

Smartcard reader in external TCP/IP device: Application Program Interface for Windows platforms

Table of Contents

	<u>Page</u>
1. General	2
2. Using the “ICCAPI”	3
3. Using the Windows PC/SC API	22
4. Using the CT-API	28

1. General

The built-in IC card (smart card) reader of the firewall/ADSL modem is accessed through a set of commands as described in a separate document ("Smartcard reader in external TCP/IP device... Protocol between host and reader"). The commands are sent by the TCP protocol, who takes care of the error checking and the lower physical levels of the link. Developers of application software should normally access the reader through a ready-made API, Application Program Interface, that takes care of and hides the stuff described in the mentioned document. The information below should be sufficient for most application programmers, along with other related documentation (for example the Windows PC/SC manuals). However, developers of drivers/API:s for new platforms must thoroughly study the protocol document mentioned.

This document describes the use of three API:s for the TCP/IP based card reader:

1. The proprietary, Intertex API "ICC-API"
2. The Windows PC/SC (Smart Card Service)
3. The CT-API

The API:s mentioned currently supports the Windows platforms 95/98/Me/NT4/2000/XP.

2. Using the “ICCAPI”

General

A straightforward way to write application programs for the card reader is to use the functions defined in a dll-file (Dynamic Link Library) called *iccapip.dll*. We call this interface “ICCAPI” (historically, the same API existed for the smartcard readers of a previous generation of PCMCIA and modem integrated products). The dll defines a set of C/C++ functions (other languages must emulate C/C++ calling conventions). The *iccapip.dll* library hides most of the message handling framework described in the protocol description, leaving only the core of telegrams to the programmer.

In the *iccapip.dll* there are C-compatible functions that do initialisation, data transmission and closing the IC card, all at a high level. The card activation, for example, does the following:

1. Sends an activation command to the smartcard reader.
2. Checks that the activation was successful and returns to the caller.

In addition to all these normal card operations, the *iccapip.dll* also has a set of functions giving full access to the message repertoire according to the protocol description for the IC card reader. For example, the display on the firewall/ADSL modem may be controlled by a C/C++ function call. The *iccapip.dll* is easy and straightforward to use but necessarily forces the application program to have a module specially written for this particular type of smartcard reader, it is not a vendor-independent API.

The library may be downloaded from www.intertex.se (or from a web-site linked by the smartcard reader configuration page in the product), packaged together with the CT-API dll file. This set of files also contains a readme file and utilities for using when setting up the card reader, and also for troubleshooting. The *iccapip.dll* may also be freely redistributed by a vendor of application programs, to facilitate program installation for the end user.

The functions in *iccapip.dll* are:

- ICC_Open_Channel
- ICC_Close_Channel
- ICC_Activate_Card
- ICC_Deactivate_Card
- ICC_Data_To_Card
- ICC_Data_From_Card
- ICC_Do_Card_Command
- ICC_Get_Channel_Handle
- ICC_Adjust_Timeouts
- ICC_HookUnhook_PCSC
- ICC_EscapeCommFunction

Normally, the application feeds and retrieves only the <data> field of the message format (ref. to the protocol description, see separate document) to/from the API. Apart from that, there are functions for opening/closing the ICC channel (the TCP connection) and for activating/deactivating the smart card. There are also three functions that enables a more direct communication with the smart card reader.

All functions returns a 32-bit signed integer. This is NULL if the function is indicating success, otherwise an error code is returned (it is *not* a Boolean TRUE/FALSE value) (ICC_HookUnhook_PCSC and ICC_EscapeCommFunction have another scheme). The functions, along with error codes, are described in a reference clause below.

A typical sequence of calls for a smart card session could look like:

```
ICC_HookUnhook_PCSC(0)                (if applicable)
ICC_Open_Channel(..)
ICC_Activate_Card(..)
ICC_Data_To_Card(..)
ICC_Data_From_Card(..)
.
.   ( a number of card transmissions )
.
ICC_Data_To_Card(..)
ICC_Data_From_Card(..)
ICC_Deactivate_Card(..)
.
.   ( maybe other cards processed )
.
ICC_Close_Channel(..)
ICC_HookUnhook_PCSC(1)                (if applicable)
```

Using the *iccapip.dll* function library

As other DLL:s (Dynamic Link Libraries), there are two ways of linking to the *iccapip.dll*: load-time dynamic linking and run-time dynamic linking.

In load-time dynamic linking, the application program can reference a library function just like a normal, external function. When the application program is loaded, it finds the *iccapip.dll* and attaches that to its adress space, making the functions directly callable. In order to solve the references, the application must be linked with a library file, *iccapip.lib*. This file is delivered with the file package, the lib-file is compatible with Microsoft Visual Studio format.

It is practical to insert the external declarations in an include (.h) file as follows:

```
extern int ICC_Open_Channel(int);
extern int ICC_Close_Channel();
extern int ICC_Activate_Card(int, unsigned char*, int*);
extern int ICC_Deactivate_Card();
extern int ICC_Data_To_Card(unsigned char*, int,
                           unsigned char*, int*, int);
extern int ICC_Data_From_Card(unsigned char*, int,
                              unsigned char*, int*);
extern int ICC_Do_Card_Command(unsigned char*, int,
                              unsigned char*, int*);
extern int ICC_Get_Channel_Handle(SOCKET*);
extern int ICC_Adjust_Timeouts(int);
extern int ICC_HookUnhook_PCSC(int);
extern BOOL ICC_EscapeCommFunction(int);
```

Using these declarations, and specifying *iccapip.lib* on the linker command line, enables the application program to call the API functions directly.

The other way of linking to *iccapip.dll* is run-time dynamic linking. In this case, when calling the linker for the application program, the *iccapip.lib* is not needed (and shall not be used). Instead, the application code must explicitly load the *iccapip.dll* by a LoadLibrary (Microsoft Win32 API) call. Furthermore, all addresses to the functions needed must be resolved by explicit calls of the GetProcAddress (Microsoft Win32 API). This may be done once for all when the application starts. The addresses can be stored on 32-bit static or global variables called by the same names as the function. Having done this, all functions may be conveniently called under their original names. This is illustrated in the following example:

```
// Include this in a .h-file:
extern BOOL SetupICCFunctions();

typedef int (*ICCPROC)();
typedef int (*ICCPROCint)(int);
typedef int (*ICCPROChand)(SOCKET*);
typedef int (*ICCPROCintbuf)(int, unsigned char*, int*);
typedef int (*ICCPROCbufbuf)(unsigned char*, int,
                             unsigned char*, int*);
typedef int (*ICCPROCbufbufint)(unsigned char*, int,
                                unsigned char*, int*, int);
typedef BOOL (*ICCPROCbool)(int);

extern HINSTANCE          ICCLib;
    // These are instead of normal function declarations:
extern ICCPROCint        ICC_Open_Channel;
extern ICCPROC           ICC_Close_Channel;
extern ICCPROCintbuf     ICC_Activate_Card;
extern ICCPROC           ICC_Deactivate_Card;
extern ICCPROCbufbufint  ICC_Data_To_Card;
extern ICCPROCbufbuf     ICC_Data_From_Card;
extern ICCPROCbufbuf     ICC_Do_Card_Command;
extern ICCPROChand       ICC_Get_Channel_Handle;
extern ICCPROCint        ICC_Adjust_Timeouts;
extern ICCPROCint        ICC_HookUnhook_PCSC;
extern ICCPROCbool       ICC_EscapeCommFunction;

// Include this in the application code:

HINSTANCE ICCLib;          // The handle to the library

// Global variables containing the function addresses
ICCPROCint        ICC_Open_Channel;
ICCPROC           ICC_Close_Channel;
ICCPROCintbuf     ICC_Activate_Card;
ICCPROC           ICC_Deactivate_Card;
ICCPROCbufbufint  ICC_Data_To_Card;
ICCPROCbufbuf     ICC_Data_From_Card;
ICCPROCbufbuf     ICC_Do_Card_Command;
ICCPROChand       ICC_Get_Channel_Handle;
ICCPROCint        ICC_Adjust_Timeouts;
ICCPROCint        ICC_HookUnhook_PCSC;
ICCPROCbool       ICC_EscapeCommFunction;

// A function for setting up all function addresses:
```

```

BOOL SetupICCFunctions()
{
    ICCLib = LoadLibrary("iccapip"); // Get the library

    if(ICCLib == NULL) return(FALSE);
    // Get the addresses to the functions and
    // store in variables
    ICC_Open_Channel = (ICCPROCint) GetProcAddress(
        ICCLib, "ICC_Open_Channel");
    ICC_Close_Channel = (ICCPROC) GetProcAddress(
        ICCLib, "ICC_Close_Channel");
    ICC_Activate_Card = (ICCPROCintbuf) GetProcAddress(
        ICCLib, "ICC_Activate_Card");
    ICC_Deactivate_Card = (ICCPROC) GetProcAddress(
        ICCLib, "ICC_Deactivate_Card");
    ICC_Data_To_Card = (ICCPROCbufbufint) GetProcAddress(
        ICCLib, "ICC_Data_To_Card");
    ICC_Data_From_Card = (ICCPROCbufbuf) GetProcAddress(
        ICCLib, "ICC_Data_From_Card");
    ICC_Do_Card_Command = (ICCPROCbufbuf) GetProcAddress(
        ICCLib, "ICC_Do_Card_Command");
    ICC_Get_Channel_Handle = (ICCPROChand) GetProcAddress(
        ICCLib, "ICC_Get_Channel_Handle");
    ICC_Adjust_Timeouts = (ICCPROCint) GetProcAddress(
        ICCLib, "ICC_Adjust_Timeouts");
    ICC_HookUnhook_PCSC = (ICCPROCint) GetProcAddress(
        ICCLib, "ICC_HookUnhook_PCSC");
    ICC_EscapeCommFunction = (ICCPROCbool) GetProcAddress(
        ICCLib, "ICC_EscapeCommFunction");

    // Check that all addresses are resolved
    if((ICC_Open_Channel==0) ||
        (ICC_Close_Channel==0) ||
        (ICC_Activate_Card==0) ||
        (ICC_Deactivate_Card==0) ||
        (ICC_Data_To_Card==0) ||
        (ICC_Data_From_Card==0) ||
        (ICC_Do_Card_Command==0) ||
        (ICC_Get_Channel_Handle==0) ||
        (ICC_HookUnhook_PCSC==0) ||
        (ICC_EscapeCommFunction==0) ||
        (ICC_Adjust_Timeouts==0)) return(FALSE);

    return(TRUE);
}

. (code)
.
.
// Do this at startup
if(SetupICCFunctions() == FALSE)
{
    // Report the error. Using the API-functions will not work
}
.
.
.
// Sample of a call: Open the TCP connection
Res = ICC_HookUnhook_PCSC(0);

```

```
Res = ICC_Open_Channel(1);
if(Res)
{
    // Display value of Res (=error code)
}
else
{
    // Successful, channel is opened
}
}
```

The *iccapip.dll* file must be in the Windows directory, Windows system directory, in the current or load directory, in a PATHed directory or may be in a directory explicitly specified in the LoadLibrary function. (See Microsoft documentation of Dynamic Link Libraries.)

Reference

On the following pages each function of the ICCAPI is described. The data types used are int, unsigned char and SOCKET. The int is a 32-bit signed integer. The SOCKET is a 32-bit unsigned integer and is typedef:ed to a handle of a Windows socket for TCP/IP connections. The DWORD is a 32-bit unsigned integer.

ICC_Open_Channel

Opens the TCP channel for smart card and reader accesses.

```
int ICC_Open_Channel(  
    int ChannelNumber  
);
```

Parameters

ChannelNumber

A value 1-4. Currently, there are no support for having more than one channel opened at a time, so the channel number is of no significance.

Return Value

NULL if success, otherwise an error code. Typical causes of open failures is that the reader is not properly connected on the (local) network. See the readme-file that comes with the CT-API/ICCAPI set of files, under “Troubleshooting”.

ICC_Close_Channel

Closes the TCP channel and disables smart card and reader accesses.

```
int ICC_Close_Channel( );
```

No parameters

Return Value

Always NULL, whether the channel was open or not.

ICC_Activate_Card

Powers up the smart card and performs the ATR (Answer-to-Reset) procedure.

```
int ICC_Activate_Card(  
    int TValue,  
    unsigned char *HistBytes,  
    int *SizeFrom  
);
```

Parameters

TValue

The T-value for the card protocol according to ISO 7816-3. At the moment, only T=0 and T=1 have protocol support in the card reader. T=99 is a special case, see below.

HistBytes

A pointer to an area where to put the "historical bytes", resulting from the card activation and according to ISO 7816-3.

SizeFrom

A pointer to an integer where to put the number of historical bytes. Upon function call, **SizeFrom* shall contain the maximum number of bytes, so the buffer will not be overflowed.

Return Value

NULL if success, otherwise an error code. Error codes in the range 128-255 result from the card reader and may indicate that the card is not installed, unknown card type or other problems (see the protocol description document for explanations). Error codes above the value 1000 may result from message transfer problems, such as TCP transfers timeouts, bad parameters or corrupted data.

Comments

If the card is already activated, the historical bytes are received and the function will return successfully.

Specifying T=99 makes the activation choose any protocol that the card uses, giving priority to T=0. This feature is convenient when the card protocol is not known in advance, it makes use of the "command 25" in the protocol description. In this case, the resulting protocol will be returned as 1:st byte in *HistBytes*, leaving the historical bytes to start at *HistBytes+1*. The *SizeFrom* will of course be one more than the actual number of historical bytes.

The full ATR-string may also be retrieved, but this has to be done by using the *ICC_Do_Card_Command*, specifying command byte=1 (see card reader protocol description).

ICC_Deactivate_Card

Powers down the smart card and deactivates the card reader contacts.

```
int ICC_Deactivate_Card();
```

No parameters

Return Value

NULL if success, otherwise an error code. If the card was already deactivated, NULL will be returned nevertheless. Other errors than message transfer problems are unlikely.

Comments

If the card is pulled out when activated, the card reader firmware will immediately switch off (deactivate) the card contacts. The next attempt to read/write to/from the card will result in error code 128, even if the same card has been inserted again. It then has to be activated again.

ICC_Data_To_Card

Transmits data to the smart card.

```
int ICC_Data_To_Card(  
    unsigned char *DataToCard,  
    int SizeTo,  
    unsigned char *ResponseFromCard,  
    int *SizeFrom,  
    int PINOffset  
);
```

Parameters

DataToCard

A pointer to the data to be sent to the card. It includes only the <data> field of the message format (see card reader protocol description), no command or parameter byte. According to ISO7816-4 terminology, this is a TPDU, not APDU. Thus, the "case 4" and the T=1 protocol have to be treated by more than one call to ICC_Data_To_Card/ICC_Data_From_Card.

SizeTo

Number of bytes to send (in the buffer pointed to by *DataToCard*).

ResponseFromCard

A pointer to an area where to put the answer from the card. For T=0 (ISO 7816-3), the answer is typically the two bytes SW1/SW2.

SizeFrom

A pointer to an integer where to put the number of bytes in the response. Upon function call, **SizeFrom* shall contain the maximum number of bytes, so the buffer will not be overflowed.

PINOffset

Not used and ignored, kept for compatibility. should be NULL otherwise.

Return Value

NULL if success, otherwise an error code. See under "ICC_Activate_Card". There may be useful data in the *ResponseFromCard* even in case of error code, so check the **SizeFrom* value. Specifically, error codes 135 and 141 means unexpected SW1/SW2 values (for T=0, ISO 7816-3) which then can be found in the *ResponseFromCard* buffer.

Comments

The procedure for writing data to the card is very similar to reading data from the card, according to ISO 7816-3. In fact, for T=1 protocol, they are exactly the same and the application can therefore use either ICC_Data_To_Card and ICC_Data_From_Card in any read/write situation. For T=0, these are not totally interchangeable: the length byte P3 (the 5:th data byte) has different functions depending on data direction (a function of the

instruction), especially if P3=0. See the ISO 7816-3 standard. Hence the need for both ICC_Data_To_Card and ICC_Data_From_Card.

Example of use of ICC_Data_To_Card (selecting a file on a Philips DX card, T=0):

```
#define  BUFSIZE 10

int err,SizeI;
unsigned char SelFile1[] ={0xa0,0xa4,0,0,2,0,0x20};
unsigned char inbuffer[BUFSIZE];

SizeI = BUFSIZE;          // Don't forget this!

err = ICC_Data_To_Card(SelFile1,7,inbuffer,&SizeI,0);
if(err)
{
    if( ((err == 135)|| (err == 141)) && (SizeI == 2) )
    {
        // Investigate inbuffer[0] and inbuffer[1],
        // they are SW1/SW2, might specify the error
    }
    else
    {
        // Other error, report it.
    }
}
else
{
    // Write (=file selection) successful
}
```

ICC_Data_From_Card

Fetches data from the smart card.

```
int ICC_Data_From_Card(  
    unsigned char *DataToCard,  
    int SizeTo,  
    unsigned char *ResponseFromCard,  
    int *SizeFrom,  
);
```

Parameters

DataToCard

A pointer to the data (telegram) to be sent to the card. It includes only the <data> field of the message format (see card reader protocol description), no command or parameter byte. According to ISO7816-4 terminology, this is a TPDU, not APDU. Thus, the "case 4" and the T=1 protocol have to be treated by more than one call to ICC_Data_To_Card/ICC_Data_From_Card.

SizeTo

Number of bytes to send (in the buffer pointed to by *DataToCard*).

ResponseFromCard

A pointer to an area where to put the data from the card.

SizeFrom

A pointer to an integer where to put the number of bytes in the response. Upon function call, **SizeFrom* shall contain the maximum number of bytes, so the buffer will not be overflowed.

Return Value

NULL if success, otherwise an error code. See under "ICC_Activate_Card". There may be useful data in the *ResponseFromCard* even in case of error code, so check the **SizeFrom* value. Specifically, error codes 135 and 141 means unexpected SW1/SW2 values (for T=0, ISO 7816-3) which then can be found in the *ResponseFromCard* buffer.

Comments

T=0: If the response is successful (or error code 135), the *ResponseFromCard* buffer contains the data, followed by the two bytes SW1/SW2 last in the buffer.

T=1: If successful, the *ResponseFromCard* buffer simply contains the prologue, information and epilogue fields of the response.

See also comments under ICC_Data_To_Card for the discussion of the data direction.

ICC_Do_Card_Command

Send a message to the card/card reader, receive an answer from the card/card reader.

```
int ICC_Do_Card_Command(  
    unsigned char *MsgToReader,  
    int SizeTo,  
    unsigned char *ResponseFromReader,  
    int *SizeFrom,  
);
```

Parameters

MsgToReader

A pointer to the message to be sent to the card reader. It includes the <command>, the <parameter> and the <data> fields of the message format (see card reader protocol description).

SizeTo

Number of bytes to send (in the buffer pointed to by *MsgToReader*).

ResponseFromReader

A pointer to an area where to put the response from the reader. The response includes the <command>, the <parameter> and the <data> fields of the message format (see card reader protocol description).

SizeFrom

A pointer to an integer where to put the number of bytes in the response. Upon function call, **SizeFrom* shall contain the maximum number of bytes, so the buffer will not be overflowed.

Return Value

NULL if success, otherwise an error code. See under "ICC_Activate_Card".

Comments

This function enables the application programmer to perform all the card reader functions as specified in the protocol description. Controlling the display, getting card insertion state etc. are examples of operations that are performed by this function. See the protocol description for a list of possible operations. In this direct mode, the command byte, the parameter byte and the data field all have to be setup in the output buffer. Likewise, the response contains the full message from the card reader, including the command byte, the parameter byte and the data field.

The use of `ICC_Do_Card_Command` is illustrated by the following example:

```
#define  BUFSIZE 10

int err,SizeI;
unsigned char GetState[]={3, 0}; // Get card status

unsigned char inbuffer[BUFSIZE];

SizeI = BUFSIZE;           // Don't forget this!

err = ICC_Do_Card_Command(GetState,2,inbuffer,&SizeI);
if(err)
{
    // Report error
}
else
{
    // Get state successful
    if ( (inbuffer[1] == 1) || (inbuffer[1] == 2) )
    {
        // Card present
    }
    else
    {
        // Card absent
    }
}
}
```

ICC_Get_Channel_Handle

Retrieve the socket handle to the TCP connection.

```
int ICC_Get_Channel_Handle(  
    SOCKET *ChanHandle  
);
```

Parameters

ChanHandle

A pointer to where to put the retrieved 32-bit handle. This handle can then be used in subsequent Win32 socket function calls (such as send(), recv(), select() etc.) directly to the TCP socket.

Return Value

NULL if the TCP channel is open, otherwise the error code ICC_ERR_NOTOPEN (in which case the handle retrieved is INVALID_SOCKET, a Win32 constant).

Comments

This function could be used if the application programmer wants to operate the TCP socket in a way not supported by the smart card API. By retrieving the SOCKET handle, the full Win32 Winsock API may be used.

IMPORTANT: Unlike typical Windows manners, the handle retrieved by ICC_Get_Channel_Handle shall NOT be closed by a CloseHandle call. The closing of the handle is done by the ICC_Close_Channel library routine. The application does not own the handle, it simply "borrows" it from the *iccapip.dll* library.

ICC_Adjust_Timeouts

Changes the receive timeout used by the specific smart card API.

```
int ICC_Adjust_Timeouts(  
    int TimeBias  
);
```

Parameters

TimeBias

A value in milliseconds specifying the time to be added to or (if negative) subtracted from the default receive timeout values otherwise used by the smart card API.

Return Value

Always returns NULL.

Comments

The functions in the *iccapip.dll* use reasonable timeout values when communicating with the smart card reader. For example, *ICC_Data_From_Card* and *ICC_Data_To_Card* use 15 seconds read timeout, most other functions in ICCAPI have 4 seconds timeout. Should the application programmer find that the timeouts are too short in any situation, the time may be increased with this function.

Only the receive timeouts are affected by this function.

A practical approach is to use this function only if the application runs into any trouble due to lengthy operations. Lowering the timeout by calling the function with a negative value should be used with great care, it does not improve performance of the card operations in general.

By calling the function with a NULL parameter, the timeouts are reset to the default values.

The timeouts used in the transmission to the smartcard (such as the ISO7816-3, clause 8.2 “work waiting time”) are taken care of by the card reader internally and need not to be bothered about.

ICC_HookUnhook_PCSC

Hooks or unhooks the smartcard reader from the PC/SC system.

```
int ICC_HookUnhook_PCSC(  
    int Function  
);
```

Parameters

Function

A function code specifying what to do:

- | | |
|---|---------------------------------------------------------------------------------------------------------------------------------------|
| 0 | Unhook the reader from the Windows PC/SC and free it for use by <i>iccapip.dll</i> . (The function takes 400 ms if PC/SC is running). |
| 1 | Hook the reader back again to the Windows PC/SC system. |

Return Value

Regardless of the *Function* code the following is returned:

- | | |
|---|----------------------------------------------------------------------------------------------------|
| 0 | The smartcard reader is not connected to PC/SC, or the PC/SC system is not running on the machine. |
| 1 | PC/SC is running and uses the reader. |

Other value: PC/SC is running but the function (hook or unhook) failed.

Comments

As described later in this document, the PC/SC subsystem for Windows gains access to the smartcard reader right from boot-time. This does not necessarily prevent non-PC/SC-aware applications from accessing the reader, but the PC/SC-session will have problem reconnecting to the reader again after such an “intrusion”. To be able to take up a PC/SC session again after an ICCAPI session, it is recommended for the (ICCAPI-) application to free the reader from the PC/SC ownership, and to give it back to PC/SC afterwards. This is done by the ICC_HookUnhook_PCSC routine. As an application programmer never can predict if the end-user would have PC/SC running on his/hers machine (for other applications) it is wise to always make use of the ICC_HookUnhook_PCSC routine. Call ICC_HookUnhook_PCSC at the beginning of the application, with *Function*=0, before the first ICC_Open_Channel. Similarly, call ICC_HookUnhook_PCSC, with *Function*=1, before exit and after the last ICC_Close_Channel call. It is also possible to open up for PC/SC in moments of no activity, provided that any card session is finished. In all cases, it is essential to use ICC_HookUnhook_PCSC in pairs (one ‘hook’ call for each ‘unhook’ call).

It is safe to call ICC_HookUnhook_PCSC even if no PC/SC has been installed on the machine.

ICC_EscapeCommFunction

Sends a specific code to the *iccapip.dll* library, performing various functions or enquiries.

```
BOOL ICC_EscapeCommFunction(  
    DWORD Escapecode  
);
```

Parameters

Escapecode

Any of following values:

- | | |
|-----|-------------------------------------------------------------------------------------|
| 503 | Test if card state (inserted/removed) has changed since last call of this function. |
| 507 | Test if smartcard channel has been opened, by ICC_Open_Channel. |

Return Value

A non-zero value for TRUE, zero for FALSE.

Comments

The syntax of this function is, for compatibility with the serial smartcard readers, much like the standard serial Win32 API **EscapeCommFunction** (for the serial card readers there were more escape-codes).

Using ICC_EscapeCommFunction(503) is a convenient way to get the API check if any state change message have arrived to the TCP receive buffer.

Error codes returned by the specific smart card API functions

Value	Symbolic name	Description
128 - 255	-	Error codes from card/card reader. See card reader protocol description.
1000	ICC_ERR_OPENFAIL	Could not open the ICC-channel. The ICC-channel, or the physical COM-port associated with it, may be used by another application or is not properly installed.
1002	ICC_ERR_TIMOSETUPFAIL	Error when setting up the timeout values of the virtual COM-port.
1004	ICC_ERR_MSGFAIL	Error when sending or receiving the binary card message. Most probably a timeout has happened.
1005	ICC_ERR_MSGCORRUPT	The checksum of the received message indicates disturbed data, or the number of received bytes is wrong.
1006	ICC_ERR_BUFTOOSMALL	The response from the card/card reader does not fit into the application's buffer. No characters are copied into it. The card operation may nevertheless have been successfully completed.
1007	ICC_ERR_NOTOPEN	The ICC channel is not open.

3. Using the Windows PC/SC API

PC/SC is the name of a standard that Microsoft Windows and other operating systems may use when communicating with smartcards. In Windows, it is sometimes referred to as “Microsoft Smart Card Base Components”, “Smart Card Service” or “Smart Card Subsystem”. Under PC/SC, applications could be written in a standardized way to support different types of card readers. Thus, a new card reader may be supported by an application program, long after that program was coded.

For Windows 2000 and XP, the PC/SC is built into the operating system, known as the “Smart Card Service” in the Control Panel (under “Administrative Tools”, “Services”). For Windows 95/98/Me/NT4 it has to be installed by running an installation program called *scbase.exe*, distributed by Microsoft. For convenience, it is also redistributed along with the drivers for this particular smartcard reader. For information on how to write application programs for the PC/SC, visit the Microsoft web-site and look for the Platform SDK documentation (currently, as it appears: msdn.microsoft.com/library/, expand “Windows Development”, “Platform SDK”, “Contents of the Platform SDK” and click “Smart Card”).

For Windows PC/SC to work, every card reader must support its own device driver, to make up the interface between the PC/SC ‘resource handler’ and the particular smartcard reader. Here, the driver file is named *ixpcscip.sys* (for Windows 95/98/Me: *ixpcscip.vxd*). This chapter describes how to use the functions in *ixpcscip.sys* (we will only use the *.sys*-name below, *ixpcscip.vxd* functions the same). For application developers, the general information in the Microsoft Platform SDK should normally be sufficient for using the *ixpcscip.sys*. However, some vendor-specific functions, described below, may be worthwhile to study. There are also some consequences of the PC/SC service on the TCP channel ownership that should be considered, they are described below.

General

The PC/SC support for this card reader has to be installed by a procedure involving copying the driver file and writing to the Windows registry.

The Windows PC/SC has been designed to control the smartcard reader right from the startup of the system. Thus, the firewall/ADSL modem has to be switched on when Windows starts, otherwise the smartcard programs will not work. The PC/SC monitor of Windows wants to find out the status of the card reader from the beginning and will also establish contact with the smartcard if there is one inserted.

Consequently, the PC/SC driver *ixpcscip.sys* opens a TCP connection to the reader right from the beginning and keeps it opened (this is an important difference from ICCAPI, where the TCP connection is opened/closed only by commands from the smartcard application). Only one TCP channel to the reader can be open at a time, so if an application using a non-PC/SC API wants to run, it must temporarily disable the PC/SC’s control of the reader. The application must later free the reader again after use, and should also try to reconnect the reader to the PC/SC system again, as has been discussed under *ICC_HookUnhook_PCSC* for the ICCAPI above.

The PC/SC smartcard driver *ixpcscip.sys* consists of support for “IOCTL” codes that are sent from the Windows Smartcard Resource Handler. These IOCTL functions do things like power on/off the card, getting the ATR string, enquiring the state of the card

(present/not present) and transmitting data to/from the card. A smartcard device driver standard library (a Microsoft product) is helping the driver to perform its functions. For example, the complex T=1 protocol overhead is entirely handled by these components and needs not to be bothered about in the application.

IOCTL codes

The IOCTL functions that are defined in the PC/SC standard will not be described here. We refer to the Microsoft Smartcard DDK (they are normally not of interest to the application developer). Below are a few specific IOCTL:s that the application may want to use, in order to reach functions in the smartcard reader that are not part of the standard PC/SC API. The recommended way to issue these function calls is to use the SCardControl function in the resource handle. The user is asked to study the Microsoft Platform SDK documentation, there is a web page for each interface function, such as the SCardControl.

The recommended way to send a specific IOCTL code to *ixpccip.sys* is by first obtaining a SCARDHANDLE to the reader (using SCardConnect, DIRECT mode if there is not a card in the reader). If a card connection is already established this is of course not necessary. Then the IOCTL code may be issued using the SCardControl service. This passes the supplied IOCTL code directly to the reader driver without interpretation.

Reference

On the following pages each specific IOCTL function of the *ixpccip.sys* smart card API is described. The selected specific code should be setup in the dwControlCode parameter of the SCardControl function (see Microsoft documentation). Likewise, the parameters lpInBuffer, nInBufferSize, lpOutBuffer, nOutBufferSize and lpBytesReturned on the following pages all refer to parameters of the SCardControl function call.

IX_IOCTL_DO_CARD_COMMAND (Value 0x312700)

Send a message to the card/card reader, receive an answer from the card/card reader.

Parameters

lpInBuffer

A pointer to the message to be sent to the card reader. It includes the <command>, the <parameter> and the <data> fields of the message format (see card reader protocol description).

nInBufferSize

Number of bytes to send (in the buffer pointed to by *lpInBuffer*).

lpOutBuffer

A pointer to an area where to put the response from the reader. The response includes the <command>, the <parameter> and the <data> fields of the message format (see card reader protocol description).

nOutBufferSize

Max. number of bytes to receive (in the buffer pointed to by *lpInBuffer*).

lpBytesReturned

A pointer to a DWORD (32bits unsigned integer) where to put the number of bytes in the response.

Return Value

Value SCARD_S_SUCCESS (=0) as defined in the Microsoft header file scarderr.h. Else an error code defined in that header file.

Comments

This function enables the application programmer to perform all the card reader functions as specified in the protocol description. Controlling the display, getting card insertion state etc. are examples of operations that are performed by this function. See the protocol description for a list of possible operations. In this direct mode, the command byte, the parameter byte and the data field all have to be setup in the output buffer. Likewise, the response contains the full message from the card reader, including the command byte, the parameter byte and the data field.

The user is reminded that most writes/reads to the smartcard itself may be done by the ordinary PC/SC card transmit functions, not through this IOCTL. This IOCTL function is a way to reach specific functions in the reader, not covered by the PC/SC definition.

IX_IOCTL_ADJ_TIMEOUTS (Value 0x312710)

Changes the receive timeout used by the *ixpcscip.sys* driver.

Parameters

lpInBuffer

A pointer to an *int* (32bits signed integer) containing a value in milliseconds specifying the time to be added to or (if negative) subtracted from the default receive timeout values otherwise used by the *ixpcscip.sys*.

nInBufferSize

Should be 4 (but is not tested).

lpOutBuffer

Not used.

nOutBufferSize

Not used.

lpBytesReturned

Not used (ScardControl wants a non-NULL pointer)

Return Value

Always the value SCARD_S_SUCCESS (=0) as defined in the Microsoft header file scarderr.h.

Comments

The functions in the *ixpcscip.sys* use reasonable timeout values when communicating with the smart card reader. Should the application programmer nevertheless find that the timeouts are too short in any situation, the time may be increased with this function.

Only the receive timeouts are affected by this function.

A practical approach is to use this function only if the application runs into any trouble due to lengthy operations. The default receive timeout, typically 15 seconds, should be enough. Lowering the timeout by calling the function with a negative value should be used with great care, it does not improve performance of the card operations in general.

By calling the function with a zeroed value, the timeouts are reset to the default values.

The timeouts used in the transmission to the smartcard are taken care of by the card reader internally and need not to be bothered about.

IX_IOCTL_PCSC_CLOSE (Value 0x312724)

Close the PC/SC's access to the card reader.

Parameters

lpInBuffer

Not used.

nInBufferSize

Not used.

lpOutBuffer

A pointer to a byte where to put a channel number, currently always 1 (mostly for compatibility with previous, serial readers where a COM-port number was returned).

nOutBufferSize

Should be 1 (or greater).

lpBytesReturned

A pointer to a DWORD (32bits unsigned integer) where to put the number of bytes (=1) in the response.

Return Value

Always SCARD_S_SUCCESS (=0) as defined in the Microsoft header file scarderr.h.

Comments

This function makes it possible for the application programmer to force the PC/SC driver close the TCP connection to the smartcard reader, freeing it for use by applications not using PC/SC at all. This command rarely has to be used, mostly it is up to the non-PC/SC application to disable the PC/SC (see "ICC_HookUnhook_PCSC" under the ICCAPI chapter above). During the disabled phase, the PC/SC will still have contact with the driver *ixpcscip.sys*, but it will not get any information from the card, nor will it know whether the card is inserted or not. As soon as the PC/SC is enabled again (through the IX_IOCTL_PCSC_REOPEN code) the PC/SC system will be informed about the new state of the card.

IX_IOCTL_PCSC_REOPEN (Value 0x312728)

Open the PC/SC's access to the card reader, after having been closed by the IX_IOCTL_PCSC_CLOSE command.

Parameters

lpInBuffer

Not used.

nInBufferSize

Not used.

lpOutBuffer

A pointer to a byte where to put a channel number, currently always 1 (mostly for compatibility with previous, serial readers where a COM-port number was returned).

nOutBufferSize

Should be 1 (or greater).

lpBytesReturned

A pointer to a DWORD (32bits unsigned integer) where to put the number of bytes (=1) in the response.

Return Value

Always SCARD_S_SUCCESS (=0) as defined in the Microsoft header file scarderr.h.

Comments

This function enables the application programmer to make the PC/SC driver open the TCP connection again after having been closed by the IX_IOCTL_PCSC_CLOSE function. There is no real feedback that PC/SC really has regained contact with the reader.

4. Using the CT-API

General

The German specification CT-API for a standard application independent API is implemented by a library *ixctapip.dll*. The standard consists mainly of three standardized function calls, the CT_init, CT_data and CT_close. The CT-API specification mainly describes the function calls for opening ports and sending card commands. The commands used for controlling the card reader is specified in an additional standard called CT-BCS, used in the German healthcare and elsewhere. The CT-API and CT-BCS together form an easy-to-use interface for the application programmer.

The specifications may be downloaded from the following web-addresses:

<http://www.darmstadt.gmd.de/~eckstein/CT/mkt.html>

<http://www.linuxnet.com/documentation/files/ctapi.html>

<http://www.linuxnet.com/documentation/files/ctbcs.html>

The CT-API works with ISO7816-4 APDU:s (application protocol data units). The cases 1,2,3 and 4 are supported (in short versions). The protocols supported are T=0 and T=1. The application sends interindustry commands and need not bother about any protocol overhead, for example the T=1 prologue and epilogue.

As for the CT-BCS commands, following commands are supported:

<u>INS-code (hex)</u>	<u>Name</u>
10	RESET (compatible with the B1 reader)
11	RESET CT
12	REQUEST ICC
13	GET STATUS
14	DEACTIVATE ICC (compatible with the B1 reader)
15	EJECT ICC

The display on the firewall/ADSL modem (with card reader) is reached only through the propriety function IX_CTAPI_Do_Card_Command (see below).

If the Windows PC/SC (see previous chapter) was running at the time of the CT_init() call, it will be turned off and kept off until CT_close() is called, so it will not mess with the CT-API card reader session. More precisely, ICC_HookUnhook_PCSC(0) is called automatically at CT_init(), and ICC_HookUnhook_PCSC(1) is called automatically at CT_close (see the ICCAPI chapter for an explanation).

dll:s

The CT-API for this reader is implemented by a 32-bit dynamic library file called *ixctapip.dll*, which (as any dll) must reside in the current directory, in the Windows directory or in a pathed directory. It makes use of the specific library *iccapip.dll*, which also

must be present in one of the directories mentioned. The CT-API interface is thus levelled above the specific interface ICCAPI, described in a previous chapter.

The CT-API also specifies "well known identifiers" for the function calls. At the moment, this is not supported by *ixctapip.dll* (contact Intertex if needed).

All functions mentioned are in the FORTRAN (or pascal) calling convention, in Microsoft's notation declared with the keyword `__stdcall` (the called function does the cleaning of the parameters on the stack).

The vendor of the application may freely redistribute the necessary dll:s. They are distributed on the web-site for the product's on-line manual, and also on www.intertex.se, in a set of files labelled "CT-API/ICCAPI interface".

The *ixctapip.dll* may be linked to, either using load-time dynamic linking or run-time dynamic linking, as was described for the *iccapip.dll* (see the ICCAPI chapter). The choice mainly affects the behaviour when the wanted dll is missing on the end-users machine: A run-time linking application may be started without the necessary dll (the programmer may decide what actions to take when the necessary dll is missing).

For load-time linking, it is practical to include the function prototypes in a .h-file as follows:

```
extern char __stdcall CT_init(WORD ctn,
                              WORD pn);
extern char __stdcall CT_data(WORD ctn,
                              BYTE *dad,
                              BYTE *sad,
                              WORD lenc,
                              BYTE *command,
                              WORD *lenr,
                              BYTE *response);
extern char __stdcall CT_close(WORD ctn);
```

Using these declarations, and specifying *ixctapip.lib* on the linker command line, enables the application program to call the API functions directly.

For run-time dynamic linking, the application must explicitly load the dll-file and resolve the addresses to the functions. Instead of the declarations above, some typedef's should be used to declare the types of the function pointers, as follows:

```
// Include this in a .h -file:
extern BOOL SetupCTAPIFunctions();

typedef char (__stdcall *CTAPIPROCinit)(WORD,WORD);
typedef char (__stdcall *CTAPIPROCdata)
             (WORD,BYTE*,BYTE*,WORD,BYTE*,WORD*,BYTE*);
typedef char (__stdcall *CTAPIPROCclose)(WORD);

extern HINSTANCE CTAPILib;
extern CTAPIPROCinit          CT_init;
extern CTAPIPROCdata          CT_data;
extern CTAPIPROCclose         CT_close;
```

In the source code of the run-time linked version, somewhere at initialization time, the following code should be executed:

```
// Include this in a source code (.c or .cpp):
HINSTANCE CTAPILib; // Library handle
CTAPIPROCinit CT_init; // Function pointer
CTAPIPROCdata CT_data; // Function pointer
CTAPIPROCclose CT_close; // Function pointer

// Get the library and setup the function pointer addresses
// Returns TRUE if success, else FALSE
BOOL SetupCTAPIFunctions()
{
    CTAPILib = LoadLibrary("ixctapip");

    if(CTAPILib == NULL) return(FALSE);

    CT_init = (CTAPIPROCinit) GetProcAddress(
        CTAPILib, "CT_init");
    CT_data = (CTAPIPROCdata) GetProcAddress(
        CTAPILib, "CT_data");
    CT_close = (CTAPIPROCclose) GetProcAddress(
        CTAPILib, "CT_close");

    if((CT_init==0) ||
        (CT_data==0) ||
        (CT_close==0)) return(FALSE);
    return(TRUE);
}

// Somewhere at init time, do:

if(SetupCTAPIFunctions() == FALSE)
{
    // Probably couldn't find the library.
    // May be OK if this card reader is not used at all,
    // else report "Could not find ixctapip.dll".
}
// OK, carry on.
:
.
```

Whether you have used run-time or load-time linking, the actual calls of the functions now looks the same, illustrated by following example:

```
char Res;
WORD PortNumber=2;
BYTE dad=1,sad=2;
BYTE command[] = {0x20,0x12,1,1,1,10}; // Request ICC,
// wait max. 10 sec.

BYTE inbuffer[50];
WORD received=50;

Res = CT_init(1,PortNumber);
Res = CT_data(1,&dad,&sad,6,
              command,&received,inbuffer);
Res = CT_close(1);
```

Specific proprietary functions

The application programmer may want to reach functions in the smartcard reader that are not part of the standard CT-API or CT-BCS. To call these functions, the general CT_data function is used, but submitting a dedicated "dad" (destination address), one for each of four functions defined. The following dad-values are used:

<u>dad-value</u>	<u>Symbolic names (not real function names)</u>
0x0a	IX_CTAPI_Do_Card_Command
0x0b	IX_CTAPI_Get_Channel_Handle
0x0c	IX_CTAPI_Adjust_Timeouts
0x0d	IX_CTAPI_EscapeCommFunction

The functions correspond to the ones with similar name in the ICCAPI (see the chapter "Using the ICCAPI"). The functions are described in the reference below.

It is also possible to use the ICCAPI functions directly, along with the CT-API. In that case, the *iccapip.dll* must also be linked to, either at load-time or at run-time as described above.

IX_CTAPI_Do_Card_Command

Send a message to the card/card reader, receive an answer from the card/card reader.

Call structure (example):

```
char err;
BYTE dad,sad=2;
BYTE MsgToReader[] = {61,8};           // Card command
BYTE ResponseFromReader[50];
WORD SizeFrom=50, SizeTo;

dad = 0x0a;           // Marks IX_CTAPI_Do_Card_Command
SizeTo = sizeof(MsgToReader);
err = CT_data(1,&dad,&sad, SizeTo, MsgToReader,
              &SizeFrom , ResponseFromReader);
```

Parameters

MsgToReader

A pointer to the message to be sent to the card reader. It includes the <command>, the <parameter> and the <data> fields of the message format (see card reader protocol description).

SizeTo

Number of bytes to send (in the buffer pointed to by *MsgToReader*).

ResponseFromReader

A pointer to an area where to put the response from the reader. The response includes the <command>, the <parameter> and the <data> fields of the message format (see card reader protocol description).

SizeFrom

A pointer to an unsigned short where to put the number of bytes in the response. Upon function call, **SizeFrom* shall contain the maximum number of bytes, so the buffer will not be overflowed.

Return Value

0	Success
10 - 19	Corresponds to codes 1000 - 1009, see the ICCAPI chapter
128 - 255	Error codes from the card reader, see card reader protocol description

Comments

This function enables the application programmer to perform all the card reader functions as specified in the protocol description. Controlling the display, getting card insertion state etc. are examples of operations that are performed by this function. See the protocol description for a list of possible operations. In this direct mode, the command byte, the

parameter byte and the data field all have to be setup in the output buffer. Likewise, the response contains the full message from the card reader, including the command byte, the parameter byte and the data field.

IX_CTAPI_Get_Channel_Handle

Retrieve the socket handle to the TCP connection.

Call structure (example):

```
char err;
BYTE dad,sad=2;
SOCKET ChanHandle;
WORD SizeFrom=4;

dad = 0x0b;           // Marks IX_CTAPI_Get_Channel_Handle
err = CT_data(1,&dad,&sad, 0, 0,
              &SizeFrom ,(BYTE *) &ChanHandle);
```

Parameters

ChanHandle

A pointer to where to put the retrieved 32-bit handle. This handle can then be used in subsequent Win32 socket function calls (such as send(), recv(), select() etc.) directly to the TCP port.

SizeFrom

A pointer to an unsigned short where to put the number of bytes in the response. Upon function call, **SizeFrom* shall contain the maximum number of bytes (≥ 4), so the buffer will not be overflowed.

Return Value

0	Success
10	Channel is not open (the handle retrieved is INVALID_SOCKET, a Win32 constant).

Comments

This function could be used if the application programmer wants to operate the TCP connection in a way not supported by the smart card API. By retrieving the SOCKET handle, the full Win32 Winsock API may be used.

IMPORTANT: Unlike typical Windows manners, the handle retrieved by IX_CTAPI_Get_Channel_Handle shall NOT be closed by a CloseHandle call. The closing of the handle is done by the CT_close library routine. The application does not own the handle, it simply "borrows" it from the *ixctapip.dll* library.

IX_CTAPI_Adjust_Timeouts

Changes the receive timeout used by the smart card API libraries.

Call structure (example):

```
char err;
BYTE dad,sad=2;
int TimeBias;

dad = 0x0c;          // Marks IX_CTAPI_Adjust_Timeouts
TimeBias = 2000;    // Example: 2 seconds prolonged timeout
err = CT_data(1,&dad,&sad, 4, (BYTE *) TimeBias,
              0 , 0);
```

Parameters

TimeBias

A value in milliseconds specifying the time to be added to or (if negative) subtracted from the default receive timeout values otherwise used by the smart card API.

Return Value

Always NULL.

Comments

The functions in the *ixctapi.dll* use reasonable timeout values when communicating with the smart card reader. Should the application programmer find that the timeouts are too short in any situation, the time may be increased with this function. An example of a possible use of this function could be before a procedure on the card, known to be unusually time consuming.

Only the receive timeouts are affected by this function.

A practical approach is to use this function only if the application runs into any trouble due to lengthy operations. The default receive timeout, typically 15 seconds, should be enough. Lowering the timeout by calling the function with a negative value should be used with great care, it does not improve performance of the card operations in general.

By calling the function with a NULL parameter, the timeouts are reset to the default values.

The timeouts used in the transmission to the smartcard are taken care of by the card reader internally and need not to be bothered about.

IX_CTAPI_EscapeCommFunction

Sends a specific code to the *iccapip.dll* library, performing various functions or enquiries.

Call structure (example):

```
BOOL Changed;  
BYTE dad,sad=2;  
DWORD Escapecode;  
  
dad = 0x0d; // Marks IX_CTAPI_EscapeCommFunction  
Escapecode = 503; // Example: Test if card state has changed  
Changed = (BOOL) CT_data(1,&dad,&sad,  
4, (BYTE *) Escapecode, 0 , 0);
```

Parameters

Escapecode

Any of following values:

- | | |
|-----|-------------------------------------------------------------------------------------|
| 503 | Test if card state (inserted/removed) has changed since last call of this function. |
| 507 | Test if smartcard channel has been opened, by ICC_Open_Channel. |

Return Value

- | | |
|---|-------------------------|
| 1 | Function returned TRUE |
| 0 | Function returned FALSE |

For Escapecodes 503-507 this reflects the output of the operation (for example: TRUE if the card state has changed).

Comments

The syntax of this function is, for compatibility with the serial smartcard readers, much like the standard serial Win32 API **EscapeCommFunction** (for the serial card readers there were more escape-codes).

Using `IX_CTAPI_EscapeCommFunction(503)` is a convenient way to get the API check if any state change message have arrived to the TCP receive buffer.