hSMAC1, abInput, &dwSize, m_hSCBK))
                return false;

        // generate the SMAC2
        memset((void*)abInput, 0x00, sizeof(abInput));
        abInput[0] = 0x01;
        abInput[1] = 0x02;
        abInput[2] = (BYTE)(usSeqCounter >> 8);
        abInput[3] = (BYTE) usSeqCounter;
        dwSize = sizeof(abInput);
        if (!ImportKey(&m_hSMAC2, abInput, &dwSize, m_hSCBK))
                return false;
        // generate the SENC
        memset((void*)abInput, 0x00, sizeof(abInput));
        abInput[0] = 0x01;
        abInput[1] = 0x82;
        abInput[2] = (BYTE)(usSeqCounter >> 8);
        abInput[3] = (BYTE) usSeqCounter;
        dwSize = sizeof(abInput);
        if (!ImportKey(&m_hSENC, abInput, &dwSize, m_hSCBK))
                return false;

        dwSize = sizeof(pbData);
        CryptExportKey(m_hSENC, NULL, PLAINTEXTKEYBLOB, 0, pbData, &dwSize);
        memcpy(m_abSENC, &pbData[dwSize-16], 16);
        dwSize = sizeof(pbData);
        CryptExportKey(m_hSMAC1, NULL, PLAINTEXTKEYBLOB, 0, pbData, &dwSize);
        memcpy(m_abSMAC1, &pbData[dwSize-16], 16);
        dwSize = sizeof(pbData);
        CryptExportKey(m_hSMAC2, NULL, PLAINTEXTKEYBLOB, 0, pbData, &dwSize);
        memcpy(m_abSMAC2, &pbData[dwSize-16], 16);

        return true;
}
```

7.8 Compute Cryptogram

```
bool CSecureChannel::ComputeCryptogram(BYTE* abINOUT, DWORD* dwSize)
{
    DWORD                    dwData   = CRYPT_MODE_ECB;
    DWORD                    dwLength = *dwSize;
    HCRYPTKEY    hDuplicateKey;
    BYTE         abIV[16];
    memset((void*)abIV, 0x00, sizeof(abIV));

    if(*dwSize != 16)
        return false;

    CryptDuplicateKey(m_hSENC, 0, 0, &hDuplicateKey);
    if (!CryptSetKeyParam(hDuplicateKey, KP_MODE, (BYTE*)&dwData, 0))
        return false;
    if (!CryptSetKeyParam(hDuplicateKey, KP_IV, abIV, 0))
        return false;

    if (!CryptEncrypt(hDuplicateKey,
                                NULL,
                                FALSE,
                                0,
                                abINOUT,
                                dwSize,
                                dwLength))
        return false;

    CryptDestroyKey(hDuplicateKey);
    return true;
}
```

7.9 Import Key

```
bool CSecureChannel::ImportKey(HCRYPTKEY* hKey, BYTE* abIN, DWORD* dwSize,
HCRYPTKEY hKeyIN)
{
    DWORD          dwLength     = *dwSize;
    bool    fOK              = true;
    BYTE*   abIV[16];
    HCRYPTKEY hDuplicateKey;
    memset((void*)abIV, 0x00, sizeof(abIV));

    struct {
        BLOBHEADER  hdr;
        DWORD                cbKeySize;
        BYTE          rgbKeyData [16];
    } myBlob;

    // generate the key
    CryptDuplicateKey(hKeyIN, 0, 0, &hDuplicateKey);
    do
    {
        if (!CryptSetKeyParam(hDuplicateKey, KP_IV, (const BYTE*)abIV, 0))
        {
            fOK = false;
            break;
        }
        if (!CryptEncrypt(hDuplicateKey,
                                 NULL,
                                 FALSE,
                                 0,
                                 abIN,
                                 dwSize,
                                 dwLength))
        {
            fOK = false;
            break;
        }
        // import the key
        myBlob.hdr.bType        = PLAINTEXTKEYBLOB;
        myBlob.hdr.bVersion     = CUR_BLOB_VERSION;
        myBlob.hdr.reserved     = 0;
        myBlob.hdr.aiKeyAlg = CALG_AES_128;
        myBlob.cbKeySize    = 16;
        memcpy(myBlob.rgbKeyData, abIN, 16);

        if (!CryptImportKey(m_hProv, (BYTE*)&myBlob, sizeof(myBlob), NULL, 0, hKey))
        {
            fOK = false;
            break;
        }
    } while (false);
    CryptDestroyKey(hDuplicateKey);

    return true;
}
```

7.10 Compute Mac

```
bool CSecureChannel::ComputeMAC(BYTE *abIV, BYTE* abIN, unsigned int uiOffset,
DWORD dwSize, BYTE* abOUT, DWORD* dwSizeOUT)
{
        DWORD           dwLength  = dwSize + 16;
        DWORD     dwNewSize = (DWORD)ceil((dwSize + 1.0) / 16.0) * 16;
        bool      fOK          = true;
        HCRYPTKEY hDuplicateKey;
        BYTE      *abPadded                 = new BYTE [dwNewSize];
        DWORD           dwPaddedSize= dwNewSize;
        BYTE      *abPadded2                = new BYTE [dwNewSize];
        DWORD           dwPaddedSize2        = dwNewSize;
        BYTE      *abIV2        = new byte[16];

        // generate the key
        CryptDuplicateKey(m_hSMAC1, 0, 0, &hDuplicateKey);

        do
        {
                if ((dwSize == 0) || ((dwSize % 16) != 0))
                {
                        if (!Pad(abIN, uiOffset, dwSize, abPadded, &dwPaddedSize))
                        {
                                fOK = false;
                                break;
                        }
                        uiOffset = 0;
                }
                else
                {
                        dwPaddedSize = dwSize;
                        memcpy((void*)abPadded, abIN, (size_t)dwPaddedSize);
                }
                if (dwPaddedSize > 16)
                {
                        memcpy(abPadded2, abPadded, dwPaddedSize);
                        dwPaddedSize2 = dwPaddedSize -16;
                        if (!CryptSetKeyParam(hDuplicateKey, KP_IV, abIV, 0))
                        {
                                fOK = false;
                                break;
                        }
                        if (!CryptEncrypt(hDuplicateKey,
                                                NULL,
                                                FALSE,
                                                0,
                                                &abPadded2[uiOffset],
                                                &dwPaddedSize2,
                                                dwLength))
                        {
                                fOK = false;
                                break;
                        }

          if (dwPaddedSize2 > 0)
          {
            memcpy(abIV2, &abPadded2[dwPaddedSize2-16], 16);
          }
        }
        else
        {
                        memcpy((void*)abIV2, (void*)abIV, 16);
        }

                CryptDestroyKey(hDuplicateKey);
                CryptDuplicateKey(m_hSMAC2, 0, 0, &hDuplicateKey);
                if (!CryptSetKeyParam(hDuplicateKey, KP_IV, abIV2, 0))
                {
                        fOK = false;
                        break;
                }

                memcpy(abPadded2, &abPadded[uiOffset + dwPaddedSize -16], 16);
                dwPaddedSize2 = 16;
                if (!CryptEncrypt(hDuplicateKey,
                            NULL,
                            FALSE,
                            0,
                            abPadded2,
                            &dwPaddedSize2,
                            dwLength))
                {
                        fOK = false;
                        break;
                }
                if (dwPaddedSize2 > *dwSizeOUT)
                {
                        fOK = false;
                        break;
                }

                memcpy(abOUT, abPadded2, dwPaddedSize2);
                *dwSizeOUT = dwPaddedSize2;
            } while (false);
            CryptDestroyKey(hDuplicateKey);
            delete[] abIV2;
            delete[] abPadded;
            delete[] abPadded2;
            return fOK;
        }
```

7.11 Pad

```
bool CSecureChannel::Pad(BYTE* abIN, unsigned int uiOffset, DWORD dwSizeIN, BYTE* abOUT, DWORD *dwSizeOUT)
{
        DWORD dwNewSize = (DWORD)ceil((dwSizeIN + 1.0) / 16.0) * 16;

        if (*dwSizeOUT < dwNewSize)
                return false;

        *dwSizeOUT = dwNewSize;
        memset((void*)abOUT, 0x00, dwNewSize);
   memcpy(abOUT, &abIN[uiOffset], (size_t)dwSizeIN);//)Array.Copy(input, offset, output, 0, length);
   abOUT[dwSizeIN] = 0x80;
        return true;
}
```

7.12 Unpad

```
bool CSecureChannel::WrapInput(BYTE* abIN, DWORD dwINSize, BYTE* abOUT, DWORD *dwOUTSize)
{
        BYTE*    abPlainPadded;
        BYTE     abComplementIV[16];
        bool         fOK = true;
        HCRYPTKEY hDuplicateKey;
        DWORD            dwPlainPadded = (DWORD)ceil((dwINSize + 1.0) / 16.0) * 16;
        DWORD    dwSizeMAC = sizeof(m_abCMAC);

        abPlainPadded          = new BYTE[dwPlainPadded];

        do
        {
                if (!Pad(abIN, 0, dwINSize, abPlainPadded, & dwPlainPadded))
                {
```

```
                fOK = false;
                break;
            }
            for (int ii = 0; ii < 16; ++ii)
            {
                abComplementIV[ii] = ~m_abRMAC[ii];
            }

            CryptDuplicateKey(m_hSENC, 0, 0, &hDuplicateKey);
            if (!CryptSetKeyParam(hDuplicateKey, KP_IV, abComplementIV, 0))
            {
                fOK = false;
                break;
            }
            if (!CryptEncrypt(hDuplicateKey,
                                    NULL,
                                    FALSE,
                                    0,
                                    abPlainPadded,
                                    &dwPlainPadded,
                                    dwPlainPadded))
            {
                fOK = false;
                break;
            }

            if (!ComputeMAC(m_abRMAC, abPlainPadded, 0, dwPlainPadded,
m_abCMAC, &dwSizeMAC))
            {
                fOK = false;
                break;
            }

            if (*dwOUTSize < (dwSizeMAC + dwPlainPadded))
            {
                fOK = false;
                break;
            }

            memcpy(abOUT, abPlainPadded, dwPlainPadded);
            memcpy(&abOUT[dwPlainPadded], m_abCMAC, dwSizeMAC);
            *dwOUTSize = dwSizeMAC + dwPlainPadded;

        } while (false);

        CryptDestroyKey(hDuplicateKey);
        delete[] abPlainPadded;
        return fOK;
    }
```

7.13 Wrap

```
bool CSecureChannel::WrapInput(CString strIN, CString* strOUT)
{
    bool fRet = true;
    BYTE *pbData1 = new BYTE[strIN.GetLength() / 2];
    DWORD dwSize1 = strIN.GetLength() / 2;
    BYTE *pbData2 = new BYTE[((DWORD)ceil((dwSize1 + 1.0) / 16.0) + 1 ) * 16];
    DWORD dwSize2 = ((DWORD)ceil((dwSize1 + 1.0) / 16.0) + 1 ) * 16;

    do{

        if ( !ConvertStringToHex(strIN, pbData1, &dwSize1) )
        {
            fRet = false;
            break;
        }
        if ( !WrapInput(pbData1, dwSize1, pbData2, &dwSize2) )
        {
            fRet = false;
            break;
        }
        if ( !ConvertHexToString(pbData2, dwSize2, strOUT) )
        {
            fRet = false;
            break;
        }
    } while (false);

    delete[] pbData1;
    delete[] pbData2;

    return fRet;
}
```

7.14 Unwrap 1

```
bool CSecureChannel::Unwrap(BYTE* abIN, DWORD dwINSize, BYTE* abOUT, DWORD
*dwOUTSize)
{
    bool  fOK          = true;
    BYTE  abComplementIV[16];
    DWORD dwEncrypted  = dwINSize - 16;
    BYTE  *abEncrypted = new BYTE[dwEncrypted];
    DWORD dwUnPadded   = dwINSize - 16;
    BYTE  *abUnPadded  = new BYTE[dwUnPadded];
    DWORD dwMACSize    = sizeof(m_abRMAC);
    HCRYPTKEY hDuplicateKey;

    CryptDuplicateKey(m_hSENC, 0, 0, &hDuplicateKey);

    do
    {
        if (dwINSize < 16)
        {
            fOK = false;
            break;
        }

        // compute and check the MAC
        if (!ComputeMAC(m_abCMAC, abIN, 0, dwINSize - 16, m_abRMAC,
&dwMACSize))
        {
            fOK = false;
            break;
        }
        if (memcmp(m_abRMAC, &abIN[dwINSize-16], dwMACSize) != 0)
        {
            fOK = false;
            break;
```

```
          }

          if (dwINSize > 16)
          {
                    for (int ii = 0; ii < 16; ++ii)
                    {
                              abComplementIV[ii] = ~m_abCMAC[ii];
                    }

                    memcpy(abEncrypted, abIN, dwINSize - 16);
                    dwEncrypted = dwINSize - 16;

                    if (!CryptSetKeyParam(hDuplicateKey, KP_IV, abComplementIV, 0))
                    {
                              fOK = false;
                              break;
                    }
                    if (!CryptDecrypt(hDuplicateKey, NULL, FALSE, 0, abEncrypted,
&dwEncrypted))

                    {
                              fOK = false;
                              break;
                    }

                    if (!UnPad(abEncrypted, dwEncrypted, abUnPadded, &dwUnPadded))
                    {
                              fOK = false;
                              break;
                    }

                    memcpy(abOUT, abUnPadded, dwUnPadded);
                    *dwOUTSize = dwUnPadded;
          }
          else
```

```
            {
                *dwOUTSize = 0;
            }

    } while (false);

    delete[] abUnPadded;
    delete[] abEncrypted;
    CryptDestroyKey(hDuplicateKey);
    if (!fOK)
            *dwOUTSize = 0;
    return fOK;
}
```

7.15 Unwrap 2

```
bool CSecureChannel::Unwrap(CString strIN, CString* strOUT)
{
    bool fRet = true;
    BYTE *pbData1;
//  pbData1 = (BYTE*)malloc(strIN.GetLength() / 2);
    pbData1 = new BYTE[(strIN.GetLength() / 2) + 1];//(strIN.GetLength() / 2) + 1
    BYTE *pbData2;
    pbData2 = new BYTE[(strIN.GetLength() / 2) + 1];
//  pbData2 = (BYTE*)malloc(strIN.GetLength() / 2);
    DWORD dwSize1;
    DWORD dwSize2;
//  memset(pbData1, 0x00, (strIN.GetLength() / 2) + 1);
    memset(pbData2, 0x00, (strIN.GetLength() / 2) + 1);

    do{
            dwSize1 = strIN.GetLength() / 2;
            if ( !ConvertStringToHex(strIN, pbData1, &dwSize1) )
            {
                fRet = false;
                break;
            }
            dwSize2 = strIN.GetLength() / 2;
            if ( !Unwrap(pbData1, dwSize1, pbData2, &dwSize2) )
            {
                fRet = false;
                break;
            }
            if ( !ConvertHexToString(pbData2, dwSize2, strOUT) )
            {
                fRet = false;
                break;
            }
        } while (false);

        delete[] pbData2;
        delete[] pbData1;

        return fRet;
    }
```

7.16 Covert String to Hex

```
bool CSecureChannel::ConvertStringToHex(CString strInput, BYTE *acOutput, DWORD
*dwLength)
{
        DWORD         dwLengthIn = *dwLength;
        int           iResult    = 0;
        DWORD         ii                   = 0;
        DWORD         bHex       = 0x00;

        for (ii = 0; ii < strInput.GetLength()/2; ++ii)
        {
                if ( ii < dwLengthIn )
                {
                        iResult = sscanf_s(strInput.Mid(ii * 2, 2), "%02x", &bHex);
                        (BYTE)acOutput[ii] = (BYTE)bHex;
                        if (iResult != 1)
                                return false;
                }
                else
                        return false;
        }
        *dwLength = ii;
        return true;
}
```

7.17 Convert Hex To String

```
bool CSecureChannel::ConvertHexToString(BYTE *acInput, DWORD dwLength, CString *strOutput)
{
    CString strTemp;

    *strOutput = "";

    for (DWORD ii = 0; ii < dwLength; ++ii)
    {
        strTemp = *strOutput;
        strOutput->Format("%s%02X", strTemp, (unsigned char)acInput[ii]);
    }
    return true;
}
```

8 Appendix Reader Configuration References

8.1 Reader Capabilities

Table 2 - Reader Capabilities Structure

Root	Branch
readerCapabilities [A0h]	deviceID [81h]
	productName [82h]
	productPlatform [83h]
	enabledCLFeatures [84h]
	frirmwareVersion [85h]
	sioProcessorVersion[86h]
	sdrVersion [87h]
	hfControllerVersion [88h]
	hardwareVersion [89h]
	hostInterfaces [8Ah]
	numberOfContactlessSlots [8Ch]
	humanInterfaces [AEh]
	vendorName [8Fh]
	sioProcessorFirmwareID [90h]
	exchangeLevel [91h]
	serialNumber [92h]
	hfControllerType [93h]

8.1.1 Device ID

Relative TLV:	A0 02 81 00
Access:	Read-only
Type:	OCTET STRING
Length:	2
Value:	00 01
Description:	Device Identifier
Get APDU	FF70076B 08 A206A004A0028100 00

8.1.2 Product Name

Relative TLV:	A0 02 82 00
Access:	Read-only
Type:	0 terminated String
Length:	13
Value:	“OMNIKEY 5326”
Description:	Name of product
Get APDU	FF70076B 08 A206A004A0028200 00

8.1.3 Product Platform

Relative TLV:	A0 02 83 00
Access:	Read-only
Type:	0 terminated String
Length:	8
Value:	“AviatoR”
Description:	Name of processor platform
Get APDU	FF70076B 08 A206A004A0028300 00

8.1.4 Enabled Contactless Features

Relative TLV:	A0 02 84 00
Access:	Read-only
Type:	OCTET STRING
Length:	2
Value:	0000 1000 0011 0000b
Description:	Flags for supported contactless features Bit 4 – SIO Processor available Bit 5 – LF Processor available (SDR) Bit 11 – PicoPass 15693-2 support available
Get APDU	FF70076B 08 A206A004A0028400 00

8.1.5 Firmware Version

Relative TLV:	A0 02 85 00
Access:	Read-only
Type:	OCTET STRING
Length:	3
Value:	XX YY ZZ
Description:	FwVersionMajor + FwVersionMinor + BuildNr
Get APDU	FF70076B 08 A206A004A0028500 00

8.1.6 SIO Processor Version

Relative TLV:	A0 02 86 00
Access:	Read-only
Type:	OCTET STRING
Length:	2
Value:	XX YY
Description:	Version number of SIO Processor
Get APDU	FF70076B 08 A206A004A0028600 00

8.1.7 SDR Version

Relative TLV:	A0 02 87 00
Access:	Read-only
Type:	OCTET STRING
Length:	6
Value:	XX XX XX XX XX 00
Description:	Version string of SDR (low frequency processor) NULL terminated string for example, “03.07”
Get APDU	FF70076B 08 A206A004A0028700 00

8.1.8 HF Controller Version

Relative TLV:	A0 02 88 00
Access:	Read-only
Type:	OCTET STRING
Length:	1
Value:	XX
Description:	CLRC663 version register
Get APDU	FF70076B 08 A206A004A0028800 00

8.1.9 Hardware Version

Relative TLV:	A0 02 89 00
Access:	Read-only
Type:	0 terminated string
Length:	variable
Value:	End Item Number ; Revision; Version Index
Description:	Hardware version string according AGILE: first part = End Item Number + Delimiter second part = Revision + Delimiter third part = Version Index
Get APDU	FF70076B 08 A206A004A0028900 00

8.1.10 Host Interface Flags

Relative TLV:	A0 02 8A 00
Access:	Read-only
Type:	OCTET STRING
Length:	1
Value:	0000 0010b
Description:	Flags for available Host interfaces Bit 1 – USB available
Get APDU	FF70076B 08 A206A004A0028A00 00

8.1.11 Number of Contact Slots

Relative TLV:	A0 02 8B 00
Access:	Read-only
Type:	UNSIGNED INTEGER 8
Length:	1
Value:	0
Description:	Number of available contact slots
Get APDU	FF70076B 08 A206A004A0028B00 00

8.1.12 Number of Contactless Slots

Relative TLV:	A0 02 8C 00
Access:	Read-only
Type:	UNSIGNED INTEGER 8
Length:	1
Value:	1
Description:	Number of available contactless slots
Get APDU	FF70076B 08 A206A004A0028C00 00

8.1.13 Number of Antennas

Relative TLV:	A0 02 8D 00
Access:	Read-only
Type:	UNSIGNED INTEGER 8
Length:	1
Value:	1
Description:	Number of available high frequency antennas
Get APDU	FF70076B 08 A206A004A0028D00 00

8.1.14 Human Interfaces Descriptor

Relative TLV:	A0 02 AE 00
Access:	Read-only
Type:	UNSIGNED INTEGER 8
Length:	variable
Value:	Constructed TLV structure
Description:	TLV Description of available human interfaces
Get APDU	FF70076B 08 A206A004A002AE00 00

8.1.15 Vendor Name

Relative TLV:	A0 02 8F 00
Access:	Read-only
Type:	0 TERMINATED STRING
Length:	11
Value:	“HID Global”
Description:	Vendor name
Get APDU	FF70076B 08 A206A004A0028F00 00

8.1.16 SIO Processor Firmware ID

Relative TLV:	A0 02 90 00
Access:	Read-only
Type:	OCTET STRING
Length:	6
Value:	XX XX XX XX XX XX
Description:	SIO processor firmware identifier
Get APDU	FF70076B 08 A206A004A0029000 00

8.1.17 Exchange Level Flags

Relative TLV:	A0 02 91 00
Access:	Read-only
Type:	OCTET STRING
Length:	1
Value:	0000 0001b
Description:	CCID exchange level flags: Bit 0 – Short APDU exchange level with CCID
Get APDU	FF70076B 08 A206A004A0029100 00

8.1.18 Serial Number

Relative TLV:	A0 02 92 00
Access:	Read-only
Type:	OCTET STRING
Length:	9
Value:	XX XX XX XX XX XX XX XX XX
Description:	Unique IFD serial number
Get APDU	FF70076B 08 A206A004A0029200 00

8.1.19 HF Controller Type

Relative TLV:	A0 02 93 00
Access:	Read-only
Type:	0 TERMINATED STRING
Length:	6
Value:	“RC663”
Description:	Type of integrated high frequency front end controller
Get APDU	FF70076B 08 A206A004A0029300 00

8.1.20 Size of User EEPROM

Relative TLV:	A0 02 94 00
Access:	Read-only
Type:	UNSIGNED INTEGER 16
Length:	2
Value:	04 00
Description:	Size of user EEPROM for free use
Get APDU	FF70076B 08 A206A004A0029400 00

8.2 Reader Configuration Control

Root	Branch
readerConfigurationControl [A9h]	applySetValues [80h]
	restoreFactoryDefaults [81h]

8.2.1 Apply Set Values

Relative TLV:	A9 02 80 00
Access:	Not accessible
Type:	COMMAND
Length:	0 byte
Description:	Apply the Configuration items for the runtime system
Get APDU	FF70076B 08 A206A104A9028000

8.2.2 Restore Factory Defaults

Relative TLV:	A9 02 81 00
Access:	Not accessible
Type:	COMMAND
Length:	0 byte
Description:	Restore the factory defaults. This means that any custom values will be lost.
Get APDU	FF70076B 08 A206A104A9028100

9 Appendix Definitions, Abbreviations and Symbols

AES	Advanced Encryption Standard
APDU	Application Protocol Data Unit
API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules
CLA	Class byte of an APDU
DER	Distinguished Encoding Rules
MAC	Message Authentication Code
MSDN	Microsoft® Developer Network
OID	Object Identifier
PAC	Physical Access Control
PACS	PAC Physical Access Control Services
PDU	Protocol Data Unit
PC/SC	Personal Computer/Smart Card
SIO	Secure Identity Object

10 Appendix References

[ISO 7816-4]	ISO 7816-4 Identification cards — Integrated circuit cards - Part 4: Organization, security and commands for Interchange Second edition - 2005-01-15
[ISO 8825]	ISO/IEC8825 ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER) Fourth edition 2008-12-15 or X.690 Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)
[ISO 9797-1]	ISO 9797-1 Information technology - Security techniques – Message Authentication Codes (MACs) Part 1: Mechanisms using a block cipher Second edition - 2011-03-01
[PCSC-3-Sup-CL]	Interoperability Specification for ICCs and Personal Computer Systems Part 3. Supplemental Document for Contactless ICCs Revision 2.02.00
[PCSC-3]	Interoperability Specification for ICCs and Personal Computer Systems Part 3. Requirements for PC-Connected Interface Devices Revision 2.01.09
[PCSC-3-Sup]	Interoperability Specification for ICCs and Personal Computer Systems Part 3. Supplemental Document Revision 2.01.08
[PCSC-3-AMD]	Interoperability Specification for ICCs and Personal Computer Systems Part 3. Requirements for PC-Connected Interface Devices - AMENDMENT 1 Revision 2.01.09

HID Global Headquarters:

North America: +1 949 732 2000

Toll Free: 1 800 237 7769
Europe, Middle East, Africa: +49 6123 791 0

Asia Pacific: +852 3160 9800

Latin America: +52 477 779 1492

support.hidglobal.com

hidglobal.com