INTERNATIONAL STANDARD

ISO/IEC 30106-1

First edition
2016-03-15

Information technology — Object oriented BioAPI —

Part 1: Architecture

Technologies de l'information — Objet orienté BioAPI —

Partie 1: Architecture

Reference number
ISO/IEC 30106-1:2016(E)

ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

Contents

Page

Foreword ..... **iv**
Introduction ..... **v**
1 Scope **1**
2 Normative references **1**
3 Terms and definitions **1**
4 Symbols and abbreviated terms **1**
5 Object Oriented BioAPI architecture **2**
5.1 Summary of BioAPI architecture 2
5.2 BioAPI compatibility requirements 4
5.3 Graphical User Interface (GUI) 4
5.4 Implementation guidelines 6
5.4.1 Basic concepts 6
5.4.2 BioAPI_Unit development 6
5.4.3 BFP development 6
5.4.4 BSP development 7
5.4.5 Framework and component registry 8
5.4.6 Application development 8
6 BioAPI CBEFF Patron Formats **8**
6.1 General 8
6.2 Simple BIR 8
6.3 Complex BIR 14
6.3.1 Structure 14
6.3.2 Child BIR 14
6.3.3 Parent BIR 14
7 Constants **17**
7.1 General 17
7.2 Biometric types 17
7.3 Biometric subtypes 17
7.4 Error codes 18
8 OO BioAPI UML structure **21**
8.1 General 21
8.2 Relationships among data structures 22
8.2.1 Class BIR 22
8.2.2 Class UnitSchema 23
8.2.3 Class BFPSchema 24
8.2.4 Class BSPSchema 24
8.2.5 Class FrameworkSchema 25
8.3 BioAPI_Unit structures 25
8.3.1 IArchive 25
8.3.2 IComparison 26
8.3.3 IProcessing 26
8.3.4 ISensor 27
8.4 BFP Structure 27
8.5 BSP Structure 28
8.6 Framework structure 28
8.7 Application related structure 29

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the WTO principles in the Technical Barriers to Trade (TBT) see the following URL: Foreword - Supplementary information

The committee responsible for this document is ISO/IEC JTC 1, *Information technology*, SC 37, *Biometrics*.

ISO/IEC 30106 consists of the following parts, under the general title *Information technology — BioAPI for object oriented programming languages*:

— *Part 1: Architecture*

— *Part 2: Java implementation*

— *Part 3: C# implementation*

Introduction

The existing versions of BioAPI (ANSI version-INCITS 358 and ISO/IEC 19784-1) specify an application programming interface expressed in the C language. The use of a portable language like C ensures the BioAPI is accessible across multiple computing platforms and application domains. BioAPI is an appropriate fit for applications written in C and is adequate for applications written in C++.

Unfortunately, a function-based language like C does not map easily to the object oriented domain where this issue may be answered with an object oriented (OO) version of BioAPI. As noted, the function-based nature of a C API does not map easily to the object oriented paradigm (i.e. languages such as C# and Java). In particular, the use of a C API from within an object oriented application is unnatural and requires programming constructs which introduce complexity to the development of an application. Development of a OO version of BioAPI aims to increase the productivity of software practitioners who who wish to use the BioAPI whilst remaining in the object oriented domain.

A standard object oriented version of BioAPI allows, in case of Java, BSPs that are intended for loading into a Java-based application server to perform verification and/or identification operations. In those application servers, use of the OO BioAPI is more natural when developing a framework and BSPs than the C version of BioAPI.

Another area in which a standard OO version of BioAPI would be useful is that of small computing devices based on an object oriented language (OOL), where (as on the large application servers mentioned above) an OO BioAPI framework and OO BSPs would fit better than their C counterparts.

This part of ISO/IEC 30106 is expected to have the following impact:

— enable creation of BioAPI applications by developers more comfortable with Object Oriented Languages;

— create a market of standard OO BSP components which target OO environments such as Java application servers, Java applets, small Java devices, .NET servers, .NET applications, web services;

— increase the level of adoption of BioAPI by decreasing the barrier of entry for OO developers. This includes providing access to C based BSPs (as if they were OO BSPs) through special versions of the BioAPI framework, bridging a standard OO BioAPI framework to a standard C BioAPI framework.

Information technology — Object oriented BioAPI —

Part 1: Architecture

1 Scope

This part of ISO/IEC 30106 specifies an architecture for a set of interfaces which define the OO BioAPI. Components defined in this part of ISO/IEC 30106 include a framework, Biometric Service Providers (BSPs), Biometric Function Providers (BFPs) and a component registry.

NOTE Each of these components have an equivalent component specified in ISO/IEC 19784-1 as the OO BioAPI is intended to be an OO interpretation of this part of ISO/IEC 30106.

For this reason, this part of ISO/IEC 30106 is conceptually equivalent to ISO/IEC 19784-1. Concepts present in this part of ISO/IEC 30106 (for example, BioAPI_Unit and component registry) have the same meaning as in ISO/IEC 19784-1. While the conceptual equivalence of this part of ISO/IEC 30106 will be maintained with ISO/IEC 19784-1, there are differences in the parameters passed between functions and the sequence of function calls. These differences exist to take advantage of the features provided by Object Oriented Programming Languages.

2 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 2382-37, *Information technology — Vocabulary — Part 37: Biometrics*

ISO/IEC 19784-1:2006, *Information technology — Biometric Application Programming Interface — Part 1: BioAPI Specification*

ISO/IEC 19785-1:2015, *Information technology — Common Biometric Exchange Formats Framework — Part 1: Data Element Specification*

ISO/IEC 19785-3:2015, *Information technology — Common Biometric Exchange Formats Framework — Part 3: Patron format specifications*

3 Terms and definitions

For the purposes of this document, the terms and definitions in ISO 2382-37, ISO/IEC 19784-1, and ISO/IEC 19785 (all parts) apply.

4 Symbols and abbreviated terms

For the purposes of this document, the symbols and abbreviated terms defined in ISO/IEC 19784-1, ISO/IEC 19785 (all parts) and the following apply.

API	Application Programming Interface
BDB	Biometric Data Block
BFP	Biometric (OO) Function Provider

BIR	Biometric Information Record
BSP	Biometric Service Provider
CBEFF	Common Biometric Exchange Formats Framework
FMR	False Match Rate
FPI	Function Provider Interface
GUI	Graphical User Interface
ID	Identity/Identification/Identifier
MOC	Match on Card
OOBioAPI	Object Oriented BioAPI
PID	Product ID
SB	Security Block
	NOTE This term and abbreviation is imported from ISO/IEC 19785-1.
SBH	Standard Biometric Header
	NOTE This term and abbreviation is imported from ISO/IEC 19785-1.
SPI	Service Provider Interface
UUID	Universally Unique Identifier

5 Object Oriented BioAPI architecture

5.1 Summary of BioAPI architecture

Object Oriented BioAPI shall be defined in a way that allows both structured development of applications and services, as well as interoperability between application and BSPs, and among BSPs. Therefore, the definition has to be done respecting the hierarchical structure of BioAPI (19784-1). In few words, an application shall be developed using an OO BioAPI that allows the instantiation of a BSP, which is based on the instantiation of one or several BioAPI_Units. The BSP can host more than one BioAPI_Unit of each category, and several units of each type can be used at any time, no presenting any kind of limitation on the units to be used in a BSP. This makes it necessary for all methods to provide the reference to the unit to be used in each of the operations.

An application is not permitted to directly access BioAPI_Units. Therefore, this part of ISO/IEC 30106 does not define a BioAPI_Unit interface, but only an object oriented hierarchical interface/class model that may ease the implementation of the BSP. The BSP shall inherit all public methods and data structures for each BioAPI_Unit the BSP encapsulates. It is the responsibility of the BSP's implementation to determine which functions of the BioAPI_Unit are offered to clients of the BSP. Note, an exception should be returned for any BSP interface method that is not implemented in the BSP. This is represented in Figure 1.

Programming an application, which interacts directly with a BSP, introduces practical considerations for the application designer. One such consideration is dealing with third-party suppliers who provide biometric components for use in applications. For example, a supplier of sensors may desire to include support for their entire family of sensors as a single entity (e.g. a library). To achieve this, the supplier may consider implementing the library as a single BSP, but may choose to not implement monolithic methods (i.e. aggregated functionality such as Enrol, which does in one single call the capture, the processing of the sample, the creation of the biometric reference and even the archiving). Without monolithic methods, the BSP forces the application to take responsibility for the implementation of monolithic style functionality. This situation means the BSP may pass biometric data back to the application so the application may perform processing. The application may then need to pass this data on to another BSP for additional processing. In some contexts, the passing of data out to an application may simply be inefficient, in other contexts this may be a security concern. A possible solution for this inconvenience is to allow the hypothetical sensor BSP to interact directly with other BSPs, rather than involving the application in the management of this communication. To achieve this, the biometric

product provider may create an entity (e.g. a library) containing several BioAPI_Units of the same kind. This is called a Biometric Function Provider (BFP) and exhibits the following characteristics.

— The BFP shall only host BioAPI_Units of the same category.

— The BFP is meant to allow that a BSP is linked to one of its BioAPI_Units, in order to complete or adapt the functionality of the BSP.

— The BFP shall not provide functionality directly to the application, but only link to the BSP. The BSP communicates with the BFP on behalf of the application. The BSP is responsible for providing the BFP's functionality to the application.

The above mentioned situation also solves a problem from developers' point of view, which deals with simplicity in developing applications. If an application may require to use BioAPI_Units from different providers (e.g. a sensor from one provider and processing, comparison and archive BioAPI_Units from other provider), then it will have to load two different BSPs, calling each of the methods independently. As mentioned earlier, this has the inconvenience that it won't be possible to call a monolithic method, such as Verify(), which performs the data capture, the processing, extraction of the biometric reference from the database, the comparison and taking the decision, all within the same single call. Then, the application programmer will have to know which individual methods have to be called from each of the BSPs, in order to get the same functionality. To summarize, a level of abstraction and efficiency is achieved through the use of BFPs to segregate functionality into discrete sets. Each BFP aggregates a set of functionality provided by a product supplier (through a BioAPI_Unit). These BFPs are used by a BSP, where the BSP combines functionality from each BFP to offer monolithic methods for use by a client. The client application may then call the monolithic methods in the BSP without concern for the underlying implementation offered by a product supplier.

Another consideration for an application designer is the concern about security in regards to certain operations. Biometric data is typically sensitive personal data and some implementations of OO BioAPI may require the client application is not capable of directly accessing this data (ie: BIRs). By taking advantage of BFPs, and the abstraction facilities they offer, the possibility of tampering by malware or malicious users at the application level is removed. By using BFPs, all sensitive data will be handled at a level no higher than the BSP level. This means no BIR will be accessible to the application, not only simplifying application development, but eases concerns in relation to security.

The BFP shall not be accessed directly by the application. BioAPI calls are created to allow the application to know the BioAPI_Units that are contained in the BFPs so that the application may later request one BSP to include one of those BioAPI_Units of the BFP. This is done through a component registry and the procedure is further described in 5.4.

All the above ideas shall be implemented to allow the dynamic selection of components to be used by the application, but with no a-priori knowledge from the application developer. This is achieved by the inclusion of a common framework, which shall be installed in the system where the application is expected to be running. Then, the application shall request the framework for the list of the BSPs and BFPs installed, select the BSPs (with the requested selection of BioAPI_Units from BFPs) to be instantiated dynamically by the framework, and then accessing their methods and data structures through the framework. The application shall never be allowed to access the BSPs directly. This is summarized in Figure 1.

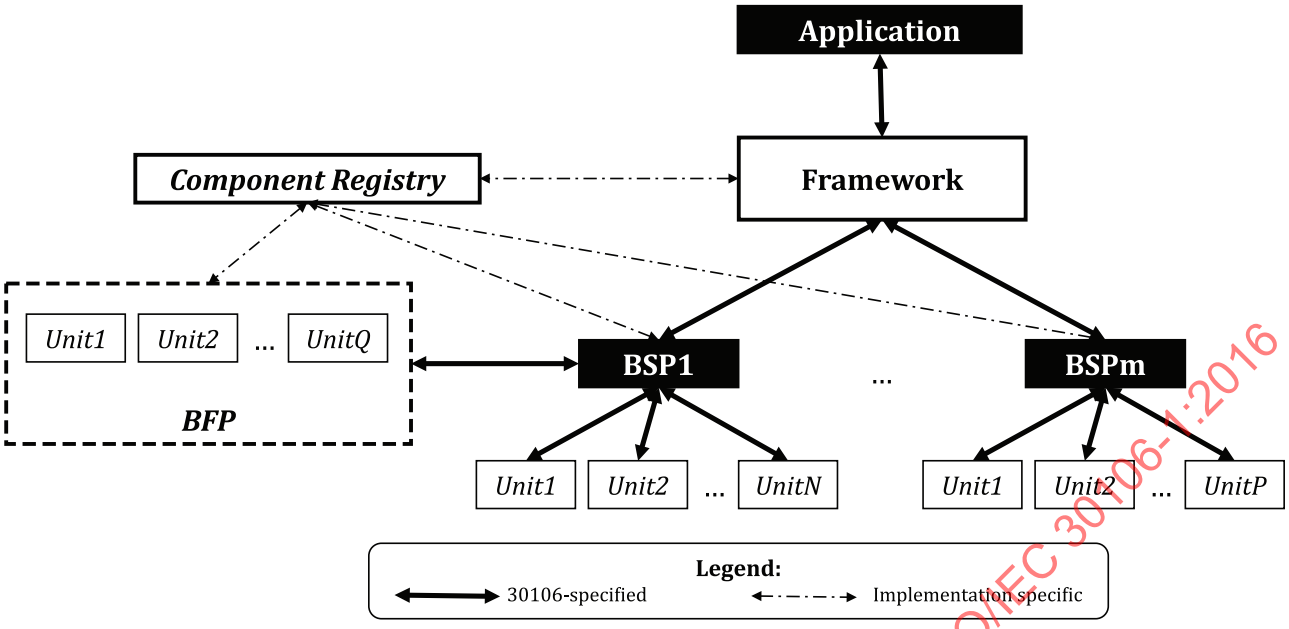

Figure 1 — Generic structure of a framework-based application

BSPs may be accessed by several applications at the same time, and it may also happen that BioAPI_Units in the BFPs are also accessed by several BSPs at the same time. It is up to the implementation of the framework the way that this implemented (e.g. this may be done by queuing requests from the different sources, or by responding to occurring events).

For those developers that are familiar with ISO/IEC 19784, it is important to provide a relationship about the interfaces there defined and the different layers that compose these International Standards. The correspondence between both International Standards is given in Table 1.

Table 1 — Correspondence between ISO/IEC 30106 and ISO/IEC 19784 interfaces

Interface in ISO/IEC 30106	Interface in ISO/IEC 19784
IFramework	API
IBSP	SPI
IBFP	SFPI

Along the ISO/IEC 30106 series, the Unit level will be also specified, not as for specifying a different interface, but for structuring in a hierarchical way the support for each of the categories of BioAPI_Units at the IBSP or IBFP interfaces.

5.2 BioAPI compatibility requirements

This part of ISO/IEC 30106 is defined as compatible with BioAPI version 3.0 (ISO/IEC 19784-1.revision 1), but the framework may be developed as to allow the cross compatibility with previous versions of BioAPI, as to allow BioAPI 2.x compliant BSPs and BFPs to be used in an ISO/IEC 30106 compliant application.

There is no provision to allow Object Oriented BSPs to be used in an ISO/IEC 19784-compliant framework and application.

5.3 Graphical User Interface (GUI)

A BSP may handle by itself, the interaction between the user and the system. For example, a BSP with a sensor BioAPI_Unit may address all the interaction by the displaying messages, illuminating LEDs

and/or activating sounds at the sensor itself. Such kind of interaction may include providing feedback to the user on

— the finger to be placed at the sensor,

— how to improve the location of the sample in the sensor (e.g. alignment of the biometric characteristic in relation to the sensor active area),

— whether the acquisition has been done in a correct way, or

— whether the biometric comparison has led to a positive or negative decision.

But in certain cases, it may be requested that such interaction is handled by the application using such BSP, showing all messages and illustrations in the computer screen. This may also be handled by the BSP itself, but this can lead to certain kinds of inconveniences such as the following:

— the BSP graphical user interface may not be adaptable to the different sizes and resolutions that the application screen is using;

— the feedback provided has to follow the same look and feel as the main application, including the use of service provider logos;

— the feedback has to be adaptable to the language of the user being identified, or where the system is deployed.

In those cases, it is recommended in order to avoid re-programming the BSP or increasing the complexity of the BSP, the interaction is handled by the application itself. As the feedback depends on the specific step that the BSP is executing, and those steps are mostly controlled by the BSP (i.e. most BSP functions are offered to the application as monolithic methods), then the way to solve this is by using callback functions.

Callback functions shall be developed for each step where a BSP is required to either interact with, or provide feedback to, a user. Naturally, the application shall provide a set of functions the BSP's callbacks will map to, which the BSP will call within the context of the original method call.

Therefore, if the BSP allows the use of callback functions for handling the GUI, the following development actions shall be taken, for each of the processes requiring such interaction (e.g. Capture).

— The application shall implement all GUI related functions for such process. These functions are of the following kinds:

  — Select Function: to show the interaction with the user at the beginning or the end of the whole process.

  EXAMPLE At the beginning the process, the following message may be provided: "Capture process for the index finger of the right hand". And at the end of the process, the message to be given could be: "Acquisition finished. Thank you!". Therefore, the callback function shall know if it is being called at the beginning of the process, or at the end. Also, the callback function can also provide the BSP with information about a user action, such as cancelling the capture process. All this information flow is provided in parameters of the callback function.

  — State Function: Sometimes the BSP process may take several internal steps, which can be designed as different states. Therefore, the state function will provide feedback depending on whether a certain internal step is to be started or finished. This leads to the need for the State Function to know which internal step is being processed, and that information will be provided as parameters to the callback function. Also, feedback provided by the user will be passed to the BSP by the parameters involved, such as cancelling the whole process.

  — Progress Function: In some other cases, it may be necessary to provide the user with real time feedback on the progression of the process.

  EXAMPLE A system may be showing in the screen the live image of the fingerprint or face within the acquisition process. In such case, a Progress callback function may be implemented to show the data stream.

— Once the callback functions have been implemented for the process, the application shall report the BSP about the callback functions to be used, before calling the process. This will be done by subscribing the BSP to those selected GUI events, and the relevant callback functions.

— Once subscribed, the application can call the process (e.g. Capture).

— Finally, when the process has been completed, the application can unsubscribe the BSP from the previously subscribed events.

NOTE Further information can be found in ISO/IEC 19784-1.

5.4 Implementation guidelines

5.4.1 Basic concepts

A common scenario for BioAPI's implementation is one where a framework is developed. A BioAPI framework provides a further abstraction, on top of BSPs and BFPs, allowing an application developer to use biometric components (BSPs or BFPs) without concern for their underlying implementation. In a framework approach, it is the responsibility of the application developer to determine which functions offered by BSPs or BFPs are available to an end-user.

In an Object Oriented BioAPI implementation, three developer roles are involved in the use of this part of ISO/IEC 30106. The following points provide guidance on which clauses are applicable to each developer role:

— Developers providing services (ie: BSPs or BFPs) for access by biometric applications, will find 5.4.2, 5.4.3 and 5.4.4 are applicable

— Framework developers involved with integrating the framework into a system and making BSPs and BPFs accessible to such systems will find 5.4.5 applicable

— Application developers who directly use third party BSPs will find 5.4.6 is applicable to this group of developers.

5.4.2 BioAPI_Unit development

BSPs and BFPs will implement BioAPI_Units. They shall be developed including all properties and methods defined in this International Standard. If when implementing a certain BioAPI_Unit, the developer does not want to provide a certain functionality, then, the following indications shall be followed.

— For each of the methods, including the overloading of them, that are not to be supported, the method shall be programmed and published, but containing only the throw of a BioAPIException indicating that such function is not supported.

— Whatever public property is involved only in the non-supported functionality, its value shall be set to NULL.

— The UnitSchema shall provide an accurate information on the functionality supported by the BioAPI_Unit.

It is necessary to remark that the implementation of BioAPI_Units is entirely proprietary and is strictly supported and implemented by BioAPI_BSP developer.

5.4.3 BFP development

As it was mentioned in 5.1, a Biometric Function Provider (i.e. BFP), is a set of BioAPI_Units of the same category (i.e. either archive, comparison, processing or sensor), that are provided for the BSPs, so that

a BSP can use one of such BioAPI_Units, either by its own decision, or by request of the application. This relationship is implemented in the following way:

a) The application loads the BSP by calling the BSPLoad method.

b) When executing the BSPLoad() method it is recommended the BSP determines which BSPs are included in the Component Registry. The Component Registry also defines the set of BioAPI_Units contained in each BSP. Probing for this information is achieved by calling the callback function BFPEnumerationCallback(). Calling BFPEnumerationCallback() will result in the return of BFPSchemas.

    1) For every BioAPI_Unit the BSP reads its UnitSchema structure to discover whether the BioAPI_Unit is compatible with the BSP and therefore a supported BioAPI_Unit.

    2) The BSP retains an internal register of supported BioAPI_Units and the BFP to which each belongs. The BSP also assigns a unique identifier, called UnitID, to each supported BioAPI_Unit.

c) After loading the BSP the application may interrogate the BSP about the supported BioAPI_Units in order to discover which BioAPI_Units are available. This is done be calling the QueryUnits() method.

d) The application shall be able to use any BioAPI_Unit of those offered by the BSP. The BioAPI_Unit can be offered by two means (this is applicable to each of the four BioAPI_Unit categories, i.e. archive, comparison, processing and sensor):

    1) The BSP already contains the BioAPI_Unit to be used.

    2) The BioAPI_Unit selected is not integrated in the BSP, but it is provided to the BSP by a compatible BFP automatically linked by the BSP during its loading process.

e) Once the link between the BSP and all offered BioAPI_Units is stated, the application can use those BioAPI_Units by calling each of the relevant methods, indicating the UnitID selected.

In addition, the developer shall include also those methods needed for the installation, de-installation and the dynamic instantiation of the available methods and properties. Also, the dynamic relationship between BSPs and BioAPI_Units into BFPs shall be covered.

Finally, when developing a BFP, all indications about the support for the implementation BioAPI_Units shall be followed (see 5.4.2).

5.4.4 BSP development

When developing a BSP, all methods defined for a BSP in this part of ISO/IEC 30106 shall be implemented, including methods related to non-included BioAPI_Units. It may also happen that a BSP does not offer detailed functionality for its BioAPI_Units and only provides the use of monolithic methods to the application (e.g. due to security concerns). For these cases, the following rules shall apply.

— Unless the BSP does not want to publish the detailed functionality of a BioAPI_Unit, the BSP methods that are related to BioAPI_Unit methods shall only contain the call to the method of the referenced BioAPI_Unit, returning the value returned by such method.

— If a method does not want to publish the detail functionality of one of its BioAPI_Units, then the non-supported method shall throw a BioAPIException indicating the non-supported functionality. Also, if there are properties related only to non-supported methods, its value shall be set to NULL. And finally, the BSPSchema shall denote this lack of functionality.

— For those cases where the BSP is not going to support the selection of new BioAPI_Units from BFPs, and that category of BioAPI_Unit is not included in the BSP, then, the relevant methods shall be implemented throwing the BioAPIException indicating the lack of that functionality.

— In case the BSP accepts the selection of new BioAPI_Units, and when loaded it discovers the existence of supported BioAPI_Units that cover that functionality, the BSP shall update its BSPSchema in

order to notify that functionality, and shall have a rule to allow the selection of any of the available BioAPI_Units.

— If a certain method within the BSP requires the previous link between the BSP and a BFP BioAPI_Unit, it shall be developed in a way that, if not BioAPI_Unit selection has been done, it throws the function not supported BioAPIException, and if the BioAPI_Unit selection has already occurred, it calls the selected BioAPI_Unit method forwarding the request from the application.

In addition, the developer shall include also those methods needed for the installation, de-installation and the dynamic instantiation of the available methods and properties. Also, the dynamic relationship between BSPs and BioAPI_Units into BFPs shall be covered.

5.4.5 Framework and component registry

The development of the framework is also related to the development of the component registry, which shall be implemented in a way that a dynamic and non-volatile listing of the installed BSPs and BFPs is stored. Also, the framework shall guarantee that an interoperable installation procedure is available for the platform where it is being executed and that the application has all BSPs and BFPs discovery mechanisms available.

5.4.6 Application development

An application developer shall be aware of the operations and options offered by the Framework, as well as which BSPs and BioAPI_Units are available via the Framework. The application developer shall ensure the application responds appropriately to exceptions raised by the Framework, whether the exceptions indicate called functionality is not available or some other exception has occurred. It is recommended, for simplicity of the application's code, the application developer uses the highest level methods available in the Bio API call stack (e.g. those in which all the biometric data is handled within the BSP and not circulated through the application).

To assist the application developer in determining the features and functions provided by the Framework, the following methods are available.

— Even before loading a BSP, the application can ask the Framework about which BSPs and BFPs are installed, by calling the methods EnumBSPs() and EnumBFPs() respectively.

— Also, the application can obtain further information about the BFPs, by calling QueryBFPs() method.

6 BioAPI CBEFF Patron Formats

6.1 General

Object Oriented BioAPI is able to use biometric data coded as Self-Identifying BIRs, either Simple BIRs or Complex BIRs, following the structure and definition of CBEFF (i.e. ISO/IEC 19785). Therefore, this Clause describes both, the Simple BIR and the Complex BIR data structures in binary format.

This patron format may be expanded by using other type of coded, such as a Self-Identifying TLV coding, as long as they are defined in the latest revision of ISO/IEC 19785-3.

6.2 Simple BIR

The data elements of a Simple BIR are provided in the following table. It is stated which elements from those described in ISO/IEC 19785 are mandatory or not.

Table 2 — CBEFF data elements of the patron format for a Simple BIR

CBEFF data element name	Field name	Length and optionality[a]	Abstract values and Encodings[b]
The following fields shall occur at most once			
CBEFF_BIR_self_id_owner	**BirSelfIdOwner**	**2, mandatory**	**1..65535 (for SC37 30106-x Simple: 257)**
CBEFF_BIR_self_id_type	**BirSelfIdType**	**2, mandatory**	**1..65535 (for SC37 30106-x Simple: 9)**
not a standard CBEFF data element	**BIRLength**	**4, mandatory**	**The complete length in bytes of the whole BIR, including header and SB (if applicable)**
CBEFF_version	cbeffVersion	1, mandatory	Major '3' and Minor '0': 0x000030
CBEFF_patron_header_ versi on	patronHeaderVersion	1, mandatory	1
CBEFF_BDB_format_owner	bdbFormatOwner	2, mandatory	0..65535
CBEFF_BDB_format_type	bdbFormatType	2, mandatory	0..65535
CBEFF_BDB_encryption_options	bdbEncryption	1; mandatory	NO ENCRYPTION: 0 ENCRYPTION: 1
CBEFF_BIR_integrity_options	birIntegrity	1, mandatory	NO INTEGRITY: 0 INTEGRITY: 1
CBEFF_subheader_count	**numChildren**	**1, mandatory**	**0**

Table 2 *(continued)*

CBEFF data element name	Field name	Length and optionality[a]	Abstract values and Encodings[b]
not a standard CBEFF data element	fieldPresence	4, mandatory	A 32-bit field containing one bit for each optional field in the patron format. The bit value '1' means that the corresponding field is present in the BIR instance. Bit position (0=least significant, 31=least significant) and corresponding optional field (those not defined yet shall be set to '0'): 0. bdbBiometricType 1. bdbBiometricSubtype 2. bdbCaptureDeviceOwner and Type 3. bdbFeatureExtractionAlgOwner and Type 4. bdbComparisonAlgOwner and Type 5. bdbCompressionAlgOwner and Type 6. bdbPADTechniqueOwner and Type 7. bdbChallengeResponse 8. bdbCreationDate 9. bdbIndex 10. bdbProcessedLevel 11. bdbProductOwner and Type 12. bdbPurpose 13. bdbQuality 14. bdbOwner and Type 15. bdbValidityPeriod 16. birCreationDate 17. birCreator 18. birIndex 19. birPayload 20. birPointer 21. birValidityPeriod 22. sbFormatOwner and Type

Table 2 *(continued)*

CBEFF data element name	Field name	Length and optionality[a]	Abstract values and Encodings[b]
CBEFF_BDB_biometric_type	bdbBiometricType	4	*This encoding is a 4 octet bitmap. NO VALUE AVAILABLE is encoded as all 0 bits. If MULTIPLE BIOMETRIC TYPES is set, other bits may also be set to enumerate the types contained in the BDB.* NO VALUE AVAILABLE: X'00 00 00' MULTIPLE BIOMETRIC TYPES: X'00 00 01' FACE: X'00 00 02' VOICE: X'00 00 04' FINGER: X'00 00 08' IRIS: X'00 00 10' RETINA: X'00 00 20' HAND GEOMETRY: X'00 00 40' SIGNATURE OR SIGN: X'00 00 80' KEYSTROKE: X'00 01 00' LIP MOVEMENT: X'00 02 00' GAIT: X'00 10 00' VEIN: X'00 20 00' DNA: X'00 40 00' EAR: X'00 80 00' FOOT: X'01 00 00' SCENT: X'02 00 00'
CBEFF_BDB_biometric_subt ype	bdbBiometricSubtype	1	*This encoding is a 1 octet bitmap. Combinations of abstract values are permitted (by ORing the encodings for those values) when the abstract value encoded in CBEFF_BDB_biometric_type represents a biometric technology that can create a BDB where multiple subtypes are supported.* NO VALUE AVAILABLE: b'0000 0000' LEFT: b'0000 0001' RIGHT: b'0000 0010' LEFT THUMB: b'0000 0101' LEFT INDEX FINGER: b'0000 1001' LEFT MIDDLE FINGER: b'0001 0001' LEFT RING FINGER: b'0010 0001' LEFT LITTLE FINGER: b'0100 0001' RIGHT THUMB: b'0000 0110' RIGHT INDEX FINGER: b'0000 1010' RIGHT MIDDLE FINGER: b'0001 0010' RIGHT RING FINGER: b'0010 0010' RIGHT LITTLE FINGER: b'0100 0010' LEFT PALM: b'1000 0101' LEFT BACK OF HAND: b'1000 1001' LEFT WRIST: b'1001 0001' RIGHT PALM: b'1000 0110' RIGHT BACK OF HAND: b'1000 1010' RIGHT WRIST: b'1001 0010'
CBEFF_BDB_capture_device_owner	bdbCaptureDeviceOwner	2	0..65535, being 0 – NO VALUE AVAILABLE
CBEFF_BDB_capture_device_type	bdbCaptureDeviceType	2	0..65535, being 0 – NO VALUE AVAILABLE
CBEFF_BDB_feature_extraction_algorithm_owner	bdbFeatureExtractionAlgOwner	2	0..65535, being 0 – NO VALUE AVAILABLE
CBEFF_BDB_feature_extraction_algorithm_type	bdbFeatureExtractionAlgType	2	0..65535, being 0 – NO VALUE AVAILABLE
CBEFF_BDB_comparison_algorithm_owner	bdbComparisonAlgOwner	2	0..65535, being 0 – NO VALUE AVAILABLE
CBEFF_BDB_comparison_algorithm_type	bdbComparisonAlgType	2	0..65535, being 0 – NO VALUE AVAILABLE
CBEFF_BDB_compression_algorithm_owner	bdbCompressionAlgOwner	2	0..65535, being 0 – NO VALUE AVAILABLE

Table 2 *(continued)*

CBEFF data element name	Field name	Length and optionality[a]	Abstract values and Encodings[b]
CBEFF_BDB_compression_algorithm_type	bdbCompressionAlgType	2	0..65535, being 0 – NO VALUE AVAILABLE
CBEFF_BDB_PAD_technique_owner	bdbPADTechniqueOwner	2	0..65535, being 0 – NO VALUE AVAILABLE
CBEFF_BDB_PAD_technique_type	bdbPADTechniqueType	2	0..65535, being 0 – NO VALUE AVAILABLE
CBEFF_BDB_challenge_response	bdbChallengeResponse	2 + 0..65535	Variable-length octet string, preceded by a 16-bit integer field containing the length (octets).
CBEFF_BDB_creation_date	bdbCreationDate	7	The string shall represent a date (or date and a time of the day). Format is: 2 bytes for the year 1 byte for the month 1 byte for the day 1 byte for the hour 1 byte for the minute 1 byte for the second All set to 0, or not present, means NO VALUE AVAILABLE
CBEFF_BDB_index	bdbIndex	2 + 0..65535	Variable-length octet string, preceded by a 16-bit integer field containing the length (octets). Shall not appear in any BIR in which numChildren is not x'00'.
CBEFF_BDB_processed_level	bdbProcessedLevel	1	RAW: 1 INTERMEDIATE: 2 PROCESSED: 3
CBEFF_BDB_product_owner	bdbProductOwner	2	Company that owns the BSP creating the BIR 0..65535, being 0 – NO VALUE AVAILABLE
CBEFF_BDB_product_type	bdbProductType	2	0..65535, being 0 – NO VALUE AVAILABLE
CBEFF_BDB_purpose	bdbPurpose	1	VERIFY: 1 IDENTIFY: 2 ENROLL: 3 ENROLL FOR VERIFICATION ONLY: 4 ENROLL FOR IDENTIFICATION ONLY: 5 AUDIT: 6
CBEFF_BDB_quality	bdbQuality	1	QUALITY NOT SUPPORTED BY BDB CREATOR: 255 QUALITY SUPPORTED BY BDB CREATOR BUT NOT SET: 254 INTEGER VALUE: 0 – 100
CBEFF_BDB_quality_algorithm_owner	bdbOwner	2	0..65535, being 0 – NO VALUE AVAILABLE

Table 2 *(continued)*

CBEFF data element name	Field name	Length and optionality[a]	Abstract values and Encodings[b]
CBEFF_BDB_quality_algorithm_type	bdbType	2	0..65535, being 0 – NO VALUE AVAILABLE
CBEFF_BDB_validity_period	bdbValidityPeriod	14	The string shall represent an interval of two dates (or date and time of the day). Both dates are coded in 7 bytes, as the bdbCreationDate.
CBEFF_BIR_creation_date	birCreationDate	7	The string shall represent a date (or date and a time of the day). Format is: 2 bytes for the year 1 byte for the month 1 byte for the day 1 byte for the hour 1 byte for the minute 1 byte for the second All set to 0, or not present, means NO VALUE AVAILABLE
CBEFF_BIR_creator	birCreator	2 + 0..65535	Variable-length ISO/IEC 10646 character string, encoded in UTF-8, and preceded by a 16-bit integer field containing the length of the UTF-8 encoding (octets).
CBEFF_BIR_index	birIndex	2 + 0..65535	Variable-length octet string, preceded by a 16-bit integer field containing the length (octets). Shall not inherit its value from any other level BIR.
CBEFF_BIR_payload	birPayload	2 + 0..65535	Variable-length octet string, preceded by a 16-bit integer field containing the length (octets). Shall not inherit its value from any other level BIR.
CBEFF_BIR_pointer	birPointer	2 + 0..65535	Variable-length octet string, preceded by a 16-bit integer field containing the length (octets). Provide the pointer (path or registry ID) to the following BIR to be addressed in a multibir schema.
CBEFF_BIR_validity_period	birValidityPeriod	14	The string shall represent an interval of two dates (or date and time of the day). Both dates are coded in 7 bytes, as the bdbCreationDate.
CBEFF_SB_format_owner	sbFormatOwner	2 , **conditional**	1..65535

Table 2 *(continued)*

CBEFF data element name	Field name	Length and optionality[a]	Abstract values and Encodings[b]
CBEFF_SB_format_type	sbFormatType	2 , **conditional**	1..65535
BDB	bdb	4 + 0..4294967295	Variable-length octet string, preceded by a 32-bit integer field containing the length (octets). If this field is present in a BIR instance (as indicated in bit 19 of the field *fieldPresence*), then no child BIRs shall be included (*numChildren* shall have the value 0). Otherwise, at least one child BIR shall be included (*numChildren* shall have a value greater than 0). NOTE - The content and encoding of the BDB are not specified by CBEFF nor by this patron format specification.
SB	sb	4 + 0..4294967295, **conditional**	Variable-length octet string, preceded by a 32-bit integer field containing the length (octets).

6.3 Complex BIR

6.3.1 Structure

The structure of a Complex BIR is as described in ISO/IEC 19785, where the BIR is composed by a hierarchy of Parent BIRs, which enclose a number or Child BIRs. This Clause defines both, the data structure of the Child BIRs and the data structure of each of the Parent BIRs.

6.3.2 Child BIR

The data structure of the Child BIR shall follow the same specification of that of the Simple BIR previously defined in this part of ISO/IEC 30106.

6.3.3 Parent BIR

For each of the Parent BIRs the CBEFF patron format shall follow the specification provided in the following table.

Table 3 — CBEFF data elements of the patron format for a Complex BIR

CBEFF data element name	Field name	Length and optionality[a]	Abstract values and Encodings[b]
The following fields shall occur at most once			
CBEFF_BIR_self_id_owner	**BirSelfIdOwner**	**2, mandatory**	**1..65535 (for SC37 30106-x Simple: 257)**
CBEFF_BIR_self_id_type	**BirSelfIdType**	**2, mandatory**	**1..65535 (for SC37 30106-x Simple: 9)**
not a standard CBEFF data element	**BIRLength**	**4, mandatory**	**The complete length in bytes of the whole BIR, including header and SB (if applicable)**
CBEFF_version	cbeffVersion	1, mandatory	Major '3' and Minor '0': 0x000030
CBEFF_patron_header_versi on	patronHeaderVersion	1, mandatory	1
CBEFF_BIR_integrity_option s	birIntegrity	1, mandatory	NO INTEGRITY: 0 INTEGRITY: 1

Table 3 *(continued)*

CBEFF data element name	Field name	Length and optionality[a]	Abstract values and Encodings[b]
CBEFF_subheader_count	**numChildren**	**1, mandatory**	**1..255**
not a standard CBEFF data element	fieldPresence	4, mandatory	A 32-bit field containing one bit for each optional field in the patron format. The bit value '1' means that the corresponding field is present in the BIR instance. Bit position (0=least significant, 31=least significant) and corresponding optional field (those not defined yet shall be set to '0'): 0. bdbBiometricType 1. bdbBiometricSubtype 2. bdbCaptureDeviceOwner and Type 3. bdbFeatureExtractionAlgOwner and Type 27. bdbComparisonAlgOwner and Type 28. bdbCompressionAlgOwner and Type 29. bdbPADTechniqueOwner and Type 30. bdbChallengeResponse 31. bdbCreationDate 32. bdbIndex 33. bdbProcessedLevel 34. bdbProductOwner and Type 35. bdbPurpose 36. bdbQuality 37. bdbOwner and Type 38. bdbValidityPeriod 39. birCreationDate 40. birCreator 41. birIndex 42. birPayload 43. birPointer 44. birValidityPeriod 45. sbFormatOwner and Type

Table 3 *(continued)*

CBEFF data element name	Field name	Length and optionality[a]	Abstract values and Encodings[b]
CBEFF_BIR_creation_date	birCreationDate	7	The string shall represent a date (or date and a time of the day). Format is: 2 bytes for the year 1 byte for the month 1 byte for the day 1 byte for the hour 1 byte for the minute 1 byte for the second All set to 0, or not present, means NO VALUE AVAILABLE
CBEFF_BIR_creator	birCreator	2 + 0..65535	Variable-length ISO/IEC 10646 character string, encoded in UTF-8, and preceded by a 16-bit integer field containing the length of the UTF-8 encoding (octets).
CBEFF_BIR_index	birIndex	2 + 0..65535	Variable-length octet string, preceded by a 16-bit integer field containing the length (octets). Shall not inherit its value from any other level BIR.
CBEFF_BIR_payload	birPayload	2 + 0..65535	Variable-length octet string, preceded by a 16-bit integer field containing the length (octets). Shall not inherit its value from any other level BIR.
CBEFF_BIR_validity_period	birValidityPeriod	1 + 17..31	Variable-length ASCII character string, preceded by an 8-bit integer field containing the length (characters). The string shall represent an interval of two dates (or date and time of the day)d.
CBEFF_SB_format_owner	sbFormatOwner	2 , **conditional**	1..65535
CBEFF_SB_format_type	sbFormatType	2 , **conditional**	1..65535
BDB	bdb	4 + 0..4294967295	Variable-length octet string, preceded by a 32-bit integer field containing the length (octets). If this field is present in a BIR instance (as indicated in bit 19 of the field *fieldPresence*), then no child BIRs shall be included (*numChildren* shall have the value 0). Otherwise, at least one child BIR shall be included (*numChildren* shall have a value greater than 0). NOTE - The content and encoding of the BDB are not specified by CBEFF nor by this patron format specification.
The following 3 fields shall occur as a group as many times as specified in the field numChildren (0..255)			

Table 3 *(continued)*

CBEFF data element name	Field name	Length and optionality[a]	Abstract values and Encodings[b]
not a standard CBEFF data element	**childBir**	**0..4294967295, mandatory**	**Coded as a simple BIR (child BIR)**
The following field shall occur at most once			
SB	sb	4 + 0..4294967295	Variable-length octet string, preceded by a 32-bit integer field containing the length (octets).

7 Constants

7.1 General

For the use of this part of ISO/IEC 30106 in its different parts, the following constants apply, which are grouped by expected use.

7.2 Biometric types

— int NoValueAvailableValue = 0x000000;

— int MultipleBiometricTypesValue = 0x000001;

— int FaceValue = 0x000002;

— int VoiceValue = 0x000004;

— int FingerValue = 0x000008;

— int IrisValue = 0x000010;

— int RetinaValue = 0x000020;

— int HandGeometryValue = 0x000040;

— int SignatureOrSignValue = 0x000080;

— int KeystrokeValue = 0x000100;

— int LipMovementValue = 0x000200;

— int GaitValue = 0x001000;

— int VeinValue = 0x002000;

— int DNAValue = 0x004000;

— int EarValue = 0x008000;

— int FootValue = 0x010000;

— int ScentValue = 0x020000;

7.3 Biometric subtypes

— byte NoValueAvailableValue = 0x00; // b'0000 0000'

— byte LeftValue = 0x01; // b'0000 0001'

— byte RightValue = 0x02; // b'0000 0010'

— byte LeftThumbValue = 0x05; // b'0000 0101'

— byte LeftIndexFingerValue = 0x09; // b'0000 1001'

— byte LeftMiddleFingerValue = 0x11; // b'0001 0001'

— byte LeftRingFingerValue = 0x21; // b'0010 0001'

— byte LeftLittleFingerValue = 0x41; // b'0100 0001'

— byte RightThumbValue = 0x06; // b'0000 0110'

— byte RightIndexFingerValue = 0x0A; // b'0000 1010'

— byte RightMiddleFingerValue = 0x12; // b'0001 0010'

— byte RightRingFingerValue = 0x22; // b'0010 0010'

— byte RightLittleFingerValue = 0x42; // b'0100 0010'

— byte LeftPalmValue = 0x85; // b'1000 0101'

— byte LeftBackOfHandValue = 0x89; // b'1000 1001'

— byte LeftWristValue = 0x91; // b'1001 0001'

— byte RightPalmValue = 0x86; // b'1000 0110'

— byte RightBackOfHandValue = 0x8A; // b'1000 1010'

— byte RightWristValue = 0x92; // b'1001 0010'

7.4 Error codes

The following table shows the error codes defined, where the error code is composed by the OR operation of one of the three error codes (i.e. the one stating the originator of the error) and the specific error found. Therefore, in an error value returned is 0x00000000 means no error occurred.

Table 4 — Error codes for Object oriented BioAPI

public const int BioAPIFrameworkError	0x00000000
public const int BioAPIBSPError	0x01000000
public const int BioAPIUnitError	0x02000000
public const int BioAPIErrInternalError	0x000101
public const int BioAPIErrMemoryError	0x000102
public const int BioAPIErrInvalidPointer	0x000103
public const int BioAPIErrInvalidInputPointer	0x000104
public const int BioAPIErrInvalidOutputPointer	0x000105
public const int BioAPIErrFunctionNotSupported	0x000106
public const int BioAPIErrOSAccessDenied	0x000107
public const int BioAPIErrFunctionFailed	0x000108
public const int BioAPIErrInvalidUUID	0x000109
public const int BioAPIErrIncompatibleVersion	0x00010a
public const int BioAPIErrInvalidData	0x00010b
public const int BioAPIErrUnableToCapture	0x00010c
public const int BioAPIErrTooManyHandles	0x00010d

Table 4 *(continued)*

public const int BioAPIErrTimeoutExpired	0x00010e
public const int BioAPIErrInvalidBIR	0x00010f
public const int BioAPIErrBIRSignatureFailure	0x000110
public const int BioAPIErrUnableToStorePayload	0x000111
public const int BioAPIErrNoInputBIRs	0x000112
public const int BioAPIErrUnsupportedFormat	0x000113
public const int BioAPIErrUnableToImport	0x000114
public const int BioAPIErrInconsistentPurpose	0x000115
public const int BioAPIErrBIRNotFullyProcessed	0x000116
public const int BioAPIErrPurposeNotSupported	0x000117
public const int BioAPIErrUserCancelled	0x000118
public const int BioAPIErrUnitInUse	0x000119
public const int BioAPIErrInvalidBSPHandle	0x00011a
public const int BioAPIErrFrameworkNotInitialized	0x00011b
public const int BioAPIErrInvalidBIRHandle	0x00011c
public const int BioAPIErrCalibrationNotSuccessful	0x00011d
public const int BioAPIErrPresetBIRDoesNotExist	0x00011e
public const int BioAPIErrDecryptionFailure	0x00011f
public const int BioAPIErrIdentifyInProgress	0x000120
public const int BioAPIErrLowQualityReferenceTemplate	0x000121
public const int BioAPIErrNoGUIEventHandler	0x000122
public const int BioAPIErrTransformationNotSupported	0x000123
public const int BioAPIErrComponentAlreadyRegistered	0x000125
public const int BioAPIErrComponentNotRegistered	0x000126
public const int BioAPIErrAlgorithmNotSupported	0x000127
public const int BioAPIErrEncodingError	0x000128
public const int BioAPIErrDecodingError	0x000129
public const int BioAPIErrConfigurationError	0x00012a
public const int BioAPIErrSegmentationError	0x00012b
public const int BioAPIErrSecurityBlockError	0x00012c
public const int BioAPIErrDataNotFreed	0x00012d
public const int BioAPIErrDataCreationError	0x00012e
public const int BioAPIErrWrongBiometricInstancesDetected	0x00012f
public const int BioAPIErrComponentFileRefNotFound	0x000201
public const int BioAPIErrBSPLoadFail	0x000202
public const int BioAPIErrBSPNotLoaded	0x000203
public const int BioAPIErrUnitNotInserted	0x000204
public const int BioAPIErrInvalidUnitID	0x000205
public const int BioAPIErrInvalidCategory	0x000206
public const int BioAPIErrInvalidDBHandle	0x000300
public const int BioAPIErrUnableToOpenDatabase	0x000301
public const int BioAPIErrDatabaseIsLocked	0x000302
public const int BioAPIErrDatabaseDoesNotExist	0x000303

Table 4 *(continued)*

public const int BioAPIErrDatabaseAlreadyExists	0x000304
public const int BioAPIErrInvalidDatabaseName	0x000305
public const int BioAPIErrRecordNotFound	0x000306
public const int BioAPIErrMarkerHandleIsInvalid	0x000307
public const int BioAPIErrDatabaseIsOpen	0x000308
public const int BioAPIErrInvalidAccessRequest	0x000309
public const int BioAPIErrEndOfDatabase	0x00030a
public const int BioAPIErrUnableToCreateDatabase	0x00030b
public const int BioAPIErrUnableToCloseDatabase	0x00030c
public const int BioAPIErrUnableToDeleteDatabase	0x00030d
public const int BioAPIErrDatabaseIsCorrupt	0x00030e
public cons tint BioAPIErrQueryExecutionFailed	0x00030f
public const int BioAPIErrLocationError	0x000400
public const int BioAPIErrOutOfFrame	0x000401
public const int BioAPIErrInvalidCrosswisePosition	0x000402
public const int BioAPIErrInvalidLengthwisePosition	0x000403
public const int BioAPIErrInvalidDistance	0x000404
public const int BioAPIErrLocationTooRight	0x000405
public const int BioAPIErrLocationTooLeft	0x000406
public const int BioAPIErrLocationTooHigh	0x000407
public const int BioAPIErrLocationTooLow	0x000408
public const int BioAPIErrLocationTooFar	0x000409
public const int BioAPIErrLocationTooNear	0x00040a
public const int BioAPIErrLocationTooForward	0x00040b
public const int BioAPIErrLocationTooBackward	0x00040c
public const int BioAPIErrQualityError	0x000501
public const int BioAPIErrTooEarly	0x000502
public const int BioAPIErrTooLate	0x000503
public const int BioAPIErrTooFast	0x000504
public const int BioAPIErrTooSlow	0x000505
public const int BioAPIErrTooLong	0x000506
public const int BioAPIErrTooShort	0x000507
public const int BioAPIErrTooLarge	0x000508
public const int BioAPIErrTooSmall	0x000509
public const int BioAPIErrSecurityProfileNotSet	0x000601
public const int BioAPIErrSecurityEncryptionAlgNotSupported	0x000610
public const int BioAPIErrSecurityEncryptionAlgNotSet	0x000611
public const int BioAPIErrSecurityEnckeyNotSet	0x000612
public const int BioAPIErrSecurityEncryptionFailure	0x000614
public const int BioAPIErrSecurityDecryptionFailure	0x000618
public const int BioAPIErrSecurityMACAlgNotSupported	0x000620
public const int BioAPIErrSecurityMACAlgNotSet	0x000621
public const int BioAPIErrSecurityMACkeyNotSet	0x000622

Table 4 *(continued)*

public const int BioAPIErrSecurityMACGenerationFailure	0x000624
public const int BioAPIErrSecurityMACVerificationFailure	0x000628
public const int BioAPIErrSecurityDigitalSignatureAlgNotSupported	0x000640
public const int BioAPIErrSecurityDigitalSignatureAlgNotSet	0x000641
public const int BioAPIErrSecurityDigitalSignatureGenerationFailure	0x000644
public const int BioAPIErrSecurityDigitalSignatureVerificationFailure	0x000648
public const int BioAPIErrSecurityChallengeNotSet	0x000701
public const int BioAPIErrSecurityBPUIOIndexNotSet	0x000702
public const int BioAPIErrSecuritySupremumBPUIOIndexNotSet	0x000704
public const int BioAPIErrSecurityMACAlgACBioNotSupported	0x000720
public const int BioAPIErrSecurityMACAlgACBioNotSet	0x000721
public const int BioAPIErrSecurityMACkeyACBioNotSet	0x000722
public const int BioAPIErrSecurityDigitalSignatureAlgACBioNot-Supported	0x000740
public const int BioAPIErrSecurityDigitalSignatureAlgACBioNotSet	0x000741
public const int BioAPIErrSecurityHashAlgACBioNotSupported	0x000780
public const int BioAPIErrSecurityHashAlgACBioNotSet	0x000781
public const int BioAPIErrTestVerifyFailed	0x000800
public const int BioAPIErrRawSampleInsufficientQuality	0x000900
public const int BioAPIErrProcessedSampleInsufficientQuality	0x000901
public const int BioAPIErrEnrolmentNotCompleted	0x000902
public const int BioAPIErrPresentationAttackDetected	0x000903
public const int BioAPIErrBIRNotFound	0x000904
public const int BioAPIErrStorageNotAvailable	0x000905
public const int BioAPIErrSampleNotIdentified	0x000906
public const int BioAPIErrUnitNotAvailable	0x000907

8 OO BioAPI UML structure

8.1 General

The BioAPI Object Oriented interface will be divided into two basic blocks.

— Block BioAPI: Contains functionality to manage units, BSPs, BFPs, the Framework and Applications.

— Block BioAPI-Data: Contains all the data structures.

NOTE The following parts could (but it is not mandatory) have additional blocks depending on the language implementation or in order to facilitate implementers build applications or BSPs (i.e. Template package).

The UML structure is going to be shown in the following subclauses, starting with the relationships among data structures (i.e. BioAPI-Data) and after that the BioAPI_Unit level will be provided, followed by the BFP and BSP levels, to end up with the Framework and the Application levels.

8.2 Relationships among data structures

8.2.1 Class BIR

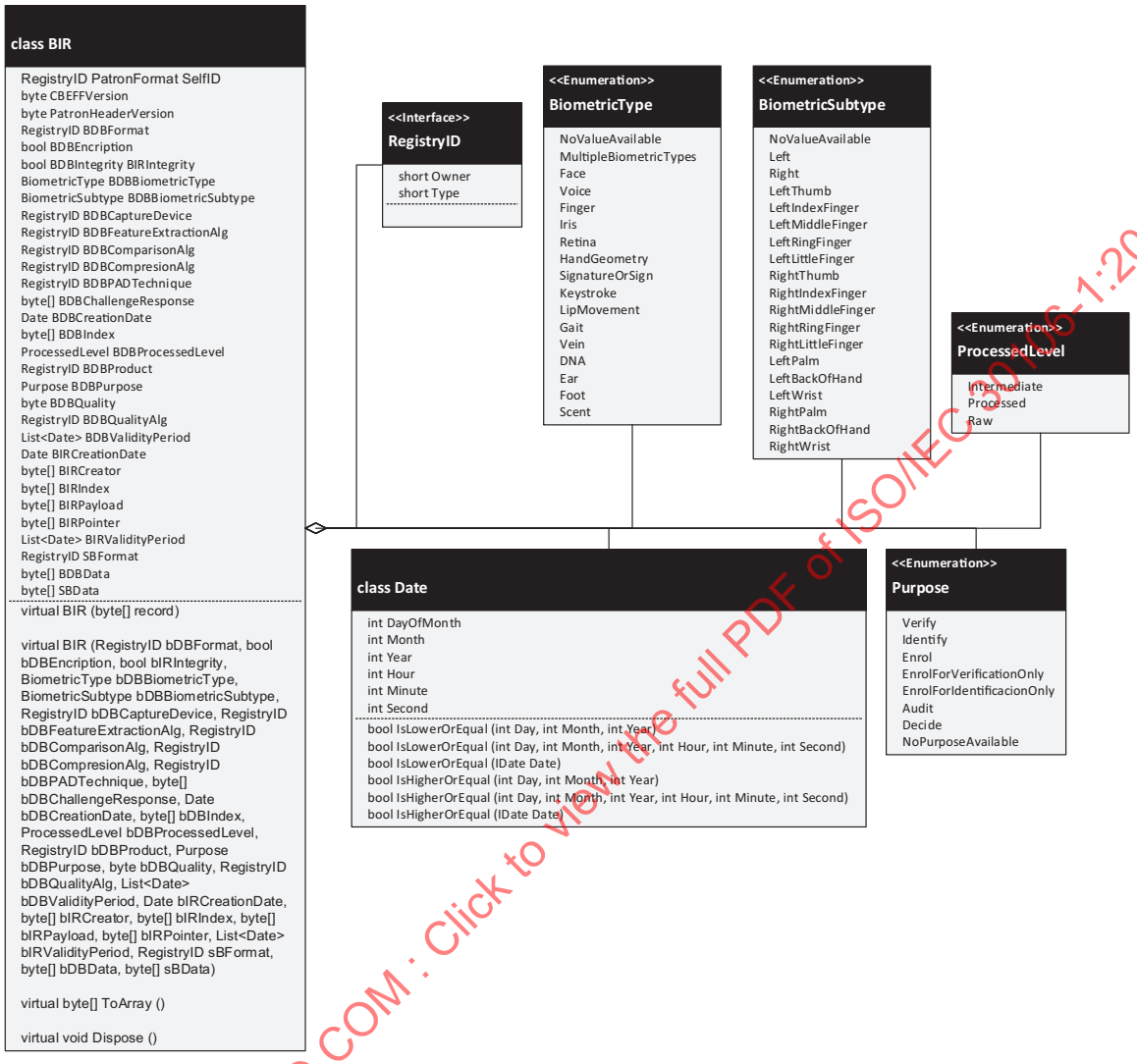

Figure 2 — UML diagram for class BIR