NAME

Chipcard::PCSC − Smart card reader interface library

SYNOPSIS

```
 my $hContext = new Chipcard::PCSC();

 @ReadersList = $hContext−>ListReaders ();

 $hContext−>GetStatusChange(\@readers_states, $timeout);

 $apdu = Chipcard::PCSC::array_to_ascii(@apdu);

 @apdu = Chipcard::PCSC::ascii_to_array($apdu);

 $hContext = undef;
```

DESCRIPTION

The PCSC module implements the Chipcard::PCSC class. Objects of this class are used to communicate with the PCSC-lite daemon (see ***pcscd** (1)* for more information).

PC/SC represents an abstraction layer to smart card readers. It provides a communication layer with a wide variety of smart card readers through a standardized API.

A PCSC object can be used to communicate with more than one reader through Chipcard::PCSC::Card objects. Please read Chipcard::PCSC::Card for extended information on how to talk to a smart card reader.

A PCSC object uses the following property: `$pcsc_object−>{hContext}` the context returned by the pcsc library

CONSTRUCTORS

The following methods can be used to construct a PCSC object:

- **`$hContext` = new Chipcard::PCSC($scope, `$remote_host`);**
  - `$scope` is the scope of the connection to the PC/SC daemon. It can be any of the following:

    ```
     $Chipcard::PCSC::SCARD_SCOPE_USER     (not used by PCSClite);
     $Chipcard::PCSC::SCARD_SCOPE_TERMINAL (not used by PCSClite);
     $Chipcard::PCSC::SCARD_SCOPE_SYSTEM   Services on the local machine;
     $Chipcard::PCSC::SCARD_SCOPE_GLOBAL   Services on a remote host.
    ```
  - `$remote_host` is the host name of the remote machine to contact. It is only used when `$scope` is equal to `$Chipcard::PCSC::SCARD_SCOPE_GLOBAL`. A null value means *localhost*.
- **`$hContext` = new Chipcard::PCSC($scope);**

  This method is equivalent to:

  ```
   $hContext = new Chipcard::PCSC($scope, 0);
  ```
- **`$hContext` = new Chipcard::PCSC();**

  This method is equivalent to:

  ```
   $hContext = new Chipcard::PCSC($Chipcard::PCSC::SCARD_SCOPE_SYSTEM, 0);
  ```

CONSTRUCTION FAILURE

Chipcard::PCSC constructors return an `undef` value when the object can not be created. `$Chipcard::PCSC::errno` can be used to get more information about the error. (See section ‘‘ERROR HANDLING’’ below for more information)

Chipcard::PCSC METHODS

Here is a list of all the methods that can be used with a PCSC object.

- **hContext−>ListReaders(`$group`);**

  This method returns the available readers in the given `$group`. If omitted, `$group` defaults to a null value meaning ‘‘all groups’’. Please note that as of this writing, `$group` can safely be omitted as it is not used by PCSClite.

  The return value upon successful completion is an array of strings: one string by available reader. If an error occurred, the undef value is returned and `$Chipcard::PCSC::errno` should be used to get more information about the error. (See section ‘‘ERROR HANDLING’’ below for more information). The following example describes the use of ListReaders:

  ```
  $hContext = new Chipcard::PCSC();
  die ("Can't create the PCSC object: $Chipcard::PCSC::errno\n")
         unless (defined $hContext);

  @ReadersList = $hContext−>ListReaders ();
  die ("Can't get readers' list: $Chipcard::PCSC::errno\n")
         unless (defined($ReadersList[0]));

  $, = "\n  ";
  print @ReadersList . "\n";
  ```

- **`$hContext`−>GetStatusChange(\@readers_states, `$timeout`);**

  The method `$hContext−>GetStatusChange(\@readers_states, $timeout)` uses a reference to a list of hashes.

  ```
  # create the list or readers to watch
  map { push @readers_states, ({'reader_name'=>"$_"}) } @ReadersList;

  @StatusResult = $hContext−>GetStatusChange(\@readers_states);
  ```

  The keys of the hash are: ’reader_name’, ’current_state’, ’event_state’ and ’ATR’.

  To detect a status change you have to first get the status and then copy the ’event_state’ in the ’current_state’. The method will return when both states are different or a timeout occurs.

  ```
  @StatusResult = $hContext−>GetStatusChange(\@readers_states);
  foreach $reader (@readers_states)
  {
    $reader−>{current_state} = $reader−>{event_state};
  }
  @StatusResult = $hContext−>GetStatusChange(\@readers_states);
  ```

- **`$hContext`−>GetStatusChange(\@readers_states);**

  This method is equivalent to:

  ```
  $hContext−>GetStatusChange(\@readers_states, 0xFFFFFFFF);
  ```

  The timeout is set to infinite.

- **`$apdu_ref` = Chipcard::PCSC::ascii_to_array($apdu);**

  The method `Chipcard::PCSC::Card::Transmit()` uses references to arrays as in and out parameters. The `Chipcard::PCSC::ascii_to_array()` is used to transform an APDU in ASCII format to a reference to an array in the good format.

  Example:

  ```
  $SendData = Chipcard::PCSC::ascii_to_array("00 A4 01 00 02 01 00");
  ```

- **`$apdu` = Chipcard::PCSC::array_to_ascii($apdu_ref);**

This method is used to convert the result of a `Chipcard::PCSC::Card::Transmit()` into ASCII format.

Example:

```
 $RecvData = $hCard->Transmit($SendData);
 print Chipcard::PCSC::array_to_ascii($RecvData);
```

ERROR HANDLING

All functions from PCSC objects save the return value in a global variable called `$Chipcard::PCSC::errno`. This variable therefore holds the latest status of PCSC.

It is a double-typed magical variable that behaves just like `$!`. This means that it both holds a numerical value describing the error and the corresponding string. The numerical value may change from a system to another as it depends on the PCSC library...

Here is a small example of how to use it:

```
 $hContext = new Chipcard::PCSC();
 die ("Can't create the PCSC object: $Chipcard::PCSC::errno\n")
     unless (defined $hContext);
```

In case the last call was successful, `$Chipcard::PCSC::errno` contains the `SCARD_S_SUCCESS` status. Here is a list of all possible error codes. They are defined as read-only variables with in the PCSC module:

```
 $Chipcard::PCSC::SCARD_S_SUCCESS
 $Chipcard::PCSC::SCARD_E_CANCELLED
 $Chipcard::PCSC::SCARD_E_CANT_DISPOSE
 $Chipcard::PCSC::SCARD_E_CARD_UNSUPPORTED
 $Chipcard::PCSC::SCARD_E_DUPLICATE_READER
 $Chipcard::PCSC::SCARD_E_INSUFFICIENT_BUFFER
 $Chipcard::PCSC::SCARD_E_INVALID_ATR
 $Chipcard::PCSC::SCARD_E_INVALID_HANDLE
 $Chipcard::PCSC::SCARD_E_INVALID_PARAMETER
 $Chipcard::PCSC::SCARD_E_INVALID_TARGET
 $Chipcard::PCSC::SCARD_E_INVALID_VALUE
 $Chipcard::PCSC::SCARD_E_NO_MEMORY
 $Chipcard::PCSC::SCARD_E_NO_SERVICE
 $Chipcard::PCSC::SCARD_E_NO_SMARTCARD
 $Chipcard::PCSC::SCARD_E_NOT_READY
 $Chipcard::PCSC::SCARD_E_NOT_TRANSACTED
 $Chipcard::PCSC::SCARD_E_PCI_TOO_SMALL
 $Chipcard::PCSC::SCARD_E_PROTO_MISMATCH
 $Chipcard::PCSC::SCARD_E_READER_UNAVAILABLE
 $Chipcard::PCSC::SCARD_E_READER_UNSUPPORTED
 $Chipcard::PCSC::SCARD_E_SERVICE_STOPPED
 $Chipcard::PCSC::SCARD_E_SHARING_VIOLATION
 $Chipcard::PCSC::SCARD_E_SYSTEM_CANCELLED
 $Chipcard::PCSC::SCARD_E_TIMEOUT
 $Chipcard::PCSC::SCARD_E_UNKNOWN_CARD
 $Chipcard::PCSC::SCARD_E_UNKNOWN_READER
 $Chipcard::PCSC::SCARD_E_UNSUPPORTED_FEATURE

 $Chipcard::PCSC::SCARD_W_REMOVED_CARD
 $Chipcard::PCSC::SCARD_W_RESET_CARD
 $Chipcard::PCSC::SCARD_W_UNPOWERED_CARD
 $Chipcard::PCSC::SCARD_W_UNRESPONSIVE_CARD
 $Chipcard::PCSC::SCARD_W_UNSUPPORTED_CARD
```

PCSClite users will also be able to use the following (PCSClite specific) codes:

```
 $Chipcard::PCSC::SCARD_INSERTED
 $Chipcard::PCSC::SCARD_REMOVED
 $Chipcard::PCSC::SCARD_RESET
 $Chipcard::PCSC::SCARD_SCOPE_GLOBAL
```

In addition, the wrapper defines:

```
 $Chipcard::PCSC::SCARD_P_ALREADY_CONNECTED
 $Chipcard::PCSC::SCARD_P_NOT_CONNECTED
```

SEE ALSO

pcscd *(1)* manpage has useful information about PC/SC lite. Chipcard::PCSC::Card manpage gives information about how to communicate with a reader and the smart card inside it.

COPYRIGHT

AUTHORS / ACKNOWLEDGEMENT

```
 Lionel VICTOR <lionel.victor@unforgettable.com>
               <lionel.victor@free.fr>

 Ludovic ROUSSEAU <ludovic.rousseau@free.fr>
```