Precise BioMatch™ Flex H User Manual

1.x

P/N: AM010056

1 INTRODUCTION

This manual describes how to develop applications using the Precise BioMatch™ Flex H Toolkit with support for Precise Match-on-Card™ technology. The reader of this manual should be familiar with Precise Match-on-Card™ technology.

For more information about Precise BioMatch™ and Precise Match-on-Card™ technologies please refer to the white papers to be found in the Documentation folder in this toolkit. For more comprehensive information about Precise Biometrics and our technology please refer to our web site www.precisebiometrics.com.

1.1 Scope

This document, the user manual, covers application issues from initial solution architecture through to end product implementation guidelines. The installation guide covers toolkit installation procedures and directory structure while the reference manuals are presented as separate html documents for the specific APIs.

1.2 Abbreviations

API	Application Programming Interface
BIR	Biometric Information Record
DLL	Dynamic Link Library
FAR	False Acceptance Rate
FRR	False Rejection Rate
GUI	Graphical User Interface
MoC	Match-on-Card
PnP	Plug and Play
RFU	Reserved for Future Use
SC	Smart Card
USB	Universal Serial Bus

1.3 Definitions

BioAPI	Standard biometric API developed by the BioAPI Consortium. For additional information please refer to www.bioapi.org

1.4 Toolkit Overview

The Precise BioMatch™ Flex H Toolkit is a development tool for integrating the Precise Match-on-Card™ technology into any smart card based application that requires secure and convenient user verification.

The essential core of the Precise BioMatch™ Flex H Toolkit is the software component Precise BioMatch™ Flex H. Precise BioMatch™ Flex H consists of enrollment, verification and support functions necessary to acquire and process biometric data. These functions are packaged in the dynamic link libraries *pb_flex_h.dll* and *pb_flex_h_net.dll*. When working with the C API, the supplied library *pb_flex_h.lib* can also be used to access the *pb_flex_h.dll* C API dll.

Below is a system overview. Precise BioMatch™ Flex H is supplied in this toolkit. An application developed with the Precise BioMatch™ Flex H Toolkit can be used together with all Precise BioMatch™ C/NET/J/M 3.0 enabled smart cards and fingerprint readers. Functionality to perform operations on the smart card such as file management and cryptographic functionality is not provided in the toolkit.

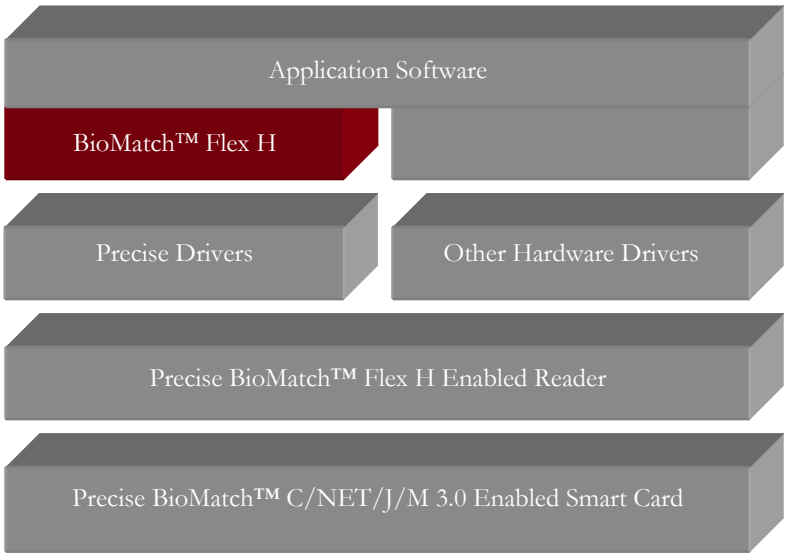

Figure 1 - Application architecture.

2 APPLICATION ARCHITECTURE

When designing a new biometric application it is important to first select the appropriate technology to be used based on user requirements such as security, personal integrity and other deployment issues. This section aims to clarify the choices available to a developer using the Precise BioMatch™ Flex H Toolkit. Furthermore, product deployment and licensing issues are covered in relation to redistribution of the proprietary Precise BioMatch™ Flex H libraries.

2.1 Security

To achieve maximum application security it is necessary to consider how biometric and non-biometric personal data is acquired, stored and processed. By restricting the flow of tangible data, a higher level of security can be achieved. The following sections will outline the key options available for adapting a Precise BioMatch™ Flex H Toolkit application to the chosen level of operating security. Keep in mind that application security is closely related to personal integrity.

2.2 Personal Integrity

Since perceived personal integrity is a key issue for end-users of biometric applications it is worth noting that different software approaches handle sensitive biometric information very differently. Therefore it is important for the developer to consider how this flow of sensitive information can be restricted appropriately.

2.3 Smart Card Reader Selection

A smart card reader must be selected to use with applications using BioMatch™ Flex H. Both Precise Biometrics' smart card readers with integrated fingerprint readers and other PC/SC compliant smart card readers manufactured by other vendors may be used. The possible solution model paths are shown in the graph below.

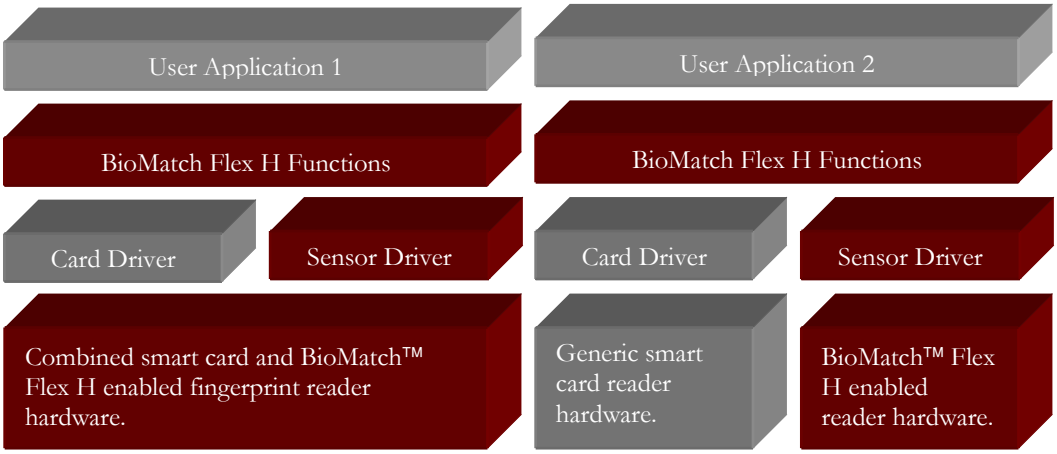

Figure 2 - Hardware selection options.

Here, *"User Application 1"* is using an integrated BioMatch™ Flex H enabled reader while *"User Application 2"* is using a BioMatch™ Flex H enabled fingerprint reader and a separate smart card reader for smart card related operations.

2.4 Supported Fingerprint Readers

This toolkit supports the following readers directly through the toolkit functions:

Vendor	**Reader**
Precise Biometrics AB	Precise 200MC
Precise Biometrics AB	Precise 250MC
Precise Biometrics AB	Precise 100XS
Crossmatch Technologies Inc	Verifier® 300 LC
UPEK	Reference implementations of the TCS1, TCS2, TCS3, TCS4 and TCS5sensors
AuthenTec	Reference implementations of the AES2501, AES2550 and AES2810 sensors
Broadcom	Sensors integrated with the Broadcom BCM5880 chip

Furthermore, many third party readers are supported indirectly through the image import/export functions described in section 3.3.5. Precise Biometrics AB takes no responsibility for the performance of third party readers or the biometric performance achieved with images captured from such readers.

2.5 System Integration & Licensing Agreements

After integrating support for Precise BioMatch™ Flex H to your application(s), the system solution should consist of the following components.

- Precise BioMatch™ Flex H (pb_flex_h.dll) or (pb_flex_h.dll and pb_flex_h_net.dll) to be installed on the target computer.
- BioMatch™ Flex H enabled fingerprint reader with integrated PC/SC smart card reader or combined with a stand-alone PC/SC smart card reader.
- Smart card supporting Precise Match-on-Card™ technology, e.g. Precise BioMatch™ C/J/NET/M 3.0 Smart Card. Please contact Precise Biometrics for a current list of Precise Match-on-Card™ compliant smart cards.

IMPORTANT: Precise BioMatch™ Flex H (*pb_flex_h.dll* and *pb_flex_h_net.dll*) are proprietary software. Sales and marketing of the application(s) developed with Precise BioMatch™ Flex H shall therefore be subject to a separate supply agreement with Precise Biometrics for Precise BioMatch™ Flex H end user licenses.

3 API OVERVIEW

This chapter explains the general ideas behind the Precise BioMatch™ Flex H Toolkit API and describes the main components.

3.1 The Design Model

The toolkit is function oriented. Each function call is issued as a standalone call to the underlying biometric framework except for the `Initialize()` and `Release()` functions accessible through the C-API. These functions are described in more detail in the C-API reference manual.

3.2 The Main Components

The API is divided into two parts with equal functionality:

- C API.
- .NET framework API.

Both APIs supply the same biometric functionality and are fully compatible in the context of capturing, processing and enrolling fingerprint images and templates.

These APIs provide functions for capturing fingerprint images, providing feedback on fingerprint quality and biometric operations on the captured images themselves.

The APIs do **NOT** provide the following functionality:

- Smart card communication functions.
- Cryptographic processing.

These types of functionality vary along with the chosen smart card edge application to be used and fall outside the scope of this toolkit.

3.3 General Programming Guidelines

In this section some programming specific issues are discussed. For optimal performance and expected functionality the guidelines in this section should be followed when using the Precise BioMatch™ Flex H Toolkit.

3.3.1 User Feedback

Applications often need to be able to give feedback to the users, so they need to get information about finger placement etc. during image capturing operations. The users application may support this need by supplying a callback function that is called every time an image is captured by the sensor. The optional callback function pointer/object is passed as a context parameter to image capturing toolkit routines. Once called, the callback function may itself query the image capture device for information on image attributes. In this way, the user may be provided quality parameters (such as if the finger is too dry/wet) during the image capture process.

The callback function may also request images that are suitable to view on the screen.

For more information on practical implementation of a callback function, please see the individual reference documents for the required API. The provided example programs also provide information on how to readily provide user feedback in a real application environment.

3.3.2 Return Values

The functions return success or an error code/exception. The error codes/exceptions are reserved for real errors (such as there is not enough memory for the operation to succeed).

The actual error codes used vary between the C and NET APIs. Please see the reference manuals for the required API for more information on return codes and/or exception handling. Information on API specific return values can be found in (APPENDIX A - .NET API Specific Issues) and (APPENDIX B - C API Specific Issues).

Note that the out-parameters should be considered to have ”undefined” values when a function returns an error code different from **OK** or throws an exception.

3.3.3 FAR Levels

Depending on what the system is going to be used for, different FAR levels can be chosen. The FAR level is specified during enrollment and the specified level is stored on the smart card as a part of the secure reference data. Though any numerical level may be chosen, five predefined FAR parameters are specified in the range 1:100 through 1:1,000,000. For example, FAR = 1:10,000 means that 10,000 imposter attempts, i.e. 10,000 fingers, are on average needed to produce one (1) false accept. FRR is defined as the rate of falsely rejected attempts against the total number of attempts made. The FAR is inversely related to the FRR such that a decreasing FAR will result in an increasing FRR, e.g. `FAR = 1:100` in the table will result in a much lower FRR than `FAR=1:1,000,000`. Therefore it is recommended not to choose a lower FAR level than required by the system. A FAR value of 1:10,000 is recommended unless the customer explicitly wants a lower value. Here is a list of predefined constant FAR values for the specific APIs:

FAR	C API	NET API
`1:100`	`PB_FAR_100`	`BM_FarLevel.Far100`
`1:1,000`	`PB_FAR_1000`	`BM_FarLevel.Far1000`
`1:10,000`	`PB_FAR_10000`	`BM_FarLevel.Far10000`
`1:100,000`	`PB_FAR_100000`	`BM_FarLevel.Far100000`
`1:1,000,000`	`PB_FAR_1000000`	`BM_FarLevel.Far1000000`

Table 1 - Predefined FAR constants.

The FRR is not only dependent on the FAR, but also on a lot of abstract parameters, such as:

- The users experience of the fingerprint reader.
- The quality of the enrolled fingerprint template.
- The skin condition of the users fingerprint.
- When the last cleaning of the fingerprint sensor was made.

This implies that different individuals will achieve different FRR although the same FAR level is used.

A custom FAR is generated by the division `0x7fffffff/n` where `n` is the required FAR level. So `0x7fffffff/10000` is equivalent to `FAR = 1:10,000`.

The FRR will always be optimized to be as low as possible given a specific FAR level.

For information on the actual constant definitions used, please consult the reference manual for the required API.

3.3.4 Quality Values

The quality values provided by the toolkit are generated in accordance with the BioAPI specifications and range from 0 to 100 where 0 is the lowest possible quality and 100 is the highest.

3.3.5 Importing and Exporting Images

The toolkit provides functionality for the import and export of standard bitmap images and WSQ images.

For bitmaps, only 8-bit grayscale images conforming to Windows V3 40-byte header standard are supported.

When exporting WSQ images, a compression bit rate of 2.25 is used.

The import functionality implicitly supports the import of images captured with third party sensors. When importing images from an external source, it is strongly recommended that the image quality is determined using the `Finger_Status()` function before it is used for enrolment or verification operations. Furthermore, visual inspection of the imported images is recommended during application development to ensure that the foreign images are correctly imported by the toolkit.

The export functionality facilitates the storage of images between operations and for archiving purposes.

3.3.6 Reserved for Future Use

When the text refers to values/bits as RFU, the application of this specification must set those values to 0.

3.4 MoC Programming Guidelines

When enrolling a user, fingerprint images from one or more fingers are captured and the characteristic information of the fingerprints is extracted and converted into fingerprint templates. The templates should then be stored on the user's personal smart card enabling biometric protection of information on that smart card.

A template pair consists of reference data and an associated biometric header. The biometric header is used by the verification application to acquire a suitable fingerprint sample. It contains information relevant to the type and form of data required by the smart card and should therefore be stored in the smart cards public area. The reference data however, should be stored in the smart cards private area as it contains the secure reference template used to perform the verification on the smart card.

By utilizing the biometric header, the verification application can acquire a fingerprint image, which is then preprocessed for transmission to the smart card where the actual verification will be executed. In this way, the total MoC verification time is minimized since the time consuming preprocessing stage is performed in a more powerful environment such as a PC.

3.4.1 MoC Enrollment

This section aims to describe the recommended procedure for enrolling Match-on-Card templates. A theoretical approach is provided for constructing MoC enrollment applications. Actual implementations that follow these guidelines can be found in the supplied toolkit example applications.

3.4.1.1 User Feedback

Feedback to the user is essential for the enrollment procedure to be as effective as possible. The user feedback function group in the API provides two functions for providing feedback on every captured image. These functions, described in detail in the individual API reference manuals, provide information on what the user should do to get a high quality template. These functions should only ever be called from within a callback function or callback triggered event handler. We strongly recommend that the functions be used following the guidelines below.

`CB_Get_Image_For_Viewing()`

This function returns a processed bitmap image representing the fingerprint that gives the user basic feedback about the finger placement on the sensor. This feedback is purely graphical, thus providing intuitive and language independent guidance for the user.

For optimal results it should however be complemented with the feedback from the other callback/event handler function described below.

`CB_Finger_Status()`

This is a multi-purpose function used to retrieve one or more fingerprint image characteristics. It is possible to select the specific characteristics to retrieve in both the C and NET API. By selecting only the characteristics necessary for the specific application, the call will execute faster. See the API reference manuals for further parameter specific information. The available characteristics are listed below:

- **Image** The latest raw image captured. (Note that this image could be quite large depending on the sensor used.)
- **Quality** The quality of the latest image captured as defined by the BioAPI specification.
- **Condition** The general condition of the latest image captured.
- **Present** A Boolean check of whether a finger is present in the latest image captured.

The **image** parameter is a full scale raw image and is usually only used for testing and evaluation purposes. It is recommended that the `CB_Get_Image_For_Viewing` function is instead used to request an image suitable for display. The **quality** parameter gives a measure of the general image quality while the **condition** parameter gives information about the condition of the finger on the sensor. Note that a too light pressure on the sensor may be interpreted as an overly dry finger. Our recommendation is to inform the user on the finger condition if it is not OK, and ask the user to perform a suitable action as described below.

Finally, the **present** parameter indicates if a finger is currently placed on the sensor or not. This can be used to direct the user to place a finger on the sensor if they have not already done so. For more information on practical API specific callback implementation see (APPENDIX A - .NET API Specific Issues) and (APPENDIX B - C API Specific Issues).

3.4.1.2 Recommended Enrollment Procedure

There are two main methods of enrollment.

- Operator assisted enrollment
- Unsupervised enrolment

Since correct operator assisted enrolment procedures vary for each implementation, only the unsupervised enrolment procedure is covered in this document. For specific information on supervised enrolment procedures, please contact Precise Biometrics AB.

The enrollment of fingerprints and creation of fingerprint templates is a very important procedure. A template with poor quality will cause a high FRR, which results in reduced biometric system performance. On the other hand the enroll procedure must not take too long to perform since this would also be irritating.

The enrollment procedure described in this section should be followed for all unsupervised enrolment implementations. It will only take a few seconds to enroll a finger and it produces fingerprint templates with high quality.

The basic idea behind this enrollment procedure is that one image is enrolled and the resulting template is then validated against a second captured image. If this validation succeeds, then the template is valid, otherwise the template is discarded, and another attempt is made. The most likely reason for this validation to fail is that the user is inexperienced, and has not placed their finger consistently between the two images. Given good feedback, the user usually enrolls correctly the third time.

The flow chart on the next page describes the recommended enrollment procedure in detail when implementing it for Precise Match-on-Card™ purposes.

For an example-implementation of the recommended enrollment procedure see the provided example programs.

Note: For simplicity, the biometric header and the reference data are seen as one template when describing the recommended enrollment procedure.

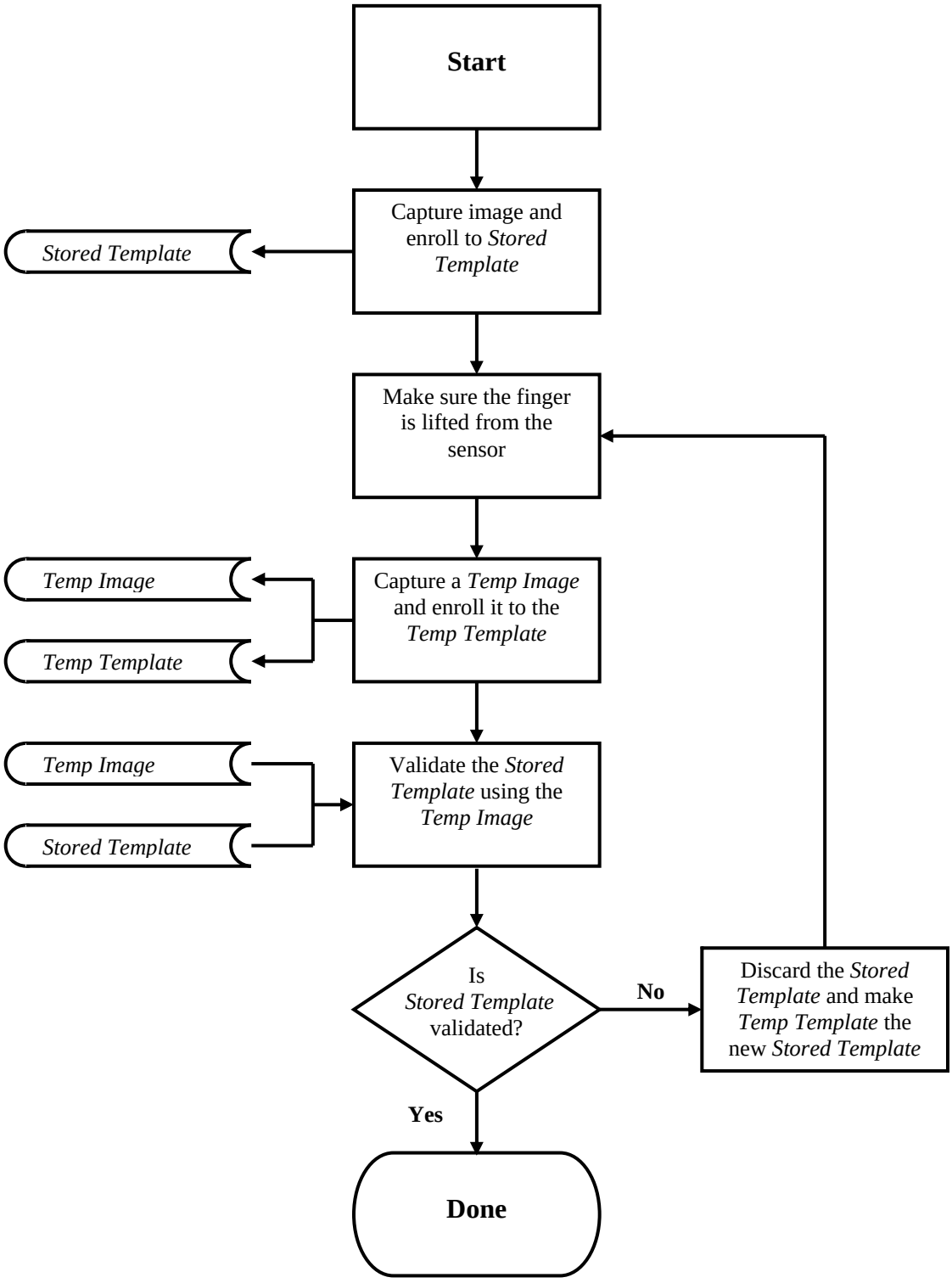

Figure 3 - Recommended enrolment procedure.

3.4.2 MoC Verification

MoC enrollment should always be performed in a supervised environment to ensure that biometric data is acquired from the intended individual. This is not always possible for MoC verification operations. For this and other reasons, the verification process differs somewhat from the enrollment process.

When verifying a user, the biometric header is first fetched from the user's smart card. This header is then used to capture and process a suitable fingerprint image from one of the user's enrolled fingers. The generated verification data is then sent to the user's smart card where it is verified against the reference data on the actual card. For optimal performance and expected functionality, the guidelines in this section should be followed when implementing verification applications.

3.4.2.1 Latent Finger Protection

The API incorporates functionality to prevent latent fingers from producing false accepts. Latent fingers originate from the skin oils of the previous finger placed on the sensor. Due to the latent finger protection functionality, the `Capture_Image()` function should not be called two times without making sure the user has lifted their finger from the sensor between the calls. If the finger is not removed between the calls it may activate the latent finger protection and no image will be captured the second time `Capture_Image()` is called.

3.4.2.2 User Feedback

Usability tests have shown that both the FRR and the time needed for verification can be significantly decreased if the user receives adequate visual feedback during the verification process. During verification, the image feedback function `CB_Get_Image_For_Viewing()` may be used to acquire an image that is suitable for viewing. Please see the API specific reference manual for further information on user feedback functions.

3.4.2.3 Recommended Verification Procedure

How the verification procedure is implemented is very important since both the FRR and the time needed for verification (a very important part of how the system is perceived by the user), depends on it. For good performance we strongly recommend that you follow the guidelines below. For an example-implementation of the recommended verification procedure see the provided MoC verification example application.

Use the `Capture_Image()` function to capture a quality image of the users finger. If possible, use the callback available functions described in section 3.4.1.1 to aid the user in correctly presenting their finger to the sensor. A recommended implementation of the callback function/event handler can be seen in the provided example program.

Use the function `Create_Verification_Data()` to generate the actual verification template that is to be sent to the smart card.

4 SUPPORT

Some questions may be answered by the FAQ available on our web site www.precisebiometrics.com. To access the FAQ, select **Support** from the menu on the website.

The support pages provide access to:

- FAQ on Precise Biometrics products.
- The latest Precise drivers.
- The technical support case entry form.

The technical support case entry form provides a gateway to Precise Biometrics customer support should the information on the FAQ pages not suffice.

APPENDIX A - .NET API Specific Issues

This appendix covers the .NET API specific programming information necessary to correctly manage the callback/event handling architecture and function return value/exception handling.

Callbacks/Events

Though use of callbacks is optional, their implementation is highly recommended since they can be used to provide useful information to the user during actual image capture operations. To correctly implement a callback simply follow these guidelines:

1. Implement at least one callback function with the correct input parameters.
2. Create an instance of the `BioMatch` class.
3. Register the callback function(s) with the toolkit's biometric event dispatcher.
4. Use the `BioMatch` class instance to search for a suitable biometric reader.
5. Use the `BioMatch` class instance to call the image capture function.

The image capture function will continuously capture images from the specified reader until the specified timeout expires or an image of sufficient quality is captured. Each attempted capture will result in the callback function(s) being invoked by the biometric event handler. Keep in mind that there are two image capturing functions that can trigger callbacks in the .NET API:

`CaptureImage()` and

`WaitForNoFinger()`

For this and other reasons, it is often useful if the callback function knows which code call to the image capturing function generated the callback. In both the callback capable image capturing functions, an optional parameter can be supplied to carry this contextual information. This `context` parameter can be any object or class and it is up to the callback function to query this information and thereby derive the context of the callback invocation. This can be done either by querying the context objects type directly or by casting the context object to a specific abstract type that can be queried as to further casting procedures.

On the following page is shown a C# .NET example of a simple image capturing program using a single callback that outputs text to a previously generated GUI TextBox object.

Note that this example code assumes that the project contains a reference to the *pb_flex_h_net.dll*, that the *pb_flex_h.dll* is in the search path and that the GUI contains the two following objects:

`button1` and

`textBox1` (Set up with multi-line = true)

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using PreciseBiometrics.BMFH;

namespace CallbackDemo1
{
    public partial class Form1 : Form
    {
        // Define an instance holder of the BioMatch class.
        private static BioMatch toolkit;
        public Form1()
        {
            // Create an instance of the BioMatch class.
            toolkit = new BioMatch();
            // Register the callback function with the BioMatch class.
            toolkit.BiometricCallback +=
                new BioMatch.BiometricEventHandler(callBack);
            InitializeComponent();
        }

        // Define a callback function to execute every time an image
        // is captured.
        void callBack(int token, object context)
        {
            BM_Image image = null;
            int quality = 0;
            BM_ImageCondition condition = BM_ImageCondition.Unknown;
            BM_ImagePresent present = BM_ImagePresent.False;

            // Request the quality and condition of latest image captured.
            toolkit.CB_FingerStatus(
                token,
                out image,
                out quality,
                out condition,
                out present,
                BM_StatusOption.Quality | BM_StatusOption.Condition);

            // Analyze the context object and present data if applicable.
            if (typeof(String) == context.GetType())
            {
                textBox1.AppendText((String)context);
                textBox1.AppendText("Quality=" + quality.ToString() + ", ");
                textBox1.AppendText("Condition=");
                textBox1.AppendText(toolkit.ConditionToString(condition));
                textBox1.AppendText("\r\n");
                textBox1.Refresh();
            }
        }
```

```
        // The button clicked event handler.
        private void button1_Click(object sender, EventArgs e)
        {
            BM_Reader[] readers = null;
            UInt32 timeout = BioMatch.TIMEOUT_FOREVER;
            BM_Image image = null;
            String message = "Capturing Image... ";
            Object context = message;
            BM_ReturnCode ret = BM_ReturnCode.Ok;

            // Find all connected biometric readers.
            toolkit.ListBiometricReaders(out readers);

            // If there is at least one reader connected, use it.
            if ((readers != null) && (readers.Length > 0))
            {
                ret = toolkit.CaptureImage(
                    readers[0],
                    timeout,
                    out image,
                    context);
            }

            // Output the results of the operation to the text box.
            if (ret == BM_ReturnCode.Ok)
            {
                textBox1.AppendText("Image captured successfully.\r\n");
            }
            else
            {
                textBox1.AppendText("Image capture failed:\r\n");
                textBox1.AppendText(BM_Error.ReturnCodeToText(ret) + "\r\n");
            }
        }
    }
}
```

Return Codes and Exception Handling

Though the previous program demonstrates callbacks and a limited error check, a fully fledged program should even check for eventual exceptions thrown by the toolkit functions.

In essence, each call to the toolkit should be wrapped in a try/catch block to ensure continuous application functionality. Exceptions will only be thrown when exceptional errors occur. The most common cause of exceptions will be improper installation of the required libraries that the toolkit comprises on the target application system. Other possible causes include, but are not limited to, insufficient system memory, missing or faulty device drivers and other operating system related execution errors. For more information on the specific exceptions thrown by the toolkit, see the NET API reference manual.

Those errors in execution that can be remedied by advice to the end user are instead relayed as return parameters where applicable. Functions that generate a return code should always be checked for correct execution before examining the function parameters as these will contain undefined values on improper execution.

The six toolkit functions that return execution status are listed in the following table.

Function Name	**Returns**				
	Reader	**TimedOut**	**Cancelled**	**Quality**	**Ok**
`CaptureImage()`	✓	✓	✓		✓
`CaptureRawImage()`	✓				✓
`WaitForNoFinger()`	✓	✓	✓		✓
`CreateEnrolTemplateFromImage()`				✓	✓
`CreateVerificationTemplateFromImage()`				✓	✓
`ValidateEnrolmentTemplateWithImage()`				✓	✓

Table 2 - Returned function error codes.

BM_ReturnCode.Reader	Occurs when the specified reader cannot be found or is unavailable for the specific operation. Usually this will be due to the reader not being plugged in. The user should be asked to (re)connect the device or to reinstall its drivers.
BM_ReturnCode.TimedOut	Occurs when the specified timeout period for the function expires before the function can complete its task. This situation should be handled by the application.
BM_ReturnCode.Cancelled	Occurs when a cancelled function call returns. This should also be handled by the application.
BM_ReturnCode.Quality	Occurs when the supplied image has insufficient quality for the desired purpose. Generally another image must be supplied that has a higher quality. The user should be prompted to capture another fingerprint image or to choose an alternative image/finger to process.
BM_ReturnCode.Ok	Occurs when the function executes correctly.

The return codes from these six functions should always be inspected after execution to ensure that the returned parameters hold valid data.

APPENDIX B - C API Specific Issues

This appendix covers the C API specific programming information necessary to correctly manage the callback architecture and function return codes.

Callbacks

Though use of callbacks is optional, their implementation is highly recommended since they can be used to provide useful information to the user during actual image capture operations. To correctly implement a callback simply follow these guidelines:

1. Implement a callback function with the correct input parameters.
2. Initialize the toolkit.
3. Pass the callback functions address to the callback capable image capture function.
4. Finally release the toolkit resources when finished.

The image capture function will continuously capture images from the specified reader until the specified timeout expires or an image of sufficient quality is captured. Each attempted capture will result in the callback function being invoked by the biometric event handler. Keep in mind that there are six image capturing functions that can trigger callbacks in the C API:

`pb_capture_imageA()`,

`pb_capture_imageW()`,

`pb_sc_capture_image()`,

`pb_wait_for_no_fingerA()`,

`pb_wait_for_no_fingerW()` and

`pb_sc_wait_for_no_finger()`

For this and other reasons, it is often useful if the callback function knows which code call to the image capturing function generated the callback. In all the callback capable image capturing functions, an optional parameter can be supplied to carry this contextual information. This `context` parameter is a void pointer and can therefore be used to communicate a variety of types and structures.

On the following page is shown a C example of a simple image capturing program using a callback that outputs text to the stdout port.

Note that this example code assumes that the program will be linked with the *pb_flex_h.lib* after compilation and that the *pb_flex_h.dll* is located in the application environments search path. The example is set to use the toolkit functions in ASCII mode. For more information on ASCII and UNICODE, please refer to the C API reference manual.

```
#include "pb_flex_h.h"
#include <stdio.h>

// This is an ASCII example application
//#define UNICODE

void callback_function(int token, void* context)
{
	char* text_message = 0;
	int result = PB_EOK;
	int quality = 0;
	int condition = 0;

	// Check for valid context pointer string.
	if (0 != context)
	{
		// Query only quality and condition of the latest captured image.
		result = pb_cb_finger_status(
			token,
			0,
			&quality,
			&condition,
			0);
		// Output to screen the combination of context and status.
		text_message = (char*)context;
		printf(text_message);
		printf("Quality=%d, Condition=%d\n",quality, condition);
	}
}

int main(int argc, char*argv[])
{
	int result = PB_EOK;
	LPSTR reader_list = 0;
	int nof_readers = 0;
	int timeout = 10000;
	pb_image_t* image;
	char context[] = "Capture Image... ";

	// Initialize toolkit
	result = pb_initialize();
	// Check result
	if (PB_EOK != result)
	{
		return result;
	}

	// List available readers.
	result = pb_list_readers(&reader_list,&nof_readers);
	// Check result.
	if ((PB_EOK != result) || (0 == reader_list[0]))
	{
		pb_release();
		return result;
	}
```

```
    // Capture an image using callbacks.
    result = pb_capture_image(
           reader_list,
           timeout,
           &image,
           &callback_function,
           (void*)context);
    // Free resources.
    pb_free(reader_list);
    // Check result.
    if (PB_EOK != result)
    {
           printf("Capture image failed.\n");
           pb_release();
           return result;
    }

    // Output success.
    printf("Capture image succeeded.\n");

    // Release toolkit resources .
    result = pb_release();

    return result;
}
```

Return Codes

The previous program demonstrates callbacks and some limited error checking. Unlike the NET API, the C API uses return codes to transmit errors and functional failures of all types. Some of these error codes may be used to generate feedback to the user, while others indicate more serious failures. The following table describes the error codes returned by the C API. Those functions with a "YES" in the User Info column may be of use to the program in guiding the user and to adjust program flow. The remainders are generally caused by application implementation error or by an incorrect installation in the target environment.

Error Code	**Description/Probable Cause**	**User Info**
PB_EOK	The function executed correctly.	**YES**
PB_EBUFFER	A supplied buffer is not initialized or too small.	NO
PB_ECANCEL	The function was cancelled by the application.	**YES**
PB_EFATAL	The function has experienced an unspecified critical error.	NO
PB_EBIR	One of the input parameter BIRs is corrupt.	NO
PB_EDATA	The data supplied to the function is invalid.	NO
PB_EREADER	An error occurred when trying to access the fingerprint reader. This is usually due to the specified reader not being properly installed or plugged in.	**YES**
PB_EMEMORY	There was an error while attempting to allocate memory within the toolkit function.	NO
PB_EINIT	The toolkit is not initialized. Usually caused by the application not having initialized the toolkit before attempting to call a biometric function.	NO
PB_ESUPPORT	The requested function parameters are not supported by the toolkit.	NO
PB_EPARAMETER	One or more function parameters are not initialized or out of range.	NO
PB_EBUSY	The reader is already in use.	NO
PB_ETIMEOUT	The functions timeout parameter expired before successful completion of the task.	**YES**
PB_EQUALITY	The image supplied to the toolkit function has insufficient quality to process	**YES**

Table 3 - Returned function error codes.