IDPrime .NET Smart Card

Integration Guide

gemalto
security to be free

GEMALTO, B.P. 100, 13881 GEMENOS CEDEX, FRANCE.

Tel: +33 (0)4.42.36.50.00 Fax: +33 (0)4.42.36.50.90

Printed in France.

Document Reference: D1265202B

May 31, 2013

Contents

Introduction xi

Who Should Read This Book . . . xi
Documentation . . . xi
Conventions . . . xi
Windows Versions . . . xi
Typographical Conventions . . . xii

PART I .NET CARD TECHNOLOGY

Chapter 1 Smart Card Background 2

Smart Card Basics . . . 2
Smart Card Hardware . . . 3
Smart Card . . . 3
Device/Interface . . . 4
Smart Card Software . . . 4
Operating System . . . 4
Applications . . . 5
The IDPrime .NET Card Components . . . 6
Windows 7 and 8 (and Server 2008 R2 and Server 2012) . . . 6
Windows Vista and Server 2008 . . . 8
Windows XP and Server 2003 . . . 8
Cryptographic Application Programming Interface (CAPI) . . . 8
Introducing the Windows Smart Card Framework Architecture . . . 9
Smart Card Standards and Specifications . . . 10
IDPrime .NET Smart Card Characteristics Summary . . . 11
Card Hardware Characteristics . . . 11
Card Software Characteristics . . . 11
Middleware Architecture . . . 11
Main Functions Supported . . . 11
Main Use Cases Supported . . . 12
Compliance with Operating Systems . . . 12
Compliance with Microsoft Applications . . . 12
Compliance with 3rd-Party Applications . . . 12

Chapter 2The IDPrime .NET Card 13

Background . . . 13
Why .NET on a Smart Card? . . . 13
IDPrime .NET Card Application Development . . . 14
IDPrime .NET Card Characteristics . . . 14
Card Contents . . . 14
File System . . . 14
CardConfig.xml File . . . 15
Access Manager Applications . . . 17
Additional Contents . . . 17
Assemblies . . . 17
Data Files . . . 17
Smart Card Profile . . . 18

.NET Card Specifications 19
IDPrime .NET Card Certifications 19
Common Language Runtime (CLR) 20
Common Language Runtime (CLR) Responsibilities 20
.NET Smart Card Framework Vs. .NET Framework 21

Chapter 3 Concepts and Models 23

Assemblies 23
Assemblies on the IDPrime .NET 23
Assembly Security 24
Loading Assemblies 24
Application Domains 25
Implementation 25
Differences between IDPrime .NET Application Domains and Standard .NET Application Domains 25
Application Lifecycle 26
Loading 26
Installation 26
Execution 26
Termination 26
Unloading 26
Remoting 27
Remoting in the .NET Smart Card Framework 27
Channels and Ports 28
Example 29
Using Custom Sinks 32
Why Make a Custom Sink? 32
What Are the Limitations? 32
Designing a Custom Sink 32
Using a Custom Sink 33
Garbage Collection 34
Garbage Collection 34
The GCControlAttribute 34
File System 35
Key Points about the IDPrime .NET File System 35
Example 35
Data Storage 36
Data Stored in Persistent Memory 36
Data Stored in Volatile Memory 36
MemoryStreams 37
Transactions 37
Why Transactions? 37
How Transactions Work 37
Out-of-Transaction Objects 38
Security 39
Access Manager 40
Application Security 41
Data Security 44
Supporting Legacy Infrastructure 45
Who Should Read This Section? 45
The Problem with Legacy Applications 45
Using Attributes to Manage APDUs 45
Returning Data from the Card 47
Handling Incorrect Requested Lengths 47
Card Reset Event 47

What Does a Reset Mean? . . . 47
Handling the Reset Event . . . 48
Card Services . . . 48
ContentManager . . . 48
SampleAccessManager . . . 49

Chapter 4 Card Explorer 51

Introduction . . . 51
Starting Card Explorer . . . 51
Connecting to the IDPrime .NET Card . . . 51
Toolbar . . . 53
Tab Layout . . . 54
Select Smartcard Reader Details . . . 55
Explorer Tab . . . 56
Services Tab . . . 58
Access Manager . . . 59
Card Element Properties . . . 60
Card Properties . . . 60
Folder/Directory and File Properties . . . 63
Public Key Tokens . . . 67
Identifying an Assembly . . . 67
Controlling Access to a File or Folder on the Card . . . 67
Managing Folders and Files . . . 68
Managing Folders . . . 68
Managing Files . . . 70

Chapter 5 Visual Studio .NET Integration 72

Managing the .NET Card Add-in . . . 72
How to Manage the Card Explorer Add-in . . . 72
Add-in Vs. Standalone Differences . . . 73
Templates . . . 74
Creating a Server Project . . . 74
Creating a Client Project . . . 74

Chapter 6 Getting Started 76

Using Templates to Make a Server Application . . . 76
Creating a New Solution . . . 76
Opening an Existing Solution . . . 77
Creating an IDPrime .NET Card Server Application . . . 77
Debugging . . . 80
Loading the Server onto the Card . . . 80
Starting a Service . . . 81
Deleting a Service . . . 81
Using Templates to Make a Client Application . . . 81
Creating a New Solution . . . 81
Opening an Existing Solution . . . 82
Creating a Client Application to Access a Service Running on an IDPrime .NET Card 82
Creating an On-Card Application without Templates . . . 85
Creating an Access Manager Project Using No On-Card Templates . . . 85
Building a Project from the Command Line . . . 87
Compiling Your Application with csc . . . 87
Building with NAnt . . . 88

Compiling Your Application Using NAnt . . . 88
Running Your On-card Application with a Microsoft Debugger . . . 88
Server-Side Code Changes . . . 88
Client-Side Code Changes . . . 89
Changes to the Project Settings . . . 90

Chapter 7 Code Samples 91

General Instructions . . . 91
SecureSession . . . 92
Description . . . 92
Running the Sample . . . 93
Code Extract . . . 93
APDUAttribute . . . 96
Description . . . 96
Execution . . . 96
Code Extract . . . 96
Transactions . . . 97
Description . . . 97
Execution . . . 97
Code Extract . . . 97

Chapter 8 Client-Side Components 98

SmartCard_Stub . . . 98
Referencing the ContentManager from Your Project . . . 98
SmartCard.Runtime . . . 99
Client Remoting Components . . . 99
CardAccessor Class . . . 99
AccessManagerClient Interface . . . 99
C++ Marshaller . . . 100
Why a C++ Marshaller? . . . 100
Where Can I Find the C++ Marshaller? . . . 100
Using the Marshaller . . . 100

PART II .NET MINIDRIVER API

Chapter 9 Introduction to Part 2 102

Chapter 10 Minidriver Interface V5/V6/V7 103

Chapter 11 Minidriver V5 Methods 105

Authentication Management Methods . . . 105
byte[] GetChallenge() . . . 105
void ExternalAuthenticate(byte[] response) . . . 105
void ExternalAuthenticateAM(byte[] response) . . . 105
void ChangeReferenceData(byte mode, byte role, byte[] oldPin, byte[] newPin, int maxTries) . . . 106
void VerifyPin(byte role, byte[] oldPin) . . . 106
int GetTriesRemaining(byte role) . . . 106
void LogOut(byte role) . . . 107
bool IsAuthenticated(byte role) . . . 107
byte MaxPinRetryCounter {get;} . . . 107
bool AdminPersonalized {get;} . . . 107

bool UserPersonalized {get;} . . . 107
Containers & Crypto Management Methods . . . 108
void CreateCAPIContainer(byte ctrIndex, bool keyImport, byte keySpec, int keySize, byte[] keyValue) . . . 108
void DeleteCAPIContainer(byte ctrIndex) . . . 108
byte[] GetCAPIContainer(byte ctrIndex) . . . 109
byte[] PrivateKeyDecrypt(byte ctrIndex, byte keyType, byte[] encryptedData) . . . 109
Information Management Methods . . . 110
int[] QueryFreeSpace() . . . 110
int[] QueryKeySizes() . . . 110
File System Management Methods . . . 110
void CreateFile(string path, byte[] acls, int initialSize) . . . 110
void CreateDirectory(string path, byte[] acls) . . . 111
void WriteFile(string path, byte[] data) . . . 111
byte[] ReadFile(string path, int maxBytesToRead) . . . 111
void DeleteFile(string path) . . . 112
void DeleteDirectory(string path) . . . 112
string[] GetFiles(string path) . . . 112
byte[] GetFileProperties(string path) . . . 112
Version Management Methods . . . 113
string Version {get;} . . . 113
void SetHostVersion(uint hostVersion) . . . 113

Chapter 12 Minidriver V6/V7 Methods 114

Authentication Management Methods . . . 114
Role Identifiers . . . 114
byte[] GetChallengeEx(byte role) . . . 114
byte[] AuthenticateEx(byte mode, byte role, byte[] pin) . . . 114
void DeauthenticateEx(byte roles) . . . 116
void ChangeAuthenticatorEx(byte mode, byte oldRole, byte[] oldPin, byte newRole, byte[] newPin, int maxTries) . . . 116
Properties Management Methods . . . 118
byte[] GetContainerProperty(byte ctrIndex, byte property, byte flags) . . . 118
byte[] SetContainerProperty(byte ctrIndex, byte property, byte[] data, byte flags) . . . 119
byte[] GetCardProperty(byte property, byte flags) . . . 119
byte[] SetCardProperty(byte property, byte[] data, byte flags) . . . 127

PART III APDU ENCODING

Chapter 13 APDU Encoding 137

Introduction . . . 137
APDU Format . . . 137
Argument Encoding . . . 138
Payload with length > FF . . . 139
MSCM Answer Interpretation . . . 140
The APDUs Exchange Flow . . . 140

Chapter 14 Hivecodes and Examples 141

Generic Answer Formation for .NET Card Services (Except MSCM) . . . 141
Computing Hivecodes . . . 141
Namespace Hivecode . . . 141
Type Hivecode . . . 142

Method Hivecode 142
MSCM Method Hivecodes 142
Namespace Hivecodes 144
Standard Type Hivecodes 145
Exception Type Hivecodes 145
Other Useful Type Hivecodes 146
Other Useful Method Hivecodes 146
APDUs Exchange Examples 147
Get Challenge 147
Get Response 147
External Authenticate 148
Log Out 148

PART IV CONFIGURING PARAMETERS

Chapter 15 Configuring Parameters 150

Introduction 150
Configurable Parameters (.NET Minidriver Assembly) 150
Using SetCardProperty 150
Using Installation Parameters 155
Configurable Parameters (IDGo 5000 Bio) 156

Appendix A Troubleshooting 159

Communication Problems 159
The Easy Checklist 159
Further Steps 159
SSO Option Deactivation Problem 160

Appendix B Marshaller Stub Sample 161

Terminology 170

Abbreviations 170
Glossary 171

References 173

Standards and Specifications 173
Recommended Reading 174
Useful Web Site Addresses 174

List of Figures

Figure 1 - .NET Smart Card . . . 3
Figure 2 - IDPrime .NET On-Card and Off-Card Components . . . 6
Figure 3 - The Smart Card CP User Interface . . . 7
Figure 4 - The Smart Card Verification CP User Interface . . . 8
Figure 5 - Microsoft Base CSP vs. Vendor-Specific Custom CSP . . . 10
Figure 6 - .NET Card Explorer . . . 15
Figure 7 - Example of CardConfig.xml File . . . 16
Figure 8 - Libraries and Profiles Relationship . . . 19
Figure 9 - Client - Server Communication Using Channels and Ports . . . 28
Figure 10 - Sample Server Code . . . 30
Figure 11 - Sample Client Code . . . 31
Figure 12 - .NET Remoting Architecture with Custom Sinks . . . 32
Figure 13 - Properties Page for Assembly . . . 43
Figure 14 - Share With Dialog . . . 43
Figure 15 - Select Smart Card Reader dialog box . . . 52
Figure 16 - Log on to .NET Smart Card dialog box . . . 52
Figure 17 - Card Explorer Toolbar . . . 53
Figure 18 - Card Explorer – Explorer Tab . . . 54
Figure 19 - Card Explorer – Services Tab . . . 54
Figure 20 - Select Smartcard Reader Dialog Box . . . 55
Figure 21 - Card Explorer - Explorer Tab . . . 56
Figure 22 - Card Explorer – Services Tab . . . 58
Figure 23 - Card Explorer - Card Element Properties . . . 60
Figure 24 - Card Properties - General Tab . . . 61
Figure 25 - Card Properties - Advanced Tab . . . 62
Figure 26 - Folder Properties - General Tab . . . 63
Figure 27 - File Properties - Security Tab . . . 64
Figure 28 - Share With... Dialog Box . . . 65
Figure 29 - Modifying Permissions for a Public Key Token . . . 66
Figure 30 - Contextual Menu for Folders . . . 68
Figure 31 - The Open Dialog Box . . . 69
Figure 32 - Contextual Menu for Executable Files . . . 70
Figure 33 - Add-In Manager . . . 73
Figure 34 - New Project Dialog Box (netCard Server) . . . 74
Figure 35 - New Project Dialog Box (netCard Client Console) . . . 75
Figure 36 - New Blank Solution (Server Applications) . . . 76
Figure 37 - New Project Dialog Box (netCard Server) . . . 77
Figure 38 - New Blank Solution (Server Applications) . . . 82
Figure 39 - New Project Dialog Box (netCard Client Console) . . . 83
Figure 40 - New Project (No On-card Templates) . . . 86
Figure 41 - Advanced Build Settings (No On-card Templates) . . . 86
Figure 42 - Add Reference (No On-card Templates) . . . 87
Figure 43 - Add Reference (SmartCard_Stub) . . . 98

List of Tables

Table 1 - The Smart Card Profile . . . 19
Table 2 - Card Explorer Toolbar Icons Descriptions . . . 53
Table 3 - Select Smartcard Reader Options . . . 55
Table 4 - Card Element Descriptions and Menu Options . . . 57
Table 5 - Card Services . . . 58
Table 6 - Menu Options . . . 59
Table 7 - Card Properties - General Tab Elements . . . 61

Table 8 - Card Properties - Advanced Tab Elements 62
Table 9 - Folder Properties - General Tab Elements 63
Table 10 - File Properties - Security Tab Elements 64
Table 11 - Permission Types for Assemblies 66
Table 12 - Permission Types for Folders 66
Table 13 - Permission Types for Non-Assembly Files 66
Table 14 - Samples provided with the SDK 91
Table 15 - Argument Types and Their Encoding 138
Table 16 - Hivecodes for V5 143
Table 17 - Hivecodes for V6/V7 144
Table 18 - Microsoft Defined Minidriver Parameters 150
Table 19 - Gemalto Proprietary Parameters 153
Table 20 - Installation Parameters 155
Table 21 - IDGo 5000 Bio Specific Configurable Parameters 156

This document describes Gemalto's IDPrime .NET smart cards, including their architecture and general concepts about .NET technology. .NET Card technology includes:

- IDPrime .NET Card, a new post-issuance programmable smart card designed to work with .NET applications
- .NET Smart Card Framework, the class libraries and managed runtime environment in which applications execute on a .NET card
- Tools to manage and develop applications for IDPrime .NET cards

Who Should Read This Book

This guide is intended for system integrators who want to integrate .NET smart cards in their systems and software engineers who want to write applications for IDPrime .NET cards.

It is assumed that users are familiar with .NET Framework concepts, and focuses on components and tools that are unique to the .NET Smart Card Framework. This documentation does not repeat topics covered in the .NET Framework documentation, which is available from the Microsoft .NET Framework Developer's Center.

The book is divided into three parts:

- Part 1: .NET Technology describes the main concepts behind .NET and shows you how to develop client and server applications.
- Part 2: .NET Minidriver API lists all of the functions that are available at the API level for IDPrime .NET cards.
- Part 3: APDU Encoding provides information about how to code APDUs to call methods.

Documentation

For documentation about IDPrime .NET Cards, please go to Gemalto Product Catalog and consult the Download section at http://www.gemalto.com/products/dotnet_card/

Conventions

The following conventions are used in this document:

Windows Versions

Where this document refers to Windows 7 and 8, it is equally applicable to Windows Server 2008 R2 and Windows Server 2012.

Typographical Conventions

.NET Smart Cards documentation uses the following typographical conventions to assist the reader of this document.

Convention	Example	Description
Bold	Type **myscript.dll**	Actual user input or screen output.
>	Select File > Open	Indicates a menu selection. In this example you are instructed to select the **"Open**" option from the "**File"** menu.

Part I

.NET Card Technology

1

Smart Card Background

The IDPrime .NET Card is a new-generation smart card. This section provides some background about smart cards in general.

Smart Card Basics

A smart card is a portable, tamper-resistant computer with a programmable data store. A conventional smart card is the same size and shape as a plastic credit card, and it is embedded with a silicon integrated circuit (IC) chip. The chip provides memory to store data, a microprocessor to manipulate that data, and sometimes a cryptographic coprocessor to perform complex calculations associated with cryptographic operations.

Smart cards that contain an IC chip are sometimes called chip cards to distinguish them from cards that offer either memory storage only, or memory storage and non-programmable logic. These memory cards store data efficiently, but cannot manipulate that data because they do not contain a computing chip. Memory cards depend on host-side applications to perform whatever processing is required on the data that they store.

Chip cards are either fixed command set cards, which are programmed and softmasked in read-only memory (ROM) during manufacturing to interact with security infrastructure systems as portable, secure tokens; or post-issuance programmable cards, which can be used for multiple purposes simultaneously (for example, a card might be used as both as a security token and a rechargeable stored-value card), and which can be upgraded or re-purposed while in the field, long after their initial manufacture-time programming and softmasking.

The IDPrime .NET Card is a post-issuance programmable smart card. This new card is based on technology from HiveMinded that is a subset of ECMA standards (language, CLR, framework) for .NET. The IDPrime .NET Card technology was introduced by Gemalto in 2002 as an on-card application programming solution for .NET infrastructures. The IDPrime .NET Card technology offers support for multi-language application programming using an appropriate subset of the .NET class libraries. This subset has been tailored for smart card applications, and provides an optimized runtime environment on the smart card to enable communication between card and terminal using .NET remoting, ensure secure simultaneous execution of multiple applications, and exploit many other .NET framework features.

Smart Card Hardware

Physically, a smart card is a component in a system that includes:

- A smart card
- A physical device or interface that enables data exchange between the smart card and applications running on a host system

Smart Card

A conventional, contact-type smart card looks like a credit card with an embedded integrated circuit chip. Its physical characteristics are defined by *ISO 7816-1*.

Here's an example of a conventional contact-type smart card.

Figure 1 - .NET Smart Card

The plastic media of the card may be printed to include a range of information, including company logos and photos (for example, for scenarios in which the smart card will also serve as an identification badge).

The working part of the smart card is the chip embedded at left center and includes:

- Electrical contacts defined by *ISO 7816-2*.
- The CPU (integrated circuit microprocessor). The chip in most smart cards is an 8- or 16-bit microprocessor, usually using the Motorola 6805 or Intel 8051 instruction set, with clock speeds up to 5 MHz. The chip in the IDPrime .NET card is a 32-bit microprocessor.
- A cryptographic coprocessor, which adds on-card capacity to perform the complex calculations needed for cryptographic operations.
- Three types of memory:
  - Random Access Memory (RAM), volatile, mutable memory; content in RAM is not preserved when power to the card is removed.
  - Read-Only Memory (ROM), persistent, nonmutable memory into which the fixed program of the card is loaded and stored during manufacturing. ROM contains the card's operating system routines, permanent data, and permanent user applications. Content in ROM is preserved when power to the card is removed.
  - Electrical Erasable Programmable Read-only Memory (EEPROM), persistent, mutable memory used for data storage on the card. Content in EEPROM is preserved when power to the card is removed.

Smart cards are also available in these alternative form factors.

Contactless Smart Cards

Contactless smart cards are chosen when physically inserting the card into a reader to enable communication is impractical (for example, when the card is used for physical access to a building). A contactless smart card communicates through electromagnetic fields using an antenna that is built into the card; the microcircuit is sealed inside the card and no contact points are visible on the face. Contactless smart cards are defined by ISO-10536, parts 1,2,3, Identification Cards - Contactless Integrated Circuit(s) Cards - Close-coupled Cards.

Smart cards used for multiple purposes are often hybrid or combination cards, configured both for contactless uses (for example, for entrance to a building) and for applications that require authentication of the user as well as recognition of the card (for example, to access networked resources).

USB-capable Smart Cards

USB-capable smart cards incorporate the USB interface electronics normally found in a smart card reader on the card itself. The alternate use of electrical contact points for USB is defined by proposed amendments to the *ISO 7816-2* standard.

Because USB-capable smart cards do not require a reader, an alternative form factor card is available: a token format that is a cut-down version of the conventional format smart card, preserving only the chip and minimal adjacent plastic card medium. The cut-down card is inserted into a plastic dongle, a key-sized receptacle, which plugs directly into a USB port on the host system.

Device/Interface

The most commonly deployed device to connect a smart card with a host system in order to exchange data is a smart card reader. The reader is attached to the host system using a serial, USB, or PC Card connection. The smart card is inserted into the reader and when the card's contact points are correctly aligned, communication between the smart card and the host system can be initiated.

A contactless smart card communicates with the host system using electromagnetic fields using an antenna that is built into the card. Physical proximity to a card reader/ terminal that also includes an antenna triggers initiation of a communication protocol that allows data exchange.

Because USB-capable smart cards incorporate the USB interface electronics normally found in a smart card reader on the card itself, a USB-capable smart card interfaces directly with the host system through a USB port. A conventional form factor USB-capable card uses a special receptacle (resembling a standard smart card reader but containing no electronics) plugged into a USB port, while a cut-down format USB-capable card is inserted into a dongle that plugs directly into the USB port.

Smart Card Software

There are several different software components that operate on modern smart cards.

Operating System

The operating system is typically responsible for managing communication and memory operations between the chip and any applications running on the smart card. If the card supports Cryptography, the operating system may also provide an interface to cryptographic hardware.

Applications

In order for a smart card to be useful, it must perform some operations that are understood by a terminal or other external smart card reader. Smart card applications range from a simple electronic purse to implementation of complex cryptographic algorithms for digital security.

Traditionally, smart card applications were developed for a specific chip, and were written in C by a specialized community of smart card developers. These applications would be written to the chip at production time, and could not be changed after the card was issued. This model of application development had some deficiencies. Moving your application to a new type of chip meant rebuilding and possibly redesigning your application for the new chip. Since applications needed to be written to the chip at production time, the entire lifecycle of the smart card needed to be known in advance - there would be no way to change the application in the field.

In the late 1990's, smart card application development changed radically. Smart card companies released smart cards known as Java Cards, which contained a Java interpreter. By writing applications in Java, smart card developers could insulate themselves from the details of the specific chip hardware. Also, Java applications were stored in non-volatile but erasable memory, so new applications could be loaded to the card even after the card was in the hands of a user. Although the Java Card represented a significant step forward in smart card development, it did not completely isolate the developer from the protocols of the smart card. Developers were still responsible for managing the communication between the card and the off-card terminal.

The IDPrime .NET Card contains an IL interpreter that allows users to develop applications for the smart card using the ECMA .NET standard. Applications can be developed in any language that can be compiled to IL.

The following two software components are present only on smart cards that have interpreters:

Runtime Environment

The runtime environment consists of two components. The first is an interpreter that is responsible for running applications that are loaded to the card. The second component is a collection of libraries that support applications. On a Java Card, these libraries would contain the types and methods that are part of the Java Card API. On the IDPrime .NET Card, these libraries contain a subset of the ECMA .NET libraries.

Loader

Since applications on cards with runtime environments can be loaded after the card is produced, there must be a software component that is responsible for loading these components to the card. The Loader on the IDPrime .NET Card is responsible for several tasks:

- Verifying that the IL being loaded to the card is safe to run.
- Verifying that the assembly being loaded to the card is properly signed.
- Ensuring that all types used by the assembly are already present on the card

The Loader is also responsible for removing applications from the card.

The IDPrime .NET Card Components

The architecture of the IDPrime .NET Solution relies on the Microsoft Base Cryptographic Service Provider (Base CSP) component as follows:

- Windows 7 and 8 (and Server 2008 R2 and Server 2012): The base CSP is V7 and is integrated already in Windows 7 and 8.
- Windows Vista (and Server 2008): The base CSP is V6. For Vista SP1, base CSP V6 is already integrated in Vista. However for pre-SP1 base CSP V6 needs to be downloaded via Windows Update.
- Windows XP and Server 2003: The base CSP is V5. The base CSP V5 must be downloaded via Windows Update.

The IDPrime .NET Solution consists of both on-card and off-card components, as shown in “Figure 2”. These include the card module assembly that resides on the IDPrime .NET smart card itself, and some libraries known as the minidriver .dll, that must be installed in the “Windows System” directory on the client computer.

This section describes the architecture for Windows 7 and 8 and the differences that exist for the Vista and XP versions.

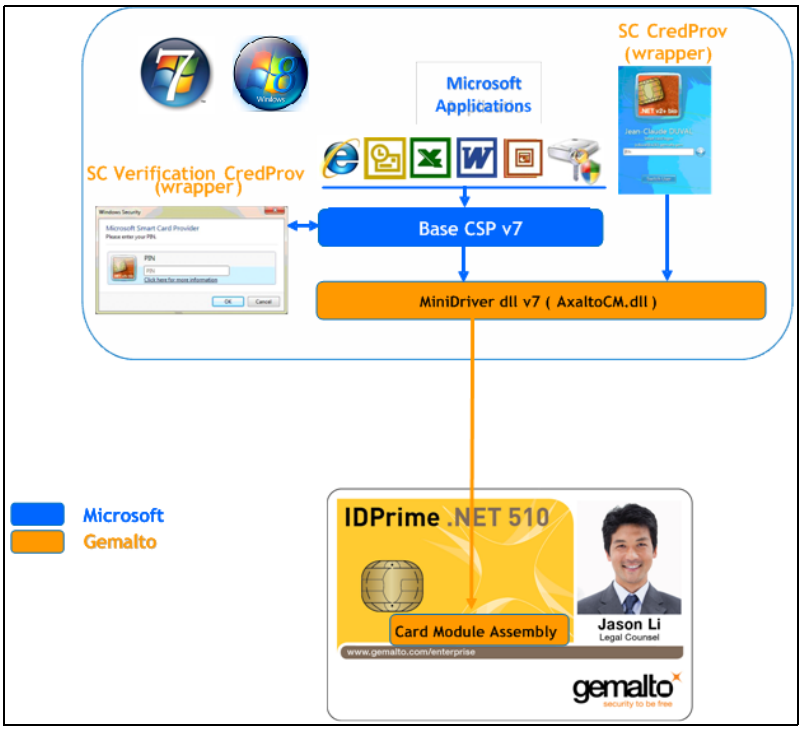
The “Windows System” directory differs according to the version of Windows as follows:

- For Windows XP, Vista and 32-bit OS version of Windows 7 and 8 - All components are installed in C:\Windows\system32
- For 64-bit OS version of Windows 7 and 8 - The 64-bit components are installed in C:\Windows\system32 and the 32-bit components are installed in C:\Windows\SysWOW64.

Windows 7 and 8 (and Server 2008 R2 and Server 2012)

The components in Windows 7 and 8 are as follows:

Figure 2 - IDPrime .NET On-Card and Off-Card Components



On-Card Components

IDPrime .NET includes the following on-card components:

- **.NET Operating System**
- **Card Module Assembly (Gemalto)** – This is the “oncard” module for the Minidriver. It is compliant with the Microsoft Minidriver v7 specification and paired with the off-card minidriver dll library.

Off-Card Components

IDPrime .NET requires the following client libraries to be installed in the “Windows System” directory (see previous page). Some implement a User Interface as shown in Figures 3 and 4.

- **Base CSP v7**: Minidriver based and integrated in Windows 7 and 8. This provides the standard Windows 7 and 8 Credentials GUIs for user authentication.
- **Minidriver dll (Gemalto)**. This is compliant with the Microsoft Minidriver V7 specification and is installed automatically by the Windows 7 and 8 “plug-and-play” feature when you insert a IDPrime .NET card in the reader.
- **Smart Card Credential Provider (CP)**. This provides:
  - PKI management (done by the MS CP)
  - Minidriver access (done by the MS CP)

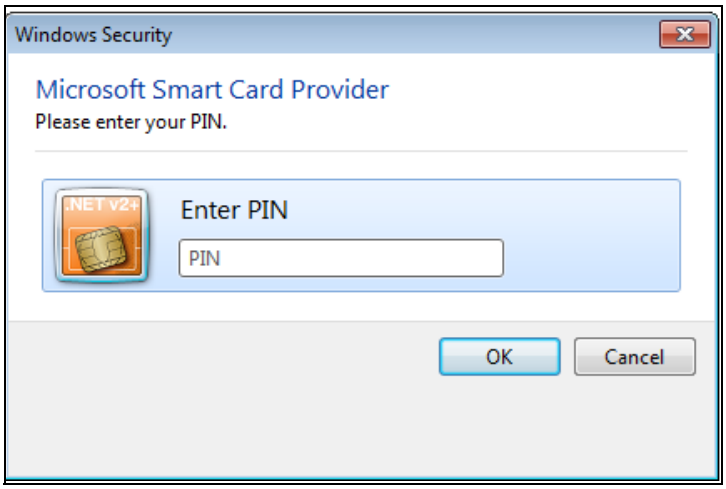
  It is the CP that manages the secure desktop functions such as smart card logon. Its GUI is shown in “Figure 3”.

Figure 3 - The Smart Card CP User Interface

- **Smart Card Verification CP**. This provides:
  - PKI management (done by the MS CP)
  - Minidriver access (done by the MS CP)

It is this CP that manages the user desktop functions. It manages the authentication needed for applications such as SSL authentication (accessing secure web sites) and signing emails.

Figure 4 - The Smart Card Verification CP User Interface



Gemalto has developed the IDGo 800 credential provider (CP) for Windows 7 and 8 which can manage up to 6 user PINs, each with its own PIN policy. The IDGo 800 CP is an optional feature which, if installed, replaces the standard MS CPs (in “Figure 3” and “Figure 4”).

Windows Vista and Server 2008

The components are exactly the same as for Windows 7 and 8. The only difference is that the Microsoft Base CSP is V6 instead of V7. The Base CSP was integrated in Vista from service pack 1. For the pre-SP1 version, it needs to be downloaded via Windows Update.

Windows XP and Server 2003

The components are exactly the same as for Windows 7 and 8. The only difference is that the Microsoft Base CSP is V5 instead of V7. The Base CSP is not integrated in Windows XP but must be downloaded via Windows Update.

Cryptographic Application Programming Interface (CAPI)

Microsoft’s Base CSP implements an API called CAPI. This interface is used in all Microsoft tools and is native to all versions of Windows from XP onwards. CAPI enables you to use the services provided by Base CSP without the need for sending APDUs to the card. The Base CSP itself uses the .NET minidriver (axaltocm.dll), which in turn communicates with the Gemalto on-card minidriver assembly (cardmodule.exe). The .NET minidriver dll can be download from the Microsoft Update site:

http://catalog.update.microsoft.com/v7/site/Search.aspx?q=gemalto%20minidriver%20net

CAPI is fully documented on the Microsoft Software Developer Network site http://msdn.microsoft.com/ at the following link:

http://msdn.microsoft.com/en-us/library/aa380256%28VS.85%29.aspx

You may also find the following link useful which lists various source codes and describes their use:

http://msdn.microsoft.com/en-us/library/aa388162%28v=VS.85%29.aspx

Introducing the Windows Smart Card Framework Architecture

In the past, smart card vendors made and maintained a monolithic *Cryptographic Service Provider* (CSP) for their own smart cards. Vendors had to write complete, custom, software CSPs to enable smart card scenarios for their cards.

The new Windows Smart Card Framework architecture is layered to separate the basic required cryptography components at the top from the unique smart card hardware interfaces at the bottom; the unique hardware-specific interface for a given smart card receives the name of *Minidriver* (formerly called *Card Module*) and takes the form of a Dynamic Link Library (dll). Minidrivers lever the common cryptographic components now included in the Windows platform.

This new architecture has been implemented in the Crypto API Next Generation (CNG) as part of the Microsoft Windows Vista OS, and is called the *Microsoft Smart Card Key Storage Provider* (KSP).

The cryptography for smart cards has been implemented natively in Windows Vista, 7 and 8 and their corresponding Windows Server versions. For Windows XP and Server 2003, it is included in the Microsoft Base CSP (available as Microsoft Windows Update # KB909520.)

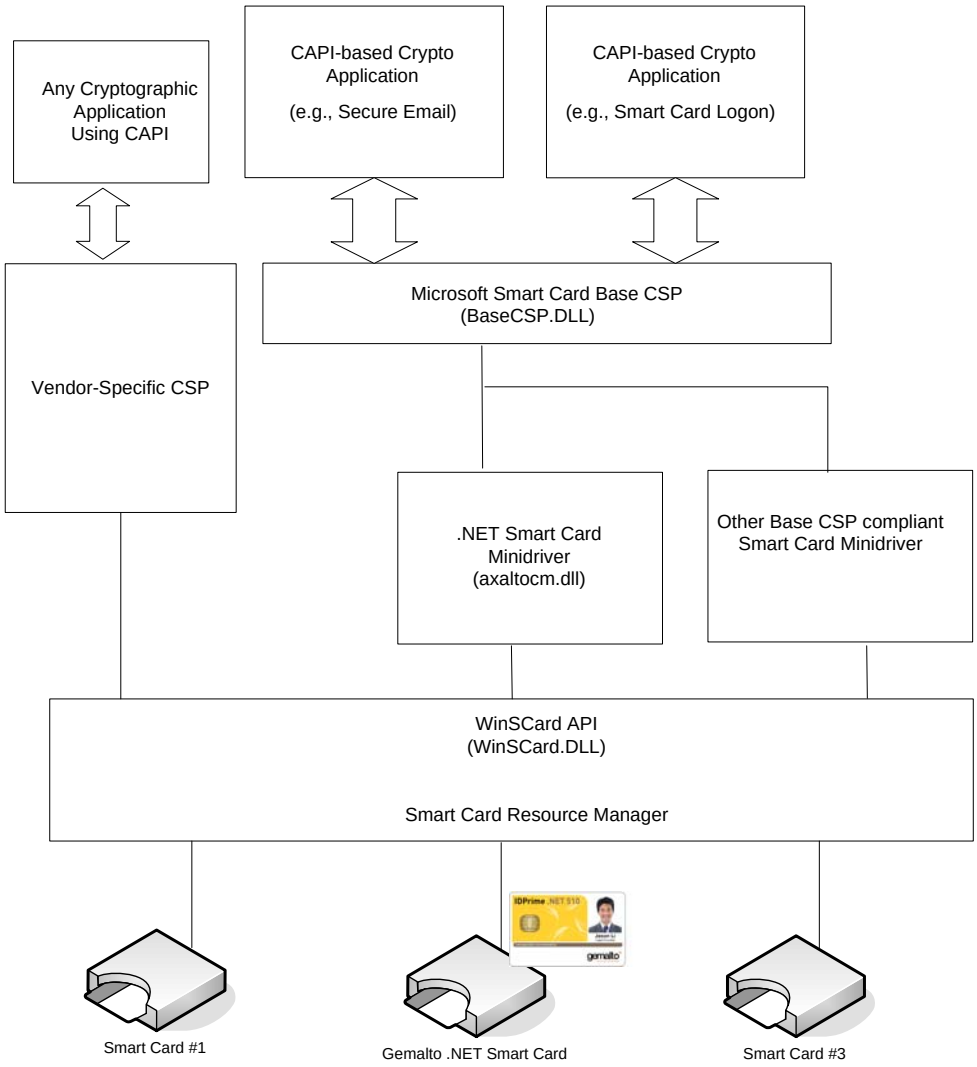
Note: The Microsoft Base Smart Card Cryptographic Service Provider should not be confused with the “Microsoft Base Cryptographic Provider v1.0”, which is the default, non-smart card software CSP in Windows.

Base CSP and KSP provide the common software cryptographic portions, while the MiniDriver of a given smart card compliant with this architecture simply plugs in to provide access to the hardware and software of that particular smart card. “Figure 5” illustrates the two Smart Card CSP architectures.

From an application developer perspective, the Base CSP, KSP and Minidriver interfaces provide a common way to access smart card features, regardless of the card type.

For users, the new architecture includes support for all preexistent smart card scenarios, and it also provides new tools for the management of the Personal Identification Number (PIN).

Figure 5 - Microsoft Base CSP vs. Vendor-Specific Custom CSP



Smart Card Standards and Specifications

The importance of developing globally-accepted standards for smart cards, smart card applications, and related devices was recognized early in the smart card industry as essential to broad acceptance of smart cards in all sectors. Some standards/ specifications offer definitions for the interfaces between parts of the whole system, from operating system-level interaction to card communication, while other specifications came out of specific industries and reflect their special interests and requirements. More recently, security-related standards have become important, especially as the participation of smart card-carried digital credentials in sensitive secure systems has grown.

For details about the International Standards relating to .NET cards, please refer to “References” on page 173.

IDPrime .NET Smart Card Characteristics Summary

This section provides a summary of the technical characteristics.

Card Hardware Characteristics

- Wide range of hybrid card options for combined physical and logical access control
- Graphical personalization on request
- Wide range of form factor: ID1 standard format, SIM-size format, card modules, various USB token devices
- High performance level based on 32-bit RISC secure chip (CC EAL 5+) with cryptographic capability: On board key generation, true Random Number Generator, RSA, DES/3DES, AES-256, Hash SHA1 / SHA256 / HMAC / MD5
- 50 KB free Flash memory space for certificate and application loading
- ISO 7816-1 to 3, T=0 compliant contact card interface with a maximum communication speed of 223 Kbps (negotiable PPS)
- Voltage range: 1.62 Vdc to 5.5 Vdc

Card Software Characteristics

- Capability to implement new on board applications
- SDK available to develop new on board applications
- Electrical personalization on request: Parameters specified by Microsoft minidriver specifications, and additional features from Gemalto (PIN policy)
- Number of 1024 / 2048 bits certificates and keys: 15
- FIP140-2 Level 3 in option

Middleware Architecture

- Compliance with Microsoft Base CSP / Crypto API and Minidriver v7 specifications: No proprietary middleware to install on the user PC
- Single minidriver certified by Microsoft for all Windows OS and platforms (axaltocm.dll)
- Automatic download of the minidriver from the Microsoft Update site (Windows 7 and 8 and corresponing Windows Server versions)
- Support of PKCS#11 crypto. architecture on Windows, Mac OS, Linux and UNIX-like Operating Systems.

Main Functions Supported

- Two Factors of Authentication with multi - PIN capability
- PKI based on X509 / PKCS#12 / PFX digital certificates
- OTP based on OATH event based specifications, with live and self provisioning capabilities as option, EMV CAP version as option
- Biometric authentication capability in option (IDGo 5000 Bio Solution)

Main Use Cases Supported

- Logon
- Change and Unblock PIN
- Encryption of e-mails, files, directories, volumes, hard disks, USB memory tokens
- Secure web and VPN access

Compliance with Operating Systems

- High level of integration with Microsoft OS: Windows XP, Vista, Seven and associated Server versions
- Compliance with UNIX-like OS such as Linux and Solaris through PKCS#11 libraries
- Compliance with Mac OS through PKCS#11 and Tokend libraries

Compliance with Microsoft Applications

- Office
- Exchange
- Internet Explorer
- EFS
- Remote Desktop
- Bit Locker and Bit Locker to Go
- Direct Access, UAG and ForeFront
- ILM / FIM Card Management System
- Active Directory and Microsoft Certificates Services

Compliance with 3rd-Party Applications

- Other Card Management Systems: Intercede, Opentrust, Passlogix, Gemalto (DAS, vSEC:CMS)

For an updated list of IDPrime .NET cards compliant applications, please refer to the IDPrime .NET pages http://www.gemalto.com/products/dotnet_card/.

2

The IDPrime .NET Card

The IDPrime .NET Card is a post-issuance programmable smart card. This new card is based on technology from HiveMinded that is a subset of ECMA standards (language, CLR, framework) for .NET. The IDPrime .NET Card was introduced by Gemalto in 2002 as an on-card application programming solution for .NET infrastructures. The IDPrime .NET Card technology offers support for multi-language application programming using an appropriate subset of the .NET class libraries. This subset has been tailored for smart card applications, and provides an optimized runtime environment on the smart card to enable communication between the card and terminal using .NET remoting, ensure secure simultaneous execution of multiple applications, and exploit many other .NET framework features.

Background

The IDPrime .NET Card is a new type of post-issuance programmable smart card. First demonstrated in 2002, the card is designed to work with .NET platform applications.

Why .NET on a Smart Card?

The .NET Card technology that encompasses the whole .NET offer (not only the IDPrime .NET Card itself, but also the .NET Smart Card Framework and the IDPrime .NET Card Add-in to Visual Studio .NET) offers key advantages over other programmable smart card platforms, including:

- A flexible on-card software architecture.
- Movement of more processing to the card for enhanced security and portability.
- Easy development of on-card applications, which is described in this section.
- A managed runtime environment that is tailored for smart cards. The .NET Smart Card Framework runtime environment (also called the .NET Smart Card Framework common language runtime or CLR), is described in “Common Language Runtime (CLR)” on page 20.

IDPrime .NET Card Application Development

IDPrime .NET Card application development mirrors .NET application development in general, offering these benefits:

- Applications can be written in any .NET-compliant programming language, which means that developers can create applications using the language that they are comfortable with and that best suits their business needs. Moreover, because they are compiled to a common intermediary language, applications written in different languages interact seamlessly within the .NET Smart Card Framework.
- .NET Card technology includes application development tools that are fully integrated into VisualStudio.NET, the standard development environment for .NET applications.
- Because .NET concepts are carried into the IDPrime .NET Card technology, the learning curve for the growing developer base of .NET software engineers who want to begin programming for IDPrime .NET cards is very short.

In addition, a key benefit to developing applications for the IDPrime .NET Cardis the communication model, which enables developers to move away from the APDU-centric communication architecture that is an underlying constraint on other programmable smart card platforms. Instead, communication between the IDPrime .NET Card and the host system uses a subset of the .NET Remoting feature, which is potentially capable of supporting standard, widely-understood protocols (for example, XML, SOAP, HTTP), as well as the traditional 7816-4 protocol APDU commands.

IDPrime .NET Card Characteristics

IDPrime .NET Card characteristics are enumerated in the CardConfig.xml file for the card, which lists the card, runtime, and chip versions; supported cryptographic algorithms; and available communication speeds. “Figure 7” on page 16 shows an example listing of the CardConfig.xml file.

Card Contents

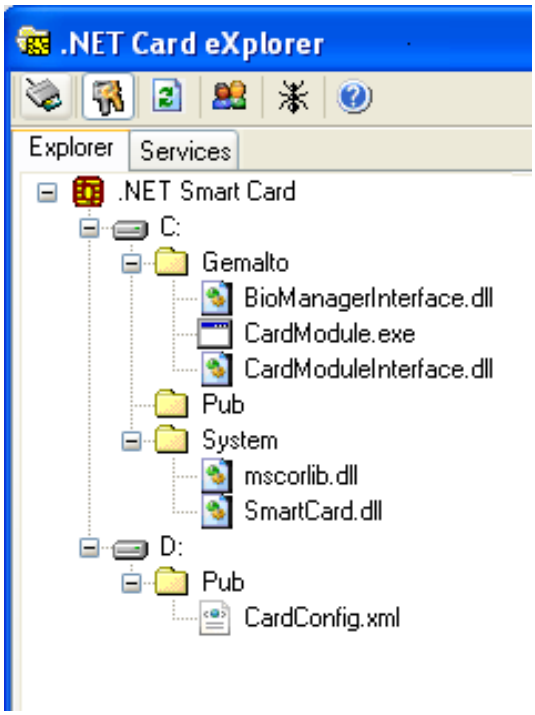
A new IDPrime .NET Card contains the .NET Smart Card Framework: the .NET Smart Card Framework libraries and common language runtime (CLR). In addition to the framework, the card contains a file system, a configuration file, and servers that enable you to communicate with the card in order to do work.

As you work with the IDPrime .NET Card, you may add new assemblies or data files to the card.

File System

When you connect to a new card the first time, the Card Explorer display shows that the card contains a file system, a set of initial folders, and some initial files:

Figure 6 - .NET Card Explorer



The file system contains these base folders:

- C:\Gemalto (Gemalto executables and libraries)
- C:\Pub (public)
- C:\System (contains the class libraries and any other libraries that are meant to be accessible to all applications)
- D:\Pub (public; pre-loaded with the CardConfig.xml file, which identifies aspects of the card's capability to work with host-side applications)

Note: While the CardConfig.xml file is found in the D:\pub directory, which is readable by all users on the card, its contents can be changed only by the cards admin user.

New files and folders can be added to the file system, subject to user permissions. Files and folders can also be deleted from the card.

CardConfig.xml File

The CardConfig.xml file is pre-loaded on the IDPrime .NET Card and is stored in the D:\Pub directory. The CardConfig.xml file contains information that can identify aspects of the IDPrime .NET Card's capability to work with host-side applications.

“Figure 7” shows an example of the contents of a CardConfig.xml file.

Figure 7 - Example of CardConfig.xml File

```
<?xml version="1.0" encoding="utf-8"?>
<config xmlns="http://www.dotnetcard.com/schemas/2004/09/dotnetcard">
    <product name="Gemalto .NET Smart Card" version="1.1.0.542026" vendor="Gemalto">
    <hardware name="SLE88CFX4000P" version="1.0" vendor="Infineon">
        <serialnumber>00865141B050BA4A</serialnumber>
    </hardware>
    <manufacturing>
        <module>
            <siteID>1942 </siteID>
            <date>4323 </date>
        </module>
        <embedding>
            <siteID>0443 </siteID>
            <date>4323 </date>
        </embedding>
        <preperso>
            <siteID>0444 </siteID>
            <date>4323 </date>
            <equipmentID>00000001 </equipmentID>
            <appCode>S1005987 A1 </appCode>
        </preperso>
        <perso>
            <siteID>FFFF </siteID>
            <date>FFFF </date>
            <equipmentID>FFFFFFFF </equipmentID>
            <appCode>FFFFFFFFFFF </appCode>
        </perso>
    </manufacturing>
    <crypto>
        <algo name="DES" type="sym" min_key_length="64" max_key_length="64"
         keygen="true" />
        <algo name="TripleDES" type="sym" min_key_length="128" max_key_length="192"
         keygen="true" />
        <algo name="Rijndael" type="sym" min_key_length="128" max_key_length="256"
         keygen="true" />
        <algo name="RSA" type="asym" min_key_length="256" max_key_length="1024"
         keygen="true" />
        <algo name="SHA1" type="hash" />
    </crypto>
    <converter version="2.0.0.0"/>
    <protocol version="1.1.0.0" transport="APDU" type="T0" />
    <comm_speed>
            <item>9600</item>
            <item>19200</item>
            <item>38400</item>
            <item>11160</item>
            <item>115200</item>
            <item>223200</item>
    </comm_speed>
</config>
```

Access Manager Applications

There are two access manager applications that enable you to communicate with the card. For more detailed descriptions, please refer to “Card Services” on page 48.

ContentManager

This is pre-loaded in the card (C:\System\SmartCard.dll). When a client application queries the card to learn what services are available on the card, the Content Management server responds by providing a list of available services.

SampleAccessManager

This is a sample from the SDK.

Additional Contents

As you develop applications for the card, you will install additional assemblies and data files to the card.

Assemblies

Applications (.exe files) and libraries (.dll files) can be added to the card. While execution of applications on the IDPrime .NET Card is not subject to user permissions, managing applications and libraries (for example, adding new application files) on the card is based on role-based access to the folders in which the application files are found.

Default card permissions permit users with the **Guest** user role to add applications and libraries to the \pub folders only. By default, the **Admin** user can add applications and library files anywhere in the file system. Other users can add applications and libraries as permitted by the user-code, path-access definition configured for each user on the card at user creation time.

Note: The Guest role applies only to SmartAccessManager and does not exist in Microsoft Card Module (MSCM). MSCM has “Admin”, “User” and “Everyone” roles only.

The rules that govern applications’ permissions to access other applications or libraries are based on the public key token list of the requested resource.

Application and library files are uploaded to the card using the Card Explorer tool, either as a stand-alone application or as a Visual Studio .NET Add-in. In the Card Explorer, identify a folder on the card to which you have access, right-click the folder, and select **Add** > **New File**. In the **Open** dialog box, navigate to the application or library file on the host system that you want to add to the card, and click **Open**. The file is copied to the selected folder on the card.

Data Files

Files that are not executables or libraries can be added to the card, subject to user permissions. The primary limitation is the amount of available non-volatile memory (EEPROM) on the card.

Generally, data files are stored on smart cards so that access to the data contained within the files can be controlled.

For example, key files associated with cryptographic services can be stored on the IDPrime .NET Card. When the owner of the IDPrime .NET Card wants to perform an operation that requires the key files (for example, to encrypt an e-mail message), he or she attaches the card to the host system and provides authentication information (often a PIN) in order to access the key files stored on the card.

Data files can also be stored on an IDPrime .NET Card for reasons unrelated to cryptographic operations. Because data stored on an IDPrime .NET Card is not only secure, it is also conveniently portable; a file might be loaded onto a card simply to transport the file from one physical location to another.

Data files are added to specific folders within the D: file system, subject to user permissions. Default card permissions permit users with the **Guest** user role to add files to the D:\pub folder only. By default, the **Admin** user can add data files anywhere in the D: file system. Other users can add data files as permitted by the user-data, path-access definition configured for each user on the card at user creation time.

Note: Again, the “Guest” role applies only to SmartAccessManager.

Data files can be uploaded to the card using the Card Explorer tool. In the Card Explorer, identify a folder on the card to which you have access, right-click the folder, and select **Add** > **New File**. In the **Open** dialog box, navigate to the file on the host system that you want to add to the card, and click **Open**. The file is copied to the selected folder on the card. Data files can also be copied to the card using the standard Windows Explorer drag-and-drop mechanism.

Smart Card Profile

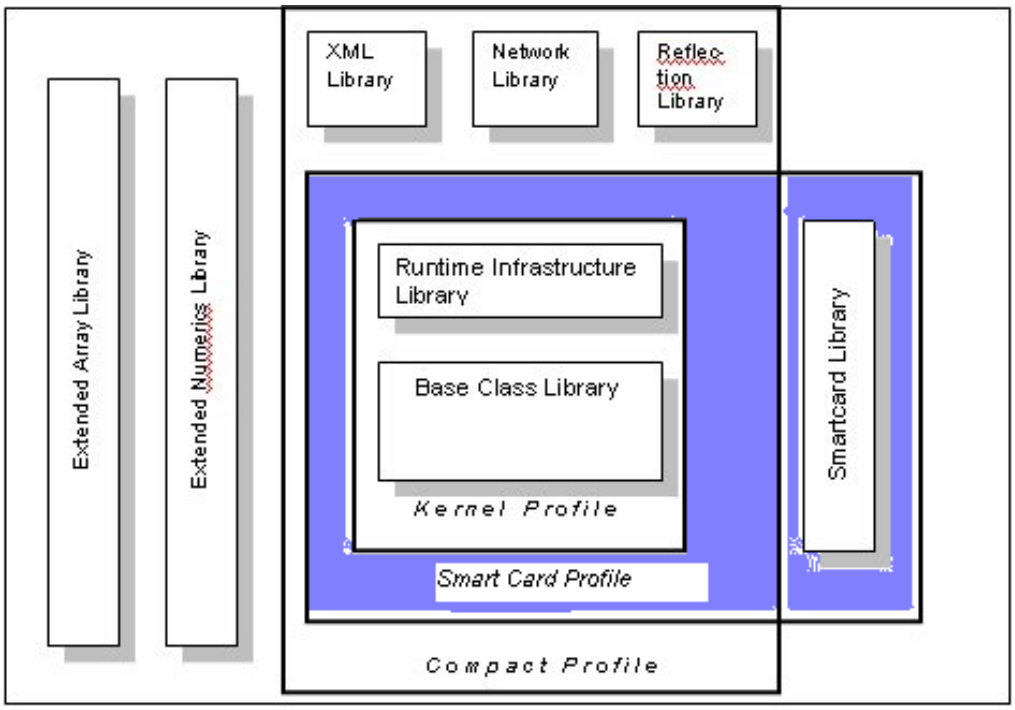
A profile is a set of libraries grouped together to provide a fixed level of functionality. The Smart Card profile allows implementation of .NET on devices with small amounts of both RAM and EEPROM, like smart cards. These devices are generally embedded, must handle random power outages, and have no user interface.

In addition to the Smart Card profile, there are currently two other standard .NET profiles:

- Kernel profile, which is the minimal possible conforming implementation of the Common Language Infrastructure (CLI).
- Compact profile, which allows implementation on devices like mobile phones and personal digital assistants (PDAs).

The graphic shows the relationship between the libraries and the three current .NET profiles.

Figure 8 - Libraries and Profiles Relationship



The Smart card profile (colored purple in the above figure) includes a strict subset of the two .NET libraries that make up the Kernel profile, plus one new, smart card-focused library as listed in the following table.

Table 1 - The Smart Card Profile

File	Description
mscorlib.dll	Strict subset of the base class library
system.xml.dll*	Strict subset of the base class library
SmartCard.dll	Smart card-specific classes.

* The system.xml.dll is present only in older versions of Gemalto smart cards. It wasremoved in OS version 2.1.3.1.

.NET Card Specifications

The core technology implemented in the .NET Smart Card Framework is based on the HiveMinded Smartcard.NET reference implementation of the CLI, which conforms to the European Computer Manufacturers Association (ECMA) Common Language Infrastructure standard, ECMA-335.

IDPrime .NET Card Certifications

One of the versions of the IDPrime .NET card has been certified as FIPS140-2 Level 3.

Common Language Runtime (CLR)

IDPrime .NET Card applications run as managed code within the .NET Smart Card Framework common language runtime (CLR), which is also called the managed runtime environment or execution environment. The .NET Smart Card Framework CLR is a new implementation that is an ECMA-compliant compatible subset of the full .NET CLR, and has been optimized for smart cards and other resource-constrained devices. The common language runtime executes using a CPU-neutral instruction format.

Common Language Runtime (CLR) Responsibilities

Some responsibilities of the CLR are very similar to those of the standard .NET Framework.

- **Application lifecycle management**

Links the card-resident binary (on-card application) and manages execution of the code throughout its lifecycle.

For more information about the application lifecycle implementation in the .NET Smart Card Framework CLR, see “Application Lifecycle” on page 26.

- **Application domain management**

The application domain model enables support for multiple applications running simultaneously and securely on one IDPrime .NET Card. The safety and integrity of each application is assured, because data in one application domain cannot directly reference data in another domain.

For more information about the application domain implementation in the .NET Smart Card Framework CLR, see “Application Domains” on page 25.

- **Garbage collection**

Garbage collection eliminates the need for programmers to explicitly free memory when an application no longer needs it; instead, a system thread periodically examines all objects in the managed heap and removes any object to which all references have disappeared. .NET Smart Card Framework implements a tailored garbage collection mechanism that is well-suited to the resource constraints and particular needs of smart cards.

For more information about the garbage collection implementation in the .NET Smart Card Framework CLR, see “Garbage Collection” on page 34.

- **Remoting management**

Provides an integrated foundation for secure communications between applications using a subset of the .NET remoting architecture.

For more information about the remoting implementation in the .NET Smart Card Framework CLR, see “Remoting” on page 27.

- **Exception handling**

Provides standard exception handling.

Other responsibilities of the CLR are more closely related to the IDPrime .NET Card implementation.

- **Evidence-based security implementation**

The evidence-based security implementation ensures the integrity and authenticity of IDPrime .NET Card assemblies during loading to the card and during execution of the loaded applications.

For more information about the security implementation in the .NET Smart Card Framework CLR, see “Application Security” on page 41.

- **Transaction management**

The .NET Smart Card Framework supports a persistent transaction model that ensures the integrity of data on the IDPrime .NET Card, despite frequent and sometimes unpredictable physical removal of the card from the system or terminal with which it is communicating. The transaction management system includes a new caching technology that greatly increases the speed of writes to EEPROM, while still maintaining data integrity.

For more information about the transaction management implementation in the .NET Smart Card Framework CLR, see “Transactions” on page 37.

- **Code access security**

Very similar to “Data Security”; that is, a public key token is required for an assembly to access a dependent library. To enable a library to be shared with another assembly, the corresponding public key token must be added as an attribute. Security policy is determined by the Access Manager.

.NET Smart Card Framework Vs. .NET Framework

The .NET Smart Card Framework is very similar to the .NET Framework in most substantive ways. These are the features and alternate implementations that are part of the .NET Smart Card Framework that are not in the .NET framework:

- A common language runtime (CLR) that contains the elements needed to manage applications loaded onto an IDPrime .NET Card (see “Common Language Runtime (CLR)” on page 20 for details).
- A special upload file format optimized for smart card profile devices. This is an alternative that produces a much smaller (by a factor of 4) binary file than a full .NET assembly, better suited to the constraints of a smart card.
- The .NET Smart Card Framework has been adapted to accommodate the smart card memory model, in which an application is stored in persistent memory and activated when an external application talks to it.
- Floating point-based types are not supported.
- Non-vector arrays (arrays with more than one dimension or with lower bounds other than zero) are not supported in the .NET Smart Card Framework.
- Reflection is not supported in the .NET Smart Card Framework.
- The .NET Smart Card Framework supports server-side remoting only.
- The *varargs* feature set (supports variable length argument lists and runtime-typed pointers) is not supported in the .NET Smart Card Framework. However, the .NET Smart Card Framework supports runtime-typed pointers.
- Assembly scope names are ignored in the .NET Smart Card Framework, and types are identified by their name alone. Two types with the same name in different assemblies are considered to be identical. Only the method signature default calling convention is supported.
- There are no implicit types in the .NET Smart Card Framework CLR. All types are explicitly defined in the metadata loaded into the CLR. In the presence of multiple loaded assemblies, it is possible to have multiple definitions for types that might normally be implicit. However, the CLR treats these multiple definitions as if there was a single one; there is no way to distinguish if there is one definition or several.
- Asynchronous calls are not supported in the .NET Smart Card Framework.
- Only *BeforeFieldInit* type-initializers are supported the .NET Smart Card Framework; all other initializers are considered to be errors.

- Finalizers are not supported in the .NET Smart Card Framework.
- New slot member overriding is not supported in the .NET Smart Card Framework. The existing slot for member overriding is supported.
- (Class Layout) Only autolayout of classes is supported in the .NET Smart Card Framework. (The loader is free to lay out the class in any way it sees fit.)
- The zero init flag is not supported in the .NET Smart Card Framework; local and memory pools are never initialized to zero.
- Locks and threads are not supported the in .NET Smart Card Framework; therefore, any types associated with these constructs are not supported.
- The security descriptor method state is not supported in the .NET Smart Card Framework.

3

Concepts and Models

The .NET Smart Card Framework supports runtime features that are described in “Chapter 2 - The IDPrime .NET Card”. This chapter, expands on those descriptions.

Assemblies

Compiled .NET software is typically distributed in the form of assemblies. Assemblies perform a number of different functions in .NET, including containing the executable code as well as defining type, security, and reference boundaries. Microsoft’s .NET Framework Developer Center provides detailed documentation of various aspects of .NET assemblies in . We assume that you are familiar with concepts related to Microsoft .NET assemblies.

The IDPrime .NET Card uses assemblies in the same manner as a Microsoft .NET environment. However, IDPrime .NET assemblies go through a conversion process in order to optimize for space usage and to ensure that the assembly does not use types that are unsupported on the card. This conversion process is hidden from the user, and is performed automatically at compilation time when developing an application using Visual Studio .NET.

Assemblies on the IDPrime .NET

There are a number of important points to keep in mind as you develop assemblies for the IDPrime .NET:

- Assemblies loaded to the card must be strong name signed (see “Glossary”). Assemblies that are not strong name signed will be rejected by the smart card. The manifest (see “Glossary”) of the signed assembly contains the public key of the key pair used for the signature, which enables certain functionalities such as:
  - It allows the smart card runtime to verify the integrity of an assembly being loaded.
  - The public key token associated with the assembly is used to grant or deny the assembly access to certain file system resources, and to grant or deny inter-application remoting calls.
- Because assemblies and types on the card are considered unique after they are signed, it is not possible to download more than one copy of an assembly to the card, even if the assemblies are in different directories. It is, however, possible to have two assemblies that contain the same types and namespaces, as long as the two assemblies are not signed with the same key.
- As a corollary to the above point, side-by-side execution on the card of different versions of the same assembly is not supported.

- An executable assembly can use the types defined in the library assembly even if they do not reside in the same directory.
- The IDPrime .NET Card has only limited support for Reflection. Only System.Reflection.Assembly and System.Reflection.AssemblyName classes with few methods are supported.
- IDPrime .NET assemblies can register themselves for deletion. An assembly might choose to do this, for example, when the application has expired (for example, a coupon application) or when an application felt it was under attack. For details on the self-deletion process, please refer to the online help (OLH) documentation in **Start** > **Programs** > .**NET Smartcard Framework SDK** > **X.X.XXX** > **Help** > .**NETSmartcardFramework**.

Note: Where X.X.XXX is the SDK version number.

In the OLH, see **System.Reflection > Assembly Class > Methods > RegisterExecutingAssemblyDeletion Method**.

Assembly Security

Security privileges of an assembly are controlled primarily by the public key token of the assembly. One assembly can grant or deny access to its methods or data by adding or removing the public key token of another assembly to or from its access control lists. For more details see “Data Security” on page 44.

When a new assembly is developed and loaded in the .NET card, it is recommended that this assembly does not use the card file system, or at least does not store any critical data in the card file system. The critical data of the assembly will be better protected if kept in the assembly’s private memory.

Loading Assemblies

Assemblies can be loaded to the card using the IDPrime .NET Card Explorer tool, an NAnt Task, or an API exposed by the SmartCard.CardAccessor.CardAccessor class.

For example:

```
// This code loads an assembly from the D:\Projects directory of the
// local hard drive to the C:\Pub directory of the card, and then
// executes the assembly.
CardAccessor ca = new CardAccessor("Schlumberger Reflex USB v2");
ca.LoadAssembly(@"D:\Projects\MyAssembly.exe", @"C:\Pub");
ca.ExecuteAssembly(@"C:\Pub\MyAssembly.exe");
```

- For more details on the CardAccessor API, see the IDPrime .NET Smart Card Client API documentation in **Start** > **Programs** > .**NET Smartcard Framework SDK** > **X.X.XXX** > **Help** > .**NETSmartcardClientAPI**.

Note: Where X.X.XXX is the SDK version number.

In the OLH, see **SmartCard.Accessor > CardAccessor Class**.

Application Domains

The application domain model enables support for multiple applications running simultaneously and securely on one .NET Card. In the .NET Smart Card Framework runtime, every application executes in a secure, isolated execution area, which enforces strict firewalls between applications and helps maintain data integrity. Data in one domain cannot be accessed from any other domain. For more details about the .NET application domain model, see Microsoft's .NET Framework Developer Center.

Implementation

The .NET Smart Card Framework uses the type *System.AppDomain* to isolate running instances of applications from one another by executing each instance in an isolated environment. The safety and integrity of each application is assured, because data in one application domain cannot directly reference data in another domain.

An application domain serves as the container for assemblies and types when loaded into memory at runtime. It can be useful to think about an application domain as the logical equivalent of a process in a Win32 application. Similar to processes, application domains can be started and stopped independently.

Differences between IDPrime .NET Application Domains and Standard .NET Application Domains

There are some key differences to keep in mind between application domains on the IDPrime .NET card and normal .NET application domains. On the IDPrime .NET Card:

- The *ExecuteAssembly* method of an *AppDomain* can only be executed from off the card, either through the Card Explorer or through the *SmartCard.CardAccessor.CardAccessor* API (see the IDPrime .NET Smart Card Client API documentation in **Start** > **Programs** > .**NET Smartcard Framework SDK** > **X.X.XXX** > **Help** > .**NETSmartcardClientAPI**. In the OLH, see **SmartCard.Accessor > CardAccessor Class**.
- An instance of AppDomain cannot be created by an application on the card.
- If an application domain does not create a service, the application domain will be garbage collected. For example, if application alpha.exe does not create a service, the .exe file will remain on the card after execution, but there will be no running application domain. If alpha.exe DOES create a service, the alpha.exe application domain continues to run (even after the Main method exits) until the service is deleted.
- One application domain can communicate with another application domain indirectly by using Activator.GetObject to obtain a reference to a remoted object in the other application domain. For more details on this process, see the documentation on Remoting.
- An application domain can delete itself by using the static AppDomain.Unload method. An application might be interested in unloading itself if it were an application that were time- or usage-based (such as a coupon application), or if the application were to reach an unrecoverable situation due to a security breach.

For example:

```
if (timesUsed > 10)
     AppDomain.Unload(AppDomain.CurrentDomain);
```

Application Lifecycle

An IDPrime .NET Card application is managed by the common language runtime (CLR) throughout its lifecycle, beginning when it is converted to a binary format that can be loaded onto the card.

Loading

An IDPrime .NET Card application can be created using any supported .NET programming language (for example, C# or VisualBasic.NET). After the code is written, it is compiled to .NETs Microsoft intermediary language (MSIL) format. This compilation produces a standard .NET assembly.

Before it is loaded onto the card, the .NET assembly is converted from its standard compiled form to the .NET Smart Card Framework's card-resident binary format, which produces a much smaller (by a factor of 4) binary than a full .NET assembly. The converted binary is called a card-resident binary. The converted binary must be strong-name signed. This is the application file that is loaded onto the IDPrime .NET Card.

Installation

Each executable binary has a single entry point, a method of the following form:

```
public static int Main
```

After successfully loading and linking a new binary onto the card, the Main method is called to execute an application-specific installation. The application must also register its remote types with the .NET Smart Card Framework runtime to allow clients to remote call methods on the newly installed application.

See “Remoting” on page 27 for more information about remoting.

Execution

The .NET Smart Card Framework implements a client/server model, in which the host system or terminal is the client, and the IDPrime .NET Card is the server. Interaction is always initiated by the client using a request/reply protocol.

Server applications running on the card are persistent; that is, the applications do not terminate when power is turned off (when the card is removed from the reader or terminal) or the card is reset. If one of these events occurs, when the card is reconnected, the application's state does not change from its previous state.

See “Transactions” on page 37 for more information about transaction persistence.

Termination

A service on the card stops running when the service is unregistered. You can do this both programmatically and by using the Card Explorer tool. When the service is unregistered, the running instance is deleted, and its memory is reclaimed by the garbage collector.

Unloading

After a service has been terminated, the binary containing that service can be removed from the card. A loaded assembly that is still exposing a service cannot be unloaded. The service must be terminated first.

Remoting

Remoting in the .NET Framework allows one operating system process or program to communicate with another process running on the same computer, or on two computers connected by a local network or the Internet.

.NET remoting provides an abstract approach to inter-process communication that separates the remotable object from a specific client- or server-application domain and from a specific mechanism of communication. As a result, it is flexible and easily customizable.

The .NET Smart Card Framework extends standard .NET remoting and allows a program executing on a PC to communicate with a process running on an IDPrime .NET Card, and also allows a program running in one application domain to access code or data in a process running in another application domain within the IDPrime .NET Card. For more information about application domains, see “Application Domains” on page 25.

Remoting in the .NET Smart Card Framework

Remoting works by having a server application expose an object to the external world by registering the object as a service either through the `RemotingConfiguration.RegisterWellKnownServiceType` method or through the `RemotingServices.Marshal` method. In order for an object to be registered as a service in .NET, that object must inherit from one of the marshalling base classes. Although the .NET framework supports marshalling either by reference or by value, the .NET Smart Card Framework supports only marshalling by reference (that is, a class extending System.MarshalByRefObject). After the server has registered the object, the object becomes available to clients that connect to the server. When the client connects to the server, it creates a local proxy of the server object. When the client wants to call a method on the remote object, the proxy object passes the method call to the system-remoting mechanism, which is responsible for marshalling the parameters and return value of the method. The current implementation of the .NET Smart Card Framework does not support the marshalling of classes. However, it does support the marshalling of all value types (including structs) and supports both out and ref parameters. Types that can be marshalled include the basic value types (byte, short, char, int, long, string, etc), structs, arrays of basic types, and MemoryStreams.

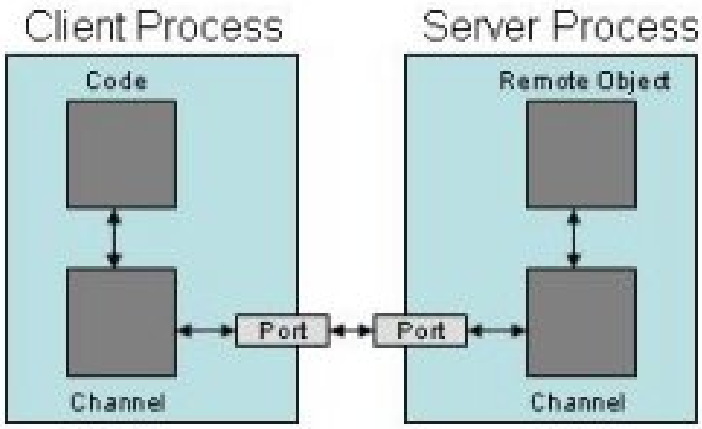
The mechanism by which a client connects to the server is completely isolated from the marshalled object. Conventional .NET remoting applications generally use either TCP or HTTP as the transport protocol for applications. The .NET Smart Card Framework uses ISO 7816-4 as a transportation protocol for communication. However, because the transportation protocol is isolated from the service object, a developer does not have to worry about the actual protocol of ISO 7816-4 communication.

All communication between the client and server takes place through a channel. The .NET Smart Card Framework defines a new type of channel known as an APDUChannel. This is referenced on the server (card) side through the APDUServerChannel class and on the client (PC) side through the APDUClientChannel class. The APDUChannel is responsible for encoding method calls to a binary format and transporting them from the client to the server using the ISO 7816-4 protocol.

Channels and Ports

In the .NET Framework, when you create a server (that is, a remotable class), you also define and register one or more channels for the class and associate each channel with a specific port. By registering a channel/port combination, you tell the .NET infrastructure to listen to that port for messages intended for that channel. When a message arrives, the framework routes it to the correct server object.

Figure 9 - Client - Server Communication Using Channels and Ports



“Figure 9” illustrates how the client and server communicate using channels and named ports in the .NET Framework.

In the .NET Smart Card Framework, identifying a specific port to associate with a channel is not always feasible, so a new mechanism for specifying the channel mode has been created. See “Server Sample Code” on page 29 for details about creating a .NET Smart Card Framework server that makes objects available for remoting.

In the .NET Framework, your client code also creates a channel associated with a specific port, and then uses the Activator class to obtain a reference to the remote object. You identify a remote object with the URL of the computer on which it is located, the name of the remote class, and a URI that you assign.

In the .NET Smart Card Framework, you also use the Activator class to obtain a reference to the remote object, using the new mechanism for specifying the channel mode previously mentioned.

The APDUChannel supports URL's of the format:

```
"apdu://<name of the smart card reader>:<the port on which the service is registered>/<the name of the service>"
```

For example:

```
"apdu://Gemalto Reflex USB v2:2222/CardService"
```

In addition to explicitly naming the reader to connect to, you can also use the reserved names “promptDialog” and “selfDiscover”. The promptDialog mechanism displays a dialog box and allows the user to select which reader to use. The selfDiscover mechanism attempts to find the requested service by attempting to connect to any .NET smart cards attached to the machine.

A simple .NET Smart Card Framework remoting example follows.

Example

Server Sample Code

First, create the server, which listens for calls from clients and connects them to the remotable class. Here's what the code does:

1. Creates a new APDUServerChannel; in the sample code:

   ```
   APDUServerChannel chan = new APDUServerChannel()
   ```

2. Registers the channel with the .NET Smart Card Framework infrastructure; in the sample code:

   ```
   ChannelServices.RegisterChannel(chan)
   ```

3. Registers the remotable class using a call to the `RemotingConfiguration.RegisterWellKnownServiceType()` method; in the sample code:

   ```
   RemotingConfiguration.RegisterWellKnownServiceType(typeof(My
   RemoteClass),"MyServiceURI", WellKnownObjectMode.Singleton;
   ```

The arguments to this call are:

- The first argument identifies the class being registered; in the sample code:

  ```
  typeof(MyRemoteClass)
  ```

- The second argument specifies the URI for the class; in the sample code: `"MyServiceURI"`. The client will use this URI when calling the class.
- The third argument specifies that if there are multiple calls to the class (from more than one client), they will all be serviced by the same instance of the class, in the sample code: `WellKnownObjectMode.Singleton`. The two available modes are:
  - *SingleCall* - Single Call objects service one and only one request coming in. Single Call objects are useful in scenarios where the objects are required to do a finite amount of work. Single Call objects are usually not required to store state information, and they cannot hold state information between method calls.
  - *Singleton* - Singleton objects service multiple clients and, hence, share data by storing state information between client invocations. They are useful in cases in which data needs to be shared explicitly between clients and also in which the overhead of creating and maintaining objects is substantial.

The sample server code follows.

Figure 10 - Sample Server Code

```
using System;
using System.Runtime.Remoting.Channels;
using SmartCard.Runtime.Remoting.Channels.APDU;

namespace RemotingDemoServer{

   public class MyRemoteClass : MarshalByRefObject
   {

      public MyRemoteClas()
      {
      }

      public String SayHello(string name)
      {
         return "Hello" + name;
      }

      public static void Main()
      {
         APDUServerChannel chan = new APDUServerChannel();
         ChannelServices.RegisterChannel(chan);
         RemotingConfiguration.RegisterWellKnownServiceType(typeof(MyRemoteClass),
                                 "MyServiceURI", WellKnownObjectMode.Singleton);
      }
   }
}
```

Sample Client Code

Next, build the client that will call the remotable class. The program does the following:

1. Creates an APDUClientChannel.
2. Registers the channel with the .NET Smart Card Framework infrastructure.
3. Attempts to obtain a reference to the remote class by calling the `Activator.GetObject()` method.
4. If the program cannot obtain a reference, it displays a message to the user. Otherwise, it calls the remote object's `SayHello()` method and returns “Hello” + name.

The `Activator.GetObject()` method accepts two arguments: The first is the type of the remote class, which you can obtain by using the typeof() method with the class's namespace and name as argument; the second argument has the following parts:

- `apdu://` - identifies the protocol; “apdu” is specified because the client and the server are using an APDUChannel for communication.
- `<mode>` - identifies the card connection mode to work around the fact that in the .NET Smart Card context, the port through which the secure channel will work cannot always be identified. In the sample code, the mode is `prompt:unknown`, which means that the application will display a dialog box in which the user will be required to select the reader in which the .NET Smart Card is inserted and active.

These are the available modes:

Mode	Description
`prompt`	Application displays a Select Card Reader dialog box requiring the user to select which reader to use.
`<hard-coded reader>`	Specifies the reader that will be used, for example, `Gemplus USB Smart Card Reader 0`. The exact string for each reader that can be specified matches the reader options that display in the Select Card Reader dialog box drop down list. This mode is appropriate only if you know which reader will be used with your application.
`SelfDiscover`	The application searches for a card containing the service that the application is requesting. If the application does not find a suitable card, the Select Card Reader dialog box is displayed, and the user must select the reader to use.
`SelfDiscoverNoPrompt`	The application searches for a card containing the service that the application is requesting. If the application does not find a suitable card, an exception is thrown. This mode is primarily used for an application that is not allowed to display dialog boxes, for example, Windows services.

- The URI associated with the remote class, which must match the URI established by the server; in the sample code: `MyServiceUri`.

Here is the sample client code:

Figure 11 - Sample Client Code

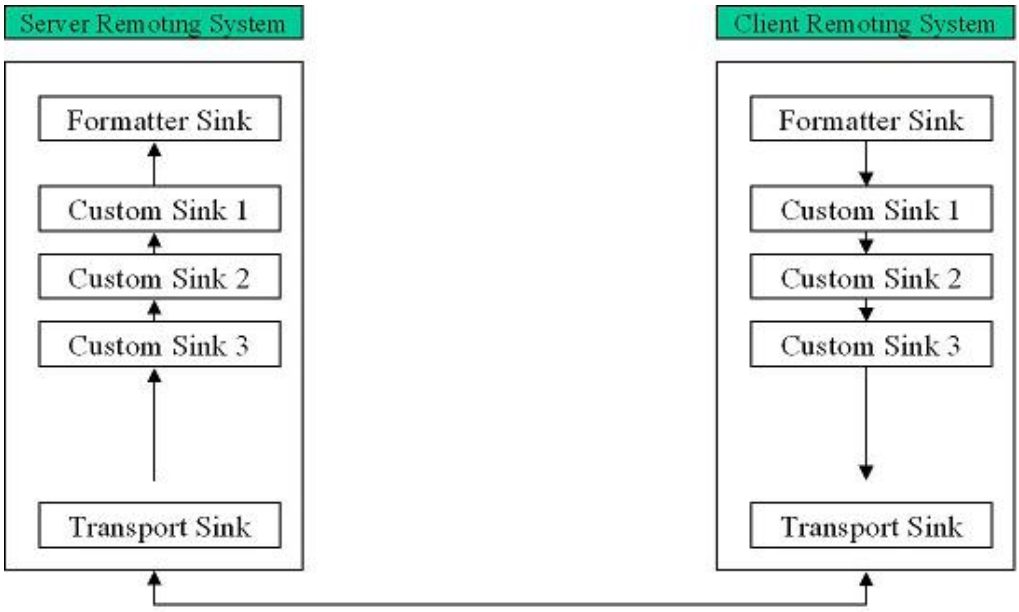
```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.APDU;

using RemotingServerDemo;

namespace RemotingClientDemo
{
   public class Client
   {
      public static int Main(string [] args)
      {
         APDUClientChannel chan = new APDUClientChannel();
         ChannelServices.RegisteredChannel(chan);

         MyRemoteClass obj = (MyRemoteClass)

         Activator.GetObject(typeof(RemotingServerDemo.MyRemoteClass),
         "apdu://prompt/MyServiceUri");

         Console.WriteLine(obj.SayHello("John");

         return 0;
      }
   }
}
```

Using Custom Sinks

As explained in “Remoting” on page 27, the Gemalto.NET card uses common .NET remoting techniques to eliminate the need for explicit handling of APDU's. One exciting feature of .NET remoting is that the remoting architecture can be extended by creating custom sinks and sink providers that perform additional manipulation of remoting data before it is sent to and from the card.

The following diagram shows the .NET remoting architecture with custom sinks sitting between the formatter and transportation sinks on both the client and server.

Figure 12 - .NET Remoting Architecture with Custom Sinks



Why Make a Custom Sink?

If you do not care about the format of the data on the wire between the PC and the card, there is no reason to create a custom sink. However, if you want to encrypt the data between the card and PC or use proprietary compression algorithms on the transmitted data, you can accomplish either of these tasks by creating custom sinks.

What Are the Limitations?

This version of the Gemalto.NET card supports only the use of MemoryStreams and FileStreams within a custom sink. You cannot use either a CryptoStream or a CustomStream as the basis for your sink manipulation. Attempting to use an unsupported stream results in a NotSupportedException.

Designing a Custom Sink

When you design a custom sink, you must write a sink that behaves properly as part of a chain of sinks. You will generally have to implement both a client- and server-sink component. There are slightly different mechanisms for sink implementation depending on whether the sink is a client sink or server sink.

When implementing a server sink, you must implement

- **System.Runtime.Remoting.Channels.IServerChannelSink**. The key method of the implementation is the `ProcessMessage` method, which is responsible for

implementing whatever transformations the sink is responsible for. A server sink may perform either inbound transformations, outbound transformations, or both. It is critical that the server sink also calls the next sink in the sink chain between processing its inbound and outbound data. In addition, you must also implement...

- **System.Runtime.Remoting.Channels.IServerChannelSinkProvider**. This class is responsible for creating the sink object and for calling other providers to create other sinks in the chain.

When implementing a client sink, you must implement

- **System.Runtime.Remoting.Channels.IClientChannelSink**. Again, the key method of the implementation is the `ProcessMessage` method. In this method, you perform outbound processing before passing the message to the next sink in the chain. When the message returns from the next sink, you perform inbound processing. You must also implement...
- **System.Runtime.Remoting.Channels.IServerChannelSinkProvider**. This class is responsible for creating the sink object and for calling other providers to create other sinks in the chain.

Using a Custom Sink

To make use of a custom sink, you must insert it into the sink chain both on the server and on the client.

On the server side, you create your sink provider, and place it **before** the APDUServerFormatterSinkProvider. Then, when you register a channel, you register your sink provider as a parameter to the channel that you will be using.

For example:

```
// Create an encryption sink provider as the first sink in the chain
IServerChannelSinkProvider newProvider = new
EncryptionServerSinkProvider(properties, null);
newProvider.Next = new APDUServerFormatterSinkProvider();

// Register the channel the server will be listening to.
ChannelServices.RegisterChannel(new APDUServerChannel(newProvider,
0x7867));
```

On the server side, you create your sink provider and place it **after** the APDUClientFormatterSinkProvider. Then, when you register a channel, you register the APDUServerFormatterSinkProvider as the parameter to the channel that you will be using.

```
// Create an encryption sink provider as the second sink in the chain
IClientChannelSinkProvider newProvider = new
APDUClientFormatterSinkProvider();
newProvider.Next = new EncryptionClientSinkProvider(properties);

// Register the communication channel
ChannelServices.RegisterChannel(new
APDUClientChannel("EncryptionSinkClient", newProvider));
```

Garbage Collection

The .NET Smart Card Framework Common Language Runtime (CLR) includes automatic memory management using a mechanism called garbage collection.

Garbage Collection

The garbage collection feature eliminates the need for programmers to explicitly free memory when an application no longer needs it; instead, the runtime manages memory. All objects are instantiated while under the control of the managed runtime, and the code is verifiably type-safe. The garbage collector then detects when there are no longer any program references to an object, and deletes all objects that are no longer required, freeing up memory previously allocated to those objects.

In the .NET Smart Card Framework runtime, garbage collection takes place in response to the following events:

- When the card is reset or powered up.
- After execution of a remoting call in which either a new object was allocated, or an uncaught exception is thrown. This is an important point to note, because if your method does not allocate an object, the garbage collector will not be invoked. You can force garbage collection in a method that does not allocate an object by using the **GCControlAttribute**

Note that unlike garbage collection on other platforms, the .NET Smart Card Framework garbage collection does not take place during execution of an application or during allocation of objects or arrays, nor does garbage collection automatically execute when the system runs out of memory. This means that if you are creating a large group of objects and setting them to null, they will not be collected until the end of the remoting call. Because of this constraint, you need to design your application in such a way that it does not rely on memory being freed immediately after objects are set to null or no longer referenced. Since a smart card is a resource-constrained device, it is recommended that you should try to keep object creation and deletion to a minimum. More object creation and deletion implies more garbage collection time, which might impact the performance of the application.

Starting the Garbage Collector Manually

If your application accesses the card module assembly directly without accessing the minidriver .dll on the client PC, the garbage collector starts automatically only when the card is low on memory. This can slow processing considerably when performing a reset. To avoid this, you can start the garbage collector manually by sending an APDU with the `ForceGarbageCollector` method, listed in “Table 17 - Hivecodes for V6/ V7” on page 144.

Note: This method is not available for the V5 version of the minidriver.

The GCControlAttribute

Although the standard behavior of the garbage control system satisfies most programming needs, there are times when you may wish to exercise more control over the garbage collection process. The GCControlAttribute allows you to either force or skip garbage collection after a remoting method.

For example:

```
[GCControl(GCControlMode.Force)]
void MyMethod()
{
  myMemberArray = null;
}
```

Normally, garbage collection would not be invoked after this method, since there is no object allocation in the method. However, in this case, garbage collection will still be invoked because we've used the GCAttribute with the GCControlMode.Force parameter. This might be useful if **myMemberArray** was large, and we wanted this memory to be available for the next remoting call.

Alternatively, we might want to skip garbage collection on a given method:

```
[GCControl(GCControlMode.Skip)]
void MyMethod()
{
  myMemberArray = new byte[3];
}
```

Normally the above method would invoke garbage collection since there has been an object allocation. However, for performance reasons, a developer might want to skip garbage collection for this method. Using GCControlMode.Skip causes the system to skip garbage collection for this method.

File System

The IDPrime .NET Card contains a file system that is fully accessible within the card from the standard .NET System.IO namespace. The file system provides developers a mechanism to separate their data from their code. This allows developers to replace an assembly with an updated version without losing data that might be associated with that assembly.

Key Points about the IDPrime .NET File System

The IDPrime .NET file system does differ slightly from a conventional Windows file system, and a developer should keep these differences in mind when designing an application to run on the card.

- Only one FileStream may be open on a given file at any time. This means that it is important to release FileStreams after returning from remoting calls in order to avoid blocking another FileStream object from accessing the file.
- FileStreams are closed when the card is Reset or when it is removed from the reader. It is never safe to assume that a FileStream is open and valid unless you have created or opened the stream in the same remoting call.
- In this version of the card, file names are case sensitive. For example, the file **readme.txt** is not the same file as **Readme.txt**.
- Security of the file system is enforced by both the card's operating system using token-based security (see “Data Security” on page 44), and by the current Access Manager using role-based security (see “Access Manager” on page 40).

Example

The following example shows on-card manipulation of the file system by attempting to create a file in each of the subdirectories of D:\Pub

```
public void Example(string filename, byte [] data)
{
   string [] dirs = Directory.GetDirectories(@"D:\Pub");
   foreach (string directory in dirs)
   {
       FileStream fs = new FileStream(@"D:\Pub\" + directory + @"\" +
filename,
```

```
            FileMode.Create);
        fs.Write(data, 0, data.Length);
        fs.Close();
    }
}
```

Data Storage

The IDPrime .NET Card contains both persistent memory and volatile memory that are used for data storage. The persistent memory acts as persistent storage for the card - data persists in it even after the card is removed from a smart card reader. Volatile memory is reset when the card loses power and cannot be used for persistent storage.

Data Stored in Persistent Memory

The persistent memory of the card is used for objects that are created by your application. Any object created with a new keyword (whether this is done by the developer or by an underlying software package) is created in persistent memory, and will remain on the card until it is no longer referenced and has been garbage collected (as described in “Garbage Collection” on page 34). In addition, any fields of an object will be stored in persistent memory.

Data stored in the file system (see “File System” on page 35) is always stored in persistent memory.

Data Stored in Volatile Memory

Local variables and parameters are stored in volatile memory.

The following code snippet illustrates which data is stored in persistent and volatile memory:

```
// My class, when it exists, will always exist in persistent memory,
since it is created
// by an application via a call like "MyClass mc = new MyClass();"
class MyClass
{
   // My fields will be in persistent memory
   int f1;
   short f2;

   // In this method, neither of the parameters are stored in persistent
memory.
   public void MyMethod(int param1, param2)
   {
      // Neither of these local variables is in persistent memory
      int i = 0;
      int j = param1;

      // But the fields of the class do remain in persistent memory.
Even after
      // reset, f1 will retain its value
      int f1 = f1 + param2;
   }
}
```

MemoryStreams

There exists a special case of objects that exist in both persistent and volatile memory. The MemoryStream object itself is in persistent memory, but the data to which it is pointing will exist in volatile memory. When a card is reset, any memory streams are disposed, and the data to which those streams pointed is lost. Memory streams provide a fast mechanism for manipulating arrays of byte data because the data is manipulated in fast volatile memory rather than slower persistent memory.

Transactions

The .NET Smart Card Framework supports a persistent transaction model that ensures the integrity of data on the IDPrime .NET Card, despite frequent and sometimes unpredictable physical removal of the card from the system or terminal with which it is communicating.

Why Transactions?

A smart card can be removed from a reader at unpredictable times. When the removal occurs, the card will lose power immediately. This can be a serious problem if you were in the middle of updating a sequence of object fields. For example, you might be updating an address in an object. If you update the “street” field, but the card is removed before you update the “city” field, you could end up with an address that is completely incorrect. You need a way to ensure that either all of the updates take place, or none of them do.

Card removals are not the only interruption that you might worry about. You might be concerned that an exception could be thrown in the middle of some field updates that could leave the card in an inconsistent state. In this case, you would want a mechanism for rolling back any field updates to the original state.

How Transactions Work

Transactions work by ensuring that changes are not committed until the end of a transaction. When you create a transaction, the card preserves the state of the object before the transaction began, and will revert to this state at power up if the transaction was unable to complete.

Any method (including the method initially called by the client) can be marked as a sub-transaction by the addition of a special transaction attribute, `Transaction`. Also, any method that is called by a method that is under transaction is also considered to be under transaction.

Note: If the transaction method returns an uncaught exception, the transaction is not committed, and objects and static data fields are returned to their previous state.

Example

In this example, the `Increment` method is marked as a transaction using the `Transaction` attribute.

```
[Transaction]
private void Increment ()
{
    counter++;
    if (counter > 10)
    {
        throw new BadCountException();
    }
}
```

In the example, a counter is incremented, and if the counter is greater than 10, an exception is thrown. The exception is not caught in this method. (It is intended to be caught by the caller.) Because executing the method results in an uncaught exception, the sub-transaction aborts and any changes made by this method are rolled back. The result is that the value of the counter will never exceed 10.

Out-of-Transaction Objects

Although in general, you would like to roll back any modifications made to your card if the operation is interrupted, there may be cases where you might want the method to be under transaction, but for a particular field of an object to be “out of transaction”. One motivation for this is the PIN class. You can imagine that the logic for a PIN class might be for the caller to send PIN data to a method, and the method would then pass the data to the PIN object. If the data does not match the PIN, the number of remaining tries on the PIN is decreased, and the method returns. What we want to avoid is for an attacker to be able to try a PIN, cut power to the card if it fails, and have the number of remaining tries reset by a transaction.

To avoid this type of attack, the .NET framework provides an **OutOfTransaction** attribute that can be applied to the fields of an object. Fields annotated by this attribute are always considered to be “out of transaction”. That means that even if it is used inside a method that is under transaction, the field will not be rolled back to its previous state if the transaction is interrupted. The PIN class of the card is built using an **OutOfTransaction** attribute.

Here's an example of the OutOfTransaction attribute in action:

```
using System;
using System.Diagnostics;

using SmartCard;
using SmartCard.Services;

public class MySecureCounter
{

   [OutOfTransaction]
   byte counter;

   public void Increase()
   {
      counter++;
   }

   public byte Value
   {
      get
      {
         return counter;
      }
   }
}

public class Test
{
   MySecureCounter secCount = new MySecureCounter()

   [Transaction]
   public void TestWithAbortedTransaction()
```

```
    {
        secCount.Increase();
        throw new Exception(); // abort
    }

    public void TestWithoutTransaction()
    {
        secCount.Increase();
        throw new Exception(); // abort
    }

    static void Main()
    {
        Test test = new Test();

        Debug.WriteLine("initial value = " test.secCount.Value);
            // expect test.secCount.Value = 0

        try
        {
            test.TestWithoutTransaction();
        }
        catch {}

        Debug.WriteLine("second value = " test.secCount.Value);
            // expect test.secCount.Value=1

        try
        {
            test.TestWithAbortedTransaction();
        }
        catch {}

        Debug.WriteLine("third value = " test.secCount.Value);
            // expect test.secCount.Value = 1

    }

}
```

Security

Security in the IDPrime .NET Card is generally discussed in one of three contexts:

- **Access Manager**. The IDPrime .NET Card supports an extensible access management system that allows developers and card deployers to define user roles that can manage the card. These user roles control the deployment of new assemblies to the card, as well as control over the .NET card file system.
- **Application Security**. Applications deployed to the IDPrime .NET Card are always signed assemblies. The public key token of these signed assemblies is used to grant or deny privileges to a given application. For example, a library assembly installed on the card might restrict unknown assemblies from using its API.
- **Data Security**. Data for IDPrime .NET applications can be stored either internally to the application or in the IDPrime .NET file system. Applications making use of the file system can be assured that file-based data is secured by access control lists associated with the public key tokens of on-card assemblies.

Access Manager

The resources such as files, directories, assemblies and their management in IDPrime .NET Card are accessible using the ContentManager service (described later in the documentation). Since these resources should only be accessed and managed by authorized entities, mechanisms for authentication and authorization are required. It is also envisioned that during the life cycle of the card, these mechanisms may need to be changed. For example, a manufacturer of a smart card may trust a particular kind of authentication mechanism that an issuer of the same smart card may think is insufficient and weak. IDPrime .NET Card provides a flexible and extensible application model such that any actor (provided it has authorization) in the lifecycle of the smart card can implement its own authentication and authorization mechanisms. Some of the authentication mechanisms for smart cards that are prevalent today are PINs, mutual authentication using symmetric key algorithms, Biometric and so on.

The service that implements the above-mentioned authentication and authorization specifics is called an **AccessManager** service. Like all other services, an AccessManager service is a .NET Remoting application and is developed with the requirement that it should extend an abstract class **SmartCard**.**AccessManager** of the **SmartCard.dll** library.

The **SmartCard.AccessManager** class provides two abstract methods that should be overridden by the extending class. These methods are:

- `ResourceAccessPolicyCheck`(string resource,AccessType accessType) : This method is invoked by the ContentManager service with the name of the resource (such as a path of a file) and the type of resource as arguments. Implementation of this method will decide whether to grant access or not. If access is not granted, an AccessManager implementation throws an UnauthorizedAccessException.
- `AccessManagerRegistrationPolicyCheck`(string objectUri,string assemblyPath) : As mentioned above, the AccessManager can only be changed by an authorized entity, and this method provides a way to determine if an authenticated authority has the privileges to do so. This method is called when the `RegisterAccessManager` method of ContentService is invoked. The name of the service that is to be made the new AccessManager and the path to the assembly containing the implementation class are passed as arguments. If the current AccessManager does not entertain this request, an UnauthorizedAccessException is thrown.

Since the above-mentioned methods of the AccessManager service are invoked whenever a resource is accessed, it is recommended that implementors of the AccessManager service should pay special attention to performance requirements and memory consumption. Also, when designing an AccessManager service, the policies to delegate control to a new AccessManager should be known in advance and be well thought out.

The IDPrime .NET SDK also provides a flexible and extensible mechanism for client applications to communicate with an AccessManager service on the card. An interface called **IAccessManagerClient** in **Gemalto.SmartCard.Runtime.dll** is provided that should be implemented and registered. A new AccessManager client application is registered by adding the path to its assembly in the Config.xml file, which is located in the “[INSTALLDIR]\Gemalto\NET SmartCard Framework SDK\vX.X.X.*\bin” directory.

Note: Where X.X.X is the SDK version number

The **SmartCard.Acessor.CardAccessor** class of the **SmartCard.Runtime.dll** provides methods to determine the correct AccessManager client. Toolbar buttons of the Card Explorer, provided in the SDK, change their behavior depending on the AccessManager client registered.

The IDPrime .NET Card shipped in the SDK contains an AccessManager service called **SampleAccessManager**, which uses username/password-based authentication and controls the access to resources using a role-based security mechanism. It is described in detail in “SampleAccessManager” on page 49.

Application Security

This section discusses issues related to ensuring the integrity and authenticity of card-resident binaries, as well as ensuring the security of applications running on the card.

Ensuring the Integrity and Authenticity of Card-Resident Binaries

To ensure the integrity and authenticity of a binary that will be loaded to an IDPrime .NET Card as a card-resident binary, converting a .NET assembly to a card-resident binary is a two phase process:

1 The .NET Assembly is converted to an interim binary format. These are the components of the interim binary that is produced by conversion:

- `metadata`
- `code`
- `public key`

Note: The metadata for a card-resident binary consists only of name and version information for the binary (in contrast to the metadata in a full .NET assembly, which includes a much larger set of data describing the assembly).

2 A hash is computed (using the SHA1 algorithm) based on the interim binary's components, and then the hash is encrypted using the private key associated with the interim binary's public key pair. The encrypted hash (which is also called the signature) is stored as part of the binary. The components of the binary after addition of the signature are:

- `metadata`
- `code`
- `public key`
- `signature`

A binary that includes a signature is ready for upload to the IDPrime .NET Card, where it becomes a card-resident binary. The binary can be loaded using the CardManager service, the Card Explorer, or the CardAccessor library, which provides a wrapper for CM methods.

During upload of the binary to the card, each block of data is checked for accuracy (unrelated to integrity/authenticity). If all blocks are loaded successfully, a hash is computed using the public key, which has been extracted from the binary on the card. The encrypted hash (signature) loaded onto the card is then decrypted, also using the public key.

If the decrypted signature matches the hash computed on the card using the public key, both the integrity and authenticity of the data are proven. Integrity is demonstrated because the hash values match. Authenticity is demonstrated because the public key can only be in the .binary loaded onto the card if the private key was known.

If the values do not match, the data integrity and authenticity of the binary cannot be assured, and a BadImageFormatException exception is thrown. (The binary is not loaded to the card.)

Matching the Card-Resident Binary to the Original .NET Assembly

A pre-conversion .NET assembly consists of these components:

- metadata
- code
- public key
- signature

When the original .NET assembly is converted to the card-resident binary format, the resulting binary file (.bin) is stored in the original assembly as a resource. After the .bin file is embedded, the original .NET assembly is re-signed so that the data in the .bin file is accounted for in the original .NET assembly's encrypted hash (signature).

Thus a post-conversion .NET assembly consists of these components:

- metadata
- code
- public key
- .bin resource
- new signature

The .NET assembly hash is stored in the converted binary and gets loaded onto the card along with the rest of the data in the card-resident binary. If needed, this hash value can be used to match the card-resident binary to the original .NET assembly on the desktop.

Ensuring Code Security

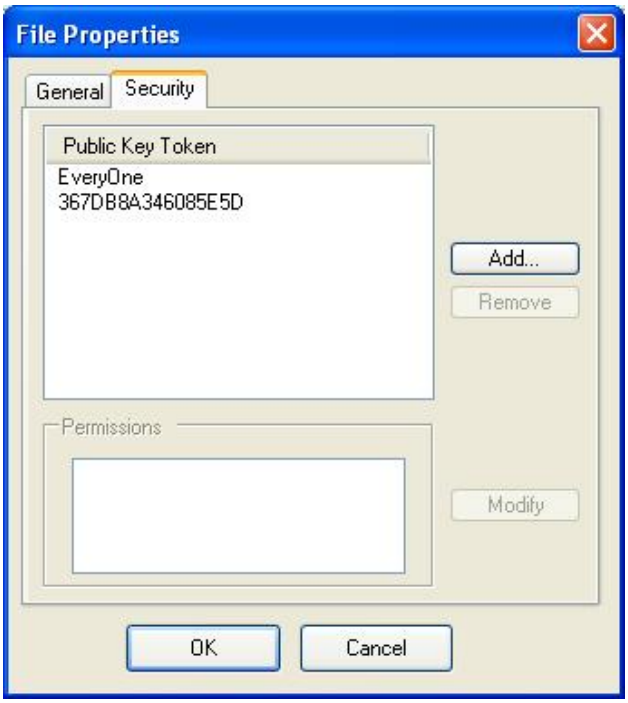
Code security is addressed by the following mechanisms:

- Requiring that all assemblies must be signed.
- If an assembly (A1) needs to access another assembly (A2), either both assemblies must have the same public key token, or assembly A1, whose public key token is PBKT1, must be granted access to assembly A2 by adding public key token PBKT1 as an attribute on assembly A2.

  A new public key token can be added as an attribute from the properties page for the assembly as follows:

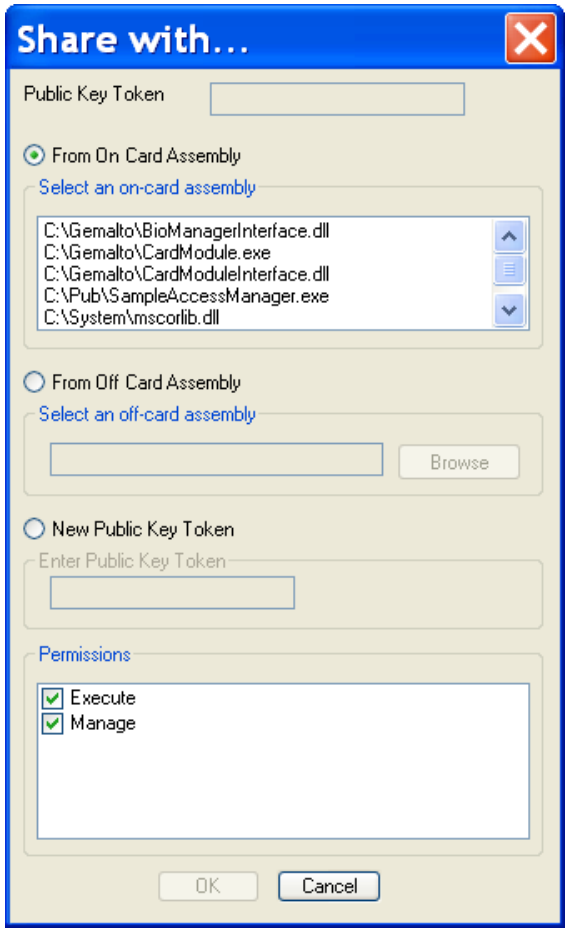
1. Open the properties page for assembly A2 and click the **Security** tab (as shown in “Figure 13”).

Figure 13 - Properties Page for Assembly



2 Click **Add** to open the **Share With** dialog box (“Figure 14”).

Figure 14 - Share With Dialog



3 Do one of the following:

- If the A1 assembly is in the card, choose **From On Card Assembly**, select the A1 assembly from the list
- If the A1 assembly is stored on the computer, choose **From Off Card Assembly**, click **Browse** and navigate to the assembly on the computer.
- Click **New Public Key Token** by typing or pasting the public key token for assembly A1.

4 In **Permissions**, check the boxes that correspond to the access rights you want to grant to the public key token attribute.

Access rules enforced by the current Access Manager define who is able to add public key token attributes to an assembly.

5 Click **OK**.

If an assembly does not have any public key token (PBTK) attributes, it is considered to be public; that is, the assembly is accessible by all other assemblies. For example, assemblies located in C:\System (including mscorlib.dll and SmartCard.dll) are accessible to all assemblies. You can confirm that an assembly is public by viewing its properties; click the **Security** tab and verify that the Public Key Tokens list is empty.

Data Security

The IDPrime .NET card supports two different types of data storage on the card. In the first type of storage, data is stored as objects in the Application Domain of its host assembly. (See “Application Domains” on page 25.) In the second type of storage, data is stored in the IDPrime .NET File System. (See “File System” on page 35.)

Data Storage in Application Domains

When data is stored in an application domain (AD), it resides in a strongly typed object (see “Glossary”), and is protected from misuse by the fact that ADs cannot communicate directly with each other. There is a strict firewall between different ADs. In order for two ADs to communicate with each other, one must export an interface, and the calling application must be trusted (see “Application Security” on page 41). The AD hosting data can add further protection to the data by simply not releasing it unless certain security criteria (such as the presentation of a key or PIN) are met. Although the security of this type of data storage is strong, it has certain drawbacks. It ties the data to the AD (remove the AD and you lose your data). The fact that another application cannot access data directly is an advantage from a security perspective, but it can be a disadvantage from a performance perspective.

Data Storage in the File System

As an alternative to using the AD to protect data, an application can choose to store data in the card's file system. Access to the file system on the card is identical to accessing the file system in a normal .NET environment. For example, we can use FileStream to create an object:

```
FileStream fs = new FileStream(@"D:\Pub\MyFile.txt", FileMode.Create);
fs.Write(someDataInAByteArray, 0, 12);
fs.Close();
```

This mechanism of data storage has two key advantages.

- It separates an application from the data it uses. This allows you to delete an application and install a new version without having to worry about otherwise preserving data.

- It simplifies sharing data. For example, one application might have information about the shipping address of the card owner. If this were stored in a file that was accessible to other applications, the same shipping address would be accessible to those applications.

The fact that the file system can be used for data sharing requires that there be policies in place to enforce ownership and sharing privileges on files in the file system.

Data files in the IDPrime .NET card are protected using a public key token system that is very similar to that used by applications (see “Application Security” on page 41). Each file has two sets of privileges associated with it:

- The **public** privileges define what any application can do to the file. For example, a file could have a public Read privilege, which would allow any application to read from the file.
- The **private** privileges are assigned to individual public key tokens. For example, a file might assign read privileges to a particular public key token that is trusted by the application.

By default, when a file is created using the System.IO namespace, no public privileges are assigned to that file, and the public key token of the assembly that created the file is the only public key token in the private privileges set. The creating assembly has full privileges to control the file. If an assembly with a public key token that is not in the private privileges set attempts to perform an operation that is forbidden in the public set, an UnauthorizedAccessException is thrown.

Supporting Legacy Infrastructure

The primary mechanism for communicating with applications on the IDPrime .NET Card is to use the APDUChannel remoting mechanism provided as part of the .NET card framework. However, there are times when using remoting is not suitable. For example, you might be in a situation where your card must work in a non-.NET environment, or must work with existing applications that cannot be ported to the APDUChannel remoting architecture. This section explains how to support legacy (APDU-based) applications using your IDPrime .NET Card.

Who Should Read This Section?

This section is targeted at people who need to support legacy applications on the card. Unless you have a really good reason for doing this, Gemalto strongly recommends using the .NET remoting architecture. If you absolutely must implement an APDU-based application on the card, this section is written for you. However, we assume that you have a basic understanding of the APDU protocol.

The Problem with Legacy Applications

Legacy applications expect to communicate with the card using a series of APDUs. Often, the APDUs will be handled by the card based on the different components of the APDU header. For a full description of the APDU protocol, see the ISO 7816 specifications. By default, applications on the IDPrime .NET Card expect that their methods will be invoked via the .NET remoting architecture. When the remoting architecture is active, all APDUs are processed by the APDUChannel as remoting calls.

Using Attributes to Manage APDUs

The IDPrime .NET Card uses .NET attributes to map APDUs to methods on the card. This is best illustrated by example:

```
[APDUException(typeof(CryptographicException), (short)0x6512)]
[APDUException(null, (short)0x6514)]
[APDU("B0300000",Mask = "00000F0F")]
public void GenerateKeyPair([APDUParam(APDUHeader.P1)]byte
privateKeyIndex,
   [APDUParam(APDUHeader.P2)]byte publicKeyIndex, byte algID, ushort
keySize, byte []
 Data)
{
   ...
}
```

Broadly, here's what this does: It defines a method `GenerateKeyPair` that returns no data and in the normal remoting world would be expecting as arguments 3 bytes, followed by an unsigned short, followed by a byte array. If you wanted to invoke this method using remoting, you'd use:

```
myRemoteObject.GenerateKeyPair(priIndex, pubIndex, algId, keySize,
bData);
```

However, if instead of using remoting, you send an APDU to the card that matches the APDU attribute, the following happens:

1. The P1 byte from the APDU is packed into the first argument.
2. The P2 byte from the APDU is packed into the second argument.
3. The first DATA byte is packed into algID.
4. The second two data bytes are packed into keySize.
5. The remainder of the DATA from the APDU is packed into the Data array.
6. If a CryptographicException is thrown by the method, it will be translated to an SW of 0x6512.
7. If any other exception is thrown by the method, it will be translated to an SW of 0x6514.

What does it mean for an APDU to match the APDUattribute? Basically the check is:

```
if ((incomingAPDU & (~Mask)) == APDUAttribute)
```

So, in our example, any APDU of the form B0300x0x would be dispatched to this method.

Here are the other rules you need to know for writing APDU's from your IDPrime .NET application:

- Your method must either return `void`, `byte[]` or `MemoryStream`
- Your method must take as parameters only basic numeric types (byte, short, int, long or their unsigned variants) or a byte array.
- If your method takes a byte array as a parameter, the byte array must be the last parameter, and there can only be one array parameter.
- The URI of the application that you marshal when you install the application should be a string containing the hexadecimal AID (e.g. private const string `REMOTE_OBJECT_URI = "A00000000101";`. By doing this, your application will respond to standard select APDUs. In the case of the above sample, 00 A4 04 00 06 A0 00 00 00 01 01 would select the assembly.

Additionally, you can define a method to be called anytime that your application is selected using APDU's.

```
[APDU("SELECT")]
public void Select(MemoryStream AID)
```

```
{
    // Do anything you want to do when selected here...
}
```

Returning Data from the Card

The card automatically handles the sending of appropriate 61 XX commands when it wants to send data to the terminal. For example, if you have a method defined as:

```
[APDU("00B20000")]
public byte [] GetAllergyData(int AllergyNumber)
{
   byte [] b = new byte[12];

   // pretend that I've filled b with data about the allergy

   return b;
}
```

When the card receives “00 B2 00 00 04 12 34 56 78", it will respond with “61 0C”, and it will be the responsibility of the terminal to send a **GetResponse** to retrieve the data.

Handling Incorrect Requested Lengths

Situations will occur when a client asks the card for an amount of data that exceeds the available data from a given method. This would occur, for example, when you ask for 20 bytes back from a method that returns a 10-byte array. You can use the **OnInvalidLe** field of the **APDUAttribute** class to specify the behavior. This field can be set to either **APDUInvalidLeAcknowledgeMode.Reject**, in which case the card returns a 0x6700 status word, or to **APDUInvalidLeAcknowledgeMode.IndicateLa**, in which case the card returns a 0x6C[La], where [La] indicates the number of available bytes.

Card Reset Event

Although the IDPrime .NET SDK is designed to insulate developers from smart card operational aspects, it is important to understand the concept of a reset on a smart card, because this is an event that takes place every time a card is inserted in a reader. External applications, and even the Windows operating system, can cause a smart card to reset itself. For example, when a card is used with the Base Smart Card Cryptographic Service Provider, the Base CSP may reset the card after use in order to prevent another application from using the keys stored on the card without presenting a PIN.

What Does a Reset Mean?

A reset on an IDPrime .NET Card causes several things to happen:

- Garbage collection is invoked (described in “Garbage Collection” on page 34).
- All open FileStream objects are closed. Further attempts to use these objects result in an ObjectDisposedException being thrown.
- All open MemoryStream objects are closed. MemoryStreams also throw an ObjectDisposedException if you attempt to use them after a reset.
- A CardReset event is triggered. An application can listen for this event to perform any operations that might be necessary to reset the card. For example, you might wish to reset data or fields that might be specific to your session.

Handling the Reset Event

The SmartCard library supplied with the SDK contains a SmartCard.SystemEvents class that contains a SessionEnded event. If you want to be notified when the card is reset, you simply add a delegate to the SessionEnded event.

For example:

class MyClass

```
{
   int iSessionCounter;
   public MyClass()
   {
      iSessionCounter = 0;
      SystemEvents.SessionEnded += new
SessionEndedEventHandler(OnCardReset);
   }

   public void RemotingMethodThatGetsCalled()
   {
      iSessionCounter++;
   }

   private void OnCardReset(object sender, SessionEndedEventArgs e)
   {
      iSessionCounter = 0;
   }
}
```

In the preceding example, the iSessionCounter is reset back to zero every time the card is reset.

Card Services

This sections describes two services that help you communicate with the card.

ContentManager

The Content Manager service is installed on the card during personalization at the factory. This service allows you to manage the lifecycle of the card by managing the file system, setting card properties, and loading/unloading assemblies. The Content Manager can be used either from on-card applications or off-card applications. To access the Content Manager from off-card applications, you can obtain a proxy to the Content Manager using standard remoting techniques, or you can use the off-card SmartCard.CardAccessor library, which provides an interface to the on-card ContentManager application and does not require the calling application to use remoting directly.

Features

Broadly, the ContentManager application allows you to do the following:

- Manage files on the card. This includes creating/deleting both directories and files, getting/setting properties associated with files, and managing the security settings associated with a given file or directory.
- Manage assemblies on the card. The API provides support for loading applications, executing assemblies, and unregistering services.

- Manage card properties. For example, you can set the chip speed, the communication speed, or the historical parts of the ATR. In addition, you can read information about the version of the card, free memory available, etc.

Examples

Here's an example of using the ContentManager from an off-card application:

```
ContentManager cm = (ContentManager)Activator.GetObject(typeof
(ContentManager),"apdu://selfDiscover/ContentManager");
int speed = cm.ChipSpeed;
```

Here's an example of using the CardManager from an on-card application:

```
ContentManager cm = (ContentManager)Activator.GetObject(typeof
(ContentManager),"ContentManager");
int speed = cm.ChipSpeed;
```

Note the close similarities. The on-card application is using the same remoting mechanisms that the off-card application is using.

More Information

- For more information about the ContentManager API, see the online help (OLH) documentation in **Start** > **Programs** > .**NET Smartcard Framework SDK** > **X.X.XXX** > **Help** > .**NETSmartcardFramework**.

Note: Where X.X.XXX is the SDK version number.

In the OLH, see **Smart Card > ContentManager Class**.

SampleAccessManager

The SampleAccessManager service extends the SmartCard.AccessManager class. The Access Manager is responsible for controlling access to system resources such as the file system and card properties.

Features

You can use the SampleAccessManager application to log on to the card as a user. This service is responsible for granting or denying privileges to system resources. You can replace the SampleAccessManager application with a different service that extends SmartCard.AccessManager. For more details about the AccessManager model, see “Access Manager” on page 40.

SampleAccessManager Roles

SampleAccessManager defines four categories of roles that can be associated with the card:

- **Administrator** has full permissions to modify the card and to control other user roles.
- **Power User** is defined as a mid-level user with more permissions than a standard user.
- **User** is a standard user.
- **Guest** is a guest role with limited privileges.

From within the SampleAccessManager client tools, you can create new users and assign them to one of these roles.

SampleAccessManager Rules

SampleAccessManager manages access to the file system based on privileges assigned to a given user or role. Individual user accounts are assigned separate **code** and **data** directories in which to install their applications. All users have access to the C:\Pub and D:\Pub directories. A guest user may access only the **C:\Pub** and **D:\Pub** partitions of the card. An administrator is granted full access to the card.

Users can create and delete roles that have lower privileges. For example, the **Administrator** can create **Power Users** and **Users**, but a **Power User** would only be able to create a **User** account. Also note that a user cannot create a user that has broader file access privileges than the creating user account. For example, if a **Power User** had access to the C:\Users\students directory, they could not create a simple **User** who had access to the C:\ or C:\User directories.

In the current implementation, there is only one **Administrator** account and one **Guest** account.

SampleAccessManager Client Information

A SampleAccessManager client is also installed as part of the SDK. The client is responsible for producing all of the dialog boxes associated with user logon and user management.

4

Card Explorer

The Card Explorer is the tool available to manage IDPrime .NET Cards. It is part of the IDPrime .NET SDK. The SDK is available free of charge from the Gemalto.com .NET pages @ http://www.gemalto.com/products/dotnet_card/resources/development.html?toggler=0

There are two versions of the SDK available: v2.3 for Visual Studio 2010, and v2.2.181 for all the other VS versions.

Introduction

The Card Explorer is a tool (also available as an add-in to Visual Studio .NET) that simplifies IDPrime .NET Card management, including viewing a card's content; managing the card's assemblies, data files, and directories; managing authentication according to the requirements of the current Access Manager; and creating and deleting services on the card.

Starting Card Explorer

If the **CardExplorer** component was selected during installation, you can start the Card Explorer tool from the Windows Start menu. Choose **Programs** > **.NET Smart Card Framework SDK** > ***X.X.X*** > **CardExplorer**.

Note: X.X.X is the version of SDK.

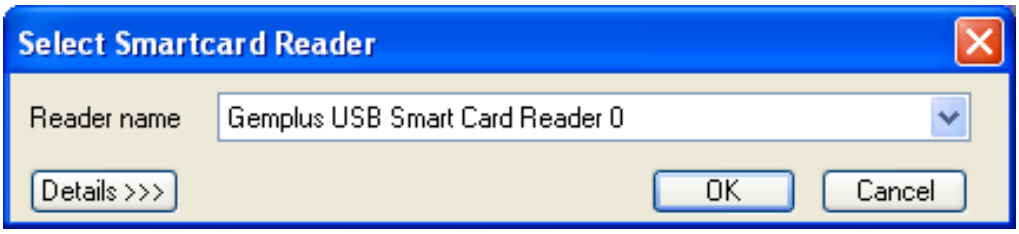
If the **CardExplorer AddIn in VS.NET** component was selected during installation, when Visual Studio.NET is launched, the Card Explorer is automatically started unless you have changed the default Startup setting (see “Managing the .NET Card Add-in” on page 72).

Connecting to the IDPrime .NET Card

These are the steps to connect to the IDPrime .NET Card and authenticate to the card:

1. Insert the IDPrime .NET Card into the smart card reader.
2. In the Card Explorer toolbar, click the **Connect** icon  to open the **Select Smart Card Reader** dialog box.

Figure 15 - Select Smart Card Reader dialog box

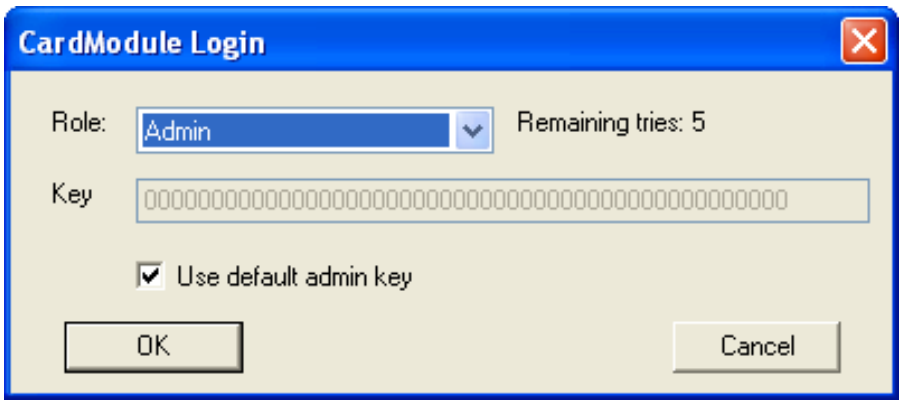


3 In **Reader name**, select the name of the reader in which the IDPrime .NET Card is inserted. If necessary, click **Details** and select other options (see “Select Smartcard Reader Details” on page 55). Click **OK**. In the Card Explorer toolbar, the **Log on** icon becomes available.

4 Click the **Log on** icon  to invoke the authentication method of the current Access Manager on the card. The authentication method might require the user to provide some sort of input (for example, a user name and password, or a biometric like a fingerprint) using a client application. If this is the case, an appropriate dialog box or other graphical user interface is displayed after you click the **Log on** icon.

For example, if the authentication method for the current Access Manager requires the user to specify a user name and password to access the card, clicking the Log on icon might result in display of a dialog box similar to the following one to gather and submit the required information.

Figure 16 - Log on to .NET Smart Card dialog box



If authentication is successful according to the requirements of the current Access Manager, the card contents are displayed in the Explorer tab of the Card Explorer.

Cards with the Access Manager Admin Key

If the card has been configured with the Access Manager Admin Key, using the -s installation parameter, the available roles in the **Role** list appear as follows:

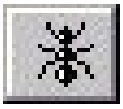
- Card Admin = Access Manager Admin Key role
- Card Module Admin = Admin Key role
- Card Module User = User PIN role
- Everyone

Toolbar

The Card Explorer toolbar provides convenient access to perform routine card management tasks.

Figure 17 - Card Explorer Toolbar

Table 2 - Card Explorer Toolbar Icons Descriptions

Icon	Tooltip	Description
	Connect	Links to the **Smart Card Reader** dialog box, in which you identify the reader into which the IDPrime .NET Card is inserted.
	(Authenticate)	Invokes the authentication method of the current Access Manager on the card. (The tooltip associated with the icon is defined by the current Access Manager; for example, the tooltip might be “**Log on**”.)
	Refresh	Refreshes the information displayed in the active Card Explorer tab.
	(Manage Access)	Enables access management (for example, user access) according to the rules specified by the current Access Manager on the card. (The tooltip associated with the icon is defined by the current Access Manager; for example, the tooltip might be “Add/Remove Users”.)
	Run NAnt	When the Card Explorer is in VisualStudio.NET add-in mode, executes the build file for the current project. When the Card Explorer is run as a standalone application, displays a Select Build File dialog box, and then executes the selected build file. For more information, please refer to “Building with NAnt” on page 88
	Help	Links to the Smartcard.NET help documentation.

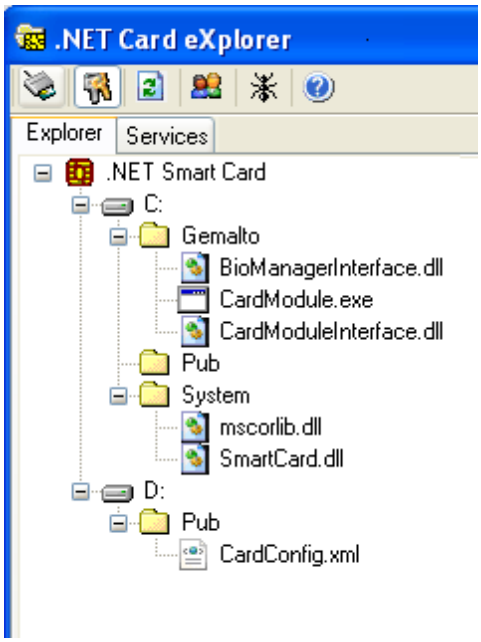
The toolbar information is always displayed, regardless of which tab is active.

Tab Layout

The management features are grouped into two tabs:

- The **Explorer** tab offers a view of the IDPrime .NET Card's contents, and the ability to manage assemblies, data files, and directories on the card.

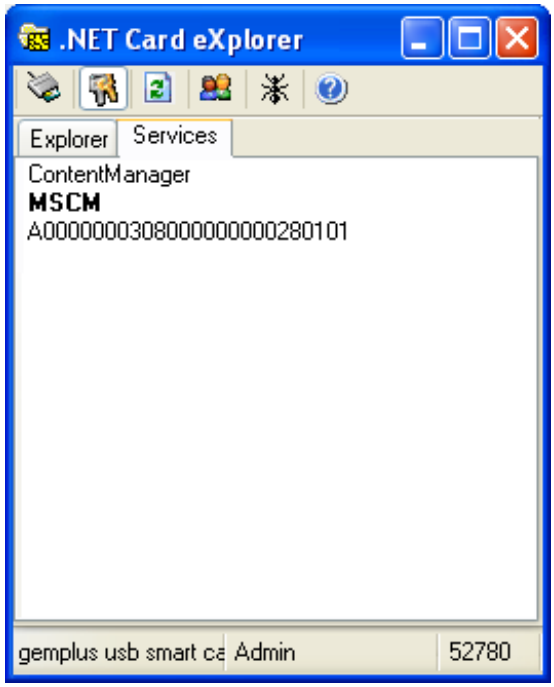
Figure 18 - Card Explorer – Explorer Tab



See “Explorer Tab” on page 56 for more information about the elements on the Explorer tab.

- The **Services** tab provides a view of services running on the IDPrime .NET Card, and provides features to manage services.

Figure 19 - Card Explorer – Services Tab



See “Services Tab” on page 58 for more information about the elements on the Services tab.

This is the bottom bar of the Card Explorer window:

The following information is displayed in the three fields in the bottom bar:

- Current reader (Gemalto Reflex USB V3 0)
- Information pertaining to the current Access Manager, for example, the name of the current user.
- Free space on the card in bytes (92952)

The bottom bar information is always displayed, regardless of which tab is active.

Select Smartcard Reader Details

This is the dialog box that displays when you click Details in the Select Smartcard Reader dialog box (“Figure 15” on page 52).

Figure 20 - Select Smartcard Reader Dialog Box

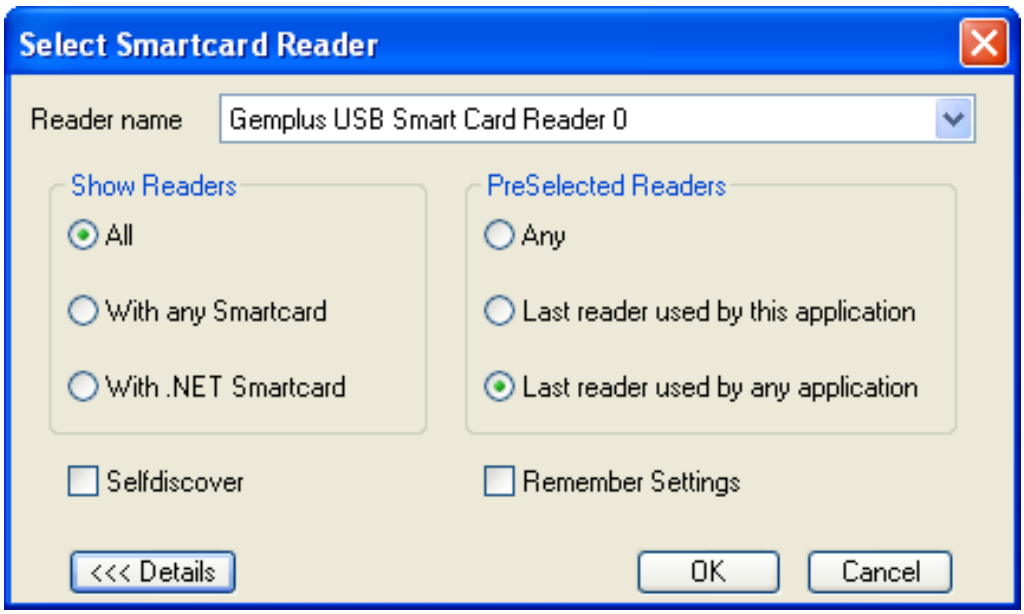


Table 3 - Select Smartcard Reader Options

Element	Description
Reader name	Select the name of the reader in which the IDPrime .NET Card is inserted.
Show Readers	Choose whether the **Reader name** list should be populated with the names of all readers connected to the machine, all readers with smart cards currently inserted, or all readers with IDPrime .NET Cards currently inserted.

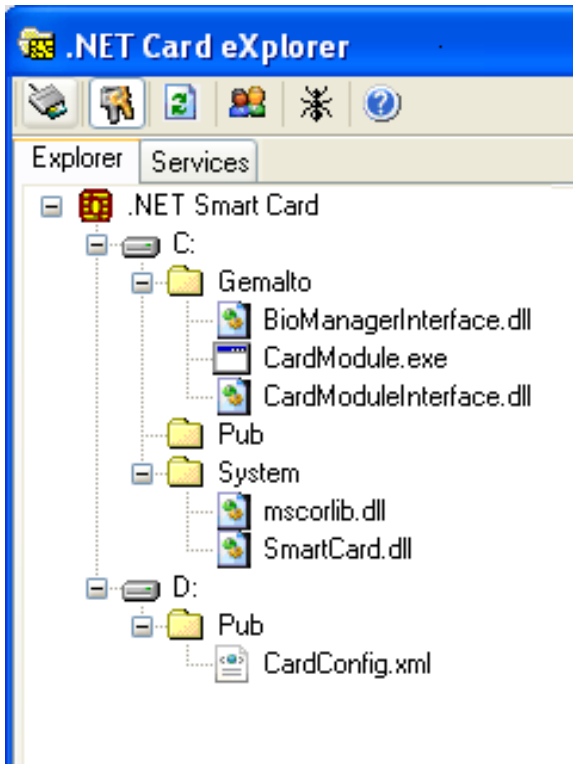
Table 3 - Select Smartcard Reader Options

PreSelected Readers	In the event more than one reader is attached to the machine, choose whether the first reader displayed in the **Reader name** list should be: ■ The last reader used by this application ■ The last reader used by any application ■ No reader (that is, no reader should be pre-selected)
Selfdiscover	Select this check box to cause the application to attempt to find a card containing the service that the application is requesting. If the service is available on an attached card, the application connects to the card without user intervention. (Otherwise, if the service is not located on the card, the Select Smartcard Reader dialog box displays.)
Remember Settings	Select this check box to save the selections you just made.

Explorer Tab

This is the Card Explorer, **Explorer** tab.

Figure 21 - Card Explorer - Explorer Tab



In the **Explorer** tab, when an element in the display is selected, a context-sensitive menu specific to that element becomes available. If you are using a mouse, this menu is available when you right-click a displayed element.

These are the context-sensitive menu options available for elements on the **Explorer** tab.

Note: All menu options are visible to all users, but a menu option can be invoked only if the user's permissions allow the action on the selected element.

The following table describes the menu options available for each type of card element:

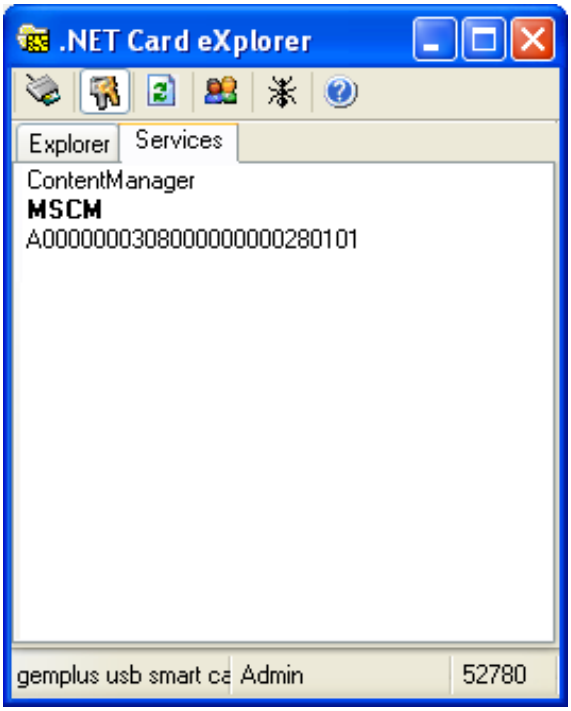
Table 4 - Card Element Descriptions and Menu Options

Card Element Type	Menu Option	Description
Card	Restart	Equivalent of removing and then re-inserting the card
	Memory Mapping	Blue designates bytes currently allocated; white designates free memory in EEPROM on the card.
	Properties	See “Card Element Properties” on page 60 for details.
Volume/Drive	New Folder	Add a new folder to the selected directory.
Directory/Folder	Delete	Remove the selected folder from the card.
	New Folder	Add a new folder to the selected folder.
	Load File	Load a new file to the card. See “Managing Folders and Files” on page 68 for details.
	Properties	See “Card Element Properties” on page 60 for details.
Executable (.exe)	Execute	Run the selected executable file.
	Delete	Remove the selected file from the card.
	Properties	See “Card Element Properties” on page 60 for details.
Library (.dll) or other non-executable file (e.g., xml file)	Delete	Remove the selected file from the card.
	View Content	Opens the file in the application configured for the file type on the machine.
	Properties	See “Card Element Properties” on page 60 for details.

Services Tab

This is the Card Explorer, **Services** tab.

Figure 22 - Card Explorer – Services Tab



The **Services** tab displays a list of services currently running on the card. By default, each card includes the following three services:

Table 5 - Card Services

Service	Description
ContentManager (C:\System\SmartCard.dll)	Card Manager service. This is the default "Access Manager" for the card.
CardModule (MSCM) (C:\Gemalto\CardModule.exe)	This is the instance of the Card Module.
OATHService (optional feature of the IDPrime .NET card) (C:\Gemalto\OATHService.exe)	The figure that displays is the AID of the OATH service that appears in the Services tab. The AID is one of the installation parameters when you install the OATH service.

The card can also contain additional services, for example, an alternative "Access Manager". Each time you instantiate a service on the card, the display changes to list the new service on the **Services** tab.

If you right-click the name of a service listed in the **Services** tab, some services display one or both of the following options:

Table 6 - Menu Options

Menu Option	Description
Set as Access Manager	See “Access Manager” for more information about this option.
Delete	Deletes a service. If you select a service and your permissions allow you access to the folder in which the application is located, the service is deleted. Note that this does not delete the server application from the card; if you want to re-instantiate the service, go to the **Explorer** tab, and either double-click the .exe or right-click the .exe, and select **Execute**.

Access Manager

An Access Manager service defines the authentication mechanism by which the information and applications on the card can be accessed. The default Access Manager, the CLOG service, requires the user to authenticate by providing a username and password. A different Access Manager might require the user to open a secure channel by verifying the AUTH, MAC, and KEK keys for the card. An alternative Access Manager might authenticate using a biometric solution, like verifying the user's fingerprint.

Because different authentication mechanisms might be appropriate at different stages in the card's lifecycle, the card's flexible architecture provides an API for authentication management applications. Any Access Manager application that implements the IAccessManager interface can provide the authentication mechanism for the card. The new Access Manager can be loaded onto the card, instantiated, and designated as the active Access Manager service. At any given time, exactly one Access Manager service can be active on the card, and the active Access Manager service is marked by an asterisk in the Services tab display.

These are the steps to change which service should be used for authentication on the card:

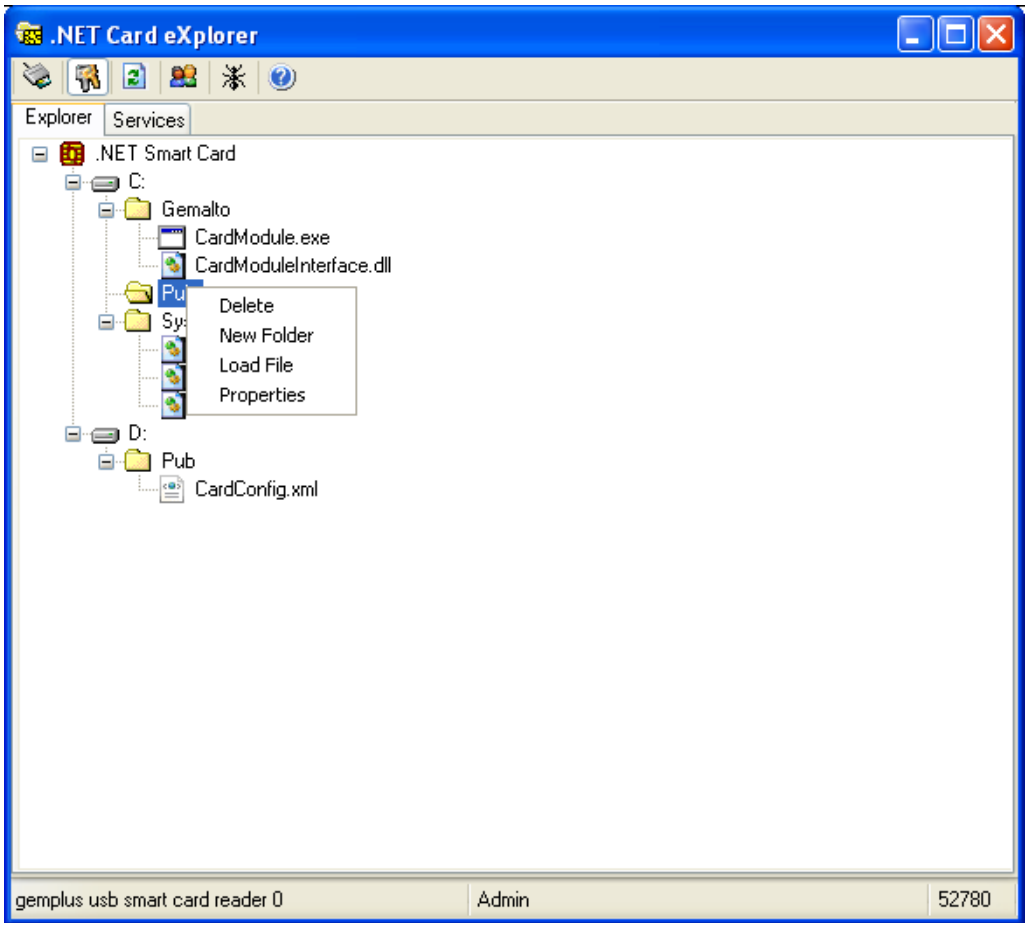
1. Create and load the new Access Manager service (see “Access Manager” on page 40 for instructions about creating and loading an alternate Access Manager server onto the card). Loading also includes instantiation, so the **Services** tab list will include the new service.
2. In the **Services** tab, right-click the newly-added service and select **Set As Access Manager**. In the display, the asterisk now designates the new Access Manager service.

Note: If you try to designate a service that does not implement the IAccessManager interface, the operation fails and the old Access Manager remains active.

Card Element Properties

When you right-click any element in the **Explorer** tab display, a menu choice to display **Properties** becomes available:

Figure 23 - Card Explorer - Card Element Properties



If you click **Properties**, a Properties sheet specific to the card element type is displayed. The ability to view information is based on permissions for the logged on user. Each label is always displayed, but if permissions for the current user do not allow access to the information, the details are not visible.

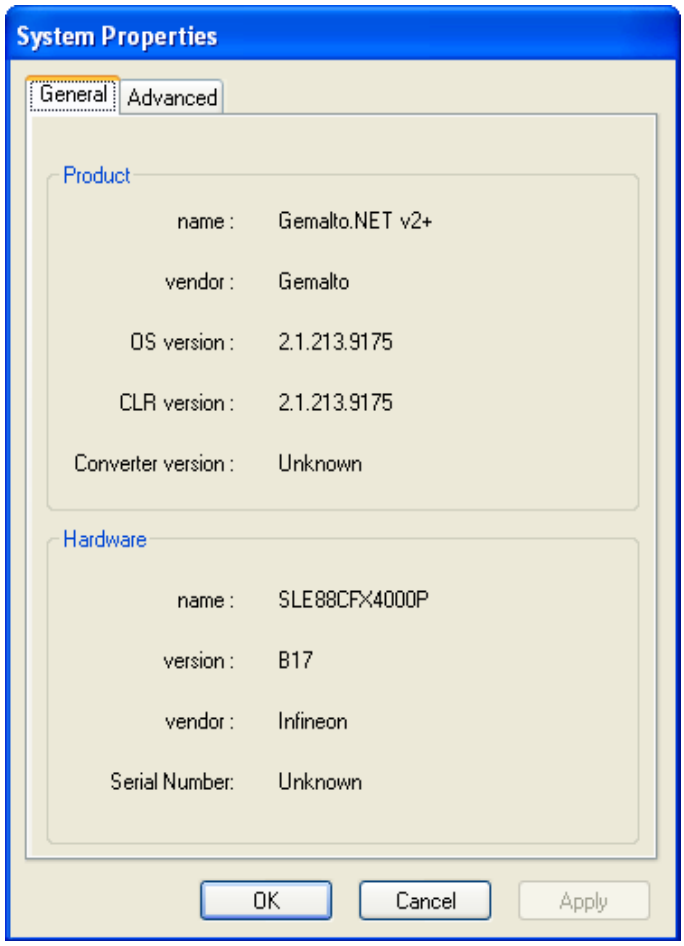
This section describes property details that are displayed for each card element type.

Card Properties

The properties sheet for the card object includes information in two tabs.

This is the card object **General** tab:

Figure 24 - Card Properties - General Tab



These are the elements on the **General** tab for the card object.

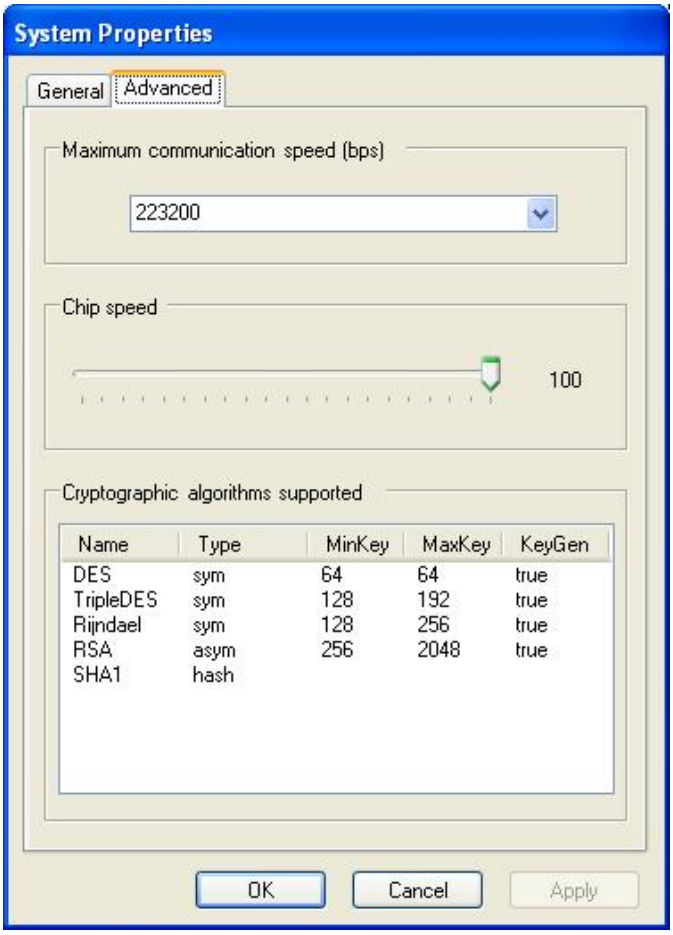
Table 7 - Card Properties - General Tab Elements

Property	Description
Product	Name and vendor of the .NET Smart Card Framework, taken from the cardconfig.xml file on the card. It also shows version information for the operating system (OS), Common Language Runtime (CLR) and converter.
Hardware	Name, vendor, version, and serial number of the chip, taken from the cardconfig.xml file on the card.

Configuring the Communication and Chip Speed

You can modify the Maximum Communication Speed and the Chip Speed in the card object **Advanced** tab:

Figure 25 - Card Properties - Advanced Tab



These are the elements on the **Advanced** tab for the card object.

Table 8 - Card Properties - Advanced Tab Elements

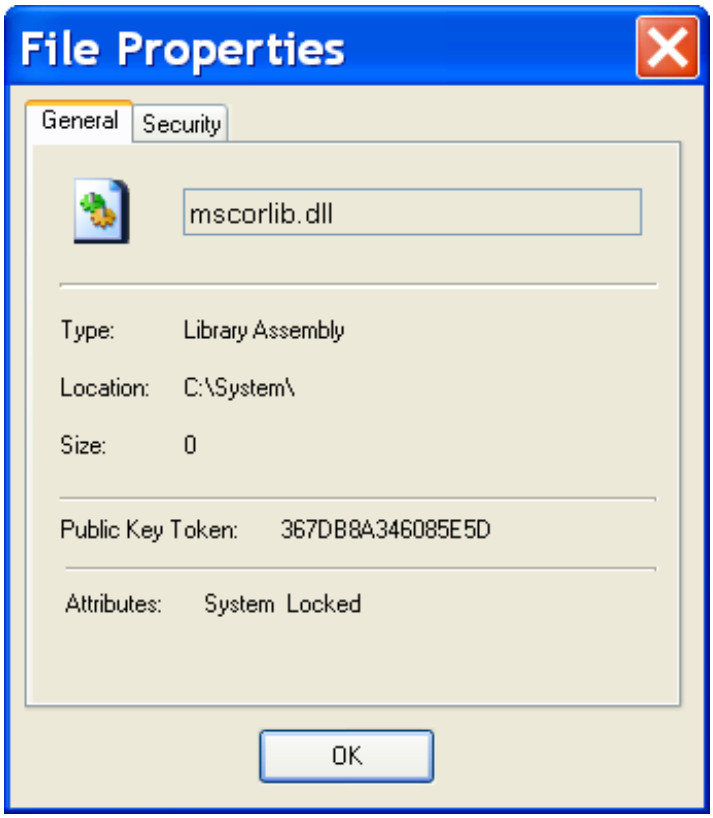
Property	Possible values	Default value	Description
Maximum connection speed (baud)	9600 19200 38400 55800 76800 111600 115200 223200	223200	Admin only may set this value. The list of allowable values is taken from the cardconfig.xml file. Typically, this value is set to the maximum speed at which negotiation between the card and the reader driver is successful. If the card and the reader are having trouble negotiating at this speed, selecting a different maximum connection speed is sometimes useful.
Chip speed	0 - 100	100	Set as a percentage. Setting a slower chip speed results in slower card transactions. 0: External Clock 1-50: Low Internal Clock (low consumption mode) Over 50:High Internal Clock (high consumption mode)
Cryptographic algorithms supported			Taken from the cardconfig.xml file.

Folder/Directory and File Properties

Folders and files have the same set of properties, organized on two tabs.

This is the **General** tab for a folder/directory (properties are similar for files on the card):

Figure 26 - Folder Properties - General Tab



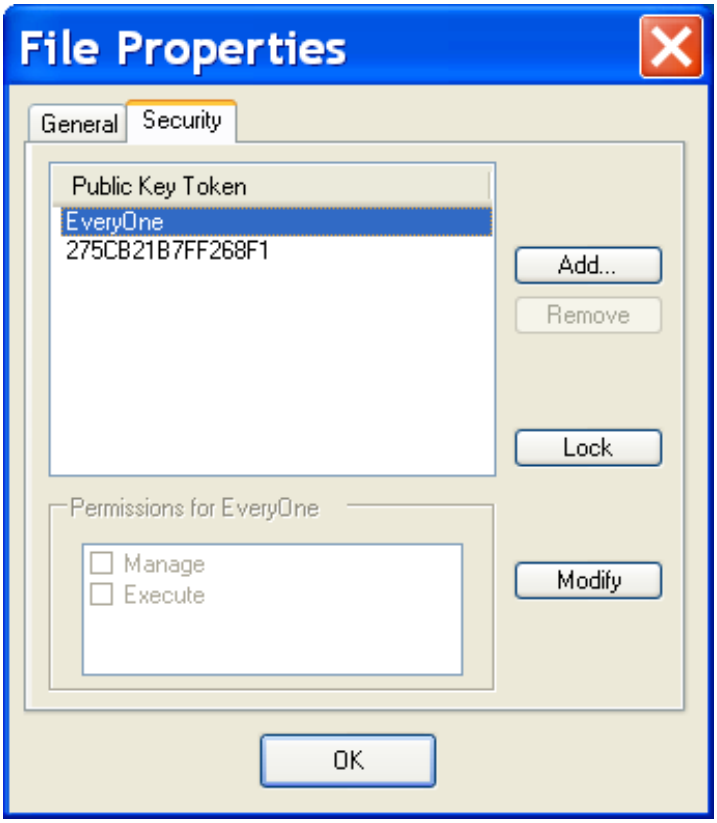
These are the elements on the **General** tab for the folder object.

Table 9 - Folder Properties - General Tab Elements

Property	Description
Type	For example, Directory, Executable Assembly, Library Assembly, XML Document, Text Document.
Location	File path to the object on the card.
Size	Size of the object in bytes.
Public Key Token	Each assembly (.exe or .dll) has an associated public key token. See “Public Key Tokens” on page 67 for more information. For other objects (folders and non-assembly files), this field is blank.
Attributes	These can take the following values: **Normal** - The file permissions are open to change provided the given user has the corresponding rights – see “Permissions” in the **Security** tab. For example you can make a text file readable or writable or remove these permissions if you so desire. **Locked** - Whatever file access permissions were set at the time of the lock remain, and cannot be changed. It is not possible to return to the “Normal” state. **System Locked** - same as “Locked”, expect that the file is also a system file that was part of the .NET framework OS, and not an assembly that was loaded later.

This is the **Security** tab for a file (properties are similar for folders/directories):

Figure 27 - File Properties - Security Tab



These are the elements on the **Security** tab for the folder object.

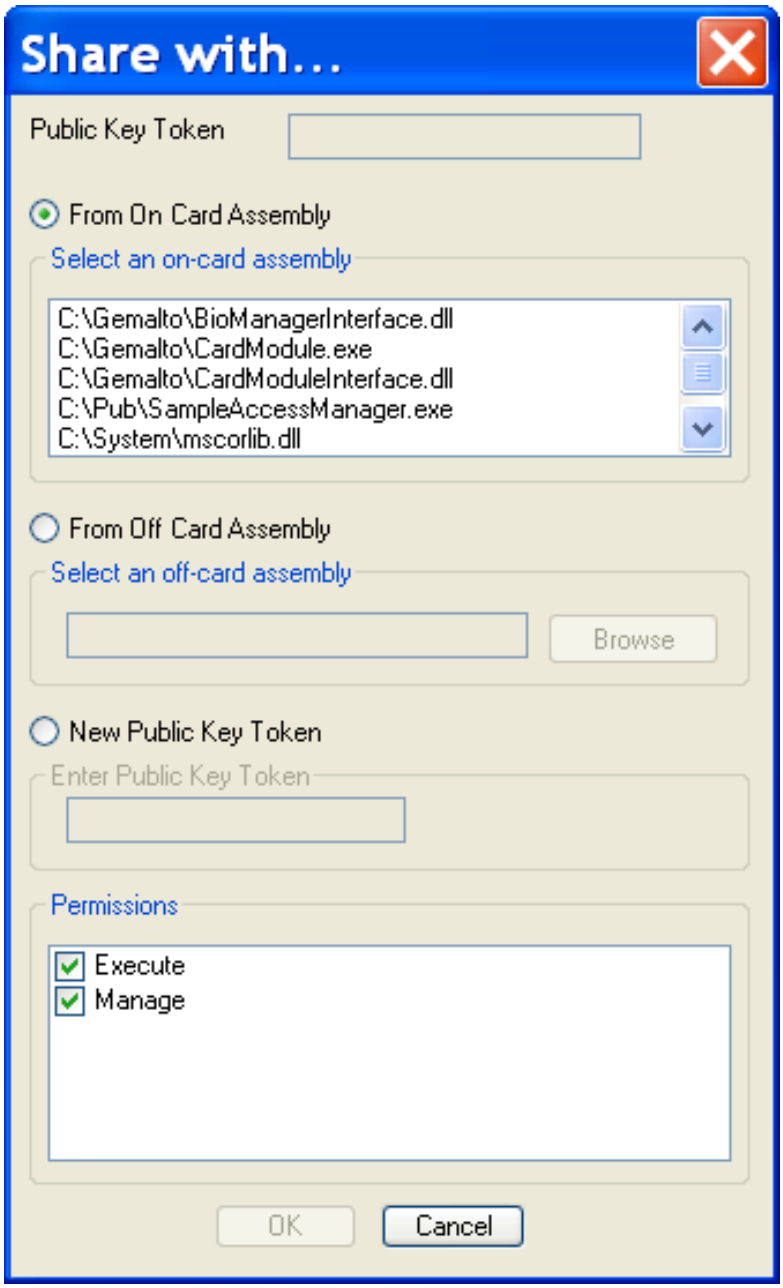
Table 10 - File Properties - Security Tab Elements

Property	Description
Public Key Token	For folders: the public key tokens for applications that are currently permitted to write files to the selected folder. For executables: the public key tokens for applications that are permitted to execute the selected application. For libraries: the public key tokens for applications that are permitted to access the selected library. For other files: the public key tokens for applications that are permitted to write to the selected file.
Permissions	A list of the permissions associated with the public key token. The type of permissions listed depends on the type of file being examined.
Add button	Add a new public key token to the list by selecting an assembly (an .exe or .dll file) from the list (the public key token for the selected assembly is added to the list).
Remove button	Remove the selected public key token from the list.
Lock button	For files, this button makes the current file permissions permanent.
Modify button	To change the permissions associated with the public key token.

To add a public key token to the list:

1 Click **Add**. This displays the **Share with**... dialog:

Figure 28 - Share With... Dialog Box



2 Choose one of the three basic sources of public key tokens as follows:

- Click **From On Card Assembly** and select an assembly that is already on the card from the list.
- Click **From Off Card Assembly** to choose an assembly that is off the card and use the **Browse** button to select the assembly.
- Click **New Public Key Token** and enter a new 8-byte hexadecimal key token.

3 In **Permissions**, you can select the permissions associated with the assembly.

The following tables show the types of permissions that can be set for different types of files.

Table 11 - Permission Types for Assemblies

Permission Type	Description
Execute	Whether or not the assembly can be executed.
Manage	Whether or not the public key token has permission to change the security parameters of the assembly.

Table 12 - Permission Types for Folders

Permission Type	Description
Add	Whether or not the public key token can add files to the folder.
Delete	Whether or not the public key token can delete files from the folder.
Enumerate	Whether or not the public key token can list the files in the folder.
Manage	Whether or not the public key token can change the security parameters of the folder.

Table 13 - Permission Types for Non-Assembly Files

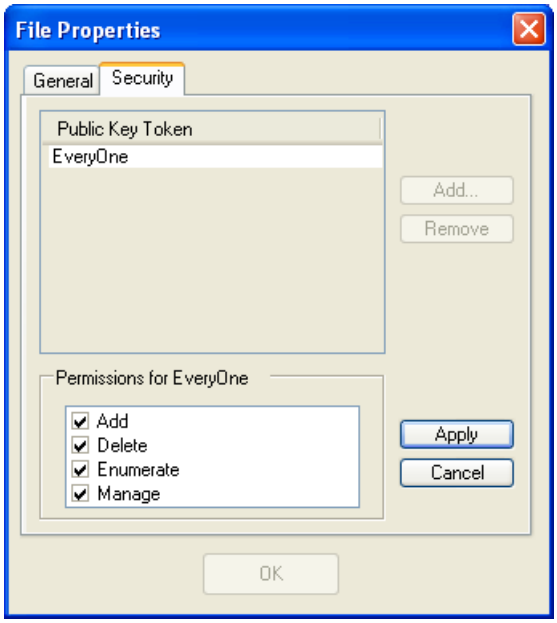
Permission Type	Description
Read	Whether or not the assembly can be executed.
Write	Whether or not the public key token has permission to change the security parameters of the assembly.
Manage	Whether or not the public key token can change the security parameters of the folder.

To modify the permissions for a public key token in the list:

1 In the **Security** tab of the **File Properties** dialog (shown in “Figure 27” on page 64), select the token whose permissions you want to modify and click **Modify**.

The dialog changes as shown in “Figure 29”:

Figure 29 - Modifying Permissions for a Public Key Token



2 Check or uncheck the boxes next to each permission to make your modifications, then click **Apply**. The **Modify** button reappears

3 Click OK to close the **File Properties** dialog.

Public Key Tokens

The use of public key tokens provides a code-access security mechanism (see “Application Security” on page 41).

The public key token is derived from a hash of the file's public key. Public key token information is used in two ways:

- To identify an assembly
- To control access to a file or folder on the card

Identifying an Assembly

Every assembly that is loaded to the card must be strong signed and, therefore, it must include a public key that identifies it to other applications. The public key token information (derived from a hash of the public key) for an assembly is found on the **General** tab, **Public Key Token** property.

Controlling Access to a File or Folder on the Card

If an assembly or folder is “Not Public” (that is, on the **General** tab for the object, the File Access or Assembly Access property is set to “Not Public”), the object is accessible only to executables with public key tokens that match the Public Key Token list on the **Security** tab for the object. This is true whether the access needed is to run an executable, access a library, or write to a file. In addition, this mechanism controls whether assemblies or files may be loaded to a selected folder on the card. (User access is set at the folder level, so constraining placement of uploaded assemblies or files to a particular location enables this access enforcement mechanism to work as intended.)

The Security tab's Public Key Token list can be modified as needed, using the **Add**, **Modify**, and **Remove** buttons as shown in described in “Figure 27” on page 64.

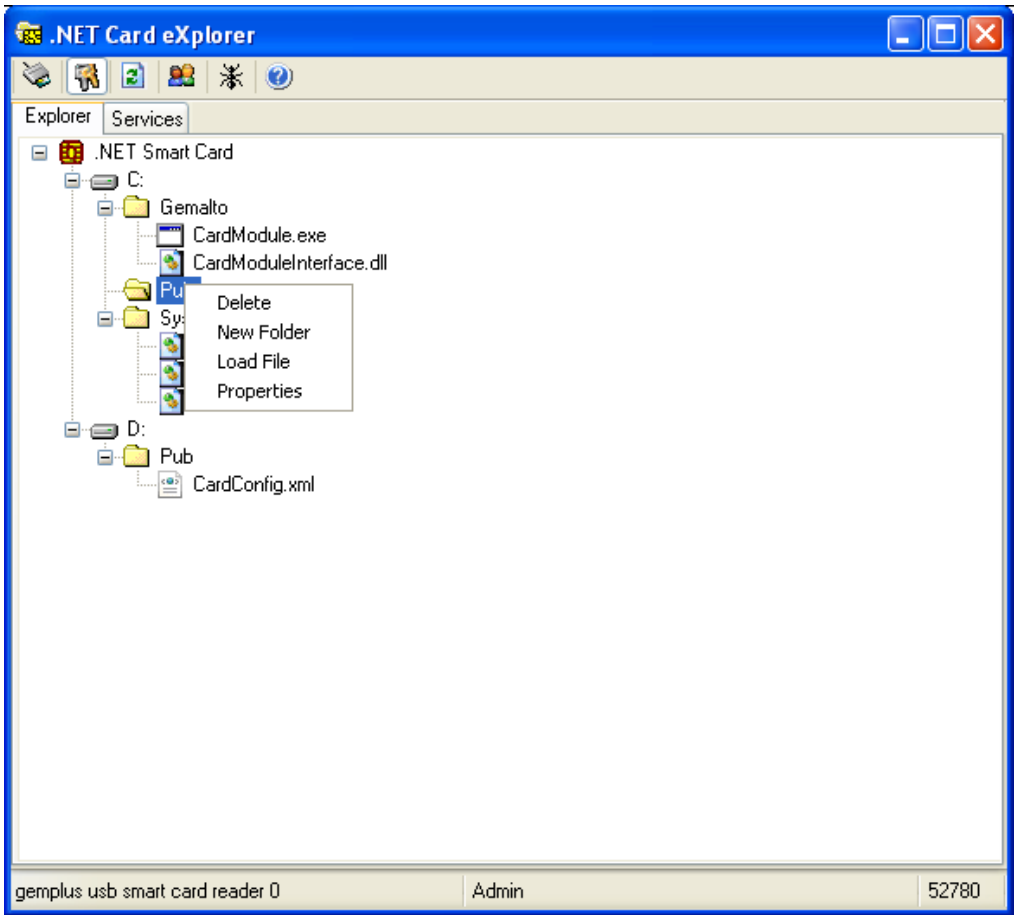
Managing Folders and Files

The Card Explorer add-in tool in Visual Studio .NET enables you to manage the contents of an IDPrime .NET Card that is connected to your computer.

Managing Folders

When you right-click a folder on the card, a menu becomes available offering several folder management options.

Figure 30 - Contextual Menu for Folders



Delete

When you right-click any folder on the card, the menu choice **Delete** is available. If you click **Delete**, a Confirm Folder Delete message asks you to confirm that you want to delete the folder. If you click Yes and your permissions allow you to access the selected folder, the folder is deleted from the card. Note that you cannot delete a folder that is not empty.

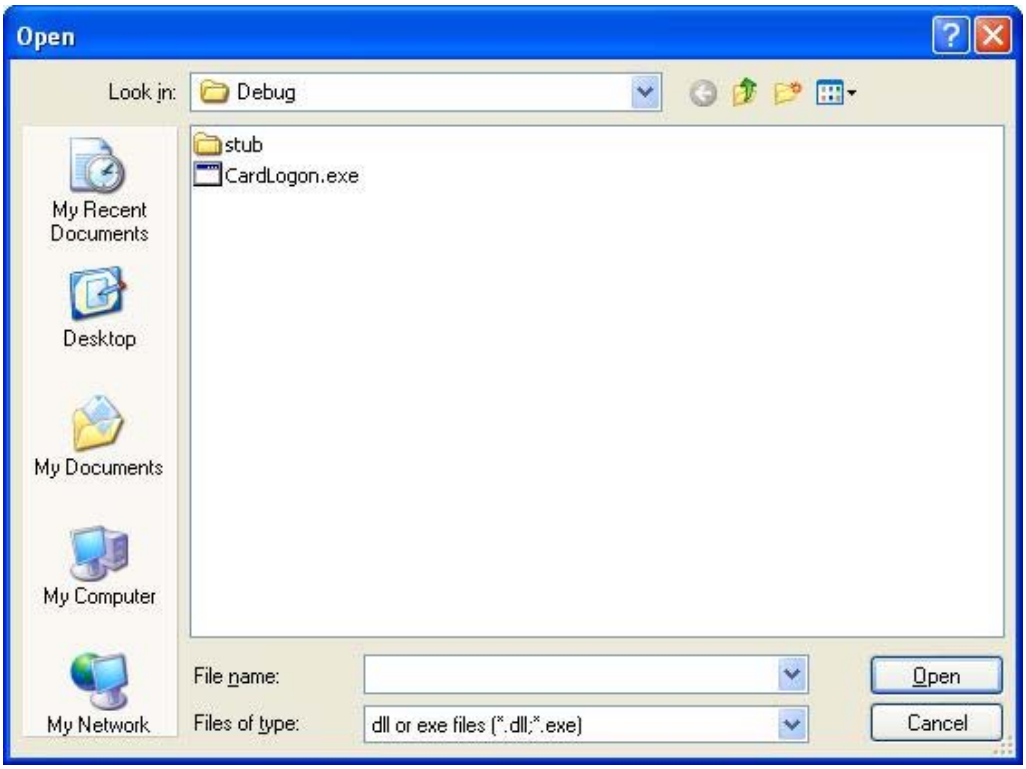
New Folder

When you right-click any folder on the card, the menu choice **New Folder** is available. If you click **New Folder** and your permissions allow you to access the selected folder, the new folder is created. You must rename the folder when you create it (while the name is still highlighted); after the folder object has been created, its name cannot be changed. A folder cannot be moved after it is created.

Load File

When you right-click any folder on the card, the menu choice **Load File** is available. If you click **Load File**, the **Open** dialog box is displayed.

Figure 31 - The Open Dialog Box



Both assemblies (.exe and .dll files) and data files can be loaded on the card. Navigate to the file you want to add to the card, and click **Open**.

If the Access Manager allows the operation and if there is room on the card, the operation succeeds.

Properties

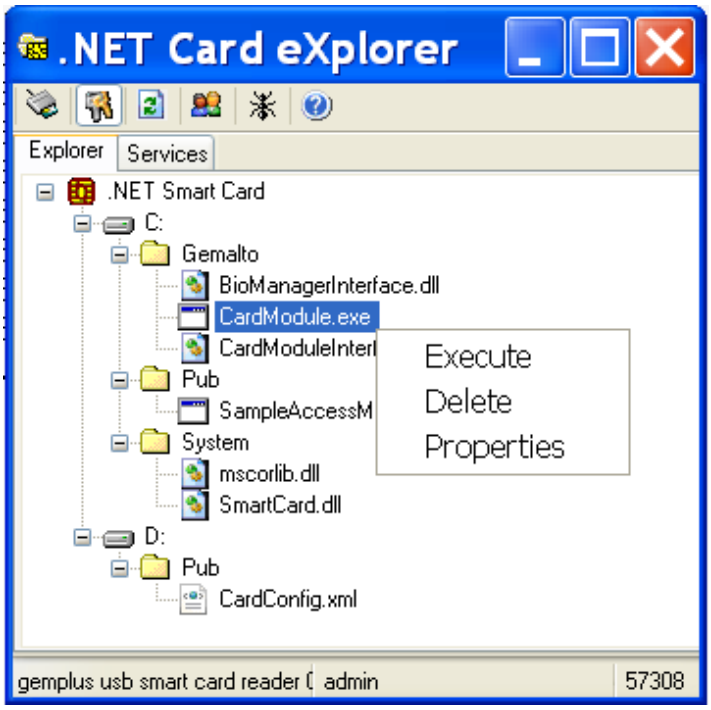
When you right-click any folder on the card, a menu option to view **Properties** is available. See “Card Element Properties” on page 60 for information about the properties associated with folders.

Managing Files

When you right-click any file on the card, a context-sensitive menu is displayed. Some functions are specific to the file type (for example, only a .exe file can be executed, while the content of uncompiled files only can be viewed).

Here is an example, which is the right-click menu for an executable file.

Figure 32 - Contextual Menu for Executable Files



Execute

When you right-click any executable file on the card, the menu option **Execute** is available. If you click **Execute** and your permissions allow you to access the selected file, the file is executed.

Delete

When you right-click any file on the card, a menu choice **Delete** is available. If you click **Delete**, a Confirm File Delete message asks you to confirm that you want to delete the object. If you click **Yes** and your permissions allow you to access the selected file, the file is deleted from the card. Note that you cannot delete an assembly that is hosting an active service. If the assembly hosts a service, you must first delete the service. See “Services Tab” on page 58 for information about deleting a service.

View Content

When you right-click any non-binary file on the card, a menu option **View Content** is available. File types that are associated with specific applications on your system will open in those applications.

Save to PC as

When you right-click any non-binary file on the card, a menu option **Save to PC as** is available. This enables you to save the file to your PC. It opens the **Save file off card** dialog which you use to select the location where you want to save the file.

Properties

When you right-click any file on the card, the menu option **Properties** is available. See “Card Element Properties” on page 60 for information about the properties associated with files.

Restrictions

You are able to perform actions according to the permissions associated with your logon identity. For example, if you are logged on as a user guest, with permission for adding files only in the D:\Pub directory, you will be unable to add files in the C:\Gemalto directory, even if you follow the instructions correctly.

5

Visual Studio .NET Integration

The v2.2.181 of the .NET Smart Card Framework SDK supports the following versions of Visual Studio:

- Visual Studio .NET 2003
- Visual Studio 2005
- Visual Studio 2008

v2.3 of the .NET Smart Card Framework SDK supports the following versions of Visual Studio:

- Visual Studio .NET 2010

Depending on install-time selections, installing the .NET Smart Card Framework SDK might make one or both of the following features available within Visual Studio .NET:

- "Card Explorer", which enables you to manage the contents of an IDPrime .NET Card that is connected to your computer
- Templates to easily create server applications and client applications that can access services running on an IDPrime .NET card.

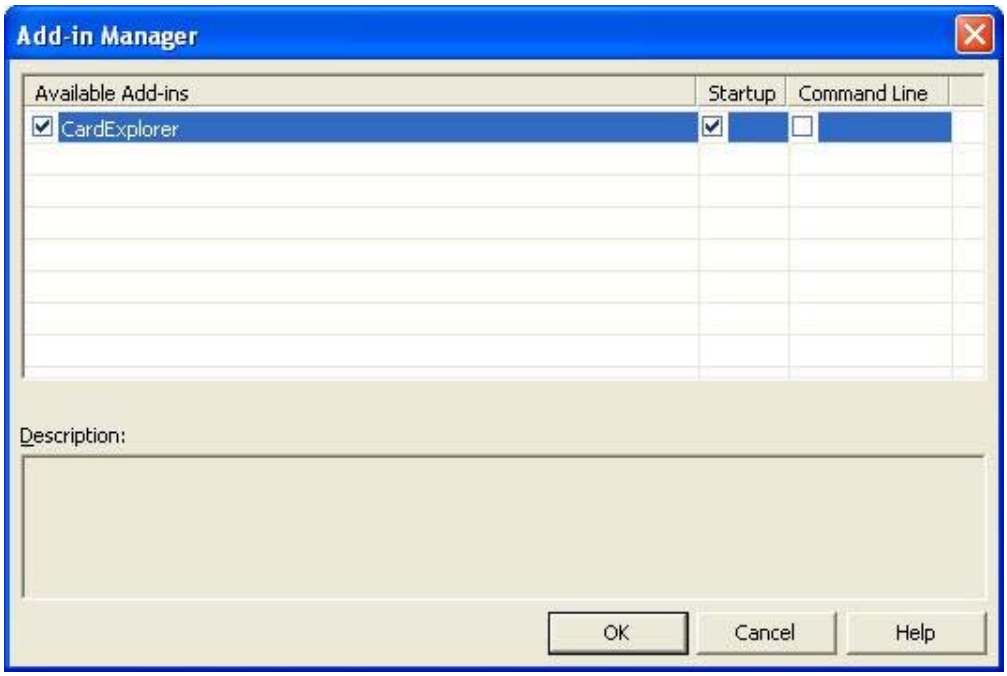
Managing the .NET Card Add-in

The Card Explorer Visual Studio add-in is a dockable component that is designed to dock with your solution explorer. Visual Studio .NET provides a tool to manage add-in modules. You can manage the Card Explorer add-in using this feature.

How to Manage the Card Explorer Add-in

To launch the Add-in Manager tool, in the Visual Studio .NET toolbar, select **Tools** > **Add-in Manager**. The Add-in Manager is displayed.

Figure 33 - Add-In Manager



The CardExplorer add-in module is listed in the Available Add-ins column. Note that if additional add-in modules to Visual Studio .NET are installed and registered on your computer, your display will list those add-ins, in addition to the CardExplorer module.

Use the Add-in Manager to perform the following tasks:

- To unload the CardExplorer add-in, deselect the check box at the left of the add-in name, and then click **OK**. Note that unloading the add-in does not uninstall the software.
- To load the CardExplorer add-in, select the check box at the left of the add-in name, and then click **OK**.
- To specify that the CardExplorer add-in should be loaded at environment startup time, select the check box in the Startup column, and then click **OK**.

It is recommended that you uncheck the command line option for the tool.

Note: The Command Line check box has no meaning. You can clear the check box if you wish.

See the Visual Studio .NET help for additional information about using the Add-in Manager.

Add-in Vs. Standalone Differences

The stand-alone CardExplorer.exe and the VS Add-in are both derived from the same CardExplorer control. However, there are several differences that you should be aware of:

- Errors in the Add-in are generally reported in the Output pane of Visual Studio.
- Errors in the stand-alone are generally reported through message boxes.
- The Add-in automatically chooses the correct build file when you use the “NAnt” button. In the stand-alone, you'll need to select the build file for your project.

Templates

The .NET Smart Card Framework SDK provides templates that can be used to create applications for the IDPrime .NET Card. Depending on install-time selections, you might have Visual C# templates, Visual Basic templates, or both installed on the machine.

For each programming language, these templates are available:

- **netCard Server** (use this template to create a server project)
- **netCard Client Console** (use this template to create a client project)

Creating a Server Project

To create a server project, in the **New Project** dialog box, click either **Visual C# Projects** or **Visual Basic Projects**, and then click **netCard Server**, as shown in “Figure 34”.

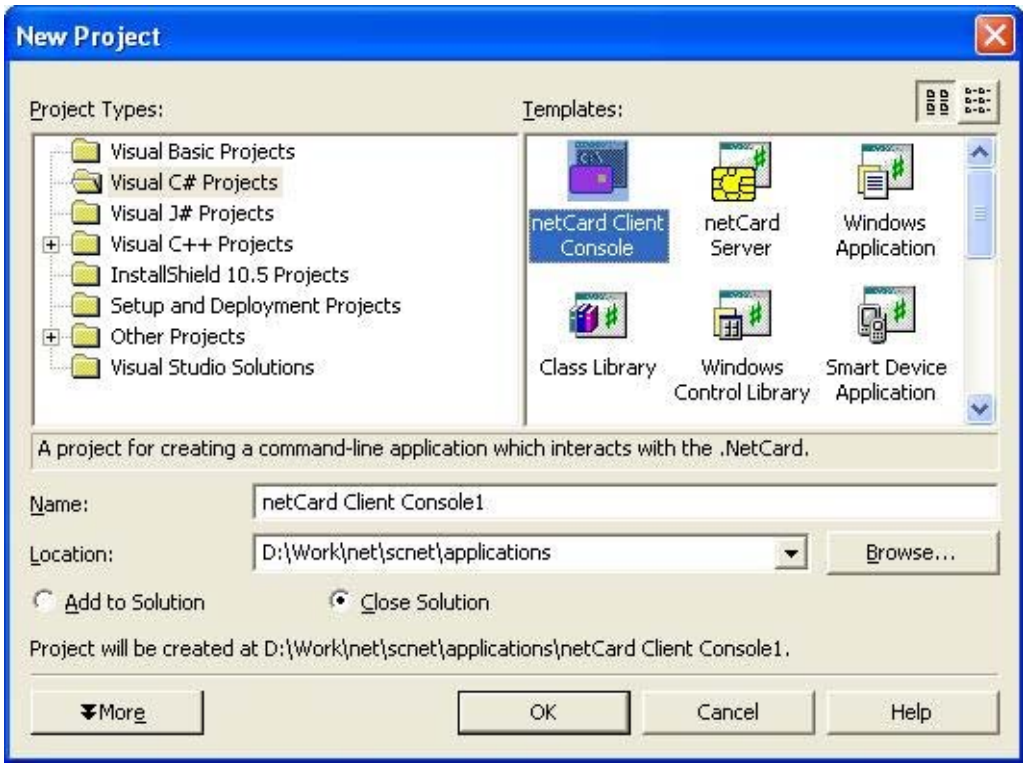
Figure 34 - New Project Dialog Box (netCard Server)

When you start a project using the netCard Server template, application code and project files are automatically created as a starting point. See the walkthrough in “Using Templates to Make a Server Application” on page 76 for instructions about using the template to create a simple server application, and for a list of files created by the template and at compile time.

Creating a Client Project

To create a client project, in the **New Project** dialog box, click either **Visual C# Projects** or **Visual Basic Projects**, and then click **netCard Client Console**, as shown in “Figure 35”.

Figure 35 - New Project Dialog Box (netCard Client Console)



When you start a project using the netCard Client Console template, application code and project files are automatically created as a starting point. See the walkthrough in “Using Templates to Make a Client Application” on page 81 for instructions about using the template to create a simple client application, and for a list of files created by the template and at compile time.

Getting Started

The walkthroughs provide instructions and examples of applications written for specific purposes.

Using Templates to Make a Server Application

This walkthrough guides you through the process to create a simple server application, build the application, load the application onto an IDPrime .NET Card, and start the service contained in the server application so that the service is available to client applications.

The IDPrime .NET Card Add-in to Visual Studio .NET includes a wizard and templates to help you create your first server application. The wizard sets up all the references and includes the necessary namespaces in the skeleton source code.

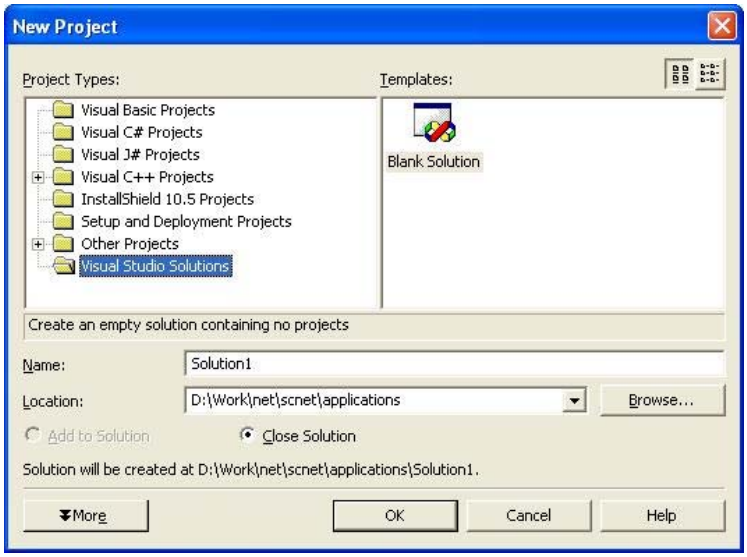
For build-time convenience, create both the server (card) and client (host computer or terminal) applications within a single Visual Studio .NET solution.

Caution: Do not close the VS Card Explorer Add-in until you have finished creating your server application

Creating a New Solution

1 In the Visual Studio .NET toolbar, select **File** > **New** > **Blank Solution**.

Figure 36 - New Blank Solution (Server Applications)



2 In the **New Project** dialog box, type in a name for the new solution and select a location for the solution's files, and then click **OK**.

The following solution directories and files are created at the location specified:

```
[solution_location]\SolutionName\
[solution_location]\SolutionName\SolutionName.sln
[solution_location]\SolutionName\SolutionName.suo
```

In the Visual Studio .NET Solution Explorer, the Solution is displayed like this:

```
Solution 'SolutionName' (0 projects)
```

Opening an Existing Solution

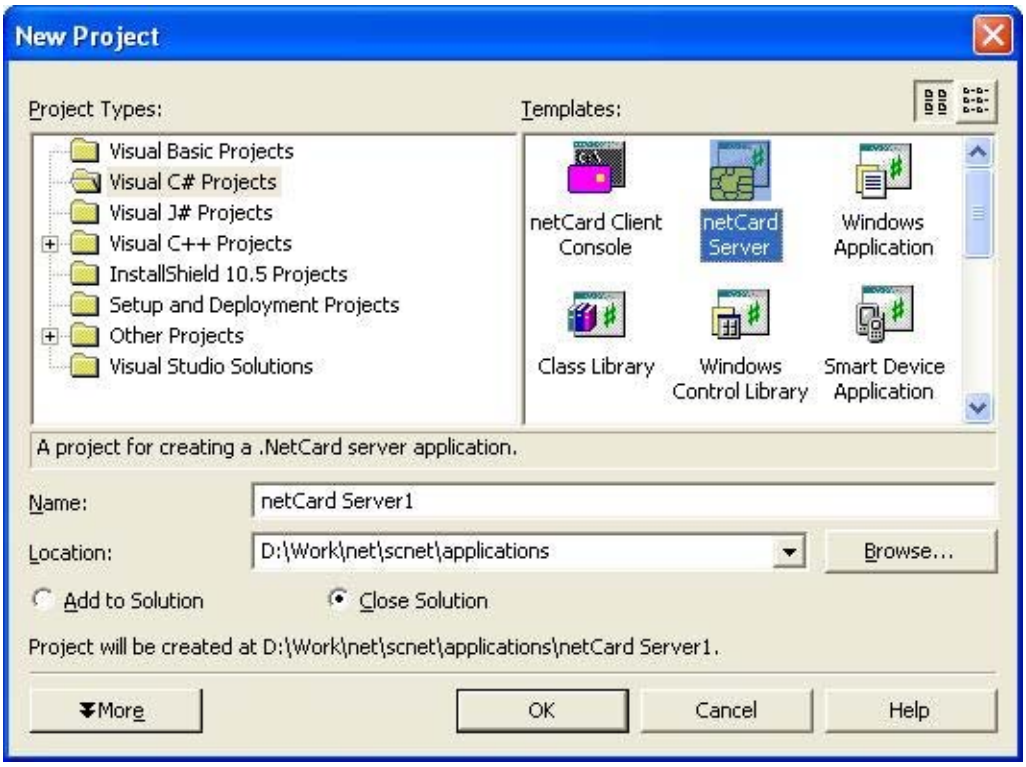
To open a solution, in the Visual Studio .NET toolbar, select **File** > **Open Solution**, and then navigate to the .sln file for the solution you want to open.

Creating an IDPrime .NET Card Server Application

1 In Solution Explorer, right-click the name of the solution and select **Add** > **New Project**.

2 In the **New Project** dialog box, click **Visual C# Projects**, and then click **netCard Server**.

Figure 37 - New Project Dialog Box (netCard Server)



Type in a name for the new project and select a location for the project's files (by default, the project files are stored under the solution), and then click **OK**.

The following project directories and files are created at the location specified in the **New Project** dialog box:

```
[project_location]\ProjectName\AssemblyInfo.cs
[project_location]\ProjectName\DummyKeyPair.snk
[project_location]\ProjectName\MyServer.cs
[project_location]\ProjectName\MyServices.cs
[project_location]\ProjectName\nant.build
[project_location]\ProjectName\ProjectName.csproj
[project_location]\ProjectName\ProjectName.csproj.user
[project_location]\ProjectName\Readme.txt

[project_location]\ProjectName\bin\Debug\

[project_location]\ProjectName\obj\Debug\
[project_location]\ProjectName\obj\Debug\ProjectName.projdata
[project_location]\ProjectName\obj\Debug\temp\
[project_location]\ProjectName\obj\Debug\TempPE
```

The **DummyKeyPair.snk** file is related to strong name signing. All applications added to the card must be signed. DummyKeyPair.snk is a key pair that you can use to test your application. Before you distribute your finished application, replace DummyKeyPair.snk with a specific, non-test key pair.

In the Visual Studio .NET Solution Explorer, the server project is displayed like this:

```
ProjectName
  References
    mscorlib
    netCard
AssemblyInfo.cs
  MyServer.cs
  MyServices.cs
  nant.build
  Readme.txt
```

3 In Solution Explorer, double-click **MyServer.cs** to open the server application source code file.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using SmartCard.Runtime.Remoting.Channels.APDU;

namespace Server
{
   /// <summary>
   /// Summary description for MyServer.
   /// </summary>
   public class MyServer
   {
      /// <summary>
      /// specify the exposed remote object URI.
      /// </summary>
         private const string REMOTE_OBJECT_URI = "myServerUri";

         /// <summary>
         /// Register the server onto the card.
         /// </summary>
         /// <returns></returns>
         public static int Main()
         {
             // Register the channel the server will be listening to.
             ChannelServices.RegisterChannel(new APDUServerChannel());


             // Register this application as a server

RemotingConfiguration.RegisterWellKnownServiceType(typeof(MyServices)
,
      REMOTE_OBJECT_URI, WellKnownObjectMode.Singleton);

             return 0;
         }
   }
}
```

4 Make changes if you prefer, and save.

5 In Solution Explorer, double-click **nant.build** to open the build-time server configuration file. The **nant.build** file is the basis for the Run nant.build icon in the Card Explorer window. The directory property defines the directory in which the assembly will be uploaded, and the service name is the name of the service in your application.

```
<?xml version="1.0"?>
<project name="netCardProject" default="build">

  <target name="build" description="load and execute assembly">
        <load file="bin\debug\netCard Server1.exe" todir="C:\Pub"
execute="true"
                                                  reload="true"
serviceName="myServerUri"/>
   </target>
 </project>
```

Make changes to reflect any changes you made in the MyServer.cs file, and then save. Specifically, if you've updated the service name or executable name, you'll need to update these elements.

More information about nant

6 Build the server application. See the Visual Studio .NET help for instructions about building an application. The following files are added to the project:

```
[project_location]\ProjectName\bin\Debug\ProjectName.exe
[project_location]\ProjectName\bin\Debug\ProjectName.hive
[project_location]\ProjectName\bin\Debug\ProjectName.hmap
[project_location]\ProjectName\bin\Debug\ProjectName.pdb

[project_location]\ProjectName\bin\Debug\stub\AssemblyInfoTemp.cs
[project_location]\ProjectName\bin\Debug\stub\MyServices.cs
[project_location]\ProjectName\bin\Debug\stub\ProjectName_stub.dll

[project_location]\ProjectName\obj\Debug\ProjectName.exe
[project_location]\ProjectName\obj\Debug\ProjectName.pdb
```

Debugging

The system log, **D:\Pub\Console.log**, helps in debugging applications. To enable this feature, include the System.Diagnostics namespace so that you have access to the Debug class. Strings logged to Debug.WriteLine() appear in the file after the program has executed. The logging feature is disabled if you compile the code in Release mode.

You can also debug your card application by developing it in the .NET Framework on the host side first. This practice makes the host's debugging tools available.

Loading the Server onto the Card

1 Insert an IDPrime .NET card into a card reader attached to your computer.

2 In the Card Explorer, click the toolbar logon icon, and type your user name and password.

3 In the Card Explorer, right-click the directory into which the server application should be installed (for example, C:\Pub), and select **Add** > **Load File**.

4 Navigate to the [project_location]\ProjectName\bin\Debug\ProjectName.exe file, and click **Open**.

The server application file is added to the card. The service contained in the server application is not yet running.

Starting a Service

To start the service, in the Card Explorer, right-click the server application file, and select **Execute**. The service contained in the server application is now running and available.

To confirm that the new service is running and available to client applications, click the **Services** tab, and click the **Refresh** icon (see “Toolbar” on page 53). The name of the service is displayed, along with all other services that are currently running on the card.

Deleting a Service

If you want to delete the server application from the card, these are the steps:

1 In the Card Explorer **Services** tab, right-click the service made available by the server application, and select **Delete**.

2 In the Card Explorer **Explorer** tab, right-click the server application file, and select **Delete**.

Note: Some services cannot be deleted.

Using Templates to Make a Client Application

This walkthrough guides you through the process to create a simple client application that will access a service running on an IDPrime .NET Card, and then build the application.

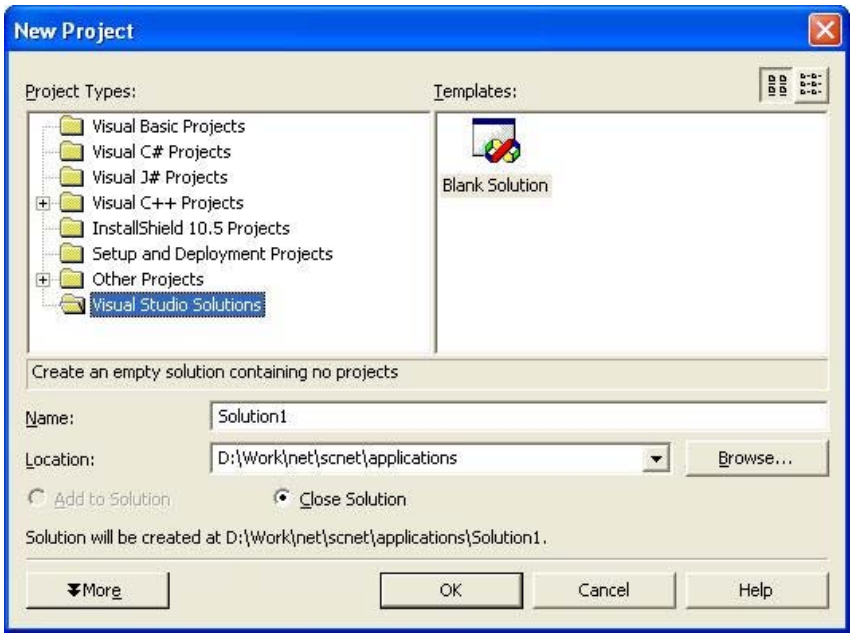
The IDPrime .NET Card Add-in to Visual Studio .NET includes a wizard and templates to help you create your first client application. The wizard sets up all the references and includes the necessary namespaces in the skeleton source code.

For build-time convenience, you can create both the server (card) and client (host computer or terminal) applications within a single Visual Studio .NET solution. If you choose to do this, set the client application project to be the startup project (right-click the client project and select **Set as Setup Project**).

Creating a New Solution

1 In the Visual Studio .NET toolbar, select **File** > **New** > **Blank Solution**.

Figure 38 - New Blank Solution (Server Applications)



2 In the **New Project** dialog box, type in a name for the new solution and select a location for the solution's files, and then click **OK**.

The following solution directories and files are created at the location specified:

```
[solution_location]\SolutionName\
[solution_location]\SolutionName\SolutionName.sln
[solution_location]\SolutionName\SolutionName.suo
```

In the Visual Studio .NET Solution Explorer, the Solution is displayed like this:

```
Solution 'SolutionName' (0 projects)
```

Opening an Existing Solution

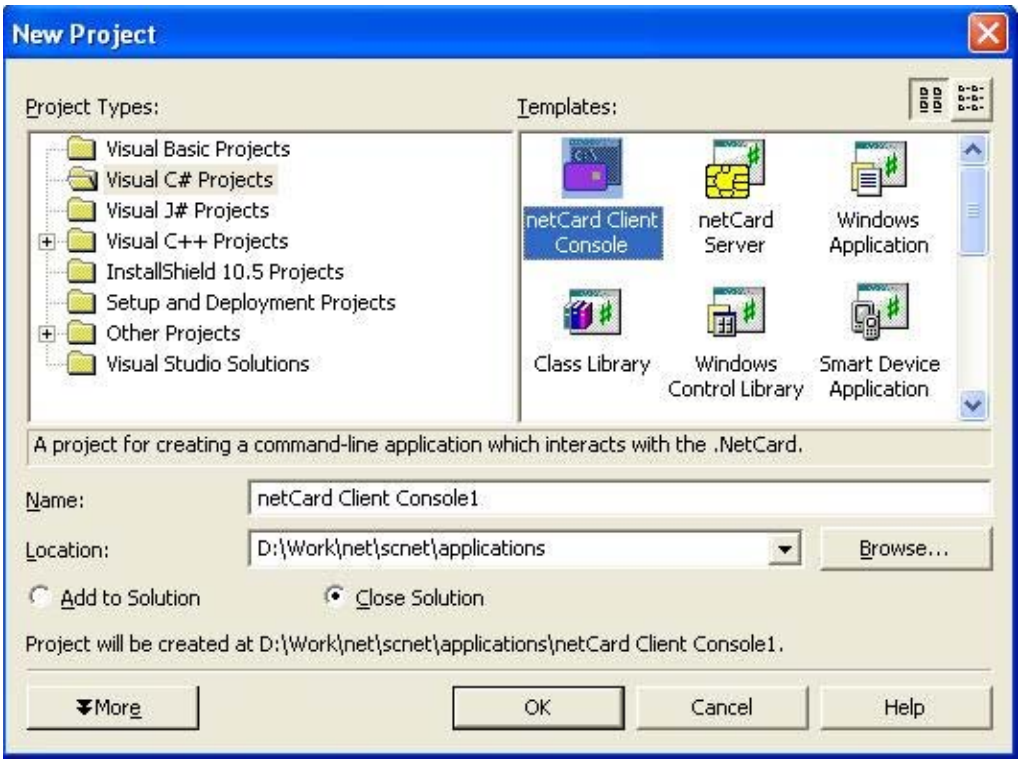
To open a solution, in the Visual Studio .NET toolbar, select **File** > **Open Solution**, and then navigate to the .sln file for the solution you want to open.

Creating a Client Application to Access a Service Running on an IDPrime .NET Card

1 In Solution Explorer, right-click the name of the solution and select **Add** > **New Project**.

2 In the **New Project** dialog box, click **Visual C# Projects**, and then click **netCard Client Console**.

Figure 39 - New Project Dialog Box (netCard Client Console)



Type in a name for the new project and select a location for the project's files (by default, the project files are stored under the solution), and then click **OK**.

The following project directories and files are created at the location specified in the **New Project** dialog box.

```
[project_location]\ProjectName\AssemblyInfo.cs
[project_location]\ProjectName\MyClient.cs
[project_location]\ProjectName\ProjectName.csproj
[project_location]\ProjectName\ProjectName.csproj.user
[project_location]\ProjectName\Readme.txt

[project_location]\ProjectName\bin\Debug\

[project_location]\ProjectName\obj\Debug\
[project_location]\ProjectName\obj\Debug\ProjectName.projdata
[project_location]\ProjectName\obj\Debug\temp\
[project_location]\ProjectName\obj\Debug\TempPE\
```

In the Visual Studio .NET Solution Explorer, the client project is displayed like this:

```
ProjectName
  References
    netCard.Runtime.Remoting
    System
    System.Data
    System.Runtime.Remoting
  AssemblyInfo.cs
  MyClient.cs
  Readme.txt
```

3 In Solution Explorer, double-click **MyClient.cs** to open the client application source code file.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using SmartCard.Runtime.Remoting.Channels.APDU;

// Make sure you add the reference to your server stub dll.
// The stub file is automatically generated for you, under
// [Server Project Output]\Stub).

namespace Client
{
   /// <summary>
   /// Summary description for MyClient.
   /// </summary>
    public class MyClient
    {
        private const string URL = "apdu://promptDialog/myServerUri";

        public MyClient()
        {
            // Create proxy of server
            Server.MyServices myServices = new Server.MyServices();

            // Example: invoke MyServiceExample.
            string result = myServices.MyServiceExample();

            // and display the result on the console output.
            Console.WriteLine(result);
        }

        #region Main Method implementation
        public static void Main()
        {
            APDUClientChannel channel = new APDUClientChannel();

            // register the communication channel
            ChannelServices.RegisterChannel(channel);

          // request access to server object
            RemotingConfiguration.RegisterWellKnownClientType(typeof
             (Server.MyServices), URL);

            new MyClient();

            // unregister the communication channel
            ChannelServices.UnregisterChannel(channel);
        }
        #endregion

    }
}
```

4 Make changes to reflect any changes you made in the server application source code file, and save. The sample code shows communication to the card using the APDU protocol using the `PromptDialog` mechanism, which means that the application will prompt the user to identify the reader in which the IDPrime .NET card is inserted. Two other alternatives are available:

- `SelfDiscover`--an attempt is made to automatically select a reader in which an IDPrime .NET card is inserted; if a reader is not automatically selected, the user is prompted to identify the reader.
- `SelfDiscoverNoPrompt`--an attempt is made to automatically select a reader in which an IDPrime .NET card is inserted; if a reader is not selected, the user is not prompted to identify the reader, and an exception is thrown.

5 Before building the client application, you must create a link to the server application be referencing the *_stub.dll file created when the server was built. In Solution Explorer, right-click the client project's \ProjectName\References, and select **Add Reference**. In the **Add Reference** dialog box, in the **.NET** tab, click **Browse**, and then navigate to the server project's ProjectName\bin\Debug\ ProjectName_stub.dll file, click **Open** in the **Browse** dialog box, and then click **OK**.

The serverProjectName_stub.dll file is added to the project in the client project [`project_location`]\ProjectName\bin\Debug directory and in the Visual Studio .NET Solution Explorer Client Project References.

6 Build the client application. See the Visual Studio .NET help for instructions about building an application. The following files are added to the project:

```
[project_location]\ProjectName\bin\Debug\ProjectName.exe
[project_location]\ProjectName\bin\Debug\ProjectName.pdb

[project_location]\ProjectName\obj\Debug\ProjectName.exe
[project_location]\ProjectName\obj\Debug\ProjectName.pdb
```

Note: If you are building a project including both a server application and a client application, set the client application project to be the startup project (right-click the client project and select **Set as Setup Project**).

Creating an On-Card Application without Templates

This walkthrough provides general guidelines on creating a project for on-card applications without using SDK templates:

- Using a Microsoft .NET Studio project with no on-card templates
- Compiling your source files without a .NET Studio project using either the csc compiler or nAnt

Creating an Access Manager Project Using No On-Card Templates

To create an Access Manager project using no on-card templates:

1 Start Microsoft Visual Studio .NET.

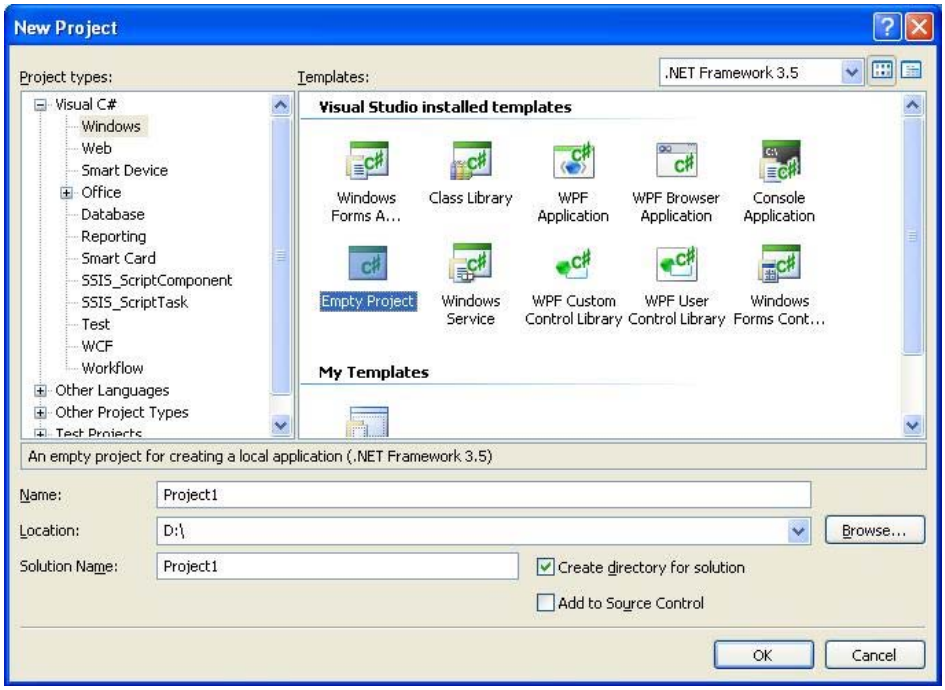
2 Click **File** > **New** > **Project**.

3 In the **New Project** dialog box, make the following choices:

- In the **Project Types** pane, click **Visual C > Windows**.

- In the **Templates** pane, click **Empty Project**.

Figure 40 - New Project (No On-card Templates)



4 Enter a name for the project in the **Name** field, and click **OK**.

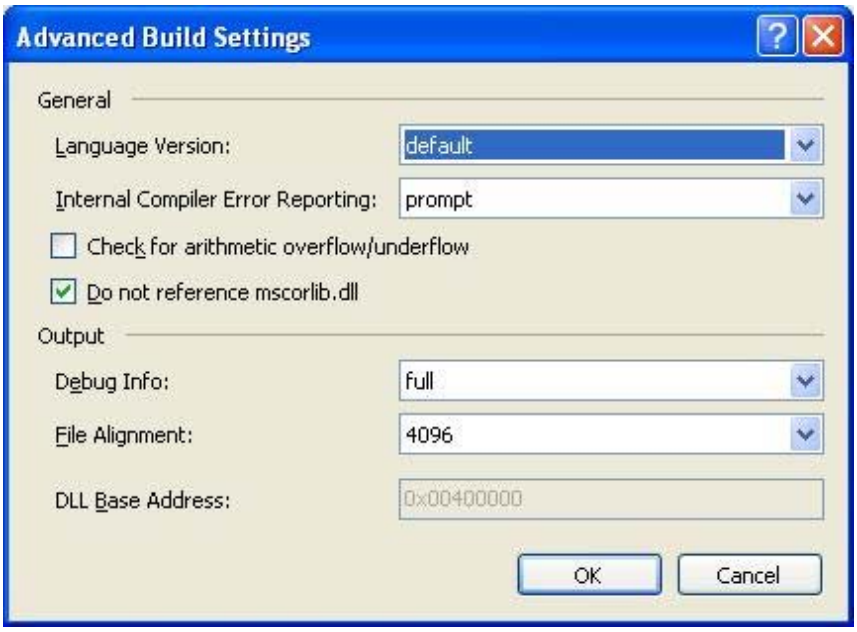
5 In the Microsoft Visual Studio window, click **Project** > **Properties**.

6 In the left pane of the **Property Pages** dialog box, click **Build**.

7 Under **Build**, click **Advanced**.

8 Check the box **Do not reference Mscorlib** as shown in the following example:

Figure 41 - Advanced Build Settings (No On-card Templates)



This action is required because you must use the version of Mscorlib provided with the .NET Framework SDK. The following steps discuss how to load the proper version of Mscorlib.

9 In the *projectname* dialog box, click **OK**.

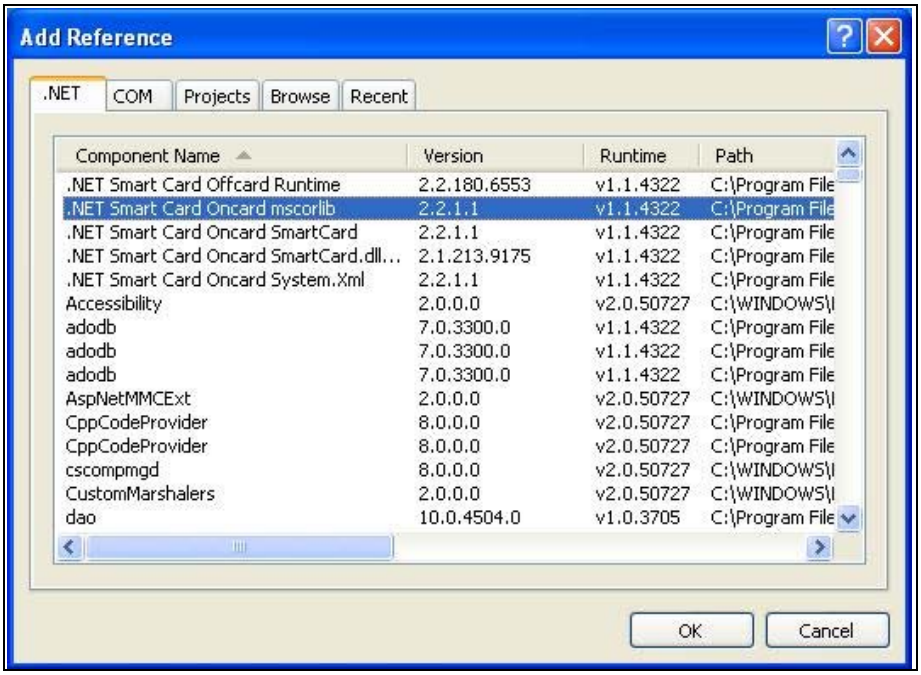
10 In the Microsoft Visual Studio window, open the **Solution Explorer** if it is not already open.

11 In the **Solution Explorer**, right-click **References**.

12 From the pop-up menu, click **Add Reference**.

13 In the **Add Reference** dialog box, click **.NET Smart Card Oncard mscorlib.dll**, and click **OK**, as shown in the following example:

Figure 42 - Add Reference (No On-card Templates)



14 In the **Add Reference** dialog box, click OK.

15 Continue adding your classes to the project as normal.

Building a Project from the Command Line

This section explains the steps to build a project using the .NET command line tools.

Compiling Your Application with csc

The .NET Smart Card SDK provides a version of mscorlib that you must use instead of the Microsoft mscorlib. For this reason, when you compile your application, you must specify /nostdlib in the command line.

An example follows:

```
csc /nostdlib /r:"C:\Program Files\Gemalto\NET Smartcard Framework SDK\
v1.1.201\Libraries\On Card\Framework Libraries\SmartCard.dll"
/r:"C:\Program Files\Gemalto\NET Smartcard Framework SDK\v2.2.181\
Libraries\On Card\Framework Libraries\mscorlib.dll" *.cs
```

Note that in the preceding example, /nodistlib instructs the compiler not to use the Microsoft mscorlib. The following switch instructs the compiler to use the version of mscorlib provided with the .NET Smart Card Framework SDK instead:

```
/r:"C:\Program Files\Gemalto\NET Smartcard Framework SDK\v2.2.181\
Libraries\On Card\Framework Libraries\mscorlib.dll"
```

Building with NAnt

You can use NAnt to build .NET Smart Card applications. You will need to make sure that your NAnt xml files use the .NET SDK libraries rather than the standard Microsoft libraries.

Compiling Your Application Using NAnt

Here is an example .xml file you could use to compile an application using NAnt:

```
<property name="lib.dir" value="C:\Program Files\Gemalto\NET Smartcard
Framework SDK\v2.2.181\Libraries\On Card\Framework Libraries"/>

<csc target="exe" output="bin\Release\CardModuleInterface.dll"
debug="false" nostdlib="true" optimize="true" noconfig="true">
<sources>
      <includes name="*.cs"/>
    </sources>
<references basedir="${lib.dir}">
      <includes name="mscorlib.dll"/>
      <includes name="SmartCard.dll"/>
</references>
</csc>
```

Running Your On-card Application with a Microsoft Debugger

Because the IDPrime .NET Card uses the standard .NET remoting architecture, you can write your application in such a way that you can run the application independently of the transport layer. In this section, we walk you through the process of setting up your application to run on top of both TCP and APDU remoting layers. When you build your application on top of the TCP layers, you can run and debug the application as a server on your local machine. For the current IDPrime .NET Card, only APDU remoting is available on the card.

To build an application that runs properly in both TCP and APDU mode, you need to take the following steps:

Server-Side Code Changes

1. Start with a server project that you have generated using the IDPrime .NET card server wizard
2. In the MyServer.cs file, change the “using” statements from:

```
using System;

using System.Runtime.Remoting;

using System.Runtime.Remoting.Channels;

using SmartCard.Runtime.Remoting.Channels.APDU;
```

to

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
#if MICROSOFT
using System.Runtime.Remoting.Channels.Tcp;
#else
using SmartCard.Runtime.Remoting.Channels.APDU;
#endif
```

3 In your Main method, you need to add another conditional compilation statement to set up either a TCP or an APDU channel, so

ChannelServices.RegisterChannel(new APDUServerChannel());

becomes

```
#if MICROSOFT
ChannelServices.RegisterChannel(new TcpServerChannel(2222));
#else
ChannelServices.RegisterChannel(new APDUServerChannel());
#endif
```

4 When running in TCP mode, the server will shut down as soon as the Main method exits. We could avoid this problem by only running in debug mode and putting break points in, but an easier way to do this is to put a Console.ReadLine into the Main method before returning. So add:

```
#if MICROSOFT
Console.ReadLine()
#endif


right before the return 0; line.
```

Client-Side Code Changes

1 Start with a client project that you have generated using the IDPrime .NET card client wizard.

2 We need to make the same changes that we made in step 2 of “Server-Side Code Changes” to the “using” statements:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
#if MICROSOFT
using System.Runtime.Remoting.Channels.Tcp;
#else
using SmartCard.Runtime.Remoting.Channels.APDU;
#endif
```

3 We'll need to set a conditional compilation statement around the URL to change it back and forth between TCP and APDU:

```
#if MICROSOFT
private const string URL = "tcp://localhost:2222/
myServerUri";
#else
private const string URL = "apdu://promptDialog/myServerUri";
#endif
```

4 Now wrap the channel declaration in a similar conditional compilation statement:

```
#if MICROSOFT
TcpClientChannel channel = new TcpClientChannel();
#else
APDUClientChannel channel = new APDUClientChannel();
#endif
```

Changes to the Project Settings

Each time you switch back and forth between Tcp and APDU modes, you'll need to make a series of changes to your project settings.

Moving from APDU to Tcp

1 In each project, define MICROSOFT in the preprocessor.

2 In the server project, under the **Advanced** tab of the Configuration Properties, you need to set “Do not use Mscorlib” to false.

3 In the references section of the server project, change references from the Gemalto on-card libraries to the standard Microsoft libraries. You will need to explicitly include the System.Runtime.Remoting library.

4 In the references section of the client project, you will need to link to a server stub library that is generated by the soapsuds application. For more information on the soapsuds command, see MSDN.

5 Compile both projects, and you can now start the server in the debugger.

Moving Back from Tcp to APDU

1 In each project, undefine MICROSOFT in the preprocessor.

2 In the server project, under the **Advanced** tab of the Configuration Properties, you need to set “Do not use Mscorlib” to true.

3 In the references section of the server project, change references from the Microsoft libraries to the Gemalto on-card libraries.

4 In the references section of the client project, link to the server stub that is generated by the IDPrime .NET build process.

5 Compile both projects, and you can now load the server to the card.

Code Samples

The .NET Smart Card Framework SDK includes a rich set of developer guidelines: sample code, and hints and “work arounds”. After installation, the sample code can be found here:

```
[install_location]\[product version]\Samples\[sample name]
```

Be sure to check the IDPrime .NET Home page for more examples.

Note: The documentation provided here supersedes the documentation that was previously released in a Word document in the samples directory.

Table 14 - Samples provided with the SDK

Sample	Description
“SecureSession”	Shows how to use sinks to set up a secure session on the card.
“APDUAttribute”	Shows how to communicate with a .NET smart card application using both remoting and legacy APDU's
“Transactions”	Shows how transactions can be used to roll back changes in case of card removal or uncaught exceptions. Also shows the use of objects that are “out of transaction”.

General Instructions

The samples are shipped as VS.NET projects, and have not been compiled. Since the samples are not compiled, the reference table in the client portion of the sample must usually be updated by hand. Take the following steps to build the projects:

1. Build the server project.
2. In the “References” folder, note the name of the stub link.
3. Delete the reference.
4. Add a new reference to the same stub, which will have been generated when you compiled the server project. You can find the stub file in the “stub” directory under the output directory for the server (debug or release).
5. Compile the client project.

SecureSession

This sample shows how to establish a secure channel between the on-card service and a client application, where data is encrypted using a session key. It demonstrates functionality similar to SSL. In SSL, the transport occurs over the Http channel, whereas in this case, it will be over the APDU channel.

Description

SecureSession means that the data exchanged between a client and the on-card service should be encrypted using a symmetric key. The symmetric key is not shared between the on-card service and client application; rather, it is communicated by the client application securely to the service and is valid only for one session. “Session” here means the communication between a smart card and two resets (see “Card Reset Event” on page 47). This symmetric key is called a “session key.”

The most interesting aspect of this sample is that it shows how to establish secure sessions using custom sinks. The encryption and decryption of exchanges between the on-card service and the client are delegated to the custom sink (see “Using Custom Sinks” on page 32), instead of being handled within the application itself. This makes the code in service smaller, portable, and independent of any cryptographic algorithm being used.

It is also envisioned that many client applications communicate with the on-card service at the same time using different session keys. This implies the on-card service needs to have one communication channel per client application (using the custom sink containing the session key generated by a given client). These channels should be created and registered dynamically at a port specified by the client application and should be unregistered at the end of the session.

As mentioned above, the session key is not shared between the service and the client, and it is securely communicated by the client to the service. In order to ensure secure communication of the session key, the public key of the RSA key pair pertaining to the on-card service is used. The Client generates a key 16 bytes in length and encrypts it with the public key of the on-card service. The Service uses its private key to decrypt this session key.

The exchange of the session key cannot occur on a secure channel, since the session key must be exchanged in order to create the secure channel itself. To accomplish the exchange, the service should have an APDUServerChannel listening at a specific port (port 46 in our sample) and should provide methods that should not be invoked over channels using the SecureSessionSink. At first, it would seem that APDUServerChannel listening on port 46 is an answer to the problem, but it actually results in a security flaw. People who understand the .NET remoting architecture should note that there is no explicit link between the transport channel and service. The system will not prevent the invocation of any service method on any channel in a given AppDomain. This shows that MyMethodOnSecureChannel() can be invoked on port 46, which does not have any security.

To avoid the above-mentioned problem, APDUServerChannel at port 46 should also include a custom sink that has the sole purpose of marking when a remote method is invoked at the channel listening at port 46. This is done by setting a static Boolean variable accessible by the service. If a method such as MyMethodOnSecureChannel() is invoked over port 46, the Boolean variable is set to true and implementation of the MyMethodOnSecureChannel() method expects this flag to be false.

Here are the steps for establishing and communicating over a secure channel:

1 Client1 creates and registers an APDUClientChannel without a custom sink.

2. Client1 invokes the GetPublicKey() method of service at the APDUServerChannel listening on port 46. The remote method returns the public modulus and exponent. Client1 then imports the public key into an RSACryptoServiceProvider.
3. Client1 generates a random key (session key) 16 bytes in length.
4. Client1 encrypts the session key generated in step 2 and the PIN of the on-card service using the RSACryptoServiceProvider created in step 1.
5. Client1 invokes the EstablishSecureChannel() method of service at the APDUServerChannel listening on port 46. The arguments passed to this method are the port number (in this case, 7655) at which the newly created channel should listen, the encrypted pin, and the encrypted session key.
6. The EstablishSecureChannel() method in the card decrypts the PIN and session key using the private key. It validates if this PIN is correct. If it is, a new APDUServerChannel listening at the port number passed in the method call is created and registered with the custom sink called SessionSink.
7. Client1 creates and registers an APDUClientChannel with SessionSink using the session key generated in step 3.
8. Client1 invokes the MyMethodOnSecureChannel() method of service at the APDUServerChannel listening at port 7655. The data sent passes through the SecureSessionSink of the APDUClientChannel registered in step 7 and is encrypted with the session key generated in step 3.
9. The SessionSink of the APDUServerChannel listening at port 7655 receives the data and decrypts it with the session key it received in step 5.
10. The MyMethodOnSecureChannel() method is invoked, and it checks if the invocation was on a secure channel by checking the static Boolean flag. The method returns a string that passes through the SessionSink of the APDUServerChannel listening at port 7655 and is encrypted with the session key.
11. The SessionSink of the APDUClientChannel created in step 7 receives the returned data and decrypts it with the session key.

These same steps will be repeated for any client that wants to communicate with the service. The APDUServerChannels with SessionSink are unregistered and destroyed on a reset (that is, the end of a smart card session).

Running the Sample

Load the assembly 'SecureSessionExample.exe' on the card, execute it to start the service. Run the client applications.

Code Extract

Below are some code snippets to illustrate the steps described above.

```
// Create and register a channel at port 46 that uses
// SesssionEstablisherSinkProvider with a ProcessMessage() method set to the
//static Boolean flag
IServerChannelSinkProvider newProvider = new
SessionEstablisherSinkProvider(null, null);
newProvider.Next = new APDUServerFormatterSinkProvider();
APDUServerChannel channel = new APDUServerChannel(newProvider,46);
ChannelServices.RegisterChannel(channel);

// ProcessMessage() method of SessionEstablisherSink
{
   // Let's mark the fact that we are coming from SessionEstablisherSink
   MyService._onEstablisherChannel = true;
```

```
   ServerProcessing srvProc = _nextSink.ProcessMessage(sinkStack, requestMsg,
requestHeaders, requestStream, out responseMsg, out responseHeaders, out
responseStream);
   // returning status information
   return srvProc;
}

// EstablishSecureChannel method of Service (invoked on channel listening at
// port 46)
public void EstablishSecureChannel(int port,byte[] pin,byte[] sessionKey)
{
       if(!_onEstablisherChannel) throw new UnauthorizedAccessException("This
method is to be called with channel having SessionEstablisherSink");
       // Decrypt the pin and sessionKey first
       pin = _rsaProvider.Decrypt(pin,false);
       sessionKey = _rsaProvider.Decrypt(sessionKey,false);
       // Verify the PIN
       _myPIN.Verify(new String(cpin));

       // Create and register the channel at the specified port and using
      //SessionSink, set up the SecureSink properties.
       Hashtable properties = new Hashtable();
       properties["key"] = sessionKey;
       IServerChannelSinkProvider newProvider = new
SessionSinkProvider(properties, null);
       newProvider.Next = new APDUServerFormatterSinkProvider();
       APDUServerChannel channel = new APDUServerChannel(newProvider,port);
       ChannelServices.RegisterChannel(channel);
}

// ProcessMessage() method of SessionSink
{
       // Decrypt the inbound message
       requestStream =
EncryptionHelper.ProcessInboundStream(requestStream,"Rijndael",_sessionKey);
       // Mark that we are coming from SessionEstalisherSink
       MyService._onEstablisherChannel = false;
       ServerProcessing srvProc = _nextSink.ProcessMessage(sinkStack,
requestMsg, requestHeaders, requestStream, out responseMsg, out
responseHeaders, out responseStream);
       // Encrypt the outbound message
       responseStream =
EncryptionHelper.ProcessOutboundStream(responseStream,"Rijndael",_sessionKey);
       // Return the status information
       return srvProc;
}

// MyMethodOnSecureChannel.
public string MyMethodOnSecureChannel(byte[] param1)
{
       // If we came through the channel listening at port, we throw an
      //exception
       if(_onEstablisherChannel) throw new UnauthorizedAccessException("You
cannot access MyMethodOnSecureChannel without SessionSink");

       // Return data will be encrypted with the session key by SessionSink.
       return "This is fun";
}

// Unregister all the channels on reset or power down
private void OnReset(object sender, SessionEndedEventArgs e)
{
```

```
     MyService._onEstablisherChannel = false;
     // On reset unregister the channels that were dynamically created and use
SessionSink
     ....
     ...
 }

 // Client calls EstablishSecureChannel at port 46
{
      APDUClientChannel channel = new APDUClientChannel("noprovider",null);
      ChannelServices.RegisterChannel(channel);
      // Connect to the service on the clear channel.
      MyService sessionEstablish =
(MyService)Activator.GetObject(typeof(MyService),MYSERVICE_URI_1,"noprovider")
;

      // This is the PIN that we share with the card
      byte[] pin = new byte[]
{(byte)'f',(byte)'i',(byte)'r',(byte)'s',(byte)'t',(byte)'P',(byte)'i',(byte)'
n'};
      // Encrypt the pin and session key using the public key of the card
      byte [] encryptedPin = rsaProvider.Encrypt(pin, false);
      byte [] encryptedSessionKey = rsaProvider.Encrypt(sessionKey, false);
      // Now call the EstablishSecureChannel method of the card using the
encrypted PIN
      // and session key. The card will set up an encrypted channel using the
provided       // session key.
      sessionEstablish.EstablishSecureChannel(7655, encryptedPin,
encryptedSessionKey);
}

// Client calls MyMethodOnSecureChannel at 7655
{
      // Set up a Sink Provider with a SessionSink attached to it using the
      // session key as a parameter for creating the SessionSink.
     Hashtable properties = new Hashtable();
     properties["key"] = sessionKey;
     IClientChannelSinkProvider provider = new
APDUClientFormatterSinkProvider();
     provider.Next = new SessionSinkProvider(properties);
     // Create and register a new channel using the sink provider that we've
      // just created.
     channel = new APDUClientChannel("SessionSinkClient", provider);
     ChannelServices.RegisterChannel(channel);
     // Now make a call to get the service again, but this time using the new
      //channel.
     // Note that we explicitly give the name of the channel as the third
     // argument to Activator.GetObject.
     MyService service =
(MyService)Activator.GetObject(typeof(MyService),MYSERVICE_URI_2,"SessionSinkC
lient");
    // The data being sent to and from the card on the secure channel is now
    // encrypted with the session key.
    Console.WriteLine(service.MyMethodOnSecureChannel(new byte[] { 1,2,3,4,5}
));
}
```

APDUAttribute

This sample shows how you can use the APDUAttribute to support both legacy applications and .NET remoting.

Description

The remote methods of the server are decorated with the APDUAttribute, which allows the IDPrime .NET Card to support communication directly using traditional ISO 7816-4 APDUs. This is important if your application must be backwards-compatible with a terminal or device that does not support .NET remoting.

The sample illustrates not only the APDUAttribute itself, but the use of the APDUException attribute, which allows you to translate .NET exceptions into ISO status words.

Two clients are provided. One is a traditional .NET smart card client application that uses remoting to communicate with the card. The second is a C++ application that uses the Winscard API to send APDUs directly to the card.

Execution

Install the 'Gemalto.Samples.APDUAttribute.exe' on the card. You should try running both the CppClient and RemotingClient against the server.

Code Extract

Here is a sample method from the server application. This method is decorated with the APDU attribute as well as the APDUException attribute. The remoting client calls this method by invoking the SampleMethodWhichThrowException method on a proxy object. The C++ client calls this method by sending the APDU “B0340000010x” to the card, where x is passed to the method as a parameter and determines which exception is thrown.

```
[APDU("B0340000")]
[APDUException(typeof(ApplicationException),  "6D01")]
[APDUException(typeof(ArgumentException),     "6D02")]
[APDUException(typeof(ArgumentNullException), "6D03")]
[APDUException(typeof(ArithmeticException),   "6D04")]
public void SampleMethodWhichThrowException(byte b)
{
   WriteLine("SampleMethodWhichThrowException called");

   switch(b)
   {
            case 0x00:
                WriteLine("Throwing ApplicationException..");
                throw new ApplicationException();
            case 0x01:
                WriteLine("Throwing ArgumentException..");
                throw new ArgumentException();
            case 0x02:
                WriteLine("Throwing ArgumentNullException..");
                throw new ArgumentNullException();
            case 0x03:
                WriteLine("Throwing ArithmeticException..");
                throw new ArithmeticException();
```

```
    }
}
```

Transactions

This sample shows how to use Transaction attributes (see “Transactions” on page 37) to ensure that your card data remains in a well-known state when the card is removed or an uncaught exception is thrown. It also shows how the TryCounter class can be used to keep data “out of transaction”.

Description

The service has two fields: **field**, which is an **int**, and **tc**, which is a **TryCounter**. It also has three methods. Each method increments both fields, and then throws an exception that is not caught on the card. The three methods illustrate the different behaviors when methods are or are not decorated with the Transaction attribute. In the methods decorated with the Transaction attribute, any changes made to the int are rolled back when the exception is thrown. The TryCounter, however, continues to increment, because the TryCounter is a special class that is not affected by Transactions. One of the methods (NestedTransaction) calls a method that is not explicitly under transaction in order to demonstrate that methods that are called by a method under transaction are also considered to be under transaction.

Execution

Install the Gemalto.Samples.TransactionSample.exe' on the card, and execute the client on the host.

Code Extract

Here is an extract of a method decorated with the Transaction attribute.

```
[Transaction]
   public void UnderSimpleTransaction()
   {
       field++;       // field is an int
       tc.Value++;    // tc is an instance of the TryCounter class
       throw new ApplicationException();
   }
```

In the above sample, the field value will be reset to its original value when the ApplicationException is thrown, while the TryCounter will be incremented.

8

Client-Side Components

Some components in the .NET Smart Card Framework SDK reside on the client machine.

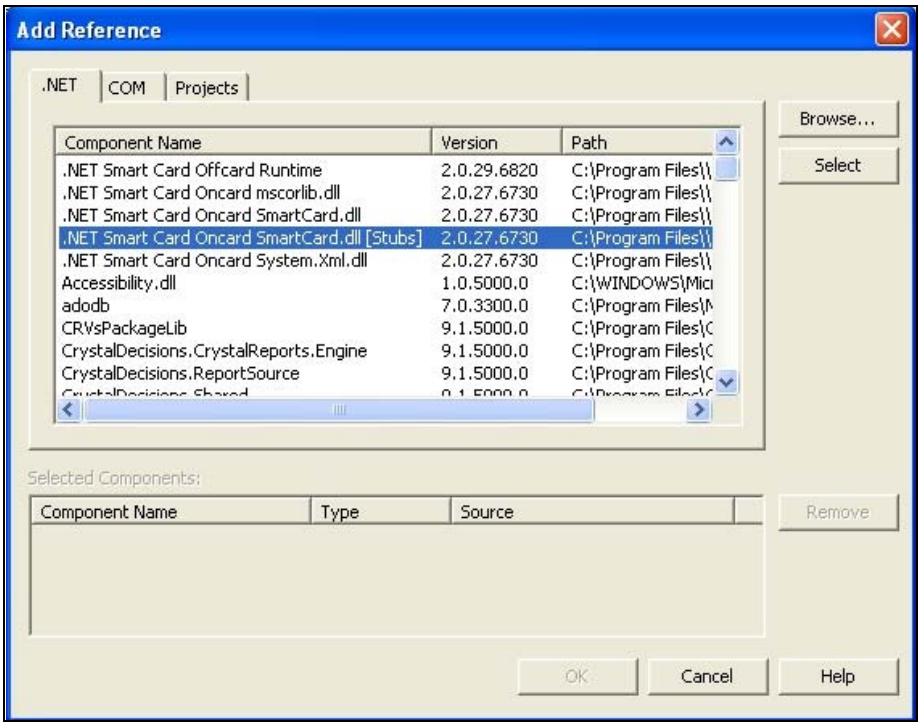
SmartCard_Stub

The SmartCard_Stub.dll contains stub declarations for the card's ContentManager service (described on page 48. You would link to the SmartCard_Stub when you want to access the ContentManager directly using remoting. The full description of the ContentManager service is provided in the .NET Card Framework API documentation.

Referencing the ContentManager from Your Project

You can add the SmartCard_Stub to your project by adding it as a reference (see “Figure 43”).

Figure 43 - Add Reference (SmartCard_Stub)



Once you have added the SmartCard_Stub to your project, you can use remoting to connect to the ContentManager service that is pre-installed on the card.

The .NET Smart Card SDK also supplies a client side wrapper for the ContentManager service called CardAccessor that creates its own remoting connection to the card (see “CardAccessor Class”).

SmartCard.Runtime

The SmartCard.Runtime.dll (which we'll refer to from now on as the Client runtime library) contains several items of interest to developers:

- It contains the client remoting framework to allow you to communicate with .NET smart card services using the standard .NET remoting architecture.
- It contains the CardAccessor class, which is a wrapper for the ContentManager service.
- It contains the definitions of the IAccessManagerClient interface, which you will need in order to build a custom access manager.

A brief overview of each of these is given here. However, the API for each of these is described in full in the .NET Smartcard Client API help file.

Client Remoting Components

The client runtime library contains the APDUClientChannel as well as its supporting classes. When using the .NET remoting framework, you use the APDUClientChannel to talk to your .NET Smart Card service. For example:

```
APDUClientChannel channel = new APDUClientChannel();
ChannelServices.RegisterChannel(channel);
MyOnCardService service =
(MyOnCardService)Activator.GetObject(typeof(MyOnCardService), "apdu://
selfDiscover/MyURI");
```

CardAccessor Class

The CardAccessor class provides a wrapper to the ContentManager class. When using the CardAccessor class, you do not need to set up remoting channels to access the ContentManager. This is handled silently by the CardAccessor class.

AccessManagerClient Interface

The IAccessManagerClientInterface allows you to implement a custom client for an Access Manager (see page 40) that you write. The IAccessManagerClientInterface will be called by the Card Explorer tool to log you into the card when using a card with your AccessManager installed. In order for the Card Explorer to find the right implementation of IAccessManagerClient, you must register your client assembly in the **Config.xml** file of your installation \bin directory.

The SDK ships with an AccessManager installed (for example the Card Module). The full source code is available for both the client and server components. You can find this source code in the \Samples\src\SampleAccessManager directory of your SDK.

C++ Marshaller

Why a C++ Marshaller?

The IDPrime .NET card allows application developers to concentrate on building service interfaces on the card without worrying about the implementation of the APDU transport protocol. By default, this abstraction takes place using the .NET remoting architecture. However, you may wish to write an application on the IDPrime .NET card that does not operate in a .NET environment. This could happen if, for example, you were writing a service such as a GINA, which would be invoked at boot time, and might need to launch before the .NET runtime could be loaded.

Where Can I Find the C++ Marshaller?

This is provided free of charge as part of the IDPrime .NET SDK.

When you install the SDK, the C++ Marshaller files can be found in the directory

```
[Install Directory]\[Install Version]\Libraries\Off Card\Marshaller
```

Using the Marshaller

When you build an IDPrime .NET Card server application using Visual Studio, there are three post-build processes that take place. First, a converter runs that manipulates the IL code in your assembly to prepare it for loading to the card. Then, a stub DLL is generated for your public interface and placed in the “stub” directory, which will be located under the target directory for your .exe (typically ./bin/Debug/stub or ./bin/Release/stub). Finally, the C++ marshaller creates a C++ header file (.h) and C++ source file (.cpp) in the stub directory.

To use the generated C++ files, include them in your project. You'll then need to extend your Include path to:

```
[Install Directory]\[Install Version]\Libraries\Off Card\Marshaller
```

You will also need to link to Marshaller.lib and extend your Link path to:

```
[Install Directory]\[Install Version]\Libraries\Off Card\Marshaller
```

Part II

.NET Minidriver API

Introduction to Part 2

Part 2 of this document lists all of the functions that are available at the API level for IDPrime .NET cards. These functions can be called using .NET Remoting, that is, there is no need to build an explicit APDU to call them. In order to call these methods, it is necessary to include the CardModuleService.cpp and CardModuleService.h files in the client application program. These files can be found in the SDK associated with the .NET card.

The card API has an interface that is common to the V5, V6 and V7 specifications. The V5 methods are different to the V6 and V7 methods, so this document describes them separately.

In a Base CSP architecture, this card API can be called by the host Minidriver .dll that is installed on the PC and is compliant with the Microsoft Minidriver V5, V6 and V7 specifications.

10

Minidriver Interface V5/V6/V7

This chapter shows the code for the Minidriver interface which is common to the three versions of the Microsoft specifications, V5, V6 and V7 supported by IDPrime .NET cards.

Note: The end of the code has a part that is specific to the V7 of the minidriver.

```
namespace Axalto.netCard.CardModuleInterface
{
    using System;

    public interface ICardModuleInterface
    {
        // Auhentication management methods
        byte[] GetChallenge();
        void ExternalAuthenticate(byte[] response);
        void ExternalAuthenticateAM(byte[] response);
        void ChangeReferenceData(byte mode, byte role, byte[] oldPin,
             byte[] newPin, int maxTries);
        void VerifyPin(byte role, byte[] oldPin);
        int GetTriesRemaining(byte role);
        void LogOut(byte role);
        bool IsAuthenticated(byte role);
        byte MaxPinRetryCounter { get; }
        bool AdminPersonalized { get; }
        bool UserPersonalized { get; }

        // Containers & Crypto management methods
        void CreateCAPIContainer(byte ctrIndex, bool keyImport,
             byte keySpec, int keySize, byte[] keyValue);
        void DeleteCAPIContainer(byte ctrIndex);
        byte[] GetCAPIContainer(byte ctrIndex);
        byte[] PrivateKeyDecrypt(byte ctrIndex, byte keyType,
               byte[] encryptedData);

        // Information management methods
        int[] QueryFreeSpace();
        int[] QueryKeySizes();

        // Filesystem management methods
        void CreateFile(string path, byte[] acls, int initialSize);
        void CreateDirectory(string path, byte[] acls);
        void WriteFile(string path, byte[] data);
```

```
            byte[] ReadFile(string path, int maxBytesToRead);
            void DeleteFile(string path);
            void DeleteDirectory(string path);
            string[] GetFiles(string path);
            byte[] GetFileProperties(string path);

            // Version management
            string Version { get; }
            void SetHostVersion(uint hostVersion);

        // MiniDriver V6/V7 specific


        // Auhentication management methods
            byte[] GetChallengeEx(byte role);
            byte[] AuthenticateEx(byte mode, byte role, byte[] pin);
            void   DeauthenticateEx(byte roles);
            void   ChangeAuthenticatorEx(byte mode, byte oldRole,
                   byte[] oldPin, byte newRole, byte[] newPin,
                   int maxTries);

            // Properties management methods
            byte[] GetContainerProperty(byte ctrIndex, byte property,
                   byte flags);
            void   SetContainerProperty(byte ctrIndex, byte property,
                   byte[] data, byte flags);
            byte[] GetCardProperty(byte property, byte flags);
            void   SetCardProperty(byte property, byte[] data,
                   byte flags);
        }
    }
```

11

Minidriver V5 Methods

This chapter lists the methods that are available for the V5 minidriver, that is those defined in V5 of the Microsoft Smart Card Minidriver Specification.

Note: The methods in this section are those used by the card. The minidriver Axalto.dll transforms the methods defined in the Microsoft V5 specification to those in this chapter. The names of the methods in this chapter are not identical to their MS equivalents.

Some of these methods use the role identifier “access manager admin key”. This key only applies if the card module was installed with the -s parameter. If it was not installed with the -s parameter, the “access manager admin key” is the same as the default “admin key” (role ID #2)

Authentication Management Methods

byte[] GetChallenge()

Returns a challenge value (without 0x00) of 8 bytes. It is possible to have consecutive repeated calls to this function.

void ExternalAuthenticate(byte[] response)

Authenticates the Admin role if response = 3DES_CBC-Encrypt (*challenge*, *Admin_Key*).

The challenge is 8 bytes of data returned by GetChallenge() and *Admin_Key* is a 24-byte 3DES key with a default value = 00…00.

void ExternalAuthenticateAM(byte[] response)

Authenticates the Card Access Manager Admin role if response = 3DES_CBC-Encrypt (*challenge*, *Access_Manager_Admin_Key*).

The challenge is 8 bytes of data returned by GetChallenge() and *Access_Manager_Admin_Key* is a 24-byte 3DES key with a default value = FF…FF.

void ChangeReferenceData(byte mode, byte role, byte[] oldPin, byte[] newPin, int maxTries)

Unblocks the User PIN or changes the User PIN, 3DES Admin Key, or 3DES Access Manager Admin Key.

mode: 0 = Change, 1 = Unblock.

role: 0x01 = User PIN, 0x02 = Admin Key, 0x80 = Access Manager Admin Key.

Unblock (*mode* = 1):

- ***role***: The role to be unblocked, it must be 1. Only a User PIN can be unblocked.
- ***oldPin***: It must be a valid response (8 bytes) of a challenge obtained with GetChallenge().
- ***newPin***: The new PIN value to be set for User PIN. PIN value can be 4 to 256 bytes long.

The command cannot be used to unblock the Admin Key or Access Manager Admin Key.

Caution: If you block the Admin key, the card can no longer be used.

Change (*mode* = 0):

- ***role***: The role identifier to be changed, User PIN, Admin Key or Access Manager Admin Key.
- ***oldPin***:
  - If ***role*** = Admin Key or Access Manager Admin Key, it must be a valid response (8 bytes) of a challenge obtained with GetChallenge().
  - If ***role*** = User, it must be the current value of User PIN.
- ***newPin***:
  - If ***role*** = Admin Key or Access Manager Admin Key, it must be a 24-byte 3DES key value that will be the new Admin Key.
  - If ***role*** = User PIN, it must be the new PIN value of the User PIN. PIN value can be 4 to 256 bytes long.

In all cases ***maxTries*** is the new max tries counter for the changed/unblocked PIN/Key. If ***maxTries*** = -1 it means that the current max tries counter is not modified.

If the Change/Unblock operation fails an exception is thrown.

void VerifyPin(byte role, byte[] oldPin)

Verifies the User PIN.

role: 1 = User PIN, other values are RFU.

oldPin: The PIN value to be verified. PIN value can be 4 to 256 bytes long.

int GetTriesRemaining(byte role)

Returns the current retries counter of the requested ***role***.

role: 1 = User PIN, 2 = Admin Key.

void LogOut(byte role)

Deauthenticates the specified ***role***.

role takes one of the following values:

- 0x01 = User PIN
- 0x02 = Admin Key
- 0X04 = PIN role #3
- 0X08 = PIN role #4
- 0X10 = PIN role #5
- 0X20 = PIN role #6
- 0X40 = PIN role #7
- 0X80 = Access Manager Admin Key

bool IsAuthenticated(byte role)

Indicates if the specified ***role*** is currently authenticated.

role takes one of the following values:

- 0x01 = User PIN
- 0x02 = Admin Key
- 0X04 = PIN role #3
- 0X08 = PIN role #4
- 0X10 = PIN role #5
- 0X20 = PIN role #6
- 0X40 = PIN role #7
- 0X80 = Access Manager Admin Key

Returns False if not authenticated, True if authenticated.

byte MaxPinRetryCounter {get;}

Gets the maximum User PIN retry counter value.

bool AdminPersonalized {get;}

Gets indication if the Admin Key (role 2) has been changed from initial value. TRUE is already changed, else FALSE.

bool UserPersonalized {get;}

Gets indication if the User PIN (role 1) has been changed from initial value. TRUE is already changed, else FALSE.

Containers & Crypto Management Methods

void CreateCAPIContainer(byte ctrIndex, bool keyImport, byte keySpec, int keySize, byte[] keyValue)

Creates an RSA key in the specified container. The key can be generated on board or imported. A key container can contain up to 2 RSA key-pairs, the Exchange key and the Signature key. The 2 keys in a container can have different sizes from 512 bits to 2048 bits.

The User PIN must be verified to authorize access to this command.

If the key already exists, it will be overwritten.

ctrIndex: The container index to be created, from 0 to N (N = max number of containers supported by the minidriver, 15 in current version).

keyImport: TRUE if the key is imported, FALSE if the key must be generated on board.

keySpec: 1 = Exchange key, 2 = Signature key.

keySize: The size in bits of the key to be created (between 512 and 2048).

keyValue: A key blob (byte array) containing the RSA key-pair value when it is imported. Null if the key must be on board generated.

The key blob must be formatted as follows:

Key Blob = Prime P || Prime Q || Inverse Q || DP || DQ || Private Exponent D || Modulus || Public Exponent E

- Prime P length = Key_Size_Bytes / 2 bytes
- Prime Q length = Key_Size_Bytes / 2 bytes
- Inverse Q length = Key_Size_Bytes / 2 bytes
- DP length = Key_Size_Bytes / 2 bytes
- DQ length = Key_Size_Bytes / 2 bytes
- Private Exponent D length = Key_Size_Bytes bytes
- Modulus length = Key_Size_Bytes bytes
- Public Exponent E length = 4 bytes

void DeleteCAPIContainer(byte ctrIndex)

Deletes the key(s) in the specified container. If the container contains both Exchange and Signature keys, both will be deleted.

The User PIN must be verified to authorize access to this command.

ctrIndex: The container index to be deleted, from 0 to N (N = max number of containers supported by the minidriver, 15 in current version).

byte[] GetCAPIContainer(byte ctrIndex)

Returns a byte array blob containing the public key(s) in the selected container.

This command execution is free but it throws an exception if the container is empty (no signature key and no exchange key).

ctrIndex: The container, from 0 to N (N = max number of containers supported by the minidriver, 15 in current version).

The returned blob is formatted as follows:

Blob = [Signature_Pub_Key] || [Exchange_Pub_Key]

Signature_Pub_Key and Exchange_Pub_Key are optional depending on which key exists in the container and are sequences of 3 TLV formatted as follows:

- T_Key_Type = 0x03
- L_Key_Type = 0x01
- V_Key_Type = 0x01 for Exchange_Pub_Key or 0x02 for Signature_Pub_Key

- T_Key_Pub_Exp = 0x01
- L_Key_Pub_Exp = 0x04
- V_Key_Pub_Exp = Value of Public key Exponent on 4 bytes.

- T_Key_Modulus = 0x02
- L_Key_Modulus = Key_Size_Bytes >> 4 (1 byte !)
- V_Key_Modulus = Value of Public key Modulus on Key_Size_Bytes bytes.

The 4-bit shift on L_Key_Modulus is to be able to pass the Modulus length on 1 byte for values 64 to 256 (512 bits to 2048 bits).

byte[] PrivateKeyDecrypt(byte ctrIndex, byte keyType, byte[] encryptedData)

Performs an RSA raw decryption with the selected key.

The User PIN must be verified to authorize access to this command.

ctrIndex: The container index to be used, from 0 to N (N = max number of containers supported by the minidriver, 15 in current version).

keySpec: 1 = Exchange key, 2 = Signature key.

encryptedData: A byte array containing the data to be decrypted. Size of the encrypted data must be equal to the key size in bytes.

This command returns a byte array containing the decrypted data.

Information Management Methods

int[] QueryFreeSpace()

Returns a 3 integer array with the following information:

Int[0]: Number of used containers.

Int[1]: Maximum number of containers managed by the minidriver.

Int[2]: Free memory size (in bytes) on the card.

This command execution is free.

int[] QueryKeySizes()

Returns a 4 integer array with the following information:

Int[0]: Minimum key size (512).

Int[1]: Maximum key size (2048).

Int[2]: Key size increment (128).

Int[3]: Default key size (1024).

This command execution is free.

File System Management Methods

void CreateFile(string path, byte[] acls, int initialSize)

Creates a file in the card that will be used to store information (byte array format).

The User PIN or the Admin key must be verified to authorize access to this command.

path: The full name of the file. Only one level of directory is supported, so a name can be “fname” if the file is in the root directory or “dname\fname” if the file is in the “dname” directory.

- Directory and file names are limited to 8 characters.
- If the file (or a directory with the same name as the file to be created) already exists an exception is thrown.
- If the directory name does not exist an exception is thrown.

acls: The access conditions list of the file to be created. The ***acls*** parameter is a 3-byte array formatted as follows:

- acl[0]: Admin AC.
- acl[1]: User AC.
- acl[2]: Everyone AC.

Each AC is a bit mask of the following rights:

- Right_Execute = 1 (RFU)
- Right_Write = 2
- Right_Read = 4

initialSize: The size that must be reserved for the file at creation time. The file size is variable, which means that when writing in the file the initial size can be increased or decreased depending on the size of the data to write.

void CreateDirectory(string path, byte[] acls)

Creates a directory in the root (only one level) of the file system.

The User PIN or the Admin key must be verified to authorize access to this command.

path: The name of the directory.

- Directory and file names are limited to 8 characters.
- If the directory (or a file with the same name as the directory to be created) already exists an exception is thrown.

acls: The access conditions list of the directory to be created. The ***acls*** parameter is a 3- byte array formatted as follows:

- acl[0]: Admin AC.
- acl[1]: User AC.
- acl[2]: Everyone AC.

Each AC is a bit mask of the following rights:

- Right_Write = 2 // Apply to CreateFile and DeleteFile in this directory
- Right_Read = 4 // Apply to EnumFiles in this directory

Note: The root directory acls must be:

- acl[0]: 6 // Admin Write + Read
- acl[1]: 6 // User Write + Read
- acl[2]: 4 // Everyone read

void WriteFile(string path, byte[] data)

Writes data into an existing file.

The write AC for file must be respected (User/Admin/Everyone authenticated depending on file acls).

path: The full name of the file.

- Directory and file names are limited to 8 characters.
- If the file does not exist, an exception is thrown.
- If the directory name does not exist, an exception is thrown.

data: A byte array containing the data to write in the file. If the data length is different from the current file size the file is resized according to the new size.

byte[] ReadFile(string path, int maxBytesToRead)

Reads data from an existing file.

The AC for file read must be respected (User/Admin/Everyone authenticated depending on file acls).

path: The full name of the file.

- Directory and file names are limited to 8 characters.
- If the file does not exist, an exception is thrown.

maxBytesToRead: RFU, must be 0.

This command returns a byte array with the contents of the file.

void DeleteFile(string path)

Deletes an existing file.

The write AC for the parent directory must be respected (User/Admin/Everyone authenticated depending on directory acls).

path: The full name of the file.

- Directory and file names are limited to 8 characters.
- If the file does not exist, an exception is thrown.

void DeleteDirectory(string path)

Deletes an existing directory.

The write AC for parent directory (always root directory) must be respected (User/Admin/Everyone authenticated depending on root directory acls).

path: The name of the directory.

- Directory and file names are limited to 8 characters.
- If the directory does not exist, an exception is thrown.
- If the directory is not empty an exception is thrown.

string[] GetFiles(string path)

Returns the list of files contained in a directory.

The read AC for parent directory must be respected (User/Admin/Everyone authenticated depending on directory acls).

path: The name of the directory containing the files to be listed.

- Directory and file names are limited to 8 characters.
- If the directory does not exist, an exception is thrown.
- Null means root directory.

This command returns a string array where each string is a short file name (without directory name) contains in the directory.

byte[] GetFileProperties(string path)

Returns information about an existing file.

The read AC file must be respected (User/Admin/Everyone authenticated depending on file acls).

path: The full name of the file.

- Directory and file names are limited to 8 characters.
- If the file does not exist, an exception is thrown.

The returned information is a 7-byte array formatted as follows:

- byte[0]: acls[0] of the file.
- byte[1]: acls[1] of the file.
- byte[2]: acls[2] of the file.
- byte[3…6]: File length with (LSB->MSB).

Version Management Methods

string Version {get;}

Returns a version string that identifies the MiniDriver on card application:

- “5.0.0.0”
- “6.0.0.0”
- “7.0.0.0”
- “7.1.0.0”

void SetHostVersion(uint hostVersion)

This is used by the host minidriver to inform the card of it's version number (version of the host). The minidriver assembly can use this information to adapt some behavior to the host version.

12

Minidriver V6/V7 Methods

This chapter lists the methods that are available for the V6 and V7 minidrivers.

Note: Some of these methods use the role identifier “access manager admin key”. This key only applies if the card module was installed with the -s parameter. If it was not installed with the -s parameter, the “access manager admin key” is the same as the default “admin key” (role ID #2)

Authentication Management Methods

Role Identifiers

In the following methods the ***role*** parameter can have the following values:

- User PIN = 0x01
- Admin Key = 0x02
- PIN#3 = 0x04
- PIN#4 = 0x08
- PIN#5 = 0x10
- PIN#6 = 0x20
- PIN#7 = 0x40
- Access Manager Admin Key = 0x80

byte[] GetChallengeEx(byte role)

Returns a challenge value (without 0x00) of 8 bytes. It is possible to have consecutive repeated calls to this function.

role: Admin Key or Access Manager Admin Key, other values are RFU in current version.

byte[] AuthenticateEx(byte mode, byte role, byte[] pin)

Authenticates the requested ***role*** depending on the authentication ***mode***.

mode: 0 = Plaintext, 1 = Get session PIN, 2 = Verify session PIN.

role: User PIN, Admin Key, PIN#3, PIN#4, PIN#5, PIN#6, PIN#7 or Access Manager Admin Key.

Plaintext Authentication (*mode* = 0):

This mode is similar to VerifyPin() v5 method if ***role*** = User PIN or ***role*** = PIN#3 to PIN#7 and it is similar to ExternalAuthenticate() v5 method if ***role*** = Admin Key or Access Manager Admin Key.

- If ***role*** = User PIN or ***role*** = PIN#3 to PIN#7, ***pin*** is the PIN value to be verified for this role. PIN value can be 4 to 256 bytes long.
- If **role** = Admin Key, ***pin*** must be a *response* to a previous *challenge* with *response* = 3DES_CBC-Encrypt(*challenge*, *Admin_Key*). The *challenge* is 8 bytes of data returned by GetChallengeEx() and *Admin_Key* is a 24 bytes 3DES key with a default value = 00…00.
- If **role** = Access Manager Admin Key, ***pin*** must be a *response* to a previous *challenge* with *response* = 3DES_CBC-Encrypt(*challenge*, *Access_Manager_Admin_Key*). The *challenge* is 8 bytes of data returned by GetChallengeEx() and *Access_Manager_Admin_Key* is a 24 bytes 3DES key with a default value = FF…FF.

Session PIN Authentication (*mode* = 1 / 2):

This mode enables verification of a PIN without sending the PIN value to the card. The Session PIN Authentication process is split into 2 steps, a first call to get a Session PIN token (***mode*** = 1) and a second call to verify a Session PIN cryptogram computed by the application (***mode*** = 2). The Session PIN Authentication mode is not supported with Admin Key or Access Manager Admin Key.

- If ***mode*** = 1 (Get Session PIN), the card returns an 8-byte Session PIN *token* = 3DES_ECB-Encrypt(*random*, *3DES_PIN_Key*).
  - *random* = An 8-byte random computed by the card.
  - *3DES_PIN_Key* = SHA1(*Real_PIN_Value*) || 00000000 24 bytes).

    Real_PIN_Value = The real PIN value known by the card.
- If ***mode*** = 2 (Verify Session PIN), ***pin*** must be a 20-byte Session PIN *cryptogram* = SHA1(*random* || *Real_PIN_Value*) to be successfully verified by the card.
  - *random* = The 8-byte random computed by the card.
  - *Real_PIN_Value* = The real PIN value known by the card.

An external application that wants to use the Session PIN mode to authenticate a User, must process as follows:

- Call AuthenticateEx() with ***mode*** = 1 to get the Session PIN *token*.
- Collect the *PIN_Value* known by the User (ex: UI for PIN capture).
- Decrypt the Session PIN token to retrieve the supposed card *random* where *supposed_random* = 3DES_ECB-Decrypt (*token*, *3DES_PIN_Key*)
- *3DES_PIN_Key* = SHA1(*PIN_Value*) || 00000000 24 bytes).
- Build the Session PIN *cryptogram* = SHA1(*supposed_random* || *PIN_Value*).
- Call AuthenticateEx() with ***mode*** = 2 to verify the Session PIN cryptogram.
- The card will only authenticate the user if the entered PIN was correct and permitted to decrypt properly the Session PIN token encrypted by the card with the real PIN value.

The PIN ratification counter is decremented during Get Session PIN, it is reset to maximum counter value on a successful Verify Session PIN.

The ***pin*** parameter is not used during Get Session PIN.

This method throws an exception if authentication fails.

void DeauthenticateEx(byte roles)

Deauthenticates the specified ***roles***.

roles: Bits mask of the roles identifiers to be deauthenticated. Multiple roles deauthentication is possible using this method.

void ChangeAuthenticatorEx(byte mode, byte oldRole, byte[] oldPin, byte newRole, byte[] newPin, int maxTries)

Changes the User PIN or Unblocks the User PIN, Admin Key, PIN#3, PIN#4, PIN#5, PIN#6, PIN#7 or Access Manager Admin Key.

mode: 1 = Unblock, 2 = Change.

oldRole: User PIN, Admin Key, PIN#3, PIN#4, PIN#5, PIN#6, PIN#7 or Access Manager Admin Key.

newRole: User PIN, Admin Key, PIN#3, PIN#4, PIN#5, PIN#6, PIN#7 or Access Manager Admin Key.

Unblock (mode = 1):

- ***oldRole***: The role identifier to be verified to authorize the unblock command. By default the unblock role is Admin Key (0x02) for all the PINs, it can be modified by changing the PIN Info property (see SetCardProperty() method).
- ***oldPin***:
  - If ***oldRole*** = Admin Key, it must be a valid response (8 bytes) of a challenge obtained with GetChallengeEx(). This is the default case since PIN Info property was not changed.
  - If ***oldRole*** = User PIN or PIN#3 to PIN#7, it must be the current value of ***oldRole*** session PIN.
- ***newRole***: The role identifier to be unblocked (User PIN or PIN#3 to PIN#7).
- ***newPin***: The new PIN value to be set for ***newRole*** PIN. PIN value can be 4 bytes to 256 bytes long. The PIN value is in clear text.

You cannot use the command to unblock the Admin Key or Access Manager Admin Key.

Caution: If you block the Admin Key or the Access Manager Admin Key, the card can no longer be used.

Change (mode = 2):

- ***oldRole***: The role identifier to be changed.
- ***oldPin***:
  - If ***oldRole*** = Admin Key or Access Manager Admin Key, it must be a valid response (8 bytes) of a challenge obtained with GetChallengeEx().
  - If ***oldRole*** = User PIN or PIN#3 to PIN#7, it must be the current value of ***oldRole*** session PIN.
- ***newRole***: is the role to changed. It must be the same role identifier than ***oldRole*** (***newRole*** = ***oldRole***).
- ***newPin***:
  - If ***newRole*** = Admin Key or Access Manager Admin Key, it must be a 24 bytes 3DES key value that will be the new Admin Key or Access Manager Admin Key.

- If ***newRole*** = User PIN or PIN#3 to PIN#7, it must be the new PIN value of ***newRole*** PIN. The new PIN value is encrypted using the session PIN.

In all cases ***maxTries*** is the new max tries counter for the changed/unblocked PIN/Key. If ***maxTries*** = -1 it means that the current max tries counter is not modified.

If the Change/Unblock operation fails an exception is thrown.

Example Scenario to Change a PIN with ChangeAuthenticatorEx():

This scenario describes how to change a PIN with the ChangeAuthenticatorEx() function. In this description **ClearOldPin** is the plaintext value of the old PIN and **ClearNewPin** is the plaintext value of the new PIN:

Step 1: Compute Session PIN:

1. Call AuthenticateEx() with mode = 0x01 (Generate Session PIN) -> **CardRnd**
2. Generate a 3DES key = SHA1(**ClearOldPin**) || 00000000 -> **3DESKey**
3. Decrypt the card random = 3DES_ECB_Decrypt(**CardRnd**, **3DESKey**) -> **ClearCardRnd**
4. Compute Session PIN = SHA1(**ClearCardRnd** || **ClearOldPin**) -> ***SessionPin***

Step 2: Encrypt the new PIN:

1. Compute padded new clear PIN = PKCS#7(**ClearNewPin**) -> **PaddedClearNewPin**
2. Encrypt the padded clear new PIN = 3DES_ECB_Encrypt(**PaddedClearNewPin**, **3DESKey**) -> ***EncryptedNewPin***

Step 3: Change the PIN:

1. Call CardChangeAuthenticatorEx() with oldPin = ***SessionPin*** and newPin = ***EncryptedNewPin***

Properties Management Methods

byte[] GetContainerProperty(byte ctrIndex, byte property, byte flags)

Gets a property of a key container.

ctrIndex: The container index to get property, from 0 to N (N = max number of containers supported by the minidriver, 15 in current version).

property: The container property to get:

- CONTAINER_INFO (0x00):

  Returns a byte array blob containing the public key(s) in the selected container.

  The returned blob is formatted as follows:

  Blob = [Signature_Pub_Key] || [Exchange_Pub_Key]

  Signature_Pub_Key and Exchange_Pub_Key are optional depending on which key exists in the container and it's a sequence of 3 TLV formatted as follows:

  T_Key_Type = 0x03

  L_Key_Type = 0x01

  V_Key_Type = 0x01 for Exchange_Pub_Key or 0x02 for Signature_Pub_Key

  T_Key_Pub_Exp = 0x01

  L_Key_Pub_Exp = 0x04

  V_Key_Pub_Exp = Value of Public key Exponent on 4 bytes.

  T_Key_Modulus = 0x02

  L_Key_Modulus = Key_Size_Bytes >> 4 (1 byte !)

  V_Key_Modulus = Value of Public key Modulus on Key_Size_Bytes bytes.

  The 4-bit shift on L_Key_Modulus is to be able to pass the Modulus length on 1 byte for values 64 to 256 (512 bits to 2048 bits).

- PIN_IDENTIFIER (0x01):

  Returns a byte array of one byte that indicates the role (User PIN or PIN#3 to PIN#7) that must be authenticated to use the keys in the container ctrIndex.

  This command execution is free.

- CONTAINER_TYPE (0x80):

  Returns a byte array of two bytes that indicates if the keys in the container ctrIndex were imported or generated on board.

  The returned blob is formatted as follows:

  blob[0] = 0x00 if signature key was imported, 0x01 if it was generated on board.

  blob[1] = 0x00 if exchange key was imported, 0x01 if it was generated on board.

  This command execution is free.

flags: RFU.

byte[] SetContainerProperty(byte ctrIndex, byte property, byte[] data, byte flags)

Sets a property of a key container.

ctrIndex: The container index to set property, from 0 to N (N = max number of containers supported by the minidriver, 15 in current version).

property: The container property to set:

- PIN_IDENTIFIER (0x01):

  Sets the role (User PIN or PIN#3 to PIN#7) that must be authenticated to use the keys in the container ctrIndex.

data: Must be a byte array of one byte that contains the role identifier to be associated with the container. Admin Key role is not allowed.

The User PIN role or the Admin Key role must authenticated to authorize execution of this method.

flags: RFU.

byte[] GetCardProperty(byte property, byte flags)

Gets a card property. This command execution is free.

property: The card property to get:

- CARD_FREE_SPACE (0x00):

  Returns a byte array blob of 12 bytes containing the free space information.

  The returned blob is formatted as follows:

  blob[0-3]: Free memory size (in bytes) on the card (big-endian).

  blob[4-7]: Number of free containers on the card (big-endian).

  blob[8-11]: Maximum number of containers on the card (big-endian).

- CARD_KEY_SIZES (0x02):

  Returns a byte array blob of 16 bytes containing the key sizes information.

  The returned blob is formatted as follows:

  blob[0-3]: Minimum key length (big-endian).

  blob[4-7]: Default key length (big-endian).

  blob[8-11]: Maximum key length (big-endian).

  blob[12-15]: Increment key length (big-endian).

- CARD_READ_ONLY (0x03):

  Returns a byte array blob of 1 byte that indicates if card is read-only or not.

  The returned blob is formatted as follows:

  blob[0] = Read-only mode:
  - 0x00 -> The card is read/write (default).
  - 0x01 -> The card is read-only.

- CARD_CACHE_MODE (0x04):

  Returns a byte array blob of 1 byte that indicates if the card cache mode.

  The returned blob is formatted as follows:

  blob[0] = Cache mode :
  - 0x01 -> Global cache mode (default).
  - 0x02 -> Session cache mode.
  - 0x03 -> No cache mode.

- CARD_GUID (0x05):

  Returns a byte array blob of 16 bytes that indicates the GUID (unique identifier) of the card. This is the same value as the 'cardid' file content.

  The returned blob is formatted as follows:

  blob[0-15] = 16 bytes GUID (unique identifier).

  By default the GUID is 0x2E4E4554 (4-byte fixed value || Card Serial Number (12 bytes)

- CARD_SERIAL_NUMBER (0x06):

  Returns a byte array blob of 12 bytes that indicates the Serial Number of the card. This is the unique chip serial number.

  The returned blob is formatted as follows:

  blob[0-11] = 12 bytes Serial Number (unique identifier).

- CARD_PIN_INFO (0x07):

  Returns a byte array blob of 12 bytes that indicates the PIN Information of a role.

  The ***flags*** parameter indicates the role identifier (User PIN, Admin Key, PIN#3, PIN#4, PIN#5, PIN#6 or PIN#7).

  The returned blob is formatted as follows:

  blob[0] = PIN Type:

  - 0x00 -> Normal Alphanumeric PIN (default for User PIN and PIN#3 to PIN#7).
  - 0x01 -> External PIN (used for Biometrics or Pinpad).
  - 0x02 -> Challenge/Response PIN (Default for Admin Key).
  - 0x03 -> No PIN (used for not protected keys).

  blob[1] = PIN Purpose:

  - 0x00 -> Authentication PIN.
  - 0x01 -> Digital Signature PIN.
  - 0x02 -> Encryption PIN.
  - 0x03 -> Non repudiation PIN.
  - 0x04 -> Admin PIN (default for Admin Key role).
  - 0x05 -> Primary PIN (default for User PIN and PIN#3 to PIN#7 roles).
  - 0x06 -> Unblock only.

  blob[2] = Bits mask of roles identifier that permits unblock of the PIN. Default is Admin Keys for all PIN roles.

  blob[3] = PIN Cache type, it's used by the Base CSP to manage PIN caching:

  - 0x00 -> Normal cache: the Base CSP maintains cache per application (default).
  - 0x01 -> Timed cache: the PIN is invalidated in the cache after an indicated period of time (value given in seconds).
  - 0x02 -> No Cache: the Base CSP submits PIN to card but does not maintain a cache. Subsequent operations must occur before Base CSP transaction time-out occurs.
  - 0x03 -> Always prompt: The Base CSP does not maintain a cache, but the transaction time-out does not apply. Instead it prompts the user for the PIN each time that it needs it.

  blob[4-7] = Time period (in seconds) if PIN cache type is a timed cache (big-endian).

  blob[8-11] = RFU.

- CARD_ROLES_LIST (0x08):

  Returns a byte array blob of 1 byte that indicates the roles supported by the card.

  The returned blob is formatted as follows:

  blob[0] = Bits mask of supported roles identifiers. 0x7F in current version -> All roles.

- CARD_AUTHENTICATED_ROLES (0x09):

  Returns a byte array blob of 1 byte that indicates the roles currently authenticated by the card.

  The returned blob is formatted as follows:

  blob[0] = Bits mask of currently authenticated roles identifiers.

- CARD_PIN_STRENGTH (0x0A):

  Returns a byte array blob of 1 byte that indicates the PIN strength of a role.

  The ***flags*** parameter must indicates the role identifier (User PIN, Admin Key, PIN#3, PIN#4, PIN#5, PIN#6 or PIN#7).

  The returned blob is formatted as follows:

  blob[0] = Bits mask of PIN strength of the role:

  - 0x01 -> Supports plaintext mode verification.
  - 0x02 -> Supports session PIN mode verification.

- CARD_X509_ENROLL (0x0D):

  Returns a byte array blob of 1 byte that indicates if the card supports X509 certificates enrollment/renewal.

  The returned blob is formatted as follows:

  blob[0] = X509 certificates mode:

  - 0x00 -> The card does not support X509 certificates enrollment.
  - 0x01 -> The card supports X509 certificates enrollment (default).

- CARD_UNBLOCK_FP_SYNC (0xF9):

  Linked to “-u” installation parameter. Returns a byte array blob of 1 byte that indicates if unblocking PIN authentication also unblocks FP authentication.

  The returned blob is formatted as follows:

  blob[0] = Unblocking PIN also unblocks FP:

  - 0x00 -> The card does not support the feature (default).
  - 0x01 -> The card supports the feature.

- CARD_PIN_POLICY (0x80):

  Returns a byte array blob of 14 bytes that indicates the PIN Policy currently sets on the card.

  The returned blob is formatted as follows:

  blob[0] = Maximum attempts before the PIN is blocked (1-255), default value is 5.

  blob[1] = Minimum PIN length (4-255), default value is 4.

  blob[2] = Maximum PIN length (4-255), default value is 255.

  blob[3] = Authorized char set(s). This is a bits mask with the following char sets:

  – 0x01 -> Numeric (0x30...0x39)
  – 0x02 -> Alphabetic uppercase (0x41...0x5A)
  – 0x04 -> Alphabetic lowercase (0x61...0x7A)
  – 0x08 -> Non alphanumeric (0x20...0x2F + 0x3A...0x40 + 0x5B...0x60 + 0x7B...0x7F)
  – 0x10 -> Non ASCII (0x80...0xFF)
  – 0x20 -> Alphabetic all (0x41...0x5A + 0x61...0x7A)

Note: If “alphabetic all” (0x20) is selected, 0x02 and 0x04 are automatically disabled.

  blob[4] = Number of different characters that can be repeated at least once (0-255), 255 if no limitation (default). This is also known as complexity rule 1.

  blob[5] = Maximum number of times a character can appear (1-255), 255 if no limitation (default). This is also known as complexity rule 2.

  blob[6] = Adjacent characters policy:

  – 0x00 -> Repeated characters can't be adjacent.
  – 0x01 -> Repeated characters can be adjacent (default).

  blob[7] = Number of previous PIN values a new PIN can't match (0-10), 0 if no history (default).

  blob[8] = Unblock policy:

  – 0x00 -> PIN unblock is not permitted.
  – 0x01 -> PIN unblock is permitted (default).

blob[9] = SSO policy:

- 0x00 -> PIN SSO is not activated (default).
- 0x01 -> PIN SSO is activated.

Note: The effect of activating SSO differs according to the Windows operating system.

Windows 7 and 8: If using the IDGo 800 CP, the user needs to present the user PIN once only during a session (such as logging on) as long as the IDPrime .NET card is not removed or reset. If the user is using the standard Microsoft CP, SSO is not supported for smart card logon (but is supported for other operations, such as digital signature).

Vista SP1 and later: The IDGo 800 CP is not available for Vista. If SSO is activated, the IDGo 800 CP can be used with the standard Microsoft CP but not for smart card logon.

XP and Vista (before SP1): SSO is not supported.

---

blob[10] = One character from each set usage policy:

- 0x00 -> Do not enforce PIN value composed with at least one character of each char set (default).
- 0x01 -> Enforce PIN value composed with at least one character of each char set.

blob[11] = Mandatory char set(s). This is a bits mask with the following char sets:

- 0x01 -> Numeric (0x30...0x39)
- 0x02 -> Alphabetic uppercase (0x41...0x5A)
- 0x04 -> Alphabetic lowercase (0x61...0x7A)
- 0x08 -> Non alphanumeric (0x20...0x2F + 0x3A...0x40 + 0x5B...0x60 + 0x7B...0x7F)
- 0x10 -> Non ASCII (0x80...0xFF)
- 0x20 -> Alphabetic all (0x41...0x5A + 0x61...0x7A)
- 0x1F -> All chars (0x20...0xFF) (default)

Note: If “alphabetic all” (0x20) is selected, 0x02 and 0x04 are automatically disabled.

---

blob[12] = Maximum length of character sequences e.g., 1,2,3,4 or a,b,c,d. For example, if set to 4, 1,2,3,4,a,5 is allowed, but 1,2,3,4,5,a is not allowed. Range is 1-255. Default is 255 (no limitation)

blob[13] = Maximum number of adjacent characters. Range is 1-255. Default is 255 (no limitation). This byte is ignored if adjacent characters policy is set to 00 (not allowed). This value cannot exceed the value of complexity rule 2.

- CARD_CHANGE_PIN_FIRST (0xFA):

  Returns a byte array blob of 1 byte that indicates the status of the “Force PIN change at first use” property.

  The ***flags*** parameter indicates the role identifier (User PIN, PIN#3, PIN#4, PIN#5, PIN#6 or PIN#7).

  The returned blob is formatted as follows:

data[0] = Change PIN at first use mode:

- 0x00 -> Feature is not activated.
- 0x01 -> Feature is activated.

- CARD_IMPORT_ALLOWED (0x90):

  Linked to “-k” installation parameter. Returns a byte array blob of 1 byte that indicates if RSA key injection is permitted or not.

  The returned blob is formatted as follows:

  blob[0] = Key injection allowed mode:

  - 0x00 -> The card does not support RSA key injection.
  - 0x01 -> The card supports RSA key injection (default).

- CARD_IMPORT_CHANGE_ALLOWED (0x91):

  Linked to “-l” installation parameter. Returns a byte array blob of 1 byte that indicates if the CARD_IMPORT_ALLOWED property can be changed or not.

  The returned blob is formatted as follows:

  blob[0] = CARD_IMPORT_ALLOWED change mode:

  - 0x00 -> CARD_IMPORT_ALLOWED property cannot be changed.
  - 0x01 -> CARD_IMPORT_ALLOWED property can be changed (default).

- CARD_DATA_EVERYONE (0xA0):

  Returns a persistent byte array blob of the data requested. The data blob can be of any length.

  The ***flags*** parameter must indicate the data identifier (0-255). If the data identifier doesn't exist an ArgumentException exception is thrown.

  The byte array is one that has been written using the **SetCardProperty(CARD_DATA_EVERYONE)** function. The command is unprotected.

- CARD_DATA_USER (0xA1):

  Returns a persistent byte array blob of the data requested. The data blob can be of any length.

  The ***flags*** parameter must indicate the data identifier (0-255). If the data identifier doesn't exist an ArgumentException exception is thrown.

  The byte array is one that has been written using the **SetCardProperty(CARD_DATA_USER)** function. The command is protected by the User PIN role.

- CARD_DATA_ADMIN (0xA2):

  Returns a persistent byte array blob of the data requested. The data blob can be of any length.

  The ***flags*** parameter must indicate the data identifier (0-255). If the data identifier doesn't exist an ArgumentException exception is thrown.

  The byte array is one that has been written using the **SetCardProperty(CARD_DATA_ADMIN)** function. The command is protected by the Admin Key role.

- CARD_PKI_OFF (0xF7):

  Returns a byte array blob of 1 byte that indicates if the PKI mode is disabled or not.

  The returned blob is formatted as follows:

  blob[0] = PKI Off mode:

  – 0x00 -> PKI Off is not activated (PKI is ON, default mode).
  – 0x01 -> PKI Off is activated.

  When PKI is off, the following limitations apply:

  + Not possible to generate or import a key container.
  + Not possible to delete a key container.
  + All containers are seen as empty and free.
  + PKI operations are not possible.
  + 'mscp' and 'p11' directories are seen as empty (no files in them).
  + Not possible to create 'mscp' and 'p11' directories.
  + Not possible to delete 'mscp' and 'p11' directories.
  + Not possible to create file in 'mscp' and 'p11' directories.
  + Not possible to delete file in 'mscp' and 'p11' directories.
  + 'cardcf' file is seen as empty (all bytes are 0x00).
  + Not possible to modify 'cardcf' file contents (write / delete).

- CARD_VERSION_INFO (0xFF):

  Returns a byte array blob of 4 bytes that indicates the exact version number of the assembly.

  The returned blob is formatted as follows:

  blob[0-3] = 4 bytes Version Number.

- CARD_SECURE_AM (0xFB):

  Linked to “-s” installation parameter. Returns a byte array blob of 1 byte that indicates if the Card Access Manager Admin key is different from the standard Admin Key.

  The returned blob is formatted as follows:

  blob[0] = Card Access Manager Admin Key different:

  – 0x00 -> The Card Access Manager Admin Key is the same as the standard Admin key (default).
  – 0x01 -> The Card Access Manager Admin Key is different from the standard Admin key (default).

byte[] SetCardProperty(byte property, byte[] data, byte flags)

Sets a card property.

The Admin Key role must be verified to authorize access to this command.

property: The card property to set:

- CARD_READ_ONLY (0x03):

  Sets the read-only mode of the card.

  data: Must be a byte array of one byte that contains the read-only mode.

  data[0] = Read-only mode:

  – 0x00 -> The card must be read/write.
  – 0x01 -> The card must be read-only.

- CARD_CACHE_MODE (0x04):

  Sets the card cache mode.

  data: Must be a byte array of one byte that contains the card cache mode:

  data[0] = Cache mode :

  – 0x01 -> Global cache mode (default).
  – 0x02 -> Session cache mode.
  – 0x03 -> No cache mode.

- CARD_GUID (0x05):

  Sets the card GUID. The 'cardid' file content is automatically synchronized when this property is changed.

  data: Must be a byte array blob of 16 bytes that indicates the GUID of the card.

  data[0-15] = 16 bytes GUID (unique identifier).

- CARD_SERIAL_NUMBER (0x06):

  Sets the card serial number.

  data: Must be a byte array blob of 12 bytes that indicates the card serial number.

  data[0-11] = 12-byte Serial Number (unique identifier).

- CARD_PIN_INFO (0x07):

  Sets the PIN Information of a role.

  The ***flags*** parameter indicates the role identifier (User PIN, Admin Key, PIN#3, PIN#4, PIN#5, PIN#6 or PIN#7).

  data: Must be a byte array blob of 12 bytes that indicates the PIN Information of the role.

  data[0] = PIN Type:

  - 0x00 -> Normal Alphanumeric PIN (default for User PIN and PIN#3 to PIN#7).
  - 0x01 -> External PIN (used for Biometrics or Pinpad).
  - 0x02 -> Challenge/Response PIN (Default for Admin Key).
  - 0x03 -> No PIN (used for not protected keys).

  data[1] = PIN Purpose:

  - 0x00 -> Authentication PIN.
  - 0x01 -> Digital Signature PIN.
  - 0x02 -> Encryption PIN.
  - 0x03 -> Non repudiation PIN.
  - 0x04 -> Admin PIN (default for Admin Key role).
  - 0x05 -> Primary PIN (default for User PIN and PIN#3 to PIN#7 roles).
  - 0x06 -> Unblock only.

  data[2] = Bits mask of roles identifier that permit unblock of the PIN. Default is Admin Keys for all PIN roles.

data[3] = PIN Cache type, it's used by the Base CSP to manage PIN caching:

- 0x00 -> Normal cache, the Base CSP maintains cache per application (default).
- 0x01 -> Timed cache: the PIN is invalidated in the cache after an indicated period of time (value given in seconds).
- 0x02 -> No Cache: the Base CSP submits PIN to card but does not maintain a cache. Subsequent operations must occur before Base CSP transaction time-out occurs.
- 0x03 -> Always prompt: The Base CSP does not maintain a cache, but the transaction time-out does not apply. Instead it prompts the user for the PIN each time that it needs it.

data[4-7] = Time period (in seconds) if PIN cache type is a timed cache (big-endian).

data[8-11] = RFU.

- CARD_X509_ENROLL (0x0D):

  Sets the support of X509 certificates enrollment/renewal property.

  data: Must be a byte array blob of 1 byte that indicates if the card supports X509 certificates enrollment/renewal.

  data[0] = X509 certificates mode:

  - 0x00 -> The card not supports X509 certificates enrollment.
  - 0x01 -> The card supports X509 certificates enrollment (default).

- CARD_UNBLOCK_FP_SYNC (0xF9):

  Linked to “-u” installation parameter. Receives a byte array blob of 1 byte that indicates if unblocking PIN authentication also unblocks FP authentication.

  The received blob is formatted as follows:

  blob[0] = Unblocking PIN also unblocks FP:

  - 0x00 -> The card does not support the feature (default).
  - 0x01 -> The card supports the feature.

- CARD_PIN_POLICY (0x80):

  Sets the PIN Information of a role.

  The ***flags*** parameter indicates the role identifier (User PIN, PIN#3, PIN#4, PIN#5, PIN#6 or PIN#7). If set to 0x00, the policy applies to all PINs.

  There are two ways of updating the PIN policy, described here as a) and b)

  a) data is null.

  To allow the replacement of the current PIN Policy on the card it's necessary to copy a text file named 'PinPolicy.txt' in the 'D:\Pub\' directory of the .NET card file system before executing this command. This file must contain the new PIN Policy to install on the card. The file is automatically erased after execution of this command.

  The 'PinPolicy.txt' file format must be as follows:

```
; PIN Policy description file for .NET MiniDriver
; This file must be located in D:\Pub\PinPolicy.txt
; For performance reasons it's recommended to remove comment lines before
copying file in the card.
; Comment lines begin with ';' character.

;MAX_ATTEMPTS range : 1...255
MAX_ATTEMPTS=5

;MIN_LENGTH range : 4...255
MIN_LENGTH=4

;MAX_LENGTH range : 4...255
MAX_LENGTH=255

;CHAR_SET values : Combination of
;01   : Numeric              = 0x30...0x39
;02   : Alphabetic uppercase = 0x41...0x5A
;04   : Alphabetic lowercase = 0x61...0x7A
;08   : Non alphanumeric     = 0x20...0x2F + 0x3A...0x40 + 0x5B...0x60 + ;
0x7B...0x7F
;10   : Non ASCII            = 0x80...0xFF
;20   : Alphabetic All = 0x41...0x5A + 0x61...0x7A

; If "Alphabetic All" charset is selected, "Alphabetic uppercase" and
; "Alphabetic lowercase" charsets are automatically disabled during PIN
; policy setting.

CHAR_SET=1F

;COMPLEXITY_RULE_1 value : Number of different characters that can be
repeated at least once (255 if no limitation)
COMPLEXITY_RULE_1=255

;COMPLEXITY_RULE_2 value : Max number of times a character can appear (255
if no limitation)
COMPLEXITY_RULE_2=255

;ADJACENT_ALLOWED value : YES or NO. If NO, the PIN must not contain
repeated chars in adjacent positions.
ADJACENT_ALLOWED=YES

;HISTORY value : Number of previous PIN values a new PIN cannot match
HISTORY=0
```

```
;ALLOW_UNBLOCK value : YES or NO. If NO, the PIN cannot be unblocked.
ALLOW_UNBLOCK=YES

;ALLOW_SSO value : YES or NO. If YES, the minidriver will implement a SSO
behavior (OS >= VISTA SP1 only).
ALLOW_SSO=NO

;ONE_OF_EACH_CHAR_SET value : YES or NO. If YES, the new PIN must contain at
least one character of each char set defined in CHAR_SET
ONE_OF_EACH_CHAR_SET=NO

;MANDATORY_CHAR_SET values : Combination of
;01   : Numeric                        = 0x30...0x39
;02   : Alphabetic uppercase           = 0x41...0x5A
;04   : Alphabetic lowercase           = 0x61...0x7A
;08   : Non alphanumeric               = 0x20...0x2F + 0x3A...0x40 +
0x5B...0x60 + 0x7B...0x7F
;10   : Non ASCII                      = 0x80...0xFF
;20   : Alphabetic All                 = 0x41...0x5A + 0x61...0x7A

; If "Alphabetic All" charset is selected, "Alphabetic uppercase" and
"Alphabetic lowercase" charsets are automatically disabled during PIN
policy setting.

; It's not allowed to set a value different from '00' if
ONE_OF_EACH_CHAR_SET is set to YES.

; It's not allowed to set a charset that is not defined in the CHAR_SET
rule.

MANDATORY_CHAR_SET=00

;MAX_SEQUENCE_LEN value : Maximum length of a sequence of characters.
; 255 if no limitation.
MAX_SEQUENCE_LEN=255

;MAX_ADJACENT_NB value : Maximum number of times a character can be
repeated in adjacent positions.
; 255 if no limitation.
; Ignored if ADJACENT_ALLOWED is set to NO.
; Can't have a value greater than COMPLEXITY_RULE_2.
MAX_ADJACENT_NB=255
```

b) In the second method, ***data*** is a byte array blob of 14 bytes that indicates the PIN Policy of the PIN role.

data[0] = MAX_ATTEMPTS

data[1] = MIN_LENGTH

data[2] = MAX_LENGTH

data[3] = CHAR_SET

data[4] = COMPLEXITY_RULE_1

data[5] = COMPLEXITY_RULE_2

data[6] = ADJACENT_ALLOWED

data[7] = HISTORY

data[8] = ALLOW_UNBLOCK

data[9] = ALLOW_SSO

Note: The effect of activating SSO differs according to the Windows operating system.

Windows 7 and 8: If using the IDGo 800 CP, the user needs to present the user PIN once only during a session (such as logging on) as long as the IDPrime .NET card is not removed or reset. If the user is using the standard Microsoft CP, SSO is not supported for smart card logon (but is supported for other operations, such as digital signature).

Vista SP1 and later: The IDGo 800 CP is not available for Vista. If SSO is activated, the IDGo 800 CP can be used with the standard Microsoft CP but not for smart card logon.

XP and Vista (before SP1): SSO is not supported.

---

data[10] = ONE_OF_EACH_CHAR_SET

data[11] = MANDATORY_CHAR_SET

data[12] = MAX_SEQUENCE_LEN

data[13] = MAX_ADJACENT_NB

NOTES About PIN Policies:

- All PIN policy parameters are optional in the file and if a value is not present the default value applies.
- When a PIN policy is set, the card checks the consistency of parameters. The PIN policy update fails if the file contains inconsistent parameters.
- If a PIN Policy file is present on the card before Minidriver assembly installation, the PIN policy description is applied to ALL PINs during installation.
- If SetCardProperty(0x80, data, flags) is called with ‘flags’ parameter set to 0x00, the new PIN policy is applied to all PINs.
- If SetCardPoperty(0x80, data, flags) is called with ‘flags’ parameter set to 0x01, 0x03, 0x04, 0x05, 0x06 or 0x07, the new PIN policy is applied to the PIN role whose ID corresponds to the ‘flags’ parameter value. No other values are supported.
- If SetCardPoperty(0x80, data, flags) is called with ‘data’ length = 0, the card tries to apply a PIN policy from “D:\Pub\PinPolicy.txt” file.
- If SetCardPoperty(0x80, data, flags) is called with ‘data’ length != 0, the card tries to apply a PIN policy from the ‘data’ passed in the parameters.
- If GetCardProperty(0x80, flags) is called with ‘flags’ parameter set to 0x00, the card returns the PIN policy of default User PIN (Id = 0x01).
- If GetCardPoperty(0x80, flags) is called with ‘flags’ parameter set to 0x01, 0x03, 0x04, 0x05, 0x06 or 0x07, the card returns the PIN policy of the PIN role whose ID corresponds to the ‘flags’ parameter value. No other values are supported.

- CARD_CHANGE_PIN_FIRST (0xFA):

  Sets the “Force PIN change at first use” property.

  The ***flags*** parameter indicates the role identifier (User PIN, PIN#3, PIN#4, PIN#5, PIN#6 or PIN#7).

  data: Must be a one-byte value that indicates if the feature is activated or not.

  data[0] = Change PIN at first use mode:

  – 0x00 -> Feature is not activated.

  – 0x01 -> Feature is activated.

- CARD_PIN_CHECK (0x81):

  This is not really a SetCardProperty command but a hook to test if a PIN is valid regarding the PIN Policy for the specified role. An exception is thrown if the PIN is not compliant with the PIN Policy:

Note: For this specific “property”, Admin Key role authentication is not required.

The ***flags*** parameter indicates the role identifier (User PIN, PIN#3, PIN#4, PIN#5, PIN#6 or PIN#7).

data: The PIN value to be checked.

```
case PinPolicy.ERR_PIN_TOO_SHORT:
   throw new Exception("0xFFFF0001");

case PinPolicy.ERR_PIN_TOO_LONG:
   throw new Exception("0xFFFF0002");

case PinPolicy.ERR_PIN_CHARSET:
   throw new Exception("0xFFFF0003");

case PinPolicy.ERR_PIN_MANDATORY_CHARSET:
   throw new Exception("0xFFFF0004");

case PinPolicy.ERR_PIN_COMPLEXITY_1:
   throw new Exception("0xFFFF0005");

case PinPolicy.ERR_PIN_COMPLEXITY_2:
   throw new Exception("0xFFFF0006");

case PinPolicy.ERR_PIN_ADJACENT:
   throw new Exception("0xFFFF0007");

case PinPolicy.ERR_PIN_HISTORY:
   throw new Exception("0xFFFF0008");

case PinPolicy.ERR_PIN_MAX_SEQUENCE:
   throw new Exception("0xFFFF0009");

case PinPolicy.ERR_PIN_MAX_ADJACENT:
   throw new Exception("0xFFFF000A");
```

- CARD_IMPORT_ALLOWED (0x90):

Linked to “-k” installation parameter. Receives a byte array blob of 1 byte that indicates if RSA key injection is permitted or not.

The received blob is formatted as follows:

blob[0] = Key injection allowed mode:

0x00 -> The card does not support RSA key injection.

0x01 -> The card supports RSA key injection (default).

- CARD_IMPORT_CHANGE_ALLOWED (0x91):

Linked to “-l” installation parameter. Receives a byte array blob of 1 byte that indicates if the CARD_IMPORT_ALLOWED property can be changed or not.

The received blob is formatted as follows:

blob[0] = CARD_IMPORT_ALLOWED change mode:

0x00 -> CARD_IMPORT_ALLOWED property cannot be changed.

0x01 -> CARD_IMPORT_ALLOWED property can be changed (default).

- CARD_DATA_EVERYONE (0xA0):

  Sets a persistent byte array blob of data. The data blob can be of any length and is written “as is”.

  The ***flags*** parameter must indicate the data identifier (0-255). If the data identifier already exists, it is overwritten with the new data.

  The command is unprotected.

- CARD_DATA_USER (0xA1):

  Sets a persistent byte array blob of data. The data blob can be of any length and is written “as is”.

  The ***flags*** parameter must indicate the data identifier (0-255). If the data identifier already exists, it is overwritten with the new data.

  The command is protected by the User PIN role.

- CARD_DATA_ADMIN (0xA2):

  Sets a persistent byte array blob of data. The data blob can be of any length and is written “as is”.

  The ***flags*** parameter must indicate the data identifier (0-255). If the data identifier already exists, it is overwritten with the new data.

  The command is protected by the Admin Key role.

- CARD_PKI_OFF (0xF7):

  Sets a byte array blob of 1 byte that indicates if the PKI mode is disabled or not.

  The blob is formatted as follows:

  blob[0] = PKI Off mode:

  – 0x00 -> PKI Off is not activated (PKI is ON, default mode).
  – 0x01 -> PKI Off is activated.

  When PKI is off, the following limitations apply:

  + Not possible to generate or import a key container.
  + Not possible to delete a key container.
  + All containers are seen as empty and free.
  + PKI operations are not possible.
  + 'mscp' and 'p11' directories are seen as empty (no files in them).
  + Not possible to create 'mscp' and 'p11' directories.
  + Not possible to delete 'mscp' and 'p11' directories.
  + Not possible to create file in 'mscp' and 'p11' directories.
  + Not possible to delete file in 'mscp' and 'p11' directories.
  + 'cardcf' file is seen as empty (all bytes are 0x00).
  + Not possible to modify 'cardcf' file contents (write / delete).

- CARD_ZAP_DATA (0xF8):

  Destroys (zaps) set(s) of persistent data in the card.

  The data is a byte array blob of one byte that represents a bit mask of the data to be destroyed. The blob is formatted as follows:

  blob[0] = Bit mask of data to be destroyed:

  – 0x01 -> ZD_FILE_SYSTEM: Zap file system by restoring original Base CSP file system.
  – 0x02 -> ZD_CONTAINER: Zap contents of all containers so they become empty.
  – 0x04 -> ZD_PIN: Zap all PINs by restoring their initial values, properties and PIN policies.
  – 0x08 -> ZD_ADMIN: Zap Minidriver Admin key and Card Admin key if it exists; original values are restored.
  – 0x10 -> ZD_PROPERTY: Zap all MS standard data that can be modified through properties; original values are restored.
  – 0x20 -> ZD_BIO: Zap unblock FP on unblock PIN capabilty, original value is restored.
  – 0x40 -> ZD_DATA_STORAGE: Zap the “Everyone”, “User” and “Admin” storage areas, i.e. those that can be filled using the CARD_DATA_EVERYONE, CARD_DATA_USER and CARD_DATA_ADMIN properties. All storage areas are cleared.
  – 0x80 -> ZD_CUSTOM: Zap custom data.
  – 0xFF -> All data sets.

Part III

APDU Encoding

13

APDU Encoding

Introduction

The services in .NET cards expose methods available to be called from external applications. At the current stage of .NET card development APDUs provide the means of transportation for these method calls. Each method call is encoded with its arguments into one or more APDUs before being sent to the smart card. Part 3 of this Integration Guide describes how to encode APDUs for a service method call.

Each service in a .NET card has a specific name (URI), an associated type (typeof) and it listens on a specific port. Each method of a service has a specific name. The arguments or parameters of the method are formed using the argument values.

All the above information is then encoded to form APDUs for a method call.

APDU Format

Here is the general structure of the APDU encoding for a method call:

APDU = **APDU Header** + **APDU Payload**

- **APDU Header** = 80C20000 [APDU Payload length (1 byte)]
- **APDU Payload** = [D8] [Service port number (2 bytes)] [6F] [Service Namespace Hivecode (4 bytes)] [Service Type Hivecode (2 bytes]] [Method Hivecode (2 bytes)] [Length of service name (2 bytes]] [UTF8 encoded Service name] [Encoded Arguments]

Note: Information on how to compute the hivecodes is given in “Chapter 14 - Hivecodes and Examples”.

For example, the card module service in a .NET card is named as “MSCM”, its port is “0005”, its namespace hivecode is “00C04B4E” and its type hivecode is “7FBD”.

So the APDU for the method GetChallenge() which doesn't require any argument and whose method hivecode is “FA3B” will be:

APDU = [80C20000 13] [D8] [Port (0005)] [6F] [namespace hivecode (00C04B4E)] [type hivecode (7FBD)] [method hivecode (FA3B)] [service name length (0004)] [service name - MSCM (4D53434D)]

Or

APDU = 80C20000 13 D8 0005 6F 00C04B4E 7FBD FA3B 0004 4D53434D

Argument Encoding

Arguments are formed by concatenation of the encoded argument values. Each argument value is encoded accordingly to its type as follows:

- The value of a *boolean* or a *byte* is encoded in one byte.
- The value of a *character* or an unsigned short is encoded in 2 bytes. The value of an integer is encoded in 4 bytes. A long is encoded in 8 bytes. A string is formed by concatenating the length of its UTF8 representation, as 2 bytes, followed by its UTF8 representation. An array is formed by concatenating its length (number of elements) in 4 bytes and then each element value encoded according to the element type. For example, an array of 2 ushort [1, 2] will be encoded as: “00000002 0001 0002" because ushort values are encoded in 2 bytes.

Note: “null” references for arrays are encoded using a length of “FFFFFFFF”. “null” reference for strings are encoded using a UTF8 representation length of “FFFF”.

Table 1 shows the possible types and their encoding convention in .NET cards:

Table 15 - Argument Types and Their Encoding

Argument types	Meaning (.NET)	Encoding
MARSHALLER_TYPE_IN_BOOL	boolean	1 byte - true : 01 - false: 00
MARSHALLER_TYPE_IN_S1	sbyte	1 byte
MARSHALLER_TYPE_IN_U1	byte	1 byte
MARSHALLER_TYPE_IN_CHAR	char	2 bytes
MARSHALLER_TYPE_IN_S2	short	2 bytes
MARSHALLER_TYPE_IN_U2	ushort	2 bytes
MARSHALLER_TYPE_IN_S4	Int32 (int)	4 bytes
MARSHALLER_TYPE_IN_U4	UInt32 (uint)	4 bytes
MARSHALLER_TYPE_IN_S8	Int64 (long)	8 bytes
MARSHALLER_TYPE_IN_U8	UInt64 (ulong)	8 bytes
MARSHALLER_TYPE_IN_STRING	string	[length (2 bytes)] + UTF8 encoded string - for null string, length is FFFF
MARSHALLER_TYPE_IN_BOOLARRAY	Array of boolean	[length (4 bytes)] + [values (1 byte each)]
MARSHALLER_TYPE_IN_S1ARRAY	Array of sbyte	[length (4 bytes)] + [values (1 byte each)]
MARSHALLER_TYPE_IN_U1ARRAY	Array of byte	[length (4 bytes)] + [values (1 byte each)]
MARSHALLER_TYPE_IN_MEMORYSTREAM	Memorystream - the same as Array of byte	[length (4 bytes)] + [values (1 byte each)]
MARSHALLER_TYPE_IN_CHARARRAY	Array of chars	[length (4 bytes)] + [values (2 bytes each)]

Table 15 - Argument Types and Their Encoding (continued)

Argument types	Meaning (.NET)	Encoding
MARSHALLER_TYPE_IN_S2ARRAY	Array of short	[length (4 bytes)] + [values (2 bytes each)]
MARSHALLER_TYPE_IN_U2ARRAY	Array of ushort	[length (4 bytes)] + [values (2 bytes each)]
MARSHALLER_TYPE_IN_S4ARRAY	Array of Int32	[length (4 bytes)] + [values (4 bytes each)]
MARSHALLER_TYPE_IN_U4ARRAY	Array of UInt32	[length (4 bytes)] + [values (4 bytes each)]
MARSHALLER_TYPE_IN_STRINGARRAY	Array of string	[length (4 bytes)] + [each element is encoded as a string as defined in MARSHALLER_TYPE_IN_STRING]
Null value	null	- FFFF for string - FFFFFFFF for other array type

Payload with length > FF

After forming the method call, if the APDU Payload is longer than the maximum conventional APDU payload length (FF - ISO7816-4 short APDU) then the payload is sent in sections (=< FF) to the smart card, and the smart card internally reconstructs the original payload.

Each section is used to form a new “sub”-payload. The sub-payload is formatted as follows:

Payload of each new APDU for the first section =

D8 FFFF [total length (4 bytes)] [section length (4 bytes)] [section data]

Payload of each new APDU for a following section =

D8 FFFF [offset (4 bytes)] [section length (4 bytes)] [section data]

For example if we have the following original payload (after section 1 formatting) with total length of W bytes (>FF) - where X + Y + Z = W

X bytes	Y bytes	Z bytes

We will divide W into 3 sections, such that X, Y and Z are all <= 244 bytes.

The payload of the first section (length X) is:

Payload-1 = D8 FFFF || [W (4 bytes)] || [X length (4 bytes)] || [X data bytes]

The payload of the second section (length Y) is:

Payload-2 = D8 FFFF || [Offset = X length (4 bytes)] || [Y length (4 bytes)] || [Y data bytes]

(The offset is marks the start of the second section and is therefore the length of X)

The payload of the third section (length z) will be:

Payload-3 = D8 FFFF || [Offset = X length + Y length (4 bytes)] || [Z length (4 bytes)] || [Z data bytes]

(The offset is marks the start of the third section and is therefore the length of X + the length of Y)

The total payload of the first section (Payload-1) is:

1 (D8) + **2** (FFFF) + **4** (W) + **4** (X) + **X** <= FF.

i.e. X <= (FF - 1 + 2 + 4 + 4), or X <= F4.

Y and Z must respect the same formula.

The APDU headers for these sub-sections are identical:

80C20000 [Payload-1 length in 1 byte] [Payload-1]

80C20000 [Payload-2 length in 1 byte] [Payload-2]

80C20000 [Payload-3 length in 1 byte] [Payload-3]

MSCM Answer Interpretation

For compatibility reasons, the Microsoft Card Module service (MSCM) differs from other services when it comes to the card answer protocol. This section details interactions with the MSCM service only. For interactions with other services (Content Manager, OATH, etc…) refer to the “Generic Answer Formation for .NET Card Services (Except MSCM)” on page 141.

The MSCM service formats the answer for each method call as follows:

[Namespace hivecode of exception or return type (4 bytes)] || [Type hivecode of exception or return type (2 bytes)] || [Data]

Where:

If exception:

Data = [optional UTF8 encoded exception message]

If no exception:

Data = [encoded return value]

In “Chapter 14 - Hivecodes and Examples” you will find all the namespace hivecodes and type hivecodes for all MSCM exception and return types.

The APDUs Exchange Flow

The external application communicates with the service inside the card using the following procedure:

1 The external application sends an APDU representing a method call to a certain service in the card

2 The card responses with a status word. This status word may be:

- 0x9000: good - no data returned
- 0x61xx: there are xx bytes of data to be returned by the card.

3 In case of 61xx - the external application needs to send a **GetResponse** APDU command: 00 C0 00 00 xx to the card to retrieve the data. If the card answers with the 61xx status word, the application needs to repeat the **GetResponse** APDU and concatenate the obtained data with the previous ones. The process is repeated until a 9000 status word is received.

4 The data may represent an exception or a nominal method call return answer.

14

Hivecodes and Examples

Generic Answer Formation for .NET Card Services (Except MSCM)

The answer for each method call to a service is formed according to the following convention:

Answer = [Status (1 byte)] || [Data]

Where:

Status = 01 if everything is ok - method calls successful

= FF if there is an exception

- If exception

  Data = [Namespace hivecode of the exception] [Type hivecode of exception] [optional UTF8 encoded exception message]

- If no exception (all ok)

  Data = [encoded return value] || [encoded reference parameters values]*

In the case of “no exception”, the return and reference parameter values are encoded according to their argument types as given in “Table 15” on page 138. The order is:

1. the method return value
2. the ref/out parameters in the order of declaration of the method prototype.

Computing Hivecodes

Hivecodes are computed by applying an MD5 hash to a particular string.

Namespace Hivecode

Namespace string consists of:

[Public Key Token] || ["."] || [Type namespace].

For example, the Namespace string for a namespace called “Company.Stuff” of an assembly having a public key token being 42AA56EAF12382CE would be:

"42AA56EAF12382CE.Company.Stuff".

The corresponding hivecode would be the last 4 bytes of MD5(“42AA56EAF12382CE.Company.Stuff”).

Type Hivecode

Type string consists of:

[Type Name].

For example, the Type string for a type called Company.Stuff.MyClass would be: “MyClass”.

The corresponding hivecode would be the last 2 bytes of MD5(“MyClass”).

Note: If referring to an array of type, the array information should be ignored, i.e. “MyClass[]” would become “MyClass”

Method Hivecode

Method string consists of:

[Fully qualified Method Name].

For example, the Method string for a method called “void Foo(byte b, ref int I, out short[] sarray)" would be

“System.Void Foo(System.Byte,System.Int32,System.Int16[])”.

The corresponding hivecode would be the last 2 bytes of

MD5(“System.Void Foo(System.Byte,System.Int32,System.Int16[])”).

MSCM Method Hivecodes

Note: Please refer to the minidriver standard document of Microsoft for a description of the method specification.

The card access role is a byte (1 byte) with the following values:

- USER = 0x01
- ADMIN = 0x02
- EVERYONE = 0x03

The access condition list (ACL) is a 3-byte array (3 bytes) that determines the access rights for the three roles in the following format:

Admin rights || User rights || Everyone rights.

Each byte in the array codes the associated rights for the corresponding card access role. It is encoded using the following flags (which can be OR-ed):

- EXECUTE = 0x01
- WRITE = 0x02
- READ = 0x04

For example: ACLs = {0x06, 0x06, 0x04} represents:

Admin rights = Read/Write, User rights = Read/Write, and Everyone’s rights = Read.

The Hivecodes differ according to the version of the Microsoft specification. The hivecodes for the methods in V5 of the Microsoft specification (described in “Chapter 11 - Minidriver V5 Methods”) are given in “Table 16”.

Table 16 - Hivecodes for V5

Method info (name, return type, arguments)	**Method Hivecode**	**Meaning**
byte[] GetChallenge()	FA3B	Cf. Microsoft minidriver standard document
void ExternalAuthenticate(byte[] response)	24FE	Response: cryptogram generated from challenge using Admin key.
void ExternalAuthenticateAM(byte[] response)	7E25	Response: cryptogram generated from challenge using Access Manager Admin Key.
void ChangeReferenceData(byte mode, byte role, byte[] oldPin, byte[] newPin, int maxTries)	E08A	To change the user PIN, Admin Key or Access Manager Admin Key. To unblock the user PIN. ■ mode: - CHANGE=0 - UNBLOCK=1 ■ role: - USER=1 - ADMIN=2 - ACCESS MANAGER ADMIN KEY = 128 (0x80) ■ oldPin: - the old value (CHANGE) - the cryptogram (UNBLOCK) ■ newPin: - the new value (CHANGE) - the new value (UNBLOCK) ■ maxTries: maximum retries. Use -1 to use the PINs default max retries value.
void VerifyPin(byte role, byte[] oldPin)	506B	
int GetTriesRemaining(byte role)	6D08	
void CreateCAPIContainer(byte ctrIndex, bool keyImport, byte keySpec, int keySize, byte[] keyValue)	0234	
void DeleteCAPIContainer(byte ctrIndex)	F152	
byte[] GetCAPIContainer(byte ctrIndex)	9B2E	
byte[] PrivateKeyDecrypt(byte ctrIndex, byte keyType, byte[] encryptedData)	6144	
int[] QueryFreeSpace()	00E5	
int[] QueryKeySizes()	5EE4	
void CreateFile(string path, byte[] acls, int initialSize)	BEF1	
void CreateDirectory(string path, byte[] acls)	ACE9	

Table 16 - Hivecodes for V5 (continued)

Method info (name, return type, arguments)	Method Hivecode	Meaning
void WriteFile(string path, byte[] data)	F20E	
byte[] ReadFile(string path, int maxBytesToRead)	744C	
void DeleteFile(string path)	6E2B	
void DeleteDirectory(string path)	9135	
string[] GetFiles(string path)	E72B	
byte[] GetFileProperties(string path)	A01B	
void LogOut(byte role)	C4E4	
bool IsAuthenticated(byte role)	9B0B	
byte MaxPinRetryCounter {get;}	FEAB	
bool AdminPersonalized {get;}	CFBE	
bool UserPersonalized {get;}	E710	
string get_Version()	DEEC	

The hivecodes for the methods in V6 and V7 of the Microsoft specification (described in “Chapter 12 - Minidriver V6/V7 Methods”) are given in “Table 17”.

Table 17 - Hivecodes for V6/V7

Method info (name, return type, arguments)	Method Hivecode	Meaning
byte[] GetChallengeEx(byte role)	8F0B	
byte[] AuthenticateEx(byte mode, byte role, byte[] pin)	5177	
void DeauthenticateEx(byte roles)	BD7B	
void ChangeAuthenticatorEx(byte mode, byte oldRole, byte[] oldPin, byte newRole, byte[] newPin, int maxTries)	9967	
byte[] GetContainerProperty(byte ctrIndex, byte property, byte flags)	279C	
byte[] SetContainerProperty(byte ctrIndex, byte property, byte[] data, byte flags)	98D1	
byte[] GetCardProperty(byte property, byte flags)	8187	
byte[] SetCardProperty(byte property, byte[] data, byte flags)	B0E4	
public void ForceGarbageCollector ( )	3D38	Forces garbage collector to start.

Namespace Hivecodes

System = 00D25D1C

System.IO = 00D5E6DB

System.Runtime.Remoting.Channels = 0000886E

System.Runtime.Remoting = 00EB3DD9
System.Security.Cryptography = 00ACF53B
System.Collections = 00C5A010
System.Runtime.Remoting.Contexts = 001F4994
System.Security = 00964145
System.Reflection = 0008750F
System.Runtime.remoting.Messaging = 00DEB940
System.Diagnostics = 0097995F
System.Runtime.CompilerServices = 00F63E11
System.Text = 00702756
SmartCard = 00F5EFBF

Standard Type Hivecodes

System.Void = CE81
System.Int32 = 61C0
System.Int32[] = 61C1
System.Boolean = 2227
System.Boolean[] = 2228
System.SByte = 767E
System.SByte[] = 767F
System.UInt16 = D98B
System.UInt16[] = D98C
System.UInt32 = 95E7
System.UInt32[] = 95E8
System.Byte = 45A2
System.Byte[] = 45A3
System.Char = 958E
System.Char[] = 958F
System.Int16 = BC39
System.Int16[] = BC3A
System.String = 1127
System.String[] = 1128
System.Int64 = DEFB
System.Int64[] = DEFC
System.UInt64 = 71AF
System.UInt64[] = 71B0
System.IO.MemoryStream = FED7

Exception Type Hivecodes

System.Exception = D4B0

System.SystemException	= 28AC
System.OutOfMemoryException	= E14E
System.ArgumentException	= AB8C
System.ArgumentNullException	= 2138
System.NullReferenceException	= C5B8
System.ArgumentOutOfRangeException	= 6B11
System.NotSupportedException	= AA74
System.InvalidCastException	= D24F
System.InvalidOperationException	= FAB4
System.NotImplementedException	= 3CE5
System.ObjectDisposed Exception	= 0FAC
System.UnauthorizedAccessException	= 4697
System.IndexOutOfRangeException	= BF1D
System.FormatException	= F3BF
System.ArithmeticException	= 6683
System.OverflowException	= 20A0
System.BadImageFormatException	= 530A
System.ApplicationException	= B1EA
System.ArrayTypeMismatchException	= 3F88
System.DivideByZeroException	= DFCF
System.MemberAccessException	= F5F3
System.MissingMemberException	= 20BB
System.MissingFieldException	= 7366
System.MissingMethodException	= 905B
System.RankException	= B2AE
System.StackOverflowException	= 0844
System.TypeLoadException	= 048E
System.IO.IOException	= 3BBE
System.IO.DirectoryNotFoundException	= 975A
System.IO.FileNotFoundException	= 07EB
System.Runtime.Remoting.RemotingException	= D52A
System.Security.Cryptography.CryptographicException	= 8FEB

Other Useful Type Hivecodes

SmartCard.ContentManager	= B18C

Other Useful Method Hivecodes

SmartCard.ContentManager.GetAssociatedPort	= 7616

APDUs Exchange Examples

This section provides some examples of the APDU exchanges with the MSCM service.

The important information for the MSCM service is:

- Service name is “MSCM”,
- Port is “0005”,
- Namespace hivecode is “00C04B4E”
- Type hivecode is “7FBD”.

A typical scenario is to authenticate the Admin role as follows:

1 Issue a **Get Challenge** APDU to generate an 8-byte random number

2 Issue a **Get Response** APDU to retrieve the 8-byte challenge

3 Issue an **External Authenticate** APDU to authenticate the Admin role (the 8-byte challenge is used as part of this command).

4 Issue a **Log Out** APDU command to log out as the Admin role.

Here are example commands for this scenario:

Get Challenge

The APDU for the method **byte[] GetChallenge()** which doesn't require any argument and whose method hivecode is “FA3B” is:

APDU = [80C20000 13] [D8] [Port (0005)] [6F] [namespace hivecode (00C04B4E)] [type hivecode (7FBD)] [method hivecode (FA3B)] [service name length (0004)] [service name - MSCM (4D53434D)]

Or

APDU = 80C20000 12 D8 0005 6F 00C04B4E 7FBD FA3B 0004 4D53434D

The status word for this APDU is 6112

Get Response

The APDU command to get the data is:

00C0000012

The on-card MSCM service answers with:

00D25D1C 45A3 00000008D90B49AA6690E797

Analysis of the Response

00D25D1C is the namespace hivecode => System

45A3 is the type hivecode => System.Byte[].

This means that the **GetChallenge()** method successfully executed and it returned a byte array.

The “Argument encoding” section (page 138) shows that a byte array is encoded as follows:

[array length (4 bytes)][values of each byte]

So the length of the returned byte array is: 00000008 or 8 bytes.

The remaining data (D90B49AA6690E797) are the 8 bytes of the card challenge.

External Authenticate

After calculating the card response (for example, the cryptogram value BC287ED3692474A9) we need to use the method **void ExternalAuthenticate(byte[] response)** to send the cryptogram to the card in order to authenticate the Admin role.

The **ExternalAuthenticate()** method hivecode is 24FE and we need to encode the response parameter or cryptogram (BC287ED3692474A9) as a byte array of length 8.

So the APDU for the **ExternalAuthenticate()** method is:

APDU = 80C20000 1E D8 0005 6F 00C04B4E 7FBD 24FE 0004 4D53434D 00000008 BC287ED3692474A9

If the card accepts the cryptogram then it returns 9000 (No data is expected as the method returns “void”) else it returns 61xx and an exception is encoded in the answer.

Log Out

If we want to log-out the Admin role, we need to execute the **LogOut()** method.

The **void LogOut(byte role)** method hivecode is C4E4 and since we want to log-out the Admin role, the parameter value is 02. The resulting APDU is:

80C20000 13 D8 0005 6F 00C04B4E 7FBD C4E4 0004 4D53434D 02

Since the method returns “void”, no data is expected, hence a proper execution would return directly with the status word 9000.

Part IV

Configuring Parameters

15

Configuring Parameters

Introduction

There are two main ways of configuring parameters in Gemalto's .NET cards.

- From V6 of the Microsoft specification, certain parameters can be configured in the card by using the **SetCardProperty** method, defined on page 127. Similarly, the values of certain parameters can be read using the **GetCardProperty** method, defined on page 119.
- Certain parameters can be configured by loading the card module assembly and executing it by specifying the parameters you want to change from their defaults.

This chapter lists the parameters that can be configured using either or both of these two ways.

Note: It is also possible to configure the Maximum Communication Speed and the Chip Speed by using the Card Explorer. This is explained in "Configuring the Communication and Chip Speed" on page 61.

Configurable Parameters (.NET Minidriver Assembly)

Using SetCardProperty

The following table lists the properties defined in the Microsoft specification that can be configured in the card module assembly using the **SetCardProperty** method along with their default values and other information.

Note: Those that are marked as READ ONLY cannot be modified (you cannot use **SetCardProperty**) but their values can be retrieved using the **GetCardProperty** method.

Table 18 - Microsoft Defined Minidriver Parameters

Parameter	Possible values	Default value	Comments
CARD_FREE_SPACE (12 bytes) READ ONLY			
Size of Free Memory in bytes (4 bytes)	Any	N/A	
Number of free containers in the card (4 bytes)	Any	N/A	

Table 18 - Microsoft Defined Minidriver Parameters (continued)

Parameter	Possible values	Default value	Comments
Maximum number of containers allowed in the card (4 bytes)	Any	N/A	
CARD_KEY_SIZES (16 bytes) READ ONLY			
Minimum Key Length (in bits) (4 bytes)	Any	512 bits	
Default Key Length (in bits) (4 bytes)	Any	1,024 bits	
Maximum Key Length (in bits) (4 bytes)	Any	2, 048 bits	
Increment Key Length (in bits) (4 bytes)	Any	256 bits	
CARD_READ_ONLY (1 byte)			
	Read/Write (R/W) Read Only (RO)	R/W	Determines if card is read-only or not
CARD_CACHE_MODE (1 byte)			
	Global cache Session cache No cache	Global	Indicates the cache mode
CARD_GUID (16 bytes)			
	Any	None	Unique Identifier
CARD_SERIAL_NUMBER (12 bytes)			
	Any	None	Unique Identifier Card serial number = chip serial number
CARD_PIN_INFO (12 bytes)			
PIN Type (1 byte)	■ Normal Alphanumeric PIN ■ External PIN (for Biometrics or PinPad). ■ Challenge/Response PIN ■ No PIN (used for unprotected keys)	■ Normal for User PIN and PINs #3 to #7 ■ Chal/Resp for Admin Key	If the IDPrime .NET card is configured as External PIN, the IDGo 500 PKCS#11 libraries allows only the use of a PIN pad reader for verifying the PIN, and not a standard PC/SC reader without keypad
PIN Purpose (1 byte)	■ Authentication ■ Digital Signature ■ Encryption ■ Non repudiation ■ Admin ■ Primary ■ Unblock only	■ Admin for Admin Key ■ Primary for others	
Bit Mask (roles identifier) (1 byte)	Any defined in CARD_ROLES_LIST	Admin Key	Defines roles that can unblock the PIN

Table 18 - Microsoft Defined Minidriver Parameters (continued)

Parameter	Possible values	Default value	Comments
PIN cache type (1 byte)	■ Normal, one cache per application ■ Timed cache, Base CSP empties its cache after a time period ■ No Cache ■ Always prompt	Normal	Cache is maintained by the Base CSP This parameter is not managed if the SSO parameter is set to **Yes**. It is compliant with Base CSP / MD v6 (Vista) and above. For more detials, please see page 121.
Time Period (in seconds) (4 bytes)	Any	N/A	Used if PIN cache type is timed cache Compliant with Base CSP / MD v6 (Vista) and above.
RFU (4 bytes)			
CARD_ROLES_LIST (1 byte) READ ONLY			
Bit Mask (roles identifier) (1 byte)	Any combination	0x7F - All Roles	These are the roles supported by the card. They are User PIN, Admin Key, PIN#3 — PIN#7)
CARD_AUTHENTICATED_ROLES (1 byte) READ ONLY			
Bit Mask (roles identifier) (1 byte)	Any combination	N/A	Stores the roles currently authenticated by the card
CARD_PIN_STRENGTH (1 byte) READ ONLY			
Bit Mask of PIN strengths of role	Supports plaintext mode verification Supports session PIN mode verification	Both modes are set	
CARD_X509_ENROLL (1 byte)			
	Does not support X509 Certificates enrollment Supports X509 Certificates enrollment	X509 supported	

PIN Policy and Other Gemalto Proprietary Parameters

In addition, Gemalto has defined a proprietary parameter for the PIN policy, which can use these same **SetCardProperty** and **GetCardProperty** methods.

If you have the IDGo 800 CP, your IDPrime .NET cards can support up to 7 PIN roles (the Admin Key role and 6 User PIN roles). Each of the 6 User PIN roles has its own PIN with its own PIN Policy. The roles are defined in the Microsoft Minidriver spec (the CARD_ROLES_LIST parameter in “Table 18”.

The following table shows the information for the proprietary PIN Policy and other Gemalto proprietary properties.

Table 19 - Gemalto Proprietary Parameters

Parameter	Possible values	Default value	Comments
CARD_PIN_POLICY (14 bytes)			
Max # attempts PIN verification (1 byte)	1-16	5	The maximum number of attempts allowed for **Unblock / Change PIN** commands. The PIN's **max attempts counter** is initialized to this value and decremented with each incorrect attempt. It is blocked when the PIN max attempts counter reaches 0.
Min. PIN Length (1 byte)	4-255	4	
Max PIN Length (1 byte)	4-255	255 (0xFF)	
PIN character set (1 byte)	Any combination. Each bit represents one character set. example, 0x07 represents the first three character sets	1Fh	Authorized character sets for the PIN. 0x01: Numeric (0-9) ASCII 0x30-0x39 0x02:Alphabetic uppercase (A-Z) ASCII 0x41-0x5A 0x04: Alphabetic lowercase (a-z) ASCII 0x61-0x7A 0x08: Non alphanumeric ASCII 0x20-0x2F, 0x3A-0x40, 0x5B-0x60, 0x7B-0x7F 0x10: Extended characters (non ASCII): ASCII 0x80-0xFF 0x20 All alphabetic characters: ASCII 0x41-0x5A and 0x61-0x7A (A to Z, a to z): If selected, 0x02 and 0x04 are disabled You can combine these to give for example: 0x1F: All characters not case sensitive: ASCII 0x20-0xFF 0x1D: All characters case sensitive (alphabetic must be lowercase): ASCII 0x20-0x40 + 0x5B-0xFF
PIN Complexity Rule 1: Number of different characters that can be repeated at least once (1 byte)	0-255	255 (0xFF - no limitation)	Example, 01 means one character can be repeated but you cannot specify which one.
PIN Complexity Rule 2: Maximum number of times a character can appear (1 byte)	1-255	255 (0xFF - no limitation)	Note: You cannot specify which characters.
Adjacent repeats allowed	Yes, No	Yes	Indicates if repeated characters can be adjacent
PIN History	0..10	0	Number of previous PIN values a new PIN cannot match
Allow unblock	Yes, No	Yes	Defines if it is possible to unblock a PIN

Table 19 - Gemalto Proprietary Parameters (continued)

Parameter	Possible values	Default value	Comments
SSO	Yes, No	No	Cached for all applications. The effect of activating SSO differs according to the Windows operating system. Windows 7 and 8: ■ If using the IDGo 800 CP, the user needs to present the user PIN once only during a session (such as logging on) as long as the IDPrime .NET card is not removed or reset. ■ If the user is using the standard Microsoft CP, SSO is not supported for smart card logon but is supported for other operations, such as digital signature). For example, the user must type his/her PIN when logging on, and then retype it a second time when the first application requires an authentication. After that, no further PIN entries will be required during the same session Vista SP1 and later: The IDGo 800 CP is not available for Vista. If SSO is activated, the IDGo 800 CP can be used with the standard Microsoft CP but not for smart card logon. XP and Vista (before SP1): SSO is not supported.
PIN: One character from each character set	Yes, No	No	If set, the PIN must contain **at least one** character from each of the sets defined in “PIN Character Set”
Mandatory character set	Subset of the “PIN character set” parameter Ex: 0x03 if PIN character set = 0x07	0x00 (no Mandatory char. set)	The PIN must contain at least one character from each character set of this list. The characters which are not part of this list are allowed, if they are part of the PIN character set list. This parameter can be set to a non 0x00 value only if the previous “One character from each character set” parameter is set to “No”. Example with: ■ “PIN character set” = 0x0F ■ “One character from each character set” = No ■ “Mandatory character set” = 0x07 **1abcE** and **1abcE@>** are allowed PIN values **1abcE@>é** and **1abcde** are not allowed PIN values
Max. sequence of characters	1-255	255 (0xFF)	The number of times that a sequence of characters is allowed, for example 1,2,3,4,5,6... Example with this parameter is set to 4: **1234c5** and **abcd1e** are allowed PIN values **12345c** and **abcde1** are not allowed PIN values
Max. # adjacent characters	1-255	255 (0xFF)	The number of times that a character can be repeated in adjacent positions. Ignored if adjacent repeats allowed is set to no. This value cannot exceed PIN complexity rule 2. Example with this parameter set to 4: 1111c1 and aaaa1a are allowed PIN values 11111c and aaaaa1 are not allowed PIN values
CARD_PIN_CHECK (Var. bytes)			
PIN value			Checks the PIN value’s validity against the PIN policy for the specified role. Can be used with SetCardProperty but not GetCardProperty.

Table 19 - Gemalto Proprietary Parameters (continued)

Parameter	Possible values	Default value	Comments
CARD_IMPORT_ALLOWED (1 byte)			
Key Injection allowed	0: No 1: Yes	1: Yes	Linked to “-k” installation parameter. Can be used with SetCardProperty and GetCardProperty
CARD_IMPORT_CHANGE_ALLOWED (1 byte)			
Card Import Allowed property can be changed or not	0: No 1: Yes	1: Yes	Linked to “-l” installation parameter. Can be used with SetCardProperty and GetCardProperty
CARD_VERSION_INFO (4 bytes)			
Card module assembly version number			Returned by GetCardProperty. Cannot be set with SetCardProperty
CARD_SECURE_AM (1 byte)			
Access Manager Admin Key	0: No 1: Yes	0: No	Linked to “-s” installation parameter. Returned by GetCardProperty. Indicates if Access Manager Admin key is different from the standard Admin Key.
CARD_UNBLOCK-FP_SYNC (1 byte)			
Unblock PIN also unblocks FP authentication	0: No 1: Yes	0: No	Linked to “-u” installation parameter. Can be used with SetCardProperty and GetCardProperty.

Using Installation Parameters

The following parameters can be determined by the second method (executing the card module assembly with parameter values)

Table 20 - Installation Parameters

Parameter	Description	Possible values	Default value	Sample
-i	Creates the file system required by the Microsoft Base Smart Card CSP.	None	This parameter is optional since the v7.1 version of the MiniDriver assembly and required for lower versions.	-i
-r	Creates a read only MiniDriver.	None	The smart card is read/ write.	-r (to create read-only)
-s	Activates an Access Manager (AM) administrator key different from the MiniDriver (MD) administrator key.	None	If absent: AM and MD keys are the same (default = 24 X 00...00) If present: AM key = (default = 24 X FF...FF); MD key = (default = 24 X 00...00)	-s
-c	Sets the size (in bytes) of the challenge.	Value multiple of 8 bytes	8 bytes	-c:16

Table 20 - Installation Parameters (continued)

Parameter	Description	Possible values	Default value	Sample
-p	Sets the list of the supported extended PINs.	1 byte bit mask where: bit 1 = PIN role#3 bit 2= PIN role#4 bit 3 = PIN role#5 bit 4 = PIN role#6 bit 5 = PIN role#7	All extended PINs supported.	-p:03 (PIN roles #3 and #4 supported)
-f	Force the user to change all PINs at first use. This applies to all PINs;	None	The PINs do not have to be changed.	-f
-k	Disables key injection.	None	Key injection is allowed.	-k
-l	Disables possibility to revert the '-k' configuration. If the '-l' parameter is set then key injection is no more possible.	None	Key injection can be enabled after a "-k" configuration.	-l
-u	Unblock PIN also unblocks FP If set, unblocking the PIN also unblocks the FP authentication. **Note**: This feature is available from V7.1.0.1 of the minidriver .dll. It is used only if the Biomanager is present and the UVM is not 01 (PIN only)	1 = Yes 0 = No	0	-u:0 -u:1
-n	Sets the maximum number of containers. The number of containers cannot be modified after installation (no property).	1-15	15	-n:12 -n:15

The -r parameter is the same as the CARD_READ_ONLY parameter in “Table 18”.

Configurable Parameters (IDGo 5000 Bio)

The IDGo 5000 Bio solution offers a number of configurable parameters. Unless stated otherwise, these are the same for both versions of IDGo 5000 Bio (XP, and Windows 7/8).

The IDGo 5000 Bio parameters can be modified by passing parameters when executing the .NET BioManager assembly (Precise). “Table 21” presents the list of configurable parameters and provides an example when using the “installation parameter” method.

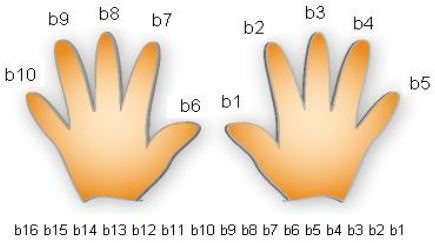
Table 21 - IDGo 5000 Bio Specific Configurable Parameters

Parameter	Description	Possible values	Default value	Example
-p	Sets the UVM of the card. For practical reasons, this is usually set to PIN only. Otherwise it is not possible to logon with the card in modes 2 and 4 which require an FP because no FPs are enrolled on the card yet.	One of the following values: **01: PIN Only (UVM1)** 02: Fingerprint (FP) Only (UVM2) 04: PIN OR Fingerprint (UVM3) 08: PIN AND Fingerprint (UVM4)	01	-p:01 -p:02 -p:04 -p:08

Table 21 - IDGo 5000 Bio Specific Configurable Parameters (continued)

Parameter	Description	Possible values	Default value	Example
-a	Access Condition to FMA.	*admin:* Admin support (Challenge - Response) is required to change the biometric settings and the UVM. *pin*: User can change the biometric settings and UVM without admin support. He or she just has to authenticate him/herself according to the current UVM.	Admin	-a:pin -a:admin
-m	Allowed UVMs. In case the above parameter is set to *PIN*, this parameter allows the administrator to restrict the list of UVMs the user can choose from.	One byte bit mask combination of the following values: **01: PIN Only (UVM1)** 02: Fingerprint (FP) Only (UVM2) 04: PIN OR FP (UVM3) 08: PIN AND FP (UVM4)	0F: All	-m:0F (all UVM allowed) -m:03 (UVM1 + UVM2)
-n	Min. required number of fingerprints enrolled For example when set to 2, implies the requirement is for the user to enroll at least a 2nd finger as backup.	0..10	1	-n:1
-r	Maximum number of attempts before fingerprint verification is blocked	1.. 255	10	-r:5 -r:10
-t	Maximum number of fingerprint templates that can be stored on the card	1..10	10	-t:1 -t:7 -t:10

Table 21 - IDGo 5000 Bio Specific Configurable Parameters (continued)

Parameter	Description	Possible values	Default value	Example
-i	Indicates which of the fingers of each hand can be enrolled and used for authentication This value must be interpreted as a bits field 'b16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1' where the 10 lsb of the value are used to set/unset the fingerprints of both hands. Bits b11 to b16 are not used. The fingerprints are represented as follows in this bits field:  b1 = 1: thumb of right hand is allowed. b2 = 1: index finger of right hand is allowed. b3 = 1: middle finger of right hand is allowed. b4 = 1: ring finger of right hand is allowed. b5 = 1: little finger of right hand is allowed. b6 = 1: thumb of the left hand is allowed. b7 = 1: index finger of left hand is allowed. b8 = 1: middle finger of left hand is allowed. b9 = 1: ring finger of left hand is allowed. b10 = 1: little finger is of left hand is allowed. b11 – b16 are ignored.	Any combination of fingers and thumbs: 0x0001..0x03FF	All (0x03FF)	Both thumbs: '00000000 00100001' (0x0021) Fingers 2,3,7,8: '00000000 11000110' (0x00C6). All fingers and both thumbs: '00000011 11111111' (0x03FF).
-f	False Acceptance Rate (FAR) This is the measure of the likelihood that the biometric security system will incorrectly accept an access attempt by an unauthorized user Recommended range is 5,000-50,000. 1 false match accepted out of the FAR value. The parameter stored in the card is not the FAR itself but 0x7FFFFFFFh / FAR.	1,000 - 1,000,000	10,000	FAR = 1K: -f: 214748 (result of 0x7fffffff/1000 in decimal format) Note: The value **must** be in decimal format

The desired values for these configurable parameters can be set at the time of production of the cards. Otherwise they could be set through a web or client tool designed for such a purpose.

A

Troubleshooting

This appendix includes information to help solve problems encountered while developing IDPrime .NET Card applications.

Communication Problems

The following is a checklist of things to do when you suspect that you are having a communication problem with your card. For the purposes of this document, a communication problem is something that occurs when your software seems to be properly installed, but you can't connect to or see the card.

The Easy Checklist

- Ensure that the smart card really is in the reader.
- Ensure that the correct smart card really is in the reader.
- Ensure the reader is plugged in.
- Check to see if the driver for the reader is installed. To do this, look at your device manager, and ensure that the reader you are using is visible under the “Smart Card Readers” section.
- Ensure that the smart card service is running. To do this, look at the service manager for your machine, and ensure that “Smart Card” is started.

Further Steps

- If another IDPrime .NET Card works in the same reader, there's a good chance that the non-working card is dead. Please post on the Support forum to check the replacement policy for your card.
- If other (non .NET) cards work in the reader, but your IDPrime .NET Card does not, it may be that there is a speed negotiation conflict between the cards and your reader. If you have access to another reader on which the IDPrime .NET Card works, you can use the Card Explorer tool to set the card to a lower communication speed. If you have this problem (and whether setting the communication speed lower fixes the problem or not), please post your issue on the Support forum with the make and model of your reader. We closely track compatibility problems and will work to resolve them.

SSO Option Deactivation Problem

Windows 7 and 8 come with a power saving mode by default. This feature sends the **Power Off** command (63 00 00 …) to the reader after about 20-30 seconds after any transaction to the smart card is completed. When this command is sent to the reader, the reader essentially powers off the card, which triggers the SSO to be deactivated. As a workaround, you can configure the registry key to change the delay period, so that the OS sends the command after a longer period of inactivity.

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\calais\

“CardDisconnectPowerDownDelay”= dword:xxh

Type: REG_DWORD

Value: xx is the delay period in seconds.

To modify the registry key, use the registry editor (from Start > Run, type regedit).

B

Marshaller Stub Sample

This appendix describes a C++ visual project sample that reproduces the steps of a scenario using the stub and marshaller.

The steps to perform are:

1. Connect to the minidriver assembly.
2. Verify the User PIN
3. Read all certificates and decompress them in a storage array
4. Choose the certificate/container for PKI operations (the first one in this sample)
5. Sign some data with a card using PKCS#1
6. Encrypt some data (with RSA software algorithm and card public key)
7. Decrypt the data with the card

```
#define  _WIN32_WINNT   0x0400

#undef UNICODE
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <windows.h>
#include <winscard.h>

#include "CardModuleService.h"
#include "zlib/zlib.h"
#include "rsa/cr_rsa.h"


//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
#define MAX_CONTAINERS (15)

typedef struct _DATA_
{
   BYTE  *pData;
   DWORD  dwDataLen;
} DATA;

static CardModuleService* mdService = NULL;

static string* readerDiscovery = NULL;
static string* instanceName = NULL;
```

```
static DATA  pCertificate[MAX_CONTAINERS];
static DWORD dwNbCert = 0;

static BYTE pubKeyModulus[256];
static BYTE pubKeyExponent[4];
static DWORD dwPubKeyByteLen;

static BYTE dataToSign[256];
static BYTE dataToEncypt[256];
static BYTE encryptedData[256];
static DWORD dwEncryptedLen;

// Default Hash OID for SHA-1
static BYTE  SHA_DER_SIGN_HEADER[] =
{
   0x30, 0x21, 0x30, 0x09, 0x06, 0x05, 0x2B, 0x0E, 0x03, 0x02, 0x1A, 0x05, 0x00,
   0x04, 0x14
};


//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
static BOOL connectMiniDriver()
{
   if (!mdService)
   {
      try
      {
         // Connect to Minidriver Service
         readerDiscovery = new string("selfdiscover");
         instanceName = new string("MSCM");

         // Connect in self discovery mode
         mdService = new CardModuleService(readerDiscovery, // Reader self
discovery mode
                                           5, // Minidriver port number = 5,
mandatory for v1 compatibility
                                           instanceName // Service name = MSCM
                                          );

         delete readerDiscovery;
         readerDiscovery = NULL;

         delete instanceName;
         instanceName = NULL;
      }

      catch(...)
      {
         return FALSE;
      }
   }

   return TRUE;
}

//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
static BOOL releaseMiniDriver()
{
   if (mdService)
```

```
   {
      try
      {
         delete mdService;
         mdService = NULL;
      }
      catch(...)
      {
      }
   }

   return TRUE;
}

//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
void main(void)
{
    BOOL bRes;

    //
    // Connect to MiniDriver Service
    //
    bRes = connectMiniDriver();

    if (!bRes)
    {
        goto _END;
    }

    //
    // Verify User PIN (hard-coded value = "0000")
    //
    try
    {
        u1Array *userPIN = new u1Array(4);
        memcpy_s(userPIN->GetBuffer(), 4, "0000", 4);
        mdService->VerifyPin(1, userPIN);

        printf("User PIN verified\n");

        // Cleanup
        if (userPIN != NULL)
        {
            delete userPIN;
        }
    }

    catch(...)
    {
        printf("PIN verification failed :(\n");

        goto _END;
    }
    printf("\n");

    //
    // Read all certificates and uncompress them
    //
    for (int i = 0; i < MAX_CONTAINERS; i++)
    {
        u1Array *compCert = new u1Array(0);
```

```
            string *fileName = NULL;

            // Read compressed certificate in mscp\kxcXX file (XX = container index)
            try
            {
                char szName[32];

                sprintf(szName, "mscp\\kxc%02x", i);

                fileName = new string(szName);
                compCert = mdService->ReadFile(fileName, 0);

                dwNbCert++;

                // Uncompress Certificate value with zlib and store it in
pCertificate array
                int err = 0;

                pCertificate[i].dwDataLen = compCert->GetBuffer()[2] + (compCert-
>GetBuffer()[3] * 256);
                pCertificate[i].pData = (BYTE *)malloc(pCertificate[i].dwDataLen);

               err = uncompress(pCertificate[i].pData, &pCertificate[i].dwDataLen,
&compCert->GetBuffer()[4], compCert->GetLength() - 4);
            }
            catch (...)
            {
                // Unable to read file -> Certificate not exists for this container
                pCertificate[i].pData = NULL;
                pCertificate[i].dwDataLen = 0;
            }

            // Cleanup
            if (compCert != NULL)
            {
                delete compCert;
            }

            if (fileName != NULL)
            {
                delete fileName;
            }
        }

        printf("Number of certificate(s): %d\n", dwNbCert);

        if (dwNbCert == 0)
        {
            goto _END;
        }
        printf("\n");

        //
        // Analyze certificates to find the one to use
        // In this sample we take the first available certificate
        //
        BYTE bContainerIdx = 0;

        for (int i = 0; i < MAX_CONTAINERS; i++)
        {
            if (pCertificate[i].pData != NULL)
            {
```

```
            bContainerIdx = (BYTE)i;
            break;
        }
    }

    //
    // Get public key of selected container
    //
    try
    {
        u1Array *container = new u1Array(0);

        container = mdService->GetCAPIContainer(bContainerIdx);

        // Scan the container buffer to find the public key value (Modulus /
Exponent) of Exchange key
        // See the Integration guide for details about buffer format
        BYTE *ptr = container->GetBuffer();
        int i = 0;
        BYTE keyType;

        while (i < (int)container->GetLength())
        {
            // Key type TLV (Tag = 0x03) - L is coded on 1 byte, V is always 1
byte
            if (ptr[i] == 0x03)
            {
                // Store key type (Exchange or Signature) contained in V
                keyType = ptr[i+2];

                i += 3; // Skip T, L and V bytes
            }

            // Public Exponent TLV (Tag = 0x01) - L is coded on 1 byte, V is
always 4 bytes
            if (ptr[i] == 0x01)
            {
               // Store Public Exponent if the current key is the exchange key
(Type = 0x01)
                if (keyType == 0x01)
                {
                    memcpy(pubKeyExponent, &ptr[i+2], 4);
                }

                i += 6; // Skip T, L and V bytes
            }

            // Public Modulus TLV (Tag = 0x02) - L is coded on 1 byte, V is
variable length
            if (ptr[i] == 0x02)
            {
              // Byte length of Modulus, see the Integration guide for details
about the <<4 bit shift!
                dwPubKeyByteLen = (ptr[i+1]<<4);

               // Store Public Exponent if the current key is the exchange key
(Type = 0x01)
                if (keyType == 0x01)
                {
                    memcpy(pubKeyModulus, &ptr[i+2], dwPubKeyByteLen);
                }
```

```
                i += (2+dwPubKeyByteLen); // Skip T, L and V bytes
            }
        }

        // cleanup
        if (container != NULL)
        {
            delete container;
        }

        // Display Public key Modulus
        printf("Public key Modulus: ");

        for (int i = 0; i < (int)dwPubKeyByteLen; i++)
        {
            printf("%02X", pubKeyModulus[i]);
        }

        printf("\n");

        // Display Public key Exponent
        printf("Public key Exponent: ");

        for (int i = 0; i < 4; i++)
        {
            printf("%02X", pubKeyExponent[i]);
        }

        printf("\n");
    }
    catch(...)
    {
        printf("Get Public key container failed :(\n");

        goto _END;
    }
    printf("\n");

    //
    // Sign a data with RSA private key in container associated to selected
    // certificate. In this sample we assume that we will sign a SHA-1 on 20
    // bytes
    //
    BYTE hashData[20];

    // Fake SHA-1 for this sample = A1...A1 (20 bytes)
    memset(hashData, 0xA1, sizeof(hashData));

    // Add PKCS#1 padding for signature
     memset(dataToSign, 0x00, sizeof(dataToSign));
     dataToSign[0] = 0x00;
     dataToSign[1] = 0x01;
    memset(&dataToSign[2], 0xFF, dwPubKeyByteLen - sizeof(SHA_DER_SIGN_HEADER)
- sizeof(hashData) - 3);
     dataToSign[dwPubKeyByteLen - sizeof(SHA_DER_SIGN_HEADER) -
sizeof(hashData) - 1] = 0x00;

     // Add SHA-1 OID
     memcpy(&dataToSign[dwPubKeyByteLen - sizeof(SHA_DER_SIGN_HEADER) -
sizeof(hashData)],
            SHA_DER_SIGN_HEADER,
            sizeof(SHA_DER_SIGN_HEADER)
```

```
            );

         // Add SHA-1 hashed data
         memcpy(&dataToSign[dwPubKeyByteLen - sizeof(hashData)],
                hashData,
                sizeof(hashData)
               );

        // Send the data to sign using RSA raw decryption
        try
        {
            u1Array *signature = new u1Array(0);
            u1Array *data = new u1Array(dwPubKeyByteLen);
            data->SetBuffer(dataToSign);

            // Display data to sign
            printf("Data to sign: ");

            for (int i = 0; i < (int)data->GetLength(); i++)
            {
                printf("%02X", data->GetBuffer()[i]);
            }

            printf("\n");

            signature = mdService->PrivateKeyDecrypt(bContainerIdx, // Container
Index
                                                     0x01, // Exchange key type
                                                      data // Data to sign
                                                     );

            // Display Signature
            printf("Signature: ");

            for (int i = 0; i < (int)signature->GetLength(); i++)
            {
                printf("%02X", signature->GetBuffer()[i]);
            }

            printf("\n");

            // cleanup
            if (data != NULL)
            {
                delete data;
            }

            if (signature != NULL)
            {
                delete signature;
            }
        }
        catch(...)
        {
            printf("Signature failed :(\n");

            goto _END;
        }
        printf("\n");

        //
        // Decrypt a data with RSA private key in container associated to selected
```

```
        // certificate. In this sample we encrypt a fake 3DES key data with software
        // RSA and card public key, then the data is decypted using the card
        //
        BYTE des3Key[24];

        // Fake 3DES key for this sample = 3D...3D (24 bytes)
        memset(des3Key, 0x3D, sizeof(des3Key));

        // Add PKCS#1 padding for encryption
         memset(dataToEncypt, 0x00, sizeof(dataToEncypt));
         dataToEncypt[0] = 0x00;
         dataToEncypt[1] = 0x02;
         memset(&dataToEncypt[2], 0xFF, dwPubKeyByteLen - sizeof(des3Key) - 3);
         dataToEncypt[dwPubKeyByteLen - sizeof(des3Key) - 1] = 0x00;

         // Add 3DES key data
         memcpy(&dataToEncypt[dwPubKeyByteLen - sizeof(des3Key)],
                des3Key,
                sizeof(des3Key)
               );

        // Encrypt Data with RSA software and RSA plublic key read from card
        R_RSA_PUBLIC_KEY rsaKeyPublic;

        rsaKeyPublic.bits = dwPubKeyByteLen*8;

        memcpy(rsaKeyPublic.modulus, pubKeyModulus, dwPubKeyByteLen);

        memset(rsaKeyPublic.exponent, 0x00, dwPubKeyByteLen) ;
        memcpy(&rsaKeyPublic.exponent[dwPubKeyByteLen - 4], pubKeyExponent, 4);

        dwEncryptedLen = dwPubKeyByteLen;

        RSAPublicBlock(encryptedData,
                       (unsigned int *)&dwEncryptedLen,
                       dataToEncypt,
                       dwPubKeyByteLen,
                       &rsaKeyPublic
                      );

        // Send the data to RSA raw decryption
        try
        {
            u1Array *clearData = new u1Array(0);
            u1Array *data = new u1Array(dwEncryptedLen);
            data->SetBuffer(encryptedData);

            // Display encrypted data
            printf("Encrypted data: ");

            for (int i = 0; i < (int)data->GetLength(); i++)
            {
                printf("%02X", data->GetBuffer()[i]);
            }

            printf("\n");

            clearData = mdService->PrivateKeyDecrypt(bContainerIdx, // Container
    Index
                                                     0x01, // Exchange key type
                                                     data // Data to decrypt
```

```
                                                    );

            // Display Decrypted data
            printf("Decrypted data: ");

            for (int i = 0; i < (int)clearData->GetLength(); i++)
            {
                printf("%02X", clearData->GetBuffer()[i]);
            }

            printf("\n");

            // cleanup
            if (data != NULL)
            {
                delete data;
            }

            if (clearData != NULL)
            {
                delete clearData;
            }
        }
        catch(...)
        {
            printf("Signature failed :(\n");

            goto _END;
        }
        printf("\n");


    _END:
        //
        // Release MiniDriver Service
        //
        releaseMiniDriver();

        //
        // pCertificate array cleanup
        //
        for (int i = 0; i < MAX_CONTAINERS; i++)
        {
            if (pCertificate[i].pData != NULL)
            {
                free(pCertificate[i].pData);
            }
        }

        printf("Press a key to exit...");
        getch();
    }
```

Abbreviations

ACL	Access Condition List
AD	Application Domain
API	Application Programming Interface
CLR	Common Language Runtime
CP	Credential Provider
CSP	Cryptographic Service Provider
ECMA	European Computer Manufacturers Association
EEPROM	Electrical Erasable Programmable Read-only Memory
GUID	Gemalto Unique Identifier
IL	Intermediate Language
MSCM	Microsoft Card Module
MSDN	Microsoft Developer Network
OLH	Online Help
OS	Operating System
PIN	Personal Identification Number
PKI	Public Key Infrastructure
RAM	Random Access Memory
ROM	Read-Only Memory
SSO	Single Sign-on

Glossary

Application Domain	Every application executes in a secure, isolated execution area, which enforces strict firewalls between applications and helps maintain data integrity. Data in one domain cannot be accessed from any other domain.
Access Manager Admin Key	An optional additional 3DES key that can be activated using the -s installation parameter. It can be used as an alternative to the Admin Key for some (but not all) functions. The role of the Access Manager Admin key is to be able to administer the .NET card. For example, you need to authenticate yourself with this role if you want to load a new assembly.
Admin Key	A 3DES key used by the administrator to calculate the response to a challenge when unblocking the card.
Base (CSP)	Microsoft's default software library that implements the Cryptographic Application Programming Interface (CAPI).
Card Explorer	The Card Explorer is the tool available to manage IDPrime .NET Cards. It is part of the IDPrime .NET SDK.
Certificate	A certificate provides identification for secure transactions. It consists of a public key and other data, all of which have been digitally signed by a CA. It is a condition of access to secure e-mail or to secure Web sites.
Certificate Authority	An entity with the authority and methods to certify the identity of one or more parties in an exchange (an essential function in public key crypto systems).
Common Language Runtime (CLR)	A core component of .NET. It is Microsoft's implementation of the Common Language Infrastructure (CLI) standard, which defines an execution environment for program code. In the CLR, code is expressed in a form of bytecode called the Common Intermediate Language (CIL, previously known as MSIL—Microsoft Intermediate Language).
Credential Provider (CP)	A CP is the Windows 7 (and later versions) software module in charge of the authentication of the user based on various technologies. The standard Windows CP handles the Username / Password and smart card based PIN methods. Gemalto's IDGo 800 CP provides additional features such as the "PIN change at first use" option, or the biometrics authentication method.
Cryptography	The science of transforming confidential information to make it unreadable to unauthorized parties.
Digital Signature	A data string produced using a Public Key Crypto system to prove the identity of the sender and the integrity of the message.
Encryption	A cryptographic procedure whereby a legible message is encrypted and made illegible to all but the holder of the appropriate cryptographic key.

Gemalto Unique identifier	Unique Id of .NET card. By default 4 byte fixed value (0x2E4E4554) || 12-byte card serial number.
Intermediate Language (IL)	A new language designed to be efficiently converted into native machine code on different types of devices. IL is based on an abstract stack-based processor architecture.
Key	A value that is used with a cryptographic algorithm to encrypt, decrypt, or sign data. Secret key crypto systems use only one secret key. Public key crypto systems use a public key to encrypt data and a private key to decrypt data.
Key Length	The number of bits forming a key. The longer the key, the more secure the encryption. Government regulations limit the length of cryptographic keys.
Manifest	An assembly manifest is a text file containing metadata about .NET assemblies. It describes the relationship and dependencies of the components in the assembly, versioning information, scope information and the security permissions required by the assembly.
PKCS#11	Standard and open software library specified by RSA Laboratories and implementing smart card cryptographic functions. Refer to http://www.rsa.com/rsalabs/node.asp?id=2133
Remoting	.NET Remoting allows an application to make an object (termed remotable object) available across remoting boundaries, which includes different application domains, processes or even different computers connected by a network
Single Sign-on (SSO)	A mechanism provided with the IDGo 800 CP, where the user needs to present the User PIN once only during a session, as long as the .NET card is not removed. If the standard Microsoft credential provider is used, activating the SSO mechanism has no effect and the user PIN may need to be presented more than once during a session.
Sinks	Sinks are links in a chain that can be used by a proxy to send a message to a remote object. Usually the sinks modify the data before sending it to the next sink.
Strong name signing	Strong-name signing, or strong-naming, gives a software component a globally unique identity that cannot be spoofed. Strong names are used to guarantee that component dependencies and configuration statements map to exactly the right component and component version. A strong name consists of the assembly's identity (simple text name, version number, etc.), plus a public key token and a digital signature.
Strongly typed object	A class that contains data which is written and read using access functions. Since each data has a specific type, errors are caught at compile time instead of runtime.

Standards and Specifications

- *Smart Card Minidriver Specification for Windows Base Cryptographic Service Provider (Base CSP) and Smart Card Key Storage Provider (KSP), Version 5.07*, September 12, 2007 from Microsoft.
- *Windows Smart Card Minidriver Specification, Version 6.02*, March 7, 2008 from Microsoft.
- *Windows Smart Card Minidriver Specification, Version 7.06*, July 1, 2009 from Microsoft.

Note: These specifications can be downloaded from http://msdn.microsoft.com/en-us/windows/hardware/gg487500.aspx.

- *ISO/IEC 7816: Information Technology – Identification cards – Integrated circuit cards with contacts*
  - *Part 1: Physical characteristics*
  - *Part 2: Cards with contacts -- Dimensions and location of the contacts*
  - *Part 3: Electronic signals and transmission protocols*
    Defines the characteristics of the electronic signals exchanged between the card and the terminal, and the two possible low-level communication protocols that can be used. (T = 0 is an asynchronous half-duplex character transmission protocol; T = 1 is an asynchronous half-duplex block transmission protocol).
  - *Part 4: Industry commands for interchange*
    Defines a set of standard commands for smart cards, as well as a hierarchical file system structure for cards. These commands are the base of most existing card protocols.
- *ISO/IEC-10536: Identification Cards - Contactless Integrated Circuit(s) Cards - Close-coupled Cards*
  - *Part 1: Physical characteristics*
  - *Part 2: Dimensions and Location of Coupling Areas*
  - *Part 3: Electronic Signals and Reset Procedures*
- *ECMA-335: Common Language Infrastructure*
  Defines the Common Library Infrastructure (CLI), which ensures that applications written in multiple high-level languages may be executed in different system environments without the need to rewrite the application to take into consideration the unique characteristics of those environments.
- *PC/SC: Interoperability Specification for ICCs and Personal Computer Systems*
  Defines low-level device interfaces and device-independent application APIs as well as resource management, to allow multiple applications to share smart card devices attached to a system.
- *NIST: FIPS PUB 140-2: Security Requirements for Cryptographic Modules*
  Specifies the security requirements that will be satisfied by a cryptographic module utilized within a security system protecting sensitive but unclassified information.

Recommended Reading

- For further reading about IDPrime .NET Cards, please go to the Gemalto product catalog at http://www.gemalto.com/products/dotnet_card/
- *Advanced .NET Remoting* from Ingo Rammer; APress January 1, 1970); ISBN: 1590590252
- *Common Language Infrastructure Annotated Standard* from James S. Miller, Susann Ragsdale, Jim Miller; Addison-Wesley Pub Co, 1st edition (October 24, 2003); ISBN: 0321154932
- *.NET Framework Security* from Brian A. LaMacchia, Sebastian Lange, Matthew Lyons, Rudi Martin, Kevin T. Price; Addison-Wesley Pub Co, 1st edition (April 24, 2002); ISBN 067232184X

Useful Web Site Addresses

Microsoft's - .NET Framework Developer Center (msdn.microsoft.com)