990806 MH

by Mats Hansson, Intertex Data AB
Copyright ©Intertex Data AB

Modem integrated IC card reader: Application Program Interface for Windows 95/98/NT

Table of Contents

	Page
1. General	2
2. iccom.vxd (and iccom.sys)	3
3. iccapi.dll / ixpcsc.vxd	6
4. Access methods	7
5. Installation and naming	9
6. Using iccom.vxd (and iccom.sys)	10
7. Routing through a modem driver	18
8. Using the specific high level API: iccapi.dll	19
9. Using the PC/SC high level API: ixpcsc.vxd	39
10. Using the CT-API high level API: ixctapi.dll	53
11. Using the OCF Java-based API	62

1. General

The built-in IC card (smart card) reader of the modem is accessed through a set of commands as described in a separate document (”Modem integrated IC card reader: Protocol between modem and host computer”). These commands have to share the same physical serial line with the data flow and commands for the modem itself. Consequently, there may be two running applications that have to syncronise their command flow: One for the IC card task and one for another communication task (such as a terminal program, an Internet browser etc.). This is made possible by the command structure and the DLE encapsulation of the IC card messages (framing each message by the special character <DLE>). It is also facilitated through the IC card API described in this document.

The IC card API (application program interface) consists of two software components:

1. A virtual device driver ***iccom.vxd (iccom.sys)***
2. A function library ***iccapi.dll*** alternatively ***ixpcsc.vxd (ixpcsc.sys)***

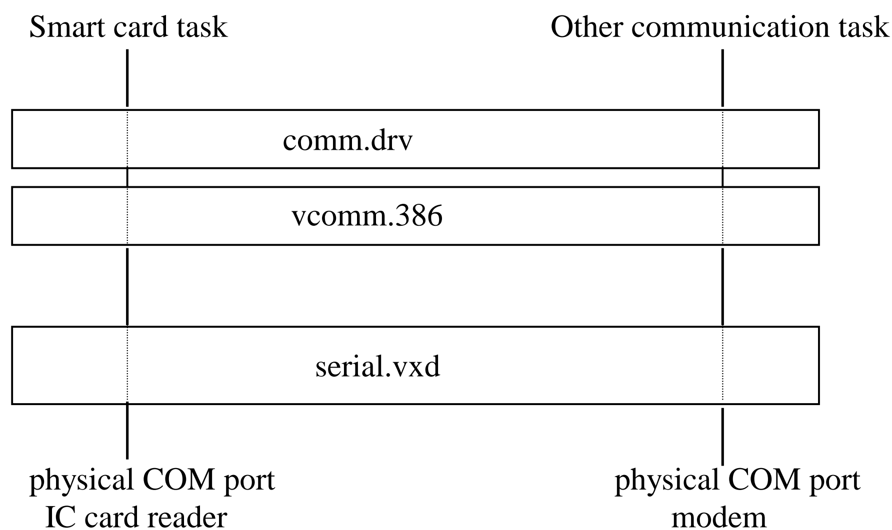
Depending on various needs, a system designer/programmer may want to use only the first, both or none of these components. These three methods of negotiating with the IC card are described later. Below follows a general description of these two components and their role in accessing the IC card/modem COM port. The driver and file names mentioned are the Windows 95/98 names. For Windows NT4.0, the handling is essentially the same, the drivers bearing the extension *”.sys”* instead of *”.vxd”*.

2. iccom.vxd (and iccom.sys)

The virtual device driver *iccom.vxd* does mainly two things:

1. it lets two ”virtual” COM ports merge into one physical COM port
2. it takes care of the insertion/deletion of extra DLE characters necessary for the IC card protocol.

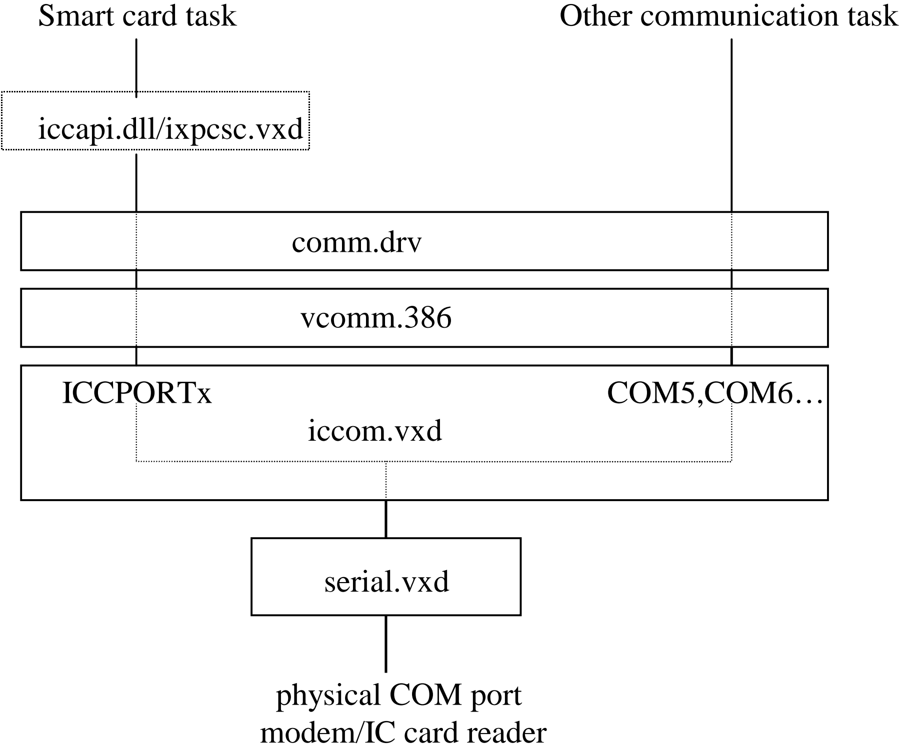
Let us consider a configuration where a modem and an IC card reader reside in two separate units (which is the normal case), taking up two physical serial channels:



Comm.drv is a Windows system component, as is the virtual device driver, vcomm.386. The user interface, as seen by a program written in C, C++ or Pascal, is a set of exported functions in comm.drv to open a port, setup, read and write, and to close the port. This is the Windows communications API. (On Windows NT, the system components are normally referred to as the ”I/O subsystem”).

The communications driver comm.drv and vcomm.386 rely on a VxD, a hardware port driver, *serial.vxd*, that comes with Windows but can be replaced by other virtual device drivers. The port driver, *serial.vxd*, can open a port under the name COM1, COM2, etc. One COM port cannot be accessed by more than one application at a time.

Now, for the modem with built-in IC card reader the communication has to be channelled through an extra virtual device driver that combines two ports into one:



The virtual device driver, *iccom.vxd*, is a ”port-virtualization VxD” that handles no hardware but uses *serial.vxd* for that purpose. Upwards, it behaves like a port driver, like *serial.vxd*. It can open and handle a port, say COM2, under two names: ICCPORT2 and COM6. The first name is for the IC card application, the second name is for the other, ”normal” modem application. The naming conventions is as follows:

”Real”, physical port	Virtual port used by IC-card appl.	Virtual port, 2:nd name
COM1	ICCPORT1	COM5
COM2	ICCPORT2	COM6
COM3	ICCPORT3	COM7
COM4	ICCPORT4	COM8

In this way, the *iccom.vxd* knows the role of the calling task (in Windows terminology: the virtual machine, VM), which is necessary for a proper syncronisation of messages to the modem/IC card reader. It is easily understood that *iccom.vxd* has to know where to send response data that are coming from the modem and must keep track of what is part of the IC card session and what is not. The IC card session may be undertaken right in the middle of a modem data connection without disturbing it notably.

The applications do no longer use the names COM1, COM2, etc. but implicitly refers to these by the names ICCPORT1, ICCPORT2… and COM5, COM6…resp. (this is not totally true, see below under chapter ”Installation and naming”). Consequently, Windows must be configured with two more port entries for each modem/IC card reader: In the example ICCPORT2 and COM6, with their port driver set as *iccom.vxd*. This is done in an installation procedure described briefly in chapter 5. For each physical modem port, only one ”virtual COM-port” is installed, providing the two necessary entries.

In the protocol description for the IC card reader, chapter 4, the DLE encapsulation is examined. The extra DLE characters needed in the messages and in the normal data stream (from the other communication task) are inserted by the *iccom.vxd*, so the application programmer does not have to bother with that. The same goes for the corresponding deletion of the extra DLE characters coming back from the modem/IC card reader. The DLE:s before the frame characters, STX and ETX, are not affected but are let through.

At the moment, the platforms Windows 95, 98 and NT4.0 are supported.

3. iccapi.dll / ixpcsc.vxd

There are two components to facilitate for the programmer working with IC card applications in C/C++. One is a set of functions in a dynamic library called *iccapi.dll* (other languages must emulate C/C++ calling conventions). The other, *ixpcsc.vxd,* is a driver for the PC/SC Workgroup standard, working in conjunction with the Microsoft's smartcard base component. Both the *iccapi.dll* and *ixpcsc.vxd* library/driver hides most of the message handling framework desribed in the protocol description, leaving only the core of telegrams to the programmer. These components must be used in conjunction with *iccom.vxd*, as they depend on the syncronisation and DLE processing done by that VxD.

In the *iccapi.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 modem/IC card reader.
2. Checks that the activation was successful and returns to the caller.

The *iccapi.dll* implements a specific Intertex API for smart card applications. In addition, it has also 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 modem may be controlled by a C/C++ function call. The *iccapi.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.

A more generalised interface is the PC/SC driver *ixpcsc.vxd.* It has similar functions as *iccapi.dll.* It interfaces to the Microsoft smartcard resource handler and serves it with all the standardised functions, like powering on the smartcard, protocol specification and data transmission. On top of that, just like *iccapi.dll,* the *ixpcsc.vxd* has got functions for the vendor specific operations like controlling the display and getting PIN-data from the keypad on the modem. However, if the application needs to use these specific functions some vendor specific code will be needed, as the PC/SC definition has no support of those functions.

Instead of the PC/SC, one may use the German CT-API interface, described in chapter 10, or the Java-based OCF described in chapter 11.

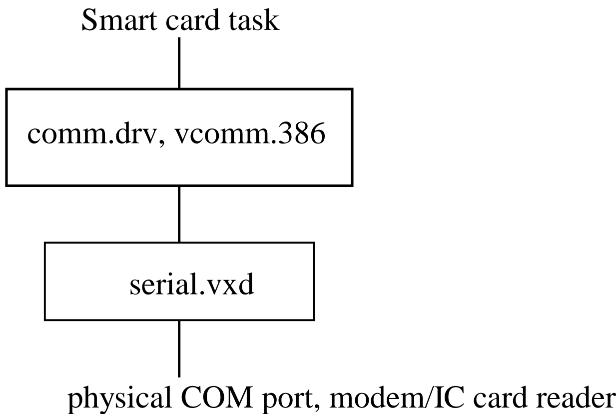
At the moment, the platforms Windows 95, 98 and NT4.0 are supported for the API:s mentioned.

4. Access methods

Depending on which component(s) of the IC card API that are being used one can define three methods (or levels) of access:

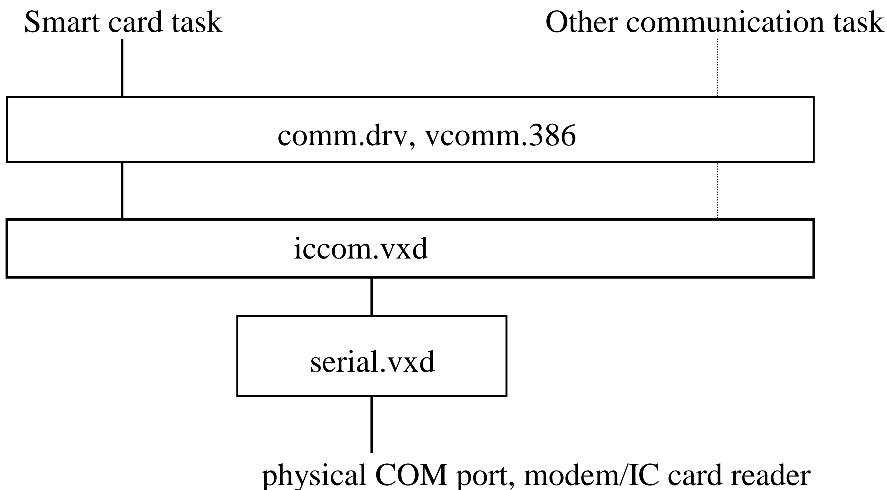
The direct method
With this method, none of the IC card API components are used. The application program opens the COM port for exclusive access and performs all functions as described in the protocol description. Message frames, checksum, DLE insertion/deletion etc. all have to be performed by the application task (typically by a front end module of that task).

The direct method may be used in an application where one use the modem only as a smart card reader, without simultaneously using the modem functionality. Another possible use of the direct mode is when the application itself must have detailed control and is capable of handling the syncronisation between IC card messages and other data communication.



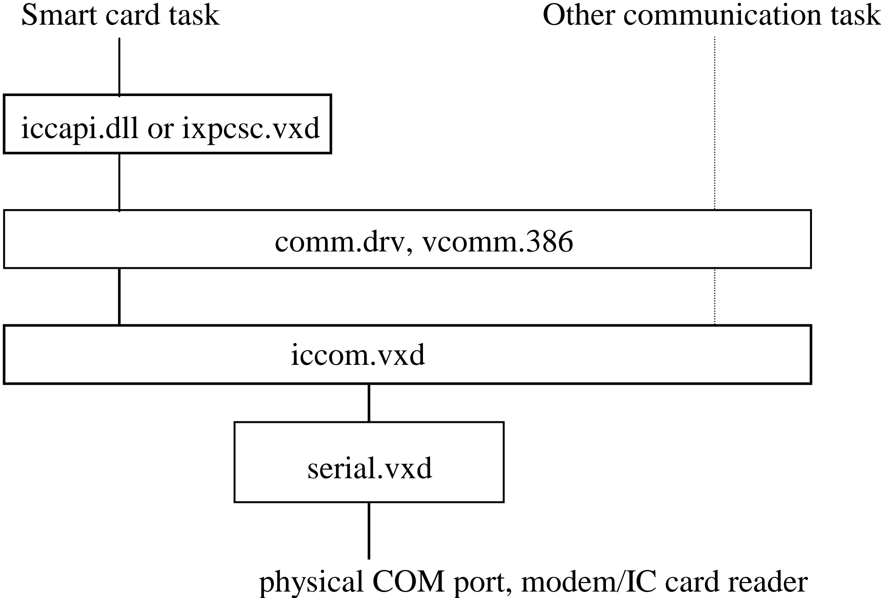
The low level method

This method uses the virtual device driver *iccom.vxd*, but not the IC card API (*iccapi.dll/ixpcsc.vxd*). In this way, the IC card application has a fairly detailed control of the IC card handling. It treats the IC card reader like it was residing on a separate COM-port, though it is accessed as ICCPORTx. Other communication tasks may be alive and concurrently accessing the modem (under name COM5, COM6 etc.). The message formats and procedures are as described in the protocol description (except that DLE insertion/deletion is being taken care of by *iccom.vxd*).



The high level method

With this method both *iccom.vxd* and *iccapi.dll* (or *ixpcsc.vxd*) are being used. This is the easiest way to create a working IC card task from scratch. The programmer does not have to bother about details in the communication and syncronisation procedures. Other communication tasks may be accessing the modem simultaneously.



5. Installation and naming

To use the virtual device driver *iccom.vxd* one has to install one or more virtual COM ports, a procedure very much like installing a new hardware. Such a virtual COM port is simply a way to attach the names ICCPORTx and COM5-8 to their physically connected port COMx. To simplify the installation process, the program *Setup_Sc.exe* has been developed, which quickly installs both the virtual port, a ”virtual modem and possibly the support for the PC/SC interface. We refer to the installation description in a separate document, delivered with the smartcard modem.

The names COM5-8, under which the virtual COM-ports are opened, are default names. For Windows 95/98 these may be changed by editing the *iccom.inf* file before the installation[2]. (The names ICCPORTx should not be changed).

This also solves the problem with application programs that can only use the names COM1-COM4. This may be case for an ”off-the-shelf” communication software product. Assume, for example, that the modem/IC card reader is plugged into the COM2 connector of the computer and that the communication program will share that port with the IC card task. Selecting COM2 will not do, since it locks that port exclusively (the IC card task cannot access it simultaneously) and the *iccom.vxd* functions would be out of reach. The solution is to find a free, unused port name, say ”COM4”, and put it instead of the ”COM6” name in the *iccom.inf* file, under the ”Strings” paragraph (last in the file). The installation is then done by adding ”Virtual COM port 2” as described in the installation document.

(For Windows NT, the changing of the virtual port name must be done by editing the registry after the installation. The key to edit is HKEY_LOCAL_MACHINE\ System\CurrentControlSet\Services\iccom…\Parameters\PortNumberHigh.)

[1)] In practice, for Windows 95/98, there are four driver files, *iccom1.vxd*, *iccom2.vxd*. . . ., one for each possible physical COM-port number. By this built-in relationship, the mapping to the physical port is fast and straightforward.

6. Using *iccom.vxd (and iccom.sys)*

This chapter describes how to use the *iccom.vxd*. It is important information specially for the application programmers that will ***not*** use the high-level API (*iccapi.dll/ixpcsc.vxd*). For the ones that do use the *iccapi.dll* or *ixpcsc.vxd* (”on top of” *iccom.vxd*) some of the things described here are taken care of by that API. It is essential information nevertheless. At this point, the reader is also asked to view the command description in the document ”Modem integrated IC card reader: Protocol between modem and host computer”. The concepts of ‘command mode’ and ‘on-line data mode’ are important to understand.
(The function calls in bold typeface in this chapter are Microsoft® products and as such are only briefly described here. The calling convention of these API functions are depending on the programming environment.)

The two channels

The virtual device driver *iccom.vxd* has been designed to handle two logical communication channels, not generally, but provided that one of the channels is the smart card device built into the modem. This channel is opened by using the name ICCPORTx, in the following referred to as the ”ICC channel”. The ”x” is the COM-port number. The other channel, used for any ”normal” modem communication is called the ”non-ICC channel” and opens the port under the name COM5-8. The *iccom.vxd* has one set of queues, flags, status data etc. for each of the two channels. By the opening calls, one for each channel, the roles are established and two separate handles are given back to the Windows application(s). The handles are then to be used in subsequent calls (such as **WriteFile, ReadFile** etc.) by the application(s) in the same way as described in the Windows documentation (under Communications). Depending on the use of these channels (=handles) one can expect three possible configurations:

1. One application owns the ICC channel (using that handle), another application owns the non-ICC channel.

2. Both the ICC channel and the non-ICC channel are owned by the same application.

3. Only the ICC channel is used, is owned by a single application.

Using only the non-ICC channel is possible, but not practical, since the *iccom.vxd* in this case does not add any functionality not provided by the standard COM-port driver.

A reason for case 3 above would be to make use of the DLE insertion/deletion being taken care of by *iccom.vxd*, though that is not a hard task for the application itself. The same goes for case 2, but in this case it may also be convenient to work with two separate channels, specially if using the on-line data mode (smart card communication when the modem has CONNECT state), where *iccom.vxd* will filter the smart-card messages out of the data stream and redirect them to the ICC channel.

The benefit in case 1 is obvious: Without *iccom.vxd* the two applications cannot own the same physical COM-port. Of course, both channels need not always be opened. One application can close its ICC channel, leaving the other application working on the non-ICC channel alone. At a later time, the same application (or another) can reopen the ICC channel again and start communicating with the smart card in the modem.

Operations

Opening the COM-port for ICC channel is done by the normal open procedure, in the following example (for the COM2 port) showed in Microsoft C/C++ style:

```
char comport[] ="\\\\.\\ICCPORT2";
HANDLE  ICCHand;          // Handle to the COM-port
ICCHand = CreateFile(comport, GENERIC_READ|GENERIC_WRITE,
          0, NULL,OPEN_EXISTING, 0, NULL);
```

The data type "HANDLE" is defined in Microsoft C/C++ and is a 32-bit integer. Consequently, ICCHand becomes an identifier of the ICC channel (= the COM2 port opened for smart card communication). The name "ICCPORT2" is the default name assigned in the *.inf*-file used when installing the virtual COM2-port (see "Installation").
The characters "`\\.\`" must be used when opening a device (as opposed to a file, see Microsoft's documentation), each backslash must be repeated according to the C/C++ string quotation standard. Hence the many backslashes in the name string. If "COM3" or "COM4" is the name, as edited in the *iccom.inf*-file, then the backslashes can be left out.

In the same way, to open the COM2 port for the non-ICC channel is simply:

```
char comcomport[] ="\\\\.\\COM6";
HANDLE  ComHand;             // Handle to the COM-port
ComHand = CreateFile(comcomport, GENERIC_READ|GENERIC_WRITE,
         0, NULL, OPEN_EXISTING, 0, NULL);
```

When this has been done, the handles ICCHand and ComHand resp. will be used in all further calls for setup and I/O to/from the COM2-port. Setting up baudrate etc. is very much like the normal COM port handling. It is generally recommended that the application first gets the current settings (of the DCB, Device Control Block), then changes the parameters that need to be changed. Example:

```
DCB chanDCB;                        // Device Control Block
BOOL state;
state = GetCommState(ComHand,&chanDCB);
          // State is FALSE if any error occured
chanDCB.BaudRate = 57600;
chanDCB.fOutxCtsFlow = TRUE;        // CTS output flow control
chanDCB.fOutxDsrFlow = FALSE;       // DSR output flow control
chanDCB.fDtrControl = DTR_CONTROL_ENABLE; // DTR flow control
chanDCB.fOutX = FALSE;              // XON/XOFF out flow control
chanDCB.fInX = FALSE;               // XON/XOFF in flow control
chanDCB.fRtsControl = RTS_CONTROL_ENABLE;   // RTS flow control
state = SetCommState(ComHand,&chanDCB);
```

However, now it's time to remember that we are actually using only one physical COM port. There is no point in setting up these physical parameters again using the ICC handle. On the contrary, if baudrate, flow control etc. are to be set up twice (the software might force it), it is important that the values are not conflicting. To help controlling this, *iccom.vxd* only allow the DCB to be setup by the ICC channel if it's not previously setup by the non-ICC channel. The latter may always change the DCB, it has got the priority.

Setting the communication timeouts may be done separately for each channel, we show it here for the ICC channel only (**GetCommTimeouts** is really not necessary in the example since all five values are explicitly setup):

```
COMMTIMEOUTS timo;
BOOL state;
state = GetCommTimeouts(ICCHand, &timo);
timo.ReadIntervalTimeout = 0;
timo.ReadTotalTimeoutMultiplier = 0;
timo.ReadTotalTimeoutConstant = 20000;
timo.WriteTotalTimeoutMultiplier = 0;
timo.WriteTotalTimeoutConstant = 20000;
state = SetCommTimeouts(ICCHand, &timo);
```

Sending data to the ICC channel:

```
char comstring[] ="AT*SC\r";
int bytessent;              // No. of bytes actually sent
if(WriteFile(ICCHand,(LPCVOID)comstring,strlen(comstring),
         (LPDWORD)&bytessent,NULL) == TRUE)
{
         // The call was successful.
         // Timeout, however unlikely, is not treated as an
         // error! Best check that bytessent equals comstring.
}
else
{
         // Report error
}
```

The receiving from the ICC channel is done similarly:

```
char inbuffer[100];
int received;               // No. of bytes actually received
if(ReadFile(ICCHand,
         (LPVOID)inbuffer,
         15,                   // number of bytes to read
         (LPDWORD) &received, // pointer: number of bytes read
         NULL) == TRUE)
{
         // Timeout not treated as error! Important to
         // check that any bytes actually were received.
}
else
{
         // Report error
}
```

When reading from the ICC channel the ”number of bytes to read” parameter is not critical. The *iccom.vxd* is keeping control of what is received from the IC card reader, either in an

AT*SC command sequence or in the on-line data mode (see the protocol description). Thus, end-of -reception is established when appropriate, even if the requested number of characters are not fulfilled. For example, when <CR><LF>CONNICC<CR><LF> is fully received, end-of-reception happens and the control returns to the application without timeout. This frees the application programmer from the tedious task of calculating the expected number of characters in a response (which may be dependent of whether echo is on or off). A standard value of 300 bytes covers all possible smart card responses and can be used in each call of **ReadFile**.

Sending and receiving data to from/to the non-ICC channel is quite similar. Simply use the ”ComHand” instead of the ”ICCHand” in the examples above. However, for the non-ICC channel the driver *iccom.vxd* cannot possibly determine when the received data is ended, as for the ICC channel. The number of expected bytes must therefore be correct or a timeout will happen (which may of course be normal and planned).

Another feature of the *iccom.vxd* is the deletion/insertion of DLE-characters. In the protocol description (chapter 4) one can read about the DLE encapsulation needed to isolate the IC card messages from other modem data. The framing <DLE><STX> and <DLE><ETX> must of course be sent and received to the ICC channel application. But the application programmer does not have to bother about any duplicating of DLE:s when calling **WriteFile**. This is true for both the ICC channel and the non-ICC channel. Likewise, any extra DLE:s inserted in the data stream from the modem to the PC will be stripped off before the data enters in the receive buffer in the application. For this to work, two rules has to be taken into consideration:

1. When calling **WriteFile** (or whatever the send function is called) to send binary data to the IC-card (that is data starting with <DLE><STX>…), it is important that the number of bytes stated is correct and covers the frame (<DLE><STX>……<DLE><ETX>). Nothing more than this single, framed message may be sent by that WriteFile call.

2. In command mode, when starting a command sequence, the AT*SC must be sent in that order, no characters between ”AT” and the asterisk. The characters AT*SC must be sent in one **WriteFile** call, without split.

After use, at end of program or after a session, the ICC channel and the non-ICC channel should be closed just like an ordinary COM-port:

```
CloseHandle(ICCHand);
```

resp.

```
CloseHandle(ComHand);
```

The COM-port is not free for use by other Windows application until both channels have been closed. A channel is also closed by the VCOMM if the application (window) that opened it is closed, but it is good practice to make use of the **CloseHandle** call before that happens.

Should any error occur during transmission or reception of data, it may be wise to call **PurgeComm** to empty all buffers, discard the data in them and terminate pending read or write operations. See Windows documentation. **PurgeComm** is called using the handle (ICC- or non-ICC) as usual.

More on command mode/on-line data mode

The protocol description (see separate document) explains the two modes of operation of the built-in smart card reader. Obviously, sending a message to the smart card is done quite differently whether the modem is in on-line data mode or AT command mode. In the first case, the message is sent directly, in binary format and framed by DLE-sequences, the latter mode involves a sequence starting with the AT*SC string.

Consequently, the ICC application must know in what state the modem is, before issuing any smart card command. This is easily done by two function calls to the driver *iccom.vxd*. These functions are special to *iccom.vxd* but are called using the general **EscapeCommFunction**, which otherwise is used for setting and clearing interface bits (DTR, RTS etc.) and other ”extended” functions. The use of the special function calls is demonstrated by following example:

```
BOOL state;
state =  EscapeCommFunction( ICCHand,500 );
//state =  ICC_EscapeCommFunction( 500 ); //See below
if(state == TRUE)
{
         // The IC card reader is in on-line data mode,
         // send binary data to smart card.
}
else
{
         // Check if IC card reader is in AT command mode:
         state =  EscapeCommFunction( ICCHand,501 );
         if (state == TRUE)
         {
                  // AT command mode confirmed.
                  // Start AT command sequence.
         }
         else
         {
                  // Not possible to send any data to IC card
                  // The modem may be on-line without the
                  // special on-line data mode. This should
                  // not happen.
         }
}
```

The values 500 and 501 above are the function codes:

BOOL **EscapeCommFunction**(HANDLE hand,**500**); // Test if on-line data mode
BOOL **EscapeCommFunction**(HANDLE hand,**501**); // Test if command mode
For Windows NT, **EscapeCommFunction** cannot be used for these non-standard escape codes. For this purpose the **ICC_EscapeCommFunction** in the *iccapi.dll* has been supplied. System programmers that want to make their applications NT-compatible should therefore always use the *iccapi.dll* library function **ICC_EscapeCommFunction** instead of the standard **EscapeCommFunction**. (**ICC_EscapeCommFunction** is described in a following chapter.)

For the NT-compatible calls using *iccapi.dll*, the handles should be omitted:
ICC_EscapeCommFunction(**500**);   / / Test if on-line data mode
ICC_EscapeCommFunction(**501**);   / / Test if command mode

If not using *iccom.vxd*, the command AT*SM=1 would have to be sent prior to the data connection (the modem is on line), otherwise it would not be possible to access the smart card at all (see protocol description). However, the *iccom.vxd* have another signalling scheme to make the modem enter the on-line data mode during next connection. Therefore, users of *iccom.vxd* (and the higher levels API:s *iccapi.dll/ixpcsc.vxd*) do not have to bother about the AT*SM=1 command at all.

In practice, it will not be possible to go online through the *iccom.vxd* driver without also entering the smartcard online data mode, involving DLE insertion/deletion. This will introduce a small overhead to the data transmitted, but the data contents will not be affected in any direction. If a pure data connection without the DLE handling is desired, the user should open and use the normal (physical) COM-port driver directly, in which case the smartcard is not simultaneously accessible.

State change enquiry

As described in the card reader protocol description (command no. 8), the modem/IC card reader may be set up to send a special character each time the card is inserted or pulled out. The special character is taken care of by *iccom.vxd,* which does not forward this character (<DC4> or <DLE><DC4>) to the application. Instead, the fact that a state change has occured is remembered by *iccom.vxd.* The application can enquire about a state change by an **EscapeCommFunction**:

BOOL **EscapeCommFunction**(HANDLE hand,**503**);   / / Test if state has changed
BOOL **ICC_EscapeCommFunction**(**503**);   / / Same, using NT-compatible *iccapi.dll*

If this function returns TRUE, then a change of state has taken place since last call of **EscapeCommFunction(. . .,503)** or since the state change alert mode is switched on (see command 8 in the protocol description). The **EscapeCommFunction** does not involve any data transmission to the modem, nor any heavy processing. It can therefore be placed in a background loop in the application, thus monitoring the card insertion state. The **EscapeCommFunction** does not tell whether the card is inserted or not. That has to be enquired by the application by the use of the ”Get status” command (command no. 3).

Open-by-other-application enquiry

The smartcard application may want to know, by any reason, if a non-ICC application also has opened the virtual port (the non-ICC channel is open). This could be done by the use of **EscapeCommFunction(handle,506)** (or **ICC_EscapeCommFunction(506)**). If the function returns TRUE, the non-ICC channel is open, if FALSE it is closed.

In the same way, **EscapeCommFunction(handle,507)** tells whether the smartcard application has opened the virtual port (the ICC channel is open) or not.

These functions could be useful, for example, if the smartcard application should perform some specific actions when an Internet session or another communication session is finished (regularly or unexpectedly), or if the applications involved wants to synchronize their access to the port in a particular way.

Some restrictions

The virtual COM-port driver *iccom.vxd* attempts to hide the fact that both the modem and the smart card reader resides on the same physical COM-port. To the applications, it will present what may be regarded as two fully separate COM-port interfaces. However, if two separate applications are controlling the COM-port, unsynchronized and as if each were the only owner of the port, there may be situations where *iccom.vxd* simply cannot do the simulation to 100%. In order to reduce the complexity to a reasonable level, a few restrictions are necessary and must be kept in mind:

1. The baudrate, parity etc. have already been discussed above (under DCB, SetCommState). When using the ICC channel in the command mode, the system programmer must keep in mind that the modem autobauds (detects the baudrate and parity automatically) at every AT-sequence. So it really doesn't matter if the baudrate is changed by the non-ICC applications. Knowing the priority rule described above, it is good practise in the ICC application to set up a nice and fast baudrate (for example 57600 baud) when opening the ICC-channel but accept that the non-ICC application may change the speed later. If the non-ICC applikation was the first one to open the port, then *iccom.vxd* will stick to that baudrate, regardless of the ICC application's wish.

2. Only hardware flow control using RTS is supported for input flow control, and only for the non-ICC channel. It is assumed that the messages from the smart card never makes the input queue overflow. The XON/XOFF flow control is not supported for input (receive) flow control. Output flow control has no restrictions.

3. By the **EscapeCommFunction** the DTR and RTS signals may be explicitly set/reset for a ”normal” COM-port driver. However, for the *iccom.vxd* the use is restricted. For the ICC channel those calls are only permitted if the non-ICC channel is closed (the ICC channel is the sole user of the port). The non-ICC channel, on the other hand, can use these functions at any time and consequently has got the priority of these signals.

4. The maximum size of the receive buffer is 6 Kbytes and for the transmit buffer 16 Kbytes. For the ICC channel the maximum size is 300 bytes which should cover a 256-byte message plus all the frames at several levels.

5. It is easily understood that the physical COM-port cannot send data from both the ICC channel and the non-ICC channel at exactly the same time. When one of them has started a transmission, the other will have to wait until it (=that output buffer) is finished. This syncronisation is taken care of by *iccom.vxd*, but it has to be taken into consideration when setting up timeout time values. Apart from that syncronization, there is also a mechanism in the *iccom.vxd* that makes the AT-commands from both channels not conflicting with each other. An AT-command from one channel will have to wait until the other channel is finished, after an ”OK” or ”ERROR” answer from the modem has arrived. Without this syncronisation, the answers from the modem would be mixed up and difficult to trace to what channel they belong. In this syncronisation, the AT*SC-

sequence is not considered finished until the full sequence, involving ”CONNICC” and binary data, has ended.

6. It should not be regarded as possible for the non-ICC channel to send ICC commands and vice versa: for the ICC channel to establish and use the normal modem communication. However, it is possible for the ICC channel to send ordinary AT-commands to the modem. Nevertheless, it is good practice to let the ICC channel be unaware of the fact that it is a modem (and not just a smart card reader) at the other end.

7. Routing through a modem driver

A common way for a Windows application to communicate with the modem is through an installed modem device driver, rather than directly accessing the COM-port. So if the non-ICC channel described in the previous chapters is designed to use Windows TAPI or an installed modem driver, this installed modem has to be redirected to the virtual COM-port, in our example the virtual COM2. If the modem has been detected and installed automatically by Windows PnP, it is not possible, and perhaps not desirable, to permanently change the port attachment for that modem. The solution is to manually install another copy of the modem and to choose the virtual COM-port for that modem. The application should then be setup to use this ”virtual” modem instead.

It is assumed that this problem does not apply to the smart card application. Such an application has no reason to use modem functions and is most probably designed to use a COM-port directly. Consequently, the name for the non-ICC, modem application to use when opening the virtual COM-port should always be the non-ICC name, in the example ”COM6” (or the substituted name edited in the *.inf*-file).

For the installation procedure, we refer to the separate description.

After the installation of the extra modem copy, the user of a communication application should be able to select and setup this installed ”virtual” modem, perform any normal action on the modem through the COM-port and being able to share that port with a smart card application.

8. Using the specific high level API: *iccapi.dll*

This chapter describes how to use the functions in *iccapi.dll*. This library relies on *iccom.vxd* for much of its functionality. Using *iccapi.dll* without *iccom.vxd* is not possible. At the moment, the *iccapi.dll* includes only the specific Intertex smart card API, described here. It is also possible to use the German standard CT-API (also used elseware), described in chapter 10. The CT-API implementation described there make use of the *iccapi.dll*. Likewise, the OCF (Java API) makes use of the *iccapi.dll* as described in chapter 11.

The library, along with some sample code, is included on the diskette marked ”Developers Kit”.

General

The specific API consists of a set of nine C/C++ functions that hides most of the transport level of the smart card reader, the COM-port driver and *iccom.vxd*. No fiddling of DLE-characters is needed, nor has the application programmer to bother about the checksum in the message frame (there may still be checksums associated with the higher (inner) levels of a protocol, such as T=1 of ISO 7816-3). The AT*SM=1 (see protocol description) command does not have to be issued as the *iccom.vxd* takes care of that signalling.

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 virtual COM-port) and for activating/deactivating the smart card. There are also three functions that enables a more direct communication with the smart card reader.

The functions 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

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)
```

These functions, all defined in *iccapi.dll,* uses the ICC channel as described in chapter 6. It opens the virtual COM-port under the name ”ICCPORTx” where x is the COM-port number specified by the application in the ICC_Open_Channel call. Needless to say, the virtual COM-port with this number must have been correctly installed (using the procedures described in the previous chapter) in order to use the *iccapi.dll* functions.

Using the *iccapi.dll* function library

As other DLL:s (Dynamic Link Libraries), there are two ways of linking to the *iccapi.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 *iccapi.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, *iccapi.lib*. This file is delivered on the Developers Kit diskette, along with *iccapi.dll*.

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(HANDLE*);
extern int ICC_Adjust_Timeouts(int);
extern int ICC_HookUnhook_PCSC(int);
extern BOOL ICC_EscapeCommFunction(int);
```

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

The other way of linking to *iccapi.dll* is run-time dynamic linking. In this case, when calling the linker for the application program, the *iccapi.lib* is not needed (and shall not be used). Instead, the application code must explicitly load the *iccapi.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)(HANDLE*);
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("iccapi");    // 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 virtual COM-port
int PortNumber=2;          // Use COM2
Res = ICC_HookUnhook_PCSC(0);
Res = ICC_Open_Channel(PortNumber);
if(Res)
{
         // Display value of Res (=error code)
}
else
{
         // Successful, channel is opened
}
```

The sample code showed here is included on the installation diskette, in file *icsample.c*, to facilitate editing the application source code.

The *iccapi.dll* file must be in the Windows 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 specific Intertex smart card API is described more thoroughly. The data types used are int, unsigned char and HANDLE. The int is a 32-bit signed integer. The HANDLE is a 32-bit unsigned integer and is typedef:ed to a handle of a COM-port (may be named differently in other programming environments). The DWORD is a 32-bit unsigned integer.

ICC_Open_Channel

Opens the virtual COM-port for smart card accesses.

```
int ICC_Open_Channel(
          int ComportNumber
          );
```

Parameters

ComportNumber

A value 1-4 specifying which COM-port to use, or rather, which of the names ICCPORT1, ICCPORT2, ICCPORT3 or ICCPORT4 to be used when opening the ICC-channel.

Return Value

NULL if success, otherwise an error code. Typical causes of open failures are COM-port opened by another application, or virtual COM-port not properly installed.

Comments

If the channel is not opened, subsequent calls to the API will return the error code ICC_ERR_NOTOPEN.
When opening, the DTR signal of the modem will be raised, if the non-ICC channel has not been opened. If the non-ICC channel has been opened, it will leave the DTR untouched.
The opening procedure sets the XON/XOFF flow control to OFF but leaves most other communication parameters untouched, such as baudrate, parity etc. (see the discussion of DCB, baudrate etc. in chapter 6).

If knowing the COM-port number presents a problem, the application may call Windows API function to read in the registry. The number to look for may be found in one of the keys under HKEY_LOCAL_MACHINE/Enum/Root/Ports (Windows 95/98). Alternatively, one can try to open all four possible numbers 1-4 and check for a successful return, the ICC_Open_Channel will only try to open the virtual COM-port that exists. If there are more than one virtual COM-port installed (deliberately or by mistake), these strategies will fail.

ICC_Close_Channel

Closes the virtual COM-port for smart card accesses.

```
int ICC_Close_Channel();
```

No parameters

Return Value

Always NULL, whether the COM-port was open or not.

Comments

When closing, the DTR signal of the modem will be lowered, if the non-ICC channel is not open. If the non-ICC channel is still open, this function will leave the DTR untouched. When the application exits, and if this function has not been called, the ICC channel will be closed anyway.

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.

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. Error codes above the value 1000 may result from message transfer problems, such as COM-port timeouts, bad message frames or corrupted data.

Comments

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

Used only after a PIN-code entry, should be NULL otherwise. After a PIN-code entry, this value may specify an offset in the *DataToCard* buffer where the card reader shall copy the (coded) PIN digits. Upon call of this function, that part of the *DataToCard* buffer shall be padded with suitable ”background” values if the PIN digits don’t fill the area completely. (See also under ”Command 7” and ”Command 21” in the card reader protocol description.)

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 interchangeble: 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 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. Entering of PIN-code, 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. The checksum byte (LRC) and the DLE-framing are being taken care of and should not be included. Likewise, the response contains the full message from the card reader, including the command byte, the parameter byte and the data field, but not including the LRC checksum or the DLE frames.

The use of ICC_Do_Card_Command is illustrated by the following example:

```
#define  BUFSIZE 10

int err,SizeI;
unsigned char GetPIN[]={7, // Accept PIN-code entry
                  0,       // ASCII coding
                  4,       // 4 digits expected
                  100,     // 10 seconds timeout
                  35       // Use ”#” as ENTER-key
                  };
unsigned char inbuffer[BUFSIZE];

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

err = ICC_Do_Card_Command(GetPIN,5,inbuffer,&SizeI);
if(err)
{
    switch(err)
    {
         case 148:   // Too many digits entered
         case 149:   // Timeout
         case 150:   // Illegal key pressed
             // Report the error to operator
             break;
         default:
             // Other error
             break;
    }
}
else
{
    // PIN-code entry successful
    if(inbuffer[1] != 4)   // Check number of digits entered
    {
         // Maybe we don’t accept fewer than 4 digits
    }
    else
    {
         // Successful
         err = ICC_Data_To_Card(...);   // Send PIN to card
    }
}
```

ICC_Get_Channel_Handle

Retrieve the handle to the virtual COM-port used for smart card access

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

Parameters

ChanHandle

A pointer to where to put the retrieved 32-bit handle. This handle can then be used in subsequent Win32 communication function calls (such as WriteFile, GetCommState etc.) directly to the virtual COM-port.

Return Value

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

Comments

This function could be used if the application programmer wants to operate the COM-port in a way not supported by the smart card API. By retrieving the port handle, the full Win32 communication API may be used. The (virtual) COM-port may be written to or read from, the DCB (Device Control Block) including baudrate, parity etc. may be setup. A certain care is recommended when using this direct mode, the programmer should study the chapter 6.

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 *iccapi.dll* library.

An example of the use is illustrated below, the task here is to reset the modem:

```
int err;
unsigned int bytessent;
HANDLE hand;
char ResetString[] = ”ATZ\r”;

err = ICC_Get_Channel_Handle(&hand);
if(err)
{
         // Channel not opened, maybe we shall open it.
}
else
{        // Good handle: Send ATZ to the modem
    WriteFile(             // Win32 function
         hand,             // using the handle
         (unsigned char *) ResetString,      // pointer
         strlen(ResetString),                // length
```

```
        &bytessent,                  // actually sent
        NULL                         // OVERLAPPED not used
        );
}
```

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 *iccapi.dll* use reasonable timeout values when communicating with the smart card reader. The *iccom.vxd* normally detects end of transmissions based on message contents, so the send/receive procedures should rarely be timed out at all. 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. Such a lengthy procedure, the PIN entry, is already taken care of in the API (when that function is ordered by ICC_Do_Card_Command), so no need for adjusting timeout then.

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 6 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.

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 specifiying what to do:

0	Unhook the reader from the Windows PC/SC and free it for use by *iccapi.dll.* (The function takes 500 ms if PC/SC is running).
1	Hook the reader back again to the Windows PC/SC system.
2	No function, returns information only.

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.
1-4	COM-port number that the reader is connected to. Also signals that 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 prevents non-PC/SC-aware applications from accessing the reader. Consequently, an application using *iccapi.dll* has to temporarily free the reader from the PC/SC ownership. 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 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 *iccom* driver, performing various functions or enquiries.

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

Parameters

Escapecode

Any of following values:

500	Test if in on-line data mode
501	Test if in off-line command mode
503	Test if card state (inserted/removed) has changed since last call of this function.
506	Test if non-smartcard channel (COM5-COM8) has been opened
507	Test if smartcard channel (ICCPORTx) has been opened
CLRDTR	Standard Windows code, see Windows documentation
SETDTR	”
CLRRTS	”
SETRTS	”
RESETDEV	”

(For Windows 95/98 there are other standard escape codes defined)

Return Value

A non-zero value for TRUE, zero for FALSE. For Escapecodes 500-507 this reflects the output of the operation (for example: TRUE if the card state has changed).

Comments

This function may be used instead of the standard Win32 API **EscapeCommFunction**. For Windows 95/98 it functions pretty much the same, except that the handle to the COM-port does not have to be specified (it is already known by the *iccapi.dll* after a ICC_Open_Channel call.).

For Windows NT, it allows the use of the specific escape codes 500-507, which would otherwise not be allowed. These codes are converted into IOCTL-codes by the *iccapi.dll*, before the actual driver is called. Applications that have to be compatible with both Windows 95/98 and Windows NT therefore have to use this function instead of **EscapeCommFunction**.

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.
1001	ICC_ERR_DCBFAIL	Error when setting up the DCB, device descriptor block.
1002	ICC_ERR_TIMOSETUPFAIL	Error when setting up the timeout values of the virtual COM-port.
1003	ICC_ERR_ATSCFAIL	Error in the 1:st phase of the AT*SC command procedure. Most probably, there is no contact with the modem.
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.
1009	ICC_ERR_NOCMDMODE	The card reader is neither in command mode, nor in on-line data mode and cannot be accessed.

9. Using the PC/SC high level API: *ixpcsc.vxd*

PC/SC is the name of a standard that Windows 95/98/NT and other operating systems may use when communicating with smartcards. 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.

This chapter describes how to use the functions in *ixpcsc.vxd* (for Windows NT4.0 the name is *ixpcsc.sys*, for simplicity we will only use the *.vxd*-name below). This library relies on *iccom.vxd* for much of its functionality and also on the Microsoft PC/SC base component for smartcards (the base component is normally delivered with the application, not with the card reader). Using *ixpcsc.vxd* without *iccom.vxd* is not possible. For the general funcionality, we refer to the Microsoft documentation in the Windows Smartcard SDK. It should normally be sufficient for using the *ixpcsc.vxd.* However, some vendor-specific functions, described below, may be wortwhile to study. There are also some consequences of the PC/SC support on the port ownership that should be considered, they are described below.

General

The PC/SC support has to be installed by a procedure involving copying some files and writing to the Windows registry.

For Windows 95/98 the files are *ixpcsc.vxd* (the driver itself) and a utility program, *ixpcscst.exe* needed to start the card state monitoring in the driver. The latter is started at Windows startup. Along with these files, the program *ScPort* (see below) is copied to the harddisk drive. See the installation description for further information.

For Windows NT4.0 only the files *ixpcsc.sys* and *ScPort* are copied.

The Windows PC/SC has been designed to control the smartcard reader right from the startup of the system. Thus, the 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 *ixpcsc.vxd* opens the non-ICC channel right from the beginning and keeps it opened (this is an importanf difference from the specific API, *iccapi.vxd,* where the port is opened/closed only on commands from the smartcard application). This means that any communication program must access the COM-port through the non-ICC channel, addressed as COM5, COM6, COM7 or COM8 or indirectly using a modem routed to one of those. A problem arises if a program must open the (physical) COM2 (or whatever port is used), since this is owned by the virtual COM-port driver *iccom.vxd.* A DOS program, for example, cannot open COM5-8. In that case the PC/SC must first close the ICC-channel, so the physical port is released. Another way to look at it is that the smartcard handling must be disabled for some time, while the DOS application is running.

The disabling/enabling of the smartcard access is done by the enclosed program *ScPort.exe.* With this program the user can temporarily disable the smartcard handling, to let the COM-port be freed. Later on, the user can click the ”Enable” button in the *ScPort* program to resume the smartcard handling and to let PC/SC open the ICC-channel again. For a

Windows program that simply cannot be made to open the virtual port (COM5-8) or the ”virtual” modem, the same procedure must be used. Consequently, such a program can’t be used simultaneously with a smartcard application.

The enable/disable of the PC/SC handling may also be performed from an application program, by use of the release/resume IOCTL codes, see the reference chapter below.

The PC/SC smartcard driver *ixpcsc.vxd* 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.

The user are reminded that the AT*SM=1 (see protocol description) command does not have to be issued as the *iccom.vxd* takes care of that signalling.

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. Below are the specific IOCTL that the application may want to use, in order to reach function in the smartcard modem 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 document ”Smart Cards, Calais Implementation Design for Windows”.

The recommended way to send a specific IOCTL code to *ixpcsc.vxd* 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.

Troubleshooting

The PC/SC interface model introduces several layers of components that process a smartcard message before it is actually sent to the physical smartcard reader. Therefore, when troubleshooting, it may be of interest to see what messages that actually have been sent out and received from the reader. A special debug version of *ixpcsc.vxd* contains functions to log all such message to a disk file, that can be studied afterwards. The log-file, if opened, is named *ixpcslog.txt* and resides in the current directory.

The log-file is a text-file containing time-stamped contents of messages in hexadecimal form as they are sent/received to/from the virtual COM-port driver *iccom.vxd*. The purpose is to detect errors in the chain: application/service providers/resource handler/driver, rather than physical problems with the serial communication (in which case a true line monitoring would be needed). Consequently, there is no logging of data routed through the non-IC-card channel of the *iccom.vxd* (see previous chapters) as these are normally not subjects to the smartcard application, nor passing *ixpcsc.vxd* at all. To understand the contents of *ixpcslog.txt* one has to be a bit familiar with the smartcard reader protocol as described in the separate protocol description. In seeking support the *ixpcslog.txt* may be useful to supply to the programmer.

The normal release version of ixpcsc.vxd does not contain the log feature for security reasons. The use of IX_IOCTL_OPEN_LOG (described below) on a non-debug version returns an error code. A resonable strategy in an application program is to be able to select a debug mode where calls of IX_IOCTL_OPEN_LOG and IX_IOCTL_OPEN_LOG are included. Should a log file be needed, the user can then switch the normal *ixpcsc.vxd* to the debug version (in the Windows system directory). The debug version must be specially obtained from Intertex Data AB. Likewise, a program called *Logix.exe* may be obtained. This has got two click-buttons for opening and closing the log-file, by the use of the IOCTL codes mentioned.

This example shows the layout of a log-file:

```
********* Log-file for smart card reader. ******************************
********* Started at: 25/03/1998 08:33:56 ******************************

HOST:   (after 00:00:01.89) (doing IOCTL: IOCTL_SMARTCARD_POWER)
41 54 2a 53 43 0d                                      AT*SC.

READER: (after 00:00:02.04)
41 54 2a 53 43 0d 0d 0a 43 4f 4e 4e 49 43 43 0d 0a      AT*SC...CONNICC..

HOST:   (after 00:00:02.04)
10 02 14 00 14 10 03                                    .......

READER: (after 00:00:03.16)
10 02 14 7e 41 c0 02 f4 04 00 00 01 90 00 88 10 03      ...~A...........
0d 0a 4f 4b 0d 0a                                       ..OK..

HOST:   (after 00:00:03.16)
41 54 2a 53 43 0d                                      AT*SC.

READER: (after 00:00:03.31)
41 54 2a 53 43 0d 0d 0a 43 4f 4e 4e 49 43 43 0d 0a      AT*SC...CONNICC..

HOST:   (after 00:00:03.31)
10 02 01 00 01 10 03                                    .......

READER: (after 00:00:03.62)
10 02 01 7e 3b fa 11 00 02 40 20 41 c0 02 f4 04 00      ...~;....@ A.....
00 01 90 00 2f 10 03 0d 0a 4f 4b 0d 0a                  ..../....OK..

HOST:   (after 00:00:05.19) (doing IOCTL: IOCTL_SMARTCARD_TRANSMIT)
41 54 2a 53 43 0d                                      AT*SC.

READER: (after 00:00:05.34)
41 54 2a 53 43 0d 0d 0a 43 4f 4e 4e 49 43 43 0d 0a      AT*SC...CONNICC..

HOST:   (after 00:00:05.34)
10 02 15 00 a0 a4 00 00 02 00 20 33 10 03               .......... 3..

READER: (after 00:00:05.60)
10 02 15 7e 90 00 fb 10 03 0d 0a 4f 4b 0d 0a            ...~.......OK..
```

The first two lines in the file shows the base time and date for the log. All timestamps in the file is time elapsed (in hours, minutes, seconds and 10ms units) after that time. For each message, the hexadecimal value to the left is converted to an ASCII string to the right. Only meaningful ASCII characters are converted, other bytes are marked with a dot. The IOCTL code, whenever received by the *ixpcsc.vxd*, is reported as ”doing IOCTL: . . .”. Only the

IOCTL codes that actually lead to message transmission is reported, there may be other IOCTL:s received in between.

In the example it can be seen that the IOCTL call ”IOCTL_SMARTCARD_POWER” basically lead to two messages with responses, the rest is protocol overhead. The ”real” smartcard messages are always framed with 10 02 . . . . 10 03 which is the DLE,STX and DLE,ETX (see the protocol description).

Reference

On the following pages each specific IOCTL function of the *ixpcsc.vxd* smart card API is described. The codes are defined in a header file, *ixpcsc.h,* that are supplied on the diskette labeled ”Developers Kit”. 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. Entering of PIN-code, 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. The checksum byte (LRC) and the DLE-framing are being taken care of and should not be included. Likewise, the response contains the full message from the card reader, including the command byte, the parameter byte and the data field, but not including the LRC checksum or the DLE frames.

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_SET_PIN_OFFSET (Value 0x312704)

Prepare for entering a recently typed PIN-code into the card.

Parameters

lpInBuffer

A pointer to a DWORD (32bits unsigned integer) containing the offset in next subsequent card data to transmit to the card.

nInBufferSize

Should be 4 (but is not tested).

lpOutBuffer

Not used.

nOutBufferSize

Not used.

lpBytesReturned

Not used.

Return Value

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

Comments

This function is used only after a PIN-code entry, just before sending data to the card in which to include the entered PIN-code. The PIN-code is temporarily stored in the modem, awaiting the subsequent data write command. When such a command arrives, the IC-card reader copies the PIN-code onto a position, relative to the beginning of the data, specified by this PIN-offset. That write command data shall be padded with suitable ”background” values if the PIN digits don’t fill the area completely. (See also under ”Command 7” and ”Command 21” in the card reader protocol description.)

The PIN-coded is stored in the card reader only up to next card/card reader command, so a subsequent data write command using the entered PIN-code should follow directly, with no other commands in between.

IX_IOCTL_SEND_RECEIVE_ANY (Value 0x31270c)

Send any command, without any protocol whatsoever, to the modem/card reader, receive an answer from the modem/card reader.

Parameters

lpInBuffer

A pointer to a DWORD (32bits unsigned integer) specifying a timeout value in milliseconds, followed by the command to be sent.

nInBufferSize

Number of bytes in the *lpInBuffer*. It is the size of DWORD (=4) plus the number of bytes to send to the modem/card reader.

lpOutBuffer

A pointer to an area where to put the response from the modem/reader.

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 is a fully transparent send/receive. Normally, there should be no need for an application to use this command, as long as the application is only interested in smartcard operations. However, knowing that there is a modem at the end of the serial channel, the application may want to send a modem AT-command. This should be used carefully. As described previously in this document, the normal communication application running on the same channel is the one that shall have the control over the modem behaviour.

The timeout in the *lpInBuffer* buffer should always be setup properly. It is the max. time in milliseconds to wait for all the number of bytes (as stated by *nOutBufferSize*) to be fully received. The driver makes no assumptions of suitable timeout values, nor of the contents of the data to be sent and received. A timeout, if it happens, is not treated as an error code. Thus, the application shall check *lpBytesReturned* in order to find out if the reception was satisfying.

IX_IOCTL_ADJ_TIMEOUTS (Value 0x312710)

Changes the receive timeout used by the *ixpcsc.vxd* driver.

Parameters

lpInBuffer

A pointer to a DWORD (32bits unsigned 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 *ixpcsc.vxd.*

nInBufferSize

Should be 4 (but is not tested).

lpOutBuffer

Not used.

nOutBufferSize

Not used.

lpBytesReturned

Not used.

Return Value

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

Comments

The functions in the *ixpcsc.vxd* use reasonable timeout values when communicating with the smart card reader. The *iccom.vxd* normally detects end of transmissions based on message contents, so the send/receive procedures should rarely be timed out at all. 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. Such a lengthy procedure, the PIN entry, is already taken care of in the *ixpcsc.vxd* (when that function is ordered by IX_IOCTL_DO_CARD_COMMAND), so no need for adjusting timeout then.

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 6 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_IOCTL_ICCOM_ESCAPE_FUNC (Value 0x312714)

Send an ”escape comm function” to the serial driver iccom.vxd to carry out an extended function.

<u>Parameters</u>

lpInBuffer

A pointer to a DWORD (32bits unsigned integer) containing the function number to carry out.

nInBufferSize

Should be 4 (but is not tested).

lpOutBuffer

Not used.

nOutBufferSize

Not used.

lpBytesReturned

Not used.

<u>Return Value</u>

The value SCARD_S_SUCCESS (=0) if the extended function returned TRUE. Else an error code is returned, reflecting a FALSE return from the extended function.

<u>Comments</u>

The extended functions of a serial port driver carry out functions like setting/resetting the DTR or RTS signals. The functions are described in Microsoft Windows documentation of communication functions.For *iccom.vxd* there are additional functions to enquire whether the modem/IC-card reader is in the ”on-line data mode” or in command mode. See chapter 6 above for a description of these additional functions. (Function no. 503 is not allowed here since it would disturb the PC/SC drivers own state monitoring)

IX_IOCTL_OPEN_LOG (Value 0x312718)

Opens a log-file named ixpcslog.txt in the current directory. All future transmission to the card reader will be logged. (Only available in the debug version).

Parameters

None.

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

See under ”Troubleshooting” above.

IX_IOCTL_CLOSE_LOG (Value 0x31271c)

Closes a log-file previously opened by IX_IOCTL_OPEN_LOG. (Only available in the debug version).

Parameters

None.

Return Value

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

Comments

See under ”Troubleshooting” above.

IX_IOCTL_PCSC_CLOSE (Value 0x312724)

Close the PC/SC's access to the card reader. Will free the COM-port used for it.

Parameters

lpInBuffer

Not used.

nInBufferSize

Not used.

lpOutBuffer

A pointer to a byte where to put the COM-port number in use (or rather, the port that just became free). If the COM-port number reported is 0, the COM-port has never been opened (and PC/SC cannot operate the card reader).

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 close the ICC-channel port. If the non-ICC channel is not opened by another application, the associated physical COM-port (COM1-4) will also be closed. This enables other applications, along with DOS programs, to directly access that port. The PC/SC will still have contact with the driver *ixpcsc.vxd,* 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.

After this command, it may take a second before the COM-port is closed and free to use.

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. Will acquire the COM-port again.

Parameters

lpInBuffer

Not used.

nInBufferSize

Not used.

lpOutBuffer

A pointer to a byte where to put the COM-port number in use. If the COM-port number reported is 0, the COM-port has never been opened (and PC/SC cannot operate the card reader).

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 ICC-channel port again after having been closed by the IX_IOCTL_PCSC_CLOSE function. If the physical COM-port (COM1-4) connected to the modem/card reader has not been closed before this, the function will fail. However, there is no reporting of the success or failure of the function. Therefore, an application should check that the COM-port (COM1-4) is closed (by trying to open it and then close it again) before issuing this IOCTL command. The correct port number (1-4) can be retrieved (if not remembered after the IX_IOCTL_PCSC_CLOSE command) by the use of IX_IOCTL_GET_PORTNUMBER.

IX_IOCTL_GET_PORTNUMBER (Value 0x31272c)

Retrieve the port number, 1-4, for the COM-port to which the modem/card reader is connected (and in use by the PC/SC driver).

Parameters

lpInBuffer

Not used.

nInBufferSize

Not used.

lpOutBuffer

A pointer to a byte where to put the COM-port number in use. If the COM-port number reported is 0, the COM-port has never been opened (and PC/SC cannot operate the card reader).

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 IOCTL will report the actual COM-port number even if that port is released from PC/SC by the IX_IOCTL_PCSC_CLOSE function. No other action is performed.

10. Using the CT-API high level API: *ixctapi.dll*

General

The German specification CT-API for a standard application independent API is implemented by a library *ixctapi.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.dtag.de/angebot/telesec/produkte/anwendung/ct-api/right.htm
http://www.darmstadt.gmd.de/~eckstein/CT/mkt.html
http://www.linuxnet.com/ctapi/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:

INS-code (hex)	Name
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 keypad on (some of) the card readers, as well as the display, is reached only through the propriety function IX_CTAPI_Do_Card_Command (see below).

dll:s

The CT-API for this reader is implemented by a 32-bit dynamic library file called *ixctapi.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 *iccapi.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 chapter 8. The driver components *iccom.vxd/iccom.sys* must be used for the CT-API, as for the other API:s.

The CT-API also specifies ”well known identifiers” for the function calls. Thus, the function calls **CT_init**, **CT_data** and **CT_close** may in this case be called under the names **CTSEIXD_init**, **CTSEIXD_data** and **CTSEIXD_close** respectively. These names come into use when an application is working with more than one reader type, to avoid

conflicting function names. The CTSEIXD_… functions are contained in a dll called *ctixd.dll*. This makes use of the other dll:s mentioned, so all three dll:s must be present on directories that the application will find. The function prototype (parameters, return value) for CTSEIXD_… is exactly the same as their corresponding CT_… function. 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 not installed with the installation files that comes with the modem/card reader. The dll:s, as well as .lib-files and a short sample file, may be obtained in the developers kit from Intertex Data AB.

The *ixctapi.dll* or *ctixd.dll* may be linked to either using load-time dynamic linking or run-time dynamic linking, as was described for the *iccapi.dll* (chapter 8). 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 (the code fragments in this chapter may be copied from the sample file *ctsample.c* on the developers kit diskette):

```
extern char __stdcall CTSEIXD_init(WORD ctn,
                       WORD pn);
extern char __stdcall CTSEIXD_data(WORD ctn,
                       BYTE *dad,
                       BYTE *sad,
                       WORD lenc,
                       BYTE *command,
                       WORD *lenr,
                       BYTE *response);
extern char __stdcall CTSEIXD_close(WORD ctn);
```

Using these declarations, and specifying *ctixd.lib* on the linker command line, enables the application program to call the API functions directly. If using the names CT_init, CT_data and CT_close, the *ixctapi.lib* should be linked instead (and the names changed accordingly in the declarations above).

For run-time dynamic linking, the application must explicitly load the dll-file and resolve the adresses 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                CTSEIXD_init;
extern CTAPIPROCdata                CTSEIXD_data;
extern CTAPIPROCclose               CTSEIXD_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     CTSEIXD_init;      // Function pointer
CTAPIPROCdata     CTSEIXD_data;      // Function pointer
CTAPIPROCclose    CTSEIXD_close;     // Function pointer

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

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

  CTSEIXD_init = (CTAPIPROCinit) GetProcAddress(
                CTAPILib,"CTSEIXD_init");
  CTSEIXD_data = (CTAPIPROCdata) GetProcAddress(
                CTAPILib,"CTSEIXD_data");
  CTSEIXD_close = (CTAPIPROCclose) GetProcAddress(
                CTAPILib,"CTSEIXD_close");

  if((CTSEIXD_init==0)||
         (CTSEIXD_data==0)||
         (CTSEIXD_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 the error "Could not find ctixd.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 = CTSEIXD_init(1,PortNumber);
  Res = CTSEIXD_data(1,&dad,&sad,6,
                       command,&received,inbuffer);
  Res = CTSEIXD_close(1);
```

(Switch to CT_… names instead of CTSEIXD_… where appropriate in the sample code above, if the standard names may be used)

Specific proprietary functions

The application programmer may want to reach functions in the smartcard modem 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 adress), one for each of four functions defined. The following dad-values are used:

dad-value	Symbolic names (not real function names)
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 chapter 8). The functions are described in the reference below.

It is also possible to use the iccapi functions (chapter 8) directly, along with the CT-API. In that case, the *iccapi.dll* must also be linked to, either at load-time or at run-time as described in chapter 8.

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 = CTSEIXD_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 end of chapter 8.
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. Entering of PIN-code, 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. The checksum byte (LRC) and the DLE-framing are being taken care of and should not be included. Likewise, the response contains the full message from the card reader, including the command byte, the parameter byte and the data field, but not including the LRC checksum or the DLE frames.

IX_CTAPI_Get_Channel_Handle

Retrieve the handle to the virtual COM-port used for smart card access

Call structure (example):

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

dad = 0x0b;        // Marks IX_CTAPI_Get_Channel_Handle
err = CTSEIXD_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 communication function calls (such as WriteFile, GetCommState etc.) directly to the virtual COM-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_HANDLE_VALUE, a Win32 constant).

Comments

This function could be used if the application programmer wants to operate the COM-port in a way not supported by the smart card API. By retrieving the port handle, the full Win32 communication API may be used. The (virtual) COM-port may be written to or read from, the DCB (Device Control Block) including baudrate, parity etc. may be setup. A certain care is recommended when using this direct mode, the programmer should study the chapter 6.

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 CTSEIXD_close library routine. The application does not own the handle, it simply ”borrows” it from the *ctixd.dll/ixctapi.dll* libraries.

For an example of use, see function ICC_Get_Channel_Handle in chapter 8.

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 = CTSEIXD_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 *ctixd.dll/ixctapi.dll* use reasonable timeout values when communicating with the smart card reader. The *iccom.vxd/iccom.sys* normally detects end of transmissions based on message contents, so the send/receive procedures should rarely be timed out at all. 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. Such a lengthy procedure, the PIN entry, is already taken care of in the API (when that function is ordered by IX_CTAPI_Do_Card_Command), so no need for adjusting timeout then.

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 6 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 *iccom* driver, 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) CTSEIXD_data(1,&dad,&sad,
                  4, (BYTE *) Escapecode, 0 , 0);
```

Parameters

Escapecode

Any of following values:

500	Test if in on-line data mode
501	Test if in off-line command mode
503	Test if card state (inserted/removed) has changed since last call of this function.
506	Test if non-smartcard channel (COM5-COM8) has been opened
507	Test if smartcard channel (ICCPORTx) has been opened
CLRDTR	Standard Windows code, see Windows documentation
SETDTR	”
CLRRTS	”
SETRTS	”
RESETDEV	”

(For Windows 95/98 there are other standard escape codes defined)

Return Value

1	Function returned TRUE
0	Function returned FALSE

For Escapecodes 500-507 this reflects the output of the operation (for example: TRUE if the card state has changed).

Comments

This function may be used instead of the standard Win32 API **EscapeCommFunction**. For Windows 95/98 it functions pretty much the same, except that the handle to the COM-port does not have to be specified (it is already known by the *ctixd.dll/ixctapi.dll* after a CTSEIXD_init call.).

For Windows NT, it allows the use of the specific escape codes 500-507, which would otherwise not be allowed. These codes are converted into IOCTL-codes by the *ctixd.dll/ixctapi.dll*, before the actual driver is called. Applications that have to be compatible with both Windows 95/98 and Windows NT therefore have to use this function instead of **EscapeCommFunction**.

11. Using the OCF Java based API

General

OCF stands for Open Card Framework, an object-oriented software architecture for smartcard access. It is specified by a consortium where IBM, Bull, Sun, Gemplus and Schlumberger have played the main parts. All information about OCF may be found on `http://www.opencard.org/`. The necessary documentation and binaries can be downloaded from that web-site. The Java program development tools may also be downloaded from the web (`http://java.sun.com/products/jdk/`).

Though OCF, like Java, is designed to be platform independent, the component described here will only run on Windows 95/98/NT4.0 platforms. This is because the handling of the virtual COM-port is needed, implemented in the *iccom.vxd (iccom.sys)* drivers as described earlier.

The versions that have been used here is 1.1.1 for the OCF binaries and 1.1.8 of the Java Delopment Kit.

The classes

In OCF, the card reader interface is performed by a class named CardTerminal. This class is instantiated not by the application directly, but by the OCF framework itself through a ”factory” object, an object that creates the CardTerminal instance. The factory class is called CardTerminalFactory (for a full understanding of this it is recommended to study the OpenCard Framework Programmer’s Guide).

An OCF component called CardTerminalRegistry keeps tracks of terminals (=card readers) that it knows of. When the framework starts, that is when an application using the OCF starts, the CardTerminalRegistry calls the factory class to instantiate the CardTerminal class. The name of the factory class for our particular terminal is fetched from the file *opencard.properties*. This file must be set up in the end-users environment to contain the settings for this card reader. In practise only one line of text is crucial. Apart from that, no other settings specific for our reader (the ”Smartix” reader) will be needed in the application’s Java code.

The CardTerminal class for our reader is designed so that a smartcard is powered up and negotiated/activated immediately when it is inserted. This may save time in the application, so that when the card is requested by (for example)

```
Card = SmartCard.waitForCard(cardrequest);
```

the wait is immediately satisfied if the card has previously been inserted. The CardID (ATR-string) is then ready to be examined and the card may be talked to.

Like the CT-API described in the previous chapter, the OCF works with ISO7816-4 APDU:s (application protocol data units). (In fact, the OCF terminal described here uses the CT-API for much of its functionality). The APDU 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.

The library and the property file

The necessary classes (named `se.intertex.opencard.smartix.TheCardTerminalFactory` and `se.intertex.opencard.smartix.TheCardTerminal` according to the OCF convention) are collected in a .jar file (a Java library file) called *smartix.jar*. The terminal name will be ”Smartix1” and the terminal type will be ”Smartix”. These names will not have to be used directly in the application code but only in the *opencard.properties* file (and possibly in a call for a specific sendTerminalCommand, see below). The *smartix.jar* file must be accessible to the application, normally by setting up the CLASSPATH environment variable to that file (see the Java Development Kit documentation).

Along with each OCF application there must be a file named *opencard.properties* that defines what card terminals (=readers), smartcard services etc. that the OCF framework needs to know and access. This file may reside in the current directory (but also in the Java library directory and other places, see the OpenCard Framework Programmer’s Guide). For our reader, it is necessary to include any of the following entries:

```
OpenCard.terminals =
se.intertex.opencard.smartix.TheCardTerminalFactory|Smartix1|Smartix|any
```
or
```
OpenCard.terminals =
se.intertex.opencard.smartix.TheCardTerminalFactory|Smartix1|Smartix|COM1
```
or
```
OpenCard.terminals =
se.intertex.opencard.smartix.TheCardTerminalFactory|Smartix1|Smartix|COM2
```
or
```
OpenCard.terminals =
se.intertex.opencard.smartix.TheCardTerminalFactory|Smartix1|Smartix|COM3
```
or
```
OpenCard.terminals =
se.intertex.opencard.smartix.TheCardTerminalFactory|Smartix1|Smartix|COM4
```

The last field of the property defines the COM-port used for the card reader. If the COM-port is not known, the ”any” may be used, in which case the card terminal will be searched for among the four possible ports, when the OCF framework starts (normally when the application starts). This may take 1-2 seconds. If the COM-port is known, it is better to use the specified COM-port number, thus avoiding the extra search delay. In any case, the CardTerminal instantiation will of course fail if the card reader is not switched on or connected at all.

A sample *opencard.properties* file is included on the Smartcard Developers Kit diskette and is downloadable from our web-site, along with the binaries (see below).

The dll:s

For the card terminal class to work, three .dll-files are needed in Windows. These are the *ixctapi.dll* (the .dll file already mentioned in the CT-API chapter above), the *iccapi.dll* (the specific API dll described in chapter 8) and the *ixocfcta.dll*. The latter is a ”bridge”-dll to bring the other two Windows dll:s accessible to the Java environment. All three dll:s must be placed in a directory accessible to the application, either in the Windows, system, current or a pathed directory.

Distribution

The OCF CardTerminal components described here may freely be used by any programmer or end-user. They are included on the Intertex Smartcard Developers Kit. The parts necessary for the end-user (the .jar-file, the *opencard.properties* sample file, the three dll:s and a readme file) may also be downloaded through our web-site `http://www.intertex.se`.

To get it all work, it is of course necessary to have the card reader (with the included modem) properly installed by the diskettes that comes with the product (the virtual port, *iccom.vxd* etc. are thereby installed).

The specific reader commands (the sendTerminalCommand function)

The CardTerminal class and the OCF takes care of all "normal" card operations, such as powering-on/off, status detection, transmission of data etc. without the application actually having to access the CardTerminal object explicitly. Instead, all accesses to the reader and the smartcard is done by code using high-level classes, like CardServices, CardRequest, CardID and similar. Nevertheless, there are some functions in the card reader that cannot be used through these standardised upper layers of the OCF API. Examples are functions like controlling the display of the modem/card reader (a three digit display, not conforming to a standard OCF optional display), getting information about the non-ICC channel of the virtual port being opened etc. To make this reader-specific operations available, the OCF defines an optional interface, a method of the CardTerminal class called **sendTerminalCommand**. In this way, a specific command may be sent to the terminal without going through the CardService layer. The input and the output from the method are byte arrays. Normally, this data is sent directly to the card reader, the commands and responses are described in the document "Modem integrated IC card reader: Protocol between modem/reader and computer". Essentially, this function does the same as "ICC_Do_Card_Command" in *iccapi.dll*, described in chapter 8. For some specific functions accessing the drivers and dll:s, rather than the reader, the first byte is interpreted specially, see the reference below.

In order to call the **sendTerminalCommand**, the terminal object must be acquired. This can be done according to following example:

```
import opencard.core.terminal.CardTerminalRegistry;

// Example of use of sendTerminalCommand: Toggle "Crd"/"In" on display
// at a frequency of 0.8 seconds

// The CardTerminalRegistry
    CardTerminalRegistry ctr;
// The card terminal in use
    se.intertex.opencard.smartix.TheCardTerminal term;
// The byte arrays used
    byte[] command = {61,8};        // The command (no data or LRC)
    byte[] result1;                 // This will carry the respons,
                                    // in this case probably {61,126}


    ctr = CardTerminalRegistry.getRegistry();
    term = (se.intertex.opencard.smartix.TheCardTerminal)
                        ctr.cardTerminalForName("Smartix1");
    if(term != null)
    {
        try
        {
            result1 = term.sendTerminalCommand(command);
```

```
            // Check the result, if desired.
        }
        catch(CardTerminalException e)
        {
            // Failed to send the command to the reader.
            // Report the problem.
        }
    }
    else
    {
        // Could not get the terminal object, maybe it was not initiated.
    }
```

The specific reader command reference

The specific calls of **sendTerminalCommand** are listed in the following pages.

EscapeCommFunction: command[0] = 128

Sends a specific code to the *iccom* driver, performing various functions or enquiries.

Call structure (example):

```
byte[] Escape506 = {(byte)128,(byte)0xfa,1,0,0};
                                          // 0x01fa = 506
byte[] result;
    try
    {
         result = term.sendTerminalCommand(Escape506);
         if(result[0] == 1); // Do something; ”true” code
         else;   // Do something; ”false” code
    }
    catch(CardTerminalException e){ /* Command failed */ }
```

Parameters

Byte[0]

The code 128, meaning ”Submit an EscapeCommFunction to the driver”

Byte[1]-Byte[4]

Any of following values, encoded with least significant byte in byte[0]:

500	Test if in on-line data mode
501	Test if in off-line command mode
(503	Test if card state changed; this will not work in OCF due to the frameworks own handling of card state changes)
506	Test if non-smartcard channel (COM5-COM8) has been opened
507	Test if smartcard channel (ICCPORTx) has been opened (will almost always be true in OCF)
CLRDTR	Standard Windows code, see Windows documentation
SETDTR	”
CLRRTS	”
SETRTS	”
RESETDEV	”

(For Windows 95/98 there are other standard escape codes defined)

Return Value, an array of one byte:

1	Function returned TRUE
0	Function returned FALSE

For Escapecodes 500-507 this reflects the output of the operation (for example: TRUE if the non-ICC channel is opened).

Comments

This method replaces the standard Win32 API **EscapeCommFunction** which is not accessible in the Java environment. See also the Windows documentation for the Win32 API function **EscapeCommFunction**. (See also chapter 6).

Adjust Timeouts: command[0] = 129

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

Call structure (example):

```
byte[] AdjTimo = {(byte)129,(byte)0x10,(byte)0x27,0,0};
                                    // 0x2710 = 10000 ms.
byte[] result;
    try
    {
         result = term.sendTerminalCommand(AdjTimo);
         // No need to test the result, will always be 0.
    }
    catch(CardTerminalException e){ /* Command failed */ }
```

Parameters

Byte[0]

The code 129, meaning ” change the receive timeout”

Byte[1]-Byte[4]

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. Encoded with least significant byte in byte[0]:

Return Value, an array of one byte:

Always null.

Comments

The functions in the API (the dll:s called by the CardTerminal object) use reasonable timeout values when communicating with the smart card reader. The *iccom.vxd/iccom.sys* normally detects end of transmissions based on message contents, so the send/receive procedures should rarely be timed out at all. 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. Such a lengthy procedure, the PIN entry, is already taken care of in the API (when that function is ordered by ”Do Card Command” as below), so no need for adjusting timeout then.

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 6 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.

Only the receive timeouts are affected by this function. 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.

Do Card Command: command[0] = command byte according to protocol description

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

Call structure (example):

```
byte[] command = {61,8};  // Toggle display of the modem
byte[] result;
    try
    {
         result = term.sendTerminalCommand(command);
         if(result[1] == 126)
         {
                  // The reader sent an "OK" response.
                  // Do something;
         }
         else
         {
                  // Reader responded with error code
                  // Do something; "false" code
         }
    }
    catch(CardTerminalException e)
    {
         // Fundamental error (e.g. no contact with reader)
    }
```

Parameters

The byte array as a whole is taken for a command to the reader, as specified in the protocol description. It is assumed that the 1:st byte is neither 128 nor 129 but is a byte according to the command byte in the reader protocol description.

Returns:

A byte array of a length according to the reader protocol description.

Comments

This function enables the application programmer to perform all the card reader functions as specified in the protocol description. Entering of PIN-code, controlling the display, getting card insertion state etc. are examples of operations that are performed by this function. See the protocol description document for a list of possible operations. In this direct mode, the command byte, the parameter byte and the data field (using the protocol description vocabulary) all have to be setup in the byte array. The checksum byte (LRC) and the DLE-framing are being taken care of and should not be included. Likewise, the response contains the full message from the card reader, including the command byte, the parameter byte and the data field, but not including the LRC checksum or the DLE frames.