Developer Guide

DigitalPersona®Pro

Platinum SDK

Version 3.3.0

digitalPersona.

Should you have any questions concerning this document, or if you need to contact DigitalPersona for any other reason, write to:

DigitalPersona, Inc.
720 Bay Road
Suite 100
Redwood City, CA 94063
USA

Document Publication Date: 05/04/07

Table of Contents

Introduction 1

The DigitalPersona Platinum SDK Programmer's Guide shows programmers how to use the Platinum SDK to integrate fingerprint recognition functionality in their applications.

What's in This Guide

This chapter describes the requisite knowledge a programmer must possess in order to use this guide and the SDK. It also explains your technical support options, as well as the conventions used in this guide.

Following a description of each chapter in this guide:

Chapter 2, Installing the SDK, provides system requirements and installation instructions for the Platinum SDK.

Chapter 3, Using the SDK, describes how to use the Engine and Operations Layers of the Platinum SDK to incorporate fingerprint recognition functionality in applications.

Chapter 4, Managing User Data, shows ways for programmers to use the User Layer to manage user fingerprint data in the Digital Persona-supplied user database.

Chapter 5, SDK Reference, describes the functions, interfaces, methods and properties of the Platinum SDK COM components and ActiveX controls.

A Reference Index is provided at the end of this guide to quickly and easily locate the contents of Chapter 5, *SDK Reference*.

Requisite Knowledge

Programmers who want to use the Platinum SDK are required to be familiar with the following subjects:

- Familiarity with the Component Object Model (COM) and a working knowledge of COM-based technologies, such as distributed COM and ActiveX controls.
- Any programming language that can interface with COM objects, such as Visual Basic or C++.

Support Resources

In addition to this guide, the following resources are provided for additional support:

- A Readme file is provided on the product CD, which contains last-minute information about the product.
- The Digital Persona Web site (http://www.digitalpersona.com) provides an online technical support form in the Support section. You can describe your issue and include your contact information and a technical support representative will contact you by e-mail or phone.
- E-mail support is available at `techsupport@digitalpersona.com`.
- Phone support can be reached at (877) 378-2740 in the U.S. only.
  Outside the U.S., call +1 650-474-4000.

Typographic Conventions

The following typographic conventions are used in this guide:

- `Courier` indicates text that is either typed by the user or data displayed in a command line interface program, such as the Terminal window or MS-DOS. It is also used to display lines of code.
  Example:
  “Type `http://www.digitalpersona.com/` in the Address text box.”
  You would only type “http://www.digitalpersona.com/” and would not type any surrounding text.
- Text in **`Courier bold`** and surrounded by brackets **`[ ]`** indicates information that varies depending on a particular circumstance. This information is always supplied by you.
  Example:
  “Type `http://`**`[your company web site URL]`**/ in the Address text box.”
  You would type “http://”, then type your company web site URL—not the words “[your company web site URL]”—and then “/”.
- When describing sample code, *Italics* are used to indicate variables, parameters, etc. They are not part of the Platinum SDK, but are supplied as

examples and can be substituted by the programmer.

Example:

“`Dim db As DPUsersDB`”

The above line of sample code contains the reference variable, *db*, which can be substituted with any other variable name.

Notational Conventions

The following notational conventions are used in this guide to call attention to information of special importance:

Note

A note highlights information that may help you better understand the text and its concepts.

Warning

A warning advises you that failure to take or avoid a specific action could result in your inability to complete the required tasks.

Naming Conventions

For brevity and easier reading of this guide, the following naming conventions are used to describe the DigitalPersona Platinum SDK and fingerprint reader hardware:

- Platinum SDK and SDK sometimes replace the full name, DigitalPersona Platinum SDK.
- Reader—in both upper and lower case—is always used without the preceding U.are.U. It replaces the full product name, U.are.U Fingerprint Reader.

Your Feedback Requested

The information in this guide has been thoroughly reviewed and tested. If you find errors or have suggestions for future publications, contact Digital Persona at:

720 Bay Road, Suite 100
Redwood City, California 94063 USA
(650) 474-4000
(650) 298-8313 FAX

Installing the SDK 2

This chapter describes the installation of the DigitalPersona Platinum SDK.

Developer System Requirements

Before installing the Platinum SDK, ensure your system meets the following minimum requirements:

- Pentium-class processor
- Windows 2000, Windows XP, Windows Vista, or Windows Server 2003
- 32 MB minimum physical RAM (64 MB physical RAM recommended)
- CD-ROM drive
- 32 MB minimum hard disk space during installation
- 5 MB hard disk space for installation without DigitalPersona Pro or 1.5 MB hard disk space for installation when DigitalPersona Pro is already installed

Running the Setup Application

This section describes how to install the DigitalPersona Platinum SDK using the setup program. To install the SDK “silently” (from the command line), see “Installing the SDK From the Command Line” on page 6.

To install the DigitalPersona Platinum SDK

1. Insert the DigitalPersona Platinum SDK CD in the CD-ROM drive and double-click the setup.exe file, located in the Install folder.
2. When the DigitalPersona Platinum SDK Setup dialog box opens, click Next to begin the installation.
3. Review the license agreement and, if you agree to the terms, click I accept the license agreement and then click Next.
4. Click Next again to install DigitalPersona Platinum SDK in the default destination folder, C:/Program Files/DigitalDigitalPersona Persona.
   If you want to install the software in another location, click Browse and specify a different location before clicking Next.
5. When installation is complete, click Finish and then restart the computer when prompted.

The installer creates the DigitalPersona Platinum SDK folder in the location you specified in step 4. Inside, sample applications that use the Platinum SDK are provided in both Visual Basic and C++.

Note
If you are using the DigitalPersona Platinum SDK on a workstation, configure the workstation to use the same DNS as your DigitalPersona Pro Server machine. For more information,refer to the DigitalPersona Pro Administrator Guide.

Note
You can use these sample applications in conjunction with the code examples in Chapter 3, Using the SDK and Chapter 4, Managing User Data to see how they behave in an actual application.

Installing the SDK From the Command Line

To install the SDK from the command line, enter:

```
msiexec.exe/i Setup.mis/qn
```

Testing and Deploying Your Applications

If you want to test your application on a PC that does not have any Digital Persona software installed on it—presumably to gauge the end-user experience—you must install DigitalPersona Platinum Fingerprint Recognition Software.

The DigitalPersona Platinum Fingerprint Recognition Software package comes with a reader and a CD containing the required drivers and other files to use applications that provide fingerprint recognition functionality using DigitalPersona Platinum SDK software.

When deploying your application to end-users, they will also have to install DigitalPersona Platinum Fingerprint Recognition Software.

If you want to simplify the installation process by removing this added step for your audience, you can install DigitalPersona Platinum Fingerprint Recognition

Software in silent mode. The following instructions describe the procedure for performing this task:

1 Copy all the files from the DigitalPersona Platinum Fingerprint Recognition Software CD: \Install folder to your hard disk.

2 Open the Setup.ini file. Locate the last line of code in the file (shown below):
```
;cmdline=/qn /I
```
3 Remove the semicolon (;) at the beginning of the last line to uncomment the silent install command (shown below):
```
cmdline=/qn /I
```
4 Save your changes to the Setup.ini file. Setup.exe, which installs DigitalPersona Platinum Fingerprint Recognition Software, will run in silent mode.

Using the SDK 3

This chapter introduces the fingerprint recognition operation and describes how to use the Engine and Operations Layers of the SDK to implement it in your software. In addition, two methods are described for adding extra security to the fingerprint recognition operation.

Fingerprint Recognition Operation

The fingerprint recognition operation identifies the processes involved in registering and verifying fingerprints from a developer perspective. You must be familiar with this operation and the related terminology to use the SDK to integrate fingerprint recognition functionality in your application.

The following processes comprise the fingerprint recognition operation:

1. **Acquire a fingerprint scan.** The first step in the fingerprint recognition operation is to acquire a fingerprint scan. When a user touches the reader, a fingerprint scan—called a raw sample—is compressed and encrypted by the reader and sent to the PC.
2. **Decrypt and decompress the raw sample.** When the raw sample is received from the reader, it is decrypted and decompressed into a sample from which features can be extracted to create a template.
3. **Create a template.** After determining the intended operation—either registration or verification—create the appropriate template. Created from the sample, a template is a mathematical description of the fingerprint characteristics and is assigned one of two types: a pre-registration or verification template.
4. **Perform registration or verification operation.** Following is a description of the registration and verification processes:
   - **Register.** If a new fingerprint is being registered, you must acquire four preregistration templates which are used to create a single registration template. The registration template can then be stored for later use during the verification process.
   - **Verify.** In the verification operation, a verification template is acquired and compared to an existing registration template for matching.

Choosing a Layer

Which layer you choose to implement fingerprint recognition functionality in your application can be based on several factors, ranging from the level of control over the fingerprint recognition operation you require to the degree of experience you have as a programmer.

The Engine Layer is intended for programmers who require control over every process in the fingerprint recognition operation. The Operations Layer is best for those who would benefit from a faster approach to implementation, as well as a less complex one.

Engine Layer

The Engine Layer allows you to facilitate—and control every aspect of—the processes in the fingerprint recognition operation. In this section, the procedure for implementing the processes using the Engine Layer is described, followed by sample code written in Visual Basic.

Note

Only the minimum methods and properties are used in the sample code to implement a particular process. A description of all other methods and properties can be found in Chapter 5, "SDK Reference," on page 45.

Acquiring a Fingerprint Scan

To acquire a fingerprint scan, identify which readers will acquire it and subscribe for the event that is fired when the user touches the reader.

To acquire a fingerprint scan using the Engine Layer components

1 Create an instance of the FPDevices (sensor manager) object to provide a reference for each reader connected to the PC.
2 Using the FPDevice object, point to each reader that will acquire a fingerprint scan using the reference to the reader in the FPDevices object.
3 Connect the FPDevice object to its event interface to receive—among other events—the SampleAcquired event, which is fired when a user places a finger on the reader and the fingerprint scan is acquired.

The following sample code shows one way to implement these steps using Visual Basic:

```
'sensor manager object variable
Dim WithEvents myDevices As FPDevices

'variable pointer to the selected sensor
Dim WithEvents dev As FPDevice

'create sensor manager object
Set myDevices = New FPDevices

'enumerate sensors
For Each dev In myDevices
'connect each sensor to the event interface
dev.SubScribe Dp_StdPriority, Me.hWnd
Next
```

The sample code creates two variables:

- *myDevices*, the FPDevices object that holds references to each reader and monitors plug-and-play events when the sensor manager object is created.
- *dev*, the FPDevice object used as a pointer to each sensor reference in *myDevices*.

A For Each statement loops for as many sensors as there are referenced in *myDevices* and points *dev* to a reference in it. The loop executes a line of code that connects *dev* to its event interface using the SubScribe method.

The SubScribe method requires two parameters: the event priority and the window handle. In the sample code, standard priority is indicated with the *Dp_StdPriority* value. It is the most commonly used priority and requires the window specified by the handle to have focus for the application to be notified of the event.

Decrypting and Decompressing the Raw Sample

A handler for the SampleAcquired event—fired when the users places a finger on the reader—receives the encrypted and compressed raw sample as a parameter and decrypts and decompresses it into a sample.

To convert a raw sample into a sample using the Engine Layer components

1 Create an instance of FPRawSamplePro, the sample processor object.
2 Use the Convert method to convert the raw sample into a sample and store it in a FPSample object.

The following code shows how to convert the raw sample into a sample and exercises the option to display the resulting fingerprint scan:

```
Private Sub dev_SampleAcquired(ByVal pRawSample As
  Object)
Dim sample as FPSample

'sample processor object
Dim smpPro As FPRawSamplePro

'create the sample processor object
Set smpPro = New FPRawSamplePro

'perform the conversion
smpPro.Convert pRawSample, sample

'set the orientation of the image
sample.PictureOrientation = Or_Portrait

'resize it to the size of the picture box
sample.PictureWidth = picSample.Width /
  Screen.TwipsPerPixelX
```

```
sample.PictureHeight = picSample.Height /
  Screen.TwipsPerPixelY
'display it
picSample.Picture = sample.Picture

End Sub
```

The event handler in the sample code accepts the raw sample as a parameter (*pRawSample*) and creates two variables:

- *sample*, the FPSample object that will contain the decrypted and decompressed sample.
- *smpPro*, the FPRawSamplePro object whose Convert method will convert the raw sample into a sample and other methods will be used to display the fingerprint scan.

An instance of the FPRawSamplePro object (*smpPro*) is created. Using the Convert method, *pRawSample* is converted and the result is stored in *sample*.

Before displaying the fingerprint scan, two methods are used to determine its characteristics, i.e., orientation and dimensions. The PictureOrientation property of *sample* is set to portrait with the Or_Portrait value. The PictureWidth and PictureHeight properties are used to determine the width and height of the scan in pixels. A PictureBox object, *picSample*, is used to specify the value of the width and height properties.

The last line of code in the event handler displays the scan, using the Picture property of sample to assign a value for the Picture property of the PictureBox object, picSample.

Creating a Template

To create a template, extract features from the sample and specify its type, which is based on the intended operation, i.e., registration or verification.

To create a template using the Engine Layer

1 Create an instance of the FPFtrEx (feature extraction) object.

2 Use the Process method of the feature extraction object to extract features from the sample (supplied to the method as a parameter), specify the template type and create the template. In addition to the template, a rating of the sample quality is returned.

The following code shows how to extract features from *sample*, the FPSample object described in “Decrypting and Decompressing the Raw Sample” on page 10, specify whether the template will be either a preregistration or verification template and obtain the sample’s quality rating:

```
'feature extraction object
Dim ftrex As FPFtrEx

'template object
Dim template as FPTemplate

'quality code variable
Dim qt As AISampleQuality

'create the feature extraction object
Set ftrex = New FPFtrEx

'perform feature extraction (preregistration)
ftrex.Process sample, Tt_PreRegistration, template, qt
```

There are three variables created in the sample code:

- *ftrex*, is the FPFtrEx (feature extraction) object that provides the Process method.
- *template*, is the FPTemplate (template) object that contains the template created by the feature extraction process.
- *qt*, is that AISampleQuality object that contains the quality rating of the *sample*.

After the FPFtrEx object, *ftrex*, is instantiated, the Process method is called. It receives the sample (*sample*) and the parameter that specifies the type of template, i.e., Tt_PreRegistration, which indicates a preregistration template. It returns the template (*template*) and the quality rating of the sample (*qt*).

To return a verification template, the Tt_Verification parameter should be used, as shown in the following:

```
'perform feature extraction (verification)
ftrex.Process mySample, Tt_Verification, template, qt
```

The quality rating of sq_Good must be returned for either type of template or the template variable will be returned empty.

Creating a Registration Template

A registration template is created from four preregistration templates. Once created, the registration template is stored for later use during the verification process, as described in "Verifying a Template" on page 16.

To create a registration template

1 Create an instance of the FPRegister (preregistration) object.
2 Acquire four preregistration templates and add them to the preregistration object using the Add method.
3 Retrieve the final registration template using the RegistrationTemplate property of the FPRegister object.
4 Store the registration template for use during the verification operation.

The following code initializes a registration object:

```
'declare registration object
Dim register As FPRegister

'create the registration object
Set register = New FPRegister
```

```
'configure registration component to produce a template
'to be used for verification (not identification)
register.NewRegistration Rt_Verify
```

In the sample code above, an FPRegister object, *register*, is instantiated. Then, the NewRegistration method is used to reinitialize the object. The Rt_Verify parameter—which is currently defined but not implemented—is supplied to indicate that the registration template will be used for the verification operation. When a preregistration template (*template*) is acquired, as described in “Create a Template” on page 11, add it to *register* using the Add method:

```
Dim bDone As Boolean
register.Add template, bDone
```

The sample code creates a boolean variable, *bDone*. When a preregistration template is added to the FPRegister object, *bDone* is false until the fourth preregistration template is acquired and added.

When four preregistration templates are added to *register, bDone* will equal true. To get the final registration template, use the RegistrationTemplate property to initialize a variable of type FPTemplate:

```
Dim regtemplate As FPTemplate
Set regtemplate = register.RegistrationTemplate
```

The sample code creates an FPTemplate object, *regtemplate*, and stores the registration template by assigning it the value of the FPRegister object’s RegistrationTemplate property.

When the registration template is created, it can be saved to either a database or file by extracting the template as an encoded and signed blob of data.

Note

The template size may not stay the same from one version of the SDK to another. When storing templates in the database, do not assume that the template will always be the same size.

The following sample code extracts the template as a blob of data and saves it to a file:

```
'binary blob containing the template data
Dim blob As Variant

'same blob as an array of bytes (to be used with Put)
Dim blobarray() As Byte

'extract the template data
regtemplate.Export blob
'store it in the array of bytes and save it to file
blobarray = blob
Open "c:\template.fpt" For Binary As #1
Put #1, , blobarray
Close #1
```

The sample code uses the Export method of FPTemplate object (*regtemplate*) to extract the template data and stores the blob in the variant variable blob. The data in blob is assigned to the byte array, *blobarray*, which is then written to a file.

By default, the registration template is a byte array. Some developers may want to convert the byte array to a hexadecimal string before saving it to a file. If the database used does not support binary data, then use these routines to convert from the binary to string data type. The following code performs this task:

```
Function arraytohex(arr() As Byte) As String
  Dim templatestr As String
  Dim tempstr As String
  Dim i As Integer

  templatestr = ""
    For i = LBound(arr) To UBound(arr)
      tempstr = Hex$(arr(i))
      If Len(tempstr) = 1 Then tempstr = "0" + tempstr 'pad
      hex
      templatestr = templatestr + tempstr
    Next i

  arraytohex = templatestr
End Function
```

The following sample code shows how to save fingerprint data in a database table using ADOs. The example uses a table “users” which has a field “fingerprint” of type binary, and this database is accessable via ODBC having Data Source Name = DSName.

```
Set cnx = New Connection
Set rs = New Recordset
Dim bvariant As Variant
Dim blob_write() As byte

bvariant = Null

cnx.Open "DSName", "", ""
rs.Open "select * from users", cnx, adOpenKeyset,
  adLockOptimistic

register.Export bvariant

blob_write = bvariant

rs.AddNew
rs("fingerprint") = blob_write
rs.Update
```

Using the XTF Registration Template

The XTF template is an extended form of the registration template and provides improved verification performance. The XTF template combines data from each of the four registration fingerprint scans. This is the default format used for templates.

The interfaces for implementing the XTF template are derived from the basic template interfaces. The UsingXTFTemplate property for each of the XTF template interfaces must be set to the boolean value of True to use the XTF template. The interfaces that include a UsingXTFTemplate property are the following:

- IFPRegister2
- IFPRegisterTemplateX2
- IFPRegisterUserX2
- IFPRegisterTemplate2
- IFPRegisterUser2

Verification time when using the the XTF template is on average 10% longer than the basic registration. At the most, verification time with the XTF template can be up to four times longer than the basic template.

The XTF template takes up approximately four times more space than the basic template when saved to a file or to a database. Developers who have already created programs using the basic template and want to switch to the XTF template must consider if the size increase will require more space allocated in their existing databases.

Verifying a Template

To verify a template, obtain the registration template from its stored location—e.g., file or database—and acquire a verification template and compare them for a match.

To verify a template using the Engine Layer

1. Create an instance of the FPTemplate object, acquire the blob from its source—e.g., a file or database—and store it in the object using the Import method.
2. Acquire a verification template, as described in “Acquiring a Fingerprint Scan” on page 9, “Decrypting and Decompressing the Raw Sample” on page 11 and “Creating a Template” on page 12.
3. Create an instance of the FPVerify object and use the Compare method to compare the verification template against the registration template.

The following sample code loads a registration template from a file and stores it in an FPTemplate object:

```
'declare the blob variable
Dim blob() As Byte

'create an instance of an FPTemplate object
Set template = New FPTemplate

'read the template data from the file
Open "c:\template.fpt" For Binary As #1
ReDim blob(LOF(1))
Get #1, , blob()
Close #1

'import the template data into the template object
template.Import blob
```

The code creates an instance of the FPTemplate object, *template*. It retrieves the template data from a file and stores it in the byte variable, *blob*. Using the Import method, the template data in *blob* is stored in *template*.

If you stored the registration template to a file as a hexadecimal string, you must convert it to a byte array before using it in the verification operation. The following code performs this task:

```
Public Sub hextoarray(inphex As String, outarray() As
  Byte)
  ReDim outarray(0 To Len(inphex) / 2) As Byte
  Dim i As Integer
  For i = 1 To Len(inphex) Step 2
    outarray(((i + 1) / 2) - 1) = Val("&H" + Mid$(inphex,
      i, 2))
  Next i
End Sub
```

The following sample code compares the registration template to the verification template:

```
'declare matching object
Dim verify As FPVerify

'declare output variables of the matching operation
Dim result As Boolean
Dim score As Variant
Dim threshold As Variant
Dim learn As Boolean
Dim sec As AISecureModeMask

'create matching object
Set verify = New FPVerify

'perform the match
verify.Compare regtemplate, vertemplate, result, score,
  threshold, learn, sec
```

There are six variables created in the sample code:

- *verify*, the FPVerify object that will be used to perform the match.
- *result*, a boolean that is true if a match is successful.
- *score*, the matching score indicating the quality of the match in percentages.
- *threshold*, a variant indicating the false acceptance rate, or the tolerance for a match.
- *learn*, a boolean that returns true if learning on the features occurred during the match, which updates the registration template using the verification template to keep the template up-to-date and more accurate. This only occurs when the score is very high and the registration template was created with the learning capability.
- *sec*, returns the value of the SecureMode property, as described in "Evaluating the SecureMode Property" on page 24.

regtemplate is the registration template object. *vertemplate* is assumed to contain the verification template, which is acquired as described in step 2 on page 8. An instance of the FPVerify object (*verify*) is created and the Compare method is called to perform the match for *regtemplate* and *vertemplate*.

Exporting a Gold Template

A Gold template is a fingerprint template (registration or verification template) in the format used by the DigitalPersona Gold SDK and other DigitalPersona products. The Platinum template is simply a superset of the Gold template, with a Platinum header and signature added to the raw data stored in the Gold template.

You can retrieve the raw ‘Gold’ data using the TempIData property of the FPTemplate object.

```
 TempIData([out,retval] VARIANT *pVal)
```

The Gold template is returned in a SAFEARRAY of Variants that can then be saved to a file or in a database. See also: “FPTemplate” on page 55.

Operations Layer

Similiar to the Engine Layer, the Operations Layer allows you to facilitate the fingerprint recognition operation. The programmer, however, is shielded from much of the details. You only need to decide which process you want to perform: registration or verification. Then, you write event handlers for the events generated by these processes to control them and provide user feedback. As a result, writing all applications with the Operations Layer is much simpler and faster than with the Engine Layer, although you have less control over the other aspects of the operation.

This section shows you how to initiate the registration and verification processes using the Operations Layer and is followed by sample code (written in Visual Basic) that provides examples of handling events generated by these processes.

Registering a Fingerprint Template

You can start the fingerprint template registration process in three steps with the Operations Layer.

To start the registration process

1 Create an instance of the FPRegisterTemplate object.
2 Connect it to its event interface.

3 Call the Run method of the FPRegisterTemplate object to start the registration process.

The following sample code shows this process:

```
Dim WithEvents op As FPRegisterTemplate
Set op = New FPRegisterTemplate
op.Run
```

In the code, an instance of FPRegisterTemplate (*op*) is created and connected to its event interface. Then, the Run method is called and the registration process is initiated.

Handling Events from the Registration Process

While the Operations Layer does not enable you to control the entire registration process—such as raw sample-to-sample conversion, etc.—you can handle the events it generates. This allows you to supply the user with various forms of feedback, as well as make programming decisions to affect the registration process.

This section provides event handler code samples for the following events:

- SampleReady, fired when a sample is acquired from the reader
- SampleQuality, fired when feature extraction on the sample is complete
- Done event of the FPRegisterTemplate component, fired when registration is complete
- DevConnected and DevDisconnected events, fired when a device is connected or disconnected from the PC

Displaying the Sample Scan

When a sample is acquired by the reader, the SampleReady event is fired. The following sample code displays the scan of the acquired sample:

```
Private Sub op_SampleReady(ByVal pSample As Object)
  pSample.PictureOrientation = Or_Portrait
  pSample.PictureWidth = picSample.Width /
    Screen.TwipsPerPixelX
  pSample.PictureHeight = picSample.Height /
```

```
    Screen.TwipsPerPixelY
   picSample.Picture = pSample.Picture
   lblEvents.Caption = "Sample ready"
 End Sub
```

The sample code receives the sample acquired by the reader as a FPSample object, which contains various methods to process and display the scan. These methods are also used in “Decrypting and Decompressing the Raw Sample” on page 10. *lblEvents* is a label object, presumably created prior to running the event handler, which displays “Sample ready” when the event handler is run.

Evaluating Sample Quality

You can evaluate the quality of an acquired sample by writing an event handler for the SampleQuality event. The sample code below is a handler for the Sample Quality event and displays the quality of the sample passed to it:

```
 Private Sub op_SampleQuality(ByVal Quality As
   DpSdkEngLib.AISampleQuality)
   Select Case Quality
     Case AISampleQuality.Sq_Good
       lblQuality.Caption = "Good"
     Case AISampleQuality.Sq_LowContrast
       lblQuality.Caption = "Low contrast"
     Case AISampleQuality.Sq_NoCentralRegion
       lblQuality.Caption = "No central region"
     Case AISampleQuality.Sq_None
       lblQuality.Caption = "No quality info"
     Case AISampleQuality.Sq_NotEnoughFtr
       lblQuality.Caption = "Not enough features"
     Case AISampleQuality.Sq_TooDark
       lblQuality.Caption = "Too dark"
     Case AISampleQuality.Sq_TooLight
       lblQuality.Caption = "Too light"
     Case AISampleQuality.Sq_TooNoisy
       lblQuality.Caption = "Too noisy"
   End Select
   lblEvents.Caption = "Sample quality"
```

```
End Sub
```

The event handler receives the quality rating of the sample in the variable *Quality*. A case statement is used to display the quality of the sample in a label object, *lblQuality*, according to the value in *Quality*. *lblEvents*, another label, displays “Sample quality” when the event handler is run.

Detecting a Completed Registration

When registration is complete, the Done event of the FPRegisterTemplate component is fired and the final registration template is passed as a parameter to the event handler:

```
Private Sub op_Done(ByVal pTemplate As Object)
 lblEvents.Caption = “Done”
 Set regtemplate = pTemplate
End Sub
```

The registration template is saved to a global variable for subsequent use during the verification process. In a real-world scenario, you would save the template to the database or to a file.

Detecting Reader Connection Changes

To provide feedback to the user when a reader is connected to or disconnected from, two events can be handled:

• When a reader is connected to the PC, the DevConnected event is fired:

```
Private Sub op_DevConnected()
 lblEvents.Caption = “Device connected”
End Sub
```

• When a reader is disconnected from the PC, the DevDisconnected event is fired:

```
Private Sub op_DevDisconnected()
 lblEvents.Caption = “Device disconnected”
End Sub
```

The sample code displays the nature of the event in a label, *lblEvents*. Programmers can substitute this code with other methods to notify users of these events.

Verifying a Fingerprint Template

The following sample code initiates the verification process using the Operations Layer. It assumes that you have acquired the registration template from the database, a file or just acquired it and stored it in the global variable, *regtemplate*:

```
Dim WithEvents op As FPVerifyTemplate
Set op = New FPVerifyTemplate
op.Run regtemplate
```

In the sample code, an instance of FPVerifyTemplate (*op*) is created and connected to its event interface. Then, the Run method is called, passing the vari- able containing the registration template (*regtemplate*) to it, and the verification process is initiated.

Handling Events from the Verification Process

Similiar to the registration process, you cannot use the Operations Layer to control each step in the verification process, fired when verification is complete. You can, however, handle events generated by it to provide various forms of feedback to the user.

This section provides event handler sample code for the Done event of the FPVerifyTemplate component. Event handlers for the following events are identical to those of the registration process:

- SampleReady, fired when a sample is acquired from the reader, described in “Displaying the Sample Scan” on page 22.
- SampleQuality, fired when feature extraction on the sample is complete, described in “Evaluating Sample Quality” on page 23.

- DevConnected and DevDisconnected events, fired when a device is connected or disconnected from the PC, described in “Detecting Reader Connection Changes” on page 24.

Detecting a Completed Verification

When verification is complete, the Done event of the FPVerifyTemplate object is fired. The following sample code displays all the values acquired when the veri- fication process is complete:

```
Private Sub op_Done(ByVal VerifyOK As Boolean, pInfo() As
  Variant, ByVal Val As DpSdkEngLib.AISecureModeMask)
  lblEvents.Caption = “Done”
  lblResult.Caption = Format(VerifyOK)
  lblScore.Caption = CStr(pInfo(0))
  lblThreshold.Caption = CStr(pInfo(1))
  lblLearning.Caption = Format(CBool(pInfo(2)))
End Sub
```

The sample code contains several label objects that display the parameter values generated when the verification process is complete. Following is a description of each parameter:

- VerifyOK returns True if there is a match between the registration and verification templates
- pInfo(0) returns the matching score
- pInfo(1) returns the matching threshold
- pInfo(2) returns True if learning has occurred
- Val returns the SecureMode property value

Note

For a description of score, threshold, learning and information about the level of security, refer to “Verifying a Template” on page 16.

Adding Security to the Fingerprint Recognition Operation

The Platinum SDK provides security mechanisms that prevent a sample or verification template from being used more than once for matching (known as a replay attack).

The FPRawSample, FPSample and FPTemplate objects contain two properties —SecureMode and Nonce—which are used to add security to the verification process.

Evaluating the SecureMode Property

The SecureMode property of FPRawSample, FPSample and FPTemplate is used to evaluate the level of security applied to the verification process, allowing you to determine whether adequate security measures were in place during the verifi- cation process.

When acquiring a raw sample, converting to a sample or performing feature extraction, the SecureMode property will return any combination of the following values:

- Sm_None indicates that no security features were in place during the verification process.
- Sm_DevNonce indicates that the nonce was created and embedded in the raw sample object by the fingerprint recognition device. It is only returned when the nonce is embedded in a FPRawSample object.
- Sm_DevSignature indicates that the raw sample data was signed by the fingerprint recognition device. This value is set by the device and cannot be changed.
- Sm_DevEncryption indicates that the raw sample data was encrypted by the fingerprint recognition device. This value is set by the device and cannot be changed.
- Sm_FakeFingerDetection is returned if the fingerprint recognition device is able to recognize fake fingerprints. This value is set by the device and cannot be changed.
- Sm_NonceNotVerified indicates the nonce was not verified. The object can still be used, but should be considered non-secure.

• Sm_SignatureNotVerified indicates that the signature of the data object was not verified on import. The object can still be used, but should be considered non-secure.

Note

When Sm_NonceNotVerified and Sm_SignatureNotVerified is returned, it does not necessarily indicate that the object is corrupt or altered; rather, that the object was created on a system where a trust relationship could not be established.

Using a Nonce

A randomly-generated number, or nonce, is used to ensure that when a FPRaw-Sample, FPSample or FPTemplate object is processed, i.e., feature extraction, etc., the return object can be trusted.

A nonce is generated using the GenerateNonce method of the DPDataSecurity component and is set in a processing object using the SetNonce method. When the object is processed, the SecureMode property can be evaluated to determine if the returned object can be trusted. If Sm_NonceNotVerified is returned, the nonce could not be verified in the return object.

The following sample code demonstrates the use of a nonce in the feature extraction process:

```
Dim noncemgr As DPDataSecurity
Dim mynonce As Long
Dim ftrex As FPFtrEx
Dim qt As AISampleQuality
Dim pTemplate As FPTemplate

Set noncemgr = New DPDataSecurity

mynonce = noncemgr.GenerateNonce

Set ftrex = New FPFtrEx

ftrex.SetNonce mynonce
ftrex.Process pSample, Tt_Verification, pTemplate, qt
```

The code extracts features from a sample, as described in “Create a Template” on page 11, and creates a DPDataSecurity object, *noncemgr*. The GenerateNonce method is used to generate a nonce, *mynonce*. Using the Set-Nonce method and passing the nonce as an argument, the nonce is embedded in the feature extraction object. The Process method of the feature extraction object is used to create a verification template.

Assuming the verification template was saved to a file, the following sample code retrieves it and evaluates its SecureMode property to determine if the nonce can be verified:

```
Dim res As AIErrors
Dim blob() As Byte
Dim template As FPTemplate

Set template = New FPTemplate
Open “c:\template.fpt” For Binary As #1
ReDim blob(LOF(1))
Get #1, , blob()
Close #1

res = template.Import(blob)
  If res = Er_OK Then ‘ if import is successful
    if template.SecureMode And Sm_NonceNotVerified Then
      lblTemplateNonce.Caption = “Nonce not verified”
    Else
      lblTemplateNonce.Caption = “Nonce verified”
    End If
  Else ‘ if import is not successful
    lblTemplateImport.Caption = “Import failed”
End If
```

After loading the verification template from the file, the SecureMode property of the FPTemplate object, *template*, is evaluated for Sm_NonceNotVerified. The result of the evaluation, as well as the result of the import, is displayed in text boxes (presumably created elsewhere in the application).

Managing User Data 4

The DigitalPersona Platinum SDK includes the User layer, which provides access to various user database functions, such as importing, exporting and modifying user fingerprint data. It is intended for programmers who are not using their own database to store information for registered users, but rather the database functionality provided by the User layer.

This chapter is comprised of specific database functions, e.g., opening the database, etc., and is accompanied by a description of the various components used to facilitate the functionality. Each description is further illustrated with sample code in Visual Basic.

About the Database

There are three pieces of information in a user record in the Digital Persona-supplied database: the user name, user ID and the set of credentials, both password and fingerprints.

Note

All users must be Microsoft Windows users. All features of Active Directory are supported. If your application is deployed in a network using DigitalPersona Pro Server, the user data is stored in Active Directory. Refer to the *DigitalPersona Pro for Active Directory dministrator Guide* for more information.

Note

The fingerprint credential is referenced by a number, ranging 0 (left pinkie) through 9 (right pinkie).

Opening the Database

Before performing any database functions, you must first open the database.

Note

It is important to open the database on the same domain that is used in the SetHost method of operations. Do not mix local users and remote users in the same operation. Use the same domain name in SetHost for operations that was used to open the database.

To open the database using the User layer

1 Create an instance of the DPUsersDB object.

2 Use the OpenSystemDB method to open the database.

The following sample code shows how to use the User Layer to open the user database:

```
Dim db As DPUsersDB
Set db = New DPUsersDB

db.OpenSystemDB "[Domain name or local computer]"
```

In the sample code, a variable is created, *db*, which is then used to create an instance of DPUsersDB. Then, the OpenSystemDB method is used to open the database, located on the computer specified by its network name.

Enumerating Users in the Database

You can obtain a list of user names and IDs in the database using the User Layer to enumerate each user.

To enumerate users in the database using the User Layer

1 Open the database, as described in “Opening the Database” on page 30.

2 Use the AddItem method of the DPUser component to acquire the name or ID of user and add it to a container, such as a listbox.

The following sample code opens the database and shows how to add the name and ID of each user to listboxes:

```
Dim db As DPUsersDB
Dim pUser As DPUser
Set db = New DPUsersDB

db.OpenSystemDB "[Domain name or local computer]"

'add all the users in db into a listbox
For Each pUser In db
      lstUsers.AddItem pUser.name
      lstUserIDs.AddItem pUser.UserID
Next
```

In the code, the database is opened by creating an instance of DPUsersDB, *db*. In addition, since properties from a DPUser object are required to get user data, the variable, *pUser,* is created. A For Each loop is then used to loop the number of users in the database, *db,* while *pUser* provides access to the data in each record.

List boxes, *lstUsers*, will hold the user names, and *lstUserIDs*, will hold the user IDs, and, presumably, have been created beforehand. The AddItem method adds the user name and ID to the corresponding list box using the name and UserID properties of the DPUser object.

Finding a User Name in the Database

With the User Layer, you can retrieve a user by name and gain access to its record with read/write permissions, as well as create a pointer to it, by referencing the user name.

To find a user name in the database

1 Open the database, as described in "Opening the Database" on page 30.
2 Create a variable from the DPUser object to hold the pointer reference to the record.
3 Use the FindByName method of the DPUsersDB object to locate the record by its user name, require its read/write permissions and obtain a pointer reference to it.

The following sample code opens the database and queries a record by its user name:

```
Dim db As DPUsersDB
Dim pUser As DPUser
Set db = New DPUsersDB

db.OpenSystemDB "[Domain name or local computer]"

db.FindByName "JohnSmith", False, pUser
```

In the code, the database (*db*) is opened and a variable is created from the DPUser object, *pUser*, which will contain the pointer reference to the record. The FindByName method of the DPUserDB object is called with three parameters:

- The fictitious parameter, "*JohnSmith*," is the user name of the record.
- *False* indicates the record cannot be modified; otherwise, *True* would be used.
- *pUser* is the variable containing the pointer reference to the record.

Enumerating User Credentials

The User Layer allows you to retrieve all the credentials for a specific user from the database. You can then compare the credentials to the verification template for matching or obtain specific properties of the credentials for other purposes.

To get credentials for a specific user

1 Open the database, as described in "Opening the Database" on page 30.
2 Find a user name in the database, as described in "Finding a User Name in the Database" on page 32.
3 Create an instance of the DPUserCredentials object, setting its value to the Credentials property of the DPUser object.

The following sample code shows how to get the credentials registered for a specific user and list each one in a listbox:

```
Dim creds As DPUserCredentials
Dim pUser As DPUser
Dim db As DPUsersDB
Dim pTempl As FPTemplate

Set db = New DPUsersDB

db.OpenSystemDB "[Domain name or local computer]"

db.FindByName "JohnSmith", False, pUser

Set creds = pUser.Credentials(Cr_Fingerprint)
```

```
For Each pTempl In creds
  lstCredentials.AddItem pTempl.InstanceID
Next

lblCredCount.Caption = Format(creds.Count)
```

In addition to the two variables required for opening the database and finding a user name, the sample code creates two additional variables:

- *creds,* the DPUserCredentials object that contains a reference to the credentials for each user.
- *pTempl*, the FPTemplate object that contains reference to the fingerprint template.

The sample code then opens the database, finds a user by their user name and stores the reference in *pUser.*

Then, the value of *creds* is set to the enumerator of the user's fingerprint credentials using the Credentials method of the DPUser object (*pUser*). The Cr_Fingerprint parameter, which is passed to the Credentials method, indicates the credentials returned should be fingerprint templates.

A For Each statement loops the number of fingerprint credentials referenced in *creds* and stores the referenced fingerprint template in *pTempl*. The line of code in the loop adds the instance ID of the fingerprint template (using the InstanceID method of the FPTemplate object) to a listbox, *lstCredentials*. After the loop is complete, the Count method of the DPUserCredentials object*, creds*, sets the value of a label, *lblCredCount*, to the number of credentials returned.

Exporting a User Record to a File

This section describes how to export a user record to a file using the User Layer.

To export a user record to a file

1 Open the database, as described in “Opening the Database” on page 30.
2 Find a user name in the database, as described in “Finding a User Name in the Database” on page 32, to store a pointer reference to the record in a DPUser object.
3 Use the Export method of the DPUser object to export the record as a data blob.
4 Write the data blob to a file.

The following sample codes demonstrates a way to export a user record to a file:

```
Dim db As DPUsersDB
Dim pUser As DPUser
Dim vblob As Variant
Dim blob() As Byte

Set db = New DPUsersDB

db.OpenSystemDB "[Domain name or local computer]"
db.FindByName "JohnSmith", False, pUser

pUser.Export vblob
blob = vblob
Open "c:\userrec.blb" For Binary As #1
Put #1, , blob
Close #1
```

The sample code opens the database, finds a user record by its user name and stores the reference in *pUser*. The Export method of the DPUser object, *pUser*, exports the user record data to a blob, *vblob*. The data in v*blob* is assigned to the byte array, *blob*, which is then written to a file.

Importing a User Record from a File

This section describes how to import a user record from a file that was previously exported, as described in “Exporting a User Record to a File” on page 35. With the User Layer, you can import a user record, specify its read/write permission and obtain a pointer reference to the record.

To import a user record from a file

1 Open the database, as described in “Opening the Database” on page 30.
2 Load the record from a file and store the data in a data blob.
3 Use the ImportUser method of the DPUsersDB to import the record into the database, as well as specify its read/write permission and obtain a pointer reference to it.

The following sample code imports a user record from a file:

```
Dim db As DPUsersDB
Dim pUser As DPUser
Dim blob() As Byte

Set db = New DPUsersDB

db.OpenSystemDB "[Domain name or local computer]"

Open "c:\userrec.blb" For Binary As #1
ReDim blob(LOF(1))
Get #1, , blob()
Close #1

db.ImportUser blob, False, pUser
```

After opening the database, the sample code reads the blob of data from the file and stores it in the byte array, *blob*. The ImportUser method of the DPUsersDB object, *db*, imports the user record into the database. The parameter, False, indicates that the record cannot be modified and the pointer reference to the record is stored in the DPUser object, *pUser*.

Removing a User Record from the Database

The User Layer provides a way to remove user records from the database.

To remove a user record from the database

1 Open the database, as described in “Opening the Database” on page 30.

2 Pass the user name to the RemoveByName method of the DPUsersDB object to remove the user record from the database.

Following is sample code that removed a user record from the database:

```
Dim db As DPUsersDB

Set db = New DPUsersDB

db.OpenSystemDB "[Domain name or local computer]"

db.RemoveByName "[user name]"
```

The sample code opens the database, finds a user record by its username and stores the pointer reference in a DPUser object, *pUser.* The Remove method of the DPUsersDB object, *db*, is used to remove the record specified by the UserID property of the DPUser object, *pUser.*

Registering Fingerprints for a User in the Database

You can add or replace registration templates for an existing user in the database using the FPRegisterUser object of the User Layer. The first step in this process is to initiate the registration process.

To initiate the registration process

1 Open the database, as described in “Opening the Database” on page 30.

2 Find a user name in the database, as described in “Finding a User Name in the Database” on page 32, to store a pointer reference to the record in a DPUser object and set the write permissions to true.

Note

Unlike the other examples in this chapter, ensure you set the write permissions to true or you will not be able to add a registration template to the user record.

3 Create an instance of the FPRegisterUser component and connect it to its event interface.
4 Use the Run method of the FPRegisterUser component to begin the registration process.

The following sample code initiates the registration process for an existing user in the database:

```
Dim WithEvents op As FPRegisterUser
Dim db As DPUsersDB
Dim pUser As DPUser

Set op = New FPRegisterUser
Set db = New DPUsersDB

db.OpenSystemDB "[Domain name or local computer]"
db.FindByName "JohnSmith", True, pUser

op.Run pUser
```

After opening the database, a reference to an existing user is stored in the DPUsersDB object, *pUser*, using the FindByName method. The second parameter, which determines the read/write permission for a record, is set to True. This allows registration templates to be added to or replaced in the user record. Then, the Run method of the FPRegisterUser object initiates the registration process for the user specified in *pUser*.

Handling Events from the Registration Process

Several events are fired during the registration process which allow you to make decisions about how your application will function and provide user feedback.

Identifying Registered Fingerprints

When the registration process is initiated, the RegisteredFingers event is fired, allowing you to determine which fingers are registered for a particular user.

The following sample code uses the bitmask passed to the RegisteredFingers event handler to determine which fingers are registered. It assumes there is a label created for each finger and, if the code determines a finger has a registered fingerprint, the border and font properties of the corresponding label is changed:

```
Private Sub op_RegisteredFingers(ByVal FingerMask As
  Long)
  Dim mask As Long
  Dim i As Integer

  mask = 1

  For i = 0 to 9
    If mask and FingerMask Then
      finger(i).ForeColor = &HFF
      finger(i).FontBold = True
    Else
      finger(i).ForeColor = 0
      finger(i).FontBold = False
    End If
    mask = mask * 2
  Next i
  lblEvents.Caption = "Registered fingers"
End Sub
```

The sample code contains a For statement that loops through the entire range of fingers. It contains an If statement that evaluates whether the current finger is registered by performing a logical AND operation on the value of *mask* and the bitmask (*FingerMask*) passed to the event handler. If the finger is registered, the font and border style of the corresponding finger label is changed. *mask* is then incremented (*mask* * 2) for comparison with *FingerMask* on the next pass of the For loop.

Starting the Registration Template Process

Use the following line of code to start the registration process and indicate which finger the new registration template is associated with:

```
op.RegisterFinger [finger index]
```

The sample code uses the RegisterFinger method of the FPRegisterUser object, *op*, and passes the value of the finger index, ranging from 0 to 9. The index starts with the left pinkie finger (index 0); index 9 represents the right pinkie finger.

When a pre-registration template is acquired, the op_SampleReady and op_SampleQuality events are fired, as described in “Displaying the Sample Scan” on page 22 and “Evaluating Sample Quality” on page 23, respectively.

User Feedback for a Completed Registration

When the registration process is complete, the FingerRegistered event is fired. With a handler for this event, you can get the index of the newly registered fingerprint and provide feedback to the user.

The following sample code is an event handler for the FingerRegistered event. It indicates to the user which finger was registered by changing the border and font style of the corresponding finger label, which were introduced in “Identifying Registered Fingerprints” on page 39.

```
Private Sub op_FingerRegistered(ByVal fingerID As
  DpSdkUsrLibCtl.AIFingers)

  Dim i As Integers

  finger(fingerId).ForeColor = &HFF
  finger(fingerId).FontBold = True
  finger(fingerID).BorderStyle = 0
  lblEvents.Caption = "Finger " + Format(fingerId) + " has
    been registered"
End Sub
```

The sample code accepts the registered finger index, *fingerId*, and uses it to reference the corresponding finger label, *finger()*, to change its style properties.

Deleting a Registration Template

Instead of registering a fingerprint when the registration process is initiated, you can delete a fingerprint. You can use the same code in "Identifying Registered Fingerprints" on page 39 to determine which fingers are already registered. Then, you can use the following line of code to delete a registered fingerprint:

```
op.DeleteFinger [finger index]
```

The sample code uses the DeleteFinger method of the FPRegisterUser object, *op*, and passes the value of the finger index, ranging from 0 to 9. The index starts with the left pinkie finger (index 0); index 9 represents the right pinkie finger.

User Feedback for a Deleted Fingerprint

When a registered fingerprint is deleted, the FingerDeleted event is fired. In the event handler, you can determine which fingerprint was deleted and update the user interface accordingly:

```
Private Sub op_FingerDeleted(ByVal fingerId As
  DpSdkUsrLibCtl.AIFingers)
  finger(fingerId).ForeColor = 0
  finger(fingerId).FontBold = False
  finger(fingerId).BorderStyle = 0
  lblEvents.Caption = "Finger " + Format(fingerId) + " has
    been deleted"
End Sub
```

The code accepts the finger index of the deleted fingerprint in *fingerID*. It is used as an index reference for the corresponding finger label, *finger()*, to change the style properties of the label, indicating to the user which fingerprint was deleted.

Saving Registration Data Changes

Whether a fingerprint was registered or deleted, you must use the following line of code to save the changes to the database:

```
pUser.Save
```

Using the Save method of the DPUser object, *pUser*, saves any changes made to the database.

Warning

If you do not perform this step, the changes made to the user fingerprint data will not be stored in the database.

Verifying Fingerprints for a User in the Database

The User Layer enables you to match a verification template to a registration template stored in the database.

To match a verification template to a store template in the database

1 Open the database, as described in “Opening the Database” on page 30.
2 Find a user name in the database, as described in “Finding a User Name in the Database” on page 32, to store a pointer reference to the record in a DPUser object.
3 Create an instance of the FPVerifyUser component and connect it to its event interface.
4 Use the FingerMask property of the FPVerifyUser component to specify which fingerprints should be matched.
5 Use the Run method of the FPVerifyUser, passing the reference to the record in the database, to initiate the matching process.

The following sample code implements these steps:

```
Dim WithEvents op As FPVerifyUser
Dim db As DPUsersDB
Dim pUser As DPUser
Set op = New FPVerifyUser
Set db = New DPUsersDB
db.OpenSystemDB "[Domain name or local computer]"
db.FindByName "JohnSmith", False, pUser
op.FingerMask = -1
op.Run pUser
```

After opening the database and storing a reference to a specific user in a DPUser object (*pUser*), creates an instance of FPVerifyUser, *op*, and connects it to its event interface.

The FingerMask property is set to -1, which indicates that any registered fingerprint can be used for matching; otherwise, you would supply a hexadecimal value to indicate which fingers will be accepted for verification. Then, the Run method is called to initiate the matching process.

Handling Events from the Verification Process

The following sections identify several events fired during the verification template matching process. Similiar to events fired by the registration process, these events allow you to control application behavior, as well as provide user feedback.

Indicating the Fingers Required for Verification

You can indicate which fingers a user should supply for the verification process by determining the fingers your application requires for verification and the number of fingers registered for the user.

The following sample code determines which fingers are required for verification. To indicate these to the user, it assumes an array of 10 labels has been created—each corresponding to a finger—and changes the border style of a finger label if two conditions are met:

1 The fingerprint must be registered by the user.
2 The finger must be specified as a required finger by the FingerMask property, set in “Verifying Fingerprints for a User in the Database” on page 42.

```
Private Sub op_FingersToUse(ByVal FingerMask As Long)
  Dim mask As Long
  Dim i As Integer
  mask = 1
  For i = 0 to 9
    If mask And FingerMask Then
      finger(i).BorderStyle = 1
    Else
      finger(i).BorderStyle = 0
    End If
    mask = mask * 2
  Next i
lblEvents.Caption = "FingerToUse"
End Sub
```

The sample code contains a For statement that loops through the entire range of fi ngers. It contains an If statement that evaluates whether the current finger is both registered and required by performing a logical AND operation on the value of *mask* and the bitmask (*FingerMask*) passed to the event handler. If the finger is registered and required, the border style of the corresponding finger label is changed. *mask* is then incremented (*mask* * 2) for comparison with *FingerMask* on the next pass of the For loop.

Detecting an Acquired Sample and Feature Extraction

After the FingersToUse event is fired, two events that are identical to those of the FPRegisterTemplate and FPVerifyTemplate components are fired:

- op_SampleReady, fired when a sample is acquired from the reader, described in “Displaying the Sample Scan” on page 22
- op_SampleQuality, fired when feature extraction on the sample is complete, described in “Evaluating Sample Quality” on page 23

Done Event of the FPVerifyTemplate Component

When the verification is complete, the Done event is fired, allowing you to determine the result of the match, as well as obtain information about the match through various properties:

```
Private Sub op_Done(ByVal VerifyOk As Boolean, pInfo() As
  Variant, ByVal Val As DpSdkEngLib.AISecureModeMask)
  lblEvents.Caption = "Operation done"
  lblResult.Caption = Format(VerifyOk)
  lblScore.Caption = CStr(pInfo(0))
  lblThreshold.Caption = CStr(pInfo(1))
  lblLearning.Caption = Format(pInfo(2))
  lblFinger.Caption = Format(pInfo(3))
  lblUserID.Caption = pInfo(4)
End Sub
```

The sample code assumes a label has been created to display the value of each property. The VerifyOk property (boolean) returns true if there was a match between the verification template and a registration template. The pInfo(3) returns the index of the finger that matched and pInfo(4) returns the ID of the user the finger belongs to.

Note

The pInfo(0), pInfo(1) and pInfo(2) parameters are described in "Detecting a Completed Verification" on page 26.

SDK Reference 5

This chapter describes the functions, events, properties and error codes for each COM component and ActiveX control. Refer to "Data types" on page 107 for possible values and definitions.

FPDevices

A COM component that contains a collection of references to each fingerprint device connected to the PC. Use it to enumerate devices, reference a specific device by its serial number and monitor plug-n-play events.

Interface

`IFPDevices`

Methods

`Device([in]BSTR serNum, [out,retval]IDispatch **ppDev)`

Returns the device whose serial number is specified in *serNum* parameter. If the device is not connected, *ppDev* equals null.

Properties

`Count([out,retval] int *pCount)`

Returns the number of connected devices (read-only).

`Item([in] int devNum, [out,retval] IDispatch **ppDev)`

The standard Item property for collections. The value of *devNum* parameter starts at 1 (read-only).

`_NewEnum([out, retval] IUnknown **ppunkEnum)`

The standard _NewEnum property for the collections. It returns a pointer to an IEnumVARIANT interface (read-only).

Event interface

`_IFPDevicesEvents`

Event methods

`DeviceConnected([in] BSTR serNum)`

A device has been connected to the PC. The serial number is returned (*serNum*).

`DeviceDisconnected([in] BSTR serNum)`

A device with the serial number, *serNum*, was disconnected.

Return codes

None

Libraries

`DpSdkEng.dll`

FPDevice

A COM component that points to a reference in a FPDevices collection. It provides information on the device characteristics and an event interface to receive reader events, for example, when an scan is acquired.

Interface

`IFPDevice`

Methods

```
SubScribe([in] AIDevPriorities prio, [in] LONG hwnd
  [out,retval] AIErrors *pErr)
```

Register the calling application as a client for the device multiplexer with the specified priority in *prio*. If the hwnd parameter is null, the multiplexer will dispatch the events when any of the windows belonging to the calling process become active; otherwise, only when the given window is active. If the application does not call the SubScribe method, no events will be received.

`UnSubScribe([out,retval] AIErrors *pErr)`

Deregister the calling application as a client for the device.

`SetNonce([in] VARIANT nonce, [out,retval] AIErrors *pErr)`

Prepares a nonce to be embedded in a raw sample when it is processed.

```
SetParameter([in] LONG paramID, [in] VARIANT value,
  [out,retval] AIErrors *pErr)
```

Sets a device-specific parameter.

`GetParameter([in] LONG paramID, [in, out] VARIANT *pValue, [out,retval] AIErrors *pErr)`

Gets a device-specific parameter.

Properties

`Language([out, retval] LONG *pLanguage)`

Returns the language code for the device (read-only).

`Vendor([out, retval] BSTR *pVendor)`

Returns the string description of the device vendor (read-only).

`Product([out, retval] BSTR *pProduct)`

Returns the product name which depends on the type of reader connected to the PC (read-only).

`SerialNumber([out, retval] BSTR *pSerial)`

Returns the serial number for the device (read-only).

`HWRevision([out, retval] BSTR *pHWRev)`

Returns the hardware revision number of the device (read-only).

`FWRevision([out, retval] BSTR *pFWRev)`

Returns the firmware revision number of the device (read-only).

`Identifier([out, retval] BSTR *pDevId)`

Returns the device identifier, which varies when other USB devices are plugged into the PC (read-only).

`Type([out, retval] BSTR *pType)`

Returns code type for the device (read-only).

`SecurityCaps([out, retval] LONG *pSecCaps)`

Returns a bit mask describing the security capabilities of the device (read-only). The possible values are any combination of Sm_DevNonce, Sm_DevSignature, Sm_DevEncryption and Sm_FakeFingerDetection.

`NonceSize([out, retval] LONG *pSize)`

Returns the size (in bytes) of the nonce that the device can handle (read-only).

`DeviceCaps([out, retval] LONG *pDevCaps)`

Returns a bit mask with the device capabilities (read-only).

`ImageType([out, retval] AIImageType *pType)`

Returns the type of the fingerprint scan acquired by the reader (read-only).

`ImageWidth([out, retval] int *pWidth)`

Returns the width (in pixels) of the fingerprint scan acquired by the reader (read-only).

`ImageHeight([out, retval] int *pHeight)`

Returns the height (in pixels) of the fingerprint scan acquired by the reader (read-only).

`Xdpi([out, retval] LONG *pXdpi)`

Returns the resolution (dots per inch) on the X axis of the fingerprint scan (read-only).

`Ydpi([out, retval] LONG *pYdpi)`

Returns the resolution (dots per inch) on the Y axis of the fingerprint scan (read-only).

`BitsPerPixel([out, retval] LONG *pBpp)`

Returns the number of bits per pixel in the fingerprint scan (read-only).

`Padding([out, retval] AIImagePadding *pPad)`

Returns the alignment of the fingerprint scan (read-only).

`SignificantBpp([out, retval] LONG *pSBpp)`

Returns the number of significant bits per pixel in the fingerprint scan (read-only).

`Polarity([out, retval] AIPolarity *pPolarity)`

Returns the polarity of the scan produced by the reader (read-only).

`RGBMode([out, retval] AIRGBMode *pMode)`

Returns the color mode of the scan produced by the reader (read-only).

`Planes([out, retval] LONG *pPlanes)`

Returns the number of bit planes (read-only).

`SecureMode(AISecureModeMask)`

Sets/gets the secure mode for the device. See “AISecureModeMask” on page 108 for possible values.

Event interface

`_IFPDeviceEvents`

Event methods

`FingerTouching()`

The user touched the reader.

`FingerLeaving()`

The user lifted the finger from the reader.

`SampleAcquired(IDispatch *pRawSample)`

A fingerprint scan has been acquired and an FPRawSample object is returned.

`Error([in] AIErrors errcode)`

The device malfunctioned.

Return codes

Er_OK is returned if successful.

Er_InvalidArg is returned by:

- Subscribe method when the selected priority is not within the permitted ones.
- SetNonce method if the nonce size is not valid.

Er_DevBroken is returned by the Error event in case of a device malfunction.

Libraries

`DpSdkEng.dll`

FPRawSample

A COM component that contains the encrypted and compressed fingerprint scan acquired by the reader, called a raw sample. It is used by the FPSamplePro component to create a decrypted and decompressed sample (FPSample).

Interface

`IFPRawSample`

Methods

`Export([out] VARIANT* pVal, [out,retval] AIErrors *pErr)`

Exports the raw sample object as a signed blob of bytes.

`Import([in] VARIANT Val, [out,retval] AIErrors *pErr)`

Verifies the signature of a blob of bytes representing a raw sample object and imports it into a FPRawSample object.

Properties

`InstanceID([out,retval] BSTR* pVal)`

Returns the GUID of the object in string format (read-only).

`Version([out,retval] BSTR* pVal)`

Returns the version number of the object (read-only).

`TypeID([out,retval] AIDataTypes* pVal)`

Returns the type of the object, Dt_RawSample (read-only).

`CredType([out,retval] AICredentials* pVal)`

Returns the type of credential this object refers to, Cr_Fingerprint (read-only).

`VendorID([out,retval] BSTR* pVal)`

Returns the GUID for the vendor in string format (read-only).

`SecureMode([out, retval] AISecureModeMask *pVal)`

Returns the security mode of the object (read-only). See "AISecureModeMask" on page 108 for possible values.

`Nonce([out, retval] VARIANT *pVal)`

Returns the nonce contained in the object, if it exists; otherwise, an empty variant is returned (read-only).

Event interface
None

Event methods
None

Return codes
Er_OK is returned if successful.

Er_InvalidArg is returned by:

- Import method when the input parameter is not valid
- Export method when the parameter is not valid

Er_BadSignature is returned by the Import method when the signature is not verified.

Er_AlreadyCreated is returned by the Import method when the object contains data.

Libraries
`DpSdkEng.dll`

FPSample
The COM component that contains the sample object, produced from a FPRawSample object using the Convert method of the FPRawSamplePro object.

Interface
`IFPSample`

Methods
```
Export([out] VARIANT* pVal, [out, retval] AIErrors *pErr)
```

Exports the sample object as a signed blob of bytes.

```
Import([in] VARIANT Val, [out, retval] AIErrors *pErr)
```

Verifies the signature of a blob of bytes representing a sample object and imports it into a FPSample object.

Properties

`InstanceID([out,retval] BSTR* pVal)`

Returns the GUID of the object in string format (read-only).

`Version([out,retval] BSTR* pVal)`

Returns the version number of the object (read-only).

`TypeID([out,retval] AIDataTypes* pVal)`

Returns the type of the object, Dt_Sample (read-only).

`CredType([out,retval] AICredentials* pVal)`

Returns the type of credential this object refers to, Cr_Fingerprint (read-only).

`VendorID([out,retval] BSTR* pVal)`

Returns the GUID for the vendor in string format (read-only).

`SecureMode([out, retval] AISecureModeMask *pVal)`

Returns the security mode of the object (read-only). See "AISecureModeMask" on page 108 for possible values.

`Nonce([out, retval] VARIANT *pVal)`

Returns the nonce contained in the object. If no nonce is present, an empty variant is returned (read-only).

`Width([out,retval] VARIANT *pVal)`

Returns the width of the fingerprint scan as it was acquired from the reader (in pixels) (read-only).

`Height([out,retval] VARIANT *pVal)`

Returns the height of the fingerprint scan as it was acquired from the reader (in pixels) (read-only).

`Orientation([out,retval] AIOrientation* pVal)`

Returns the orientation of the fingerprint scan as it was acquired from the reader (read-only).

`PictureWidth([out,retval] VARIANT *pVal)`

Sets/gets the new width (in pixels) of the fingerprint scan.

`PictureHeight([out,retval] VARIANT *pVal)`

Sets/gets the new height (in pixels) of the fingerprint scan.

`PictureOrientation([out,retval] AIOrientation* pVal)`

Sets/gets the new fingerprint scan orientation.

`Picture([out,retval] IDispatch** ppPicture)`

Returns a pointer to an IPicture interface (read-only).

Event interface

None

Event methods

None

Return codes

Er_OK is returned if successful.

Er_InvalidArg is returned by:

- Import method when the input parameter is not valid
- Export method when the parameter is not valid

Er_BadSignature is returned by:

- Import method when the signature is not verified

Er_AlreadyCreated is returned by:

- Import method when the object is not empty but already contains data

Libraries

`DpSdkEng.dll`

FPTemplate

A COM component containing a fingerprint template. The object is characterized by its type, either a preregistration, registration or verification template.

Interface

`IFPTemplate`

Methods

`Export([out] VARIANT* pVal, [out,retval] AIErrors *pErr)`

Exports the template object as a signed blob of bytes.

`Import([in] VARIANT Val)`

Evaluates the signature of a blob of bytes representing the signed template object and, if verified, imports the data into the FPTemplate object.

Properties

`InstanceID([out,retval] BSTR* pVal)`

Returns the GUID of the object in string format (read-only).

`Version([out,retval] BSTR* pVal)`

Returns the version number of the object (read-only).

`TypeID([out,retval] AIDataTypes* pVal)`

Returns the type of the object, Dt_Template (read-only).

`CredType([out,retval] AICredentials* pVal)`

Returns the type of credential this object refers to, Cr_Fingerprint (read-only).

`VendorID([out,retval] BSTR* pVal)`

Returns the GUID for the vendor in string format (read-only).

`SecureMode([out, retval] AISecureModeMask *pVal)`

Returns the security mode of the object (read-only). See "AISecureModeMask" on page 108 for possible values.

`Nonce([out, retval] VARIANT *pVal)`

Returns the nonce contained in the object. If no nonce is present, an empty variant is returned (read-only).

`TemplType([out,retval] AITemplateTypes* pVal)`

Returns the type of the template (read-only).

`TemplData([out,retval] VARIANT *pVal)`

Returns the raw template data, without the header and signature (read-only).

Event interface

None

Event methods

None

Return codes

Er_OK is returned if successful.

Er_InvalidArg is returned by:

- Import method when the input parameter is not valid
- Export method when the parameter is not valid

Er_BadSignature is returned by:

- Import method when the signature is not verified

Er_AlreadyCreated is returned by:

- Import method when the object is not empty but already contains data

Libraries

`DpSdkEng.dll`

DPObjectSecurity

A COM component used to generate a single-use, random number, or nonce, that can be embedded in the FPRawSample, FPSample and FPTemplate objects. When one of these objects contain a nonce, the nonce is signed with the object data and information. When the object is imported, the nonce is evaluated and, if it cannot be verified, the secureMode property of the object is set to Sm_NonceNotVerified.

Interface
`IDPObjectSecurity`

Methods
`GenerateNonce([out, retval] VARIANT* pNonce)`

Generates a nonce.

Properties
None

Event interface
None

Event methods
None

Return codes
None

Libraries
`DpSdkEng.dll`

FPRawSamplePro

A COM component that converts a raw sample into a sample by decrypting and decompressing the raw sample. A raw sample must be converted to a sample before features can be extracted from it to create a template.

Interface
`IFPRawSamplePro`

Methods
`SetHost([in]BSTR vHost, [out,retval] AIErrors *pErr)`

Selects the host on which the operation will be performed. vHost must be the name of an AD or NT domain. If not specified, the operation is performed on the local machine. This method is obsolete. If older programs call it, the conversion occurs on the local machine only.

`Convert([in]IDispatch *pRawSample, [out] IDispatch **ppSample, [out, retval] AIErrors *pErr)`

Converts a raw sample into a sample.

`SetNonce([in]VARIANT nonce, [out,retval] AIErrors *pErr)`

Prepares a nonce to be embedded in a sample when raw sample-to-sample conversion is performed.

Properties

`SecureMode(AISecureModeMask)`

Sets/gets the security mode for the raw sample-to-sample conversion. See "AISecureModeMask" on page 108 for possible values.

Event interface

None

Event methods

None

Return codes

Er_OK is returned if successful.

Er_InvalidArg is returned by:

- SetNonce method when the nonce is not valid
- Convert method when a FPRawSample object is not supplied as a parameter

Er_NoHost is returned by:

- SetHost method when a connection to the given host cannot be established

Er_System is returned by:

- Convert method when a system error occurs, e.g., installation problems, etc.

Libraries

`DpSdkEng.dll`

FPFtrEx

A COM component that performs the feature extraction on a sample, provided the quality of the sample is adequate.

Interface

`IFPFtrEx`

Methods

`SetHost([in]BSTR vHost, [out,retval] AIErrors *pErr)`

Selects the host on which the operation will be performed. vHost can be only the name of an AD or NT domain. If not specified, the operation is performed on the local machine. This method is obsolete. If older programs call it, all feature exrtactions are performed on the local machine only.

```
Process([in]IDispatch *pSample, [in] AITemplateTypes
  TemplType, [out] IDispatch **ppTemplate, [out]
  AISampleQuality **pQuality, [out, retval] AIErrors
  **pErr)
```

Extracts features from a sample and creates a template. The TemplType parameter specifies the type of template to produce, either preregistration (Tt_PreRegistration) or verification (Tt_Verification). If the quality of the sample is not adequate, the template is not created. The pQuality parameter indicates the quality rating of the sample.

`SetNonce([in]VARIANT nonce, [out, retval] AIErrors **pErr)`

Preapres the nonce to be embedded in the template object when it is created.

Properties

`SecureMode(AISecureModeMask)`

Sets/gets the security mode for the feature extraction process. See “AISecureModeMask” on page 108 for possible values.

Event interface

None

Event methods
None

Return codes
Er_OK is returned if successful.

Er_InvalidArg is returned by:

- SetNonce method when the nonce is not valid
- Process method when a FPSample object is not supplied as a parameter

Er_NoHost is returned by:

- SetHost method when a connection with the given host cannot be established

Er_System is returned by:

- Process method when a system error occurs, e.g., installation problems, etc.

Libraries
`DpSdkEng.dll`

FPRegister
A COM component that creates a registration template from a set of preregistration templates.

Interface
`IFPRegister`

Methods
```
SetHost([in]BSTR vHost, [out,retval] AIErrors *pErr)
```

Selects the host on which the operation will be performed. In the current implementation, vHost can be only the name of an AD or NT domain. If not specified, the operation is performed on the local machine.

```
NewRegistration([in] AIRegTargets Target, [out,retval]
  AIErrors *pErr)
```

Starts a new registration for the specified target. This version only supports Rt_Verify value for the Target parameter

`Add([in] IDispatch *pPreReg, [out] VARIANT_BOOL *pbDone, [out,retval] AIErrors *pErr)`

Add a preregistration template to the set. If the number of preregistration templates is adequate to produce a registration template, the pbDone parameter returns VARIANT_TRUE; otherwise, it is VARIANT_FALSE.

Properties

`SecureMode(AISecureModeMask)`

Sets/gets the security mode for the registration template. See "AISecureModeMask" on page 108 for possible values.

`Learning(VARIANT_BOOL)`

Sets/gets learning on the registration template.

`RegistrationTemplate([out, retval] IDispatch **ppReg)`

Returns the registration template created (read-only). This property must be read after the Add method returns true in the boolean variable, pdDone.

Interface

`IFPRegister2`

Derived from IFPRegister.

Properties

`UsingXTFTemplate(VARIANT_BOOL)`

When set to True, uses the XTF registration template. The XTF template requires more space but provides fewer false rejects during validation. For more information in using the XTF template, see "Using the XTF Registration Template" on page 17.

Event interface

None

Event methods

None

Return codes

Er_OK is returned if successful.

Er_InvalidArg is returned by:

- NewRegistration method when the target parameter is not valid
- Add method when a preregistration template is not supplied as a parameter

Er_NoHost is returned by:

- SetHost method when a connection to the given host cannot be established

Er_System is returned by:

- Add method when a system error occurs, e.g., installation problems, etc.
- NewRegistration method when a system error occurs, e.g., installation problems, etc.

Libraries

`DpSdkEng.dll`

FPVerify

A COM component that compares a verification template to a registration template for a match using its Compare method. The Compare method returns data regarding the match, such as the level of security, the score, etc.

Interface

`IFPVerify`

Methods

```
SetHost([in]BSTR vHost, [out,retval] AIErrors *pErr)
```

Selects the host on which the operation will be performed. In the current implementation, vHost can be only the name of an AD or NT domain. If not specified, the operation is performed on the local machine.

```
Compare([in] IDispatch *pRegTemplate, [in] IDispatch
  *pVerTemplate, [out] VARIANT_BOOL *pVerifyOk, [out]
  VARIANT *pScore, [out] VARIANT *pThreshold, [out]
  VARIANT_BOOL *pLearnDone, [out] AISecureModeMask
  *pSecurity, [out, retval] AIErrors *pErr)
```

Compares the two given templates and returns information about the match:

- pVerifyOk returns VARIANT_TRUE if a match occurs; otherwise, VARIANT_FALSE is returned
- pScore returns the matching score
- pThreshold returns the matching threshold
- pLearnDone returns VARIANT_TRUE if learning occurred on the features
- pSecurity returns the security mode based on the security mode of the two input templates

Properties

`SecureMode(AISecureModeMask)`

Sets/gets the security mode for matching process. See "AISecureModeMask" on page 108 for possible values.

`Learning(VARIANT_BOOL)`

Enables/disables the learning on the features if a successful matching occurs.

`SecurityLevel(VARIANT)`

Sets/gets the security level for the matching process. The security level is expressed as a probability of false acceptance/false reject.

Event interface

None

Event methods

None

Return codes

Er_OK is returned if successful.

Er_InvalidArg is returned by:

- Compare method when the input objects are not either a registration or verification template

Er_NoHost is returned by:

- SetHost method when a connection cannot be established with the given host

Er_System is returned by:

- Compare method when a system error occurs, e.g., installation problems, etc.

Er_RegFailed is returned by:

- Compare method when the registration template cannot be created, i.e., a set of samples was supplied from different fingers

Libraries

`DpSdkEng.dll`

FPGetSampleX

An ActiveX component that retrieves a fingerprint sample from the reader. It instantiates the components necessary for acquiring a raw sample and converting it to a sample.

Interface

`IFPGetSampleX`

Methods

`SetHost([in]BSTR vHost, [out,retval] AIErrors *pErr)`

Selects the host on which the operation will be performed. DigitalPersona Pro Server does not support samples processing. This method is ignored.

`CreateOp([out,retval] AIErrors *pErr)`

Used prior to the Run method, CreateOp instantiates and links all necessary components for the sample acquisition process.

`Run([out,retval] AIErrors *pErr)`

Runs the operation, which can be canceled any time. The Done event is fired when the operation terminates. While the operation is running, plug-n-play events are also fired. If the device is disconnected during the operation, the operation will pause until the device is reconnected.

`Cancel([out,retval] AIErrors *pErr)`

Stops the operation, discarding all intermediate data, if any.

`SetDevicePriority([in] AIDevPriorities Priority, [in] LONG hWnd, [out,retval] AIErrors *pErr)`

Select the device priority for the client. In most cases, it is strongly recommended to use the standard priority, Dp_StdPriority. The standard priority allows the system to direct reader events to the active client based on the window handle supplied through the hWnd parameter. Refer to "AIDevPriorities" on page 108 for a complete list of priorities.

`SelectDevice([in] BSTR serNum, [out,retval] AIErrors *pErr)`

Selects the device to be used for the operation. Calling this method is necessary if there is more than one device connected to the computer.

`SetNonce([in] VARIANT Val, [out,retval] AIErrors *pErr)`

Prepares a nonce to be embedded in a sample when raw sample-to-sample conversion is performed.

Properties

`SecureMode(AISecureModeMask)`

Sets/gets the security mode for the raw sample-to-sample conversion. See "AISecureModeMask" on page 108 for possible values.

Event interface

`_IFPGetSampleXEvents`

Event methods

`Done([in] IDispatch *pSample)`

Indicates the operation is complete. The pSample parameter points to the FPSample object generated.

`DevDisconnected()`

The device was disconnected during the operation. If a specific device was selected, the operation will resume after the same device is reconnected; otherwise, it is canceled and must be run again after reconnecting that same device.

`DevConnected()`

The device has been reconnected.

`Error([in] AIErrors errcode)`

An error occurred. The errcode parameter returns the error code, as described in “AIErrors” on page 109.

Return codes

Er_OK is returned if successful

Er_InvalidArg is returned by:

- SetDevicePriority method when the priority specified is not permitted
- SetNonce method when the given nonce is invalid

Er_NoHost is returned by:

- SetHost method when the given host cannot be contacted

Er_System is returned by:

- SelectDevice method when a system error occurs (maybe due to installation problems)

Er_NoDevice is returned by:

- SelectDevice method when there are no devices connected
- Run method when either the selected device or the default one is not connected
- CreateOp method when either the selected device or the default one is not connected

Er_InvalidID is returned by:

- SelectDevice method when the requested serial number does not exist

Er_AlreadyCreated is returned by:

- SetHost method when called after the operation has been created

Er_OperNotRunning is returned by:

- Cancel method when called without having created the operation

Libraries

`DpSdkOps.dll`

FPGetTemplateX

An ActiveX component that retrieves a fingerprint sample from the reader. It instantiates the components necessary for acquiring a raw sample and converting it to a sample and extracting features from the sample to create a template.

Interface

`IFPGetTemplateX`

Methods

`SetHost([in]BSTR vHost, [out,retval] AIErrors *pErr)`

Selects the host on which the operation will be performed. DigitalPersona Pro Server does not support samples processing. This method is ignored.

`CreateOp([out,retval] AIErrors *pErr)`

Used prior to the Run method, CreateOp instantiates and links all necessary components for the sample acquisition and feature extraction process.

`Run([out,retval] AIErrors *pErr)`

Runs the operation, which can be canceled any time. The Done event is fired when the operation terminates. While the operation is running, plug-n-play events are also fired. If the device is disconnected during the operation, the operation will pause until the same device is reconnected.

`Cancel([out,retval] AIErrors *pErr)`

Stops the operation, discarding all intermediate data, if any.

```
SetDevicePriority([in] AIDevPriorities Priority, [in] LONG
  hWnd, [out,retval] AIErrors *pErr)
```

Select the device priority for the client. In most of the cases it is strongly recommended to use the device at the standard priority (Dp_StdPriority): this way the system can direct reader events to the active client based on the window handle supplied through the hWnd parameter. See the Data Types section for further details on the available priorities.

`SelectDevice([in] BSTR serNum, [out,retval] AIErrors *pErr)`

Selects the device to be used for the operation. Calling this method is necessary only if there is more than one reader connected to the computer.

`SetNonce([in] VARIANT Val, [out,retval] AIErrors *pErr)`

Sets a nonce to be included in the produced FPTemplate object.

Properties

`SecureMode(AISecureModeMask)`

Sets/gets the security mode for the processing.

`TemplateType(AITemplateTypes)`

Sets/gets the template type to be created. Permitted values are Tt_Verification and Tt_PreRegistration.

Event interface

_IFPGetTemplateXEvents

Event methods

`Done([in] IDispatch *pTemplate)`

The operation is complete and the pTemplate parameter points to the FPTemplate object generated.

`DevDisconnected()`

The device has been disconnected while the operation was running. If a specific device was selected the operation will resume after the same device is reconnected; otherwise, it is canceled and must be run again after reconnecting that same device.

`DevConnected()`

The device has been reconnected.

`Error([in] AIErrors errcode)`

An error occurred. The errcode parameter returns the error code, as described in “AIErrors” on page 109.

Return codes

Er_OK is returned if successful

Er_InvalidArg is returned by:

- SetDevicePriority method when the given priority is not within the permitted ones
- SetNonce method when the given nonce is invalid

Er_NoHost is returned by:

- SetHost method when the given host cannot be contacted

Er_System is returned by:

- SelectDevice method when a system error occurs, e.g., such as installation problems, etc.

Er_NoDevice is returned by:

- SelectDevice method when there are no devices connected
- Run method when either the selected device or the default one is not connected
- CreateOp method when either the selected device or the default one is not connected

Er_InvalidID is returned by:

- SelectDevice method when the requested serial number does not exist

Er_AlreadyCreated is returned by:

- SetHost method when called after the operation has been created

Er_OperNotRunning is returned by:

- Cancel method when called without having created the operation

Libraries

`DpSdkOps.dll`

FPRegisterTemplateX

An ActiveX component that allows the user to register a fingerprint. It instantiates the components necessary for acquiring a four raw samples and converting them to samples and extracting features from them to create a registration template.

Interface

`IFPRegisterTemplateX`

Methods

`SetHost([in]BSTR vHost, [out,retval] AIErrors *pErr)`

Selects the host on which the operation will be performed. In the current implementation, vHost can be only the name of an AD or NT domain. If not specified, the operation is performed on the local machine.

`CreateOp([out,retval] AIErrors *pErr)`

Used prior to the Run method, CreateOp instantiates and links all necessary components for the registration process.

`Run([out,retval] AIErrors *pErr)`

Runs the operation, which can be canceled any time. The Done event is fired when the operation terminates. While the operation is running, plug-n-play events are also fired. If the device is disconnected during the operation, the operation will pause until the same device is reconnected.

`Cancel([out,retval] AIErrors *pErr)`

Stops the operation, discarding all intermediate data, if any.

```
SetDevicePriority([in] AIDevPriorities Priority, [in] LONG
  hWnd, [out,retval] AIErrors *pErr)
```

Select the device priority for the client. In most of the cases it is strongly recommended to use the device at the standard priority (Dp_StdPriority): this way the system can direct reader events to the active client based on the window handle supplied through the hWnd parameter. See the Data Types section for further details on the available priorities.

`SelectDevice([in] BSTR serNum, [out,retval] AIErrors *pErr)`

Selects the device to be used for the operation. Calling this method is necessary only if there is more than one reader connected to the computer.

Properties

`SecureMode(AISecureModeMask)`

Sets/gets the security mode for the processing.

`Learning(VARIANT_BOOL)`

Sets/gets the learning for the registration template that will be produced.

`Target(AIRegTargets)`

Sets/gets the target for the registration. The only permitted value in this implementation is Rt_Verify.

Interface

`IFPRegisterTemplateX2`

Derived from IFPRegisterTemplateX.

Properties

`UsingXTFTemplate(VARIANT_BOOL)`

When set to True, uses the XTF registration template. The XTF template requires more space but provides fewer false rejects during validation. For more information in using the XTF template, see “Using the XTF Registration Template” on page 17.

Event interface

`_IFPRegisterTemplateXEvents`

Event methods

`Done([in] IDispatch *pRegTemplate)`

The operation is complete and the pRegTemplate parameter points to the FPTemplate object generated. The type of the object will be Tt_Registration.

`DevDisconnected()`

The device has been disconnected while the operation was running. If a specific device was selected, the operation will resume after the same device is reconnected; otherwise, it is canceled and must be run again after reconnecting that same device.

`DevConnected()`

The device has been reconnected.

`Error([in] AIErrors errcode)`

An error occurred. The errcode parameter returns the error code, as described in “AIErrors” on page 109.

Return codes

Er_OK is returned if successful

Er_InvalidArg is returned by:

- SetDevicePriority method when the given priority is not within the permitted ones

Er_NoHost is returned by:

- SetHost method when the given host cannot be contacted

Er_System is returned by:

- SelectDevice method when a system error occurs (maybe due to installation problems)

Er_NoDevice is returned by:

- SelectDevice method when there are no devices connected
- Run method when either the selected device or the default one is not connected
- CreateOp method when either the selected device or the default one is not connected

Er_InvalidID is returned by:

- SelectDevice method when the requested serial number does not exist

Er_AlreadyCreated is returned by:

- SetHost method when called after the operation has been created

Er_OperNotRunning is returned by:

- Cancel method when called without having created the operation

Er_RegFailer is returned:

- As a parameter of the Error event when the registration fails

Libraries

`DpSdkOps.dll`

FPVerifyTemplateX

An ActiveX component that facilitates the verification process. It instantiates the components necessary for acquiring a raw sample and converting it to a sample and extracting features from it to create a verification template. Then, it performs a match between the acquired verification template and a given registration template.

Interface

`IFPVerifyTemplateX`

Methods

`SetHost([in]BSTR vHost, [out,retval] AIErrors *pErr)`

Selects the host on which the operation will be performed. In the current implementation vHost can be only the name of an AD or NT domain. If not specified, the operation is performed on the local machine.

`CreateOp([out,retval] AIErrors *pErr)`

Used prior to the Run method, CreateOp instantiates and links all necessary components for the verification process.

`Run([in] VARIANT Val, [out,retval] AIErrors *pErr)`

Runs the operation, which can be canceled any time. The Done event is fired when the operation terminates. While the operation is running, plug-n-play events are also fired. If the device is disconnected during the operation, the operation will pause until the same device is reconnected. The parameter Val contains a pointer to the FPTemplate object of type Tt_Register for matching.

`Cancel([out,retval] AIErrors *pErr)`

Stops the operation, discarding all intermediate data, if any.

`SetDevicePriority([in] AIDevPriorities Priority, [in] LONG hWnd, [out,retval] AIErrors *pErr)`

Select the device priority for the client. In most of the cases it is strongly recommended to use the device at the standard priority (Dp_StdPriority): this way the system can direct reader events to the active client based on the window handle supplied through the hWnd parameter. See the Data Types section for further details on the available priorities.

`SelectDevice([in] BSTR serNum, [out,retval] AIErrors *pErr)`

Selects the device to be used for the operation. Calling this method is necessary only if there is more than one reader connected to the computer.

Properties

`SecureMode(AISecureModeMask)`

Sets/gets the security mode for the processing.

`Learning(VARIANT_BOOL)`

Sets/gets the learning mode. If the given registration template was created with the learning feature, the system will attempt to perform the learning on the features.

`SecurityLevel(VARIANT)`

Sets/gets the security level for the matching process. The security level is expressed as probability of false acceptance/false rejectance.

Event interface

`_IFPVerifyTemplateXEvents`

Event methods

`Done([in] VARIANT_BOOL VerifyOk, [in] SAFEARRAY(VARIANT)* pInfo, [in] AISecureModeMask Val)`

The operation is complete and the VerifyOk parameter contains the boolean result of the comparison. The pInfo parameter contains three elements: a score, the used threshold and a Boolean value to indicate whether the learning has occurred. The last parameter gives the security mode based on the requested mode and the mode of the given registration template.

`DevDisconnected()`

The device has been disconnected while the operation was running. If a specific device was selected, the operation will resume after the same device is reconnected; otherwise, it is canceled and must be run again after reconnecting that same device.

`DevConnected()`

The device has been reconnected.

`Error([in] AIErrors errcode)`

An error occurred. The errcode parameter returns the error code, as described in “AIErrors” on page 109.

Return codes

Er_OK is returned if successful

Er_InvalidArg is returned by:

- SetDevicePriority method when the given priority is not within the permitted ones

Er_NoHost is returned by:

- SetHost method when the given host cannot be contacted

Er_System is returned by:

- SelectDevice method when a system error occurs (maybe due to installation problems)

Er_NoDevice is returned by:

- SelectDevice method when there are no devices connected
- Run method when either the selected device or the default one is not connected

- CreateOp method when either the selected device or the default one is not connected

Er_InvalidID is returned by:

- SelectDevice method when the requested serial number does not exist

Er_AlreadyCreated is returned by:

- SetHost method when called after the operation has been created

Er_OperNotRunning is returned by:

- Cancel method when called without having created the operation

Libraries

`DpSdkOps.dll`

FPRegisterUserX

An ActiveX component that facilitates the registration process for a given user. It starts the registration process for a given finger, as well as removes the fingerprint template associated with a given finger.

Interface

`IFPRegisterUserX`

Methods

`SetHost([in]BSTR vHost, [out,retval] AIErrors *pErr)`

Selects the host on which the operation will be performed. In the current implementation, vHost can be only the name of an AD or NT domain. If not specified, the operation is performed on the local machine.

`CreateOp([out,retval] AIErrors *pErr)`

Used prior to the Run method, CreateOp instantiates and links all necessary components for the user registration process.

`Run([in] VARIANT Val, [out,retval] AIErrors *pErr)`

Runs the operation, which can be canceled any time. The Done event is fired when the operation terminates. The Val parameter contains a pointer to the user record. The operation never terminates unless an error occurs; however, during the operation, events are fired when a new fingerprint is successfully registered or deleted. The developer can decide to update the user record in the file or database at any time.

`Cancel([out,retval] AIErrors *pErr)`

Stops the operation, discarding all intermediate data, if any.

`SetDevicePriority([in] AIDevPriorities Priority, [in] LONG hWnd, [out,retval] AIErrors *pErr)`

Select the device priority for the client. In most cases, it is strongly recommended to use the standard priority, Dp_StdPriority. The standard priority allows the system to direct reader events to the active client based on the window handle supplied through the hWnd parameter. Refer to "AIDevPriorities" on page 108 for a complete list of priorities.

`SelectDevice([in] BSTR serNum, [out,retval] AIErrors *pErr)`

Selects the device to be used for the operation. Calling this method is necessary only if there is more than one reader connected to the computer.

Properties

`SecureMode(AISecureModeMask)`

Sets/gets the security mode for the processing.

`Learning(VARIANT_BOOL)`

Sets/gets the learning for the registration template(s) that will be produced.

`Target(AIRegTargets)`

Sets/gets the target for the registration. The only permitted value in this implementation is Rt_Verify.

Interface

`IFPRegisterUserX2`

Derived from IFPRegisterUserX.

Properties

`UsingXTFTemplate(VARIANT_BOOL)`

When set to True, uses the XTF registration template. The XTF template requires more space but provides fewer false rejects during validation. For more information in using the XTF template, see “Using the XTF Registration Template” on page 17.

Event interface

`_IFPRegisterUserXEvents`

Event methods

`FingerRegistered([in] AIFingers fingerId)`

The registration of a fingerprint is complete and the fingerId parameter returns the identifier of the modified finger. AIFingers values are listed in “AIFingers” on page 109.

`FingerDeleted([in] AIFingers fingerId)`

The cancellation of a finger is complete and the fingerId parameter returns the identifier of the deleted finger. AIFingers values are listed in “AIFingers” on page 109.

`DevDisconnected()`

The device has been disconnected while the operation was running. If a specific device was selected, the operation will resume after the same device is reconnected; otherwise, it is canceled and must be run again after reconnecting that same device.

`DevConnected()`

The device has been reconnected.

`Error([in] AIErrors errcode)`

An error occurred. The errcode parameter returns the error code, as described in “AIErrors” on page 109.

Return codes

None

Libraries

`DpSdkUsr.dll`

FPVerifyUserX

An ActiveX component that facilitates the verification process for a given user. It starts the verification process, using the specified fingers.

Interface

`IFPVerifyUserX`

Methods

`SetHost([in]BSTR vHost, [out,retval] AIErrors *pErr)`

Selects the host on which the operation will be performed. In the current implementation vHost can be only the name of an AD or NT domain. If not specified, the operation is performed on the local machine.

`CreateOp([out,retval] AIErrors *pErr)`

Used prior to the Run method, CreateOp instantiates and links all necessary components for the user verification process.

`Run([in] VARIANT Val, [out,retval] AIErrors *pErr)`

Runs the operation, which can be canceled any time. The Val parameter is a pointer to the record of the user to be verified. The Val parameter can also be a pointer to an IEnumVARIANT interface, i.e., an enumerator of users. In this case, the first matching user is used.

Note

This is not an identification function and it can only be used on a very small set of users (less than 10).

The Done event is fired when the operation terminates and results of the match are returned.

`Cancel([out,retval] AIErrors *pErr)`

Stops the operation, discarding all intermediate data, if any.

`SetDevicePriority([in] AIDevPriorities Priority, [in] LONG hWnd, [out,retval] AIErrors *pErr)`

Select the device priority for the client. In most cases, it is strongly recommended to use the standard priority, Dp_StdPriority. The standard priority allows the system to direct reader events to the active client based on the window handle supplied through the hWnd parameter. Refer to “AIDevPriorities” on page 108 for a complete list of priorities.

`SelectDevice([in] BSTR serNum, [out,retval] AIErrors *pErr)`

Selects the device to be used for the operation. Calling this method is necessary if there is more than one reader connected to the computer.

Properties

`SecureMode(AISecureModeMask)`

Sets/gets the security mode for the verification process. See “AISecureModeMask” on page 108 for possible values.

`Learning(VARIANT_BOOL)`

Sets/gets the learning mode on the verification template.

`SecurityLevel(VARIANT)`

Sets/gets the security level for the matching process. The security level is expressed as probability of false acceptance/false rejectance.

`FingerMask(VARIANT)`

Sets/gets the bit mask of the fingers to be checked for the given user. With a default value of –1, any matching finger will be accepted. If a different value is specified, only the fingers corresponding to those in the bit mask will be accepted. The left pinkie finger is represented by bit 0 and the right pinkie finger is represented by bit 9.

Event interface

`_IFPVerifyUserXEvents`

Event methods

`Done([in] VARIANT_BOOL VerifyOk, [in] SAFEARRAY(VARIANT)* pInfo, [in] AISecureModeMask Val)`

The operation is complete and the VerifyOk parameter contains the boolean result of the comparison. The pInfo parameter contains five elements: a score, the used threshold, a Boolean value to indicate whether the learning has occurred, the AIFinger value for the matched finger and the ID of the user. The last parameter gives the security mode, calculated by two factors: the requested security mode and the security mode of the matched registration template.

`DevDisconnected()`

The device was disconnected during the operation. If a specific device was selected, the operation will resume after the same device is reconnected; otherwise, it is canceled and must be run again after reconnecting that same device.

`DevConnected()`

The device was reconnected.

`Error([in] AIErrors errcode)`

An error occurred. The errcode parameter returns the error code, as described in “AIErrors” on page 109.

Return codes

None

Libraries

`DpSdkUsr.dll`

DPUsersDB

A COM component representing the user database object.

Note
Make sure to open this database on the same domain that is used in the SetHost method. First, open the database, then if you need to force the connection, you can use operations to use the new remote host.

Interface
`IDPUsersDB`

Methods
`SetHost([in]BSTR vHost)`

Not implemented in the current version.

`Open([in] BSTR dbName, [in, optional] BSTR TypeID)`

Not implemented in the current version.

```
OpenSystemDB([in] BSTR dbName, [in, optional] DBLevels
  dbLevel)
```

Opens the database for the domain specified in the dbName parameter. The dbLevel parameter specifies whether the main database or the local cache should be used. It is recommended to use the new OpenSystemDB2 method in the interface IDPUsersDB2 instead.

```
Add([in] BSTR Name, [in] VARIANT_BOOL bAccess,
  [out,retval] IDispatch **ppUser)
```

Adds a user whose name is Name to the database and returns the user record in ppUser. If the user already exists in the database, the function is equivalent to the FindByName method (described below). If the bAccess parameter is VARIANT_TRUE, the user record is accessed for writing and locked. Any attempt to access the same record for writing will fail; otherwise, the record is accessed for reading and multiple access is allowed.

`Remove([in] BSTR UserID)`

Remove the user from the database whose identifier is UserID. The user record is deleted, as well as all the credentials associated with it.

`RemoveByName([in] BSTR Name)`

Remove the user from the database whose name is Name. The user record is deleted, as well as all the credentials associated with it.

```
FindByName([in] BSTR Name, [in] VARIANT_BOOL bAccess,
  [out,retval] IDispatch **ppUser)
```

Search for the user whose name is Name and try to get the access specified in bAccess. For details on the use of the bAccess parameter, refer to the Add method (described above).

```
Find([in] BSTR UserID, [in] VARIANT_BOOL bAccess,
  [out,retval] IDispatch **ppUser)
```

Search for the user whose identifier is UserID and try to get the specified access. For details on the use of the bAccess parameter, refer to the Add method (described above).

```
SetFlags([in] LONG flag, [in] LONG Value)
```

Sets a value for the specified flag (reserved for future use).

```
GetFlags([in] LONG flag, [in,out] LONG *pValue)
```

Gets the value for the specified flag (reserved for future use).

```
ImportUser ( [in] VARIANT blob, [in] VARIANT_BOOL bAccess,
  [out] IDispatch **ppUser, [out,retval] AIErrors *pErr)
```

Imports a blob of bytes representing the signed user record object. It checks the signature and, if verified, imports the data in the DPUser object returned in the ppUser parameter. The user record is also added to the database.

Properties

```
_NewEnum([out,retval] IUnknown **ppEnum)
```

The standard _NewEnum property for the collections. It returns a pointer to an IEnumVARIANT interface (read-only).

Interface

```
IDPUsersDB2
```

Derived from IDPUsersDB.

Methods

`OpenSystemDB2([in] BSTR dbName, [in, out] DBLevels* dbLevel, [in] VARIANT_BOOL bForceRemote)`

Opens the database for the domain specified in the dbName parameter. The dbLevel parameter specifies whether the main database or the local cache should be used. The parameter returns whether the main or cache database was opened. Use ForceRemote to check for the remote server. You can force an attempt to connect to a remote server by setting the bForceRemote parameter to True. If cached credentials are disallowed in the DigitalPersona Pro Group Policy, users are not allowed to work locally.

Use this method instead of OpenSystemDB.

Event interface

None

Event methods

None

Return codes

None

Libraries

`DpSdkUsr.dll`

DPUser

A COM component representing the user record object. The user record can be referenced by name or ID and contains the user credentials.

Interface

`IDPUser`

Methods

`Save()`

Saves the user record to a file.

`Reload()`

Re-reads the user record from the disk. All unsaved changes will be lost.

`Export([out, retval] VARIANT* pVal)`

Exports the user record object as a signed blob of bytes.

`Import([in] VARIANT Val)`

Imports a blob of bytes representing the signed user record object, checks the signature and, if verified, imports the data into the DPUser object.

`SetFlags([in] LONG flag, [in] LONG Value)`

Sets the value for the specified flag (reserved for future use).

`GetFlags([in] LONG flag, [in,out] LONG *pValue)`

Gets the value for the specified flag (reserved for future use).

Properties

`UserID([out,retval] BSTR *pVal)`

Returns the string format of the GUID identifying the user (read-only).

`CreationTime([out,retval] DATE *pVal)`

Returns the date and time the record was created (read-only).

`ModificationTime([out,retval] DATE *pVal)`

Returns the date and time the record was last modified (read-only).

`Name([out,retval] BSTR *pVal)`

Returns the name of the user (read-only).

```
Credentials([in]AICredentials CredentType, [out,retval
  IUnknown **pCredents)
```

It returns a pointer to an IDPUserCredentials interface (read-only).

Event interface

None

Event methods

None

Return codes

None

Libraries

`DpSdkUsr.dll`

DPUserCredentials

A COM component that implements a collection of the credentials associated with a user.

Interface

`IDPUserCredentials`

Methods

`Add([in]VARIANT attribute, [in]IUnknown *pTemplate)`

Adds a new credential.

`Remove([in]VARIANT attribute)`

Removes the specified credential.

```
Item([in]VARIANT nNumber, [out,retval]IDispatch
  **pCredent)
```

Get the credential whose number is nNumber

Properties

```
Exist([in] VARIANT attribute, [out,retval] VARIANT_BOOL
  *pVal)
```

Returns VARIANT_TRUE if the credential whose attribute is given as a parameter exists; otherwise, VARIANT_FALSE.

`Filter(AICredentials)`

Sets/gets the filter on the credential type to be enumerated. This implementation allows only values of type Cr_Fingerprint.

`Count([out,retval] VARIANT *pVal)`

Returns the number of elements in the collection (read-only).

`_NewEnum([out,retval] IUnknown **ppEnum)`

The standard _NewEnum property for the collections. It returns a pointer to an IEnumVARIANT interface (read-only).

Event interface

None

Event methods

None

Return codes

None

Libraries

`DpSdkUsr.dll`

FPGetSample

A COM component that retrieves a fingerprint sample from the reader. It instantiates the components necessary for acquiring a raw sample and converting it to a sample. Although it does not provide a user interface, like FPGetSampleX, it does provide events that allow a developer to provide user feedback.

Interface

`IFPGetSample`

Methods

`SetHost([in]BSTR vHost, [out,retval] AIErrors *pErr)`

Selects the host on which the operation will be performed. DigitalPersona Pro Server does not support samples processing. This method is ignored.

`CreateOp([out,retval] AIErrors *pErr)`

Used prior to the Run method, CreateOp instantiates and links the necessary components for the operation.

`Run([out,retval] AIErrors *pErr)`

Runs the operation, which can be canceled any time. The Done event is fired when the operation terminates. While the operation is running, plug-n-play events are also fired. If the device is disconnected during the operation, the operation will pause until the same device is reconnected.

`Cancel([out,retval] AIErrors *pErr)`

Stops the operation, discarding all intermediate data, if any.

`SetDevicePriority([in] AIDevPriorities Priority, [in] LONG hWnd, [out,retval] AIErrors *pErr)`

Select the device priority for the client. In most cases, it is strongly recommended to use the standard priority, Dp_StdPriority. The standard priority allows the system to direct reader events to the active client based on the window handle supplied through the hWnd parameter. Refer to "AIDevPriorities" on page 108 for a complete list of priorities.

`SelectDevice([in] BSTR serNum, [out,retval] AIErrors *pErr)`

Selects the device to be used for the operation. Calling this method is necessary only if there is more than one reader connected to the computer.

`SetNonce([in] VARIANT Val, [out,retval] AIErrors *pErr)`

Sets a nonce to be included in the produced FPSample object.

Properties

`SecureMode(AISecureModeMask)`

Sets/gets the security mode for the raw sample-to-sample conversion. See "AISecureModeMask" on page 108 for possible values.

Event interface

`_IFPGetSampleEvents`

Event methods

`Done([in] IDispatch *pSample)`

The operation is complete and the pSample parameter points to the FPSample object generated.

`DevDisconnected()`

The device was disconnected while the operation was running. If a specific device was selected, the operation will resume after the same device is reconnected; otherwise, it is canceled and must be run again after reconnecting that same device.

`DevConnected()`

The device was reconnected.

`Error([in] AIErrors errcode)`

An error occurred. The errcode parameter returns the error code, as described in “AIErrors” on page 109.

Return codes

Er_OK is returned if successful

Er_InvalidArg is returned by:

- SetDevicePriority method when the given priority is not within the permitted ones
- SetNonce method when the given nonce is invalid

Er_NoHost is returned by:

- SetHost method when the given host cannot be contacted

Er_System is returned by:

- SelectDevice method when a system error occurs (maybe due to installation problems)

Er_NoDevice is returned by:

- SelectDevice method when there are no devices connected
- Run method when either the selected device or the default one is not connected
- CreateOp method when either the selected device or the default one is not connected

Er_InvalidID is returned by:

- SelectDevice method when the requested serial number does not exist

Er_AlreadyCreated is returned by:

- SetHost method when called after the operation has been created

Er_OperNotRunning is returned by:

- Cancel method when called without having created the operation

Libraries

`DpSdkOps.dll`

FPGetTemplate

A COM component that retrieves a fingerprint sample from the reader. It instantiates the components necessary for acquiring a raw sample and converting it to a sample and extracting features from the sample to create a template.

Interface

`IFPGetTemplate`

Methods

`SetHost([in]BSTR vHost, [out,retval] AIErrors *pErr)`

Selects the host on which the operation will be performed. DigitalPersona Pro Server does not support samples processing. This method is ignored.

`CreateOp([out,retval] AIErrors *pErr)`

Used prior to the Run method, CreateOp instantiates and links the necessary components for the operation.

`Run([out,retval] AIErrors *pErr)`

Runs the operation, which can be canceled any time. The Done event is fired when the operation terminates. While the operation is running, plug-n-play events are also fired. If the device is disconnected during the operation, the operation will pause until the same device is reconnected.

`Cancel([out,retval] AIErrors *pErr)`

Stops the operation, discarding all intermediate data, if any.

`SetDevicePriority([in] AIDevPriorities Priority, [in] LONG hWnd, [out,retval] AIErrors *pErr)`

Select the device priority for the client. In most cases, it is strongly recommended to use the standard priority, Dp_StdPriority. The standard priority allows the system to direct reader events to the active client based on the window handle supplied through the hWnd parameter. Refer to "AIDevPriorities" on page 108 for a complete list of priorities.

`SelectDevice([in] BSTR serNum, [out,retval] AIErrors *pErr)`

Selects the device to be used for the operation. Calling this method is necessary if there is more than one device connected to the computer.

`SetNonce([in] VARIANT Val, [out,retval] AIErrors *pErr)`

Prepares a nonce to be embedded in a template object when feature extraction is performed on the sample.

Properties

`SecureMode(AISecureModeMask)`

Sets/gets the security mode for the feature extraction process. See "AISecureModeMask" on page 108 for possible values.

`TemplateType(AITemplateTypes)`

Sets/gets the type of template type to create. Permitted values are Tt_Verification and Tt_PreRegistration.

Event interface

`_IFPGetTemplateEvents`

Event methods

`Done([in] IDispatch *pTemplate)`

The operation is complete. The pTemplate parameter points to the FPTemplate object.

`DevDisconnected()`

The device was disconnected while the operation was running. If a specific device was selected, the operation will resume after the same device is reconnected; otherwise, it is canceled and must be run again after reconnecting that same device.

`DevConnected()`

The device was reconnected.

`SampleReady([in] IDispatch* pSample)`

A sample was acquired from the reader.

`SampleQuality([in] AISampleQuality Quality)`

The feature extraction process is complete. The Quality parameter contains the quality of the sample used. Refer to “AISampleQuality” on page 109 for possible values for this parameter.

`Error([in] AIErrors errcode)`

An error occurred. The errcode parameter returns the error code, as described in “AIErrors” on page 109.

Return codes

Er_OK is returned if successful

Er_InvalidArg is returned by:

- SetDevicePriority method when the specified priority is not valid
- SetNonce method when the given nonce is invalid

Er_NoHost is returned by:

- SetHost method when a connection cannot be established with the given host

Er_System is returned by:

- SelectDevice method when a system error occurs, e.g., installation problems, etc.

Er_NoDevice is returned by:

- SelectDevice method when there are no devices connected

- Run method when either the selected device or the default device is not connected
- CreateOp method when either the selected device or the default device is not connected

Er_InvalidID is returned by:

- SelectDevice method when the requested serial number does not exist

Er_AlreadyCreated is returned by:

- SetHost method when called after the operation is started

Er_OperNotRunning is returned by:

- Cancel method when called without creating the operation with the CreateOp method

Libraries

`DpSdkOps.dll`

FPRegisterTemplate

A COM component that allows the user to register a fingerprint. It instantiates the components necessary for acquiring a four raw samples and converting them to samples and extracting features from them to create a registration template.

Interface

`IFPRegisterTemplate`

Methods

`SetHost([in]BSTR vHost, [out,retval] AIErrors *pErr)`

Selects the host on which the operation will be performed. In the current implementation, vHost can be only the name of an AD or NT domain. If not specified, the operation is performed on the local machine.

`CreateOp([out,retval] AIErrors *pErr)`

Used prior to the Run method, CreateOp instantiates and links all necessary components for the registration process.

`Run([out,retval] AIErrors *pErr)`

Runs the operation, which can be canceled any time. The Done event is fired when the operation terminates. While the operation is running, plug-n-play events are also fired. If the device is disconnected during the operation, the operation will pause until the same device is reconnected.

`Cancel([out,retval] AIErrors *pErr)`

Stops the operation, discarding all intermediate data, if any.

`SetDevicePriority([in] AIDevPriorities Priority, [in] LONG hWnd, [out,retval] AIErrors *pErr)`

Select the device priority for the client. In most cases, it is strongly recommended to use the standard priority, Dp_StdPriority. The standard priority allows the system to direct reader events to the active client based on the window handle supplied through the hWnd parameter. Refer to "AIDevPriorities" on page 108 for a complete list of priorities.

`SelectDevice([in] BSTR serNum, [out,retval] AIErrors *pErr)`

Selects the device to be used for the operation. Calling this method is necessary if there is more than one reader connected to the computer.

Properties

`SecureMode(AISecureModeMask)`

Sets/gets the security mode for the registration process. See "AISecureModeMask" on page 108 for possible values.

`Learning(VARIANT_BOOL)`

Sets/gets learning from the registration template produced.

`Target(AIRegTargets)`

Sets/gets the target for the registration. The only permitted value in this implementation is Rt_Verify.

Interface

`IFPRegisterTemplate2`

Derived from IFPRegisterTemplate.

Properties

`UsingXTFTemplate(VARIANT_BOOL)`

When set to True, uses the XTF registration template. The XTF template requires more space but provides fewer false rejects during validation. For more information in using the XTF template, see “Using the XTF Registration Template” on page 17.

Event interface

`_IFPRegisterTemplateEvents`

Event methods

`Done([in] IDispatch *pRegTemplate)`

The operation is complete. The pRegTemplate parameter points to the FPTemplate object generated. The type of the FPTemplate object is Tt_Registration.

`DevDisconnected()`

The device was disconnected during the operation. If a specific device was selected, the operation will resume after the same device is reconnected; otherwise, it is canceled and must be run again after reconnecting that same device.

`DevConnected()`

The device was reconnected.

`SampleReady([in] IDispatch* pSample)`

A sample was acquired from the reader.

`SampleQuality([in] AISampleQuality Quality)`

The feature extraction process is complete. The Quality parameter contains the quality of the sample used. Refer to “AISampleQuality” on page 109 for possible values for this parameter.

`Error([in] AIErrors errcode)`

An error occurred. The errcode parameter returns the error code, as described in “AIErrors” on page 109.

Return codes

Er_OK is returned if successful.

Er_InvalidArg is returned by:

- SetDevicePriority method when the given priority is invalid

Er_NoHost is returned by:

- SetHost method when a connection with the given host cannot be established

Er_System is returned by:

- SelectDevice method when a system error occurs, e.g., installation problems, etc.

Er_NoDevice is returned by:

- SelectDevice method when there are no devices connected
- Run method when either the selected device or the default device is not connected
- CreateOp method when either the selected device or the default device is not connected

Er_InvalidID is returned by:

- SelectDevice method when the requested serial number does not exist

Er_AlreadyCreated is returned by:

- SetHost method when called after the operation is created with the CreateOp method

Er_OperNotRunning is returned by:

- Cancel method when called without creating the operation with the CreateOp method

Er_RegFailer is returned:

- As a parameter of the Error event when the registration fails

Libraries

`DpSdkOps.dll`

FPVerifyTemplate

A COM component that facilitates the verification process. It instantiates the components necessary for acquiring a raw sample and converting it to a sample and extracting features from it to create a verification template. Then, it performs a match between the acquired verification template and a given registration template.

Interface

`IFPVerifyTemplate`

Methods

`SetHost([in]BSTR vHost, [out,retval] AIErrors *pErr)`

Selects the host on which the operation will be performed. In the current implementation, vHost can be only the name of an AD or NT domain. If not specified, the operation is performed on the local machine.

`CreateOp([out,retval] AIErrors *pErr)`

Used prior to the Run method, CreateOp instantiates and links the necessary components for the operation.

`Run([in] VARIANT Val, [out,retval] AIErrors *pErr)`

Runs the operation, which can be canceled any time. The Done event is fired when the operation terminates. While the operation is running, plug-n-play events are also fired. If the device is disconnected during the operation, the operation will pause until the same device is reconnected. The parameter Val contains a pointer to the FPTemplate object of type Tt_Register for matching.

`Cancel([out,retval] AIErrors *pErr)`

Stops the operation, discarding all intermediate data, if any.

`SetDevicePriority([in] AIDevPriorities Priority, [in] LONG hWnd, [out,retval] AIErrors *pErr)`

Select the device priority for the client. In most cases, it is strongly recommended to use the standard priority, Dp_StdPriority. The standard priority allows the system to direct reader events to the active client based on the window handle supplied through the hWnd parameter. Refer to "AIDevPriorities" on page 108 for a complete list of priorities.

`SelectDevice([in] BSTR serNum, [out,retval] AIErrors *pErr)`

Selects the device to be used for the operation. Calling this method is necessary if there is more than one reader connected to the computer.

Properties

`SecureMode(AISecureModeMask)`

Sets/Gets the security mode of the verification template. See "AISecureModeMask" on page 108 for possible values.

`Learning(VARIANT_BOOL)`

Sets/gets the learning on the verification template used in the verification process.

`SecurityLevel(VARIANT)`

Sets/gets the security level for the matching process. The security level is expressed as probability of false acceptance/false rejectance.

Event interface

`_IFPVerifyTemplateEvents`

Event methods

`Done([in] VARIANT_BOOL VerifyOk, [in] SAFEARRAY(VARIANT)* pInfo, [in] AISecureModeMask Val)`

The verification process is complete. The VerifyOk parameter contains the boolean result of the comparison. The pInfo parameter contains three elements: a score, threshold and Boolean value to indicate whether the learning occurred. The last parameter, Val, returns the security mode, calculated by two factors: the requested security mode and the security mode of the given registration template.

`DevDisconnected()`

The device was disconnected during the operation. If a specific device was selected, the operation will resume after the same device is reconnected; otherwise, it is canceled and must be run again after reconnecting that same device.

`DevConnected()`

The device was reconnected.

`SampleReady([in] IDispatch* pSample)`

A sample was acquired from the reader.

`SampleQuality([in] AISampleQuality Quality)`

The feature extraction process is complete. The Quality parameter returns the quality of the sample used. Refer to "AISampleQuality" on page 109 for possible values for this parameter.

`Error([in] AIErrors errcode)`

An error occurred. The errcode parameter returns the error code, as described in "AIErrors" on page 109.

Return codes

Er_OK is returned if successful.

Er_InvalidArg is returned by:

- SetDevicePriority method when the given priority is invalid

Er_NoHost is returned by:

- SetHost method when a connection with the given host cannot be established

Er_System is returned by:

- SelectDevice method when a system error occurs, e.g., installation problems, etc.

Er_NoDevice is returned by:

- SelectDevice method when there are no devices connected
- Run method when either the selected device or the default device is not connected

- CreateOp method when either the selected device or the default device is not connected

Er_InvalidID is returned by:

- SelectDevice method when the requested serial number does not exist

Er_AlreadyCreated is returned by:

- SetHost method when called after the operation was created with the CreateOp method

Er_OperNotRunning is returned by:

- Cancel method when called without creating the operation with the CreateOp method

Libraries

`DpSdkOps.dll`

FPRegisterUser

A COM component that facilitates the registration process for a given user. It starts the registration process for a given finger, as well as removes the fingerprint template associated with a given finger.

Interface

`IFPRegisterUser`

Methods

`SetHost([in]BSTR vHost, [out,retval] AIErrors *pErr)`

Selects the host on which the operation will be performed. In the current implementation, vHost can be only the name of an AD or NT domain. If not specified, the operation is performed on the local machine.

`CreateOp([out,retval] AIErrors *pErr)`

Used prior to the Run method, CreateOp instantiates and links all necessary components for the registration process.

`Run([in] VARIANT Val, [out,retval] AIErrors *pErr)`

Runs the operation, which can be canceled any time. The Done event is fired when the operation terminates. The Val parameter contains a pointer to the user record. The operation never terminates unless an error occurs; however, during the operation, events are fired when a new finger is successfully registered or deleted. The developer can decide to update the user record in the file or database at any time.

`Cancel([out,retval] AIErrors *pErr)`

Stops the operation, discarding all intermediate data, if any.

`SetDevicePriority([in] AIDevPriorities Priority, [in] LONG hWnd, [out,retval] AIErrors *pErr)`

Select the device priority for the client. In most cases, it is strongly recommended to use the standard priority, Dp_StdPriority. The standard priority allows the system to direct reader events to the active client based on the window handle supplied through the hWnd parameter. Refer to "AIDevPriorities" on page 108 for a complete list of priorities.

`SelectDevice([in] BSTR serNum, [out,retval] AIErrors *pErr)`

Selects the device to be used for the operation. Calling this method is necessary if there is more than one reader connected to the computer.

`RegisterFinger([in] AIFingers finger)`

Starts the registration process for the specified finger. For a listing of possible values for the AIFingers parameter, refer to "AIFingers" on page 109.

`DeleteFinger([in] AIFingers finger)`

Removes the registration template associated with the given finger. For a listing of possible values for the AIFingers parameter, refer to "AIFingers" on page 109.

Properties

`SecureMode(AISecureModeMask)`

Sets/gets the security mode for the registration process. See "AISecureModeMask" on page 108 for possible values.

`Learning(VARIANT_BOOL)`

Sets/gets learning for the registration template(s) that are produced.

`Target(AIRegTargets)`

Sets/gets the target for the registration. The only permitted value in this implementation is Rt_Verify.

Interface

`IFPRegisterUser2`

Derived from IFPRegisterUser.

Properties

`UsingXTFTemplate(VARIANT_BOOL)`

When set to True, uses the XTF registration template. The XTF template requires more space but provides fewer false rejects during validation. For more information in using the XTF template, see "Using the XTF Registration Template" on page 17.

Event interface

`_IFPRegisterUserEvents`

Event methods

`FingerRegistered([in] AIFingers fingerId)`

The registration process is complete. The fingerId parameter is the identifier of the modified finger. For a listing of possible values for the AIFingers parameter, refer to "AIFingers" on page 109.

`FingerDeleted([in] AIFingers fingerId)`

The finger, specified by the fingerId parameter, is deleted. For a listing of possible values for the AIFingers parameter, refer to "AIFingers" on page 109.

`DevDisconnected()`

The device was disconnected during the operation. If no specific device was selected, the operation will resume after the device is reconnected; otherwise, it is canceled and must be run again after reconnecting that specific device.

`DevConnected()`

The device was reconnected.

`SampleReady([in] IDispatch* pSample)`

A sample was acquired from the reader.

`SampleQuality([in] AISampleQuality Quality)`

The feature extraction process is complete. The Quality parameter contains the quality of the sample used. Possible values for the AISampleQuality parameter are listed in “AISampleQuality” on page 109.

`RegisteredFingers([in] LONG fingerMask)`

After the Run method has been called, this event is fired and returns a bit mask containing a reference to the registered fingers.

`Error([in] AIErrors errcode)`

An error occurred. The errcode parameter returns the error code, as described in “AIErrors” on page 109.

Return codes
None

Libraries
`DpSdkUsr.dll`

FPVerifyUser
A COM component that allows the developer to verify a given user. It takes care of instantiating the necessary components, connecting to the fingerprint reader, subscribing for events and taking care of the plug-n-play.

Interface
`IFPVerifyUser`

Methods

`SetHost([in]BSTR vHost, [out,retval] AIErrors *pErr)`

Selects the host on which the operation will be performed. In the current implementation, vHost can be only the name of an AD or NT domain. If not specified, the operation is performed on the local machine.

`CreateOp([out,retval] AIErrors *pErr)`

Used prior to the Run method, CreateOp instantiates and links all necessary components for the registration process.

`Run([in] VARIANT Val, [out,retval] AIErrors *pErr)`

Run the operation, starting the interaction with the user. Once running, the operation can be canceled at any time. The Val parameter contains a pointer to the user record of the user to be verified. When the verification is complete the Done event is fired to the client with the results of the matching. The Val parameter can also be a pointer to an IEnumVARIANT interface, i.e. an enumerator of users: in this case the system will try to find the first matching user. This is not an identification function and it can only be used with a set of users numbering less than 10.

`Cancel([out,retval] AIErrors *pErr)`

Stops the operation, discarding all intermediate data, if any.

`SetDevicePriority([in] AIDevPriorities Priority, [in] LONG hWnd, [out,retval] AIErrors *pErr)`

Select the device priority for the client. In most cases, it is strongly recommended to use the standard priority, Dp_StdPriority. The standard priority allows the system to direct reader events to the active client based on the window handle supplied through the hWnd parameter. Refer to “AIDevPriorities” on page 108 for a complete list of priorities.

`SelectDevice([in] BSTR serNum, [out,retval] AIErrors *pErr)`

Selects the device to be used for the operation. Calling this method is necessary if there is more than one reader connected to the computer.

Properties

`SecureMode(AISecureModeMask)`

Sets/gets the security mode for the registration process. See "AISecureModeMask" on page 108 for possible values.

`Learning(VARIANT_BOOL)`

Sets/gets learning on the verification template.

`SecurityLevel(VARIANT)`

Sets/gets the security level for the matching process. The security level is expressed as probability of false acceptance/false rejectance.

`FingerMask(VARIANT)`

Sets/gets the bit mask of the fingers to be checked for the given user. The default value is –1, meaning that any matching finger will be accepted. If a different value is specified, only the fingers corresponding to the ones in the bit mask are accepted. In the mask, bit 0 represents the left pinkie and bit 9 represents the right pinkie.

Event interface

`_IFPVerifyUserEvents`

Event methods

```
Done([in] VARIANT_BOOL VerifyOk, [in] SAFEARRAY(VARIANT)*
  pInfo, [in] AISecureModeMask Val)
```

The verification process is complete. The VerifyOk parameter contains the boolean result of the comparison. The pInfo parameter contains five elements: a score, threshold, a Boolean value to indicate whether the learning occurred, the AIFinger value for the matched finger and the ID of the user. The last parameter returns the security mode, calculated by two factors: the requested security mode and the security mode of the matched registration template.

`DevDisconnected()`

The device was disconnected during the operation. If a specific device was selected, the operation will resume after the same device is reconnected; otherwise, it is canceled and must be run again after reconnecting that same device.

`DevConnected()`

The device was reconnected.

`SampleReady([in] IDispatch* pSample)`

A sample was acquired from the reader.

`SampleQuality([in] AISampleQuality Quality)`

The feature extraction process is complete. The Quality parameter contains the quality of the sample used. Possible values for the AISampleQuality parameter are listed in “AISampleQuality” on page 109.

`Error([in] AIErrors errcode)`

An error occurred. The errcode parameter returns the error code, as described in “AIErrors” on page 109.

Return codes
None

Libraries
`DpSdkUsr.dll`

Data types

This section describes the data types used by the methods, events and properties of the Platinum SDK.

Note

In the subsections that follow, data types that are used in the current release are in bold type. Data types not shown in bold type are reserved for future use.

AICredentials

Cr_Unknown = 0,

Cr_Password = 1,

Cr_Fingerprint = 2,

Cr_Voice = 3,

Cr_Iris= 4,

Cr_Retina = 5,

Cr_Smartcard = 6

AIDataTypes

Dt_Unknown = 0,

Dt_RawSample = 1,

Dt_Sample = 2,

Dt_Template = 3

AITemplateTypes

Tt_Unknown = 0,

Tt_Registration = 1,

Tt_Verification = 2,

Tt_PreRegistration = 3,

Tt_RegistrationForId = 4,

Tt_VerificationForId = 5,

Tt_PreRegistrationForId = 6

AIOrientation

Or_Unknown = 0,

Or_Portrait = 1,

Or_Landscape = 2

AISecureModeMask

Sm_None = 0,

Sm_DevNonce = 0x00000001,

Sm_DevSignature = 0x00000010,

Sm_DevEncryption = 0x00000100,

Sm_FakeFingerDetection = 0x00001000,

Sm_NonceNotVerified = 0x00010000,

Sm_SignatureNotVerified = 0x00100000

AIDevPriorities

Dp_HighPriority = 1 (indicates the application will always get the acquired scan),

Dp_StdPriority = 2 (indicates the application gets the acquired scan only if the window is active),

Dp_LowPriority = 3 (indicates the application gets the acquired scan is there are no active applications waiting for it)

AIRegTargets

Rt_Verify = 1,

Rt_Identify = 2

AIErrors

Er_OK = 0,

Er_NoDevice = -1,

Er_NoHost = -2,

Er_OperNotRunning = -3,

Er_System = -4,

Er_InvalidId = -5,

Er_DevBroken = -6,

Er_OperAlreadyCreated = -7,

Er_RegFailed = -8,

Er_InvalidArg = -9

Er_BadSignature = -10

AIFingers

Fn_LeftPinkie = 0,

Fn_LeftRing = 1,

Fn_LeftMiddle = 2,

Fn_LeftIndex = 3,

Fn_LeftThumb = 4,

Fn_RightThumb = 5,

Fn_RightIndex = 6,

Fn_RightMiddle = 7,

Fn_RightRing = 8,

Fn_RightPinkie = 9

AISampleQuality

Sq_Good = 0,

Sq_None = 1,

Sq_TooLight = 2,

Sq_TooDark = 3,

Sq_TooNoisy = 4,

Sq_LowContrast = 5,

Sq_NotEnoughFtr = 6,

Sq_NoCentralRegion = 7

AIImageType

It_Unknown = 0,

It_BlackWhite = 1,

It_GrayScale = 2,

It_Color = 3

AIImagePadding

Ip_NoPadding = 0,

Ip_Left = 1,

Ip_Right = 2

AIPolarity

Po_Unknown = 0,

Po_Negative = 1,

Po_Positive = 2

AIRgbMode

Rm_NoColor = 0,

Rm_Planar = 1,

Rm_Interleaved = 2

DBLevels

Dl_Main = 1,

Dl_Cache = 2

Regulatory Information 6

DigitalPersona U.are.U® Fingerprint Reader Regulatory Information

Warning

To protect against risk of fire, bodily injury, electric shock or damage to the equipment:

- Do not immerse any part of this product in water or other liquid.
- Do not spray liquid on this product or allow excess liquid to drip inside.
- Do not use this product if it has sustained damage, such as damaged cord or plug
- Disconnect this product before cleaning.

Tested to comply with FCC Standards. For home or office use. Any changes or modifications not expressly approved by Digital Persona, Inc. could void your authority to operate this equipment. This device is rated as a commercial product for operation at +32°F (+0°C) to +104°F (+40°C).

The U.are.U Fingerprint Reader has been tested and found to comply with the limits for a Class B digital device under Part 15 of the Federal Communications Commission (FCC) rules, and it is subject to the following conditions: a) It may not cause harmful interference, and b) It must accept any interference received, including interference that may cause undesired operation.

This device conforms to emission product standards EN55022(B) and EN50082-1 of the European Economic Community and AS/NZS 3548 Class B of Australia and New Zealand.

This digital apparatus does not exceed the Class B limits for radio noise emission from digital apparatus as set out in the radio interference regulations of the Canadian Department of Communications.

Le présent appareil numérique n'émet pas de bruits radioélectriques dépassant les limites applicables aux appareils numéri-ques de Classe B prescrites dans le règlement sur le brouillage radioélectrique édicté par le Ministère des Communications du Canada.

This product has been tested to comply with International Standard IEC 60825-1:1993, A1:1997, A2:2001; IEC 60825-2:2000

CAUTION - USE OF CONTROLS OR ADJUSTMENTS OR PERFORMANCE OF PROCEDURES OTHER THAN THOSE SPECIFIED HEREIN MAY RESULT IN HAZARDOUS RADIATION EXPOSURE.

Attention - L'utilisation de contrôles et de réglages ou l'application de procédures autres que ceux spécifiés dans le présentdocument peuvent entraîner une exposition à des radiations dangereuses.

Achtung - Die hier nicht aufgeführte Verwendung von Steuerelementen, Anpassungen oder Ausführung von Vorgängen kann eine gefährliche Strahlenbelastung verursachen.

Precaución - La utilización de controles, ajustes o procedimientos distintos a los aquí especificados puede dar lugar a niveles de radiación peligrosos.

Attenzione - L'utilizzo di controlli, aggiustamenti o di procedure diverse da quelle qui specificate puo' portare all'esposizione ad un livello di radiazioni pericoloso.

小心:對控制或調整的規程用途或執行不同於那些指定,可能導致輻射暴露危害。

This product uses LEDs that are inherently Class 1.

Appendix

Fingerprint Reader Usage and Maintenance

This section provides reader usage and maintenance guidelines, which are intended to maximize fingerprint registration and authentication performance.

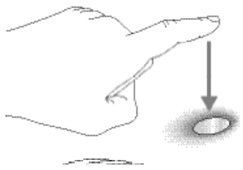
Proper usage of the reader during fingerprint registration and authentication, as well as a well-maintained reader, is crucial to achieving optimal fingerprint recognition performance.

The next section, "Proper Fingerprint Reader Usage," describes the proper way to use the reader to register fingerprints and authenticate using them. It is followed by reader maintenance instructions, provided in "Cleaning the Reader" on page 115.

Proper Fingerprint Reader Usage

To reduce the number of false rejects, you must place a finger on the reader correctly when registering fingerprints and authenticating.

During both processes, you must place the pad of your finger—not the tip or the side—in the center of the oval window of the reader in order to maximize the area of the finger that touches the reader window.



Place the entire pad of your finger squarely on the reader window

Apply even pressure. Pressing too hard will distort the scan; pressing too lightly will produce a faint, unusable scan. Do not "roll" your finger.

To complete the fingerprint scan, hold your finger on the reader until you see the reader light blink. This may take longer if the skin is dry. When the light blinks and, if configured, a sound plays, you may lift your finger.

If the reader is capturing your fingerprint scan as indicated by the reader blink, but DigitalPersona Pro consistently rejects it, you may need to reregister that finger by first deleting it and then registering it again.

Cleaning the Reader

The condition of the reader window has a large impact on the ability of the reader to obtain a good quality scan of a fingerprint. Depending on the amount of use, the reader window may need to be cleaned periodically.

To clean it, apply the sticky side of a piece of adhesive cellophane tape on the window and peel it away.

Under heavy usage, the window coating on some readers may turn cloudy from the salt in perspiration. In this case, gently wipe the window with a cloth (not paper) dampened with a mild ammonia-based glass cleaner.

Reader Maintenance Warnings

There are several things you should never do when cleaning or using the reader:

- Do not pour the glass cleaner directly on the reader window.
- Do not use alcohol-based cleaners.
- Never submerge the reader in liquid.
- Never rub the window with an abrasive material, including paper.
- Do not poke the window coating with your fingernail or any other item, such as a pen.

The fingerprint reader is for indoor home or office use only.

Index