

Biometric Application Programming Interface (API) for Java Card™

Prepared by:
NIST/Biometric Consortium
Biometric Interoperability, Assurance, and Performance Working Group

Version: 1.1

Document #: 02-0019

7 August 2002
Version 1.1

Revision History:

Version	Date	Initials	Comments
0.0	May, 2002	<i>BB</i>	Draft Initial Version based on presentations made to the BCWG meetings.
1.0	9 th July, 2002	<i>BB</i>	<ol style="list-style-type: none">1. Cast document in Working Group document format.2. Added section to correlate calls between JC Biometric API and BioAPI (for BioAPI audience) – TBD.3. Added section describing typical call sequences for enrolling and matching.
1.1	7 th August, 2002	<i>BB, KK</i>	<ol style="list-style-type: none">4. Added document number5. Addressed comments from Christophe Musial (dt 7/22/2002)6. Updated development process section to include reference to stub API files and export file.7. Updated references to be complete include X9.84

TABLE OF CONTENTS

1.	INTRODUCTION.....	1
1.1	Java Cards	1
1.2	Biometrics Verification.....	2
1.3	Biometric Match on Card.....	3
1.4	A Biometric API for Java Card.....	4
2.	REQUIREMENTS	4
3.	ARCHITECTURE	5
3.1	The JC Biometric API Interfaces & Classes.....	5
3.2	Correlation to BioAPI.....	6
3.3	API Method Call Sequences	6
4.	USING THE JC BIOMETRIC API	6
4.1	Development Process.....	7
5.	ISSUES NOT ADDRESSED BY THE API.....	8
6.	CONCLUSIONS	8
7.	ACKNOWLEDGEMENTS	8
8.	REFERENCES.....	8
	APPENDIX A – Overview of API Methods.....	10
	APPENDIX B – Example Biometric Server & Client Applets	12

ABSTRACT

Java Card has benefited from interoperability both at the binary and the Application Programming Interface (API) level. Biometric technologies can build on this foundation by way of a high level and biometric neutral on-card API. The Java Card Forum has addressed this need with a concise internal application programming interface within the Java Card. Specifically, this API supports secure biometric Match-on-Card so that sensitive biometric data never leaves the card, all while consuming a minimal footprint of memory. This paper describes the requirements, rationale and design of a biometric API for Java Card that evolved under the purview of the Java Card Forum Biometric Task Force and the Biometric Consortium Working Group. The API builds on existing Java Card API designs for security and maximal functionality.

1. INTRODUCTION

As Java Cards grow into today's smallest standardized computing platform, developers wish to ensure the interoperability of many biometric technologies with Java Cards, and to allow multiple, independent applications on a card to access the biometric functionality of that card. The Java Card Forum (JCF) [4] addresses this need with a card-internal API within the Java Card (JC). This paper presents this JC biometric API, and offers guidance for the usage of this API.

The API is developed and accepted by the members of the Java Card Forum Biometric Task Force. It has also been presented to and received feedback from the NIST Biometric Consortium Working Group [9].

1.1 Java Cards

Access to a Java Card's data is usually gained via a PIN code, password, or decryption key; biometrics implemented with match-on-card may also be used. Thus modest amounts of very sensitive data (such as passwords, financial records, and medical records) may be stored on a smart card with much greater safety than, for instance, the hard disk of a desktop system, and may even help avoid the need to transmit this sensitive data over a potentially unreliable network link.

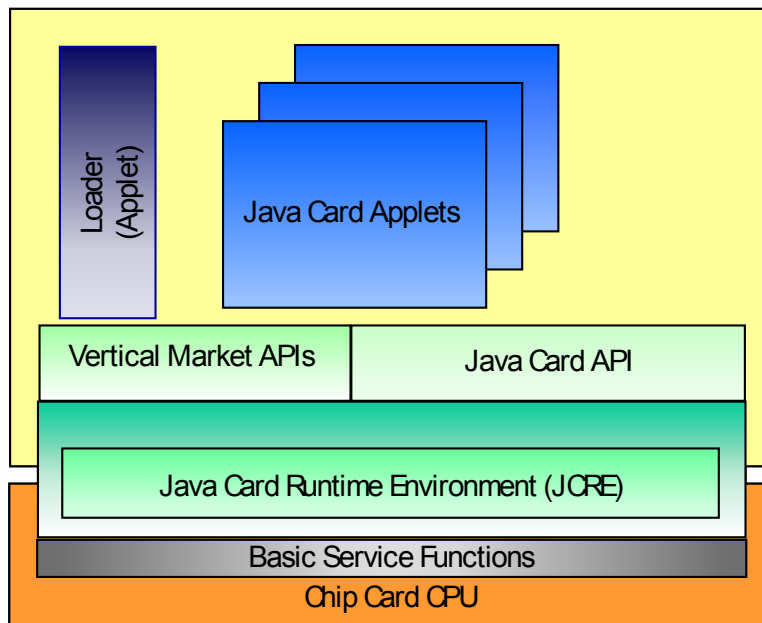


Figure 1. Java Card Architecture

A drawback of Java Card Virtual Machine (JCVM) interpretation of Java programs is that the execution is typically slower than for code compiled directly to the processor's native machine instructions. A native code version of the function (linked through the JCVM) may circumvent this performance penalty. Because native code must be trusted code that is specifically written to respect the Java firewalls, it may only be placed on the card prior to card issuance. Native routines cannot be added after card issuance.

1.2 Biometrics Verification

To use a biometric for either verification, the enrollment acquires an initial biometric image from the individual in question. This image may be processed to extract characteristic features from the image. The image or some characteristic representation of the image is stored securely for future reference. During a verification or identification procedure, a new image is taken from a candidate individual and compared with the enrolled reference, and a decision made as to whether or not the candidate is a match [see figure].

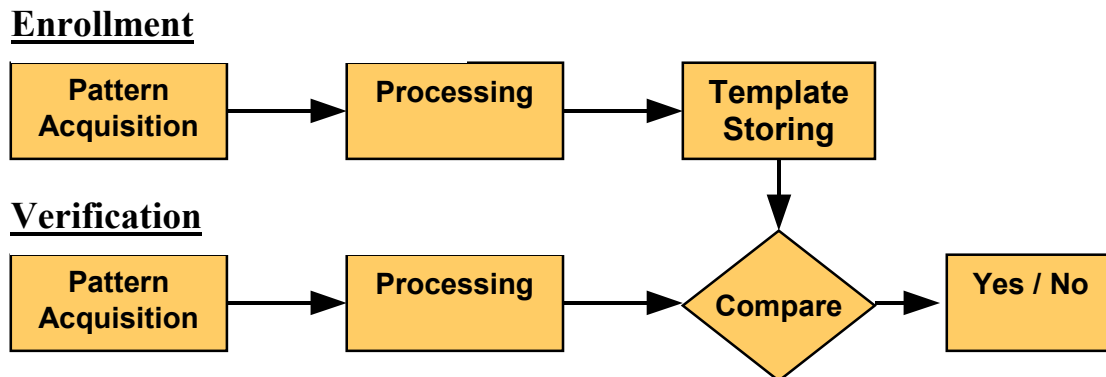


Figure 2. Biometric Data Acquisition and Comparison

Biometric images are usually much larger in data size than passwords, and require much more processing to perform robust correlations of candidate images with enrolled images. While the data and processing requirements are usually small for a modern PC, they can challenge Java Card capabilities.

1.3 Biometric Match on Card

Because applications usually use biometrics to enhance security, they must also handle the biometric data in a secure fashion, much as is the case for computer passwords or PIN codes. This need for secure storage is heightened by the fact that a person's biometric cannot be changed as is done when a password is compromised. Smart cards offer an ideal platform for the handling of biometric data, as their data storage and processing offer great security. If an application stores enrolled biometric data in a central database then a compromise of security on the database would reveal the biometric data for the entire population of users. If, on the other hand, the biometric data is stored on individual smart cards then that risk is avoided. The application then also avoids the risk of transmitting the biometric data over potentially unsafe or failure-prone networks. Yet the sensitive biometric data might still be threatened if it must be extracted from the card for off-card match comparison. For greater security, the match processing may be performed on the card; with this Match-On-Card (MOC), the enrolled biometric data never leaves the secure environment of the card. This is especially attractive when used to access sensitive data inside the card, such as medical records or a signature key; in this case, both the protected information and the security that implements the protection are isolated in the trusted environment of the Java Card. Thus we see a type of nested security where the card and the biometry work in concert to secure each other.

In order to efficiently manage the demands of biometrics with the limited resources of a smart card, an implementation must be frugal in its consumption of memory, it must use processor cycles efficiently, and it must take advantage of common libraries on the card. A raw biometric image typically consumes a large amount of memory; a biometric technology using smart cards will typically extract characteristic features from the image and store only those features on the card, in order to avoid excessive memory consumption. The feature extraction processing can be intensive, and may be

incorporated into the biometric sensor device, or otherwise handled outside the card (e.g., on a PC). The match processing, if done on the card, must be efficiently performed in a carefully crafted algorithm. Because a smart card typically contains libraries for communication, decryption, etc., those libraries should be used rather than duplicated, to avoid excessive code footprint on the card.

Biometrics are often touted for their ability to replace PIN codes or passwords for security, yet they also have a great potential to complement passwords for improved security. This applies both to card-external security, such as building access, and card-internal access, such as the access to electronic cash. Top security concerns often request that access be granted to a person only when that person has all three of 1) something known only to that person, e.g., a password, 2) something physically characteristic of that person, e.g., a biometric, and 3) some token physically possessed by that person, e.g., a smart card. We will see that the JC Biometric API provides the means to easily move forward with such a system.

1.4 A Biometric API for Java Card

Given the many biometric technology vendors and the several vendors of Java Cards, combined with the huge number of potential biometric applications, the prospect of designing special interfaces for each combination of biometric, card, and client application is clearly impractical. A widespread adoption of biometrics on Java Cards will require product interoperability.

Thus, the Java Card Forum presents an on-card Biometric API that allows an application to function on various Java Cards and use various biometric technologies. This paper describes the motivation behind the JC Biometric API and how it can be used to facilitate the development of biometric-enabled applications.

2. REQUIREMENTS

The following requirements served as the basis of the JC Biometric API design. The API must:

- support the abilities to securely enroll a reference biometric on the card and to perform validation of a candidate biometric against a reference without exposing the reference data outside the card.
- be simple and compact, yet the API must also be flexible enough to accommodate a variety of biometric enrollment & matching algorithms currently in use by industry.
- allow multiple biometrics to be enrolled on a card.
- support functionality to protect the security of the biometrics; it must support limits on successive failed match attempts (as is done with the PIN API of Java Card).
- re-use components from existing interfaces such as the PIN API of Java Card. This provides a similar look and feel, and also inherits the applicable security strengths from those components.

- respect the existing features of Global Platform [12], such as the CVM interface.
- allow independent development of biometric technology and clients for that technology. Where possible, the API should agree with existing biometric standards such as BioAPI [13] and CBEFF [7].

3. ARCHITECTURE

In creation of the JC Biometric API, the greatest challenge was to resolve the requirements of a flexible system able to support the large variety of technologies, while at the same time remaining small and simple enough to fit inside the limited resources of a Java Card. The chosen API is designed to support the large majority of technologies with little or no modifications, and it includes only the core functionality for biometrics.

As shown in Figure 3, the API acts as an interface between one or more matching algorithms and the on-card applets that make use of the biometrics. This figure also shows the place of the biometric enabled Java Card within the context of a larger biometric system that includes a biometric sensor and a PC.

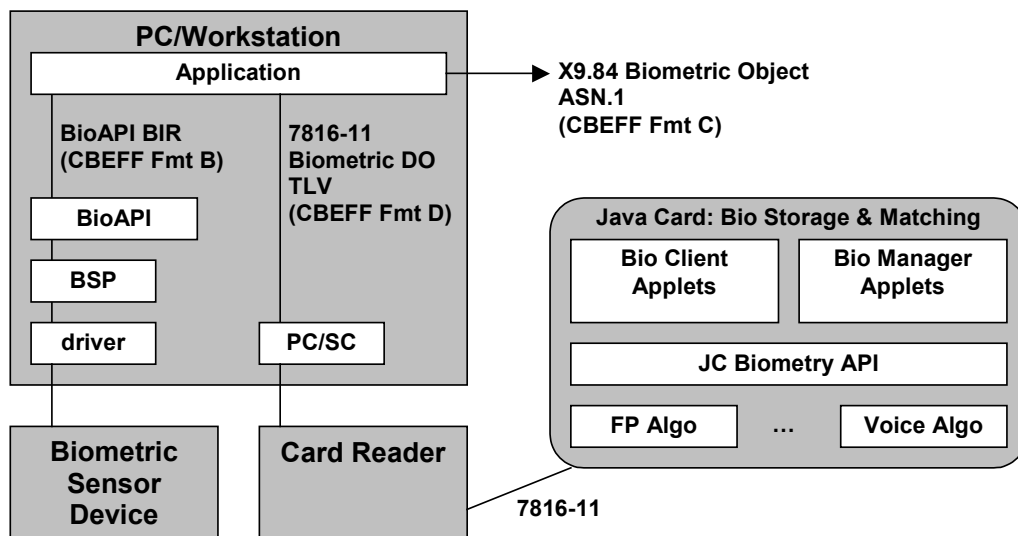


Figure 3. Interactions between various existing biometric standards.

3.1 The JC Biometric API Interfaces & Classes

The JC Biometric API consists of three interfaces and two classes to support biometrics. Emulating the Java Card PIN API design, it includes the BioTemplate interface for matching and the OwnerBioTemplate interface for enrollment. A SharedBioTemplate interface extends BioTemplate with shareability so that a client applet may utilize a biometric that is enrolled by a separate biometric manager or server applet. The BioBuilder class is a factory for new OwnerBioTemplate objects. The BioException class provides exception types specific to biometrics.

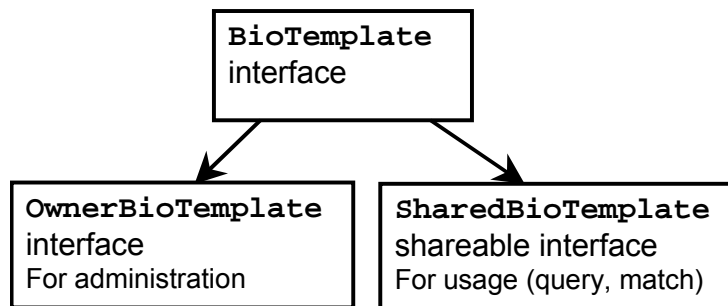


Figure 4. Interfaces in the Java Card Biometric API

The JC Biometric API is constructed as a Java package containing the Java interfaces and classes. This API is described in its entirety in an appendix to this paper.

3.2 Correlation to BioAPI

This section (to be added) describes the correlation of the BioAPI methods and the methods in this JC Biometric API.

3.3 API Method Call Sequences

The enrollment procedure uses the following call sequence (in OwnerBioTemplate):

- `init()` This is required to begin an enrollment.
- `update()` This optionally provides additional enrollment data that was not provided by the `init()` method.
- `doFinal()` This is required to finish an enrollment and make the biometric available for matching.

The verification procedure uses the following call sequence (in BioTemplate):

- `initMatch()` This is required to begin an verification.
- `match()` This optionally provides additional candidate data that was not provided by the `initMatch ()` method.

4. USING THE JC BIOMETRIC API

A biometric application is typically divided into a server applet and a client applet. The server applet manages the enrollment of the biometric templates using the OwnerBioTemplate interface. This server may also provide a proxy to one or more clients using the SharedBioTemplate interface. The client could take many forms and additional clients could be downloaded to the card during the card's lifetime. Thus, the secure usage of biometrics can be safely provided by a server applet to an untrusted client applet, without worry that the client could view or corrupt an enrolled template. Note that for many applications, the server and client may be combined into a single package. The figure below shows these possibilities graphically. Because the API specifies only the core functionality, customized functionality can still be easily added in the biometric server and client packages.

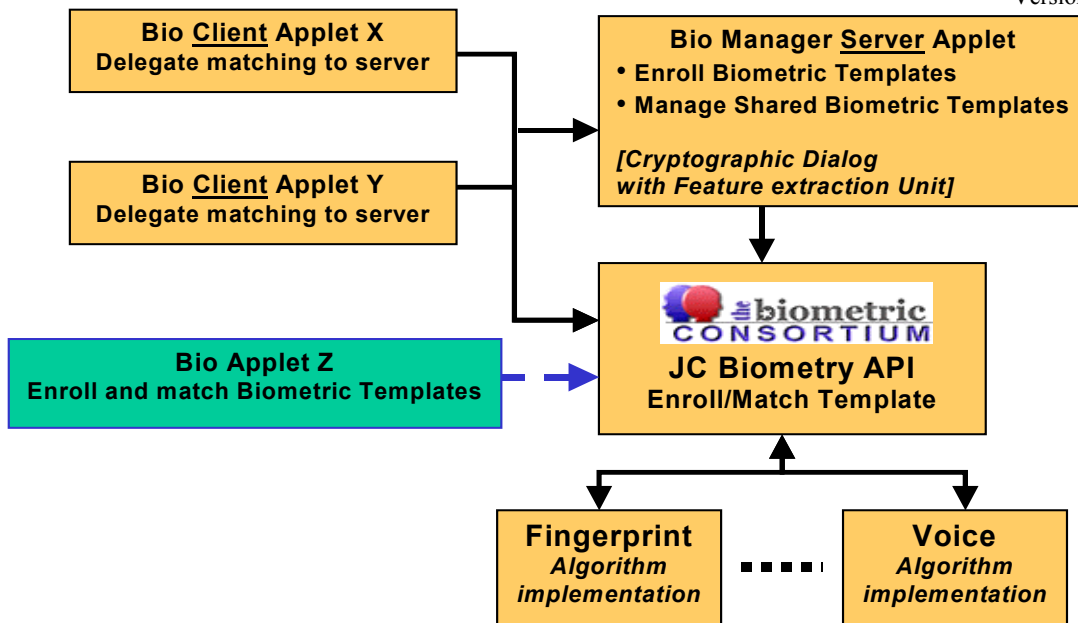


Figure 5. Two possible Biometric Applet Implementation Designs

Another benefit of the JC Biometric API is the ability for a biometric vendor to independently develop an implementation, without need for special assistance (e.g. a mask) from the Card manufacturer or issuer. This allows fast initial development of biometric applications.

In the case when the performance (e.g., speed of matching) requires improvement, the API may be implemented as a service of the underlying operating system with further development of the surrounding biometric support code. The net effect is a significantly shortened total development time.

The actual implementation of the API enrollment and matching methods is of course dependent on the biometric technology. Just as the JCVM interface specifications are the same for everybody while the JCRE implementations are open to the JC manufacturer's preferences, so too does the JC Biometric API provide a standard protocol for the card biometrics; the details of each implementation can vary with the product, and such implementations can be either all Java or Java plus native.

4.1 Development Process

The incorporation of new biometric technologies into the JC biometric API library may be handled as follows. The biometric matching algorithm and sample data is provided to a Java Card mask developer. This developer will connect the algorithm's code to the Java Card runtime environment (JCRE), and make it available via the JC Biometric API. The sample data will be used to test the port of the native code.

Simultaneously, a biometric server applet may be developed. This applet would provide the interface to other applets on the card and to off-card systems. Sample code for such an applet can be found in an appendix to this paper.

Once the biometric code is masked onto a Java Card, it may be tested with the server applet and other components.

The JC Biometric API may be used as is used any other JC API [5]. The API stub sources for compilation and export file for conversion can be found in:
<http://www.javacardforum.org/Documents/biometry>
and may be used for the development of the biometric server and client applets.

5. ISSUES NOT ADDRESSED BY THE API

An existing Biometric enabled JC may be upgraded after issuance with new biometric support, by downloading a new Java library and/or applet. The support for the new biometric would be provided by a new package, where this package includes a class that implements OwnerBioTemplate. This new package would not benefit from the implementations in the preexisting biometry library.

The capability to quantize matching scores (even to the point of a boolean result) aids to prevent hill climbing attacks. The API is compatible with this quantization, yet does not directly address this capability; instead, it leaves that detail to the implementation of the biometric matching algorithm.

6. CONCLUSIONS

The Java Card Forum Biometric API provides a compact and industry approved means for ensuring the interoperability of the core components in biometric systems within a Java Card, while also facilitating rapid product development. The full API is described at the JCF web site: <http://www.javacardforum.org/Documents/JCFBioAPIV1A.pdf>
Java Cards will continue to increase in both computing power and storage capacity; with these developments will come the potential to include ever-more advanced biometric technologies on a Java Card. Particular among these developments will be the support for multiple biometrics on a single card, for instance allowing a card to choose between recognition of a voice or a fingerprint. As the future unfolds, the JC Biometric API may be supplemented with new methods as needed, and we may perhaps someday support even the DNA scans of today's science fiction.

7. ACKNOWLEDGEMENTS

The API presented in this paper is the result of the cooperation of the members of the Java Card Forum biometrics task force, and has benefited from the input of the NIST Biometrics Consortium Working Group.

8. REFERENCES

- [1] Guthery, Scott B., and Jurgensen, Timothy M., Smart Card Developer's Kit, Macmillan Technical Publishing, 1998.
- [2] ISIO-7816, Information Technology - Identification cards – integrated circuit cards with contacts.

- [3] Sun Microsystems Inc., Java Card 2.1 & 2.2 Specifications, <http://java.sun.com/products/javacard/>
- [4] Java Card Forum, <http://www.javacardforum.org>
- [5] Zhiqun Chen. Java Card Technology for Smart Cards: Architecture and Programmer's Guide, Addison-Wesley Publishing, June 2000.
- [6] Biometrics, Access Control, Smart Cards: A not so Simple Combination; G. Hachez, et al. In Josep Domingo-Ferrer, David Chan, Anthony Watson (Eds.): Smart Card Research and Advanced Applications, proceedings of the Fourth Working Conference on Smart Card Research and Advanced Applications, CARDIS 2000, September 20-22, 2000, Bristol, UK. IFIP Series, Vol. 180, Kluwer, 2000, ISBN 0-7923-7953-5
- [7] The Common Biometric Exchange File Format (CBEFF), <http://www.nist.gov/cbeff>
- [8] ISO/IEC CD2 7816-11, Information technology - Identification cards - Integrated circuit(s) cards with contacts - Part 11: Personal verification through biometric methods
- [9] NIST Biometric Consortium Working Group, <http://www.nist.gov/bcwg>
- [10] The Biometrics Resource Center Website, <http://www.itl.nist.gov/div895/biometrics>
- [11] American National Standard X9.84-2001 Biometric Information Management and Security http://www.ncits.org/tc_home/m1htm/docs/m1020002.pdf
- [12] Global Platform, <http://www.globalplatform.org/>
- [13] BioAPI, <http://www.bioapi.org/>

APPENDIX A – Overview of API Methods

Template Creation – The BioBuilder Class

buildBioTemplate

This method is a factory to create new OwnerBioTemplate objects according to the specifications of the biometric application. Once an object is created, enrollment may commence. The buildBioTemplate method will reject requests for biometric types that are not supported on a card.

Biometric Types

The following biometric types are defined in the initial version of the bioBuilder: facial feature, voice print, fingerprint, iris scan, retina scan, hand geometry, written signature, keystroke dynamics, lip movement, thermal face image, thermal hand image, gait style, body odor, DNA scan, ear geometry, finger geometry, palm geometry, vein pattern.

Enrollment Process – The OwnerBioTemplate Interface

init

This required method begins an enrollment procedure for a biometric template object. The call may optionally include data for the enrollment.

update

This optional method continues an enrollment after **init** and before **doFinal**. It may be used to provide enrollment template data beyond that which is provided by the **init** method.

doFinal

This required method performs the final checks on the enrollment and makes it available for match procedures.

resetUnblockAndSetTryLimit

This method unblocks a biometric template and sets the number of successive failed match attempts that cause a biometric template to become blocked.

Match Process – The BioTemplate Interface

Note that SharedBioTemplate is an extension of this class, used to share the match methods of an enrolled biometric without yielding access to the enrollment methods.

getBioType

This identifies at a basic level the biometric type, such as fingerprint, iris, voice, etc.

getVersion

This method returns the version and ID of the matching algorithm in use for the

biometric. Note that this may be distinct from the enrolled template storage format (see `getPublicTemplateData`).

isInitialized

This Boolean method indicates whether a biometric template is fully enrolled and ready to commence a match procedure.

getPublicTemplateData

A public template may contain data about an enrolled template, such as version information, enrolled template storage format, and template processing specifications. This method's use is dependent on the biometric technology and might not be used for all biometric implementations.

initMatch

Resets match status and starts new match procedure. Match configuration or other data may be sent into `initMatch` for this purpose. The method returns a score indicating the status of the match.

match

Continues match procedure, receiving candidate image data and processing that data as appropriate for the biometric technology. The method returns a score indicating the status of the match.

Match Scores: **MINIMUM_SUCCESSFUL_MATCH_SCORE**

Any returned score of this value or higher indicates a successful match. The least significant 14 bits of the returned score may convey other information about the match, as is appropriate for the technology and the application.

Match Scores: **MATCH_NEEDS_MORE_DATA**

This returned score indicates that the match procedure is not complete, and that more candidate data is required.

isValidated

This Boolean method returns true after a successful match and false after a failed match; the value is reset to false by a card tear, a new enrollment, or by a call to the `reset` method.

reset

Resets the reference validated flag to false.

getTriesRemaining

This method returns the number of successive failed attempts that will result in a locked biometric.

APPENDIX B – Example Biometric Server & Client Applets

Biometric Server Example

```
package com.jcf.biometrics.bioServer;

/***** IMPORTS *****/
// Javacard APDU access.
import javacard.framework.*;
// Java Card Forum Biometry API.
import org.javacardforum.javacard.biometry.*;

/**
 * The <b>JCF_SampleBioServer class </b> is a sample manager/owner
 * of biometric templates on a Java card.
 *
 * For this example, a password is used as the biometric template.
 *
 * @version 1.0
 */

public class JCF_SampleBioServer extends javacard.framework.Applet
{
    // ----- Constants -----

    // CLass code in command APDU.
    //private final static byte BIO_SERVER_CLA = (byte)0xNN;
    // INstruction codes for command APDU (APDU header).
    final static byte INS_ISO_CHANGE_REF_DATA = (byte)0x24;
    final static byte INS_ISO_VERIFY = (byte)0x20;
    // Other constants for this example.
    final static byte TRY_LIMIT = (byte)3;

    // ----- Data members -----
    OwnerBioTemplate fullBio; // Full access for server.
    ProxyBioTemplate proxyBioForClient; // Limited access for client.

    // ----- Constructors -----
    /**
     * constructors
     * The JCF_SampleBioServer constructor creates a new bio template object and
     * sets up the shareable interface to that object.
     */
    public JCF_SampleBioServer() {
        proxyBioForClient = new ProxyBioTemplate();
        fullBio = BioBuilder.buildBioTemplate(BioBuilder.PASSWORD, TRY_LIMIT); // Create the
template instance & assign try limit.
        proxyBioForClient.fullBio = fullBio; // Set up limited access for client.
    }

    // ----- install method -----
    /**
     * Applet install method.
     */
    public static void install( byte[] bArray, short bOffset, byte bLength ) throws
ISOException {
        // Instantiate the applet and register it with the JCRE.
        new JCF_SampleBioServer().register();
    }

    // ----- API methods -----
    /**
     * Implement the shareable lookup by client via JCRE.
     */
}
```

7 August 2002
Version 1.1

```
*/
public Shareable getShareableInterfaceObject(AID clientAid, byte parameter) {
    // Use the 'parameter' to select template when multiple templates are enrolled.
    return proxyBioForClient;
}

/**
 * APDU Handling
 */
public void process(javacard.framework.APDU apdu) {
    byte[] apduBuffer = apdu.getBuffer(); // Get the APDU header.
    if (selectingApplet()) return; // Return if the APDU is the applet SELECT command.
    //if (apduBuffer[ISO7816.OFFSET_CLA] != BIO_SERVER_CLA) // Verify the CLA byte.
    // ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
    short numTemplateBytes = apdu.setIncomingAndReceive();
    switch (apduBuffer[ISO7816.OFFSET_INS]) {
        case INS_ISO_CHANGE_REF_DATA: // Enroll a new bio template.
            // Initialize the enrollment process & perform enroll.
            fullBio.init(apduBuffer, ISO7816.OFFSET_CDATA, numTemplateBytes);
            // Continue enrollment of the template.
            // fullBio.update not needed for this example.
            fullBio.doFinal(); // Finalize the enrollment.
            break;
        case INS_ISO_VERIFY:
            // Initialize the match, then execute it.
            // Fully explicit implementation with separate calls to initMatch and match:
            fullBio.initMatch(apduBuffer, ISO7816.OFFSET_CDATA, (short)0);
            fullBio.match(apduBuffer, ISO7816.OFFSET_CDATA, numTemplateBytes);
            // Compact implementation passes candidate data with initMatch:
            // fullBio.initMatch (apduBuffer, ISO7816.OFFSET_CDATA, numTemplateBytes);
            if (!fullBio.isValidated()) // Examine match result.
                ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
            break;
        default:
            ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
    }
}

/**
 * Select method returns true if applet selection is supported.
 * @return boolean status of selection.
 */
public boolean select()
{
    return true;
}
}
```

Biometric Matching Proxy Example

```
package com.jcf.biometrics.bioServer;

/***** IMPORTS *****/
import javacard.framework.*;
import org.javacardforum.javacard.biometry.*;

/**
 * The <b>ProxyBioTemplate class </b> contains the biometric template
 * and provides access to matching functions to other applets.
 */

/* The <b>BioTemplate interface </b> provides to an application
 * the means for accessing biometric matching functionality.
 * This interface does not allow enrollment nor alteration of the reference templates.
 */
public class ProxyBioTemplate implements SharedBioTemplate {

    // ----- Data -----

    OwnerBioTemplate fullBio; // Package accessible and not directly shared outside the
    package.

    // ----- Methods -----

    public boolean isValidated() {
        return fullBio.isValidated();
    }

    public boolean isInitialized() {
        return fullBio.isInitialized();
    }

    public short getPublicTemplateData(short publicOffset,
                                       byte[] dest, short destOffset,
                                       short length) {
        return fullBio.getPublicTemplateData(publicOffset, dest, destOffset, length);
    }

    public byte getBioType() {
        return fullBio.getBioType();
    }

    public short getVersion(byte[] dest, short destOffset) {
        return fullBio.getVersion(dest, destOffset);
    }

    public void reset() {
        fullBio.reset();
    }

    public byte getTriesRemaining() {
        return fullBio.getTriesRemaining();
    }

    public short initMatch(byte[] applicantTemplate, short offset, short length) {
        return fullBio.initMatch(applicantTemplate, offset, length);
    }

    public short match(byte candidate[], short offset, short length) {
        return fullBio.match(candidate, offset, length);
    }
}
```

Biometric Client Example

```
package com.jcf.biometrics.bioClient;

/***** IMPORTS *****/
// Javacard APDU access.
import javacard.framework.*;
// Java Card Forum Biometry API.
import org.javacardforum.javacard.biometry.*;

/*****
 * The <b>JCF_SampleBioClient class </b> is a sample user
 * of biometric technology on a Java card.
 * This client does not enroll, but does access the template for matching.
 * The template is enrolled by the sample BioServer class via OwnerBioTemplate interface.
 *
 * For this example, a password is used as the biometric template.
 *
 * @version 1.0
 */

public class JCF_SampleBioClient extends Applet
{
    // ----- Constants -----
    // CLass code in command APDU.
    //final static byte BIO_CLIENT_CLA                = (byte)0xNN;
    // INStRuction codes for command APDU (APDU header).
    final static byte INS_ISO_VERIFY                = (byte)0x20;
    final static byte CHECK_VERIFY_RESULT          = (byte)0x22;

    final static short SW_ISO_VERIFICATION_FAILED    = (short)0x69C0;

    // ----- Data members -----
    byte expectedServerAID_bytes[] = {0x11,0x22,0x33,0x44,0x55,(byte)0xB0,0x00};
    SharedBioTemplate bioServersBioTemplate;

    // ----- Constructors -----
    /**
     * constructor
     * We need to get from BioManager a shared reference to a template.
     */
    public JCF_SampleBioClient() {
        AID bioManagerAID =
        JCSysTem.lookupAID(expectedServerAID_bytes,(short)0,(byte)expectedServerAID_bytes.length)
        ;
        if (bioManagerAID == null) ISOException.throwIt(SW_ISO_VERIFICATION_FAILED);
        register(); // Register new BioClient cardlet instance with the JCRC.
        bioServersBioTemplate =
        (SharedBioTemplate)JCSysTem.getAppletShareableInterfaceObject(bioManagerAID, (byte)0);
        if (bioServersBioTemplate == null) ISOException.throwIt(SW_ISO_VERIFICATION_FAILED);
    }

    // ----- install method -----
    public static void install( byte[] bArray, short bOffset, byte bLength ) throws
    ISOException {
        // Instantiate the cardlet.
        new JCF_SampleBioClient();
    }

    // ----- API methods -----
    /**
     * APDU Handling
     */
    public void process(javacard.framework.APDU apdu) {
        byte[] apduBuffer = apdu.getBuffer(); // Get the APDU header.
    }
}
```

7 August 2002
Version 1.1

```
if (selectingApplet()) return; // Return if the APDU is the applet SELECT command.
//if (apduBuffer[ISO7816.OFFSET_CLA] != BIO_CLIENT_CLA) // Verify the CLA byte.
// ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
short rxdDataLength = apdu.setIncomingAndReceive(); // Set up JCRE to receive
template data into APDU.
switch (apduBuffer[ISO7816.OFFSET_INS]) {
    case INS_ISO_VERIFY:
        // Attempt match of bio template from APDU data:
        // We assume all the APDU data is template data.
        // Fully explicit implementation with separate calls to initMatch and match:
        bioServersBioTemplate.initMatch(apduBuffer, ISO7816.OFFSET_CDATA, (short)0);
        bioServersBioTemplate.match(apduBuffer, ISO7816.OFFSET_CDATA, rxdDataLength);
        // Compact implementation passes candidate data with initMatch:
        // bioServersBioTemplate.initMatch(apduBuffer, ISO7816.OFFSET_CDATA,
rxdDataLength);
        if (!bioServersBioTemplate.isValidated()) // Examine match result.
            ISOException.throwIt(SW_ISO_VERIFICATION_FAILED); // Match failed.
        break; // Match succeeded.
    case CHECK_VERIFY_RESULT:
        // Check result of previous verify operation.
        if (!bioServersBioTemplate.isValidated()) // Examine match result.
            ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
        break;
    default:
        ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
}
}

/* Select method */
public boolean select() {
    return true;
}
}
```