Introduction

In this followup to the article Accessing the Belgian Identity Card from VB.NET you will see how you can monitor a smartcard reader. Contained within is the necessary code to detect two simple events, namely when a card was inserted into the reader and when it was ejected.

The material in this article doesn't build upon the source code of the previous one. If you haven't read the first part of this series you can download the source code here. It might be good to explore it first. If you just want to find out how to monitor a smart card reader then this article alone suffices.

Unlike the first article I've choosen C# for this project, but you can easily convert the code back to VB.NET. Should you require assistence you can find decent convertors on the following websites:

- Convert C# to VB.NET
- Convert VB.NET to C#

In an attempt not to be so verbose this article will mainly consist out of source code with brief comments along the way explaining its purpose. So without further ado, let's get started...

Table Of Contents

The Winscard Dynamic Link Library

For working with smartcard readers Microsoft already provides a rich set of API's contained in the Winscard Dynamic Link Library. You'll need to import four functions from this library with the Platform Invocation interoperability mechanism (PInvoke). If you want to learn more about the smartcard API you can visit the following sections on MSDN:

- Smart Card Functions
- Smart Card Structures
- Smart Card Error Values

The following listing displays the code for importing the four necessary functions.

Listing 1 - Complete UnsafeNativeMethods.cs File

```
using System;
using System.Runtime.InteropServices;

namespace SmartcardLibrary
{
    internal enum ScopeOption
    {
        //User
        None = 0,
        Terminal = 1,
        System = 2
    }

    internal sealed partial class UnsafeNativeMethods
    {
         #region WinScard.DLL Imports

         [DllImport("WINSCARD.DLL", EntryPoint = "SCardEstablishContext", CharSet = CharSet.Unicode,
             SetLastError = true)]
         static internal extern uint EstablishContext(ScopeOption scope, IntPtr reserved1,
             IntPtr reserved2, ref SmartcardContextSafeHandle context);

         [DllImport("WINSCARD.DLL", EntryPoint = "SCardReleaseContext", CharSet = CharSet.Unicode,
             SetLastError = true)]
         static internal extern uint ReleaseContext(IntPtr context);

         [DllImport("WINSCARD.DLL", EntryPoint = "SCardListReaders", CharSet = CharSet.Unicode,
             SetLastError = true)]
         static internal extern uint ListReaders(SmartcardContextSafeHandle context, string groups,
             string readers, ref int size);

         [DllImport("WINSCARD.DLL", EntryPoint = "SCardGetStatusChange", CharSet = CharSet.Unicode,
             SetLastError = true)]
         static internal extern uint GetStatusChange([In(), Out()] SmartcardContextSafeHandle context,
             [In(), Out()] int timeout, [In(), Out()] ReaderState[] states, [In(), Out()] int count);

         #endregion
    }
}
```

This sealed class UnsafeNativeMethods is simular to the one used in the first article of this series. It imports the following four functions from the Winscard.dll library:

- SCardEstablishContext: This function establishes the resource manager context within which operations are performed. It creates a communication context to the PC/SC resource manager. This must be the first function called in a PC/SC application
- SCardReleaseContext: This function closes the previously established context. Call this function when you are done communicating with the PC/SC resource manager.
- SCardListReaders: Provides a list of readers while automatically eliminating duplicates.
- SCardGetStatusChange: Blocks execution until the current availability of the cards in a specific set of readers changes.

The function EstablishContext (SCardEstablishContext) takes one parameter of the enumerated type ScopeOption which is also declared in the UnsafeNativeMethods file. This parameter specifies the scope of the resource manager context. Just pass ScopeOption.System to perform operations within the domain of the system.

You can eliminate the hardcoded WINSCARD.DLL reference by introducing a constant. In the section Invoking The Middle of the first article this was done by introducing the static holder type GlobalConstants. Have a look at that section if you want to change this. For the sake of keeping this article shorter I eliminated this here.

Top of page

SafeHandle

If you were to compile now you would get a few errors because the above code references some types that you have not declared yet. The most import one is the SmartcardContextSafeHandle type.

The operating system handle (context) returned by the EstablishContext function is wrapped in a class derived from the SafeHandle class. Since this class is abstract you must implement a derived class. This class provides finalization of handle resources, preventing them from being reclaimed prematurely by garbage collection.

This was introduced in the .NET Framework 2.0. Before the release of that version all operating system handles could only be encapsulated in the IntPtr managed wrapper object.

With this SafeHandle descendant you be sure that your handle will be disposed of properly when it is no longer needed. Listing 2 Shows the entire code for this class.

Listing 2 - Complete SmartcardContextSafeHandle.cs File

```
using System;
using System.Security.Permissions;
using System.Runtime.InteropServices;

namespace SmartcardLibrary
{
    internal sealed class SmartcardContextSafeHandle : SafeHandle
    {
        public SmartcardContextSafeHandle()
            : base(IntPtr.Zero, true)
        {
        }

        //The default constructor will be called by P/Invoke smart
        //marshalling when returning MySafeHandle in a method call.
        public override bool IsInvalid
        {
            [SecurityPermission(SecurityAction.LinkDemand,
                UnmanagedCode = true)]
            get { return (this.handle == IntPtr.Zero); }
        }

        //We should not provide a finalizer. SafeHandle's critical
        //finalizer will call ReleaseHandle for us.
        protected override bool ReleaseHandle()
        {
            SmartcardErrorCode result =
                (SmartcardErrorCode)UnsafeNativeMethods.ReleaseContext(handle);
            return (result == SmartcardErrorCode.None);
        }
    }
}
```

Top of page

The ReaderState Structure

Another important type referenced by the UnsafeNativeMethods class is the ReaderState structure. This structure is used by functions for tracking smart cards within readers. You will need this later on when dealing with the GetStatusChange function to specifiy which card readers you want to monitor. If you want more detailled information about the ReaderState structure then have a look at the SCARD_READERSTATE page on MSDN. Listing 3 shows you the code for this structure.

Listing 3 - Complete ReaderState.cs File

```
using System;
using System.Runtime.InteropServices;

namespace SmartcardLibrary
{
    //Wraps the SCARD_READERSTATE structure of PC/SC.
    [StructLayout(LayoutKind.Sequential, CharSet = CharSet.Auto)]
    internal struct ReaderState
    {
        #region Member Fields
        //Points to the name of the reader being monitored.
        [MarshalAs(UnmanagedType.LPWStr)]
        private string _reader;
        //Not used by the smart card subsystem but is used by the application.
        private IntPtr _userData;
        //Current state of reader at time of call
        private CardState _currentState;
        //State of reader after state change
        private CardState _eventState;
        //Number of bytes in the returned ATR
        [MarshalAs(UnmanagedType.U4)]
        private int _attribute;
        //ATR of inserted card, with extra alignment bytes.
        [MarshalAs(UnmanagedType.ByValArray, SizeConst = 36)]
        private byte[] _rgbAtr;
        #endregion

        #region Methods
        public byte[] RGBAttribute()
        {
            return this._rgbAtr;
        }
        #endregion

        #region "Properties"
        public string Reader
        {
            get { return this._reader; }
            set { this._reader = value; }
        }

        public IntPtr UserData
        {
            get { return this._userData; }
            set { this._userData = value; }
        }

        public CardState CurrentState
        {
            get { return this._currentState; }
            set { this._currentState = value; }
        }

        public CardState EventState
        {
            get { return this._eventState; }
        }
        #endregion
    }
}
```

The ReaderState structure in its turn references a new enumerated type which you need to declare. For the sake of being complete the code for this type is shown below in listing 4. This new type named CardState will eventually be used to determine if a card was inserted in or ejected from the card reader.

Listing 4 - Complete CardState.cs File

```
using System;

namespace SmartcardLibrary
{
    //CardState enumeration, used by the PC/SC function SCardGetStatusChanged.
    internal enum CardState
    {
        //Unaware
        None = 0,
        Ignore = 1,
        Changed = 2,
        Unknown = 4,
        Unavailable = 8,
        Empty = 16,
        Present = 32,
        AttributeMatch = 64,
        Exclusive = 128,
        InUse = 256,
        Mute = 512,
        Unpowered = 1024
    }
}
```

Top of page

Error Handling

If you take a look at the UnSafeNativeMethods class you'll see that all of its methods return an 32-bit unsigned integer value. All of the functions contained within the Winscard.dll library return this type of return value. If the function succeeds it return zero (SCARD_S_SUCCESS), if not then it returns an error code. For more information see Smart Card Return Values or Smart Card Error Values.

There are alot of possible error codes. I've declared all of these in an enumerated type appropriatly called SmartcardErrorCode. Listing 5 shows you the entire code for this type.

Listing 5 - Complete SmartcardErrorCode.cs File

```
using System;
using System.ComponentModel;

namespace SmartcardLibrary
{
    internal enum SmartcardErrorCode : uint
    {
        [Description("Function succeeded")]
        None = 0,
        [Description("An internal consistency check failed.")]
        InternalError = 2148532225,
        [Description("The action was canceled by a SCardCancel request.")]
        Canceled = 2148532226,
        [Description("The supplied handle was invalid.")]
        InvalidHandle = 2148532227,
        [Description("One or more of the supplied parameters could not be properly interpreted.")]
        InvalidParameter = 2148532228,
        [Description("Registry startup information is missing or invalid.")]
        InvalidTarget = 2148532229,
        [Description("Not enough memory available to complete this command.")]
        NoMemory = 2148532230,
        [Description("An internal consistency timer has expired.")]
        WaitedTooLong = 2148532231,
        [Description("The data buffer to receive returned data is too small for the returned data.")]
        InsufficientBuffer = 2148532232,
        [Description("The specified reader name is not recognized.")]
        UnknownReader = 2148532233,
        [Description("The user-specified timeout value has expired.")]
        Timeout = 2148532234,
        [Description("The smart card cannot be accessed because of other connections outstanding.")]
        SharingViolation = 2148532235,
        [Description("The operation requires a smart card, but not smard card is currently in the device.")]
        NoSmartcard = 2148532236,
        [Description("The specified smart card name is not recognized.")]
        UnknownCard = 2148532237,
        [Description("The system could not dispose of the media in the requested manner.")]
        CannotDispose = 2148532238,
        [Description("The requested protocols are incompatible with the protocol currently in use with the smart card.")]
        ProtocolMismatch = 2148532239,
        [Description("The reader or smart card is not ready to accept commands.")]
        NotReady = 2148532240,
        [Description("One or more of the supplied parameters values could not be properly interpreted.")]
        InvalidValue = 2148532241,
        [Description("The action was canceled by the system, presumably to log off or shut down.")]
        SystemCanceled = 2148532242,
        [Description("An internal communications error has been detected.")]
        CommunicationError = 2148532243,
        [Description("An internal error has been detected, but the source is unknown.")]
        UnknownError = 2148532244,
        [Description("An ATR obtained from the registry is not a valid ATR string.")]
        InvalidAttribute = 2148532245,
        [Description("An attempt was made to end a non-existent transaction.")]
        NotTransacted = 2148532246,
        [Description("The specified reader is not currently available for use.")]
        ReaderUnavailable = 2148532248,
        [Description("The operation has been aborted to allow the server application to exit.")]
        Shutdown = 2148532248,
        [Description("The PCI Receive buffer was too small.")]
        PCITooSmall = 2148532249,
        [Description("The reader driver does not meet minimal requirements for support.")]
        ReaderUnsupported = 2148532250,
        [Description("The reader driver did not produce a unique reader name.")]
        DuplicateReader = 2148532251,
        [Description("The smart card does not meet minimal requirements for support.")]
        CardUnsupported = 2148532252,
        [Description("The Smart Card Resource Manager is not running.")]
        NoService = 2148532253,
        [Description("The Smart Card Resource Manager has shut down.")]
        ServiceStopped = 2148532254,
        [Description("An unexpected card error has occured.")]
        Unexpected = 2148532255,
        [Description("No primary provider can be found for the smart card.")]
        ICCInstallation = 2148532256,
        [Description("The requested order of object creation is not supported.")]
        ICCCreationOrder = 2148532257,
        [Description("This smart card does not support the requested feature.")]
        UnsupportedFeature = 2148532258,
        [Description("The identified directory does not exist in the smart card.")]
        DirectoryNotFound = 2148532259,
        [Description("The identified file does not exist in the smart card.")]
        FileNotFound = 2148532260,
        [Description("The supplied path does not represent a smart card directory.")]
        NoDirectory = 2148532261,
        [Description("The supplied path does not represent a smart card file.")]
        NoFile = 2148532262,
        [Description("Access is denied to this file.")]
        NoAccess = 2148532263,
        [Description("The smart card does not have enough memory to store the information.")]
        WriteTooMany = 2148532264,
        [Description("There was an error trying to set the smart card file object pointer.")]
        BadSeek = 2148532265,
        [Description("The supplied PIN is incorrect.")]
        InvalidPin = 2148532266,
        [Description("An unrecognized error code was returned from a layered component.")]
        UnknownResourceManagement = 2148532267,
        [Description("The requested certificate does not exist.")]
        NoSuchCertificate = 2148532268,
        [Description("The requested certificate could not be obtained.")]
```

```
        CertificateUnavailable = 2148532269,
        [Description("Cannot find a smart card reader.")]
        NoReadersAvailable = 2148532270,
        [Description("A communications error with the smart card has been detected. Retry the operation.")]
        CommunicationDataLast = 2148532271,
        [Description("The requested key container does not exist on the smart card.")]
        NoKeyContainer = 2148532272,
        [Description("The Smart Card Resource Manager is too busy to complete this operation.")]
        ServerTooBusy = 2148532273,
        [Description("The reader cannot communiate with the card, due to ATR string configuration conflicts.")]
        UnsupportedCard = 2148532325,
        [Description("The smart card is not responding to a reset.")]
        UnresponsiveCard = 2148532326,
        [Description("Power has been removed from the smart card, so that further communication is not possible.")]
        UnpoweredCard = 2148532327,
        [Description("The msart card has been reset, so any shared state information is invalid.")]
        ResetCard = 2148532328,
        [Description("The smart card has been removed, so further communication is not possible.")]
        RemovedCard = 2148532329,
        [Description("Access was denied because of a security violation.")]
        SecurityViolation = 2148532330,
        [Description("The card cannot be accessed because th wrong PIN was presented.")]
        WrongPin = 2148532331,
        [Description("The card cannot be accessed because the maximum number of PIN entry attempts has been reached.")]
        PinBlocked = 2148532332,
        [Description("The end of the smart card file has been reached.")]
        EndOfFile = 2148532333,
        [Description("The action was canceled by the user.")]
        CanceledByUser = 2148532334,
        [Description("No PIN was presented to the smart card.")]
        CardNotAuthenticated = 2148532335
    }
}
```

If the returned value equals SmartcardErrorCode.None then no error occured. This enumerated type like most of the other types in the SmartcardLibrary namespace is declared as internal. It is not exposed to other assemblies, if you wish to do so be sure to also change its type from uint to long for example. The uint type is not CLS compliant.

Top of page

SmartcardManager Singleton

Voila, now you are finally ready to start monitoring the smartcard readers connected to your system. Now you only need to write some code to do this. That's where the SmartcardManager class comes into play. This class implements the Singleton pattern so you can only initiate one instance of this type of object. This is done by calling the static method GetManager().

When an instance of the SmartcardManager class is created a list of the smartcard readers connected to your system is composed. The list is retrieved in the ListReaders() method which internally uses the ListReaders() method of the UnsafeNativeMethods class. Finally the constructor creates a new backgroundworker (thread) and uses the WaitChangeStatus() method as its DoWork event handler.

It is in this WaitChangeStatus() method that the list of smartcard readers is passed to the GetStatusChange() method of the UnsafeNativeMethod class. This causes the backgroundworker's thread execution to be blocked until a change occurs in the current availability of the cards in the list of monitored readers. A timeout can be set and I've choosen 1000 milliseconds as the default interval. When the execution of the thread resumes the status of readers is checked and if a card was inserted or ejected a message will be shown.

So that basically describes the purpose of the SmartcardManager class. You can find the entire code below in Listing 6. This code just shows you one possible way to monitor the smartcard readers. Feel free to improve on this trivial piece of code.

Listing 6 - Complete SmartcardManager.cs File

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;

namespace SmartcardLibrary
{
    internal enum SmartcardState
    {
        None = 0,
        Inserted = 1,
        Ejected = 2
    }

    public class SmartcardManager : IDisposable
    {
```

```
#region Member Fields

//Shared members are lazily initialized.
//.NET guarantees thread safety for shared initialization.
private static readonly SmartcardManager _instance = new SmartcardManager();

private SmartcardContextSafeHandle _context;
private SmartcardErrorCode _lastErrorCode;
private bool _disposed = false;
private ReaderState[] _states;
//A thread that watches for new smart cards.
private BackgroundWorker _worker;

#endregion

#region Methods

//Make the constructor private to hide it. This class adheres to the singleton pattern.
private SmartcardManager()
{
    //Create a new SafeHandle to store the smartcard context.
    this._context = new SmartcardContextSafeHandle();
    //Establish a context with the PC/SC resource manager.
    this.EstablishContext();

    //Compose a list of the card readers which are connected to the
    //system and which will be monitored.
    ArrayList availableReaders = this.ListReaders();
    this._states = new ReaderState[availableReaders.Count];
    for (int i = 0; i <= availableReaders.Count - 1; i++)
    {
        this._states[i].Reader = availableReaders[i].ToString();
    }

    //Start a background worker thread which monitors the specified
    //card readers.
    if ((availableReaders.Count > 0))
    {
        this._worker = new BackgroundWorker();
        this._worker.WorkerSupportsCancellation = true;
        this._worker.DoWork += WaitChangeStatus;
        this._worker.RunWorkerAsync();
    }
}

public static SmartcardManager GetManager()
{
    return _instance;
}

private bool EstablishContext()
{
    if ((this.HasContext))
    {
        return true;
    }
    this._lastErrorCode =
        (SmartcardErrorCode)UnsafeNativeMethods.EstablishContext(ScopeOption.System,
        IntPtr.Zero, IntPtr.Zero, ref this._context);
    return (this._lastErrorCode == SmartcardErrorCode.None);
}

private void WaitChangeStatus(object sender, DoWorkEventArgs e)
{
    while (!e.Cancel)
    {
        SmartcardErrorCode result;

        //Obtain a lock when we use the context pointer,
        //which may be modified in the Dispose() method.
        lock (this)
        {
            if (!this.HasContext)
            {
                return;
            }

            //This thread will be executed every 1000ms.
            //The thread also blocks for 1000ms, meaning
            //that the application may keep on running for
            //one extra second after the user has closed
            //the Main Form.
            result =
                (SmartcardErrorCode)UnsafeNativeMethods.GetStatusChange(
                this._context, 1000, this._states, this._states.Length);
        }

        if ((result == SmartcardErrorCode.Timeout))
        {
            // Time out has passed, but there is no new info. Just go on with the loop
            continue;
        }

        for (int i = 0; i <= this._states.Length - 1; i++)
        {
            //Check if the state changed from the last time.
            if ((this._states[i].EventState & CardState.Changed) == CardState.Changed)
            {
                //Check what changed.
                SmartcardState state = SmartcardState.None;
                if ((this._states[i].EventState & CardState.Present) == CardState.Present
                    && (this._states[i].CurrentState & CardState.Present) != CardState.Present)
                {
                    //The card was inserted.
                    state = SmartcardState.Inserted;
                }
                else if ((this._states[i].EventState & CardState.Empty) == CardState.Empty
                    && (this._states[i].CurrentState & CardState.Empty) != CardState.Empty)
                {
                    //The card was ejected.
                    state = SmartcardState.Ejected;
                }
                if (state != SmartcardState.None && this._states[i].CurrentState != CardState.None)
                {
                    switch(state)
                    {
                        case SmartcardState.Inserted:
                        {
                            MessageBox.Show("Card inserted");
                            break;
                        }
                        case SmartcardState.Ejected:
                        {
                            MessageBox.Show("Card ejected");
                            break;
                        }
                        default:
                        {
                            MessageBox.Show("Some other state...");
                            break;
                        }
                    }
                }
                //Update the current state for the next time they are checked.
                this._states[i].CurrentState = this._states[i].EventState;
            }
        }
    }
}

private int GetReaderListBufferSize()
{
    if ((this._context.IsInvalid))
    {
        return 0;
    }
    int result = 0;
    this._lastErrorCode =
        (SmartcardErrorCode)UnsafeNativeMethods.ListReaders(
        this._context, null, null, ref result);
    return result;
}

public ArrayList ListReaders()
{
    ArrayList result = new ArrayList();

    //Make sure a context has been established before
    //retrieving the list of smartcard readers.
    if (this.EstablishContext())
    {
        //Ask for the size of the buffer first.
        int size = this.GetReaderListBufferSize();

        //Allocate a string of the proper size in which
        //to store the list of smartcard readers.
        string readerList = new string('', size);
        //Retrieve the list of smartcard readers.
        this._lastErrorCode =
            (SmartcardErrorCode)UnsafeNativeMethods.ListReaders(this._context,
            null, readerList, ref size);
        if ((this._lastErrorCode == SmartcardErrorCode.None))
        {
            //Extract each reader from the returned list.
            //The readerList string will contain a multi-string of
            //the reader names, i.e. they are seperated by 0x00
            //characters.
            string readerName = string.Empty;
```

```
                    for (int i = 0; i <= readerList.Length - 1; i++)
                    {
                        if ((readerList[i] == ''))
                        {
                            if ((readerName.Length > 0))
                            {
                                //We have a smartcard reader's name.
                                result.Add(readerName);
                                readerName = string.Empty;
                            }
                        }
                        else
                        {
                            //Append the found character.
                            readerName += new string(readerList[i], 1);
                        }
                    }
                }
            }
            return result;
        }

        #endregion

        #region IDisposable Support

        //IDisposable
        private void Dispose(bool disposing)
        {
            if (!this._disposed)
            {
                if (disposing)
                {
                    // Free other state (managed objects).
                }

                //Free your own state (unmanaged objects).
                //Set large fields to null.
                this._states = null;
                this._worker.CancelAsync();
                this._worker.Dispose();
                this._context.Dispose();
            }
            this._disposed = true;
        }

        // Implement IDisposable.
        // Do not make this method virtual.
        // A derived class should not be able to override this method.
        public void Dispose()
        {
            Dispose(true);
            // This object will be cleaned up by the Dispose method.
            // Therefore, you should call GC.SupressFinalize to
            // take this object off the finalization queue
            // and prevent finalization code for this object
            // from executing a second time.
            GC.SuppressFinalize(this);
        }

        #endregion

        #region Properties

        private bool HasContext
        {
            get { return (!this._context.IsInvalid); }
        }

        #endregion
    }
}
```

Top of page

Implementing a Test Application

You can test all of this by just creating a new Windows Forms application project and adding one line of code. Listing 7 shows you the code.

Listing 7 - Complete Form1.cs File

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using SmartcardLibrary;

namespace SmartcardApplication
{
    public partial class Form1 : Form
    {
        private SmartcardManager manager = SmartcardManager.GetManager();

        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

Be sure to add a reference to the assembly which contains the SmartcardManager class. When you start this application you will receive the message "Card inserted" each time you insert a card in the reader and the message "Card ejected" when you remove the card from the reader.

Top of page

Summary

That concludes the second part in the series of working with the Belgian Identity card. In this article you have seen how you can monitor the status of a smartcard reader. You can now detect when a card is inserted or ejected from a reader. This ofcourse is usefull for any application dealing with smartcard readers, not just for reading identity cards. In the next and final part I'll whip the two articles together to create a component for working with smartcard readers.

If you have any problems, suggestions or comments be sure to let me know.

Top of page

Download

You can find the source code for this article on the Download page of this blog.

Top of page