BioAPI Specification Version 1.1

March 16, 2001

Developed by

The BioAPI Consortium

Specification Disclaimer and Copyright

NOTE: A reference implementation (framework software) of this BioAPI Specification is available (downloadable from the bioapi website, www.bioapi.org).

Contact information:

BioAPI Consortium: http://www.bioapi.org

Chair: Catherine J. Tilton, SAFLINK Corporation, 11417 Sunset Hills Road, Suite 106, Reston, VA 20190, 703-708-9280, fax 703-708-0014, ctilton@saflink.com.

Secretary: Fred Herr, Unisys Corporation, Mail stop B302, Unisys Way, Blue Bell, PA 19424, 215-986-5933, fax: 215-986-7744.

BioAPI Specification Technical Editor: John H. Wilson, Intel Corporation, Platform Security Division, Security Technology Lab, JF3-373, 2111 N.E. 25th Ave., Hillsboro, OR 97124, 503-264-2713, fax 503-264-6225, john.h.wilson@intel.com.

BioAPI Version 1.1
Comment Form

Copyright Grant by Reviewer: Reviewer hereby grants to the BioAPI Consortium, a non-exclusive, royalty free, worldwide license to reproduce, display, perform, prepare derivative works of and distribute Reviewer's Suggestions, in whole or in part, in the Specification in any form, media or technology now known or later developed for the full term of any intellectual property rights that may exist in such Reviewer Suggestions.

Warranty: Reviewer warrants that, to the best of Reviewer's knowledge, Reviewer's Suggestions do not infringe any third party copyrights, trade secrets or patent rights.]

Comments & Suggestions

AGREED:

Reviewer: ______________________________

Company: ________________________________

Title: __________________________________

Date: ___________________________________

E-mail@: ________________________________

Signature: ______________________________

Only comments submitted using the above form will be considered for inclusion in the BioAPI specification, but the form may be submitted by e-mail to the editor: john.h.wilson@intel.com

Major Changes in this Version

Below are listed the major changes between Version 1.1 and Version 1.0 of the BioAPI Specification.

- Integer fields within the BIR header are explicitly specified to be little-endian to facilitate interoperability between heterogeneous systems and to be consistent with the reference implementation.
- Two elements were added to the BSP schema to allow for location of the executable and to provide a displayable description of the BSP.
- A new framework function (BioAPI_EnumModules) was added to allow enumeration of installed BSPs to the application.
- The mask values for BIR_Data_Type mask values were corrected to begin at 0x01.
- BIR-Purpose definition was clarified to be a singular values and its use as an input parameter and field within the BIR header were clarified. Also, restrictions on the use of a BIR for a particular purpose was clarified.
- Extraneous parameters were deleted from the Process, CreateTemplate, and Identify functions.
- Error codes/returns were cleaned up.
- A section describing the module registry was added.
- The conformance section text was modified to match the table. (Primitive functions are optional, but highly recommended.)
- The appendix on HA-API compatibility was deleted, with the exception of explanatory material.

Table Of Contents

FOREWORD

The BioAPI Consortium was formed to develop a widely available and widely accepted API that will serve for various biometric technologies. The BioAPI Consortium first announced its formation and intent to develop a biometric API standard in April of 1998. By the end of the year, this Consortium had developed a multi-level API architecture and begun defining the associated components. In December 1998, I/O Software joined the Consortium and it was decided that the BAPI specification would be integrated as the lower level of the BioAPI specification.

In March of 1999, the Information Technology Laboratory (ITL) of the National Institute of Standards and Technology (NIST) and the US Biometric Consortium sponsored a unification meeting in which the Human Authentication API (HA-API) working group (which had published a high level biometric API in 1997) agreed to merge their activities with the BioAPI Consortium. As part of this agreement, the BioAPI Consortium agreed to restructure their organization.

In March 2000, Version 1.0 of the BioAPI Specification was released. This was followed on April 6th by a BioAPI Users' and Developers' Seminar, sponsored by NIST and the National Security Agency (NSA) and hosted by the Biometric Consortium (www.biometrics.org). In September 2000, the beta version of the BioAPI Reference Implementation was released.

The BioAPI was originally conceived by its founders as a multi-level API, and was the initial framework for discussion when the BioAPI, HA-API and BAPI were merged. The "high level" would include basic calls (e.g., enroll, verify, etc.) that would satisfy the requirements of most applications. Lower levels would address increasing detail, control, sophistication, and technology dependence. In 1999, the group determined to focus on 2 levels - the high (H) level and the lower (D, device) level. The high level definition would comprise the best of HA-API, BioAPI Level H, and BAPI Level 3. The device level would be based on BAPI Level 1 (and some of Level 2). Working groups (the AWG - Applications Working Group, and DWG - Device Working Group) were formed to define these levels. The AWG, with participation from members representing technology vendors, integrators and end users, succeeded in defining a high level interface that satisfied the known requirements of all the participants. On the other hand, proposals for interfaces at lower levels, when studied by the membership as a whole, were not believed to address issues that were broad enough, across the biometrics industry, to justify inclusion in the standard API.

So, in late 1999, it was decided that BioAPI Version 1.0, with its single layer, provided a sufficiently comprehensive and rich feature set that would address the significant market requirements for a generalized API, and Version 1.0 was voted on by the members and published in March 2000.

At the time of publication of this specification, the BioAPI Consortium Steering Committee (SC) had the following members:

SC Member organization	SC Name of representative(s)	Position
SAFLINK Corporation	Cathy Tilton	Chair
Unisys Corporation	Fred Herr	Secretary
Intel	John Wilson	Technical Editor
Compaq Computer Corp.	Manny Novoa	
Iridian Technologies	James Cambier	Treasurer

NIST	Fernando Podio	
Mytec Technolgies, Inc.	Colin Soutar	

There are 4 Working Groups currently operating under the BioAPI Consortium:

Application Level Working Group - Chair: John Wilson, Intel

External Liaisons Working Group – Chair: Fernando Podio, NIST

Reference Implementation Working Group – Chair: Dr. Colin Soutar, Mytec Technologies

Conformance Test Suite Working Group – Chair: Dr. James Cambier, Iridian Technologies

Implementation of the BioAPI will enable:

- Rapid development of applications employing biometrics
- Flexible deployment of biometrics across platforms and operating systems
- Improved ability to exploit price performance advances in biometrics
- Enhanced implementation of multiple biometric alternatives (fingerprint, voice, face, iris, etc.)

The BioAPI will enable these business benefits by providing:

- Simple application interfaces
- Standard modular access to biometric functions, algorithms, and devices
- Secured and robust biometric data management and storage
- Standard methods of differentiating biometric data and device types
- Support for biometric identification in distributed computing environments

Implementation of BioAPI will provide value to:

- Industry security administrators
- System Integrators and value added resellers
- Application developers
- End users of biometric technologies

The official web site for the BioAPI Consortium is: http://www.bioapi.org/. There are no other sites authorized to speak for the consortium.

At the time of this draft specification, the BioAPI Consortium had the following members:

Member organization	**Name of representative(s)**
Acsys Biometrics USA, Inc.	Raj Kalra
Ambition Global Co., Ltd.	Ya Fang Wu
Authentec, Inc.	Jim Waldron
Ankari (formerly American Biometric Company)	Roy Myers (*)
	Bhuwan Pharasi
Barclays Bank	James Gillies
Bergdata USA Inc.	Manuel Bach
BioFinger Tech. Corp.	Julia Chen
BioLink Technologies International	Matt Flynn
Biometix	Ted Dunstone
Biometric Identification Inc. (BII)	Julia Webb
	Brian Schrack

Biometric Verification	Greg Boyko
Biometrics.co.za	Dr. H. M. Kimmel (*)
	Peter Scholtz
BioNetrix	Tony Rochon
BioPassword Security Systems, Inc.	Crystal Webb
BITS Inc.	Christopher J. Oneto (*)
	David Michaelsen
Compaq Computer Corp.	Manny Novoa
Configate, Ltd.	Irene Mikhlin
Datastrip, Inc.	Bill Cawlfield
DCS Dialog Communications Systems, AG	Dr. Ulrich Dieckmann
Digital Persona	Enrica D'Ettorre (*)
	Keith DeBrunner
Ecryp, Inc.	Thomas Anderson
Ethentica	Scott Denis
ETrue, Inc.	Michael Kuperstein
Fidelica Microsystems Inc.	Ramesh Yadava (*)
	Gil Ross
Fingerprint Cards AB	Kenneth Jonsson
Gemplus	Gilles Pauzie
Hewlett Packard	Bruce Encke
	Mitko Mitev
Hunno Technolgies Inc.	Hyun-Sun Yoon
Identification and Verification International	Craig Arndt
Identification Systems DERMALOG GMBH	Peter Breuer
Identix/Identicator	Michael Chaudoin (*)
	Grant Evans
	Yuri Khidekel
	Erik Bowman
Image Computing Incorporated (ICI)	Todd Lowe
Infineon (formerly Siemens Semiconductor)	Brigitte Wirtz (*)
	Thomas Rosteck
	Robert Egger
	Donald Malloy
Intel Corporation	John H. Wilson (*)
	Terry A. Smith
	David Bowler
	Sunanda Menon
I/O Software, Inc.	William Saito (*)
Iridian Technolgies	Dr. James Cambier (*)
	Augusta Fuma
ISC/US	Darrel Geusz
ITT Industries	Dr. Steven Boll
J. Markowitz Consulting	Judith Markowitz
Janus Associates, Inc.	Robert Jones
Kaiser Permanente	Steven Pomerantz
Keyware Technologies	Veronique Wittebolle
	Mik Emmerechts (*)
	M. Koster
LCI SmartPen, N.V.	Bert Heesbeen
Leading Edge Security Ltd.	Colin Maddison
Locus Dialogue	Dorothee Rueff
Logico Smartcard Solutions GMBH	Alexander Keck

Miaxis Biometrics Company	Sunny, Li Sun
Mytec Technologies	Dr. Colin Soutar (*)
	Greg Schmidt
Nanyang Technological University	Dr. Wei-Yun Yau
National Biometrics Test Center	Dr. Jim Wayman
NIST / ITL	Fernando Podio
NSA	Jeffery Dunn (*)
	Cpt. Charlene Schilling
NEC Technologies	Theresa Giordano (*)
	Titikorn (TK) Tapanapaha
	Masahito Ito
	Rei Suwata
	Yukio Hoshino
Neurodynamics Limited	Mike Dell
Oki Electric Industry Co., Ltd.	Toshio Nakamura
OMNIKEY AG	Uwe Schnabel
Precise Biometrics	Krister Walfridsson (*)
	Marten Obrink
Presideo	David Flickinger (*)
	Lee Frey
	Dianne Moss
Raytheon	Eb Clark (*)
	Bob Stroud
Recognition Systems Inc.	Kevin Miller
SAFLINK Corporation	Cathy Tilton (*)
	Walter Hamilton
	Greg Jensen
	Tim Brown
Sagem-Morpho Inc.	Creed Jones (*)
	Herve Jarosv
	D. Friant
Sec2Wireless	Adi Weiser
Secugen Corp.	Josephin Kang (*)
	Kai Huang
	Won Lee
	David Paek
Sensecurity Pte Ltd	Jonathan Lewis Tan
Startek	Grant Lin
	Sammie Lin
	Tori Chang
ST Microelectronics, Inc.	Vito Fabbrizio
Systemneeds, Inc.	Knaka
Techguard Security	James Joyce (*)
	Suzanne Joyce
	Andrea Johnson
	Jeff Johnson
Telework Corporation	Jimmy Chang
Transaction Security, inc.	Rodney Beatson
Transforming Technologies	David Russell (*)
	Mia Bragg
TRW	Brian Sullivan
Unisoft Corporation	Guy Hadland
Unisys	Fred Herr (*)

	Alan Bender
	David Weston
Veridicom	Larry O’Gorman (*)
	Dan Riley
Viatec Research	Anthony T. Rivers
Visionics	Paul Griffin (*)
	Kirsten Nobel

(*) Primary POC

1 Specification Overview

1.1 Purpose

The BioAPI is intended to provide a high-level generic biometric authentication model; one suited for any form of biometric technology.

It covers the basic functions of Enrollment, Verification, and Identification, and includes a database interface to allow a biometric service provider (BSP) to manage the Identification population for optimum performance.

It also provides primitives that allow the application to manage the capture of samples on a client, and the Enrollment, Verification, and Identification on a server.

1.2 Scope

This specification defines the Application Programming Interface and Service Provider Interface for a standard biometric technology interface. It is beyond the scope of this specification to define security requirements for biometric applications and service providers, although some related information is included by way of explanation of how the API is intended to support good security practices.

1.3 Application Level API

The Application Level (formerly referred to as Level H) is the "top" level at which the basic biometric functions are implemented - those which an application would generally use to incorporate biometric capabilities for the purpose of human identification.

The top-level functions are derived from a merging from the following sources:

- HA-API 2.0, dated 22 Apr 98, plus proposed extensions from draft Version 2.02, dated 17 Feb 99
- Draft BioAPI Level H Reference Manual, dated 25 Feb 99
- BAPI SDK Version 1.1, High Level API - Level 3, dated 1 Jun 98
- Draft BioAPI/UAS Specification Release 1.0 Version 1.2, dated April 99

It is intended that the application level API contain all those functions **required** by an application for biometric authentication. Therefore, to the extent possible, the amount of optional functionality is kept to a minimum. The major optional function is Identify; only specialized BSPs will implement this function for large populations, and the database capability is in the interface primarily to allow the BSP to manage these large populations.

The approach taken is to hide, to the degree possible, the unique aspects of individual biometric technologies, and particular vendor implementations, products, and devices, while providing a high-level abstraction that can be used within a number of potential software applications. Access to the biometric mechanisms is through the set of standard interfaces defined in this document. Theoretically, BSPs supplied by vendors conforming to this interface specification could then be used within any application developed to this BioAPI definition.

The BioAPI is designed for use by both application developers and biometric technology developers. To make the integration of the technology as straightforward and simple as possible (thus enhancing its commercial viability), the approach taken is to hide or encapsulate to the extent possible the complexities of the biometric technology. This approach also serves to extend the generality of the interface to address a larger set of potential biometric technologies and applications.

This specification is designed to support multiple authentication methods, both singularly and when used in a combined or "layered" manner.

1.4 Biometric Technology

The basic model is the same for all types of biometric technology. First, the initial registration "template" of the user has to be constructed. This is done by collecting a number of samples through whatever sensor device is being used. Salient features are extracted from the samples, and the results combined into the template. The construction of this initial template is called Enrollment. The algorithms used to construct a template are usually proprietary. This initial template is then stored by the application, and essentially takes the place of a password.

Thereafter, whenever the user needs to be authenticated, live samples are captured from the device, processed into a usable form, and matched against the template that was enrolled earlier. This form of biometric authentication is called Verification, since it verifies that a user is who he says he is (i.e., verifies a particular asserted identity). Biometric Technology, however, allows a new form of authentication called Identification. In this form, a user does not have to assert an identity; the biometric service provider compares the processed samples of the user against a specified population of templates, and decides which ones match most closely. Depending on the match probabilities, the user's identity could be concluded to be that corresponding to the template with the closest match.

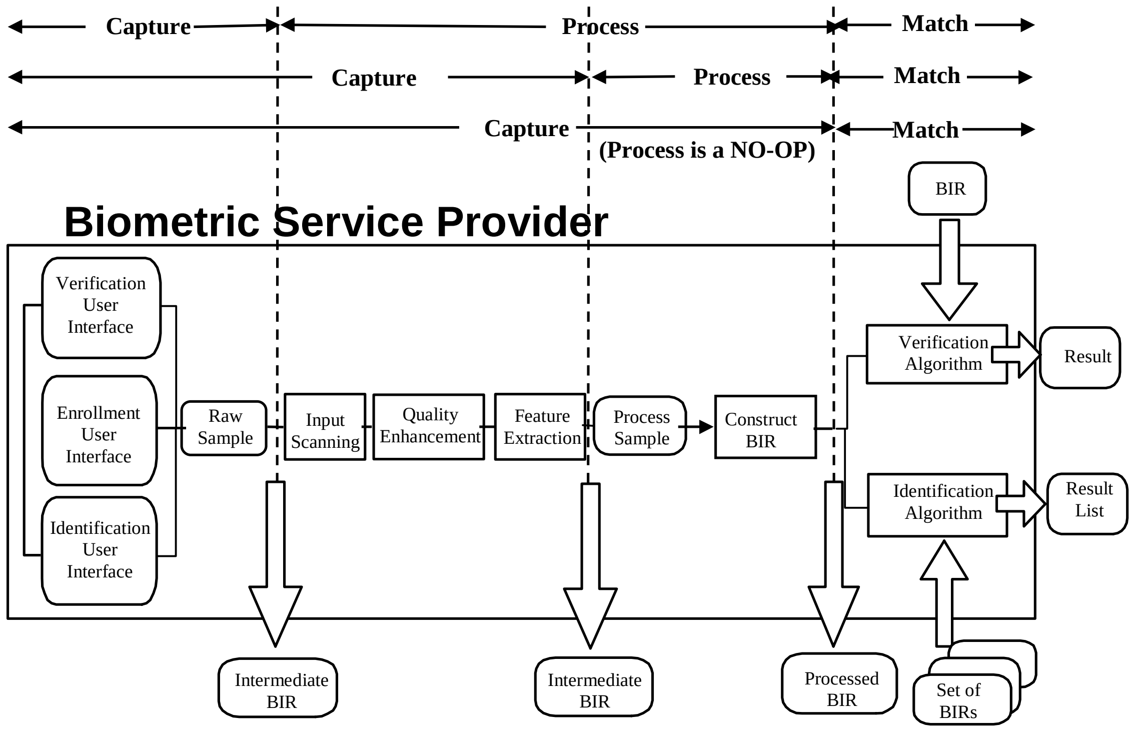

Figure 1. Possible Implementation Strategies

Figure 1 shows some possible implementation strategies. The various steps in the verification and identification operations are shown in the box labeled **Biometric Service Provider.** The stages identified above the box refer to the primitive functions of the top-level interface: **Capture**, **Process** and **Match**, and it shows that a BSP has degrees of freedom in the placement of function in these primitives. There is a degree of freedom not shown in the figure; the manufacturer is free to put most, if not all the BSP function in the sensing device itself. In fact, if the device contains a BIR database, all functions may be performed in the device.

1.5 BIRs and Templates

This standard uses the term **template** to refer to the biometric enrollment data for a user. The template must be matched (within a specified tolerance) by **sample**(s) taken from the user, in order for the user to be **authenticated**.

The term **biometric identification record** (BIR) refers to any biometric data that is returned to the application; including raw data, intermediate data, processed sample(s) ready for verification or identification, as well as enrollment data. Typically, the only data stored persistently by the application is the BIR generated for enrollment (i.e., the template). The structure of a BIR is shown in Figure 2 below.

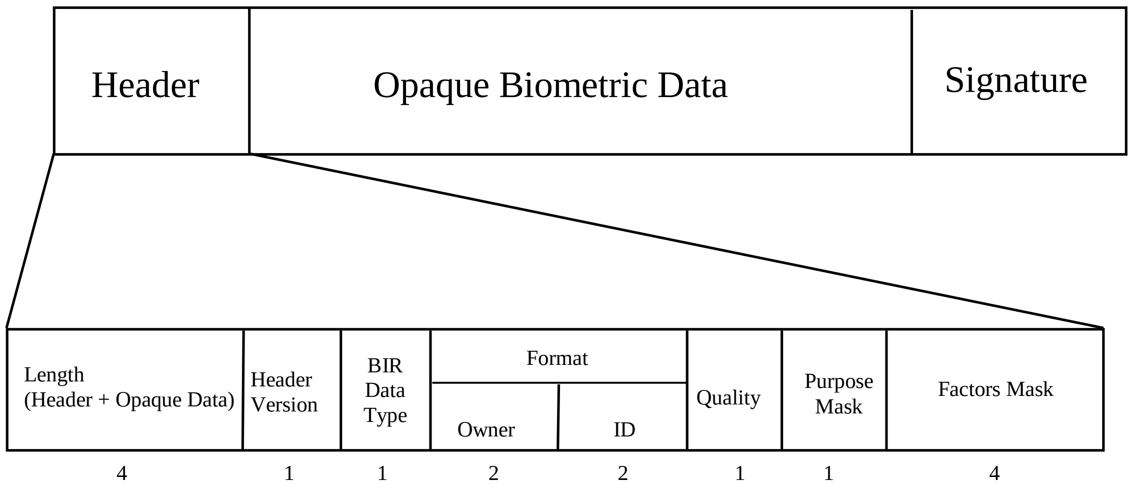

Figure 2. Biometric Identification Record (BIR)

The format of the Opaque Biometric Data is indicated by the Format field of the Header. This may be a standard or proprietary format. The Opaque data may be encrypted.

The BioAPI BIR definition is compliant with the “Common Biometric Exchange File Format (CBEFF)”, of which it is one of the CBEFF Patron Formats. CBEFF is described in the National Institute of Standards Publication, NISTIR 6529, January 3, 2001, developed by the CBEFF Technical Development Team (F. L. Podio, NIST, J.S. Dunn, NSA, L. Reinert, NSA, C. J. Tilton, SAFLINK, L. O'Gorman, Veridicom, M. P. Collier, The Biometric Foundation, M. Jerde, ANADAC, and B. Wirtz, Infineon). A description of CBEFF can be found at http://www.nist.gov.cbeff and in the IBIA web page: http://www.ibia.org/formats.htm.

Values of Format Owner are assigned and registered by the International Biometric Industry Association (IBIA), which ensures uniqueness of these values. Registered format owners then create one or more Format IDs (either published or proprietary), corresponding to a defined format for the subsequent opaque biometric data, which may optionally also be registered with the IBIA. Organizations wishing to register as a CBEFF/BioAPI biometric data format owner can do so for a nominal fee by contacting the IBIA at:

Website Registration: www.ibia.org/formats.htm
or
Telephone 202-783-7272 Fax 202-783-4345
601 Thirteenth Street, N.W., Suite 370 South, Washington, D.C. 20005
General Information: ibia@ibia.org

The signature is optional. When present, it is calculated on the Header + Opaque Biometric Data. For standardized BIR formats, the signature will take a standard form (to be determined when the format is standardized). For proprietary BIR formats (all that exists at the present time), the signature can take any form that suits the BSP. For this reason, there is no C structure definition of the signature.

The BIR Data Type indicates whether the BIR is signed and/or encrypted.

When a service provider creates a new BIR, it returns a handle to it. Most local operations can be performed without moving the BIR out of the service provider. [BIRs can be quite large, so this is a performance advantage.] However, if the application needs to manage the BIR (either store it in an application database, or send it to a server for verification/identification), it can acquire the BIR using the handle.

Whenever an application needs to provide a BIR as input, it can be done in one of three ways:

a) By reference to its handle
b) By reference to its key value in an open database
c) By supplying the BIR itself.

1.6 The API Model

There are three principal high-level **abstraction** functions in the API:

1) **Enroll**
Samples are **capture**d from a device, **process**ed into a usable form from which a **template** is constructed, and returned to the application.
2) **Verify**
One or more samples are **capture**d, **process**ed into a usable form, and then **match**ed against an input **template**. The results of the comparison are returned.
3) **Identify**
One or more samples are **capture**d, **process**ed into a usable form, and **match**ed against a **set of templates**. A list is returned showing how close the samples compare against the top candidates in the set.

However, as Figure 1 shows, the processing of the biometric data from the capture of raw samples to the matching against a template may be accomplished in many stages, with much CPU-intensive processing. The API has been defined to allow the biometric developer the maximum freedom in the placement of the processing involved, and allows the processing to be shared between the client machine (which has the biometric device attached), and a server machine. It also allows for self-contained devices, in which all the biometric processing can be done internally. Client/server support by BSPs is optional.

There are several good reasons why processing and matching may take place on a server:

1. The algorithms will execute in a more secure environment
2. The Client PC may not have sufficient power to run the algorithms well.
3. The user database (and the resources that it is protecting) may be on a server.
4. Identification over large populations can only reasonably be done on a server.

Two methods are provided to support client/server processing.

1. Using **Primitive** functions.

There are four **primitive** functions in the API which, when used in sequence on client and server, can accomplish the same result as the high-level abstractions:

Capture
Capture is always executed on the client machine; attempting to execute Capture on a machine without a biometric device will return “function not supported”. One or more samples are acquired (either for Enrollment, Verification or Identification). The Capture function is allowed to perform as much processing on the sample(s) as it sees fit, and may, in fact, for verification or identification, complete the construction of the BIR. If processing is incomplete, Capture returns an “intermediate” BIR; indicating that the Process function needs to be called. If processing is complete, Capture returns a “processed” BIR; indicating that the Process function does not need to be called. The application specifies the purpose for which the samples are intended, giving the BSP the opportunity to do special processing. This purpose is recorded in the header of the constructed BIR.

Process
The “processing algorithms” must be available on the server, but **may** also be available on the client. The Process function is intended to provide the processing of samples necessary for the purpose of verification or identification (not enrollment). It always takes an “intermediate” BIR as input, and may complete the processing of the biometric data into “final” form suitable for its intended purpose. On the client, if it completes the processing, it returns a “processed” BIR; otherwise it returns an “intermediate” BIR; indicating that Process needs to be called on the Server. On the server, it will always complete processing, and always return a “processed” BIR. The application can always choose to defer processing to the server machine, but may try to save bandwidth (and server horsepower) by calling Process on the client. [“Processed” BIRs are always smaller than “intermediate” BIRs; by how much is technology dependent, and also dependent on how much processing has already been done by Capture].

Match
Performs the actual comparison between the “processed” BIR and one template (**VerifyMatch**), or between the “processed” BIR and a set of templates (**IdentifyMatch**). The support for IdentifyMatch is optional, but the supported Match functions are always available on the server, and **may** be available on the client.

CreateTemplate
CreateTemplate is provided to perform the processing of samples for the construction of an enrollment template. CreateTemplate always takes an “intermediate” BIR as input, and constructs a template (i.e., a “processed” BIR with the recorded purpose of either “enroll_verify” and/or “enroll_identify”). Optionally, CreateTemplate can take an old template and create a new template, which is the adaptation of the old template using the new biometric samples in the “intermediate” BIR. The BSP may optionally allow the application to provide a “payload” to wrap inside the new template. (See section 1.8).

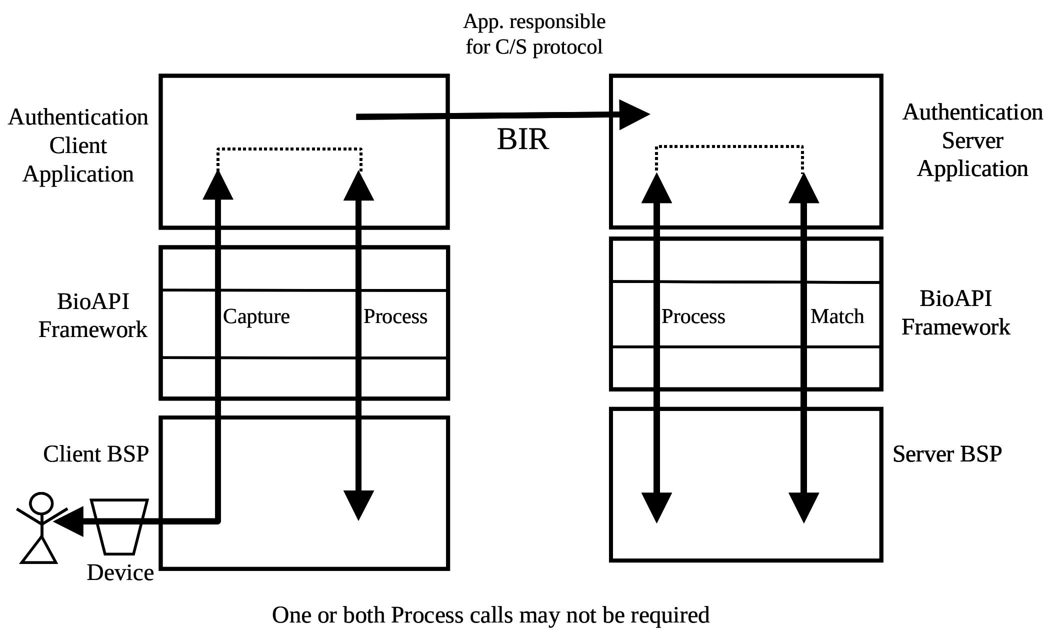

Figure 3. Client/Server Implementation Using Primitive Functions

2. Using a **Streaming Callback**

The application (on both the client and the server) is responsible for providing a streaming interface for the BSP to use to communicate the samples and return the results. In this case, the application does not need to use the **Capture**, **Process** and **Match** primitives. The **Verify**, **Identify**, and **Enroll**, functions use the streaming interface to split the BSP function between client and server. These functions may be driven from either the client or the server. In either the case, if there are Graphical User Interface (GUI) callbacks set, the client BSP will call them at the appropriate times to allow the client application to control the look and feel of the user interface.

a) The client/server application decides whether the authentication should be driven by the client or the server component. The driving component first sets a Streaming Callback interface for the BSP. This not only tells the BSP that it is going to operate in client/server mode, but also provides the interface that it will use to initiate communication with its partner BSP.

b) The application calls the appropriate high-level function, and the BSP calls the Streaming Callback to initiate the BSP-to-BSP protocol. (The protocol is the concern of the BSP implementer, but it will likely start with mutual authentication and key agreement).

c) The **Streaming Callback** is only used by the driving BSP. Whenever it is in control, and has a message to deliver to its partner BSP, it calls the Streaming Callback interface to send the message, and it receives an answer on return from the callback.

d) The **StreamInputOutput** function is used by the partner application to deliver messages to the partner BSP, and to obtain a return message to send to the driving BSP. The driving application delivers the return message by returning from the Streaming Callback.

Note: A client BSP that is servicing a self-contained device may ignore the Streaming Callback interface, and perform the requested function locally. A server BSP that is servicing a self-contained device will use the Streaming Callback to request that the client BSP perform the requested function.

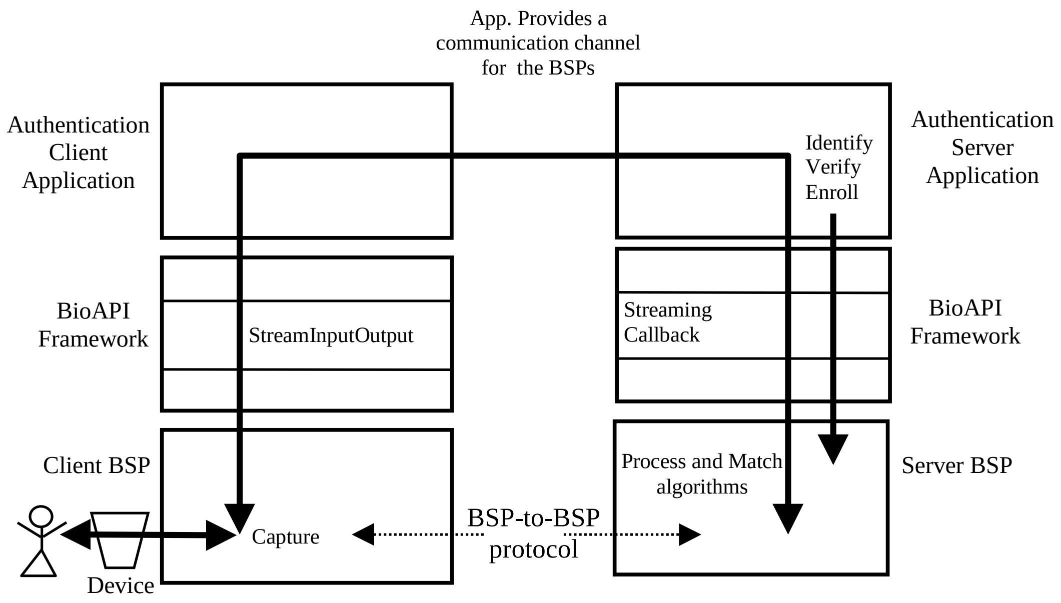
App. Provides a communication channel for the BSPs
Authentication Client Application
Authentication Server Application
Identify
Verify
Enroll
BioAPI Framework
StreamInputOutput
Streaming Callback
BioAPI Framework
Client BSP
Process and Match algorithms
Server BSP
Capture
BSP-to-BSP protocol
Device

Figure 4. Client/Server Implementation Using Streaming Callback - Server Initiated Operation

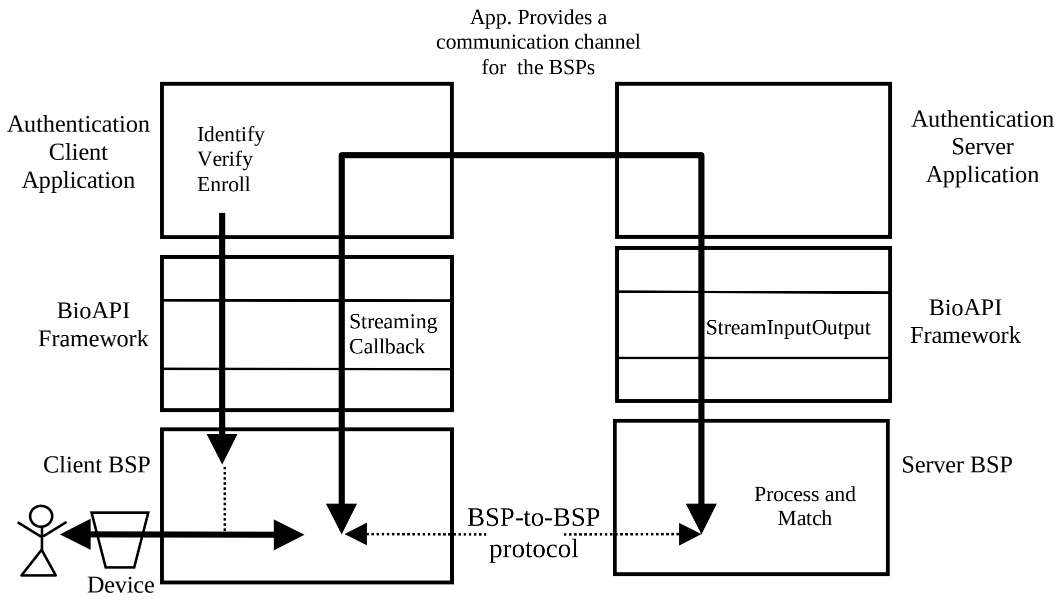

Figure 5. Client/Server Implementation Using Streaming Callbacks – Client Initated Operation

1.7 FAR and FRR

Raw biometric samples are the complex analog data streams produced by sensing devices. No two samples from a user are likely to be identical. Templates are the digital result of processing and compressing these samples; they are not precise representations of the user. Therefore, the results of any matching of samples against a stored template can only be expressed in terms of probability.

There are two possible criteria for the results of a match: False Accept Rate (FAR) and False Reject Rate (FRR). FAR is the probability that samples falsely match the presented template, whereas FRR is the probability that the samples are falsely rejected (i.e., should match, but don't). Depending on the circumstances, the application may be more interested in one than the other.

The BioAPI functions allow the application to request a match threshold in terms of maximum FAR value (i.e., a limit on the probability of a false match,) and an optional maximum FRR value. If both are provided, the application must tell the BSP which one should take precedence.

The principal result returned is the actual FAR value achieved by the match (i.e., the score). A BSP may optionally return the actual FRR value achieved. For **Identify** and **IdentifyMatch**, these results are contained in the Candidate array.

The returning of scores to an application can be a security weakness if appropriate steps are not taken. This is because a "rogue" application can mount a "hillclimbing" attack by sequentially randomly modifying a sample and retaining only the changes that produce an increase in the returned score. In this way a synthetic image can be created to fool the biometric system. However, allowing only discrete increments of score (FAR) to be returned to the application eliminates this method of attack. The level of quantization required to neutralize this attack is dependent on the type of biometric and the algorithms used.

Use of FAR/FRR values to represent match scores is done to allow a degree of normalization and comparison between differing technologies and to allow a commonly understood means of setting thresholds and interpreting results. It is not intended to imply strict performance measurement (that is, an absolute measure of FAR for a specific individual matching instance). Furthermore, the BSP vendor is responsible for accurately mapping internal scoring structure to the FAR values.

1.8 Payloads

No two biometric samples from a user are likely to be identical. For this reason, it is not possible to directly use biometric samples as cryptographic keys. The BioAPI, however, allows a template to be closely bound to a cryptographic key, which could be released upon successful verification.

In the **Enroll** and **Create Template** functions, the application may present a “payload” to be carried in the opaque data of the BIR that is being constructed. This payload is essentially wrapped inside the biometric data by the BSP. This “payload” is released to the application on successful verification of the template. The BSP may have a policy of only releasing the “payload” if the Actual FAR achieved is below a certain threshold (this threshold being recorded in the BSP’s registry entry).

The “payload” can be any data that is useful to the application; it does not have to be cryptographic; and even in a cryptographic application it could be either a key label or a wrapped key.

1.9 BIR Databases

The BioAPI does not manage user databases; only applications do that. In most cases, the user database may already exist (e.g., a database of bank accounts, the user registry of people who belong to a network domain, or those authorized to access a web server), and the biometric application is simply associating a biometric template with each user in the database, in addition to (or as a substitute for) a password. It is important that the application maintain control over who can access this database.

The BioAPI allows a BSP to manage a database of BIRs for two reasons:

1. To optimize the performance of the Identification operation over large populations
2. To provide access to the BIRs that may be stored on a self-contained sensing device.

It is the responsibility of the application to make any necessary association between the BSP's database(s) and the user database(s). To assist in this, each entry in a BSP database has a UUID associated with it. For security reasons, entries in a BSP database cannot be modified; only created and deleted. New entries get new UUIDs.

Not all service providers support identification, and not all those need to support a database interface. If the identification population is sufficiently small, it can be handled by passing an array of BIRs across the interface.

Databases can be created by name, and a service provider may have a default database. If a service provider supports a device that can store BIRs, then that device should be the default database. The default database is always open when the BSP is attached, and the handle to the open default database is always zero.

1.10 User Interface Considerations

The user interface for passwords and PINs is quite straightforward, but for biometric technology it can be quite complex and very much technology dependent, requiring multiple implementation-dependent interactions with the user. Some biometric technologies present streams of data to the user (face and voice, for example), while some require the user to validate each sample taken (face, voice, and signature, for example). During enrollment, some technologies verify each sample taken against the previous samples. The number of samples taken for a particular purpose may vary from technology to technology, and finally, the user interface is generally different for enrollment than for verification and identification.

Most biometric service providers come with a built-in user interface, and this may often be sufficient for most purposes. The API, however, allows the application to control the "look and feel" of this user interface, by allowing the application to provide callbacks for the service provider to use to present and gather samples.

One of the callbacks is used to present and gather samples and to indicate state changes to the application. All service providers implementing the "Application Controlled GUI" option must support this callback, though the state machines may vary considerably. The other GUI callback is used to present streaming data to the user, in the form of a series of bitmaps. This callback is optional, and the service provider indicates in its registry entry whether one is required. This entry also indicates whether the user must validate samples, and whether samples are verified.

The service provider is in control of the user interface state machine, and calls the state callback whenever there is a state change event. These state changes may be the completion of a sample, progress on a sample, or the need to present the user with a message. On return, the application can present the service provider with a response from the user; cancel, continue, valid sample, invalid sample, etc.

If the service provider needs to present a sample stream to the user, it calls the streaming callback in parallel to the state change events. This callback will require multi-threading in both the service provider and the application.

1.11 Module Registry

Upon installation, BioAPI components (framework and BSPs) post information about themselves in the BioAPI module registry. This information is used by the application to determine if the BioAPI framework has been installed. It is also used by the application and framework to determine what BSP devices have been installed and what the capability of these modules are. It can also be used to identify what devices have been attached. The application can use this information dynamically in its decision logic. For example, it can decide whether or not to make a specific (optional) function call to a given BSP depending on whether the registry entry for that BSP indicates that the call is supported. Additionally, it provides information regarding the biometric data formats supported by the BSP (see section 1.5) and default values of the BSP for various common parameters (such as timeouts).

The BioAPI module registry is designed to be platform independent and does not use the built-in registry of any specific operating system (such as the Windows registry). The specifics of the module registry are contained within the reference implementation.

The content (schema) of the module registry is described in section 2.2.2.

2 BioAPI - API Definition

2.1 BioAPI Data Structures

2.1.1 BioAPI

Definition of the BioAPI calling conventions.

```
#ifdef (WIN32)
#define BioAPI __stdcall
#else
#define BioAPI
#endif
```

2.1.2 BioAPI_BIR

A container for biometric data. A BIR may contain raw sample data, partially processed (intermediate) data, or completely processed data. It may be used to enroll a user (thus being stored persistently), or may be used to verify or identify a user (thus being used transiently).

The opaque biometric data is of variable length, and may be followed by a signature. The signature itself may not be a fixed length, depending on which signature standard is employed. The signature is calculated on the combined Header and BiometricData.

```
typedef struct bioapi_bir {
	BioAPI_BIR_HEADER  Header;
	BioAPI_BIR_BIOMETRIC_DATA_PTR BiometricData;	/* length indicated in header */
	BioAPI_DATA_PTR Signature;		/* NULL if no signature; length is inherent in this type */
} BioAPI_BIR, *BioAPI_BIR_PTR;
```

2.1.3 BioAPI_BIR_ARRAY_POPULATION

An array of BIRs, generally used during identification operations (as input to ***Identify*** or ***Identify_Match***).

```
typedef struct bioapi_bir_array_population {
	uint32 NumberOfMembers;
	BioAPI_BIR_PTR   *Members;		/* A pointer to an array of BIR pointers */
} BioAPI_BIR_ARRAY_POPULATION, *BioAPI_BIR_ARRAY_POPULATION_PTR;
```

2.1.4 BioAPI_BIR_AUTH_FACTORS

A mask that describes the set of authentication factors supported by an authentication service.

```
typedef uint32 BioAPI_BIR_AUTH_FACTORS;

#define      BioAPI_FACTOR_MULTIPLE           (0x00000001)
```

```
#define       BioAPI_FACTOR_FACIAL_FEATURES    (0x00000002)
#define       BioAPI_FACTOR_VOICE              (0x00000004)
#define       BioAPI_FACTOR_FINGERPRINT        (0x00000008)
#define       BioAPI_FACTOR_IRIS               (0x00000010)
#define       BioAPI_FACTOR_RETINA             (0x00000020)
#define       BioAPI_FACTOR_HAND_GEOMETRY      (0x00000040)
#define       BioAPI_FACTOR_SIGNATURE_DYNAMICS (0x00000080)
#define       BioAPI_FACTOR_KEYSTOKE_DYNAMICS  (0x00000100)
#define       BioAPI_FACTOR_LIP_MOVEMENT       (0x00000200)
#define       BioAPI_FACTOR_THERMAL_FACE_IMAGE (0x00000400)
#define       BioAPI_FACTOR_THERMAL_HAND_IMAGE (0x00000800)
#define       BioAPI_FACTOR_GAIT               (0x00001000)
#define       BioAPI_FACTOR_PASSWORD           (0x80000000)
```

NOTE: All integer values in the BIR header are little-endian.

2.1.5 BioAPI_BIR_BIOMETRIC_DATA

This comprises the “opaque” data block within a BIR containing the biometric sample(s) or template(s).

```
typedef uint8 BioAPI_BIR_BIOMETRIC_DATA;
```

NOTE: The format of this data is specified by the format field (BioAPI_BIR_BIOMETRIC_DATA_FORMAT) in the BIR header. See Section 1.5.

2.1.6 BioAPI_BIR_BIOMETRIC_DATA_FORMAT

Defines the format of the data contained within the “opaque” data block, BioAPI_BIR_BIOMETRIC_DATA.

```
typedef struct bioapi_bir_biometric_data_format {
       uint16 FormatOwner;
       uint16 FormatID;
} BioAPI_BIR_BIOMETRIC_DATA_FORMAT, *BioAPI_BIR_BIOMETRIC_DATA_FORMAT_PTR;
```

NOTE: FormatOwner values are assigned and registered by the International Biometric Industry Association (IBIA). FormatType is assigned by the Format Owner and may optionally be registered by the IBIA. Contact information for the IBIA is located in Section 1.5 of this specification.

NOTE: All integer values in the BIR header are little-endian.

2.1.7 BioAPI_BIR_DATA_TYPE

Mask bits that may be OR’d together to indicate the type of opaque data in the BIR. (Raw OR Intermediate OR Processed) OR Encrypted OR Signed.

```
typedef uint8 BioAPI_BIR_DATA_TYPE;
#define       BioAPI_BIR_DATA_TYPE_RAW                        (0x01)
#define       BioAPI_BIR_DATA_TYPE_INTERMEDIATE               (0x02)
#define       BioAPI_BIR_DATA_TYPE_PROCESSED                  (0x04)
#define       BioAPI_BIR_DATA_TYPE_ENCRYPTED                  (0x10)
#define       BioAPI_BIR_DATA_TYPE_SIGNED                     (0x20)
```

NOTE: All integer values in the BIR header are little-endian.

2.1.8 BioAPI_BIR_HANDLE

A handle to refer to BIR data that exists in the service provider.

```
typedef  sint32 BioAPI_BIR_HANDLE, *BioAPI_BIR_HANDLE_PTR;
#define      BioAPI_INVALID_BIR_HANDLE                      (-1)
#define      BioAPI_UNSUPPORTED_BIR_HANDLE                  (-2)
```

2.1.9 BioAPI_BIR_HEADER

```
typedef struct bioapi_bir_header {
      uint32 Length;   /* Length of Header + Opaque Data */
      BioAPI_BIR_VERSION HeaderVersion;
      BioAPI_BIR_DATA_TYPE Type;
      BioAPI_BIR_BIOMETRIC_DATA_FORMAT Format;
      BioAPI_QUALITY Quality;
      BioAPI_BIR_PURPOSE Purpose;
      BioAPI_BIR_AUTH_FACTORS FactorsMask;
} BioAPI_BIR_HEADER, *BioAPI_BIR_HEADER_PTR;
```

NOTE: All integer values in the BIR header are little-endian.

2.1.10 BioAPI_BIR_PURPOSE

A value which defines the purpose(s) or use(s) for which the BIR is intended (when used as an input) or suitable (when used as an output or within the BIR header).

```
typedef uint8 BioAPI_BIR_PURPOSE;
#define      BioAPI_PURPOSE_VERIFY                                (1)
#define      BioAPI_PURPOSE_IDENTIFY                              (2)
#define      BioAPI_PURPOSE_ENROLL                                (3)
#define      BioAPI_PURPOSE_ENROLL_FOR_VERIFICATION_ONLY          (4)
#define      BioAPI_PURPOSE_ENROLL_FOR_IDENTIFICATION_ONLY        (5)
#define      BioAPI_PURPOSE_AUDIT                                 (6)
```

NOTE: All integer values in the BIR header are little-endian.

NOTE: The Purpose value is utilized in two ways. First, it is used as an input parameter to allow the application to indicate to the BSP the purpose that the resulting data is intended for, thus allowing the BSP to perform the appropriate capturing and/or processing to create the proper BIR for this purpose. The second use is within the BIR header to indicate to the application (or to the BSP during subsequent operations) what purposes the BIR is suitable for. For example, some BSPs use different BIR formats depending on whether the data is to be used for verification or identification, the latter generally including additional information to enhance speed or accuracy. Similarly, many BSPs use different data formats depending on whether the data is to be used as a sample for immediate verification or as a reference template for future matching (i.e., enrollment.

Restrictions on the use of BIR data of a particular purpose include:

a) All purposes are valid in the BIR header.
b) Purposes of Verify and Identify are only valid as input to the ***Capture*** function.
c) Purposes of Enroll, Enroll_for_Verification_Only, and Enroll_for_Identification_Only are only valid as input to the ***Capture***, ***Enroll***, and ***Import*** functions.
d) The Audit purpose is not valid as input to any function, but is only used in the BIR header.
e) The ***Process*** and ***Create_Template*** functions do not have Purpose as an input parameter, but read the Purpose field from the BIR header of the input Captured_BIR.
f) The ***Process*** function may accept as input any intermediate BIR with a Purpose including Verify or Identify, and will output only BIRs with a Purpose of Verify and/or Identify.

g) The Create_Template function may accept as input any intermediate BIR with a Purpose including Enroll, Enroll_for_Verification_Only, and/or Enroll_for_Identification, and will output only BIRs with a Purpose including that of the input BIR.
h) If a BIR is suitable for enrollment for either subsequent verification or identification, then the Enroll Purpose is to be used in the returned BIR header.

2.1.11 BioAPI_BIR_VERSION

This data type is used to represent the version of a BIR header. The first version has a value of 1.

```
typedef  uint8  BioAPI_BIR_VERSION, *BioAPI_BIR_VERSION_PTR;
```

NOTE: All integer values in the BIR header are little-endian.

2.1.12 BioAPI_BOOL

This data type is used to indicate a true or false condition.

```
typedef  uint32 BioAPI_BOOL;
#define BioAPI_FALSE            (0)
#define BioAPI_TRUE             (!BioAPI_FALSE)
```

2.1.13 BioAPI_BSP_SCHEMA

A BSP schema entry as posted to the BioAPI module registry.

```
typedef struct _bioapi_bsp_schema {
             BioAPI_UUID ModuleId;
             BioAPI_DEVICE_ID DeviceId;
             BioAPI_STRING BSPName;
             BioAPI_VERSION SpecVersion;
             BioAPI_VERSION ProductVersion;
             BioAPI_STRING Vendor;
             BioAPI_BIR_BIOMETRIC_DATA_FORMAT BspSupportedFormats;
             uint32 NumSupportedFormats;
             uint32 FactorsMask;
             uint32 Operations;
             uint32 Options;
             uint32 PayloadPolicy;
             uint32 MaxPayloadSize;
             sint32 DefaultVerifyTimeout;
             sint32 DefaultIdentifyTimeout;
             sint32 DefaultCaptureTimeout;
             sint32 DefaultEnrollTimeout;
             uint32 MaxBspDbSize;
             uint32 MaxIdentify;
             BioAPI_STRING Description;
             char Path;
      }BioAPI_BSP_SCHEMA, *BioAPI_BSP_SCHEMA_PTR;

      NOTE:  See 2.2.2.2 for BSP schema definition.
```

2.1.14 BioAPI_BSP_SCHEMA_ARRAY

An array of BSP schema entries as posted to the BioAPI module registry.

```
typedef BioAPI_BSP_SCHEMA_PTR BioAPI_BSP_SCHEMA_ARRAY,
*BioAPI_BSP_SCHEMA_ARRAY_PTR;
```

2.1.15 BioAPI_CANDIDATE

One of a set of candidates returned by ***Identify*** or ***IdentifyMatch*** indicating a successful match.

```
typedef struct bioapi_candidate {
      BioAPI_IDENTIFY_POPULATION_TYPE Type;
      union {
          BioAPI_UUID_PTR BIRInDataBase;
          uint32 *BIRInArray;
      } BIR;
      BioAPI_FAR   FARAchieved,
      BioAPI_FRR   FRRAchieved,
} BioAPI_CANDIDATE, *BioAPI_CANDIDATE_PTR;
```

2.1.16 BioAPI_CANDIDATE_ARRAY

An array of candidates returned by ***Identify*** or ***IdentifyMatch***.

```
typedef BioAPI_CANDIDATE_PTR BioAPI_CANDIDATE_ARRAY, *BioAPI_CANDIDATE_ARRAY_PTR;
```

2.1.17 BioAPI_DATA

The BioAPI_DATA structure is used to associate a length, in bytes, with an arbitrary block of contiguous memory.

```
typedef struct bioapi_data{
    uint32 Length;  /* in bytes */
    uint8 *Data;
} BioAPI_DATA, *BioAPI_DATA_PTR;
```

Definitions

Length - length of the data buffer in bytes

Data - points to the start of an arbitrary length data buffer

2.1.18 BioAPI_DB_ACCESS_TYPE

This bitmask describes a user's desired level of access to a database. The BSP may use the mask to determine what lock to obtain on a database.

```
typedef uint32 BioAPI_DB_ACCESS_TYPE, *BioAPI_DB_ACCESS_TYPE_PTR;

#define BioAPI_DB_ACCESS_READ           (0x1)
#define BioAPI_DB_ACCESS_WRITE          (0x2)
```

2.1.19 BioAPI_DB_CURSOR

A handle to a record in an open BIR database. The internal state for the handle includes the open database handle and also the position of a record in that open data base. All cursors to an open database are freed when the database is closed.

```
typedef  uint32 BioAPI_DB_CURSOR, *BioAPI_DB_CURSOR_PTR;
```

2.1.20 BioAPI_DB_HANDLE

A handle to an open BIR database.

```
typedef  sint32 BioAPI_DB_HANDLE, *BioAPI_DB_HANDLE_PTR;
#define      BioAPI_DB_INVALID_HANDLE         (-1)
```

2.1.21 BioAPI_DBBIR_ID

A structure providing the handle to a database managed by the BSP, and the ID of a BIR in that database.

```
typedef struct bioapi_dbbir_id {
      BioAPI_DB_HANDLE DbHandle;
      BioAPI_UUID KeyValue;
} BioAPI_DBBIR_ID, *BioAPI_DBBIR_ID_PTR;
```

2.1.22 BioAPI_DEVICE_ID

A device ID is a 32-bit integer assigned to an attached device by a BSP that supports it.

```
typedef  uint32  BioAPI_DEVICE_ID, *BioAPI_DEVICE_ID_PTR;
```

2.1.23 BioAPI_DEVICE_SCHEMA

A device schema entry as posted to the BioAPI module registry.

```
typedef struct _bioapi_device_schema {
             BioAPI_UUID ModuleId;
             BioAPI_DEVICE_ID DeviceId;
             BioAPI_BIR_BIOMETRIC_DATA_FORMAT DeviceSupportedFormats;
             uint32 NumSupportedFormats;
             uint32 SupportedEvents;
             BioAPI_STRING DeviceVendor;
             BioAPI_STRING DeviceDescription;
             BioAPI_STRING DeviceSerialNumber;
             BioAPI_VERSION DeviceHardwareVersion;
             BioAPI_VERSION DeviceFirmwareVersion;
             BioAPI_BOOL AuthenticatedDevice;
      } BioAPI_DEVICE_SCHEMA, *BioAPI_DEVICE_SCHEMA_PTR;

      NOTE:  See 2.2.2.3 for device schema definition.
```

2.1.24 BioAPI_FAR

A 32-bit integer value (N) that indicates a probable False Accept Rate of N/(2^{31}-1). The larger the value, the worse the result.

```
typedef  sint32  BioAPI_FAR, *BioAPI_FAR_PTR;
#define      BioAPI_NOT_SET                   (-1)
```

Note: FAR is used within BioAPI as a means of setting thresholds and returning scores (see section 1.7).

2.1.25 BioAPI_FRR

A 32-bit integer value (N) that indicates a probable False Reject Rate of N/(2^{31}-1). The larger the value, the worse the result.

```
typedef  sint32  BioAPI_FRR, *BioAPI_FRR_PTR;
#define      BioAPI_NOT_SET                   (-1)
#define      BioAPI_NOT_SUPPORTED             (-2)
```

Note: FRR is used within BioAPI as an optional/alternate means of setting thresholds and returning scores (see section 1.7).

2.1.26 BioAPI_FUNC_NAME_ADDR

This structure binds a function to the runtime address of the procedure that implements the named function. Function names are limited in length to the size of a BioAPI_MAX_STR_LEN.

```
#define BioAPI_MAX_STR_LEN 255
typedef struct bioapi_func_name_addr {
             char Name[BioAPI_MAX_STR_LEN];
             BioAPI_PROC_ADDR Address;
}BioAPI_FUNC_NAME_ADDR, *BioAPI_FUNC_NAME_ADDR_PTR;
```

Definition

Name - The name of the function represented as a fixed-length string.

Address - The runtime address of the procedure implementing the named function.

2.1.27 BioAPI_GUI_BITMAP

```
typedef struct bioapi_gui_bitmap {
      uint32 Width;
      uint32 Height;
      BioAPI_DATA_PTR Bitmap;
} BioAPI_GUI_BITMAP, *BioAPI_GUI_BITMAP_PTR;
```

2.1.28 BioAPI_GUI_MESSAGE

```
typedef uint32 BioAPI_GUI_MESSAGE, *BioAPI_GUI_MESSAGE_PTR;
```

2.1.29 BioAPI_GUI_PROGRESS

```
typedef uint8 BioAPI_GUI_PROGRESS, *BioAPI_GUI_PROGRESS_PTR;
```

NOTE: See description under 2.1.29.

2.1.30 BioAPI_GUI_RESPONSE

```
typedef uint8 BioAPI_GUI_RESPONSE;
#define      BioAPI_CAPTURE_SAMPLE      (1)
#define      BioAPI_CANCEL              (2)
#define      BioAPI_CONTINUE            (3)
#define      BioAPI_VALID_SAMPLE        (4)
#define      BioAPI_INVALID_SAMPLE      (5)
```

See description under 2.1.29.

2.1.31 BioAPI_GUI_STATE

A mask that indicates GUI state, and also what other parameter values are provided in the GUI State Callback.

```
typedef uint32 BioAPI_GUI_STATE;
```

```
#define BioAPI_SAMPLE_AVAILABLE   (0x0001)
#define BioAPI_MESSAGE_PROVIDED   (0x0002)
#define BioAPI_PROGRESS_PROVIDED  (0x0004)
```

2.1.32 BioAPI_GUI_STATE_CALLBACK

A Callback function that an application supplies to allow the service provider to indicate GUI state information to the application, and to receive responses back.

```
typedef BioAPI_RETURN (BioAPI *BioAPI_GUI_STATE_CALLBACK)
       (void *GuiStateCallbackCtx,
       BioAPI_GUI_STATE GuiState,
       BioAPI_GUI_RESPONSE Response,
       BioAPI_GUI_MESSAGE Message,
       BioAPI_GUI_PROGRESS Progress,
BioAPI_GUI_BITMAP_PTR SampleBuffer);
```

Parameters

GuiStateCallbackCtx (input) - A generic pointer to context information that was provided by the original requester and is being returned to its originator.

GuiState (input) – an indication of the current state of the service provider with respect to the GUI, plus an indication of what others parameters are available.

Response (output) – The response from the application back to the service provider on return from the Callback.

Message (input/optional) – The number of a message to display to the user. Message numbers are service-provider dependent. *GuiState* indicates if a *Message* is provided; if not the parameter is NULL.

Progress (input/optional) – A Value that indicates (as a percentage) the amount of progress in the development of a Sample/BIR. The value may be used to display a progress bar. Not all service providers support a progress indication. *GuiState* indicates if a sample *Progress value* is provided in the call; if not the parameter is NULL.

SampleBuffer (input/optional) – The current sample buffer for the application to display. *GuiState* indicates if a sample *Buffer* is provided; if not the parameter is NULL.

2.1.33 BioAPI_GUI_STREAMING_CALLBACK

A Callback function that an application supplies to allow the service provider to stream data in the form of a sequence of bitmaps.

```
typedef BioAPI_RETURN (BioAPI *BioAPI_GUI_STREAMING_CALLBACK)
       (void *GuiStreamingCallbackCtx,
       BioAPI_GUI_BITMAP_PTR Bitmap);
```

Parameters

GuiStreamingCallbackCtx (input) - A generic pointer to context information that was provided by the original requester and is being returned to its originator.

Bitmap (input) – a pointer to the bitmap to be displayed.

2.1.34 BioAPI_HANDLE

A unique identifier, returned on BioAPI_ModuleAttach, that identifies an attached BioAPI service provider.

```
typedef      uint32       BioAPI_HANDLE, *BioAPI_HANDLE_PTR;
```

2.1.35 BioAPI_H_LEVEL_FRAMEWORK_SCHEMA

The H-Level (framework) schema entry as posted to the BioAPI module registry.

```
typedef struct _bioapi_h_level_framework_schema {
             BioAPI_UUID ModuleId;
             BioAPI_STRING ModuleName;
             BioAPI_VERSION SpecVersion;
             BioAPI_VERSION ProdVersion;
             BioAPI_STRING Vendor;
             BioAPI_STRING Description;
      } BioAPI_H_LEVEL_FRAMEWORK_SCHEMA,*BioAPI_H_LEVEL_FRAMEWORK_SCHEMA_PTR;

      NOTE:  See 2.2.2.1 for H-Level schema definition.
```

2.1.36 BioAPI_IDENTIFY_POPULATION

A structure used to identify the set of BIRs to be used as input to an ***Identify*** or ***Identify_Match*** operation.

```
typedef struct bioapi_identify_population {
      BioAPI_IDENTIFY_POPULATION_TYPE Type;
      union {
          BioAPI_DB_HANDLE_PTR  BIRDataBase;
          BioAPI_BIR_ARRAY_POPULATION_PTR  BIRArray;
      } BIRs;
} BioAPI_IDENTIFY_POPULATION, *BioAPI_IDENTIFY_POPULATION_PTR;
```

2.1.37 BioAPI_IDENTIFY_POPULATION_TYPE

A value indicating the method of BIR input to an Identify or Identify_Match operation, whether it be via a passed-in array or a pointer to a database.

```
typedef uint8 BioAPI_IDENTIFY_POPULATION_TYPE;
#define      BioAPI_DB_TYPE                   (1)
#define      BioAPI_ARRAY_TYPE                (2)
```

2.1.38 BioAPI_INPUT_BIR

A structure used to input a BIR to the API. Such input can be in one of three forms:

1. A BIR Handle
2. A key to a BIR in a database managed by the BSP. If the DbHandle is zero, the default database is assumed. [A DbHandle is returned when a database is opened].
3. An actual BIR

```
typedef struct bioapi_input_bir {
      BioAPI_INPUT_BIR_FORM Form;
      union {
          BioAPI_DBBIR_ID_PTR  BIRinDb;
          BioAPI_BIR_HANDLE_PTR  BIRinBSP;
          BioAPI_BIR_PTR  BIR;
      } InputBIR;
} BioAPI_INPUT_BIR, *BioAPI_INPUT_BIR_PTR;
```

2.1.39 BioAPI_INPUT_BIR_FORM

```
typedef uint8 BioAPI_INPUT_BIR_FORM;
#define      BioAPI_DATABASE_ID_INPUT         (1)
#define      BioAPI_BIR_HANDLE_INPUT          (2)
#define      BioAPI_FULLBIR_INPUT             (3)
```

2.1.40 BioAPI_MEMORY_FUNCS

Set of structures used in memory management.

```
typedef void * (BioAPI *BioAPI_MALLOC)
      ( uint32 Size,
      void * Allocref);

typedef void (BioAPI *BioAPI_FREE)
      (void * Memblock,
      void * Allocref);

typedef void * (BioAPI *BioAPI_REALLOC)
      (void * Memblock,
      uint32 Size,
      void * Allocref);

typedef void * (BioAPI *BioAPI_CALLOC)
      (uint32 Num,
      uint32 Size,
      void * Allocref);

typedef struct bioapi_memory_funcs {
     BioAPI_MALLOC Malloc_func;
     BioAPI_FREE Free_func;
     BioAPI_REALLOC Realloc_func;
     BioAPI_CALLOC Calloc_func;
     void *AllocRef;
} BioAPI_MEMORY_FUNCS, *BioAPI_MEMORY_FUNCS_PTR;
```

2.1.41 BioAPI_ModuleEventHandler

This defines the event handler interface that an application must define and implement if it wishes to receive asynchronous notification of events such as the availability of a biometric sample, or a fault detected by the service module. The event handler is registered with the BioAPI as part of the BioAPI_ModuleLoad function. This is the caller's event handler for all general module events over all of the caller's attach-sessions with the loaded module. This general event notification is processed through the BioAPI.

The event handler should not issue BioAPI calls. These circular calls may result in deadlock in numerous situations, hence the event handler should be implemented without using BioAPI services.

The BioAPI_ModuleEventHandler can be invoked multiple times in response to a single event. The handler and the calling application must track receipt of event notifications and ignore duplicates.

```
typedef BioAPI_RETURN (BioAPI *BioAPI_ModuleEventHandler)
      (const BioAPI_UUID *BSPUuid,
      void* AppNotifyCallbackCtx,
      BioAPI_DEVICE_ID DeviceID,
      uint32  Reserved,
      BioAPI_MODULE_EVENT EventType);
```

Definition

BSPUuid - The UUID of the service module raising the event.

AppNotifyCallbackCtx - The application context specified during BioAPI_ModuleLoad( ).

DeviceID - The DeviceID of the service module raising the event.

Reserved - A reserved input; should be set to 0.

EventType - The BioAPI_MODULE_EVENT that has occurred.

2.1.42 BioAPI_MODULE_EVENT

This enumeration defines the event types that can be raised by any service module. Callers can define event handling callback functions of type *BioAPI_ModuleEventHandler* to receive and manage these events. Callback functions are registered using the BioAPI_ModuleLoad function. Example events include the addition (insertion) or removal of a biometric sensor. Events are asynchronous.

BioAPI_SOURCE_PRESENT and BioAPI_SOURCE_REMOVED events are generated by devices that can detect when the user may be available to provide a biometric sample (e.g., the user has to place a finger on a USB fingerprint device). BioAPI_SOURCE_PRESENT indicates that a sample may be available, while BioAPI_SOURCE_REMOVED indicates that a sample is probably no longer available. There is no requirement that these events must occur in pairs; several BioAPI_SOURCE_PRESENT events may occur in succession.

```
typedef uint32 BioAPI_MODULE_EVENT;
#define      BioAPI_NOTIFY_INSERT             (1)
#define      BioAPI_NOTIFY_REMOVE             (2)
#define      BioAPI_NOTIFY_FAULT              (3)
#define      BioAPI_NOTIFY_SOURCE_PRESENT     (4)
#define      BioAPI_NOTIFY_SOURCE_REMOVED     (5)
```

2.1.43 BioAPI_MODULE_EVENT_MASK

This enumeration defines a mask with bit positions for event type. The mask is used to enable/disable events, and to indicate what events are supported.

```
typedef uint32 BioAPI_MODULE_EVENT_MASK;
#define      BioAPI_NOTIFY_INSERT_BIT         (0x0001)
#define      BioAPI_NOTIFY_REMOVE_BIT         (0x0002)
#define      BioAPI_NOTIFY_FAULT_BIT          (0x0004)
#define      BioAPI_NOTIFY_SOURCE_PRESENT_BIT (0x0008)
#define      BioAPI_NOTIFY_SOURCE_REMOVED_BIT (0x0010)
```

2.1.44 BioAPI_POWER_MODE

An enumeration that specifies the types of power modes the system will try to use..

```
typedef uint32 BioAPI_POWER_MODE;

/* All functions available       */
#define BioAPI_POWER_NORMAL                   (1)
```

```
/* Able to detect (for example) insertion/fingeron/person present type of events
             */
#define BioAPI_POWER_DETECT                    (2)

/* minimum mode. all functions off      */
#define BioAPI_POWER_SLEEP                     (3)
```

2.1.45 BioAPI_PROC_ADDR

Generic pointer to a BioAPI function.

```
#if defined(WIN32)
typedef FARPROC BioAPI_PROC_ADDR;
#else
typedef void (BioAPI *BioAPI_PROC_ADDR) ();
#endif
typedef BioAPI_PROC_ADDR *BioAPI_PROC_ADDR_PTR;
```

2.1.46 BioAPI_QUALITY

A value indicating the quality of the biometric data in a BIR.

```
typedef  sint8  BioAPI_QUALITY;
```

NOTE: All integer values in the BIR header are little-endian.

The performance of biometrics varies with the quality of the biometric data. Since a universally accepted definition of quality does not exist. BioAPI has elected to provide the following structure with the goal of framing the effect of quality on usage of the BSP (as envisioned by the BSP vendor). The scores as reported by the BSP are based on the purpose (BIR_PURPOSE) indicted by the application (e.g. Capture for enrollment/verify , capture for enrollment/identify; capture for verify, etc.). Additionally, the demands upon the biometric vary based on the actual customer application and/or environment (i.e. a particular application usage may require higher quality samples than would normally be required by less demanding applications).

Quality measurements are reported as an integral value in the range 0-100 except as follows:

Value of –1: BioAPI_QUALITY was not set by the BSP (reference BSP vendor's documentation for explanation).

Value of –2: BioAPI_QUALITY is not supported by the BSP.

There are two objectives in providing BioAPI_QUALITY feedback to the application:

1. The primary objective is to have the BSP inform the application how suitable the biometric sample is for the purpose (BioAPI_PURPOSE) specified by the application (as framed by the BSP vendor based on the use scenario intended by the BSP vendor).

2. The secondary objective is to provide the application with relative results (e.g. current sample is better/worse than previous sample).

Quality scores in the range 0-100 have the following interpretation:

0-25:	UNACCEPTABLE: The biometric data cannot be used for the purpose specified by the application (BioAPI_PURPOSE). The biometric data must be replaced with a new sample.

26-50:	MARGINAL: The biometric data will provide poor performance for the purpose specified by the application (BioAPI_PURPOSE) and in most application environments will compromise the intent of the application. The biometric data should be replaced with a new sample.
51-75:	ADEQUATE: The biometric data will provide good performance in most application environments based on the purpose specified by the application (BioAPI_PURPOSE). The application should attempt to obtain higher quality data if the application developer anticipates demanding usage.
76-100:	EXCELLENT:: The biometric data will provide good performance for the purpose specified by the application (BioAPI_BIR_PURPOSE). The application may want to attempt to obtain better samples if the sample quality (BioAPI_QUALITY) is in the lower portion of the range (e.g. 76, 77,…) when convenient (e.g. during enrollment).

2.1.47 BioAPI_RETURN

This data type is returned by all BioAPI functions. The permitted values include:

- BioAPI_OK
- All Error Values defined in this specification
- BSP-specific error values defined and returned by a specific H-level service provider
- All Error Values defined in the D-Level Specification
- Device-specific error values defined and returned by a specific D-level service provider

```
typedef uint32 BioAPI_RETURN;

#define BioAPI_OK   (0)
```

Definition

BioAPI_OK - Indicates operation was successful

All other values - Indicates the operation was unsuccessful and identifies the specific, detected error that resulted in the failure.

2.1.48 BioAPI_SERVICE_UID

This structure uniquely identifies a biometric service provider and a device currently attached to it.

```
typedef struct bioapi_service_uid {
       BioAPI_UUID Uuid;
       BioAPI_VERSION Version;
       BioAPI_DEVICE_ID DeviceId;
       uint32  Reserved;
} BioAPI_SERVICE_UID, *BioAPI_SERVICE_UID_PTR;
```

Definitions

Uuid - A unique identifier for a BioAPI service module.

Version - The version of the service module.

DeviceID - An identifier for a device attached to the biometric service module.

Reserved - A reserved field, which is set to 0.

2.1.49 BioAPI_STREAM_CALLBACK

```
typedef BioAPI_RETURN (BioAPI *BioAPI_STREAM_CALLBACK)
        (void *StreamCallbackCtx,
        BioAPI_DATA_PTR OutMessage,
        BioAPI_DATA_PTR InMessage);
```

A callback function that an application supplies to allow the service provider to stream data in the form of a sequence of protocol data units (messages).

Parameters

> *StreamCallbackCtx (input)* - A generic pointer to context information that was provided by the original requester and is being returned to its originator.
>
> *OutMessage (input)* – a pointer to a protocol data unit to be sent to the communication partner.
>
> *InMessage (output/optional)* – a pointer to a protocol data unit to be received back from the communicating partner.

2.1.50 BioAPI_STRING

This is used by BioAPI data structures to represent a character string inside of a fixed-length buffer. The character string is expected to be NULL-terminated.

```
typedef char BioAPI_STRING [68];
```

2.1.51 BioAPI_UUID

```
typedef      uint8        BioAPI_UUID[16];
typedef      BioAPI_UUID  *BioAPI_UUID_PTR;
```

2.1.52 BioAPI_VERSION

```
typedef struct bioapi_version {
    uint32 Major;
    uint32 Minor;
} BioAPI_VERSION, *BioAPI_VERSION_PTR;
```

This structure is used to represent the version of BioAPI components. The major number begins at 1 and is incremented by 1 for each major release.

The minor number uses two digits to represent minor releases and revisions. The revision number is represented in the least significant digit. The remaining more significant digits represent minor numbers. The first release has the value of zero. There can be 9 subsequent releases then the minor number must be incremented. For example, the minor number for the very first release of a product would be represented as “00”. Subsequent releases would be “01”, “02”, “03” etc… to “09”. If version number changes at each release then the minor numbers would increment from “00”, “10”, “20” etc… to “90”. A minor version of 10 release 1 would be “100”.

Examples:
1.0 – Major: 1 Minor: 0
1.10 – Major: 1 Minor: 10
1.11 – Major: 1 Minor: 11
1.24 – Major: 1 Minor: 24
1.249 – Major: 1 Minor: 249

1.24.38 – not possible

Definitions
Major - the major version number of the component
Minor - the minor version number of the component

2.2 BioAPI Registry Schema

The various components of the BioAPI implementation are represented by records in the Module Registry. This registry is intended to be Operating System independent, and therefore information maintained for each component should be the same. The following section defines the information that should be kept for each component type.

2.2.1 Data Definitions

2.2.1.1 BioAPI_OPERATIONS_MASK

A mask that indicates what operations are supported by the BioAPI service provider.

```
typedef uint32 BioAPI_OPERATIONS_MASK;

#define BioAPI_CAPTURE                (0x0001)
#define BioAPI_CREATETEMPLATE         (0x0002)
#define BioAPI_PROCESS                (0x0004)
#define BioAPI_VERIFYMATCH            (0x0008)
#define BioAPI_IDENTIFYMATCH          (0x0010)
#define BioAPI_ENROLL                 (0x0020)
#define BioAPI_VERIFY                 (0x0040)
#define BioAPI_IDENTIFY               (0x0080)
#define BioAPI_IMPORT                 (0x0100)
#define BioAPI_SETPOWERMODE           (0x0200)
#define BioAPI_DATABASEOPERATIONS     (0x0400)
```

2.2.1.2 BioAPI_OPTIONS_MASK

A mask that indicates what options are supported by the BioAPI Service Provider. Note that optional functions are identified within the BioAPI_OPERATIONS_MASK and not repeated here.

```
typedef uint32 BioAPI_OPTIONS_MASK;
```

#define	BioAPI_RAW	(0x00000001)	If set, indicates that the BSP supports the return of raw/audit data.
#define	BioAPI_QUALITY_RAW	(0x00000002)	If set, BSP supports the return of a quality value (in the BIR header) for raw biometric data.
#define	BioAPI_QUALITY_INTERMEDIATE	(0x00000004)	If set, BSP supports the return of a quality value (in the BIR header) for intermediate biometric data.
#define	BioAPI_QUALITY_PROCESSED	(0x00000008)	If set, BSP supports the return of quality value (in the BIR header) for processed biometric data.
#define	BioAPI_APP_GUI	(0x00000010)	If set, indicates that the BSP supports application control of the GUI.
#define	BioAPI_STREAMINGDATA	(0x00000020)	If set, indicates that the BSP provides GUI streaming data.
#define	BioAPI_USERVALIDATESSAMPLES	(0x00000040)	If set, User must validate each sample.
#define	BioAPI_VERIFYSAMPLES	(0x00000080)	If set, BSP verifies each sample.
#define	BioAPI_SOURCEPRESENT	(0x00000100)	If set, BSP supports detection of source presence.
#define	BioAPI_PAYLOAD	(0x00001000)	If set, indicates that the BSP supports payload carry (accepts payload during

			enroll/process and returns payroll upon successful verify).
#define	BioAPI_BIR_SIGN	(0x00002000)	If set, BSP returns signed BIRs.
#define	BioAPI_BIR_ENCRYPT	(0x00004000)	If set, BSP returns encrypted BIRs.
#define	BioAPI_FRR	(0x00010000)	If set, indicates BSP supports the return of actual FRR during matching operations (Verify, VerifyMatch, Identify, IdentifyMatch).
#define	BioAPI_ADAPTATION	(0x00020000)	If set, BSP supports BIR adaptation (return of Verify or VerifyMatch operation).
#define	BioAPI_BINNING	(0x00040000)	If set, BSP supports binning (parameter used in Identify and IdentifyMatch operations).
#define	BioAPI_DEFAULTDATABASE	(0x00080000)	If set, BSP supports a default database.
#define	BioAPI_LOCAL_BSP	(0x01000000)	If set, BSP can operate in standalone mode.
#define	BioAPI_CLIENT_BSP	(0x02000000)	If set, BSP can operate as a Client (i.e., can Capture).
#define	BioAPI_SERVER_BSP	(0x04000000)	If set, BSP can operate as a Server.
#define	BioAPI_STREAMINGCALLBACK	(0x08000000)	If set, BSP supports streaming callbacks and StreamInputOutput..
#define	BioAPI_PROGRESS	(0x10000000)	If set, BSP supports the return of progress.
#define	BioAPI_SELFCONTAINEDDEVICE	(0x200000000)	If set, BSP is supporting a self-contained device

2.2.2 Component Schema

2.2.2.1 Framework Schema

This schema defines attributes of an instance of the BioAPI H-level Framework.

Field Name	**Field Data Type**	**Comment**
ModuleID	STRING	UUID (in string format) uniquely identifying a BioAPI framework module.
ModuleName	STRING	Filename of Module
Spec Version	STRING	BioAPI Specification Version string (in dotted high/low format – e.g. 2.0).
Product Version	STRING	H-Framework product version string (in dotted high/low format – e.g. 2.0).
Vendor	STRING	BioAPI Vendor name in text.
Description	STRING	BioAPI description in text.

2.2.2.2 BSP Schema

The service provider schema describes capabilities of a biometric service provider (BSP) module.

Field Name	**Field Data Type**	**Comment**
BSPID	STRING	UUID (in string format) uniquely identifying BSP
DeviceID	UINT32	4 byte device ID of attached sensor device
Description	STRING	Text descriptive name of the BSP
Path	STRING	Path where BSP executable is located
BSPName	STRING	Filename of BSP Module
Spec Version	STRING	BioAPI Specification Version string (in dotted high/low format – e.g. 2.0).
Product Version	STRING	BSP product version string (in dotted high/low format – e.g. 2.0).
Vendor	STRING	Service provider vendor name in Unicode text.
Supported Formats	MULTIUINT32	An array of 2-byte integer pairs. Each pair specifying a supported biometric data format. See BioAPI_BIR_BIOMETRIC_DATA_FORMAT
Factors Mask	UINT32	A mask which indicates what forms of authentication are supported (BioAPI_BIR_AUTH_FACTORS)
Operations	UINT32	Operations supported by service provider (BioAPI_ OPERATIONS_MASK)
Options	UINT32	Options supported by the BSP BioAPI_OPTIONS_MASK
Payload Policy	UINT32	Threshold setting (minimum FAR value) used to determine when to release a payload. BioAPI_FAR
Max Payload Size	UINT32	Maximum size in bytes of a payload.
Default Verify Timeout	SINT32	Default timeout value (in milliseconds) used by a BSP for verify operations when no timeout is set by the application

Default Identify Timeout	SINT32	Default timeout value (in milliseconds) used by a BSP for identify operations when no timeout is set by the application.
Default Capture Timeout	SINT32	Default timeout value (in milliseconds) used by a BSP for capture operations when no timeout is set by the application.
Default Enroll Timeout	SINT32	Default timeout value (in milliseconds) used by a BSP for enroll operations when no timeout is set by the application.
MAX BSP DB size	UINT32	Maximum size of a BSP owned (internal) database. If NULL, no BSP database exists.
MaxIdentify	UINT32	Largest population supported by Identify function Unlimited = FFFFFFFF.

2.2.2.3 Biometric Device Schema

The information in the biometric device registry entry is updated each time a biometric device is attached to or removed from the service provider.

Field Name	**Field Data Type**	**Comment**
ModuleID	STRING	UUID (in string format) uniquely identifying service provider module
DeviceID	SINT32	4 byte device ID
Supported Formats	MULTIUINT32	BIR Formats supported by BSP + device See BioAPI_BIR_BIOMETRIC_DATA_FORMAT
Supported Events	UINT32	A BioAPI_MODULE_EVENT_MASK indicating which types of events are supported
Device Vendor	STRING	Unicode text name of device Vendor.
Device Description	STRING	Unicode text description of the biometric device.
Device Serial Number	STRING	Serial Number of biometric device
Device Hardware Version	STRING	Device hardware version string (in dotted high/low format – e.g. 2.0).
Device Firmware Version	STRING	Device Firmware version string (in dotted high/low format – e.g. 2.0).
Authenticated Device	BOOL	An indication of whether the device has been authenticated.

2.3 BioAPI Error-Handling

This section presents a specification for error handling in the BioAPI that provides a consistent mechanism for returning errors to the caller that can also be fitted into a more extensive operating system scheme for error handling.

All BioAPI functions will return a value of type BioAPI_RETURN. The value BioAPI_OK indicates that the function was successful. Any other value is an error value. Allowance has been made for returning error values architected by the BioAPI and also error values that are specific to a service provider.

2.3.1 Error Values and Error Codes Scheme

In many cases, error codes need to fit into a general schema for an operating system. This has been taken into consideration as much as possible (bearing in mind that operating systems do not have consistent error code schemes).

Error Value refers to the 32-bit BioAPI_RETURN value.

Error Code refers to the low-order portion of the Error Value that identifies the actual error situation.

The high-order portion of the Error Value indicates, in some scheme, the component (within the OS) that is raising the error. In the BioAPI, there are only four "components" that can raise errors;

a) the BioAPI Framework,
b) an H-level service provider
c) the D-level Framework
d) a D-level service provider (including a device)

There are values defined for the high-order portion of the Error Value that will differentiate between the BioAPI components. To allow for some flexibility in the operating system schema, these values are configurable.

Some Error Codes may be raised by more than one component. These are called common errors, and will have the same Error Codes no matter which component raises them. Therefore, space is reserved in the Error Code values for this common set.

In the remaining set of Error Codes, space must be reserved for the "architected" set of Error Codes and the "implementation- specific" Error Codes. The extent of these sets is also configurable.

The rest of the chapter defines:

- A list of Configurable BioAPI Error Code Constants
- A listing of BioAPI Error Code Constants
- A listing of General Error Values that may be returned by any BioAPI call
- Listings of Common Error Codes.   Common error codes are codes that can be associated with multiple components.
- A listing of Configurable BioAPI Error Code Constants for specific component types

2.3.2 Error Codes and Error Value Enumeration

2.3.2.1 Configurable BioAPI Error Code Constants

The following constants can be configured on a per system basis.

#define BioAPI_BASE_ERROR (0x00000000)

The configurable BioAPI error code base value.

#define BioAPI_ERRORCODE_COMPONENT_EXTENT (0x00001000)

#define BioAPI_ERRORCODE_COMMON_EXTENT (0x100)
The number of error codes allocated to indicate "common" errors.

The configurable number of error codes allocated for each component type. This number must be greater than BioAPI_ERRORCODE_COMMON_EXTENT, and should allow at least half the space for specification-defined error codes.

#define BioAPI_ERRORCODE_CUSTOM_OFFSET (0x00000800)
The configurable offset at which custom error codes are allocated. Must be greater than BioAPI_ERRCODE_COMMON_EXTENT and less than BioAPI_ERRORCODE_COMPONENT_EXTENT.
A BSP with "custom" error codes simply starts assigning them from this offset (without regard to any other BSPs.)

2.3.2.2 BioAPI Error Code Constants

```
#define BioAPI_H_FRAMEWORK_BASE_ERROR (BioAPI_BASE_ERROR)
#define BioAPI_H_FRAMEWORK_PRIVATE_ERROR                \
        (BioAPI_H_FRAMEWORK_BASE_ERROR + BioAPI_ERRORCODE_CUSTOM_OFFSET)

#define BioAPI_BSP_BASE_ERROR                           \
        (BioAPI_H_FRAMEWORK_BASE_ERROR + BioAPI_ERRORCODE_COMPONENT_EXTENT)
#define BioAPI_BSP_PRIVATE_ERROR                        \
        (BioAPI_BSP_BASE_ERROR + BioAPI_ERRORCODE_CUSTOM_OFFSET)

#define BioAPI_D_FRAMEWORK_BASE_ERROR                   \
        (BioAPI_BSP_BASE_ERROR + BioAPI_ERRORCODE_COMPONENT_EXTENT)
#define BioAPI_D_FRAMEWORK_PRIVATE_ERROR                \
        (BioAPI_D_FRAMEWORK_BASE_ERROR + BioAPI_ERRORCODE_CUSTOM_OFFSET)

#define BioAPI_DEVICE_BASE_ERROR                        \
        (BioAPI_D_FRAMEWORK_BASE_ERROR + BioAPI_ERRORCODE_COMPONENT_EXTENT)
#define BioAPI_DEVICE_PRIVATE_ERROR                     \
        (BioAPI_DEVICE_BASE_ERROR + BioAPI_ERRORCODE_CUSTOM_OFFSET)
```

2.3.2.3 General Error Values

The following error values can be returned by the H-Framework for any BioAPI function.

```
#define BioAPIERR_H_FRAMEWORK_INVALID_MODULE_HANDLE                \
(BioAPI_H_FRAMEWORK_BASE_ERROR + BioAPI_ERRORCODE_COMMON_EXTENT + 1)
```
The given service provider handle is not valid

```
#define BioAPIERR_H_FRAMEWORK_NOT_INITIALIZED                  \
(BioAPI_H_FRAMEWORK_BASE_ERROR + BioAPI_ERRORCODE_COMMON_EXTENT + 2)
```
A function is called without initializing the BioAPI

2.3.2.4 Common Error Codes For All Module Types

The following codes can be returned by multiple components.

#define BioAPI_ERRCODE_INTERNAL_ERROR (0x0001)
General system error; indicates that an operating system or internal state error has occurred and the system may not be in a known state.

#define BioAPI_ERRCODE_MEMORY_ERROR (0x0002)
A memory error occurred.

#define BioAPI_ERRCODE_INVALID_POINTER (0x0004)
An input/output function parameter or input/output field inside of a data structure is an invalid pointer.

#define BioAPI_ERRCODE_INVALID_INPUT_POINTER (0x0005)
An input function parameter or input field in a data structure is an invalid pointer

#define BioAPI_ERRCODE_INVALID_OUTPUT_POINTER (0x0006)
An output function parameter or output field in a data structure is an invalid pointer

#define BioAPI_ERRCODE_FUNCTION_NOT_IMPLEMENTED (0x0007)
The function is not implemented by the service provider

#define BioAPI_ERRCODE_OS_ACCESS_DENIED (0x0009)
The operating system denied access to a required resource

#define BioAPI_ERRCODE_FUNCTION_FAILED (0x000A)
The function failed for an unknown reason.

#define BioAPI_ERRCODE_INVALID_UUID (0x000C)
Invalid UUID

#define BioAPI_ERRCODE_INCOMPATIBLE_VERSION (0x0041)
Version is not compatible with the current version

Error values with the following code enumeration values may be returned from any function that takes as input a BioAPI_DATA.

#define BioAPI_ERRCODE_INVALID_DATA (0x0046)
The data in an input parameter is invalid

Error values with the following code enumeration values may be returned from any function that takes as input a DB handle.

#define BioAPI_ERRCODE_INVALID_DB_HANDLE (0x004A)
Invalid database handle

2.3.3 H-Framework Errors

2.3.3.1 H-Framework Error Values derived from the Common Error Codes

```
#define BioAPIERR_H_FRAMEWORK_INTERNAL_ERROR                \
        (BioAPI_H_FRAMEWORK_BASE_ERROR+BioAPI_ERRCODE_INTERNAL_ERROR)

#define BioAPIERR_H_FRAMEWORK_MEMORY_ERROR                  \
        (BioAPI_H_FRAMEWORK_BASE_ERROR+BioAPI_ERRCODE_MEMORY_ERROR)

#define BioAPIERR_H_FRAMEWORK_INVALID_POINTER               \
        (BioAPI_H_FRAMEWORK_BASE_ERROR+BioAPI_ERRCODE_INVALID_POINTER)

#define BioAPIERR_H_FRAMEWORK_INVALID_INPUT_POINTER         \
            (BioAPI_H_FRAMEWORK_BASE_ERROR+BioAPI_ERRCODE_INVALID_INPUT_POINTER)

#define BioAPIERR_H_FRAMEWORK_INVALID_OUTPUT_POINTER        \
            (BioAPI_H_FRAMEWORK_BASE_ERROR+BioAPI_ERRCODE_INVALID_OUTPUT_POINTER)

#define BioAPIERR_H_FRAMEWORK_FUNCTION_NOT_IMPLEMENTED              \
            (BioAPI_H_FRAMEWORK_BASE_ERROR+BioAPI_ERRCODE_FUNCTION_NOT_IMPLEMENTED)

#define BioAPIERR_H_FRAMEWORK_OS_ACCESS_DENIED                  \
            (BioAPI_H_FRAMEWORK_BASE_ERROR+BioAPI_ERRCODE_OS_ACCESS_DENIED)

#define BioAPIERR_H_FRAMEWORK_FUNCTION_FAILED                   \
            (BioAPI_H_FRAMEWORK_BASE_ERROR+BioAPI_ERRCODE_FUNCTION_FAILED)

#define BioAPIERR_H_FRAMEWORK_INVALID_UUID                      \
            (BioAPI_H_FRAMEWORK_BASE_ERROR+BioAPI_ERRCODE_INVALID_UUID)

#define BioAPIERR_H_FRAMEWORK_INCOMPATIBLE_VERSION              \
            (BioAPI_H_FRAMEWORK_BASE_ERROR+BioAPI_ERRCODE_INCOMPATIBLE_VERSION)
```

2.3.3.2 H-Framework-specific Error Values

```
/* Reserve first 16 H_FRAMEWORK Error Codes for general errors */

#define BioAPI_H_FRAMEWORK_BASE_H_FRAMEWORK_ERROR                   \
            (BioAPI_H_FRAMEWORK_BASE_ERROR+BioAPI_ERRORCODE_COMMON_EXTENT+0x10)


#define BioAPIERR_H_FRAMEWORK_MODULE_LOAD_FAILED                    \
            (BioAPI_H_FRAMEWORK_BASE_H_FRAMEWORK_ERROR+6)
```
BSP Module Load function failed

```
#define BioAPIERR_H_FRAMEWORK_MODULE_UNLOAD_FAILED                  \
            (BioAPI_H_FRAMEWORK_BASE_H_FRAMEWORK_ERROR+8)
```
BSP Module Unload function failed

```
#define BioAPIERR_H_FRAMEWORK_LIB_REF_NOT_FOUND                     \
            (BioAPI_H_FRAMEWORK_BASE_H_FRAMEWORK_ERROR+9)
```
A reference to the loaded library cannot be obtained

```
#define BioAPIERR_H_FRAMEWORK_INVALID_MODULE_FUNCTION_TABLE   \
            (BioAPI_H_FRAMEWORK_BASE_H_FRAMEWORK_ERROR+10)
```
BSP Module function table registered with CSSM is invalid

```
#define BioAPIERR_H_FRAMEWORK_MODULE_NOT_LOADED                     \
            (BioAPI_H_FRAMEWORK_BASE_H_FRAMEWORK_ERROR+14)
```
Module was not loaded

```
#define BioAPIERR_H_FRAMEWORK_INVALID_DEVICE_ID                 \
            (BioAPI_H_FRAMEWORK_BASE_H_FRAMEWORK_ERROR+15)
```
Invalid DeviceId was requested

2.3.4 BSP Errors

2.3.4.1 BSP Error Values derived from the Common Error Codes

```
#define BioAPIERR_BSP_INTERNAL_ERROR                            \
            (BioAPI_BSP_BASE_ERROR+BioAPI_ERRCODE_INTERNAL_ERROR)

#define BioAPIERR_BSP_MEMORY_ERROR                                    \
            (BioAPI_BSP_BASE_ERROR+BioAPI_ERRCODE_MEMORY_ERROR)

#define BioAPIERR_BSP_INVALID_POINTER                                 \
            (BioAPI_BSP_BASE_ERROR+BioAPI_ERRCODE_INVALID_POINTER)

#define BioAPIERR_BSP_INVALID_INPUT_POINTER                           \
            (BioAPI_BSP_BASE_ERROR+BioAPI_ERRCODE_INVALID_INPUT_POINTER)

#define BioAPIERR_BSP_INVALID_OUTPUT_POINTER                          \
            (BioAPI_BSP_BASE_ERROR+BioAPI_ERRCODE_INVALID_OUTPUT_POINTER)

#define BioAPIERR_BSP_FUNCTION_NOT_IMPLEMENTED                        \
            (BioAPI_BSP_BASE_ERROR+BioAPI_ERRCODE_FUNCTION_NOT_IMPLEMENTED)

#define BioAPIERR_BSP_OS_ACCESS_DENIED                                \
            (BioAPI_BSP_BASE_ERROR+BioAPI_ERRCODE_OS_ACCESS_DENIED)

#define BioAPIERR_BSP_FUNCTION_FAILED                                 \
            (BioAPI_BSP_BASE_ERROR+BioAPI_ERRCODE_FUNCTION_FAILED)

#define BioAPIERR_BSP_INVALID_DATA                                    \
            (BioAPI_BSP_BASE_ERROR+BioAPI_ERRCODE_INVALID_DATA)

#define BioAPIERR_BSP_INVALID_DB_HANDLE                               \
            (BioAPI_BSP_BASE_ERROR+BioAPI_ERRCODE_INVALID_DB_HANDLE)
```

2.3.4.2 BSP-specific Error Values

```
#define BioAPI_BSP_BASE_BSP_ERROR                                     \
            (BioAPI_BSP_BASE_ERROR+BioAPI_ERRORCODE_COMMON_EXTENT)
```

```
#define BioAPIERR_BSP_UNABLE_TO_CAPTURE          (BioAPI_BSP_BASE_BSP_ERROR+1)
```
BSP is unable to capture raw samples from the device

```
#define BioAPIERR_BSP_TOO_MANY_HANDLES           (BioAPI_BSP_BASE_BSP_ERROR+2)
```
The BSP has no more space to allocate BIR handles

```
#define BioAPIERR_BSP_TIMEOUT_EXPIRED            (BioAPI_BSP_BASE_BSP_ERROR+3)
```
The Function has been terminated because the timeout value has expired

```
#define BioAPIERR_BSP_INVALID_BIR                (BioAPI_BSP_BASE_BSP_ERROR+4)
```
The input BIR is invalid for the purpose required

```
#define BioAPIERR_BSP_BIR_SIGNATURE_FAILURE      (BioAPI_BSP_BASE_BSP_ERROR+5)
```
The BSP could not validate the signature on the BIR

```
#define BioAPIERR_BSP_UNABLE_TO_WRAP_PAYLOAD     (BioAPI_BSP_BASE_BSP_ERROR+6)
```
The BSP is unable to include the payload in the new BIR

```
#define BioAPIERR_BSP_NO_INPUT_BIRS              (BioAPI_BSP_BASE_BSP_ERROR+8)
```
The identify population is NULL

```
#define BioAPIERR_BSP_UNSUPPORTED_FORMAT         (BioAPI_BSP_BASE_BSP_ERROR+9)
```
The BSP does not support the data form for the Import function

```
#define BioAPIERR_BSP_UNABLE_TO_IMPORT           (BioAPI_BSP_BASE_BSP_ERROR+10)
```

The BSP was unable to construct a BIR from the input data

#define BioAPIERR_BSP_FUNCTION_NOT_SUPPORTED (BioAPI_BSP_BASE_BSP_ERROR+12)
The BSP does not support this operation.

#define BioAPIERR_BSP_INCONSISTENT_PURPOSE…… (BioAPI_BSP_BASE_BSP_ERROR+13)
The purpose recorded in the BIR, and the requested purpose, are inconsistent with the function being performed.

#define BioAPIERR_BSP_BIR_NOT_FULLY_PROCESSED…… (BioAPI_BSP_BASE_BSP_ERROR+14)
The function requires a fully-processed BIR.

#define BioAPIERR_BSP_PURPOSE_NOT_SUPPORTED…… (BioAPI_BSP_BASE_BSP_ERROR+15)
The BSP does not support the requested purpose.

#define BioAPIERR_BSP_UNABLE_TO_OPEN_DATABASE (BioAPI_BSP_BASE_BSP_ERROR+256)
BSP is unable to open specified database

#define BioAPIERR_BSP_DATABASE_IS_LOCKED (BioAPI_BSP_BASE_BSP_ERROR+257)
Database cannot be opened for the access requested because it is locked

#define BioAPIERR_BSP_DATABASE_DOES_NOT_EXIST (BioAPI_BSP_BASE_BSP_ERROR+258)
The specified database name does not exist

#define BioAPIERR_BSP_DATABASE_ALREADY_EXISTS (BioAPI_BSP_BASE_BSP_ERROR+259)
Create failed because the database already exists

#define BioAPIERR_BSP_INVALID_DATABASE_NAME (BioAPI_BSP_BASE_BSP_ERROR+260)
Invalid database name

#define BioAPIERR_BSP_RECORD_NOT_FOUND (BioAPI_BSP_BASE_BSP_ERROR+261)
No record exists with the requested key

#define BioAPIERR_BSP_CURSOR_IS_INVALID (BioAPI_BSP_BASE_BSP_ERROR+262)
The specified cursor is invalid

#define BioAPIERR_BSP_DATABASE_IS_OPEN (BioAPI_BSP_BASE_BSP_ERROR+263)
Database is already open

#define BioAPIERR_BSP_INVALID_ACCESS_REQUEST (BioAPI_BSP_BASE_BSP_ERROR+264)
Unrecognized access type

#define BioAPIERR_BSP_END_OF_DATABASE (BioAPI_BSP_BASE_BSP_ERROR+265)
End of database has been reached.

#define BioAPIERR_BSP_UNABLE_TO_CREATE_DATABASE (BioAPI_BSP_BASE_BSP_ERROR+266)
BSP cannot create the database

#define BioAPIERR_BSP_UNABLE_TO_CLOSE_DATABASE (BioAPI_BSP_BASE_BSP_ERROR+267)
BSP cannot close database

#define BioAPIERR_BSP_UNABLE_TO_DELETE_DATABASE (BioAPI_BSP_BASE_BSP_ERROR+268)
BSP cannot delete database

2.4 Framework Operations

2.4.1 BioAPI_Init

BioAPI_RETURN BioAPI BioAPI_Init

```
(const BioAPI_VERSION *Version,
uint32  Reserved1,
const void * Reserved2,
uint32  *Reserved3,
const void * Reserved4);
```

This function initializes the BioAPI and verifies that the version of the BioAPI expected by the application is compatible with the version of the BioAPI on the system. This function should be called at least once by the application.

Parameters

Version (input) - the major and minor version number of the BioAPI release the application is compatible with.

Reserved1 (input) - a reserved input; should be set to 0

Reserved2 (input) - a reserved input; should be set to NULL.

Reserved3 (input) - a reserved input; should be set to 0.

Reserved4 (input) - a reserved input; should be set to NULL.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_H_FRAMEWORK_INCOMPATIBLE_VERSION
See also the **BioAPI Error Handling**

2.4.2 BioAPI_Terminate

BioAPI_RETURN BioAPI BioAPI_Terminate ( ) ;

This function terminates the caller's use of BioAPI. BioAPI can cleanup all internal state associated with the calling application. This function must be called once for each call to BioAPI_Init().

Parameters

None

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

See also the **BioAPI Error Handling**

2.4.3 BioAPI_EnumModules

BioAPI_RETURN BioAPI BioAPI_EnumModules
(BioAPI_BSP_SCHEMA *BspSchemaArray,
uint32 ArraySize,
uint32 *ElementsNeeded,
uint32 *NumElementsReturned);

Enumerates the list of installed BSPs, and fills in BspSchemaArray with the results. If BspSchemaArray is NULL, simply counts the installed BSPs and returns the number in ElementsNeeded.

Parameters

BspSchemaArray (input) – If null, indicates function is to return only the module count.
(output) – An array containing the set of BSP module registry entries.

ArraySize (output) – The size of the returned BSPSchemaArray.

ElementsNeeded (output) – The number of BSPs installed (as located in the BSP module registry).

NumElementsReturned (output) – The number of BSP module registry entries contained in the returned BspSchemaArray. If input BspSchemaArray = null, this value is set to zero.

Return Value
A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors
See also the **BioAPI Error Handling**

2.4.4 BioAPI_ModuleLoad

BioAPI_RETURN BioAPI BioAPI_ModuleLoad
(const BioAPI_UUID *BSPUuid,
uint32 Reserved,
BioAPI_ModuleEventHandler AppNotifyCallback,
void* AppNotifyCallbackCtx);

The function BioAPI_Init must be invoked prior to calling BioAPI_ModuleLoad. The function BioAPI_ModuleAttach can be invoked multiple times per call to BioAPI_ModuleLoad.

This function initializes the service module. Initialization includes registering the application's module-event handler and enabling all events. The application can choose to provide an event handler function to receive notification of insert, remove and user-present events. Many applications can independently and concurrently load the same BSP, and each application can establish its own event handler. They will all receive notification of an event.

The scope of a ModuleEvent Handler is a process. (Note also that the scope of event enablement is also a process.) A process may establish as many event handlers as it wishes (though, since this is done through ModuleLoad, it is unlikely that it will want to establish more than one). A module event handler is identified by a combination of (address, context). When the event occurs in the BSP, the BSP will callback to the BioAPI framework once on some thread in the process (but not necessarily an application thread). No matter what thread is used to callback to the framework, all even thandlers that are currently established in the process will get called in some sequence by the BioAPI framework on that thread. There is a "use count" on the establishment of event handlers, they have to be dis-established (by ModuleUnload) as many times as they were established. When a BSP is loaded (ModuleLoad), it should raise an "insert" event immediately if there is a sensor already attached. This will indicate to the application that it can go ahead and do a ModuleAttach. If the BSP supports USB devices (for example) and there is no device attached, it should not raise the "insert" event. The "insert" event will be raised when the USB sensor is plugged in.

Parameters

BSPUuid (input) - the UUID of the module selected for loading.

Reserved (input) - a reserved input; should be set to 0.

AppNotifyCallback (input/optional) - the event notification function provided by the caller. This defines the callback for event notifications from the loaded (and later attached) service module.

AppNotifyCallbackCtx (input/optional) - when the selected service module raises an event, this context is passed as an input to the event handler specified by *AppNotifyCallback*.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_H_FRAMEWORK_MODULE_LOAD_FAIL
BioAPIERR_H_FRAMEWORK_INVALID_UUID
See also the **BioAPI Error Handling**

2.4.5 BioAPI_ModuleUnload

BioAPI_RETURN BioAPI BioAPI_ModuleUnload

```
(const BioAPI_UUID *BSPUuid,
BioAPI_ModuleEventHandler AppNotifyCallback,
void* AppNotifyCallbackCtx);
```

The function de-registers event notification callbacks for the caller identified by BSPUuid. BioAPI_ModuleUnload is the analog call to BioAPI_ModuleLoad. If all callbacks registered with BioAPI are removed, then BioAPI unloads the service module that was loaded by calls to BioAPI_ModuleLoad. Calls to BioAPI_ModuleUnload that are not matched with a previous call to BioAPI_ModuleLoad result in an error.

The BioAPI uses the three input parameters; BSPUuid, AppNotifyCallback and AppNotifyCallbackCtx to uniquely identify registered callbacks.

This function should be invoked after all necessary calls to BioAPI_ModuleDetach have been performed.

Parameters

BSPUuid (input) - the UUID of the module selected for unloading.

AppNotifyCallback (input/optional) - the event notification function to be deregistered. The function must have been provided by the caller in BioAPI_ModuleLoad.

AppNotifyCallbackCtx (input/optional) - the event notification context that was provided in the corresponding call to BioAPI_ModuleLoad.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_H_FRAMEWORK_MODULE_UNLOAD_FAILED
BioAPIERR_H_FRAMEWORK_INVALID_UUID
See also the **BioAPI Error Handling**

2.4.6 BioAPI_ModuleAttach

BioAPI_RETURN BioAPI BioAPI_ModuleAttach

```
(const BioAPI_UUID *BSPUuid,
const BioAPI_VERSION  *Version,
const BioAPI_MEMORY_FUNCS *MemoryFuncs,
BioAPI_DEVICE_ID DeviceID,
uint32  Reserved1,
uint32  Reserved2,
uint32. Reserved3
BioAPI_FUNC_NAME_ADDR *FunctionTable,
uint32 NumFunctionTable,
const void * Reserved4,
BioAPI_HANDLE_PTR NewModuleHandle);
```

This function attaches the service provider module and verifies that the version of the module expected by the application is compatible with the version on the system. The caller must specify which attached device the BSP should use.

The caller may provide a function table containing function-names for the desired operations. On output each function name is matched with an API function pointer. The caller can use the pointers to invoke the operations through BioAPI.

Parameters

BSPUuid (input) - a pointer to the BioAPI_UUID structure containing the global unique identifier for the BSP module.

Version (input) - the major and minor version number of the required level of BSP services and features. The BSP must determine whether its services are compatible with the required version.

MemoryFuncs (input) - a structure containing pointers to the memory routines.

DeviceID (input) - the DeviceID that is returned in the BioAPI_ModuleEventHandler from BioAPI_ModuleLoad should be used. (BSPs that support only one device can use zero as the device ID.)

Reserved1 (input) - a reserved input; should be set to 0.

Reserved2 (input) - a reserved input; should be set to 0.

Reserved3 (input) - a reserved input; should be set to 0.

FunctionTable (input/output/optional) - a table of function-name and API function-pointer pairs. The caller provides the name of the functions as input. The corresponding API function pointers are returned on output. The function table allows dynamic linking of BioAPI interfaces

NumFunctionTable (input) - the number of entries in the FunctionTable parameter. If no *FunctionTable* is provided this value must be zero.

Reserved4 (input) - a reserved input; should be set to NULL.

NewModuleHandle (output) - a new module handle that can be used to interact with the requested service provider. The value will be set to BioAPI_DB_INVALID_HANDLE if the function fails.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_H_FRAMEWORK_INVALID_MODULE_FUNCTION_TABLE
BioAPIERR_H_FRAMEWORK_MODULE_NOT_LOADED
BioAPIERR_H_FRAMEWORK_INVALID DEVICE_ID
BioAPIERR_H_FRAMEWORK_INVALID_UUID
See also the **BioAPI Error Handling**

2.4.7 BioAPI_ModuleDetach

BioAPI_RETURN BioAPI BioAPI_ModuleDetach
(BioAPI_HANDLE ModuleHandle);

This function detaches the application from the service provider module.

Parameters

ModuleHandle (input) - the handle that describes the service provider module.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

See also the **BioAPI Error Handling**

2.4.8 BioAPI_QueryDevice

BioAPI_RETURN BioAPI BioAPI_QueryDevice
(BioAPI_HANDLE ModuleHandle,
BioAPI_SERVICE_UID_PTR ServiceUID);

This function completes a structure containing the persistent unique identifier of the attached module and identifying the device currently attached to it.

Parameters

ModuleHandle (input) – the handle of the module subservice for which the subservice unique identifier should be returned.

ServiceUID (output) - Service UID value associated with ModuleHandle.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

See also the **BioAPI Error Handling**

2.5 BSP Operations

2.5.1 Handle Operations

2.5.1.1 BioAPI_FreeBIRHandle

BioAPI_RETURN BioAPI BioAPI_FreeBIRHandle
(BioAPI_HANDLE ModuleHandle,
BioAPI_BIR_HANDLE Handle);

Frees the memory and resources associated with the specified BIR Handle. The associated BIR is no longer referenceable through that handle. If necessary, the application must make the BIR persistent either in a BSP-managed database or an application-managed database before freeing the handle.

Parameters

ModuleHandle (input) - the handle of the attached BioAPI service provider.

Handle (input) – the BIR Handle to be freed.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

See also the **BioAPI Error Handling**

2.5.1.2 BioAPI_GetBIRFromHandle

BioAPI_RETURN BioAPI BioAPI_GetBIRFromHandle

```
(BioAPI_HANDLE   ModuleHandle,
BioAPI_BIR_HANDLE   Handle,
BioAPI_BIR_PTR *BIR);
```

Retrieves the BIR associated with a BIR handle. The Handle is invalidated. The BSP allocates the storage for both the retrieved BIR structure and its data members using the application's memory allocation callback function.

Parameters

ModuleHandle (input) - the handle of the attached BioAPI service provider.

Handle (input) – the handle of the BIR to be retrieved.

BIR (output) – the retrieved BIR.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

See also the **BioAPI Error Handling**

2.5.1.3 BioAPI_GetHeaderFromHandle

BioAPI_RETURN BioAPI BioAPI_GetHeaderFromHandle

```
(BioAPI_HANDLE   ModuleHandle,
BioAPI_BIR_HANDLE   Handle,
BioAPI_BIR_HEADER_PTR  Header);
```

Retrieves the BIR header identified by *Handle*. The BIR Handle is not freed by the BSP.

Parameters

ModuleHandle (input) - the handle of the attached BioAPI service provider.

Handle (input) – the handle of the BIR whose header is to be retrieved.

Header (output) – the header of the specified BIR.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

See also the **BioAPI Error Handling**

2.5.2 Callback and Event Operations

2.5.2.1 BioAPI_EnableEvents

BioAPI_RETURN BioAPI BioAPI_EnableEvents
(BioAPI_HANDLE ModuleHandle,
BioAPI_MODULE_EVENT_MASK *Events);

This function enables the events (indicated by the *Events* mask) from the attached BSP in the current process. All other events from this BSP are disabled for this process.

Parameters:

ModuleHandle (input) - the handle of the attached BioAPI service provider.

Events (input) - a pointer to a mask indicating which events to enable.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

See also the **BioAPI Error Handling**

2.5.2.2 BioAPI_SetGUICallbacks

BioAPI_RETURN BioAPI BioAPI_SetGUICallbacks

```
(BioAPI_HANDLE  ModuleHandle,
BioAPI_GUI_STREAMING_CALLBACK  GuiStreamingCallback,
void  *GuiStreamingCallbackCtx,
BioAPI_GUI_STATE_CALLBACK  GuiStateCallback,
void  *GuiStateCallbackCtx);
```

This function allows the application to establish callbacks so that the application may control the "look-and-feel" of the biometric user interface. [Not all BSPs provide streaming data.]

Parameters:

ModuleHandle (input) - the handle of the attached BioAPI service provider.

GuiStreamingCallback (input) - a pointer to an application callback to deal with the presentation of biometric streaming data.

GuiStreamingCallbackCtx (input) - a generic pointer to context information provided by the application that will be presented on the callback.

GuiStateCallback (input) - a pointer to an application callback to deal with GUI state changes.

GuiStateCallbackCtx (input) - a generic pointer to context information provided by the application that will be presented on the callback.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

See also the **BioAPI Error Handling**

2.5.2.3 BioAPI_SetStreamCallback

BioAPI_RETURN BioAPI BioAPI_SetStreamCallback

```
    (BioAPI_HANDLE  ModuleHandle,
    BioAPI_STREAM_CALLBACK  StreamCallback,
    void  *StreamCallbackCtx);
```

This function allows the application to establish a callback for client/server communication. The callback allows the BSP to send a protocol message to its partner BSP, and to receive a protocol message in exchange.

Parameters:

ModuleHandle (input) - the handle of the attached BioAPI service provider.

StreamCallback (input) - a pointer to an application callback to deal with the Client/Server transmission of protocol data units between BSPs

StreamCallbackCtx (input) - a generic pointer to context information provided by the application that will be presented on the callback.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors
See also the **BioAPI Error Handling**

2.5.2.4 BioAPI_StreamInputOutput

BioAPI_RETURN BioAPI BioAPI_StreamInputOutput

```
    (BioAPI_HANDLE   ModuleHandle,
    BioAPI_DATA_PTR  InMessage,
    BioAPI_DATA_PTR  OutMessage);
```

This function allows the application to pass a protocol data unit into the BSP from the partner BSP, and to obtain a response message to return to the partner BSP. (See section1.6)

Parameters:

ModuleHandle (input) - the handle of the attached BioAPI service provider.

InMessage (input) – *InMessage* contains a protocol data unit from the partner BSP.

OutMessage (output) – *OutMessage* contains a protocol data unit to be sent back to the partner BSP. If the parameter is NULL, there is no message to return.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

See also the **BioAPI Error Handling**

2.5.3 Biometric Operations

2.5.3.1 BioAPI_Capture

BioAPI_RETURN BioAPI BioAPI_Capture
(BioAPI_HANDLE ModuleHandle,
BioAPI_BIR_PURPOSE Purpose,
BioAPI_BIR_HANDLE_PTR CapturedBIR,
sint32 Timeout,
BioAPI_BIR_HANDLE_PTR AuditData);

This function captures samples for the purpose specified, and returns either an "intermediate" type BIR (if the Process function needs to be called), or a "processed" BIR (if not). The *Purpose* is recorded in the header of the *CapturedBIR*. If *AuditData* is non-NULL, a BIR of type "raw" may be returned. The function returns handles to whatever data is collected, and all local operations can be completed through use of the handles. If the application needs to acquire the data either to store it in a database or to send it to a server, the application can retrieve the data with the BioAPI_GetBIRFromHandle function. The application may request control of the GUI "look-and-feel" by providing a GUI callback pointer in BioAPI_SetGUICallbacks. Capture serializes use of the device. If two or more applications are racing for the device, the losers will wait until the timeout expires. This serialization takes place in all functions that capture data.

Parameters:

ModuleHandle (input) - The handle of the attached BioAPI service provider.

Purpose (input) - A value indicating the purpose of the biometric data capture.

CapturedBIR (output) – a handle to a BIR containing captured data. This data is either an "intermediate" type BIR, (which can only be used by either the **Process** or **CreateTemplate** functions, depending on the purpose), or a "processed" BIR, (which can be used directly by **VerifyMatch** or **IdentifyMatch**, depending on the purpose).

Timeout (input) – an integer specifying the timeout value (in milliseconds) for the operation. If this timeout is reached, the function returns an error, and no results. This value can be any positive number. A –1 value means the BSP's default timeout value will be used.

AuditData (output/optional) – a handle to a BIR containing raw biometric data. This data may be used to provide human-identifiable data of the person at the device. If the pointer is NULL on input, no audit data is collected. Not all BSPs support the collection of audit data. A BSP may return a handle value of BioAPI_UNSUPPORTED_BIR_HANDLE to indicate AuditData is not supported, or a value of BioAPI_INVALID_BIR_HANDLE to indicate that no audit data is available.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_BSP_UNABLE_TO_CAPTURE
BioAPIERR_BSP_TOO_MANY_HANDLES
BioAPIERR_BSP_TIMEOUT_EXPIRED
BioAPIERR_BSP_PURPOSE_NOT_SUPPORTED
See also the **BioAPI Error Handling**

2.5.3.2 BioAPI_CreateTemplate

BioAPI_RETURN BioAPI BioAPI_CreateTemplate

```
(BioAPI_HANDLE   ModuleHandle,
const BioAPI_INPUT_BIR  *CapturedBIR,
const BioAPI_INPUT_BIR   *StoredTemplate,
BioAPI_BIR_HANDLE_PTR  NewTemplate,
const BioAPI_DATA   *Payload);
```

This function takes a BIR containing raw biometric data for the purpose of creating a new enrollment template. A new BIR is constructed from the *CapturedBIR*, and (optionally) it may perform an adaptation based on an existing *StoredTemplate*. The old *StoredTemplate* remains unchanged. If the *StoredTemplate* contains a payload, the payload is not copied into the *NewTemplate*. If the *NewTemplate* needs a payload, then that *Payload* must be presented as an argument to the function.

Parameters:

ModuleHandle (input) - The handle of the attached BioAPI service provider.

CapturedBIR (input) – The captured BIR or its handle.

StoredTemplate (input/optional) – Optionally, the template to be adapted, or its key in a database, or its handle.

NewTemplate (output) – a handle to a newly created template that is derived from the *CapturedBIR* and (optionally) the *StoredTemplate*.

Payload (input/optional) – a pointer to data that will be wrapped inside the newly created template. This parameter is ignored, if NULL.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_BSP_BIR_SIGNATURE_FAILURE
BioAPIERR_BSP_TOO_MANY_HANDLES
BioAPIERR_BSP_UNABLE_TO_WRAP_PAYLOAD
BioAPIERR_BSP_INCONSISTENT_PURPOSE
BioAPIERR_BSP_PURPOSE_NOT_SUPPORTED
BioAPIERR_BSP_RECORD_NOT_FOUND
See also the **BioAPI Error Handling**

2.5.3.3 BioAPI_Process

BioAPI_RETURN BioAPI BioAPI_Process

```
        (BioAPI_HANDLE   ModuleHandle,
        const BioAPI_INPUT_BIR  *CapturedBIR,
        BioAPI_BIR_HANDLE_PTR  ProcessedBIR);
```

This function processes the intermediate data captured via a call to BioAPI_Capture for the purpose of either verification or identification. If the processing capability is in the attached BSP, the BSP builds a “processed” BIR, otherwise, *ProcessedBIR* is set to NULL.

Parameters:

ModuleHandle (input) - The handle of the attached BioAPI service provider.

CapturedBIR (input) – The captured BIR or its handle.

ProcessedBIR (output) – a handle for the newly constructed “processed” BIR, NULL.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_BSP_INVALID_BIR
BioAPIERR_BSP_BIR_SIGNATURE_FAILURE
BioAPIERR_BSP_TOO_MANY_HANDLES
BioAPIERR_BSP_INCONSISTENT_PURPOSE
BioAPIERR_BSP_PURPOSE_NOT_SUPPORTED
BioAPIERR_BSP_RECORD_NOT_FOUND
See also the **BioAPI Error Handling**

2.5.3.4 BioAPI_VerifyMatch

BioAPI_RETURN BioAPI BioAPI_VerifyMatch

```
        (BioAPI_HANDLE   ModuleHandle,
        const BioAPI_FAR  *MaxFARRequested,
        const BioAPI_FRR  *MaxFRRRequested,
        const BioAPI_BOOL  *FARPrecedence,
        const BioAPI_INPUT_BIR  *ProcessedBIR,
        const BioAPI_INPUT_BIR  *StoredTemplate,
        BioAPI_BIR_HANDLE  *AdaptedBIR,
        BioAPI_BOOL  *Result,
        BioAPI_FAR_PTR   FARAchieved,
        BioAPI_FRR_PTR   FRRAchieved,
        BioAPI_DATA_PTR  *Payload);
```

This function performs a verification (1-to-1) match between two BIRs; the *ProcessedBIR* and the *StoredTemplate*. The *ProcessedBIR* is the "processed" BIR constructed specifically for this verification. The *StoredTemplate* was created at enrollment. The application must request a maximum FAR value for a successful match, and may also (optionally) request a maximum FRR for a successful match. If a maximum FRR value is provided, the application must also indicate via the *FARPrecedence* parameter, which one takes precedence. The Boolean *Result* indicates whether verification was successful or not, and the *FARAchieved* is a FAR value indicating how closely the BIRs actually matched. The BSP implementation may optionally return the corresponding FRR that was achieved through the *FRRAchieved* return parameter. By setting the *AdaptedBIR* pointer to non-NULL, the application can request that a BIR be constructed by adapting the *StoredTemplate* using the *ProcessedBIR*. A new handle is returned to the *AdaptedBIR.* If the *StoredTemplate* contains a *Payload*, the *Payload* may be returned upon successful verification if the *FARAchieved* is sufficiently stringent; this is controlled by the policy of the BSP.

If the match is successful, an attempt may be made to adapt the *StoredTemplate* with information taken from the *ProcessedBIR*. (Not all BSPs perform adaptation). The resulting *AdaptedBIR* should now be considered an optimal enrollment template, and be saved in the enrollment database. (It is up to the application whether or not it uses or discards this data). It is important to note that adaptation may not occur in all cases.

In the event of an adaptation, this function stores the handle to the new BIR in the memory pointed to by the *AdaptedBIR* parameter.

Parameters:

ModuleHandle (input) - The handle of the attached BioAPI service provider.

MaxFARRequested (input) - The requested FAR criterion for successful verification.

MaxFRRRequested (input/optional) - The requested FRR criterion for successful verification. A NULL pointer indicates that this criterion is not provided.

FARPrecedence (input) – If both criteria are provided, this parameter indicates which takes precedence. BioAPI_TRUE for FAR; BioAPI_FALSE for FRR.

ProcessedBIR (input) – The BIR to be verified, or its handle.

StoredTemplate (input) – The BIR to be verified against, or its key in a database, or its handle.

AdaptedBIR (output/optional) – a pointer to the handle of the adapted BIR. This parameter can be NULL if an Adapted BIR is not desired. Not all BSPs support the adaptation of BIRs. The function may return a handle value of BioAPI_UNSUPPORTED_BIR_HANDLE to indicate that adaptation is not supported or a value of BioAPI_INVALID_BIR_HANDLE to indicate that adaptation was not possible.

Result (output) – a pointer to a Boolean value indicating (BioAPI_TRUE/BioAPI_FALSE) whether the BIRs matched or not according to the specified criteria.

FARAchieved (output) – a pointer to an FAR value indicating the closeness of the match.

FRRAchieved (output/optional) – a pointer to an FRR value indicating the closeness of the match.

Payload (output/optional) – if the *StoredTemplate* contains a payload, it is returned in an allocated BioAPI_DATA structure if the *FARAchieved* satisfies the policy of the BSP.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_BSP_BIR_SIGNATURE_FAILURE
BioAPIERR_BSP_INCONSISTENT_PURPOSE
BioAPIERR_BSP_BIR_NOT_FULLY_PROCESSED
BioAPIERR_BSP_RECORD_NOT_FOUND
See also the **BioAPI Error Handling**

2.5.3.5 BioAPI_IdentifyMatch

```
BioAPI_RETURN BioAPI BioAPI_IdentifyMatch
        (BioAPI_HANDLE   ModuleHandle,
        const BioAPI_FAR  *MaxFARRequested,
        const BioAPI_FRR  *MaxFRRRequested,
        const BioAPI_BOOL  *FARPrecedence,
        const BioAPI_INPUT_BIR  *ProcessedBIR,
        const BioAPI_IDENTIFY_POPULATION   *Population,
        BioAPI_BOOL  Binning,
        uint32  MaxNumberOfResults,
        uint32  *NumberOfResults,
        BioAPI_CANDIDATE_ARRAY_PTR  *Candidates,
        sint32 Timeout);
```

This function performs an identification (1-to-many) match between a *ProcessedBIR* and a set of stored BIRs. The *ProcessedBIR* is the “processed” BIR captured specifically for this identification. The population that the match takes place against can be presented in one of three ways:

1. in a database identified by an open database handle.
2. input in an array of BIRs
3. in the “default” database of the BSP (possibly stored in the biometric device)

The application must request a maximum FAR value criterion for a successful match, and may also (optionally) request a maximum FRR criterion for a successful match. If a maximum FRR value is provided, the application must also indicate via the *FARPrecedence* parameter, which criterion takes precedence. The FARAchieved and, optionally, the FRRAchieved are returned for each result in the *Candidate* array.

Parameters:

ModuleHandle (input) - The handle of the attached BioAPI service provider.

MaxFARRequested (input) - The requested FAR criterion for successful verification.

MaxFRRRequested (input/optional) - The requested FRR criterion for successful verification. A NULL pointer indicates that this criterion is not provided.

FARPrecedence (input) – If both criteria are provided, this parameter indicates which takes precedence. BioAPI_TRUE for FAR; BioAPI_FALSE for FRR.

ProcessedBIR (input) – The BIR to be identified.

Population (input) – the population of BIRs against which the Identify match is performed.

Binning (input) – A Boolean indicating whether *Binning* is on or off. Binning is a search optimization technique that the BSP may employ. It is based on searching the population according to the intrinsic characteristics of the biometric data. While it may improve the speed of the Match operation, it may also increase the probability of missing a candidate.

MaxNumberOfResults (input) – specifies the maximum number of match candidates to be returned as a result of the 1:N match. A value of zero is a request for all candidates.

NumberOfResults (output) – specifies the number of candidates returned in the *Candidates* array as a result of the 1:N match.

Candidates (output) – a pointer to an array of BioAPI_CANDIDATE_PTRs corresponding to the BIRs identified as a result of the match process (i.e., indices associated with BIRs found to exceed the match threshold). This list is in rank order, with the highest scoring record being first. If no matches are found, this pointer will be set to NULL. If the *Population* was presented in a database, the IDs are database IDs; if the set was presented in an in-memory array, the IDs are indexes into the array.

Timeout (input) – an integer specifying the timeout value (in milliseconds) for the operation. If this timeout is reached, the function returns an error, and no candidate list. This value can be any positive number. A –1 value means the BSP's default timeout value will be used.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_BSP_BIR_SIGNATURE_FAILURE
BioAPIERR_BSP_TIMEOUT_EXPIRED
BioAPIERR_BSP_NO_INPUT_BIRS
BioAPIERR_BSP_FUNCTION_NOT_SUPPORTED
BioAPIERR_BSP_INCONSISTENT_PURPOSE
BioAPIERR_BSP_BIR_NOT_FULLY_PROCESSED
BioAPIERR_BSP_RECORD_NOT_FOUND
See also the **BioAPI Error Handling**

Remarks

Not all BSPs support 1:N identification

Depending on the BSP and the location and size of the database to be searched, this operation can take a significant amount of time to perform

The number of match candidates found by the BSP is dependent on the *Actual FAR*.

2.5.3.6 BioAPI_Enroll

BioAPI_RETURN BioAPI BioAPI_Enroll
(BioAPI_HANDLE ModuleHandle,
BioAPI_BIR_PURPOSE Purpose,
const BioAPI_INPUT_BIR *StoredTemplate,
BioAPI_BIR_HANDLE_PTR NewTemplate,
const BioAPI_DATA *Payload,
sint32 Timeout
BioAPI_BIR_HANDLE_PTR AuditData);

This function captures biometric data from the attached device for the purpose of creating a *ProcessedBIR* for the purpose of enrollment. The **Enroll** function may be split between client and server if a streaming callback has been set. Either the client or the server can initiate the operation.

Parameters:

ModuleHandle (input) - The handle of the attached BioAPI service provider.

Purpose (input) – A value indicating the desired purpose of Enrollment.

StoredTemplate (input/optional) – Optionally, the BIR to be adapted, or its key in a database, or its handle.

NewTemplate (output) – a handle to a newly created template that is derived from the new raw samples and (optionally) the *StoredTemplate*.

Payload (input/optional) – a pointer to data that will be wrapped inside the newly created template. This parameter is ignored, if NULL.

Timeout (input) – an integer specifying the timeout value (in milliseconds) for the operation. If this timeout is reached, the function returns an error, and no results. This value can be any positive number. A –1 value means the BSP's default timeout value will be used.

AuditData (output/optional) – a handle to a BIR containing biometric audit data. This data may be used to provide a human-identifiable data of the person at the device. If the pointer is NULL on input, no audit data is collected. Not all BSPs support the collection of Audit data. A BSP may return a handle value of BioAPI_UNSUPPORTED_BIR_HANDLE to indicate AuditData is not supported, or a value of BioAPI_INVALID_BIR_HANDLE to indicate that no audit data is available.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors
BioAPIERR_BSP_UNABLE_TO_CAPTURE
BioAPIERR_BSP_TOO_MANY_HANDLES
BioAPIERR_BSP_UNABLE_TO_WRAP_PAYLOAD
BioAPIERR_BSP_TIMEOUT_EXPIRED
BioAPIERR_BSP_PURPOSE_NOT_SUPPORTED
BioAPIERR_BSP_RECORD_NOT_FOUND
See also the **BioAPI Error Handling**

2.5.3.7 BioAPI_Verify

BioAPI_RETURN BioAPI BioAPI_Verify

```
(BioAPI_HANDLE   ModuleHandle,
const BioAPI_FAR  *MaxFARRequested,
const BioAPI_FRR  *MaxFRRRequested,
const BioAPI_BOOL  *FARPrecedence,
const BioAPI_INPUT_BIR  *StoredTemplate,
BioAPI_BIR_HANDLE_PTR  AdaptedBIR,
BioAPI_BOOL *Result,
BioAPI_FAR_PTR   FARAchieved,
BioAPI_FRR_PTR   FRRAchieved,
BioAPI_DATA_PTR  *Payload,
sint32 Timeout,
BioAPI_BIR_HANDLE_PTR  AuditData);
```

This function captures biometric data from the attached device, and compares it against the *StoredTemplate*. The application must request a maximum FAR value criterion for a successful match, and may also (optionally) request a maximum FRR criterion for a successful match. If a maximum FRR value is provided, the application must also indicate via the *FARPrecedence* parameter, which criterion takes precedence. The Boolean *Result* indicates whether verification was successful or not, and the *FARAchieved* is a FAR value indicating how closely the BIRs actually matched. The BSP implementation may optionally return the corresponding FRR that was achieved through the *FRRAchieved* return parameter. If the *StoredTemplate* contains a payload, the *Payload* may be returned upon successful verification. Optionally, a new *AdaptedBIR* may be constructed. The **Verify** function may be split between client and server if a streaming callback has been set. Either the client or the server can initiate the operation.

If the match is successful, an attempt may be made to adapt the *StoredTemplate* with information taken from the *ProcessedBIR*. (Not all BSPs perform adaptation). The resulting *AdaptedBIR* should now be considered an optimal enrollment, and be saved in the enrollment database. (It is up to the application whether or not it uses or discards this data). It is important to note that adaptation may not occur in all cases.

In the event of an adaptation, this function stores the handle to the new BIR in the memory pointed to by the *AdaptedBIR* parameter.

Parameters:

ModuleHandle (input) - The handle of the attached BioAPI service provider.

MaxFARRequested (input) - The requested FAR criterion for successful verification.

MaxFRRRequested (input/optional) - The requested FRR criterion for successful verification. A NULL pointer indicates that this criterion is not provided.

FARPrecedence (input) – If both criteria are provided, this parameter indicates which takes precedence. BioAPI_TRUE for FAR; BioAPI_FALSE for FRR.

StoredTemplate (input) – The BIR to be verified against, or its key in a database, or its handle.

AdaptedBIR (output/optional) – a pointer to the handle of the adapted BIR. This parameter can be NULL if an Adapted BIR is not desired. Not all BSPs support the adaptation of BIRs. The function may return a handle value of BioAPI_UNSUPPORTED_BIR_HANDLE to indicate that adaptation is not supported or a value of BioAPI_INVALID_BIR_HANDLE to indicate that adaptation was not possible.

Result (output) – a pointer to a Boolean value indicating (BioAPI_TRUE/BioAPI_FALSE) whether the BIRs matched or not according to the specified criteria.

FARAchieved (output) – a pointer to an FAR value indicating the closeness of the match.

FRRAchieved (output/optional) – a pointer to an FRR value indicating the closeness of the match.

Payload (output/optional) – if the *StoredTemplate* contains a payload, it is returned in an allocated BioAPI_DATA structure if the *FARAchieved* satisfies the policy of the BSP.

Timeout (input) – an integer specifying the timeout value (in milliseconds) for the operation. If this timeout is reached, the function returns an error, and no results. This value can be any positive number. A –1 value means the BSP's default timeout value will be used.

AuditData (output/optional) – a handle to a BIR containing raw biometric data. This data may be used to provide human-identifiable data of the person at the device. If the pointer is NULL on input, no audit data is collected. Not all BSPs support the collection of Audit data. A BSP may return a handle value of BioAPI_UNSUPPORTED_BIR_HANDLE to indicate AuditData is not supported, or a value of BioAPI_INVALID_BIR_HANDLE to indicate that no audit data is available.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Error Codes

BioAPIERR_BSP_UNABLE_TO_CAPTURE
BioAPIERR_BSP_BIR_SIGNATURE_FAILURE
BioAPIERR_BSP_TOO_MANY_HANDLES
BioAPIERR_BSP_TIMEOUT_EXPIRED
BioAPIERR_BSP_INCONSISTENT_PURPOSE
BioAPIERR_BSP_RECORD_NOT_FOUND
See also the **BioAPI Error Handling**

2.5.3.8 BioAPI_Identify

BioAPI_RETURN BioAPI BioAPI_Identify

```
    (BioAPI_HANDLE   ModuleHandle,
    const BioAPI_FAR  *MaxFARRequested,
    const BioAPI_FRR  *MaxFRRRequested,
    const BioAPI_BOOL  *FARPrecedence,
    const BioAPI_IDENTIFY_POPULATION   *Population,
    BioAPI_BOOL  Binning,
    uint32  MaxNumberOfResults,
    uint32  *NumberOfResults,
    BioAPI_CANDIDATE_ARRAY_PTR  *Candidates,
    sint32  Timeout,
    BioAPI_BIR_HANDLE_PTR  AuditData);
```

This function captures biometric data from the attached device, and compares it against the *Population*. The application must request a maximum FAR value criterion for a successful match, and may also (optionally) request a maximum FRR criterion for a successful match. If a maximum FRR value is provided, the application must also indicate via the *FARPrecedence* parameter, which criterion takes precedence. The function returns a number of candidates from the population that match according to the specified criteria, and the FARAchieved and, optionally, the FRRAchieved are returned for each result in the *Candidate* array.

The **Identify** function may be split between client and server if a streaming callback has been set. Either the client or the server can initiate the operation.

Parameters:

ModuleHandle (input) - The handle of the attached BioAPI service provider.

MaxFARRequested (input) - The requested FAR criterion for successful identification.

MaxFRRRequested (input/optional) - The requested FRR criterion for successful identification. A NULL pointer indicates that this criterion is not provided.

FARPrecedence (input) – If both criteria are provided, this parameter indicates which takes precedence. BioAPI_TRUE for FAR; BioAPI_FALSE for FRR.

Population (input) – the population of Templates against which the **Identify** match is performed.

Binning (input) – A Boolean indicating whether binning is on or off. Binning is a search optimization technique that the BSP may employ. It is based on searching the population according to the intrinsic characteristics of the biometric data. While it may improve the speed of the Match operation, it may also increase the probability of missing a candidate.

MaxNumberOfResults (input) – specifies the maximum number of match candidates to be returned as a result of the 1:N match. A value of zero is a request for all candidates.

NumberofResults (output) – specifies the number of candidates returned in the *Candidates* array as a result of the 1:N match.

Candidates (output) – a pointer to an array of BioAPI_CANDIDATE_PTRs corresponding to the BIRs identified as a result of the match process (i.e., indices associated with BIRs found to exceed the match threshold). This list is in rank order, with the highest scoring record being first. If no matches are found, this pointer will be set to NULL. If the *Population* was presented in a database, the IDs are database IDs; if the set was presented in an in-memory array, the IDs are indexes into the array.

Timeout (input) – an integer specifying the timeout value (in milliseconds) for the operation. If this timeout is reached, the function returns an error, and no results. This value can be any positive number. A –1 value means the BSP's default timeout value will be used.

AuditData (output/optional) – a handle to a BIR containing raw biometric data. This data may be used to provide human-identifiable data of the person at the device. If the pointer is NULL on input, no audit data is collected. Not all BSPs support the collection of Audit data. A BSP may return a handle value of BioAPI_UNSUPPORTED_BIR_HANDLE to indicate AuditData is not supported, or a value of BioAPI_INVALID_BIR_HANDLE to indicate that no audit data is available.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_BSP_UNABLE_TO_CAPTURE
BioAPIERR_BSP_TOO_MANY_HANDLES
BioAPIERR_BSP_BIR_SIGNATURE_FAILURE
BioAPIERR_BSP_TIMEOUT_EXPIRED
BioAPIERR_BSP_NO_INPUT_BIRS
BioAPIERR_BSP_FUNCTION_NOT_SUPPORTED
BioAPIERR_BSP_INCONSISTENT_PURPOSE
BioAPIERR_BSP_RECORD_NOT_FOUND
See also the **BioAPI Error Handling**

Remarks

Not all BSPs support 1:N identification. See your BSP programmer's manual to determine if the BSP(s) you are using supports this capability.

Depending on the BSP and the location and size of the database to be searched, this operation can take a significant amount of time to perform. Check your BSP manual for recommended *Timeout* values.

The number of match candidates found by the BSP is dependent on the *ActualFAR* used.

2.5.3.9 BioAPI_Import

BioAPI_RETURN BioAPI BioAPI_Import

```
(BioAPI_HANDLE   ModuleHandle,
const BioAPI_DATA  *InputData,
BioAPI_BIR_BIOMETRIC_DATA_FORMAT   InputFormat,
BioAPI_BIR_PURPOSE  Purpose,
BioAPI_BIR_HANDLE_PTR  ConstructedBIR);
```

This function imports non-real-time raw biometric data to construct a BIR for the purpose specified. *InputData* identifies the memory buffer containing the raw biometric data, while *InputFormat* identifies the form of the raw biometric data. The function returns a handle to the *ConstructedBIR*. If the application needs to acquire the BIR either to store it in a database or to send it to a server, the application can retrieve the data with the BioAPI_GetBIRFromHandle function, or store it directly using BioAPI_DbStoreBIR.

Parameters:

ModuleHandle (input) - The handle of the attached BioAPI service provider.

InputData (input) – A pointer to image/stream data to import into a Processed BIR. The image/stream conforms to the standard identified by *InputFormat*.

InputFormat (input) – The format of the *InputData*.

Purpose (input) - A value indicating the Enroll purpose..

ConstructedBIR (output) – a handle to a BIR constructed from the imported biometric data. This BIR may be either an Intermediate or Processed BIR (as indicated in the header).

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_BSP_UNSUPPORTED_FORMAT
BioAPIERR_BSP_UNABLE_TO_IMPORT
BioAPIERR_BSP_TOO_MANY_HANDLES
BioAPIERR_BSP_FUNCTION_NOT_SUPPORTED
BioAPIERR_BSP_PURPOSE_NOT_SUPPORTED
See also the **BioAPI Error Handling**

2.5.3.10 BioAPI_SetPowerMode

BioAPI_RETURN BioAPI BioAPI_SetPowerMode
(BioAPI_HANDLE ModuleHandle,
BioAPI_POWER_MODE PowerMode);

This function sets the device to the requested power mode if the device supports it.

Parameters:

ModuleHandle (input) - The handle of the attached BioAPI service provider.

PowerMode (input) – a 32-bit value indicting the power mode to set the device to.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_BSP_FUNCTION_NOT_SUPPORTED
See also the **BioAPI Error Handling**

2.5.4 Database Operations

2.5.4.1 BioAPI_DbOpen

BioAPI_RETURN BioAPI BioAPI_DbOpen

```
(BioAPI_HANDLE   ModuleHandle,
const uint8  *DbName,
BioAPI_DB_ACCESS_TYPE  AccessRequest,
BioAPI_DB_HANDLE_PTR  DbHandle,
BioAPI_DB_CURSOR_PTR  Cursor);
```

This function opens the data store with the specified name under the specified access mode. A database *Cursor* is set to point to the first record in the database. Note: the default database (if any) is always open.

Parameters

ModuleHandle (input) - The handle of the attached BioAPI service provider.

DbName (input) - A pointer to the null-terminated string containing the name of the database.

AccessRequest (input) – An indicator of the requested access mode for the database, such as **read** or **write**.

DbHandle (output) - The handle to the opened data store. The value will be set to BioAPI_DB_INVALID_HANDLE if the function fails.

Cursor (output) – A handle that can be used to iterate through the database.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_BSP_UNABLE_TO_OPEN_DATABASE
BioAPIERR_BSP_DATABASE_IS_LOCKED
BioAPIERR_BSP_DATABASE_DOES_NOT_EXIST
BioAPIERR_BSP_INVALID_DATABASE_NAME
BioAPIERR_BSP_INVALID_ACCESS_REQUEST
See also the **BioAPI Error Handling**

2.5.4.2 BioAPI_DbClose

BioAPI_RETURN BioAPI BioAPI_DbClose
(BioAPI_HANDLE ModuleHandle,
BioAPI_DB_HANDLE DbHandle);

This function closes an open database. All cursors currently set to the database are freed.

Parameters

ModuleHandle (input) - The handle of the attached BioAPI service provider.

DbHandle (input) - The DB handle for an open database managed by the BSP. This specifies the open database to be closed.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

See also the **BioAPI Error Handling**

2.5.4.3 BioAPI_DbCreate

BioAPI_RETURN BioAPI BioAPI_DbCreate

```
(BioAPI_HANDLE   ModuleHandle,
const uint8  *DbName,
BioAPI_DB_ACCESS_TYPE  AccessRequest,
BioAPI_DB_HANDLE_PTR DbHandle);
```

This function creates and opens a new database. The name of the new database is specified by the input parameter DbName. The newly created database is opened under the specified access mode.

Parameters

ModuleHandle (input) - The handle of the attached BioAPI service provider.

DbName (input) - A pointer to the null-terminated string containing the name of the new database.

AccessRequest (input) – An indicator of the requested access mode for the database, such as **read** or **write**.

DbHandle (output) - The handle to the newly created and open data store. The value will be set to BioAPI_DB_INVALID_HANDLE if the function fails.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_BSP_DATABASE_ALREADY_EXISTS
BioAPIERR_BSP_INVALID_DATABASE_NAME
BioAPIERR_BSP_INVALID_ACCESS_REQUEST
See also the **BioAPI Error Handling**

2.5.4.4 BioAPI_DbDelete

BioAPI_RETURN BioAPI BioAPI_DbDelete
(BioAPI_HANDLE ModuleHandle,
const uint8 *DbName);

This function deletes all records from the specified database and removes all state information associated with that database.

Parameters

ModuleHandle (input) - The handle of the attached BioAPI service provider.

DbName (input) - A pointer to the null-terminated string containing the name of the database to be deleted.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_BSP_DATABASE_IS_OPEN
BioAPIERR_BSP_INVALID_DATABASE_NAME
See also the **BioAPI Error Handling**

2.5.4.5 BioAPI_DbSetCursor

BioAPI_RETURN BioAPI BioAPI_DbSetCursor

```
(BioAPI_HANDLE   ModuleHandle,
BioAPI_DB_HANDLE   DbHandle,
const BioAPI_UUID  *KeyValue,
BioAPI_DB_CURSOR_PTR  Cursor);
```

The *Cursor* is set to point to the record indicated by the *KeyValue* in the database identified by the *DbHandle*. A NULL value will set the cursor to the first record in the database.

Parameters

ModuleHandle (input) - The handle of the attached BioAPI service provider.

DbHandle (input) - A handle to the open database.

KeyValue (input) – The key into the database of the BIR to which the *Cursor* is to be set.

Cursor (output) – A handle that can be used to iterate through the database from the retrieved record.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_BSP_RECORD_NOT_FOUND
See also the **BioAPI Error Handling**

2.5.4.6 BioAPI_DbFreeCursor

BioAPI_RETURN BioAPI BioAPI_DbFreeCursor
(BioAPI_HANDLE ModuleHandle,
BioAPI_DB_CURSOR_PTR Cursor);

Frees memory and resources associated with the specified *Cursor*.

Parameters

ModuleHandle (input) - The handle of the attached BioAPI service provider.

Cursor (input) – The Database Cursor to be freed.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_BSP_CURSOR_IS_INVALID
See also the **BioAPI Error Handling**

2.5.4.7 BioAPI_DbStoreBIR

BioAPI_RETURN BioAPI BioAPI_DbStoreBIR

```
(BioAPI_HANDLE   ModuleHandle,
const BioAPI_INPUT_BIR   *BIRToStore,
BioAPI_DB_HANDLE   DbHandle,
BioAPI_UUID_PTR  Uuid);
```

The BIR identified by the *BIRToStore* parameter is stored in the open database identified by the *DbHandle* parameter. If the *BIRToStore* is identified by a BIR Handle, the input BIR Handle is freed. If the *BIRToStore* is identified by a database key value, the BIR is copied to the open database. A new UUID is assigned to the new BIR in the database, and this UUID can be used as a key value to access the BIR later.

Parameters

ModuleHandle (input) - The handle of the attached BioAPI service provider.

BIRToStore (input) – The BIR to be stored in the open database (either the BIR, or its handle, or the index to it in another open database).

DbHandle (input) - The handle to the open database.

Uuid (output) – A UUID that uniquely identifies the new BIR in the database. This UUID cannot be changed. To associate a different BIR with the user, it is necessary to delete the old one, store a new one in the database, and then replace the old UUID with the new one in the application account database.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

See also the **BioAPI Error Handling**

2.5.4.8 BioAPI_DbGetBIR

BioAPI_RETURN BioAPI BioAPI_DbGetBIR

```
(BioAPI_HANDLE   ModuleHandle,
BioAPI_DB_HANDLE   DbHandle,
const BioAPI_UUID  *KeyValue,
BioAPI_BIR_HANDLE_PTR  RetrievedBIR,
BioAPI_DB_CURSOR_PTR  Cursor);
```

The BIR identified by the *KeyValue* parameter in the open database identified by the *DbHandle* parameter is retrieved. The BIR is copied into BSP storage and a handle to it is returned. The *Cursor* is set to point to the next record, or the first record in the database if the retrieved BIR is the last.

Parameters

ModuleHandle (input) - The handle of the attached BioAPI service provider.

DbHandle (input) - The handle to the open database.

KeyValue (input) – The key into the database of the BIR to retrieve.

RetievedBIR (output) – A handle to the retrieved BIR.

Cursor (output) – A handle that can be used to iterate through the database from the retrieved record.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_BSP_RECORD_NOT_FOUND
See also the **BioAPI Error Handling**

2.5.4.9 BioAPI_DbGetNextBIR

BioAPI_RETURN BioAPI BioAPI_DbGetNextBIR
(BioAPI_HANDLE ModuleHandle,
BioAPI_DB_CURSOR_PTR Cursor,
BioAPI_BIR_HANDLE_PTR RetievedBIR,
BioAPI_UUID_PTR Uuid);

The BIR identified by the *Cursor* parameter is retrieved. The BIR is copied into BSP storage and a handle to it is returned, and a pointer to the UUID that uniquely identifies the BIR in the database is returned.. The *Cursor* is updated to the next record in the database, or to the first when the end of the database is reached.

Parameters

ModuleHandle (input) - The handle of the attached BioAPI service provider.

Cursor (input/output) – A handle indicating which record to retrieve.

RetrievedBIR (output) – A handle to the retrieved BIR.

Uuid (output) – The UUID that uniquely identifies the retrieved BIR in the database.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_BSP_END_OF_DATABASE
BioAPIERR_BSP_CURSOR_IS_INVALID
See also the **BioAPI Error Handling**

2.5.4.10 BioAPI_DbQueryBIR

BioAPI_RETURN BioAPI BioAPI_DbQueryBIR

```
    (BioAPI_HANDLE   ModuleHandle,
    BioAPI_DB_HANDLE   DbHandle,
    const BioAPI_INPUT_BIR   *BIRToQuery,
    BioAPI_UUID_PTR  Uuid);
```

If the BIR identified by the *BIRToQuery* parameter is in the open database identified by the *DbHandle* parameter, a pointer to its UUID is returned. Otherwise, BioAPIERR_BSP_RECORD_NOT_FOUND is returned..

Parameters

ModuleHandle (input) - The handle of the attached BioAPI service provider.

DbHandle (input) - The handle to the open database.

BIRToQuery (input) – The BIR to be queried in the open database (either the BIR, or its handle, or the key to it in another open database).

Uuid (output) – The UUID that uniquely identifies the BIR in the database.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_BSP_RECORD_NOT_FOUND
See also the **BioAPI Error Handling**

2.5.4.11 BioAPI_DbDeleteBIR

BioAPI_RETURN BioAPI BioAPI_DbDeleteBIR
(BioAPI_HANDLE ModuleHandle,
BioAPI_DB_HANDLE DbHandle,
const BioAPI_UUID *KeyValue);

The BIR identified by the *KeyValue* parameter in the open database identified by the *DbHandle* parameter is deleted from the database. If there is a cursor set to the deleted BIR, then the cursor is moved to the next sequential BIR (or set to the start of the database if there are no more records).

Parameters

ModuleHandle (input) - The handle of the attached BioAPI service provider.

DbHandle (input) - The handle to the open database.

KeyValue (input) – The UUID of the BIR to be deleted.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

BioAPIERR_BSP_RECORD_NOT_FOUND
BioAPIERR_BSP_END_OF_DATABASE
See also the **BioAPI Error Handling**

3 BioAPI Service Provider Interface

3.1 Summary

The service provider Interface (SPI) is the programming interface that a BSP must present in order to plug into the BioAPI framework. In general the SPI is a one-to-one mapping of the BioAPI down to the BSP. The framework routes API calls down to the corresponding SPI of the attached BSP (as determined by the Attach handle parameter in the API function. However, there are some exceptions.

Where the SPI is identical to the API, all that is presented in this section is the signature of the SPI function with no explanation. The reader can then refer to the corresponding API definition.

Where the SPI differs from the API, a full definition of the SPI function is presented.

Not all API functions have a corresponding SPI function; those functions are handled completely by the framework.

3.2 Data Structure for service providers

3.2.1 BioSPI_ModuleEventHandler

This defines the event handler interface that the BioAPI H-level framework defines and implements to receive asynchronous notification of events of type BioAPI_MODULE_EVENT from a service provider module. Example events include insertion or removal of a hardware service module, or fault detection.

This event structure is passed to the service module during BioSPI_ModuleLoad. This is the single event handler the service module should use to notify the BioAPI H-level framework of these event types for all of the attached sessions with the loaded module. BioAPI forwards the event to the entity that invoked the corresponding BioAPI_ModuleLoad function. The handler specified in BioSPI_ModuleEventHandler can be invoked multiple times in response to a single event (such as the insertion of a smartcard).

```
typedef BioAPI_RETURN (*BioSPI_ModuleEventHandler) (
       const BioAPI_UUID *BSPUuid,
       void* BioAPINotifyCallbackCtx,
       BioAPI_DEVICE_ID DeviceID,
       uint32  Reserved,
       BioAPI_MODULE_EVENT EventType);
```

Definition

BSPUuid - The UUID of the service module raising the event.

BioAPINotifyCallbackCtx - A BioAPI context specified during BioSPI_ModuleLoad( ).

DeviceID - The DeviceID of the service module raising the event.

Reserved - A reserved input; should be set to 0.

EventType - The BioAPI_MODULE_EVENT that has occurred.

3.2.2 BioAPI_MODULE_FUNCS

This structure is used by BSPs to return function pointers for all service provider interfaces that can be invoked by the BioAPI framework. This includes interfaces to real biometric services and interfaces to administrative functions used by the BioAPI framework and the service provider to maintain the environment. Many operating environments provide platform-specific support for strong type checking applied to function pointers. This specification allows for the use of such mechanisms when available. In the general case a function pointer is considered to be a value of type BioAPI_PROC_ADDR.

```
typedef struct bioapi_module_funcs {
       uint32 Reserved;
       uint32 NumberOfServiceFuncs;
       const BioAPI_PROC_ADDR *ServiceFuncs;
} BioAPI_MODULE_FUNCS, *BioAPI_MODULE_FUNCS_PTR;
```

Definition

Reserved - A reserved input; should be set to 0.

NumberOfServiceFuncs - The number of function pointers for the service functions contained in the table of *ServiceFuncs*.

ServiceFuncs - Memory address of the beginning of the function pointer table for BSP.

3.2.3 BSP Function Pointer Table

This structure defines the function table for all the BSP functions that a service provider can return to the BioAPI Framework on BioAPI_ModuleAttach. The BioAPI Framework uses these pointers to dispatch corresponding application programming interface functions to the BSP for processing.

```
typedef struct bioapi_bsp_funcs {
       BioAPI_RETURN (BioAPI  *FreeBIRHandle)
              (BioAPI_HANDLE   ModuleHandle,
              BioAPI_BIR_HANDLE   Handle);
       BioAPI_RETURN (BioAPI  *GetBIRFromHandle)
              (BioAPI_HANDLE   ModuleHandle,
              BioAPI_BIR_HANDLE   Handle,
              BioAPI_BIR_PTR BIR);
       BioAPI_RETURN  (BioAPI  *GetHeaderFromHandle)
              (BioAPI_HANDLE   ModuleHandle,
              BioAPI_BIR_HANDLE   Handle,
              BioAPI_BIR_HEADER_PTR  Header);
       BioAPI_RETURN (BioAPI *EnableEvents)
              (BioAPI_HANDLE  ModuleHandle,
              BioAPI_MODULE_EVENT_MASK  *Events);
       BioAPI_RETURN (BioAPI *SetGUICallbacks)
              (BioAPI_HANDLE  ModuleHandle,
              BioAPI_GUI_STREAMING_CALLBACK  GuiStreamingCallback,
              void  *GuiStreamingCallbackCtx,
              BioAPI_GUI_STATE_CALLBACK  GuiStateCallback,
              void  *GuiStateCallbackCtx);
       BioAPI_RETURN (BioAPI *SetStreamCallback)
              (BioAPI_HANDLE  ModuleHandle,
              BioAPI_STREAM_CALLBACK  StreamCallback,
              void  *StreamCallbackCtx);
       BioAPI_RETURN (BioAPI *StreamInputOutput)
              (BioAPI_HANDLE   ModuleHandle,
              BioAPI_DATA_PTR InMessage,
              BioAPI_DATA_PTR  OutMessage);
       BioAPI_RETURN (BioAPI *Capture)
              (BioAPI_HANDLE   ModuleHandle,
              BioAPI_BIR_PURPOSE   Purpose,
              BioAPI_BIR_HANDLE_PTR  CapturedBIR,
              sint32  Timeout,
              BioAPI_BIR_HANDLE_PTR  AuditData);
       BioAPI_RETURN (BioAPI *CreateTemplate)
              (BioAPI_HANDLE   ModuleHandle,
              const BioAPI_INPUT_BIR  *CapturedBIR,
              const BioAPI_INPUT_BIR   *StoredTemplate,
              BioAPI_BIR_HANDLE_PTR  NewTemplate,
              const BioAPI_DATA   *Payload);
       BioAPI_RETURN (BioAPI *Process)
              (BioAPI_HANDLE   ModuleHandle,
              const BioAPI_INPUT_BIR  *CapturedBIR,
              BioAPI_BIR_HANDLE_PTR  ProcessedBIR);
```

```
BioAPI_RETURN (BioAPI *VerifyMatch)
       (BioAPI_HANDLE   ModuleHandle,
       const BioAPI_FAR  *MaxFARRequested,
       const BioAPI_FRR  *MaxFRRRequested,
       const BioAPI_BOOL  *FARPrecedence,
       const BioAPI_INPUT_BIR  *ProcessedBIR,
       const BioAPI_INPUT_BIR  *StoredTemplate,
       BioAPI_BIR_HANDLE  *AdaptedBIR,
       BioAPI_BOOL  *Result,
       BioAPI_FAR_PTR   FARAchieved,
       BioAPI_FRR_PTR   FRRAchieved,
       BioAPI_DATA_PTR  Payload);
BioAPI_RETURN (BioAPI *IdentifyMatch)
       (BioAPI_HANDLE   ModuleHandle,
       const BioAPI_FAR  *MaxFARRequested,
       const BioAPI_FRR  *MaxFRRRequested,
       const BioAPI_BOOL  *FARPrecedence,
       const BioAPI_INPUT_BIR  *ProcessedBIR,
       const BioAPI_IDENTIFY_POPULATION   *Population,
       BioAPI_BOOL  Binning,
       uint32  MaxNumberOfResults,
       uint32  *NumberOfResults,
       BioAPI_CANDIDATE_ARRAY_PTR  *Candidates,
       sint32 Timeout);
BioAPI_RETURN (BioAPI *Enroll)
       (BioAPI_HANDLE   ModuleHandle,
       BioAPI_BIR_PURPOSE  Purpose,
       const BioAPI_INPUT_BIR   *StoredTemplate,
       BioAPI_BIR_HANDLE_PTR NewTemplate,
       const BioAPI_DATA   *Payload,
       sint32 Timeout
       BioAPI_BIR_HANDLE_PTR  AuditData);
BioAPI_RETURN (BioAPI *Verify)
       (BioAPI_HANDLE   ModuleHandle,
       const BioAPI_FAR  *MaxFARRequested,
       const BioAPI_FRR  *MaxFRRRequested,
       const BioAPI_BOOL  *FARPrecedence,
       const BioAPI_INPUT_BIR  *StoredTemplate,
       BioAPI_BIR_HANDLE_PTR  AdaptedBIR,
       BioAPI_BOOL *Result,
       BioAPI_FAR_PTR   FARAchieved,
       BioAPI_FRR_PTR   FRRAchieved,
       BioAPI_DATA_PTR  Payload,
       sint32 Timeout,
       BioAPI_BIR_HANDLE_PTR  AuditData);
       BioAPI_RETURN (BioAPI *Identify)
              (BioAPI_HANDLE   ModuleHandle,
              const BioAPI_FAR  *MaxFARRequested,
              const BioAPI_FRR  *MaxFRRRequested,
              const BioAPI_BOOL  *FARPrecedence,
              const BioAPI_IDENTIFY_POPULATION   *Population,
              BioAPI_BOOL  Binning,
              uint32  MaxNumberOfResults,
              uint32  *NumberOfResults,
              BioAPI_CANDIDATE_ARRAY_PTR  *Candidates,
              sint32  Timeout,
              BioAPI_BIR_HANDLE_PTR  AuditData);
       BioAPI_RETURN (BioAPI *Import)
              (BioAPI_HANDLE   ModuleHandle,
              const BioAPI_DATA  *InputData,
              BioAPI_BIR_BIOMETRIC_DATA_FORMAT   InputFormat,
              BioAPI_BIR_PURPOSE  Purpose,
              BioAPI_BIR_HANDLE_PTR  ConstructedBIR);
       BioAPI_RETURN (BioAPI *SetPowerMode)
              (BioAPI_HANDLE   ModuleHandle,
              BioAPI_POWER_MODE  PowerMode);
       BioAPI_RETURN (BioAPI *DbOpen)
              (BioAPI_HANDLE   ModuleHandle,
              const uint8  *DbName,
              BioAPI_DB_ACCESS_TYPE  AccessRequest,
              BioAPI_DB_HANDLE_PTR   DbHandle,
              BioAPI_DB_CURSOR_PTR   Cursor);
       BioAPI_RETURN (BioAPI *DbClose)
              (BioAPI_HANDLE   ModuleHandle,
              BioAPI_DB_HANDLE_PTR   DbHandle);
       BioAPI_RETURN (BioAPI *DbCreate)
              (BioAPI_HANDLE   ModuleHandle,
              const uint8  *DbName,
BioAPI_DB_ACCESS_TYPE  AccessRequest,BioAPI_DB_HANDLE_PTR  DbHandle);
       BioAPI_RETURN (BioAPI *DbDelete)
              (BioAPI_HANDLE   ModuleHandle,
              const uint8  *DbName);
       BioAPI_RETURN (BioAPI *DbSetCursor)
              (BioAPI_HANDLE   ModuleHandle,
              BioAPI_DB_HANDLE   DbHandle,
              const BioAPI_UUID  *KeyValue,
              BioAPI_DB_CURSOR_PTR   Cursor);
       BioAPI_RETURN (BioAPI *DbFreeCursor)
              (BioAPI_HANDLE   ModuleHandle,
              BioAPI_DB_CURSOR_PTR    Cursor);
       BioAPI_RETURN (BioAPI *DbStoreBIR)
              (BioAPI_HANDLE   ModuleHandle,
              const BioAPI_INPUT_BIR   *BIRToStore,
              BioAPI_DB_HANDLE   DbHandle,
              BioAPI_UUID_PTR   KeyValue,
              BioAPI_UUID_PTR  Uuid);
       BioAPI_RETURN (BioAPI *DbGetBIR)
              (BioAPI_HANDLE   ModuleHandle,
              BioAPI_DB_HANDLE   DbHandle,
              const BioAPI_UUID  *KeyValue,
              BioAPI_BIR_HANDLE_PTR   RetrievedBIR,
              BioAPI_UUID_PTR  Uuid,
              BioAPI_DB_CURSOR_PTR  Cursor);
       BioAPI_RETURN (BioAPI *DbGetNextBIR)
              (BioAPI_HANDLE   ModuleHandle,
              BioAPI_DB_CURSOR_PTR   Cursor,
              BioAPI_BIR_HANDLE_PTR   RetrievedBIR,
              BioAPI_UUID_PTR   Uuid);
       BioAPI_RETURN (BioAPI *DbQueryBIR)
              (BioAPI_HANDLE   ModuleHandle,
              BioAPI_DB_HANDLE   DbHandle,
              const BioAPI_INPUT_BIR   *BIRToQuery,
              BioAPI_UUID_PTR   KeyValue,
              BioAPI_UUID_PTR   Uuid);
       BioAPI_RETURN (BioAPI *DbDeleteBIR)
              (BioAPI_HANDLE   ModuleHandle,
              BioAPI_DB_HANDLE   DbHandle,
              const BioAPI_UUID  *KeyValue);
} BioAPI_BSP_FUNCS, *BioAPI_BSP_FUNCS_PTR;
```

3.2.4 BioAPI_UPCALLS

This structure is used by BioAPI to provide memory management functions to biometric service modules. A BSP can invoke these functions at anytime during the life cycle of an attach-session.

```
typedef void * (BioAPI *BioAPI_UPCALLS_MALLOC)
       (BioAPI_HANDLE ModuleHandle,
       uint32 Size);

typedef void (BioAPI *BioAPI_UPCALLS_FREE)
       (BioAPI_HANDLE ModuleHandle,
       void *Memblock);

typedef void * (BioAPI *BioAPI_UPCALLS_REALLOC)
       (BioAPI_HANDLE ModuleHandle,
       void *Memblock,
       uint32 Size);

typedef void * (BioAPI *BioAPI_UPCALLS_CALLOC)
       (BioAPI_HANDLE ModuleHandle,
       uint32 Num,
       uint32 Size);

typedef struct bioapi_upcalls {
       BioAPI_UPCALLS_MALLOC Malloc_func;
       BioAPI_UPCALLS_FREE Free_func;
       BioAPI_UPCALLS_REALLOC Realloc_func;
       BioAPI_UPCALLS_CALLOC Calloc_func;
} BioAPI_UPCALLS, *BioAPI_UPCALLS_PTR;
```

Definition

Malloc_func - The application-provided function for allocating memory in the application's memory space.

Free_func - The application-provided function for freeing memory allocated in the application's memory space using malloc_func.

Realloc_func - The application-provided function for re-allocating memory in the application's memory space that was previously allocated using malloc_func.

Calloc_func - The application-provided function for allocating a specified number of memory units in the application's memory space.

3.3 Service Provider Operations

3.3.1 Module Management Operations

3.3.1.1 BioSPI_ModuleLoad

BioAPI_RETURN BioAPI BioSPI_ModuleLoad

(const void * Reserved,
const BioAPI_UUID *BSPUuid,
BioSPI_ModuleEventHandler BioAPINotifyCallback,
void* BioAPINotifyCallbackCtx);

This function completes the module initialization process between BioAPI and the biometric service module.

The *BSPUuid* identifies the invoked module and should be used by the module to locate its entry in the module directory.

The BioAPINotifyCallback and BioAPINotifyCallbackCtx define a callback and callback context respectively. The module must retain this information for later use. The module should use the callback to notify BioAPI of module events of type BioAPI_MODULE_EVENT in any ongoing, attached sessions.

Parameters

Reserved (input) - A reserved input; should be set to NULL.

BSPUuid (input) - The BioAPI_UUID of the invoked service provider module. Used to locate the module's directory entry.

BioAPINotifyCallback (input) - A function pointer for the BioAPI event handler that manages events of type BioAPI_MODULE_EVENT.

BioAPINotifyCallbackCtx (input) - The context to be returned to BioAPI as input on each callback to the event handler defined by BioAPINotifyCallback.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

See BioAPI_ModuleLoad

3.3.1.2 BioSPI_ModuleUnload

BioAPI_RETURN BioAPI BioSPI_ModuleUnload

```
    (const void * Reserved,
    const BioAPI_UUID *BSPUuid,
    BioSPI_ModuleEventHandler BioAPINotifyCallback,
    void* BioAPINotifyCallbackCtx);
```

This function disables events and de-registers the BioAPI event-notification function. The biometric service module may perform cleanup operations, reversing the initialization performed in *BioSPI_ModuleLoad*.

Parameters

Reserved (input) - A reserved input; should be set to NULL.

BSPUuid (input) - The BioAPI_UUID of the invoked service provider module.

BioAPINotifyCallback (input) - A function pointer for the BioAPI event handler that manages events of type BioAPI_MODULE_EVENT.

BioAPINotifyCallbackCtx (input) - The context to be returned to BioAPI as input on each callback to the event handler defined by BioAPINotifyCallback.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

See BioAPI_ModuleUnload

3.3.1.3 BioSPI_ModuleAttach

BioAPI_RETURN BioAPI BioSPI_ModuleAttach

```
        (const BioAPI_UUID *BSPUuid,
        const BioAPI_VERSION  *Version,
        BioAPI_DEVICE_ID DeviceID,
        uint32 Reserved1,
        uint32 Reserved2,
        BioAPI_HANDLE ModuleHandle,
        uint32  Reserved3,
        const void * Reserved4,
        const void * Reserved5,
        const void * Reserved6,
        const BioAPI_UPCALLS  *Upcalls,
        BioAPI_MODULE_FUNCS_PTR *FuncTbl);
```

This function is invoked by BioAPI once for each invocation of BioAPI_ModuleAttach specifying the module identified by *BSPUuid*.

The service module must verify compatibility with the system version level specified by *Version*. If the version is not compatible, then this function fails. The service module should perform all initializations required to support the new attached session and should return a function table for the SPI entry points that can be invoked by BioAPI in response to API invocations. BioAPI uses this function table to dispatch requests on for the attach session created by this function. Each attach session has its own function table.

Parameters

BSPUuid (input) - The BioAPI_UUID of the invoked service provider module.

Version (input) - The major and minor version number of the required level of BSP services and features. The BSP must determine whether its services are compatible with the required version.

DeviceID (input) - The identifier for the device to use with this module. If only one device is provided by the module, then DeviceID can be zero.

Reserved1 (input) - A reserved input; should be set to 0.

Reserved2 (input) - A reserved input; should be set to 0.

ModuleHandle (input) - The BioAPI_HANDLE value assigned by BioAPI and associated with the attach session being created by this function.

Reserved3 (input) - A reserved input; should be set to 0.

Reserved4 (input) - A reserved input; should be set to NULL.

Reserved5 (input) - A reserved input; should be set to NULL.

Reserved6 (input) - A reserved input; should be set to NULL.

Upcalls (input) - A set of function pointers the service module must use to obtain selected BioAPI services and to manage application memory. The memory management functions are provided by the application invoking BioAPI_ModuleAttach. BioAPI forwards these function pointers with BioAPI service function pointers to the module.

FuncTbl (output) - A BioAPI_MODULE_FUNCS table containing pointers to the service module functions the Caller can use. BioAPI uses this table to proxy calls from an application caller to the biometric service module.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

See BioAPI_ModuleAttach

3.3.1.4 BioSPI_ModuleDetach

BioAPI_RETURN BioAPI BioSPI_ModuleDetach
(BioAPI_HANDLE ModuleHandle);

This function is invoked by BioAPI once for each invocation of BioAPI_ModuleDetach specifying the attach-session identified by *ModuleHandle*. The function entry point for *BioSPI_ModuleDetach* is included in the module function table BioAPI_MODULE_FUNCS returned to BioAPI as output of a successful *BioSPI_ModuleAttach*.

The service module must perform all cleanup operations associated with the specified attach handle.

Parameters

ModuleHandle (input) - The BioAPI_HANDLE value associated with the attach session being terminated by this function.

Return Value

A BioAPI_RETURN value indicating success or specifying a particular error condition. The value BioAPI_OK indicates success. All other values represent an error condition.

Errors

See BioAPI_ModuleDetach

3.3.2 Handle Operations

3.3.2.1 BioSPI_FreeBIRHandle

BioAPI_RETURN BioAPI BioSPI_FreeBIRHandle
(BioAPI_HANDLE ModuleHandle,
BioAPI_BIR_HANDLE Handle);

3.3.2.2 BioSPI_GetBIRFromHandle

BioAPI_RETURN BioAPI BioSPI_GetBIRFromHandle
(BioAPI_HANDLE ModuleHandle,
BioAPI_BIR_HANDLE Handle,
BioAPI_BIR_PTR BIR);

3.3.2.3 BioSPI_GetHeaderFromHandle

BioAPI_RETURN BioAPI BioSPI_GetHeaderFromHandle
(BioAPI_HANDLE ModuleHandle,
BioAPI_BIR_HANDLE Handle,
BioAPI_BIR_HEADER_PTR Header);

3.3.3 Callback and Event Operations

3.3.3.1 BioSPI_EnableEvents

BioAPI_RETURN BioAPI BioSPI_EnableEvents
(BioAPI_HANDLE ModuleHandle,
BioAPI_MODULE_EVENT_MASK *Events);

3.3.3.2 BioSPI_SetGUICallbacks

BioAPI_RETURN BioAPI BioSPI_SetGUICallbacks
(BioAPI_HANDLE ModuleHandle,
BioAPI_GUI_STREAMING_CALLBACK GuiStreamingCallback,
void *GuiStreamingCallbackCtx,
BioAPI_GUI_STATE_CALLBACK GuiStateCallback,
void *GuiStateCallbackCtx);

3.3.3.3 BioSPI_SetStreamCallback

BioAPI_RETURN BioAPI BioSPI_SetStreamCallback
(BioAPI_HANDLE ModuleHandle,
BioAPI_STREAM_CALLBACK StreamCallback,
void *StreamCallbackCtx);

3.3.3.4 BioSPI_StreamInputOutput

BioAPI_RETURN BioAPI BioSPI_StreamInputOutput
(BioAPI_HANDLE ModuleHandle,
BioAPI_DATA_PTR InMessage,
BioAPI_DATA_PTR OutMessage);

3.3.4 Biometric Operations

3.3.4.1 BioSPI_Capture

```
BioAPI_RETURN BioAPI BioSPI_Capture
        (BioAPI_HANDLE   ModuleHandle,
        BioAPI_BIR_PURPOSE   Purpose,
        BioAPI_BIR_HANDLE_PTR  CapturedBIR,
        sint32  Timeout,
        BioAPI_BIR_HANDLE_PTR  AuditData);
```

3.3.4.2 BioSPI_CreateTemplate

```
BioAPI_RETURN BioAPI BioSPI_CreateTemplate
        (BioAPI_HANDLE   ModuleHandle,
        const BioAPI_INPUT_BIR  *CapturedBIR,
        const BioAPI_INPUT_BIR   *StoredTemplate,
        BioAPI_BIR_HANDLE_PTR  NewTemplate,
        const BioAPI_DATA   *Payload);
```

3.3.4.3 BioSPI_Process

```
BioAPI_RETURN BioAPI BioSPI_Process
        (BioAPI_HANDLE   ModuleHandle,
        const BioAPI_INPUT_BIR  *CapturedBIR,
        BioAPI_BIR_HANDLE_PTR  ProcessedBIR);
```

3.3.4.4 BioSPI_VerifyMatch

```
BioAPI_RETURN BioAPI BioSPI_VerifyMatch
        (BioAPI_HANDLE   ModuleHandle,
        const BioAPI_FAR  *MaxFARRequested,
        const BioAPI_FRR  *MaxFRRRequested,
        const BioAPI_BOOL  *FARPrecedence,
        const BioAPI_INPUT_BIR  *ProcessedBIR,
        const BioAPI_INPUT_BIR  *StoredTemplate,
        BioAPI_BIR_HANDLE  *AdaptedBIR,
        BioAPI_BOOL  *Result,
        BioAPI_FAR_PTR   FARAchieved,
        BioAPI_FRR_PTR   FRRAchieved,
        BioAPI_DATA_PTR  Payload);
```

3.3.4.5 BioSPI_IdentifyMatch

```
BioAPI_RETURN BioAPI BioSPI_IdentifyMatch
        (BioAPI_HANDLE   ModuleHandle,
        const BioAPI_FAR  *MaxFARRequested,
        const BioAPI_FRR  *MaxFRRRequested,
        const BioAPI_BOOL  *FARPrecedence,
        const BioAPI_INPUT_BIR  *ProcessedBIR,
        const BioAPI_IDENTIFY_POPULATION   *Population,
        BioAPI_BOOL  Binning,
        uint32  MaxNumberOfResults,
        uint32  *NumberOfResults,
        BioAPI_CANDIDATE_ARRAY_PTR  *Candidates,
        sint32 Timeout);
```

3.3.4.6 BioSPI_Enroll

```
BioAPI_RETURN BioAPI BioSPI_Enroll
        (BioAPI_HANDLE   ModuleHandle,
        BioAPI_BIR_PURPOSE  Purpose,
        const BioAPI_INPUT_BIR   *StoredTemplate,
        BioAPI_BIR_HANDLE_PTR NewTemplate,
        const BioAPI_DATA   *Payload,
        sint32 Timeout
        BioAPI_BIR_HANDLE_PTR  AuditData);
```

3.3.4.7 BioSPI_Verify

```
BioAPI_RETURN BioAPI BioSPI_Verify
        (BioAPI_HANDLE   ModuleHandle,
        const BioAPI_FAR  *MaxFARRequested,
        const BioAPI_FRR  *MaxFRRRequested,
        const BioAPI_BOOL  *FARPrecedence,
        const BioAPI_INPUT_BIR  *StoredTemplate,
        BioAPI_BIR_HANDLE_PTR  AdaptedBIR,
        BioAPI_BOOL *Result,
        BioAPI_FAR_PTR   FARAchieved,
        BioAPI_FRR_PTR   FRRAchieved,
        BioAPI_DATA_PTR  Payload,
        sint32 Timeout,
        BioAPI_BIR_HANDLE_PTR  AuditData);
```

3.3.4.8 BioSPI_Identify

```
BioAPI_RETURN BioAPI BioSPI_Identify
        (BioAPI_HANDLE   ModuleHandle,
        const BioAPI_FAR  *MaxFARRequested,
        const BioAPI_FRR  *MaxFRRRequested,
        const BioAPI_BOOL  *FARPrecedence,
        const BioAPI_IDENTIFY_POPULATION   *Population,
        BioAPI_BOOL  Binning,
        uint32  MaxNumberOfResults,
        uint32  *NumberOfResults,
        BioAPI_CANDIDATE_ARRAY_PTR  *Candidates,
        sint32  Timeout,
        BioAPI_BIR_HANDLE_PTR  AuditData);
```

3.3.4.9 BioSPI_Import

BioAPI_RETURN BioAPI BioSPI_Import

```
(BioAPI_HANDLE   ModuleHandle,
const BioAPI_DATA  *InputData,
BioAPI_BIR_BIOMETRIC_DATA_FORMAT   InputFormat,
BioAPI_BIR_PURPOSE  Purpose,
BioAPI_BIR_HANDLE_PTR  ConstructedBIR);
```

3.3.4.10 BioSPI_SetPowerMode

BioAPI_RETURN BioAPI BioSPI_SetPowerMode

```
(BioAPI_HANDLE   ModuleHandle,
BioAPI_POWER_MODE  PowerMode);
```

3.3.5 Database Operations

3.3.5.1 BioSPI_DbOpen

```
BioAPI_RETURN BioAPI BioSPI_DbOpen
        (BioAPI_HANDLE   ModuleHandle,
        const uint8  *DbName,
        BioAPI_DB_ACCESS_TYPE  AccessRequest,
        BioAPI_DB_HANDLE_PTR   DbHandle,
        BioAPI_DB_CURSOR_PTR   Cursor);
```

3.3.5.2 BioSPI_DbClose

```
BioAPI_RETURN BioAPI BioSPI_DbClose
        (BioAPI_HANDLE   ModuleHandle,
        BioAPI_DB_HANDLE_PTR   DbHandle);
```

3.3.5.3 BioSPI_DbCreate

```
BioAPI_RETURN BioAPI BioSPI_DbCreate
        (BioAPI_HANDLE   ModuleHandle,
        const uint8  *DbName,
        BioAPI_DB_ACCESS_TYPE  AccessRequest,
        BioAPI_DB_HANDLE_PTR  DbHandle);
```

3.3.5.4 BioSPI_DbDelete

```
BioAPI_RETURN BioAPI BioSPI_DbDelete
        (BioAPI_HANDLE   ModuleHandle,
        const uint8  *DbName);
```

3.3.5.5 BioSPI_DbSetCursor

```
BioAPI_RETURN BioAPI BioSPI_DbSetCursor
        (BioAPI_HANDLE   ModuleHandle,
        BioAPI_DB_HANDLE   DbHandle,
        const BioAPI_UUID  *KeyValue,
        BioAPI_DB_CURSOR_PTR   Cursor);
```

3.3.5.6 BioSPI_DbFreeCursor

```
BioAPI_RETURN BioAPI BioSPI_DbFreeCursor
```

(BioAPI_HANDLE ModuleHandle,
BioAPI_DB_CURSOR_PTR Cursor);

3.3.5.7 BioSPI_DbStoreBIR

BioAPI_RETURN BioAPI BioSPI_DbStoreBIR
(BioAPI_HANDLE ModuleHandle,
const BioAPI_INPUT_BIR *BIRToStore,
BioAPI_DB_HANDLE DbHandle,
BioAPI_UUID_PTR KeyValue,
BioAPI_UUID_PTR Uuid);

3.3.5.8 BioSPI_DbGetBIR

BioAPI_RETURN BioAPI BioSPI_DbGetBIR
(BioAPI_HANDLE ModuleHandle,
BioAPI_DB_HANDLE DbHandle,
const BioAPI_UUID *KeyValue,
BioAPI_BIR_HANDLE_PTR RetrievedBIR,
BioAPI_UUID_PTR Uuid,
BioAPI_DB_CURSOR_PTR Cursor);

3.3.5.9 BioSPI_DbGetNextBIR

BioAPI_RETURN BioAPI BioSPI_DbGetNextBIR
(BioAPI_HANDLE ModuleHandle,
BioAPI_DB_CURSOR_PTR Cursor,
BioAPI_BIR_HANDLE_PTR RetrievedBIR,
BioAPI_UUID_PTR Uuid);

3.3.5.10 BioSPI_DbQueryBIR

BioAPI_RETURN BioAPI BioSPI_DbQueryBIR
(BioAPI_HANDLE ModuleHandle,
BioAPI_DB_HANDLE DbHandle,
const BioAPI_INPUT_BIR *BIRToQuery,
BioAPI_UUID_PTR KeyValue,
BioAPI_UUID_PTR Uuid);

3.3.5.11 BioSPI_DbDeleteBIR

BioAPI_RETURN BioAPI BioSPI_DbDeleteBIR
(BioAPI_HANDLE ModuleHandle,
BioAPI_DB_HANDLE DbHandle,
const BioAPI_UUID *KeyValue);

4 Appendix A: Conformance

Conformance to the BioAPI specification falls into the following categories:

- A BioAPI compliant application
- A BioAPI compliant Biometric Service Provider (BSP)

Conformance requirements for applications and BSPs are defined in the following sections.

4.1 BioAPI Compliant Application

To claim compliance to the BioAPI specification, a software application must, for each BioAPI function call utilized, perform that operation consistent with the specification. That is, all input parameters must be present and valid.

There is no minimum set of functions that must be called.

4.2 BioAPI Compliant BSPs

To claim compliance to the BioAPI specification, BSPs must implement mandatory functions for their category, defined below, in accordance with the SPI defined in chapter 3. BSPs are categorized as either a Verification or Identification BSP. In addition, a BSP may be categorized as a monolithic or explicit client/server BSP. A monolithic BSP is fully loaded on a single platform. A client/server BSP has components installed on two or more platforms. These components (client and server) may communicate with each other using the streaming callbacks provided by the client/server application.

BSPs must accept all valid input parameters and return valid outputs. Optional capabilities and returns are not required to claim conformance; however, any optional functions or parameters that are implemented must be implemented in accordance with the specification requirements.

Additionally, all BSPs must provide all required module registry entries. Entries to the module registry must be performed upon BSP installation.

BSPs must possess a valid and unique UUID, which may be self-generated.

Biometric data generated by the BSP must conform to the data structures defined in section 2.1 and 3.2. BSPs may only return BioAPI_BIR data containing a registered FormatOwner with an associated valid FormatType.

BSPs must perform error handling as defined in Section 2.3.

All BSPs must support all Module Management (section 3.3) and Handle (3.5) operations. Database operations (section 3.7) are optional.

The following table is a summary of BSP conformance requirements by type. Details are provided in the following sections.

Function	**Verification BSP**	**Identification BSP**
Module Management Functions		
BioSPI_ModuleLoad	X	X
BioSPI_ModuleUnload	X	X
BioSPI_ModuleAttach	X	X
BioSPI_ModuleDetach	X	X
Handle Functions		
BioSPI_FreeBIRHandle	X	X
BioSPI_GetBIRFromHandle	X	X
BioSPI_GetHeaderFromHandle	X	X
Callback and Event Functions		
BioSPI_EnableEvents		
BioSPI_SetGUICallbacks		
BioSPI_SetStreamCallback		
BioSPI_StreamInputOutput		
Biometric Functions		
BioSPI_Capture		
BioSPI_CreateTemplate		
BioSPI_Process		
BioSPI_VerifyMatch		
BioSPI_IdentifyMatch		
BioSPI_Enroll	X	X
BioSPI_Verify	X	X
BioSPI_Identify		X
BioSPI_Import		
BioSPI_SetPowerMode		
Database Functions		
BioSPI_DbOpen		
BioSPI_DbClose		
BioSPI_DbCreate		
BioSPI_DbDelete		
BioSPI_DbSetCursor		
BioSPI_DbFreeCursor		
BioSPI_DbStoreBIR		
BioSPI_DbGetBIR		
BioSPI_DbGetNextBIR		
BioSPI_DbQueryBIR		
BioSPI_DbDeleteBIR		

4.2.1 BioAPI Compliant Verification BSPs

Verification BSPs are those which are capable of performing 1:1 matching (or authentication), but not 1:N identification matching. [Note that 1:few matching may be supported as a series of 1:1 calls.] A BioAPI compliant Verification BSP must support the following biometric functions:

- BioAPI_Enroll
- BioAPI_Verify

BioAPI_Verify. Client/server implementations of this function are optional.

Only the nearest, better supported RequestedFAR must be supported; however, the BSP must return that supported value (ActualFAR).

Return of payload is required only if one is contained within the input StoredTemplate and if the score sufficiently exceeds the ActualFAR.

Return of AdaptedBIR is optional. Return of raw data (AuditData) is optional.

As a default, all BSPs must provide any GUI associated with the capture portion of the Verify operation. However, support for application control of the GUI is optional.

BioAPI_Enroll. Only 'Purpose' flags indicating 'Enroll for Verification' must be accepted. If another purpose is set, an error condition must be set as a BioAPI_RETURN. Acceptance of 'Payload' is optional.

Client/server implementations of this function are optional

4.2.2 BioAPI Compliant Identification BSPs

Identification BSPs are those, which are capable of performing both 1:N identification matching as well as 1:1 matching (or authentication). A BioAPI compliant Identification BSP must support the following biometric functions:

- BioAPI_Enroll
- BioAPI_Verify
- BioAPI_Identify

BioAPI_Verify. Client/server implementations of this function are optional.

Only the nearest, better supported RequestedFAR must be supported; however, the BSP must return that supported value (ActualFAR).

Return of payload is required only if one is contained within the input StoredTemplate and if the score sufficiently exceeds the ActualFAR.

Return of AdaptedBIR is optional. Return of raw data (AuditData) is optional.

As a default, all BSPs must provide any GUI associated with the capture portion of the Verify operation. However, support for application control of the GUI is optional.

BioAPI_Identify. Client/server implementations of this function are optional

Only the nearest, better supported RequestedFAR must be supported; however, the BSP must return that supported value (ActualFAR).

Support of binning is optional.

Return of matching Candidates is required; however, the BSP may return values for the ActualFAR field as next nearest step/increment .

As a default, all BSPs must provide any GUI associated with the capture portion of the Identify operation. However, support for application control of the GUI is optional.

BioAPI_Enroll. 'Purpose' flags indicating either 'Enroll for Verification' or 'Enroll for Identification' must be accepted. Acceptance of 'Payload' is optional

Client/server implementations of this function are optional

4.2.3 BioAPI Compliant Client/Server BSPs

A BSP instantiation may be wholly installable and loaded onto a single platform, which is typical. This is referred to as a "Local" BSP. However, a BSP may be instantiated and loadable as separate client/server components which operate and communicate together. This is referred to as a "Distributed" BSP.

If a BSP is constructed such that it is installed and operates in a distributed fashion across a network or other communications channel, with direct communication between the distributed BSP components, it is considered to be a Distributed (Client/Server) BSP. This does not include a Local BSP that can be installed in whole on both a client and a server platform, where communication between the client and the server is always at the application layer and the two instantiations of the BSP do not communicate with each other directly.

BSPs must post to the module registry whether they support Local operation, Distributed operation, or both.

Local BSPs must support all functions required by their category. These functions are both called locally (by an application running on the same platform) and executed locally (on the platform on which they are called and on which the BSP is installed). Streaming callbacks are not supported or used by local BSPs.

In order to be BioAPI compliant, Client/Server BSPs must support all functions for their category (Verification or Identification), defined above, when initiated from an application on either side. That is, all functions are supported by both the client and server components. However, although called locally, some functions may be executed remotely. These are identified below.

Executed Locally	**Executed Locally or Remotely**
BioSPI_Capture *	BioSPI_Enroll
BioSPI_CreateTemplate *	BioSPI_Verify
BioSPI_Process *	BioSPI_Identify
BioSPI_VerifyMatch *	
BioSPI_IdentifyMatch *	
BioSPI_Import *	

*Optional function

For functions that can be executed either locally or remotely, the application may dictate which mode the operation is to be performed in, via a parameter in the function call.

All module management and handle operations must be callable and executable locally.

Additionally, Distributed BSPs must support direct communication between 'partner' components by "tunneling" through the application layer using the streaming callback capability (via the BioSPI_SetStreamCallback and BioSPI_StreamInputOutput functions).

4.2.4 Optional Capabilities

The following capabilities are considered optional in terms of BSP support and compliance. Note that:

- If implemented, optional capabilities must conform to specification definitions
- BSPs are required to post to the module registry whether or not each option is supported
- BSP documentation must include a table that identifies which options are supported and which options are not supported.

4.2.4.1 Optional Functions

Primitive functions.

Although it was originally intended that the primitive functions (BioSPI_Capture, BioSPI_CreateTemplate, BioSPI_Process, BioSPI_VerifyMatch, and BioSPI_IdentifyMatch) would be mandatory, it was decided that this would place undue burden on manufacturers of "self-contained devices" in which the biometric processing/matching is performed within the device itself. Therefore, these functions have not been specified as required for BSP's to be considered "BioAPI compliant". However, it is highly recommended that, if supported by the underlying technology, these functions be included in the BSP.

BioAPI_Capture. If this function is supported, Verification BSPs need only accept ‘Purpose’ flags indicating ‘Verification’ or ‘Enroll for Verification’. If another purpose is set by the application, an error condition may be set as a BioAPI_RETURN. Similarly, this function need only return ‘CapturedBIR’ with the Purpose mask set to Verification or Enroll for Verification.

If this function is supported, Identification BSPs must accept all possible ‘Purpose’ flags (except ‘Audit’), even if there is no difference in the content or format of the returned data.

Return of raw data (AuditData) is optional.

As a default, all BSPs must provide any GUI associated with the Capture operation. However, support for application control of the GUI is optional.

BioAPI_CreateTemplate. If this function is supported, Verification BSPs need only accept input CapturedBIRs with the Purpose set to ‘Enroll for Verification’. If another purpose is set, an error condition may be set as a BioAPI_RETURN.

If this function is supported, Identification BSPs must accept input CapturedBIRs with both Enroll purposes.

Acceptance of ‘Payload’ is optional. Adaptation of an existing template is optional.

BioAPI_Process. If this function is supported, Verification BSPs need only accept input CapturedBIRs with the Purpose set to Verification. If another purpose is set, an error condition may be set as a BioAPI_RETURN.

If this function is supported, Identification BSPs must accept input CapturedBIRs with all possible 'Purpose flags (except ‘Audit’), even if there is no difference in the content or format of the returned data.

BioAPI_VerifyMatch. If this function is supported, only input BIRs (ProcessedBIR) with the 'Purpose' mask including a value of 'Verification', and (StoredTemplate) with the 'Purpose' mask including a value of 'Enroll for Verification' must be accepted. If another purpose is set, an error condition may be set as a BioAPI_RETURN.

Only the nearest, better supported RequestedFAR must be supported; however, the BSP must return that supported value (ActualFAR).

Return of payload is required only if one is contained within the input StoredTemplate and if the score is better than the ActualFAR (the BSP must post minimum FAR required to return payload in the module registry).

Return of AdaptedBIR is optional.

BioAPI_IdentifyMatch. May be supported by Identification BSPs for BIRs whose purpose is 'identify'.

Only the nearest, better supported RequestedFAR must be supported; however, the BSP must return that supported value (ActualFAR).

Support of binning is optional.

Return of matching Candidates is required; however, the BSP may return values for the Score field as next nearest step/increment

Database operations. BSPs are not required to provide an internal (or internally controlled) database. If one is provided, however, ALL database functions must be provided in order to access and maintain it. All provided database functions must conform to the definitions.

BioAPI_Import. This function, as a whole, is optional. If it is provided, the module registry must so reflect, and it must be implemented as defined.

Verification BSPs are only required to process imported data if the 'Purpose' flag includes 'Enroll for Verification'. Identification only BSPs are only required to process imported data if the 'Purpose' flag includes 'Enroll for Identification'.

BioAPI_SetPowerMode. This function, as a whole, is optional. If it is provided, the module registry must so reflect, and it must be implemented as defined.

NOTE: Although the primitive functions (BioSPI_Capture, BioSPI_Process, BioSPI_VerifyMatch, and BioSPI_IdentifyMatch) are not required for conformance, it is highly recommended that they be provided, if supported by the underlying technology.

4.2.4.2 Optional Sub-functions

The following table identifies optional capabilities, each of which the BSP must declare as being supported or not supported (in the module registry and in the BSP documentation).

Capability	Supported	Not Supported
Return of raw/audit data		
Return of quality		
Application-controlled GUI		
GUI streaming callbacks		
Detection of source presence		
Payload carry		

BIR signing		
BIR encryption		
Return of FRR		
Model adaptation		
Binning		
Client/server communications		
Supports self-contained device		

Return of raw data. Functions involving the capture of biometric data from a sensor may optionally support the return of this raw data for purposes of display or audit. If supported, the output parameter 'AuditData' will contain a pointer to this data. If not supported, the BSP will return a value of –1.

Return of Quality. Upon the new capture of biometric data from a sensor, the BSP may calculate a relative quality value associated with this data, which it will include in the header of the returned CapturedBIR (and the optional AuditData). If supported, this header field will be filled with a positive value between 1 and 100. If not supported, this field will be set to –2. This would occur during BioAPI_Capture and BioAPI_Enroll.

Similarly, when a BIR is processed, another quality calculation may be performed and the quality value included in the header of the ProcessedBIR (and the optional AdaptedBIR). This would occur during BioAPI_CreateTemplate, BioAPI_Process, BioAPI_Verify, BioAPI_VerifyMatch, BioAPI_Enroll, and BioAPI_Import (ConstructedBIR) operations.

The BSP must post to the module registry whether or not it supports the calculation of quality measurements for each type of BIR – raw, intermediate, and processed.

Application-controlled GUI. The BSP supports the necessary callbacks to allow the application to control the "look-and-feel" of the GUI.

All BioAPI compliant BSPs must provide, as a default, the capability to display user interface information (assuming such a display is present and required by the biometric technology). All BSP supplied GUIs must include an operator abort/cancel mechanism.

Optionally, the BSP may also support application controlled GUI. In this case, the BSP must support the BioAPI_EnableEvents and BioAPI_SetGUICallbacks functions.

If application controlled GUI is supported, the BSP may additionally provide streaming data (e.g., for voice and face recognition). If streaming data is provided, the BSP must support the input parameters GuiStreamingCallback and GuiStreamingCallbackCtx.

All BSPs must post to the module registry what GUI functions/options it supports.

GUI streaming callbacks. The BSP provides GUI streaming data, and supports the GUI streaming callback.

Detection of source presence. The BSP can detect when there are samples available, and supports the source present event.

Payload carry. Some BSPs may optionally support the encapsulation of a 'Payload' within a processed BIR, and the subsequent release of that payload upon successful verification. The content of the payload is unrestricted, but may include secrets such as PINs or private keys associated with the user whose biometric data is contained within the BIR. Payload data is encapsulated during a BioAPI_Enroll or BioAPI_Process operation and is released as a result of a BioAPI_Verify or BioAPI_VerifyMatch operation. If supported, BSPs must post to the module registry the maximum payload size it can accommodate. A maximum size of '0' indicates payload carry is not supported. If input payloads exceed this size, an error should be generated. If payload carry is not supported, the output Payload is set to –1.

Security features. The BioAPI supports but does not require the implementation of security features. Additionally, it does not define features that do not cross the API boundary.

The BSP must post to the module registry whether or not it provides the following:

- BIR encryption
- BIR signing

Return of FRR. During matching operations, the application may request a specific FAR threshold and the BSP must return the nearest supported threshold. Optionally, the BSP may also return the estimated FRR associated with this supported FAR. This Corresponding FRR value is an optional return from the BioAPI_Verify, BioAPI_VerifyMatch, BioAPI_Identify, and BioAPI_IdentifyMatch functions. If supported, the BSP will return its best estimate of expected FRR. If not supported, the BSP will return a value of –2.

Template/Model adaptation. Some BSPs may optionally provide the capability to utilize newly captured biometric data to update a stored BIR. This may only be performed as a result of a successful BioAPI_Verify or BioAPI_VerifyMatch (i.e., one in which 'Result' = BioAPI_TRUE). This is performed in order to keep the enrolled BIR as fresh as possible, with the highest possible quality. The BSP makes the decision as to when and if the adaptation should be performed (based on such factors as quality, elapsed time, and significant differences). If the BSP does not support adaptation, the returned AdaptedBIR will be set to –2. If the BSP supports adaptation, but for some reason is not able or chooses not to perform the adaptation, then the return AdaptedBIR is set to –1. The BSP must post to the module registry its support for adaptation.

Binning. Identification BSPs may optionally support methods of limiting the population of a database to be searched in order to improve response time. This applies to Identification type BSPs only and occurs only during BioAPI_Identify and BioAPI_IdentifyMatch operations. Identification BSPs must post whether or not they support binning. Identification BSPs that do not support binning may ignore the input Binning on/off parameter. Note that no explicit API support is provided for setting or modifying the binning strategy other than for turning it on or off.

Client/Server Communication. The BSP supports the streaming callback to allow the client and server BSPs to communicate.

Self-contained Device. The supported device is self-contained. That is, at least the verification function is entirely contained within the device, and the device may contain a database.

4.3 Conformance Testing

A conformance test suite may be provided as a part of the BioAPI reference implementation. At this time, no authorized testing or certification authority has been identified.

4.4 Branding

No branding program is yet in place for BioAPI; however, such a program may be established in the future. Such a program would be expected to include some level of conformance testing with successful completion resulting in authorization to claim compliance to the BioAPI and to display a logo to that effect.

5 Appendix B: HA-API Compatibility

Originally, it was intended that the BioAPI Reference Implementation would include a HA-API adaptation layer to support the transition of existing biometric applications and BSPs that are compliant with the existing Human Authentication API (HA-API). However, due to the rapid adoption of BioAPI throughout the industry, it was determined that the interim adaptation layer was not required and therefore was not included in the Beta release of the BioAPI Reference Implementation.

However, the below table is provided to assist HA-API application developers to transition their application software to take advantage of the BioAPI.

BioAPI Function	HA-API Function
BioAPI_Init, **BioAPI_Terminate** **BioAPI_ModuleLoad,** **BioAPI_ModuleUnload**	None
BioAPI_RETURN BioAPI BioAPI_ModuleAttach (const BioAPI_UUID *BSPUuid, const BioAPI_VERSION *Version, const BioAPI_MEMORY_FUNCS *MemoryFuncs, BioAPI_DEVICE_ID DeviceID, uint32 Reserved1, uint32 Reserved2, uint32. Reserved3 BioAPI_FUNC_NAME_ADDR *FunctionTable, uint32 NumFunctionTable, const void * Reserved4, BioAPI_HANDLE_PTR NewModuleHandle);	**HBT GetBioTechnology** (**BUID** buidBioTechnology)
BioAPI_RETURN BioAPI BioAPI_ModuleDetach (BioAPI_HANDLE ModuleHandle);	**HAAPIERROR ReleaseBioTechnology** (**HBT** hbtBioTechnology)
BioAPI_RETURN BioAPI BioAPI_QueryDevice (BioAPI_HANDLE ModuleHandle, BioAPI_SERVICE_UID_PTR ServiceUID);	**HAAPIERROR EnumBioTechnology** (**LPBIOTECH** pBioTechEnum, **DWORD** cbBuf**,** **LPDWORD** pcbNeeded**,** **LPDWORD** pcReturned)
BioAPI_FreeBIRHandle, **BioAPI_GetBIRFromHandle,** **BioAPI_GetHeaderFromHandle,**	None
BioAPI_EnableEvents, **BioAPI_SetGUICallbacks** **BioAPI_SetStreamCallback,** **BioAPI_StreamInputOutput,**	None
BioAPI_RETURN BioAPI BioAPI_Capture (BioAPI_HANDLE ModuleHandle, BioAPI_BIR_PURPOSE Purpose, BioAPI_BIR_HANDLE_PTR CapturedBIR,	**HAAPIERROR HAAPICapture** (**HBT** hbtBioTechnology, **LPSCREENATTRIBS** lpScreenAttribs, **LPRAWBIODATA** lpRawBioData)

sint32 Timeout, BioAPI_BIR_HANDLE_PTR AuditData);	
BioAPI_RETURN BioAPI BioAPI_CreateTemplate (BioAPI_HANDLE ModuleHandle, BioAPI_BIR_PURPOSE Purpose, const BioAPI_INPUT_BIR *CapturedBIR, const BioAPI_INPUT_BIR *StoredTemplate, BioAPI_BIR_HANDLE_PTR NewTemplate, const BioAPI_DATA *Payload);	**HAAPIERROR HAAPIProcess** (**HBT** hbtBioTechnology, **LPRAWBIO** lpRawBioData, **LPBIR** lpBiometricIdRec
BioAPI_RETURN BioAPI BioAPI_Process (BioAPI_HANDLE ModuleHandle, BioAPI_BIR_PURPOSE Purpose, const BioAPI_INPUT_BIR *CapturedBIR, BioAPI_BIR_HANDLE_PTR ProcessedBIR);	**HAAPIERROR HAAPIProcess** (**HBT** hbtBioTechnology, **LPRAWBIO** lpRawBioData, **LPBIR** lpBiometricIdRec
BioAPI_RETURN BioAPI BioAPI_VerifyMatch (BioAPI_HANDLE ModuleHandle, const BioAPI_FAR *MaxFARRequested, const BioAPI_FRR *MaxFRRRequested, const BioAPI_BOOL *FARPrecedence, const BioAPI_INPUT_BIR *ProcessedBIR, const BioAPI_INPUT_BIR *StoredTemplate, BioAPI_BIR_HANDLE *AdaptedBIR, BioAPI_BOOL *Result, BioAPI_FAR_PTR FARAchieved, BioAPI_FRR_PTR FRRAchieved, BioAPI_DATA_PTR Payload);	**HAAPIERROR HAAPIVerify** (**HBT** hbtBioTechnology, **LPBIR** lpSampleBIR, **LPBIR** lpStoredBIR, **LPBOOL** lpbResponse)
BioAPI_IdentifyMatch	None
BioAPI_RETURN BioAPI BioAPI_Verify (BioAPI_HANDLE ModuleHandle, const BioAPI_FAR *MaxFARRequested, const BioAPI_FRR *MaxFRRRequested, const BioAPI_BOOL *FARPrecedence, const BioAPI_INPUT_BIR *StoredTemplate, BioAPI_BIR_HANDLE_PTR AdaptedBIR, BioAPI_BOOL *Result, BioAPI_FAR_PTR FARAchieved, BioAPI_FRR_PTR FRRAchieved, BioAPI_DATA_PTR Payload, sint32 Timeout, BioAPI_BIR_HANDLE_PTR AuditData);	**HAAPIERROR HAAPILiveVerify** (**HBT** hbtBioTechnology, **LPSCREENATTRIBS** lpScreenAttribs, **LPBIR**lpStoredBIR, **int** iTimeout, **LPBOOL** lpbResponse)
BioAPI_Identify	None
BioAPI_RETURN BioAPI BioAPI_Enroll (BioAPI_HANDLE ModuleHandle, BioAPI_BIR_PURPOSE Purpose, const BioAPI_INPUT_BIR *StoredTemplate, BioAPI_BIR_HANDLE_PTR NewTemplate, const BioAPI_DATA *Payload, sint32 Timeout BioAPI_BIR_HANDLE_PTR AuditData);	**HAAPIERROR HAAPIEnroll** (**HBT** hbtBioTechnology, **LPSCREENATTRIBS** lpScreenAttribs, **LPRAWBIODATA** lpRawBioData, **LPBIR** lpBioIdRec, **LPENROLLMENTPAGES** lpPages)
BioAPI_Import, **BioAPI_SetPowerMode**	None
BioAPI_DbOpen, **BioAPI_DbClose,** **BioAPI_DbCreate,** **BioAPI_DbDelete,**	None

BioAPI_DbSetCursor, **BioAPI_DbFreeCursor,** **BioAPI_DbStoreBIR,** **BioAPI_DbGetBIR,** **BioAPI_DbGetNextBIR,** **BioAPI_DbQueryBIR,** **BioAPI_DbDeleteBIR**	
None	**HAAPIERROR HAAPIFree** (**HBT** hbtBioTechnology, **LPVOID** lpData)
None	**HAAPIERROR HAAPIInformation** (**HBT** hbtBioTechnology, **long** lRequestedInfo, **LPVOID** lpData, **DWORD** cbSize)