NIST
National Institute of
Standards and Technology
Technology Administration
U.S. Department of Commerce

Interagency Report 6887

Government Smart Card Interoperability Specification

Version 2.0

Jim Dray
Alan Goldfine
Michaela Iorga
Teresa Schwarzhoff
John Wack

July 8, 2002

NIST Interagency Report 6887

Government Smart Card Interoperability Specification

The National Institute of Standards and Technology

C O M P U T E R S E C U R I T Y

Reports on Computer Systems Technology

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof concept implementations, and technical analysis to advance the development and productive use of information technology. ITL's responsibilities include the development of technical, physical, administrative, and management standards and guidelines for the cost-effective security and privacy of sensitive unclassified information in Federal computer systems. This Interagency Report discusses ITL's research, guidance, and outreach efforts in computer security, and its collaborative activities with industry, government, and academic organizations.

Natl. Inst. Stand. Technol. Interagency Report 6887, 158 pages (July 2002)

THIS PAGE INTENTIONALLY LEFT BLANK.

Foreword

(a) The Government Smart Card Initiative

The Presidential Budget for Fiscal Year 1998 stated: "The Administration wants to adopt 'smart card' technology so that, ultimately, every Federal employee will be able to use one card for a wide range of purposes, including travel, small purchases, and building access." The General Services Administration (GSA) was requested to take the lead in developing the Federal business tools of electronic commerce and smart cards. The Federal Smart Card Implementation Plan was then developed, under which GSA will implement a pilot program to test Government smart cards and related systems. As part of the implementation plan, GSA formed the Government Smart Card Inter-Agency Advisory Board (GSC-IAB) to serve as a steering committee for the GSC program.

In 1999, the National Institute of Standards and Technology (NIST) agreed to lead development of specifications and standards related to the GSC program. NIST represents the GSC program in industry and government standards organizations as appropriate, to promote GSC technology. NIST is also charged with developing a comprehensive conformance test program for the Government Smart Card.

In May 2000, GSA awarded Contract No. GS00T00ALD0208 to five prime contractors to provide "Common Access Smart ID Card" goods and services. Information on the use and applicability of the GSA Contract can be found at http://www.gsa.gov/smartcard.

The GSC-IAB has established a Standards Technical Working Group (TWG), which consists of representatives of the contract awardees and federal agencies. The TWG, led by NIST, developed the Government Smart Card Interoperability Specification (GSC-IS), version 2.0, which is contained in this document. This Specification defines the Government Smart Card Interoperability Architecture, which satisfies the core interoperability requirements of the Common Access Smart ID Card contract and the GSC Program as a whole.

Products available under the contract will be subjected to a formal certification process to validate conformance to the requirements of the GSC-IS (see Section (c)).

(b) Change Management and Interpretation of the Specification

The GSC-IAB has the overall responsibility to develop the policy and procedures for handling revisions of the GSC-IS and any other maintenance. These procedures, and a continually updated list of corrections and other limited changes to version 2.0, will be posted on the NIST smart card program web site (see Section (d)).

It is also anticipated that additional language bindings to the Basic Services Interface (see Section (1.3)) will be developed and added to the GSC-IS.

In the longer term, it is expected that the next major revision of GSC-IS will be published about 2 years after version 2.0. It is also anticipated that the Specification will be submitted for formal standardization.

The interpretation of the GSC-IS is the responsibility of the GSC-IAB. Interpretation issues and their resolutions will be detailed on the NIST program web site (see Section (d)).

(c) Testing for Conformance

NIST is developing a comprehensive conformance test program in support of the GSC program. The goal of the conformance tests is to determine whether or not a given Government Smart Card product complies with the GSC Specification. Qualified laboratories will perform operational conformance testing.

(d) NIST Government Smart Card Program Web Site

NIST maintains a publicly accessible web site at http://smartcard.nist.gov. This page contains up-to-date information on all aspects of the GSC program related to the GSC-IS, including:

- General program descriptions and updates
- The latest version of the GSC-IS
- GSC-IS revision and standardization plans
- A list of errata and other changes to the last published version of the GSC-IS
- A list of interpretations and clarifications of the GSC-IS, as issued by the GSC-IAB
- Details of the GSC-IS interpretation procedures
- Details of the GSC-IS conformance-testing program.

Acknowledgements

The authors would like to acknowledge the efforts of the original Government Smart Card Interoperability Committee, the Government Smart Card Interagency Advisory Board, the General Services Administration, the prime contractors associated with the Smart Access Common ID Card contract, and the NIST smart card team. Composed of industry and government representatives, the Interoperability Committee developed the first Government Smart Card Interoperability Specification (version 1.0) during the summer of 2000.

The efforts of the Standards Technical Working Group of the Government Smart Card Interagency Advisory Board are also recognized. Chaired by the National Institute of Standards and Technology, the Standards Technical Working Group was responsible for reviewing the original Government Smart Card Interoperability Specification and developing an extensive change list for the production of this new version of the Government Smart Card Interoperability Specification.

THIS PAGE INTENTIONALLY LEFT BLANK.

Table of Contents

Appendix Table of Contents

Figures and Tables

THIS PAGE INTENTIONALLY LEFT BLANK.

1. Introduction

1.1 Background

A typical configuration for a smart card system consists of a host computer with one or more smart card readers attached to hardware communications ports. Smart cards can be inserted into the readers, and software running on the host computer communicates with these cards using a protocol defined by ISO 7816-4 [ISO4]. The ISO standard smart card communications protocol defines Application Protocol Data Units (APDU) that are exchanged between smart cards and host computers. This APDU based interface is referred to as the "card edge" and the two terms are used interchangeably.

Client applications have traditionally been designed to communicate with ISO smart cards using the APDU protocol through low-level software drivers that provide an APDU transport mechanism between the client application and a smart card. Smart card families can implement the APDU protocol in a variety of ways, so client applications must have intimate knowledge of the APDU set of the smart card they are communicating with. This is generally accomplished by programming a client application to work with a specific card, since it would not be practical to design a client application to accommodate the different APDU sets of a large number of smart card families.

The tight coupling between client applications and smart card APDU sets has several drawbacks. Applications programmers must be thoroughly familiar with smart card technology and the complex APDU protocol. If the cards that an application is hard coded to use become commercially unavailable, the application must be redesigned to use different cards. Customers also have less freedom to select different smart card products, since their applications will only work with one or a small number of similar cards.

This Government Smart Card Interoperability Specification (GSC-IS) provides solutions to a number of the interoperability problems associated with smart card technology. The original version of the GSC-IS (version 1.0) was developed by the GSC Interoperability Committee led by the General Services Administration (GSA) and the National Institute of Standards and Technology (NIST), in association with the Smart Access Common Identification Card contract (Contract No. GS00T00ALD0208).

1.2 Scope, Limitations, and Applicability of the Specification

The GSC-IS defines an architectural model for interoperable smart card service provider modules, compatible with both file system cards and virtual machine cards. Smart cards using both the T=0 and T=1 communications protocols are supported. The GSC-IS includes a Basic Services Interface (BSI), which addresses interoperability of a core set of smart card services at the interface layer between client applications and smart card service provider modules. The GSC-IS also defines a mechanism at the card edge layer for interoperation with smart cards that use a wide variety of APDU sets, including both file system cards and virtual machine cards.

Interoperability is not addressed for the following areas:

- Smart card initialization
- Cryptographic key management
- Communications between smart cards and card readers
- Communications between smart card readers and host computer systems.

1.3 Conforming to the Specification

A smart card service provider module implementation that claims conformance to the GSC-IS must implement each of the following:

- The Architectural Model, as defined in Chapter 2
- The Access Control Model, as defined in Chapter 3
- The Basic Services Interface, as defined in Chapter 4
- The Virtual Card Edge Interface, as defined in Chapter 5
- The Card Capabilities Container, as defined in Chapter 6
- Container Naming, as defined in Chapter 7
- One of the Container Data Models defined in Chapter 8 and the appropriate Appendix
- At least one language binding for BSI Services, as defined in the Appendices.

As used in this document, the conformance keywords “shall” and “must” (which are interchangeable) denote mandatory features of the GSC-IS. The keyword “should” denotes a feature that is recommended but not mandatory, while the keyword “may” denotes a feature whose presence or absence does not preclude conformance.

2. Architectural Model

2.1 Overview

The GSC-IS provides interoperability at two levels: the service call level and the card command (APDU) level. A brief explanation of these interoperability levels follows:

- **Service Call Level**: This level is concerned with functional calls required to obtain various services from the card (e.g., encryption, authentication, digital signatures, etc.). The GSC-IS addresses interoperability at this level by defining an Applications Programming Interface (API) called the Basic Services Interface (BSI) that defines a common high level model for smart card services. The module that implements the BSI and thus provides an interoperable set of smart card services to client applications is called the Smart Card Service Provider Module (SCSPM). These services are logically divided into three modules that provide utility, secure data storage, and cryptographic services. Since an SCSPM could potentially be implemented through a combination of hardware and software, the software component of the SCSPM is referred to as the Service Provider Software (SPS).
- **Card Command Level**: This level is concerned with the exact APDU (ISO4) that are sent to the card to obtain the required service. The GSC-IS addresses interoperability at this level by defining the API called the Virtual Card Edge Interface (VCEI) that consists of a card-independent standard set of APDUs that support the functions defined in the BSI and implemented by the SCSPM.

Certain data sets need to be available in the card to support the interoperability provided by the BSI and VCEI. To ensure that there is a standard format (or schema) for storing these data sets, and to enable uniform access and interpretation, the GSC-IS also defines Data Models. These Data Models provide data portability across GSC-IS compliant card implementations, ensuring that a core set of data elements is available on all cards. The storage entities for various categories of data sets are called containers. One of these containers, the Card Capabilities Container (CCC), is mandatory, while the other containers comprising a Data Model are optional. The CCC describes the differences between a smart card's native APDU set and the standard APDU set defined by the VCEI. An SPS retrieves a smart card's CCC and uses it to perform the translation between the VCEI and the card's native APDU set. The GSC-IS accommodates any smart card whose APDU set can be mapped to the VCEI via a CCC definition.

The components of the GSC-IS architecture are presented in **Figure 2–1: The GSC-IS Architectural Model** and are further described in the following sections. All objects below the client application layer are components of the SCSPM.

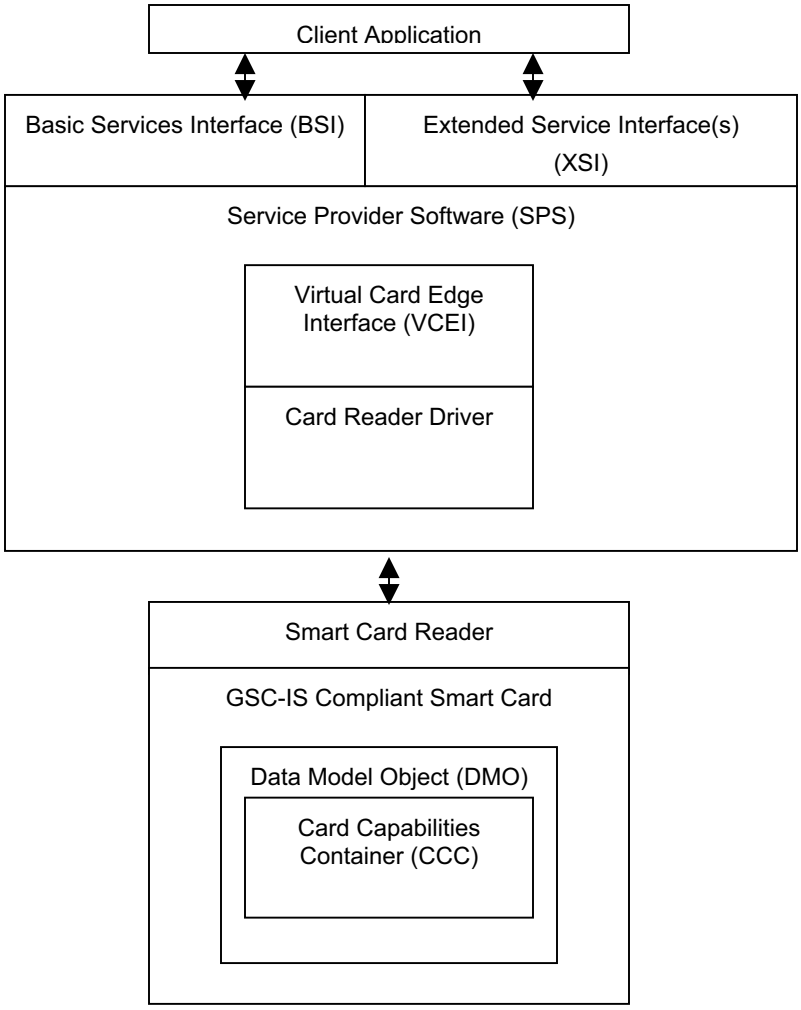


Figure 2–1: The GSC-IS Architectural Model

2.2 Basic Services Interface

All Smart Card Service Provider Modules shall implement the BSI. The BSI is logically organized into three provider modules:

- **Utility Provider Module:** Provides utility services for obtaining a list of available card readers, establishing and terminating logical connections with a smart card, etc.
- G**eneric Container Provider Module:** Provides a unified abstraction of the storage services of smart cards, presenting applications with a simple interface for managing generic containers of data elements in Tag/Length/Value format.
- **Cryptographic Provider Module:** Provides fundamental cryptographic services such as random number generation, authentication, and digital signature generation.

The capabilities of a given SCSPM depend on the smart card available to the SCSPM when a client application requests a service through a BSI call. In cases where a service is not available, the BSI call shall return an error code indicating that the requested service is not available. For example, a user may

insert a smart card that does not have public key cryptographic capabilities and then perform an operation that causes a client application to request a digital signature calculation from the associated SCSPM. Since the smart card cannot provide this service, the BSI shall return a "service not available" error code to the client application.

2.3 Extended Service Interfaces

Because the BSI is not a complete operational interface, real world SCSPM implementations must support additional functionality outside the BSI domain. Because the BSI provides an interoperable interface, it is unable to address the varying operational requirements. Therefore, real world SCSPM implementations must support additional functionality outside the BSI domain. An SCSPM may therefore include an Extended Service Interface (XSI) that provides non-interoperable functions. Since XSIs are implementation and applications specific, they are accommodated by the GSC-IS architectural model but are not defined in the GSC-IS. Card initialization and cryptographic key management are examples of functions that must currently be implemented in the XSI domain.

2.4 Virtual Card Edge Interface

ISO 7816-4 defines a hierarchical file system structure for smart cards. Smart Cards that conform to ISO 7816-4 are therefore known as "file system" cards. The Card Operating System program of a file system card is usually hard coded into the logic of the smart card integrated circuit during the manufacturing process and cannot be changed thereafter.

In recent years other smart card architectures have been created that allow developers to load executable programs onto smart cards after the cards have been manufactured. As one example, JavaCard™ (JAV) defines a Java Virtual Machine (VM) specification for smart card processors. Developers can load compiled Java applets onto a smart card containing the JavaCard™ VM, programmatically changing the behavior of the card.

Due to the widespread adoption of the JavaCard™ specification, the term "virtual machine smart card" is often used generically to refer to any smart card whose Card Operating System can be extended by loading executable programs onto the card (regardless of whether that card conforms to the JavaCard™ specification). This Specification uses the term "virtual machine smart card" in the general sense. A virtual machine smart card can theoretically be programmed to support any communications protocol, including the APDU based protocols of the ISO 7816-4 standard.

The VCEI defines default sets of interoperable APDU level commands for both virtual machine and file system smart cards. The SPS of an SCSPM shall use the information provided by a smart card's CCC to map that card's native APDU set to the VCEI default set. The VCEI shall consist of:

- A card edge definition for file system cards
- A card edge definition for VM cards, composed of three providers:
  - A generic container provider
  - A symmetric key (SKI) cryptographic service provider
  - A public key infrastructure (PKI) cryptographic service provider.

The card level providers of the VCEI directly support the service provider modules of the BSI. Card level providers are concrete implementations of the services that comprise the VCEI, and are physically implemented on GSC compliant smart cards.

2.5 Roles of the BSI and VCEI in Interoperability

The service provider modules of the BSI are a higher level abstraction of the card level providers. Standardization at the VCEI layer establishes interoperability between any GSC-compliant SPS and any GSC-compliant smart card. Similarly, standardization at the BSI layer establishes interoperability between any GSC compliant application and any GSC compliant SPS. Vendor neutrality is assured because GSC smart cards are interchangeable at the VCEI and GSC SPSs are interchangeable at both the BSI and VCEI.

2.6 GSC Data Models

Each GSC-IS compliant smart card shall conform to a GSC data model. GSC data models define the set of containers and data elements within each container for cards supporting that data model. This version of the GSC-IS defines two data models: the original J.8 data model from the GSC-IS v1.0 (Appendix C), and the Department of Defense Common Access Card data model (Appendix D). The Card Capabilities Container is the only mandatory container in either data model. The remaining containers and data elements are optional. However, if an implementation requires any of the containers and data elements defined in the data models, the containers and data elements must conform to the data model definitions. Data model requirements are presented in Chapter 8.

Containers shall be accessed through the Generic Container Provider Module of the BSI. Access to the containers shall be subject to the Access Control Rules (ACR) of the GSC-IS Security Model.

2.6.1 Card Capabilities Container

Each GSC-IS compliant card shall carry a Card Capabilities Container (CCC). The CCC is one of the mandatory containers that must be present in all GSC data models. The purpose of the CCC is to describe the differences between a given card's APDU set and the APDU set defined by the GSC-IS Virtual Card Edge Interface. The GSC-IS provides standard mechanisms for retrieving a CCC from a smart card (Section 6.2). Once the CCC for a particular card is obtained, software on the host computer (specifically, the SPS) uses this information to translate between the VCEI and the card's native APDU set. Deviations from the card's data model structure can also be represented in a CCC.

The CCC allows each GSC-IS smart card to carry the information needed by the SPS to communicate with that card. This general mechanism for dynamically translating APDU sets eliminates the need to distribute, install, and maintain card specific APDU level drivers on host computer systems.

The rules for constructing a valid CCC are defined in Section 6.3. All GSC-IS smart cards shall contain a CCC that conforms to this specification.

2.7 Service Provider Software

The SPS component of an SCSPM shall implement the BSI and the VCEI. It is responsible for retrieving CCCs from cards, using this information to translate between the smart card's native APDU set and the VCEI, and for handling the details of APDU level communications with the card. SPS implementations work with a particular card reader driver layer that transports APDUs between the SPS and the smart card.

2.8 Card Reader Drivers

The GSC-IS does not address interoperability between smart card readers and host computer systems. Several specifications already exist in this area, including the Personal Computer Smart Card (PC/SC, [PCSC]) specification and the OpenCard Framework (OCF, [OCF]). The choice of card reader driver software is influenced to some degree by the operating environment, although PC/SC and OCF have been ported to various operating systems.

Because card reader driver solutions are available and several of these have been widely adopted, the GSC-IS allows developers the freedom to choose any card reader driver that provides the reader level services required by the SPS layer:

- Transport of "raw" (unprocessed) APDUs between the SPS layer and the smart card
- Functions to provide a list of available readers, and to establish and terminate logical connections to cards inserted into readers.

Proprietary card reader drivers can also be used as long as they provide the raw APDU transport and card reader management functions required by an SPS. Some applications may have unique requirements that mandate a special purpose card reader. For example, the configuration required by a physical access control application may not be able to accommodate a PC/SC or OCF card reader driver layer and would therefore require a custom card reader driver.

The decision not to include a card reader driver layer specification in the GSC-IS has important consequences. This implies a pair-wise relationship between an SPS and the card reader driver. An SPS implementation works with a specific card reader driver, and is constrained to operate with the card readers supported by that driver. The degree of interoperability between card readers and host computer systems is entirely determined by the card reader driver component.

In cases where an industry standard card reader driver component is chosen, it is possible to take advantage of existing conformance test programs and select from a range of commercially available, compliant card readers. If a special purpose (proprietary) card reader driver is chosen, these options may not be available. In some cases proprietary card reader drivers work only with proprietary card reader designs, and may therefore require development of special purpose conformance test programs.

THIS PAGE INTENTIONALLY LEFT BLANK.

3. Access Control Model

The smart card services and containers provided by a SCSPM are subject to a set of ACRs. ACRs are defined for each card service and default container when a GSC-IS-compliant smart card is initialized. The card level service providers are responsible for enforcing these ACRs, and shall not provide a given service until the client application has fulfilled the applicable access control requirements. The GSC-IS specifies a discovery mechanism that allows client applications to determine the ACRs for a specific service provider or container.

It is important to note that an SPS acts as a transport and reformatting mechanism for the exchange of authentication data, such as PINs and cryptograms, between client applications and smart cards. When a client application and smart card service provider establish a security context, the primary job of the SPS is to reformat BSI level authentication structures into APDU level VCEI structures and vice versa. The current GSC-IS model does not include a mechanism for authenticating an SPS, and the SPS is not responsible for enforcing ACRs.

3.1 Available Access Control Rules

The ACRs available at the BSI level are as follows:

1. **Always**: The corresponding service can be provided without restrictions.
2. **Never**: The corresponding service can never be provided.
3. **External Authenticate**: The corresponding service can be provided only after a "Get Challenge" APDU.
4. **PIN Protected**: The corresponding service can be provided only if its associated personal identification number (PIN) code has been verified prior to the service request.
5. **PIN Always**: The corresponding service can be provided only if its associated PIN code has been verified immediately prior to the service request.
6. **External Authenticate or PIN**: Either one of the two controls gives access to the service. This allows for a cardholder validation when a PIN pad is available and for an external authentication when no PIN pad is available. Or, this provides an authentication method when the application cannot be trusted to perform an external authentication and to protect the external authentication key.
7. **External Authenticate then PIN**: The two methods must be chained successfully before access to the service is granted. This allows the authentication of both the client application and the user.
8. **Secure Channel (GP)**: the corresponding service can be provided through a Secure Channel managed by a Global Platform (GLOB) Secure Messaging layer.
9. **Update Once**: A target object can only be updated once during its lifetime.
10. **PIN then External Authenticate**: PIN presentation followed by an External Authentication.
11. **Secure Channel (ISO)**: the corresponding service can be provided through a Secure Channel managed by an ISO [ISO4] Secure Messaging layer.

BSI-level ACRs and associated hexadecimal values are summarized in the following table (Table 3-1). Hexadecimal values are assigned to the unACRtype member of the BSIAuthenticator structure defined in Section 4.5.1

Table 3–1: BSI Access Control Rules

Access Control Rules	Value	Meaning
BSI_ACR_ALWAYS	0x00	No access control rule is required
BSI_ACR_NEVER	0x01	Operation is never possible
BSI_ACR_XAUTH	0x02	External Authentication.
BSI_ACR_XAUTH_OR_PIN	0x03	The object method can be accessed either after an External Authentication or after a successful PIN presentation
BSI_SECURE_CHANNEL_GP	0x04	Secure Channel (Global Platform)
BSI_ACR_PIN_ALWAYS	0x05	PIN must be verified immediately prior to service request.
BSI_ACR_PIN	0x06	PIN code is required
BSI_ACR_XAUTH_THEN_PIN	0x07	External Authentication followed by a PIN presentation
BSI_ACR_UPDATE_ONCE	0x08	The target object can only be updated once during its lifetime
BSI_ACR_PIN_THEN_XAUTH	0x09	PIN presentation followed by External Authentication
BSI_SECURE_CHANNEL_ISO	0x0B	Secure Channel (ISO 7816-4)
Reserved for future use	0x0C-0xFF	

The External Authentication method shall comply with ISO 7816-4 [ISO4]. The mandated cryptographic algorithm is `DES3-ECB` [DES], with a double length key-size 16 Bytes, and a challenge of 8 Bytes. This method is described in Section 3.3.2.

The ACR for the Secure Channel implies cryptographic operations performed at the APDU level. A pass-through function is provided in the BSI (Section 4.5.9) to allow applications to create a secure channel and operate inside this channel.

3.2 Discovery Mechanism for ACRs

Applications can discover the ACR that must be fulfilled to access a specific service or container. ACR discovery mechanisms are defined for each provider module as follows:

- **Utility Service Provider Module:** No access control is applied.
- **Generic Container Service Provider Module:** ACRs for generic container services are encoded in the GCacr structure returned by the function `gscBsiGcGetContainerProperties()`. Read and Write ACRs for a specific container/file can be obtained from the Applications CardURL (Section 7.1) associated with the target container.
- **Cryptographic Service Provider Module:** ACRs for cryptographic services are encoded in the CRYPTOacr structure returned by the function `gscBsiGetCryptoProperties()`.

Each of the services associated with a provider module have a different set of allowable ACRs. When a provider module is created (instantiated), the module creator must assign the ACRs for each of the

services provided by the module from the set of supported ACRs, listed in the following tables (Tables 3-2 and 3-3).

Table 3–2: ACRs for Generic Container Provider Module Services

Service	ACR supported

`gscBsiGcDataCreate()`	`BSI_ACR_ALWAYS`
	`BSI_ACR_NEVER`
	`BSI_ACR_PIN`
	`BSI_ACR_XAUTH`

`gscBsiGcDataDelete()`	`BSI_ACR_ALWAYS`
	`BSI_ACR_NEVER`
	`BSI_ACR_PIN`
	`BSI_ACR_XAUTH`

`gscBsiGcReadTagList()`	`BSI_ACR_ALWAYS`
	`BSI_ACR_PIN`
	`BSI_ACR_XAUTH`

`gscBsiGcReadValue()`	`BSI_ACR_ALWAYS`
	`BSI_ACR_PIN`
	`BSI_ACR_XAUTH`

`gscBsiGcUpdateValue()`	`BSI_ACR_ALWAYS`
	`BSI_ACR_NEVER`
	`BSI_ACR_PIN`
	`BSI_ACR_XAUTH`

`gscBsiGcGetContainerProperties()`	`BSI_ACR_ALWAYS`

Table 3–3: ACRs for Cryptographic Provider Module Services

Service	ACR supported
`gscBsiGetChallenge()`	`BSI_ACR_ALWAYS`

`gscBsiSkiInternalAuthenticate()`	`BSI_ACR_ALWAYS`
	`BSI_ACR_PIN`
	`BSI_ACR_XAUTH`

`gscBsiPkiCompute()`	`BSI_ACR_ALWAYS`
	`BSI_ACR_PIN`
	`BSI_ACR_XAUTH`

`gscBsiPkiGetCertificate()`	`BSI_ACR_ALWAYS`
	`BSI_ACR_PIN`
	`BSI_ACR_XAUTH`

`gscBsiGetCryptoProperties()`	`BSI_ACR_ALWAYS`

3.3 Establishing a Security Context

Once a client application has determined the ACR associated with a service or a container, it must establish a security context with the card. To fulfill the ACR for a container or service, the application builds a BSIAuthenticator data structure and passes it in a call to the `gscBsiUtilAcquireContext()` function.

Establishing a security context involves authentication of the parties involved in the service exchange. These parties include the human user executing the client application, the client application, and the smart card. The GSC-IS' ACRs are based on three general authentication mechanisms: PIN Verification, External Authentication, and Secure Messaging.

External Authentication method assumes that the authentication key has been formerly distributed to both parties (client application and smart card) in a secure way.

The following three Sections (Section 3.3.1 through Section 3.3.3) describe typical BSI call sequences that a client application would use for each of the three authentication mechanisms, in order to acquire the context for the desired smart card service.

It is important to note that at the smart card level, the read and write privileges are granted sequentially therefore each granted privilege and its acquired security context is associated with either a read or a write operation through the member bWriteKey of the BSIAuthenticator structure. The structure is passed to the SCSPM by the client application in the `gscBsiUtilAcquireContext()` call. Prior to acquiring a new privilege, the client application shall release the previously acquired security context, if any exists, by calling the BSI's function `gscBsiUtilReleaseContext()`.

3.3.1 PIN Verification

For a PIN Verification known also as Card Holder Verification (CHV), the client application would make the following calls:

1. Establish a logical connection with the card through a call to the BSI's function `gscBsiUtilConnect()`.
2. Retrieve the ACRs for a desired card service through a call to either `gscBsiGcGetContainerProperties()` or `gscBsiGetCryptoProperties()`. These interface methods return the ACRs for all services available from the smart card (Sections 4.6.3 or 4.7.5, respectively). If PIN Verification is required for a particular service (i.e., `gscBsiGcReadValue()` or `gscBsiPkiCompute()`), the ACR returned in the GCacr or CRYPTOacr structure for this service must be `BSI_ACR_PIN`.
3. Call `gscBsiUtilAcquireContext()` with the BSIAuthenticator structure required to satisfy the ACR for the desired smart card service. In this example, for PIN verification, the BSI Authenticator structure shall contain the PIN value in the usAuthValue field and unACRtype set to `BSI_ACR_PIN`.
4. Access the desired smart card service through subsequent BSI calls.
5. Call `gscBsiUtilReleaseContext()` to release the security context.

3.3.2 External Authentication

A typical BSI sequence of calls for an External Authentication is described next:

1. Establish a logical connection with the card through a call to `gscBsiUtilConnect()`.
2. Retrieve the ACRs for a desired card service provider through a call to either `gscBsi GcGetContainerProperties()` or `gscBsiGetCryptoProperties()`. These interface methods return the ACRs for all services available from the smart card (Section 4.6.3 or Section 4.7.5, respectively). If External Authentication is required for a particular service (i.e., `gscBsiGcReadValue()` or `gscBsiPkiCompute()`), the ACR returned in the `GCacr` or `CRYPTOacr` structure for this service must be `BSI_ACR_XAUTH`.

3. Call `gscBsiGetChallenge()` to retrieve a random challenge from the smart card. The random challenge is retained by the smart card for use in the subsequent verification step of the External Authentication protocol. The client application calculates a cryptographic Authenticator by encrypting the random challenge using a symmetric External Authentication key. The client application may need to examine the KeyID field of the appropriate Application CardURL (see Section 7.1) to determine which External Authentication key it should use to encrypt the random challenge.

4. The client application calls the BSI's `gscBsiUtilAcquireContext()` function passing the cryptographic Authenticator computed in step 3.

5. The smart card decrypts the Authenticator using its External Authentication key, and verifies that the resulting plaintext value matches the original random challenge value. The External Authentication key shall be formerly, securely distributed to the client application and to the smart card.

6. Access the desired smart card service through subsequent BSI calls.

7. Call `gscBsiUtilReleaseContext()` to release the security context.

3.3.3 Secure Messaging

Secure messaging involves the establishment of a secure channel between the client application and the smart card at the APDU level. The BSI provides a pass-through call and the SPS shall implement it such that it allows a client application to establish a direct APDU level secure channel with a card in accordance with the Global Platform [GLOB] or ISO 7816-4 [ISO4].

THIS PAGE INTENTIONALLY LEFT BLANK.

4. Basic Services Interface

4.1 Overview

A SCSPM must provide a BSI. Client applications communicate with the SCSPM through this interface. The SPS component of the SCSPM is directly responsible for implementing the BSI. In the context of this section, the terms "BSI" and "SPS" (i.e., BSI implementation) will be used interchangeably.

This chapter defines the BSI services, using the Interface Definition Language (IDL). The set of services consists of 21 functions grouped into three functional modules as follows:

A Smart Card Utility Provider Module:

- `gscBsiUtilAcquireContext()`
- `gscBsiUtilConnect()`
- `gscBsiUtilDisconnect()`
- `gscBsiUtilGetVersion()`
- `gscBsiUtilGetCardProperties()`
- `gscBsiUtilGetCardStatus()`
- `gscBsiUtilGetExtendedErrorText()`
- `gscBsiUtilGetReaderList()`
- `gscBsiUtilPassthru()`
- `gscBsiUtilReleaseContext()`

A Smart Card Generic Container Provider Module:

- `gscBsiGcDataCreate()`
- `gscBsiGcDataDelete()`
- `gscBsiGetContainerProperties()`
- `gscBsiGcReadTagList()`
- `gscBsiGcReadValue()`
- `gscBsiGcUpdateValue()`

A Smart Card Cryptographic Provider Module:

- `gscBsiGetChallenge()`
- `gscBsiSkiInternalAuthenticate()`

- `gscBsiPkiCompute()`
- `gscBsiPkiGetCertificate()`
- `gscBsiGetCryptoProperties()`

All SCSPM implementations must provide the full set of 21 functions as specified in this chapter. Based on the capabilities of the smart card being used, a given function call may return a `BSI_NO_CARDSERVICE` error message in case the smart card does not provide the requested service. This error message may be returned by any BSI function that maps directly to a card-level operation, as follows:

- `gscBsiUtilGetCardProperties()`
- `gscBsiGcDataCreate()`
- `gscBsiGcDataDelete()`
- `gscGcGetContainerProperties()`
- `gscBsiGcReadTagList()`
- `gscBsiGcReadValue()`
- `gscBsiGcUpdateValue()`
- `gscBsiGetChallenge()`
- `gscBsiSkiInternalAuthenticate()`
- `gscBsiPkiCompute()`
- `gscBsiPkiGetCertificate()`
- `gscBsiGetCryptoProperties()`

Extensions to the BSI, in the form of an XSI (see Section 2.3), may be present in an implementation to allow additional functionality. The functions in such an XSI shall not alter the specified behavior or semantics of the BSI functions in that implementation.

ACRs for each provider module are defined in Section 3.2, Table 3–1, Table 3–2 and Table 3–3. The following sections define all the BSI return codes and the 21 functions of the BSI, using IDL.

4.2 Binary Data Encoding

Some BSI functions accept or return binary data, such as cryptograms. Binary data shall be encoded in ASCII hexadecimal string format. For example, the sequence of bytes 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F would be encoded in the NULL terminated ASCII hexadecimal string “0102030405060708090A0B0C0D0E0F”.

4.3 Mandatory Cryptographic Algorithms

The following cryptographic algorithms and associated algorithm identifiers are mandatory for all GSC smart cards. Cards may optionally support other algorithms:

- Algorithm Identifier "0x81": `DES3-ECB`, with a double length key-size, 16 Bytes.
- Algorithm Identifier "0xA3": `RSA_NO_PAD`, computation on the private key, Chinese Remainder.
- Algorithm Identifier "0x82": `DES3-CBC`, with a double length key-size, 16 Bytes.

4.4 BSI Return Codes

Table 4–1 lists all possible errors that BSI functions could return. For each function description (Sections 4.5.1 to 4.7.5), error codes are listed in order of precedence, except for the successful return with `BSI_OK`.

Table 4–1: BSI Return Codes

Label	Return Code Hexadecimal Value	Meaning
BSI_OK	0x00	Execution completed successfully.
BSI_ACCESS_DENIED	0x01	The applicable ACR was not fulfilled.
BSI_ACR_NOT_AVAILABLE	0x02	The specified ACR is incorrect.
BSI_BAD_AID	0x03	The specified Application Identifiers (AID) does not exist.
BSI_BAD_ALGO_ID	0x04	The specified cryptographic algorithm is not available.
BSI_BAD_AUTH	0x05	Invalid authentication data.
BSI_BAD_HANDLE	0x06	The specified card handle is not available.
BSI_BAD_PARAM	0x07	One or more of the specified parameters is incorrect.
BSI_BAD_TAG	0x08	Invalid tag information.
BSI_CARD_ABSENT	0x09	The card associated with the specified card handle is not present.
BSI_CARD_REMOVED	0x0A	The card associated with the specified card handle has been removed.
BSI_CREATE_ERROR	0x0B	Error encountered while trying to create the specified data.
BSI_DELETE_ERROR	0x0D	Error encountered while trying to delete the specified data.
BSI_INSUFFICIENT_BUFFER	0x0E	The buffer allocated by the calling application is too small.
BSI_NO_CARDSERVICE	0x0F	The smart card associated with the specified card handle does not provide the requested service.
BSI_NO_MORE_SPACE	0x10	There is insufficient space in the selected container to store the specified data.
BSI_PIN_LOCKED	0x11	The PIN is locked.

Label	Return Code	Meaning
BSI_READ_ERROR	0x12	Error encountered while attempting to read the specified data.
BSI_TAG_EXISTS	0x13	The tag specified for a create operation already exists in the target container.
BSI_TIMEOUT_ERROR	0x14	A connection could not be established with the card before the timeout value expired.
BSI_TERMINAL_AUTH	0X15	The card reader has performed a successful Internal Authentication with the card.
BSI_NO_TEXT_AVAILABLE	0x16	No extended error text is available.
BSI_UNKNOWN_ERROR	0x17	The requested operation has generated an unspecified error.
BSI_UNKNOWN_READER	0x18	The specified reader does not exist.
BSI_UPDATE_ERROR	0x19	Error encountered while attempting to update the specified data.

4.5 Smart Card Utility Provider Module Interface Definition

4.5.1 gscBsiUtilAcquireContext()

Purpose: This function shall establish a session with the smart card by submitting the appropriate Authenticator in the BSIAuthenticator structure. For ACRs requiring external authentication (XAUTH), the `uszAuthValue` field of the BSIAuthenticator structure must contain a cryptogram calculated by encrypting a random challenge.

The calling application passes a PIN and the appropriate External Authentication cryptogram in two BSIAuthenticator structures for ACRs that require chained authentication, such as `BSI_ACR_PIN_THEN_XAUTH`. The client application must set the `unACRtype` field of each BSIAuthenticator structure to match the type of authenticator contained in the structure. To satisfy an ACR of `BSI_ACR_PIN_THEN_XAUTH`, the application would construct an array of two BSIAuthenticators: one containing a PIN and one containing an External Authentication cryptogram. The BSIAuthenticator structure containing the PIN would have an `unACRtype` of `BSI_ACR_PIN`, and the BSIAuthenticator structure containing the External Authentication cryptogram would have an `unACRtype of BSI_ACR_XAUTH.`

Prototype:

```
unsigned long gscBsiUtilAcquireContext(
   IN UTILCardHandle    hCard,
   IN string            uszAID,
   IN sequence          <BSIAuthenticator>strctAuthenticator,
   IN unsigned int      unAuthNb
);
```

Parameters:

`hCard`: Card connection handle from **`gscBsiUtilConnect()`.**

`uszAID`: Target container AID value.

`strctAuthenticator`: An array of structures containing the authenticator(s) specified by the ACR required to access a value in the container. The authenticator is returned by **`gscBsiGcGetContainerProperties()`**. The calling application is responsible for allocating this structure.

`unAuthNb`: Number of authenticator structures contained in **`strctAuthenticator.`**

The BSIAuthenticator structure is defined as follows:

```
struct BSIAuthenticator {
   unsigned long        unACRtype;
   string               uszAuthValue;
   string               uszKeyValue;
   boolean              bWriteKey;
};
```

Variables associated with the BSIAuthenticator structure:

unACRtype:	Access Control Rule (see Table 3-1 in Section 3.1).
usAuthValue:	Authenticator, can be an external authentication cryptogram or PIN.
uszKeyValue:	Cryptographic authentication key..
bWriteKey:	At the smart card level, the read and write privileges are granted sequentially therefore each granted privilege and its acquired security context is associated with either a read or a write operation through this member of the BSIAuthenticator. If set to TRUE, this ACR controls write access. If set to FALSE, it controls read access.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_ACR_NOT_AVAILABLE
BSI_BAD_AUTH
BSI_CARD_REMOVED
BSI_PIN_LOCKED
BSI_UNKNOWN_ERROR
```

4.5.2 gscBsiUtilConnect()

Purpose: Establish a logical connection with the card in a specified reader. `BSI_TIMEOUT_ERROR` will be returned if a connection cannot be established within a specified time. The timeout value is implementation dependent.

Prototype:

```
unsigned long gscBsiUtilConnect(
   IN string            uszReaderName,
   OUT UTILCardHandle   hCard
);
```

Parameters:

hCard:	Card connection handle.
uszReaderName:	Name of the reader that the card is inserted into. If this field is a NULL pointer, the SPS shall attempt to connect to the card in the first available reader, as returned by a call to the BSI's function **gscBsiUtilGetReaderList()**.

Return Codes:

```
BSI_OK
BSI_BAD_PARAM
BSI_UNKNOWN_READER
BSI_CARD_ABSENT
BSI_TIMEOUT_ERROR
```

4.5.3 gscBsiUtilDisconnect()

Purpose: Terminate a logical connection to a card.

Prototype:
```
unsigned long gscBsiUtilDisconnect(
   IN UTILCardHandle    hCard
);
```

Parameters: **hCard**: Card connection handle from **gscBsiUtilConnect()**.

Return Codes:
```
BSI_OK
BSI_BAD_HANDLE
BSI_CARD_REMOVED
BSI_UNKNOWN_ERROR
```

4.5.4 gscBsiUtilGetVersion()

Purpose: Returns the BSI implementation version.

Prototype:

```
unsigned long gscBsiUtilGetVersion(
   INOUT string         uszVersion
);
```

Parameters:

uszVersion:	The BSI and SCSPM version formatted as "major,minor,revision, build_number\0". The value for an SCSPM compliant with this version of the GSC-IS is "2,0,0,<build number>\0". The build number field is vendor/implementation dependent.

Return Codes:

```
BSI_OK
BSI_INSUFFICIENT_BUFFER
BSI_UNKNOWN_ERROR
```

4.5.5 gscBsiUtilGetCardProperties()

Purpose: Retrieves version and capability information for the card.

Prototype:
```
unsigned long gscBsiUtilGetCardProperties(
   IN UTILCardHandle    hCard,
   INOUT string         uszCCCUniqueID,
   OUT unsigned long    unCardCapability
);
```

Parameters:

hCard: Card connection handle from **gscBsiUtilConnect()**.

uszCCCUniqueID: Buffer for the Card Capability Container version .

unCardCapability: Bit mask value defining the providers supported by the card. The bit masks represent the Generic Container Data Model, the Generic Container Data Model Extended, the Symmetric Key Interface, and the Public Key Interface providers respectively:

```
#define BSI_GCCDM      0x00000001
#define BSI_SKI        0x00000002
#define BSI_PKI        0x00000004
#define BSI_GCCDM_EXT  0x00000008
#define BSI_SKI_EXT    0x00000010
#define BSI_PKI_EXT    0x00000020
```

Return Codes:
```
BSI_OK
BSI_BAD_HANDLE
BSI_CARD_REMOVED
BSI_INSUFFICIENT_BUFFER
BSI_NO_CARDSERVICE
BSI_UNKNOWN_ERROR
```

4.5.6 gscBsiUtilGetCardStatus()

Purpose: Checks whether a given card handle is associated with a card that is inserted into a powered up reader.

Prototype:
```
unsigned long gscBsiUtilGetCardStatus(
   IN UTILCardHandle    hCard
);
```

Parameters: **hCard**: Card connection handle from **gscBsiUtilConnect()**.

Return Codes:
```
BSI_OK
BSI_BAD_HANDLE
BSI_CARD_REMOVED
BSI_UNKNOWN_ERROR
```

4.5.7 gscBsiUtilGetExtendedErrorText()

Purpose: When a BSI function call returns an error, an application can make a subsequent call to this function to receive additional error information from the card reader driver layer, if available. Since the GSC-IS architecture accommodates different card reader driver layers, the error text information will be dependent on the card reader driver layer used in a particular implementation. This function must be called immediately after the error has occurred.

Prototype:

```
unsigned long gscBsiUtilGetExtendedErrorText(
   IN UTILCardHandle    hCard,
   OUT char             uszErrorText[255]
);
```

Parameters:

hCard: Card connection handle from **gscBsiUtilConnect()**.

uszErrorText: A fixed length buffer containing an implementation specific error text string. The text string is NULL-terminated, and has a maximum length of 255 characters including the NULL terminator. The calling application must allocate a buffer of 255 bytes. If an extended error text string is not available, this function returns a NULL string and the return code BSI_NO_TEXT_AVAILABLE.

Return Codes:

BSI_OK
BSI_BAD_HANDLE
BSI_NO_TEXT_AVAILABLE
BSI_UNKNOWN_ERROR

4.5.8 gscBsiUtilGetReaderList()

Purpose: Retrieves the list of available readers.

Prototype:
```
unsigned long gscBsiUtilGetReaderList(
   INOUT string         uszReaderList
);
```

Parameters:

uszReaderList:	Reader list buffer. The reader list is returned as a multi-string, each reader name terminated by a '\0'. The list itself is terminated by an additional trailing '\0' character.

Return Codes:
```
BSI_OK
BSI_INSUFFICIENT_BUFFER
BSI_UNKNOWN_ERROR
```

4.5.9 gscBsiUtilPassthru()

Purpose: Allows a client application to send a "raw" APDU through the BSI directly to the card and receive the APDU-level response.

Prototype:

```
unsigned long gscBsiUtilPassthru(
   IN UTILCardHandle    hCard,
   IN string            uszCardCommand,
   INOUT string         uszCardResponse
);
```

Parameters:

hCard: Card connection handle from **gscBsiUtilConnect()**.

uszCardCommand: The APDU to be sent to the card.

uszCardResponse: Pre-allocated buffer for the APDU response from the card. The response must include the status bytes SW1 and SW2 returned by the card. If the size of the buffer is insufficient, the SPS shall return truncated response data and the return code BSI_INSUFFICIENT_BUFFER.

Return Codes:

BSI_OK
BSI_BAD_HANDLE
BSI_BAD_PARAM
BSI_INSUFFICIENT_BUFFER
BSI_CARD_REMOVED
BSI_UNKNOWN_ERROR

4.5.10 gscBsiUtilReleaseContext()

Purpose: Terminate a session with the card.

Prototype:

```
unsigned long gscBsiUtilReleaseContext(
   IN UTILCardHandle    hCard,
   IN string            uszAID
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
uszAID:	Target container AID value.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_CARD_REMOVED
BSI_UNKNOWN_ERROR
```

4.6 Smart Card Generic Container Provider Module Interface Definition

4.6.1 gscBsiGcDataCreate()

Purpose: Create a new data item in {Tag, Length, Value} format in the selected container.

Prototype:

```
unsigned long gscBsiGcDataCreate(
   IN UTILCardHandle    hCard,
   IN string            uszAID,
   IN GCtag             ucTag,
   IN string            uszValue
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
uszAID:	Target container AID value.
ucTag:	Tag of data item to store.
uszValue:	Data value to store.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_NO_MORE_SPACE
BSI_TAG_EXISTS
BSI_CREATE_ERROR
BSI_UNKNOWN_ERROR
```

4.6.2 gscBsiGcDataDelete()

Purpose: Delete the data item associated with the tag value in the specified container.

Prototype:
```
unsigned long gscBsiGcDataDelete(
   IN UTILCardHandle    hCard,
   IN string            uszAID,
   IN GCtag             ucTag
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
uszAID:	Target container AID value.
ucTag:	Tag of data item to delete.

Return Codes:
```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_TAG
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_DELETE_ERROR
BSI_UNKNOWN_ERROR
```

4.6.3 gscBsiGcGetContainerProperties()

Purpose: Retrieves the properties of the specified container. Access Control Rules are common to all data items managed by the selected container.

Prototype:

```
unsigned long gscBsiGcGetContainerProperties(
   IN UTILCardHandle    hCard,
   IN string            uszAID,
   OUT GCacr            strctGCacr,
   OUT GCContainerSize  strctContainerSizes,
   OUT string           uszContainerVersion
);
```

Parameters:

hCard: Card connection handle from **gscBsiUtilConnect()**.

uszAID: Target container AID value.

strctGCacr: Structure indicating access control conditions for all operations. The range of possible values for the members of this structure is defined in Table 3-1 (Section 3.1). The allowable ACRs for each function are listed in Table 3-2 (Section 3.2):

```
struct GCacr {
   unsigned long        unCreateACR;
   unsigned long        unDeleteACR;
   unsigned long        unReadTagListACR;
   unsigned long        unReadValueACR;
   unsigned long        unUpdateValueACR;
};
```

strctContainerSizes: For Virtual Machine cards, the size(in bytes) of the container specified by `uszAID`. `unMaxNbDataItems` is the size of the T-Buffer, and `unMaxValueStorageSize` is the size of the V-Buffer. For file system cards than cannot calculate these values, both fields of this structure will be set to 0.

```
struct GCContainerSize {
   unsigned long        unMaxNbDataItems;
   unsigned long        unMaxValueStorageSize;
}
```

uszContainerVersion: Version of the container. The format of this value is application dependent. In cases where the card cannot return a container version, this string will contain only the NULL terminator "\0".

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
```

```
BSI_UNKNOWN_ERROR
```

4.6.4 gscBsiGcReadTagList()

Purpose: Return the list of tags in the selected container.

Prototype:

```
unsigned long gscBsiGcReadTagList(
   IN UTILCardHandle    hCard,
   IN string            uszAID,
   INOUT sequence<GCtag>   TagArray
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
uszAID:	Target container AID value.
TagArray:	An array containing the list of tags for the selected container.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_INSUFFICIENT_BUFFER
BSI_UNKNOWN_ERROR
```

4.6.5 gscBsiGcReadValue()

Purpose: Returns the Value associated with the specified Tag.

Prototype:

```
unsigned long gscBsiGcReadValue(
   IN UTILCardHandle    hCard,
   IN string            uszAID,
   IN GCtag             ucTag,
   INOUT string         uszValue
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
uszAID:	Target container AID value.
ucTag:	Tag value of data item to read.
uszValue:	Value associated with the specified tag. The caller must allocate the buffer.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_TAG
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_INSUFFICIENT_BUFFER
BSI_READ_ERROR
BSI_UNKNOWN_ERROR
```

4.6.6 gscBsiGcUpdateValue()

Purpose: Updates the Value associated with the specified Tag.

Prototype:

```
unsigned long gscBsiGcUpdateValue(
   IN UTILCardHandle    hCard,
   IN string            uszAID,
   IN GCtag             ucTag,
   IN string            uszValue
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
uszAID:	Target container AID value.
ucTag:	Tag of data item to update.
uszValue:	New Value of the data item.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_BAD_TAG
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_NO_MORE_SPACE
BSI_UPDATE_ERROR
BSI_UNKNOWN_ERROR
```

4.7 Smart Card Cryptographic Provider Module Interface Definition

4.7.1 gscBsiGetChallenge()

Purpose: Retrieves a randomly generated challenge from the card as the first step of a challenge-response authentication protocol between the client application and the card. The client subsequently encrypts the challenge using a symmetric key and returns the encrypted random challenge to the card through a call to **`gscBsiUtilAcquireContext()`** in the `uszAuthValue` field of a BSIAuthenticator structure.

Prototype:

```
unsigned long gscBsiGetChallenge(
   IN UTILCardHandle    hCard,
   IN string            uszAID,
   INOUT string         uszChallenge
);
```

Parameters:

`hCard`:	Card connection handle from **`gscBsiUtilConnect()`**.
`uszAID`:	Target container AID value.
`uszChallenge`:	Random challenge returned from the card.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_INSUFFICIENT_BUFFER
BSI_UNKNOWN_ERROR
```

4.7.2 gscBsiSkiInternalAuthenticate()

Purpose: Computes a symmetric key cryptogram in response to a challenge. In cases where the card reader authenticates the card, this function does not return a cryptogram. In these cases a BSI_TERMINAL_AUTH will be returned if the card reader successfully authenticates the card. BSI_ACCESS_DENIED is returned if the card reader fails to authenticate the card.

Prototype:

```
unsigned long gscBsiSkiInternalAuthenticate(
   IN UTILCardHandle    hCard,
   IN string            uszAID,
   IN unsigned char     ucAlgoID,
   IN string            uszChallenge,
   INOUT string         uszCryptogram
);
```

Parameters:

hCard: Card connection handle from **gscBsiUtilConnect()**.

uszAID: SKI provider module AID value.

ucAlgoID: Identifies the cryptographic algorithm that the card must use to encrypt the challenge. All conformant implementations shall, at a minimum, support DES3-ECB (Algorithm Identifier 0x81) and DES3-CBC (Algorithm Identifier 0x82). Implementations may optionally support other cryptographic algorithms.

uszChallenge: Challenge generated by the client application and submitted to the card.

uszCryptogram: The cryptogram computed by the card.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_BAD_ALGO_ID
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_TERMINAL_AUTH
BSI_INSUFFICIENT_BUFFER
BSI_UNKNOWN_ERROR
```

4.7.3 gscBsiPkiCompute()

Purpose: Performs a private key computation on the message digest using the private key associated with the specified AID.

Prototype:

```
unsigned long gscBsiPkiCompute(
   IN UTILCardHandle    hCard,
   IN string            uszAID,
   IN octet             ucAlgoID,
   IN string            uszMessage,
   INOUT string         uszResult
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
uszAID:	PKI provider module AID value.
ucAlgoID:	Identifies the cryptographic algorithm that will be used to generate the signature. All conformant implementations shall, at a minimum, support RSA_NO_PAD (Algorithm Identifier 0xA3). Implementations may optionally support other algorithms.
uszMessage:	The message digest to be signed.
uszResult:	Buffer containing the signature.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_BAD_ALGO_ID
BSI_CARD_REMOVED
BSI_ACCESS_DENIED
BSI_NO_CARDSERVICE
BSI_INSUFFICIENT_BUFFER
BSI_UNKNOWN_ERROR
```

4.7.4 gscBsiPkiGetCertificate()

Purpose: Reads the certificate from the card.

Prototype:

```
unsigned long gscBsiPkiGetCertificate(
   IN UTILCardHandle    hCard,
   IN string            uszAID,
   INOUT string         uszCertificate
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
uszAID:	PKI provider module AID value.
uszCertificate:	Buffer containing the certificate.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_READ_ERROR
BSI_INSUFFICIENT_BUFFER
BSI_UNKNOWN_ERROR
```

4.7.5 gscBsiGetCryptoProperties()

Purpose: Retrieves the Access Control Rules associated with the PKI provider module.

Prototype:

```
unsigned long gscBsiGetCryptoProperties(
   IN UTILCardHandle    hCard,
   IN string            uszAID,
   OUT CRYPTOacr        strctCRYPTOacr
);
```

Parameters:

hCard: Card connection handle from **gscBsiUtilConnect()**.

uszAID: AID of the PKI provider.

strctCRYPTOacr: Structure indicating access control conditions for all operations. The range of possible values for the members of this structure are defined in Table 3-1 (Section 3.1), and the allowable ACRs for each function in Table 3-3 (Section 3.2):

```
struct CRYPTOacr {
   unsigned long        unGetChallengeACR;
   unsigned long        unInternalAuthenticateACR;
   unsigned long        unPkiComputeACR;
   unsigned long        unReadCertificateACR;
};
```

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_UNKNOWN_ERROR
```

THIS PAGE INTENTIONALLY LEFT BLANK.

5. Virtual Card Edge Interface

5.1 Overview

The VCEI supports the functionality of the BSI at the card edge (APDU) layer. The VCEI's primary function is to abstract the differences between heterogeneous card-level APDU sets. Because there are significant differences in the behavior of file system cards and VM cards at the card edge layer, the VCEI is divided into a Card Edge Interface for File System Cards (Section 5.2) and a Card Edge Interface for VM Cards (Section 5.3).

The terminology used for virtual machine cards is strongly influenced by the JavaCard™ world. However, the VCEI is compatible with any smart card capable of being programmed to implement the VCEI for Virtual Machine Cards.

5.2 Card Edge Interface for File System Cards

Table 5–1: APDU Set for File System Cards

APDU Set for File System Cards								
FC	**Card Function**	**CLA**	**INS**	**P1**	**P2**	**P3**	**Data**	**Notes**
1	Select DF File under MF	00	A4	01	00	L (02)	File ID (2 Bytes)	See Note 1
2	Read Binary (Transparent)	00	B0	Off/H	Off/L	L	Response	AC must be fulfilled
3	Update Binary (Normal)	00	D6	Off/H	Off/L	L	Data to Update	AC must be fulfilled
4	Update Binary (Secure Msg)	84	D6	Off/H	Off/L	L+03	Plain Data + Cryptogram	See Note 3
5	RSA Compute	80	42	00	Key #	L	Message to sign/decrypt	
6	Get Challenge	00	84	00	00	L (08)	8 byte challenge from card	See Note 6
7	Get Response	00	C0	00	00	L	Data to retrieve	See Note 5
8	Verify CHV	00	20	00	CHV	L (08)		See Note 7
9	Internal Authenticate	00	88	AI	Key #	L(08)	8 byte challenge	See Note 4 (pre-Verify CHV) (post-Get Response)
A	External Authenticate	00	82	00	Key #	L (06)		See Note 8
B	Reserved Future Use	00						
C	Select Key	80	28	00	00	L(08)	8 byte random number	
D	Select EF File under selected DF	00	A4	02	00	L (02)	File ID (2 Bytes)	See Note 1
E	Select Master File (Root)	00	A4	03	00	L (2)	File ID (2 Bytes)	See Note 1
F	Read Binary (Transparent, Secure Messaging)	84	B0	Off/H	Off/L	L+03	Plain Data +Cryptogram	See Note 2

Notes:

1. P1 specifies selection type: EF with short file identifier (=00), select DF (=01), EF under currently select DF (=02), parent DF (=03), DF by its name. P2 specifies response required (=00) or no response required (=C0). L is data field length and depends on selection type. For short file selection, L=2. Data is short file identifier or name.
2. For secure messaging CLA=04. Data is read in plaintext but a cryptogram (3 Most Significant Bytes) is appended to it. Data is padded with 0's. Encryption is 3DES with temporary administration key.
3. Data is appended with the 3-Least-significant Bytes of the Cryptogram. Use Get Response after command to obtain 3 Most Significant Bytes of Cryptogram to optionally verify card checksum. Data is padded with 0's so that it is a multiple of 8. Encryption is 3DES based.
4. AI (Algorithm ID) selects DES (=00) or 3DES (=02), 512 RSA (=C5), 768 RSA (=C7), or 1024 RSA (=C9). Command is preceded by Verify CHV and followed by Get Response. No information about the size of the returned cryptogram is provided.
5. Length parameter depends on previous command issued to card.
6. Number lost if card reset, unsuccessful Get Challenge, External Authenticate
7. CHV specifies 01 to 0F, if key. Use 10 if CHV
8. Key # is from 01 to 0F. Data is 6 MSB of cryptogram from standard DES/ 3DES encryption of random challenge.

5.3 Card Edge Interface for VM Cards

The Card Edge Interface for VM Cards shall be made up of three provider modules that provide Generic Container management services, symmetric key cryptographic services, and public (asymmetric) key cryptographic services. These provider modules shall be implemented as on-card applets. For virtual machine cards, the terms "provider" and "applet" are synonymous.

Each provider class shall be responsible for applying ACRs to the services it provides, and some cards may not implement all three classes. A number of interface methods must therefore be duplicated across the three classes, since provider instantiations must be functionally independent. Common interface methods that must be implemented by all providers are described first, and the methods unique to each provider class are described in subsequent sections.

5.3.1 Virtual Machine Card Access Control Rule Configuration

Each smart card service provider shall present its services through a set of APDUs implemented and managed by the provider. The ACRs associated with card level services vary depending on the application. For this reason, the GSC-IS defines the range of possible Access Control Rules that can be assigned to a given card level service in the form of three Access Control Levels (ACL). An ACL must be assigned to each service offered by each smart card service provider when the card is initialized. These ACLs are defined for each Virtual Machine Card Edge Provider in Section 5.3.

The following table assigns a set of allowable authentication methods for each of the three ACLs. Cells with an "X" indicate that an implementer may select the corresponding authentication method when defining an ACL.

Table 5–2: ACL Table

ACL	Always	PIN	XAUTH	XAUTH then PIN	XAUTH or PIN	Secure Channel	Never
Level1	X	X	X	X	X	X	X
Level2	X	X					X
Level3	X	X	X	X	X	X	X

Security levels shall be coded as a single byte value (range 0x00 - 0xFF) as follows. The range 0x0C-0xFF is reserved for future use:

Table 5–3: Security Levels

Security Levels	Value
Always	0x00
Never	0x01
External Authenticate	0x02
External Authenticate or PIN	0x03
Secure Channel (Global Platform)	0x04
PIN Always	0x05
PIN	0x06
External Authenticate then PIN	0x07
Update Once	0x08
PIN then External Authenticate	0x09
Secure Channel (ISO)	0x0B
Reserved	0x0C-0xFF

5.3.2 Virtual Machine Card Edge General Error Conditions

The following General Error Conditions table applies to all virtual machine card edge interface methods:

Table 5–4: General Error Conditions

Status	Meaning
0x6200	Applet or instance logically deleted
0x6581	Memory failure
0x6700	Incorrect parameter Lc
0x67LL	Wrong length in Le parameter, the 'LL' value is expected
0x6982	Security status not satisfied
0x6985	Conditions of use not satisfied
0x6A80	Invalid data in command Data Field
0x6A84	Insufficient memory space to complete command

Status	Meaning
0x6A86	Incorrect P1 or P2 parameter
0x6A88	Referenced data not found
0x6D00	Unknown instruction given in the command
0x6E00	Wrong class given in the command
0x6F00	Technical problem with no diagnostic given
0x9000	Normal ending of the command

5.3.3 Common Virtual Machine Card Edge Interface

5.3.3.1 Access Control

A fixed set of Access Control Rules, also referred to as Security levels, shall be assigned to the Common Virtual Machine Card Edge Interface APDU commands as follows:

Table 5–5: Security levels assigned to the Common VM CEI

APDU	Security Level
Get Properties	Always
Get Challenge	Always
External Authenticate	Always
Pin Verify	Always

5.3.3.2 Get Properties

This command is used to retrieve applet instance properties.

The **Get Properties** command message is coded according to the following table:

CLA	0x80
INS	0x56
P1	0x00
P2	0x00
Lc	0x00
Data Field	Empty
Le	Expected applet instance properties length

Response message

Data field returned in the response message

The Data fields returned in the response message contain the following properties with their current value:

- Applet family (1 byte)
- Applet version (4 bytes)

- Level1 access control/ Level2 access control (1 byte)
- Level3 access control/ RFU (1 byte)
- RFU byte
- RFU byte
- ID-applet AID length (1 byte)
- ID-applet AID (16 bytes padded with 0)
- Key Set Version (1 byte)
- Key Set Id (1 byte)
- T-Buffer length (2 bytes)
- V-Buffer length (2 bytes)
- X bytes RFU to complement to 46 bytes

Processing state returned in the response message

See General Error Conditions in this section.

5.3.3.3 Get Challenge

Each GC applet instance can receive a **Get Challenge** command, in order to perform a host authentication in the following sent command. This command must be sent by a host just before a **External Authenticate** command, and corresponds to the first step of host authentication before sending a command having an external authenticate security level. The computed challenge is valid only in the APDU following the **Get Challenge** APDU.

Command message

The **Get challenge** command message shall be coded according to the following table:

CLA	0x80
INS	0x84
P1	0x00
P2	0x00
Lc	0x00
Data Field	Empty
Le	Challenge length (has to be 8 bytes)

Response message

Data field returned in the response message

The response message contains the challenge used later for authentication.

This challenge has to be memorized inside the applet instance, in order to calculate the corresponding response. If the response is not sent (using **External Authenticate** command) in the command following the **Get Challenge** command, the calculated challenge is then lost.

Processing state returned in the response message

See General Error Conditions in this section.

5.3.3.4 External Authenticate

Just after receiving a card challenge from the **Get Challenge** command, a cryptogram is computed by the host using the 8-Bytes card challenge, and a special 16-Bytes 3DES key known by the host and the static applet.

The cryptogram is the result of the 3DES algorithm with the appropriate key and the challenge as the input.

The key used to protect the level 1 operation shall have a key set version set to 1.

The key used to protect the level 3 operation shall have a key set version set to 3.

The version of the key set must be specified in the command so the applet can know which key set to use.

The corresponding response is then sent to the card using the **External Authenticate** command.

This command must be sent by a host just before a command with an External Authentication security level, and corresponds to the second step of host authentication. Successful cryptogram verification is mandatory to perform the following command.

If the command is not sent just after the **Get Challenge** command, card challenge is then lost.

The External Authentication ACR is valid as long as the session with the card is established and is terminated whenever the session is terminated.

Command message

The **External Authenticate** command message shall be coded according the following table:

CLA	0x80
INS	0x82
P1	Key set version
P2	0x00
Lc	0x08
Data Field	Host cryptogram
Le	Empty

Data field sent in the command message

Host cryptogram.

Response message

Success: 0x9000
Access denied: 0x6982

Data field returned in the response message

The data field returned is always empty

Processing state returned in the response message

See General Error Conditions in this section.

5.3.3.5 PIN Verify

Each GC applet instance can receive a **PIN Verify** command, in order to verify a PIN code, or to check if a PIN code has been already verified or not.

When receiving those commands, the selected instance forwards the information to the ID applet instance containing the PIN number specified in the command, through its shared functions.

Command message

The **PIN Verify** command message shall be coded according the following table:

CLA	0x80
INS	0x20
P1	0x00
P2	0x00
Lc	0x00-0xNN (Effective PIN length expected)
Data Field	PIN code to be verified
Le	Empty

Note 1: The effective PIN length value is retrieved from the ID applet.
Note 2: The maximum effective PIN length is dependent on the card platform, but is typically limited to 127 bytes.

Data field sent in the command message

If the data field sent in the command message does not include a PIN code, the command corresponds to a PIN verify check command, to find out if the PIN code has been already verified or not.

If the data field includes the PIN code to be verified, the PIN code value corresponds to the 8 first bytes of the data field.

If the verification fails, the PIN-tries-remaining flag is decremented, and the PIN-verified flag value does not change. The PIN-always flag value is set to 0x00. If the PIN-tries-remaining flag value is 0x00, the

PIN code is considered as locked. If the verification succeeds, the PIN-verified flag value and the PIN-always flag value are both set to 0x01.

If **P2 =0x01**, the AID of the ID applet instance containing the PIN code to be verified or checked is stored just after the PIN code to verify, or at the beginning of the data field if the command corresponds to a **PIN verified check** command.

Response message

Data field returned in the response message

The data field in the response message is always empty.

Processing state returned in the response message

If PIN verification or PIN verified checking fails, the status code returned is SW1 = 0x63, SW2 = PIN tries remaining. SW2 = 0xFF means infinite tries.

Table 5–6: Meanings of status codes for PIN Verify

Status	Meaning
0x63LL	PIN verify rejected and PIN tries remaining is 'LL'
0x6981	No PIN code defined
0x6983	PIN code blocked

5.3.4 Generic Container Virtual Machine Card Edge Interface

5.3.4.1 Access Control

The following Global Service Levels shall be assigned to the Generic Container Virtual Machine Card Edge Interface APDU commands:

Table 5–7: Global Service Levels for the Generic Container VM CEI

APDU	Global Service Level
Update Buffer	Level1
Read Buffer	Level3

Update Buffer

This command allows updating a part of, or the totality of a buffer.

Command message

The **Update Buffer** command message shall be coded according the following table:

CLA	0x80
INS	0x58
P1	Reference Control Parameter P1
P2	Reference Control Parameter P2

Lc	Length of data + 1
Data Field	Buffer type + data to be updated
Le	Empty

Reference control parameter P1/P2

The reference control parameters P1 and P2 shall be used to store the offset from which data are to be written. This offset is calculated by concatenating the P1 and P2 parameters (P1 = MSB, P2 = LSB).

Data field sent in the command message

The first byte of the data field shall be used to indicate which buffer is to be updated.

The possible values are:

0x01: T-buffer
0x02: V-buffer

The other bytes correspond to the data to be updated.

Response message

Data field returned in the response message

The data field in the response message is always empty.

Processing state returned in the response message

Table 5–8: Meaning of Status Code for Update Buffer

Status	Meaning
0x6981	No corresponding buffer

Read Buffer

This command allows reading a part of, or the totality of, a buffer.

Command message

The **Read buffer** command message shall be coded according the following table:

CLA	0x80 (whatever the access conditions are)
INS	0x52
P1	Reference Control Parameter P1
P2	Reference Control Parameter P2
Lc	0x01 + 0x01
Data Field	Buffer type + data length to read
Le	Empty

Reference control parameter P1/P2

The reference control parameters P1 and P2 shall be used to store the offset from which data are to be read. This offset is calculated by concatenating the P1 and P2 parameters (P1 = MSB, P2 = LSB).

Data field sent in the command message

The data field shall be used to indicate which buffer is to be read.

The possible values are:

0x01: T-buffer
0x02: V-buffer

Response message

Data field returned in the response message

The data field in the response message corresponds to the data read from the card, according to the P1,P2 parameter (offset where to read data).

Processing state returned in the response message

Table 5–9: Meaning of Status Code for Read Buffer

Status	Meaning
0x6981	No corresponding buffer

5.3.5 Symmetric Key Virtual Machine Card Edge Interface

5.3.5.1 Access Control

The following Global Service Levels shall be assigned to the Symmetric Key Virtual Machine Card Edge Interface APDU commands:

Table 5–10: Global Security Levels for Symmetric Key VM CEI

APDU	Global Service Level
Internal Authenticate	Level 1
RFU	Level 2
RFU	Level 3

5.3.5.2 Internal Authenticate

This command is used to perform a 3DES (DES) challenge-response authentication.

Command message

The **Internal Authenticate** command message shall be coded according the following table:

CLA	0x80 (whatever the access control rules are)
INS	0x88
P1	0x00
P2	0x00
Lc	Challenge length (retrieved from a Get properties)
Data Field	Challenge
Le	0x00

Data field sent in the command message

The data field contains the data to be signed using the selected key.

Response message

Data field returned in the response message

The data field in the response message contains the data signed. The length of the response may vary and depends on the configuration of the applet.

Processing state returned in the response message
See General Error Conditions in this section.

5.3.6 Public Key Virtual Machine Card Edge Interface

5.3.6.1 Access Control

The following Global Service Levels shall be assigned to the Public Key Virtual Machine Card Edge Interface APDU commands:

Table 5–11: Global Service Level for the Public Key VM CEI

APDU	Global Service Level
Private sign & decrypt	Level 2
Read certificate	Level 3

5.3.6.2 Private Sign/Encrypt

This command is used to perform a RSA signature or data encryption.

Command message

The **Private Sign/Encrypt** command message shall be coded according the following table:

CLA	0x80 (whatever the access control rules are)
INS	0x42
P1	0x00
P2	0x00
Lc	Data Field length (modulus length)
Data Field	Data to sign
Le	Expected length of the signature/encryption

Data field sent in the command message

The data field contains the data to be signed using the selected RSA key pair.

The data have to be already padded according to the standard used (PKCS#1 for example), before the message is sent. The message is never padded in the card.

Response message

Data field returned in the response message

The data field in the response message contains the data signed or decrypted.

Processing state returned in the response message

See General Error Conditions in this section.

5.3.6.3 Get Certificate

This command is used to retrieve the certificate associated with an RSA key pair.

Command message

The **Get Certificate** command message shall be coded according the following table:

CLA	0x80 (whatever the access control rules are)
INS	0x36
P1	0x00
P2	0x00
Lc	0x00
Data Field	Empty
Le	0x00

Response message

Data field returned in the response message

The data field in the response message contains n bytes of certificate (or less). The SPS determines if there is more data to read depending on the Status word returned from the applet: 0x9000 indicates that the complete certificate has been read, 0x63xx means that xx bytes are still available for reading(in practice, it is advisable to set n to 0x64. This is application/vendor specific.)

Processing state returned in the response message

Table 5–12: Meanings of the Status Codes for Get Certificate

Status	Meaning
0x6981	No corresponding certificate
0x6310	More data is available

THIS PAGE INTENTIONALLY LEFT BLANK.

6. Card Capabilities Container

6.1 Overview

To accommodate variations in smart card APDU set implementations, the GSC-IS defines a VCEI and a general mechanism for mapping a smart card's native APDU set to the VCEI. Each GSC-IS compliant smart card shall contain a CCC and support a standard procedure for accessing it. The contents of a CCC shall comply with the formal card capabilities grammar defined in this chapter.

The CCC describes the differences between a smart card's APDU set and the standard APDU set defined by the VCEI. Virtual Machine cards can be programmed to directly implement the VCEI APDU set. However, Virtual Machine cards shall still contain a minimal CCC.

In general, an SPS retrieves a card's CCC immediately after establishing a logical connection to the card. Without the card's CCC, the SPS cannot communicate with the card at the APDU level.

6.2 Procedure for Accessing the CCC

6.2.1 General CCC Retrieval Sequence

The CCC shall be stored under a known AID on Virtual Machine cards. An SPS first attempts to retrieve the CCC using this AID. If this fails, the SPS then attempts to read the CCC using the file system card procedure below. If that also fails, the SPS assumes that the card does not contain a valid CCC and is not GSC compliant.

6.2.2 CCC Retrieval Sequence for File System Cards

The SPS sends a sequence of APDUs to the card until the CCC is successfully read. This sequence selects the Master File (MF), then the CCC Elementary File (EF), and then performs a binary read operation on the CCC EF:

1. Send command APDU as follows to select MF:

CLA	INS	P1	P2	P3	FIDH	FIDL
TEST CLA	A4	00	00	02	3F	00

   The default TEST CLA values are: 0x00, 0xC0, 0xF0, 0x80, 0xBC, 0x01. (Additional test values for CLA are: 0x90, 0xA0, 0xB0-0xCF.)

2. Wait for Status bytes. If Status Bytes are "0x6E00", Class is not supported. Loop back and attempt another CLA.

3. If Status Bytes are "0x9000" or "0x61XX", correct command structured and CLA

4. If Status Bytes are none of the above, set P2=0x0C and repeat steps 1 through 3.

5. Once CLA has been determined, select DF under MF.

CLA	INS	P1	P2	P3	FIDH	FIDL
Determined CLA	A4	00	P2	02	DFIDH	DFIDL

1. If Status Bytes return error codes (values other than 0x9000 or 0x61XX), set P1=0x01.

2. To select a EF under a selected DF:

CLA	INS	P1	P2	P3	FIDH	FIDL
Determined CLA	A4	00	P2	02	EFIDH	EFIDL

1. If Status Bytes return error codes (values other than 0x9000 or 0x61XX), set P1=0x02.

2. To Read a binary file with no secure messaging, use the following APDU:

CLA	INS	P1	P2	P3
Determined CLA	B0	Off/H	Off/L	L

Note 1: P1 and P2 point to the first byte of data to be read and L is the number of bytes to read.

Note 2: SPS implementations should define a timeout value, to avoid an infinite wait for a response from the card. The timeout mechanism and value are application specific, since in some cases the card reader driver layer may provide this. The SPS will return `BSI_TIMEOUT_ERROR` in response to a **`gscBsiUtilConnect()`** if a connection cannot be established before the timeout value expires.

6.3 Card Capabilities Container Structure

The GSC-IS CCC shall be contained in the Master Directory (0x3F00) and is designated by the Capabilities Application Identifier (AID: GSA-RID||DB00) as well as the FID: 0xDB00.

The CCC shall consist of a collection of SIMPLE-TLV data objects. It is configured for ALWAYS READ. However, it is up to each agency to define its write/modify rules.

Table 6–1: CCC Fields

Card Capabilities Container		EF DB00	Always Read
Data Element (TLV)	**Tag**	**Type**	
Card Identifier	0xF0	Variable	
Capability Container version number	0xF1	Fixed: 1 Byte	
Capability Grammar version number	0xF2	Fixed: 1 Byte	
Applications CardURL	0xF3	Variable – Multiple Objects	
PKCS#15	0xF4	Fixed: 1 Byte	
Registered Data Model number	0xF5	Fixed: 1 Byte	
Redirection Tag	0xFA	Variable	
Capability Tuples (CTs)	0xFB	Variable: Collection of 2 Byte Tuples	
Status Tuples (STs)	0xFC	Variable: Collection of 3 Byte Tuples	

Card Capabilities Container		EF DB00	Always Read
Data Element (TLV)	**Tag**	**Type**	
Next CCC	0xFD	Fixed: 7 Bytes	
Optional Issuer Objects	Issuer Def	Variable	
Error Detection Code	0xFE	LRC	

The following sections describe the above fields in the CCC. The card issuer may include additional TLV objects in the Card Capabilities Container for application specific purposes. These are not needed for interoperability but may be used to facilitate extended applications. They may be ignored by any implementation without affecting interoperability. Any optional objects that are not recognized shall be ignored.

6.4 Card Identifier Description

The Card Identifier shall be specified by each issuing organization for each card type. Among other things, the Card Identifier allows a client application to determine the type of card it is communicating with. The CardUniqueIdentifier is defined by the following ASN.1 sequence:

```
CardUniqueIdentifier ::= SEQUENCE {
      GSC-RID                 OCTET STRING SIZE(5)
      ManufacturerID          BIT STRING SIZE(8),
      CardType,
      CardID                  STRING
}

cardType ::=            CHOICE {
      fileSystemCard          [0] BIT STRING SIZE(8) : '0x0001'B,
      javaCard                [1] BIT STRING SIZE(8) : '0x0002'B,
      Multos                  [2] BIT STRING SIZE(8) : '0x0003'B
      ...
}
```

6.5 CardURL Structure

The Card Capabilities Container may contain multiple instances of Applications CardURL tags. They can be assembled into a list of the applications, including FIDs and paths, Key Identifiers and Access Control Methods, which are supported by the card (see Section 7.1).

6.6 Next CCC Description

The next CCC field, if non-zero, points to the location of another CCC. The values on the second CCC would extend or override values in the primary CCC.

6.7 PKCS#15 Field

The PKCS#15 Field, if non-zero, indicates that the card conforms to PKCS#15. If the field is non-zero, it shall indicate the version of PKCS#15.

6.8 Registered Data Model Number

The Registered Data Model number indicates the registered data model in use by the card (see Chapter 8).

6.9 Redirection Tag

In the case an Agency decides that a specific subset of Tags need a particular Security Context and that a specific access control rule should be enforced, it is possible to create a Container for this set of Tags.

The Redirection Tag can be used to indicate to the BSI Provider which J.8 Tags are being "redirected" to the Container.

The "value" part of the TLV for this redirection Tag can be described as follows:

```
Redirection_value ::=   SEQUENCE {
      dedicatedFileID         BIT STRING SIZE(16),
      Tags
}

Tags :=                 SEQUENCE {
      tagID                   BIT STRING SIZE(8),
      ...
}
```

where each "tagID" is a redirected tag.

A J8 Container can have any number of "redirection flags" to handle Tag level exceptions to the J8 nominal model.

6.10 Capability and Status Tuples

The CCC shall contain a single Capability Tuple (CT) object, which consists of a collection of two byte tuples defining the capabilities, formats and procedures supported by the card. The VCEI defines a default set of APDUs that represent a generic implementation of the ISO 7816 standard. It is only necessary to include CT's to indicate a variance between a given card's capabilities and the default set.

The CCC may contain a single Status Tuple (ST) object, consisting of a collection of three byte tuples that define the possible status codes for each function. It is only necessary to include STs that differ from the card's status codes and the status codes defined in ISO 7816-4.

The following sections describe the construction of tuples in more detail.

6.10.1 APDU Review

The format and characteristics of an APDU are defined in the ISO 7816 standard which specifies two kinds of APDU, a command APDU and a response APDU:

Command APDU						
CLA	INS	P1	P2	Lc	DataC	LE

Response APDU		
DataR	SW1	SW2

Legend:

CLA	Class byte (for Java applets, specifies the AID)
INS	Instruction byte; ISO 7816 defines a set of common commands, e.g., 'B0' is Read Binary
P1	Instruction parameter 1
P2	Instruction parameter 2
Lc	Number of bytes present in data field of the command

DataC	String of bytes sent in the data field of the command
Le	Maximum number of bytes expected in the data field of the response to the command
DataR	String of bytes received in the data filed of the response
SW1	Command processing status, i.e., the return code from the smart card
SW2	Command processing qualifier (secondary return code, supplies further information on SW2)

6.10.2 Capability Tuples

The CCC shall contain a sequence of two-byte elements called tuples. Each tuple comprises a C-byte and a V-byte as shown in Table 6–2. Each tuple describes one piece of an APDU for a particular command. For example, one tuple may define the value of the CLA byte for a Select File command, while another tuple may define the value of P1 for the same command.

Table 6–2: Tuple Byte Descriptions

C - Code Byte							
7	6	5	4	3	2	1	0
0= Const	Parameter			Function Code			
1= Desc							

V – Value/Descriptor Byte							
7	6	5	4	3	2	1	0
If C bit 7 = 0 Then V contains a constant value							
If C bit 7 = 1 Then V contains a Descriptor code							

The C-byte of the tuple is the Code Byte. It identifies the particular command and parameter that is being defined. The V-byte is the Value Byte, which provides either the value to be used for the parameter or a descriptor code that represents the definition of the parameter, that is, what the parameter is in the APDU. This could be, for example, the most-significant byte of the offset for a Read Binary command, or the CHV level for a Verify Pin command. Whether the V-byte is a constant value or a descriptor code, is determined by the 7^{th} bit of the C-byte. If this bit is 0, the V-byte contains a value while, if it is 1, the V-byte contains a descriptor code. Bits 6 through 4 of the C-byte identify the parameter and bits 3 through 0 identify the particular command.

The possible values for the codes used in the C and V-bytes are summarized here in Table 6–3.

6.10.2.1 Extended Function Codes

The construction of the Code Byte allows only four bits for the designation of the function code; however, it may, at times, be necessary to use more than the allocated commands. For example, prefix or suffix commands that are card specific may be required to fulfill the processing for the GSC-IS command on a particular card.

The reserve function code 0x00 is used to define a *shift* tuple. This tuple is used in the sequence of tuples to place all following function codes in a shift state defined by the high-order four bits of the shift key. The function codes are logically or'ed with the current shift tuple to create an extended function code. Placing another shift tuple in the tuple stream places function codes in an un-shift or other shift state.

Table 6–3: Parameter and Function Codes

Parameter Codes		Function Codes	
0x00	DATA	0x00	Reserved, Used for Shift Operation
0x01	CLA	0x01	Select File (DF)
0x02	INS	0x02	Transparent Read (Binary)
0x03	P1	0x03	Update Binary File
0x04	P2	0x04	Update Binary File (Secure Messaging)
0x05	P3	0x05	RSA Compute
0x06	Prefix	0x06	Get Challenge
0x07	Suffix	0x07	Get Response
		0x08	Verify PIN (CHV)
		0x09	Internal Authenticate
		0x0A	External Authenticate
		0x0B	RFU
		0x0C	Select KEY
		0x0D	Select EF (under current DF)
		0x0E	Select MF (root)
		0x0F	Read Binary (Secure Messaging)

6.10.2.2 Prefix and Suffix Codes

Parameter codes 06 (hexadecimal) and 07 represent prefix and suffix commands respectfully. These are commands (function codes) that must execute before or after the specified function code. For example, on some cards, a **Get Response** must succeed a cryptographic function, or a **Verify Pin** must precede a **Read Binary** with secure messaging.

6.10.2.3 Descriptor Codes

The descriptor codes are used to add processing information for data values or parameters. Parameters can be described by at most one descriptor code, whereas data values can be described by multiple, successive descriptor codes.

Table 6–4: Descriptor Codes

Descriptor Codes			
0x00 – 0x0F	Execute Function Code	0x11	Challenge
0x12	Algorithm Identifier	0x13	6 MSB of Cryptogram
0x14	3 LSB of Cryptogram	0x15	Length
0x16	MSB of Offset	0x17	LSB of Offset
0x18	Key Level	0x19	Key Identifier
0x1A	CHV Level	0x1B	CHV Identifier
0x1C	AID	0x1D	EF
0x1E	System ID	0x1F	RFU

Descriptor Codes			
0x21	2 Byte FID	0x22	Short FID
0x23	File Name	0x29	3 LSB MAC
0x2C	Key File Identifier	0x2D	3 LSB MAC
0x2E	4 MSB of Cryptogram	0x2F	8 Byte Random Number
0x38	4 LSB of Cryptogram	0x3E	Pad = 00
0x43	Terminal Random Number	0x45	Key File Short ID
0x46	MSB of Offset in Words	0x47	LSB of Offset in Words
0x48	Chaining Information	0x49	Block Length
0x4A	TLV Format	0x4B	Operation Mode
0x4C	LOUD	0x4D	6 MSB CBC Cryptogram
0x4E	8 Byte Cryptogram	0x4F	Session Key 1
0x50	Length + X	0x51	Pad with X 0xFF Bytes
0x52	6 MSB of MAC	0x53	Length + 8
0x54	8 Byte MAC	0x55	Session Key 2
0x56	TLV Command Data for Update Binary	0x57	TLV Response for Update Binary
0x58	TLV Command Data for Read Binary	0x59	TLV Response Data for Read Binary
0x5A	4 MSB of MAC (TLV Command MAC 1)	0x5B	4 MSB of MAC (TLV Response MAC 1)
0x5C	4 MSB of MAC (TLV Command MAC 2)	0x5D	4 MSB of MAC (TLV Response MAC 2)
0x5E	8 Byte MAC 2	0x5F	Key Number << 1
0x60	Key Level Flag	0x61	Length + #Padding
0x62	Length of RSA Response	0x63	RSA Response Data
0x64	Pad Hashed Data (PKCS#1)	0x65	Swap Data Bytes
0x66	TLV Key ID	0x67	TLV Hash Algorithm ID
0x68	Key Length Padded Hash Data	0x69	Key Length + 2
0xA0-0xDF	Implementation Dependant	0xE0	Output Data Bytes
0xE1-0xEF	E*n*: Next *n* bytes are Data Bytes	0xF0-0xFC	Reserved
0xFD	Interpret Response	0xFE	Command not available
0xFF	User Input Required		

6.10.3 CCC Formal Grammar Definition

Using a modified Backus-Naur notation, a formal definition for the Card Capability Grammar is presented as follows:

```
Command_Unit,[Command_Unit,...]
Command_Unit:(
      FC:(function_code,[extension]),
      Command:(
            APDU:(
                  CLA:(class,[qual=0xFE]),
```

```
        INS:instruction,
        P1:((p_constant|<value>),def:{code,...}),
        P2:((p_constant|<value>,def:{code,...}),
        P3:(length,def:{code,...}),                //of data
        DATA:(composition:data_type[+data_type(...)])
        ),
[Prefix:function_code],    //could depend on extension
[Suffix:function_code]     //could depend on extension
)
```

The rules for building and APDU definition according to the formal grammar are as follows:

1. The sequence of tuples is organized in groups called command units; all tuples pertaining to a single command unit must be presented in contiguous sequence.
2. The sequence of tuples is important and must be presented in the order defined by the formal grammar
3. Each command unit consists of a required function code and optional APDU elements.
4. When present, the CLA element may have only a constant value (and/or one qualifier code equal to FE hex, which indicates the command is not available on the card).
5. When present, the INS element must have a constant value.
6. When present, the P1 element may optionally have a constant value and/or one definition code.
7. When present, the P2 element may optionally have a constant value and/or one definition code.
8. When present, the P3 element may have a constant value; P3 always refers to the length of the DATA element.
9. The DATA element may have multiple data type codes; when combined the data type codes define the composition of the value to be placed in the APDU data field.

As an example of using the Card Capability Grammar, consider the following GSC-IS-default APDU for a Select Dedicated File command along with the same command for the Schlumberger Cryptoflex card:

Table 6–5: Default vs Schlumberger DF APDU

Select Dedicated File (DF)							
Card Type	**CLA**	**INS**	**P1**	**P2**	**P3**	**Data**	**Notes**
GSC-IS Default	00	A4	01	00	L (02)	File ID (2 Bytes)	
Schlumberger Cryptoflex	C0	A4	00	00	L (02)	File ID (2 Bytes)	

The formal grammar definition of the Cryptoflex command is as follows:

FC:01, CLA:C0, INS:A4, P1:00, P2:00, P3:(02,def:15), DATA:21

which translates into the following tuple sequence:

11C0 21A4 3100 4100 5102 D115 8121

The method for creating the tuple sequence is shown in the following table, where the C-Byte and V-Byte are built from the parameter, function, and descriptor codes given in the Table 6–6:

Table 6–6: Tuple Creation Sequence

#	C-Byte			V-Byte	Description			Tuple
	S	P	FC		Function, Parm	V/D	Value/Descriptor	
1	0	1	1	C0	Select File, CLA	V	"C0"	11C0
2	0	2	1	A4	Select File, INS	V	"A4"	21A4
3	0	3	1	00	Select File, P1	V	"00"	3100
4	0	4	1	00	Select File, P2	V	"00"	4100
5	0	5	1	02	Select File, P3	V	"02"	5102
6	1	5	1	15	Select File, P3	D	Length	D115
7	1	0	1	21	Select File, Data	D	Short FID	8121

The above table shows the complete tuple sequence to define the Select DF command for the Cryptoflex card according to the CC Grammar; however, the only differences in the APDU between the GSC-IS Default and the Cryptoflex card are the CLA byte and the P1 parameter. Therefore, only two tuples are necessary since the rest of the APDU is defined by the GSC-IS defaults. The tuples required to define this command for the Cryptoflex card would be:

Table 6–7: Derived DF Tuple

#	C-Byte			V-Byte	Description			Tuple
	S	P	FC		Function, Parameter	V/D	Value/Descriptor	
1	0	1	1	C0	Select File, CLA	V	"C0"	11C0
2	0	3	1	00	Select File, P1	V	"00"	3100

6.10.4 Status Tuple Construction

Status Tuples shall consist of three bytes, labeled S, SW1 and SW2, which describe the possible status conditions for each function. Multiple sets of SW1 and SW2 may translate into a single Status Condition. The purpose of the Status Tuples is to map a card's non-standard status response SW1 & SW2 into a common set of status conditions for a given function. It is not mandatory to list any status conditions that conform to ISO-7816.

Table 6–8 and Table 6–9 describe the status tuple construction and status condition codes.

Table 6–8: Status Tuples

S									SW	
7	6	5	4	3	2	1	0			
Status Condition				Function Code					SW1	SW2

Table 6–9: Standard Status Code Responses

Status Conditions	
0x00	Successful Completion
0x01	Successful Completion – Warning 1
0x02	Successful Completion – Warning 2
0x03	Reserved
0x04	Reserved
0x05	Reserved
0x06	Reserved
0x07	Reserved
0x08	Access Condition not Satisfied
0x09	Function not Allowed
0x0A	Inconsistent Parameter
0x0B	Data Error
0x0C	Wrong Length
0x0D	Function not compatible with file structure
0x0E	File/Record not Found
0x0F	Function Not Supported

7. Container Naming

The GSC-IS architecture isolates client applications from the differences between virtual machine and file system cards. Virtual machine cards use AID to identify applets, and file system cards use File Ids (FID) to identify files. These differences are abstracted by defining Applications CardURL and Universal AID structures that are common to both virtual machine and file system cards. In this context the terms "applet", "service", "container" and "file" are synonymous. The term "container" will be used preferentially throughout this section.

7.1 Discovery: the Applications CardURL

Client applications need a way to discover the AIDs associated with specific containers on a card. The preferred method of accessing a container on a virtual machine smart card is to select it with an ID. This is actually the only method that works with a JavaCard™. Unfortunately, this method is not supported by most file system cards. Therefore, the GSC-IS introduces the concept of an Applications CardURL, which can be used to uniquely reference a container offered by the smart card. This structure also provides a mechanism for client applications to determine the ACRs and PIN and key labels associated with a given container.

Applications CardURL structures are stored in the CCC (reference Section 6). All GSC conformant smart cards shall provide a CardURL for each container present on the card. Several fields are optional because they are not applicable to Virtual Machine cards: DedicatedFileID, PINID, ReadAccessKeyInfo, and WriteAccessKeyInfo. For file system cards, all Dedicated Files that comprise the card's data model shall be stored in the card's Master File.

```
GSC-ISContainerURL ::=     SEQUENCE {
      Rid                     OCTET STRING SIZE(5),
      CardApplicationType,
      FileID
      dedicatedFileID         BIT STRING SIZE(16),
      FileAccessProfile,
      pinID                   BIT STRING SIZE(8),
      readAccessKeyInfo       AccessKeyInfo,
      writeAccessKeyInfo      AccessKeyInfo
}

CardApplicationType ::=    CHOICE {
      genericContainer        [0]   BIT STRING SIZE(8) : '0x01'B,
      ski                     [1]   BIT STRING SIZE(8) : '0x02'B,
      pki                     [2]   BIT STRING SIZE(8) : '0x04'B,
      genericContainerExt     [3]   BIT STRING SIZE(8) : '0x08'B,
      cryptoSKIExt            [4]   BIT STRING SIZE(8) : '0x10'B,
      cryptoPKIExt            [5]   BIT STRING SIZE(8) : '0x20'B
}

FileID ::=                 CHOICE {
      generalInfo             [0]   BIT STRING SIZE(16)
      proPersonalInfo         [1]   BIT STRING SIZE(16),
      accessControl           [2]   BIT STRING SIZE(16),
      login                   [3]   BIT STRING SIZE(16),
      cardInfo                [4]   BIT STRING SIZE(16),
      biometrics              [5]   BIT STRING SIZE(16),
      digitalSigCert          [6]   BIT STRING SIZE(16)
}
```

```
FileAccessProfile ::=      SEQUENCE OF { ACRS }

ACRS ::=                   SEQUENCE  {
      readACR                 ACR,
      writeACR                ACR
}


ACR ::=                    CHOICE {
      always                  [0]   BIT STRING SIZE(4) : '0x0000'B,
      never                   [1]   BIT STRING SIZE(4) : '0x0001'B,
      extAuthent              [2]   BIT STRING SIZE(4) : '0x0002'B,
      extAuthentOrPIN         [3]   BIT STRING SIZE(4) : '0x0003'B,
      secureChannelGP         [4]   BIT STRING SIZE(4) : '0x0004'B,
      pinAlways               [5]   BIT STRING SIZE(4) : '0x0005'B,
      pinProtected            [6]   BIT STRING SIZE(4) : '0x0006'B,
      extAuthentThenPIN       [7]   BIT STRING SIZE(4) : '0x0007'B,
      updateOnce              [8]   BIT STRING SIZE(4) : '0x0008'B,
      pinThenExtAuthent       [9]   BIT STRING SIZE(4) : '0x0009'B,
      SecureChannelISO        [10]  BIT STRING SIZE(4) : '0x000B'B
}


AccessKeyInfo ::=       SEQUENCE  {
      keyFileID               BIT STRING SIZE(16),
      keyNumber               BIT STRING SIZE(8),
      keyCryptoAlgorithm
}

keyCryptoAlgorithm         CHOICE {
         3DES16                [0] BIT STRING SIZE(8),
         AES                   [1] BIT STRING SIZE(8)
}
```

7.2 Selection: The Universal AID

Client applications use Universal AIDs to select generic containers to perform data oriented operations on, and to select cryptographic service modules. For generic container references, Universal AIDs are constructed by concatenating the RID value with the File ID of the desired container. For example, the Universal AID of the Card Capabilities Container on a card that conforms to the GSC Data Model (Appendix C) would be 0xA000000116DB00. Applications can determine File IDs for each of the generic containers on a card by examining the registered container data model for that card.

An application forms the Universal AID for selecting cryptographic service modules by concatenating the GSC RID value with the File ID of the desired cryptographic key file (symmetric or asymmetric). The AID of the key file is therefore the same as the AID of the cryptographic service module (PKI or SKI). Applications can obtain key File IDs from the card's container data model (PKI) or the appropriate Applications CardURL (SKI).

8. Container Data Models

8.1 Data Models

Container Data Models define a set of containers (files) and associated data elements in Tag-Length-Value (TLV) format. The only mandatory container is the CCC. With the exception of the CCC, a GSC-ISv2.0 compliant card may implement all, some, or none of the containers associated with a Data Model. However, if the card uses any of the data elements defined in its Data Model then it must use the container and TLV format specified by that Data Model. As an example, assume that a particular implementation only requires the last name field of the General Information container. That card would only have a CCC and a General Information container with two TLV data elements: the Last Name field and the mandatory Error Detection Code field.

This specification defines two container Data Models. The first Data Model was developed for version 1.0 of the GSC-IS (see Appendix C). This Data Model is sometimes referred to as the "J.8" Data Model, since it was first defined in Section J.8 of the Smart Access Common ID Card contract. The second Data Model was developed for the DoD Common Access Card (CAC) and is referred to as the CAC Data Model(See Appendix D).

A GSC-ISv2.0 compliant smart card must implement one of these two Data Models. Applications can discover which Data Model a given card supports by examining the Registered Data Model field of the card's Card Capabilities Container (see Chapter 6). The Registered Data Model field shall contain a 1 if the card is using the GSC (J.8) Data Model defined in Appendix C, or a 2 if the card conforms to the CAC Data Model in Appendix D. Error Detection Codes are only mandated for the GSC Data Model.

8.2 Internal TLV Format

All container data elements are stored in TLV format. Each SIMPLE-TLV data object shall consist of 2 or 3 consecutive fields:

The tag field T shall consist of a single byte encoding only a number from 1 to 254. No class or construction types are coded. The tag value 0xFE is reserved for the mandatory EDC data object in each container. The scope of tag values is at the container level, so the same tag value could appear in different containers. Unique tag values are used across all containers in the current GSC-IS Data Models, although this is not a mandatory requirement.

The length field shall consist of 1 or 3 consecutive bytes. If the leading byte of the length field is in the range from '00' to 'FE', then the length field shall consist of a single byte encoding an integer L valued from 0 to 254. If the leading byte is equal to 'FF', then the length field continues on the two subsequent bytes in least significant byte-most significant byte order, which encode an integer L with a value from 0 to 65,535.

If L is not null, then the value field V shall consist of L consecutive bytes. If L is null or if a tag is omitted from its file/buffer, then the data object must be empty: there is no value field for that tag.

THIS PAGE INTENTIONALLY LEFT BLANK.

Appendix A—Normative References

[DES] National Institute of Standards and Technology, "DES Modes of Operation", Federal Information Processing Standards Publication 81, December 1980, http://csrc.nist.gov/publications/fips.

[GLOB] GlobalPlatform Specification v2.1, http://www.globalplatform.org.

[ISO3] ISO/IEC 7816-3: Electronic Signals and Transmission Protocols, http://www.iso.ch.

[ISO4] ISO/IEC 7816-4: Interindustry Commands for Interchange

[ISO9] International Organization for Standardization, "Information Processing Systems -- Data Communication High-Level Data Link Control Procedure--Frame Structure", IS 3309, October 1984, 3rd Edition.

[SEIWG] Security Equipment Integration Working Group 012 Specification – 28 February 1994

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix B—Informative References

[OCF] The OpenCard Framework, http://www.opencard.org.

[JAV] Java Card 2.1.1 Platform Documentation, http://java.sun.com/products/javacard/javacard21.html

[PCSC] Personal Computer/Smart Card Workgroup Specifications, http://www.pcscworkgroup.com.

THIS PAGE INTENTIONALLY LEFT BLANK.

Appendix C—GSC Data Model

The RID for the GSC Data Model is 0xA000000116.

File/Buffer Description	FID	Maximum Length (Bytes)	Read Access Condition
Capability	DB00		Always
General Information	2000	509	Always
Protected Personal Information	2100	19	After Verify CHV
Access Control	3000	59	Always
Login	4000	141	After Verify CHV
Card Information	5000	165	Always
Biometrics – X.509 Certificate	6000	2013	Always
PKI – Digital Signature Certificate	7000	3017	After Verify CHV

General Information File / Buffer		EF 2000	Always Read
Data Element (TLV)	**Tag**	**Type**	**Max. Bytes**
First Name	01	Variable	20
Middle Name	02	Variable	20
Last Name	03	Variable	20
Suffix	04	Variable	4
Government Agency	05	Variable	30
Bureau Name	06	Variable	30
Agency Bureau Code	07	Variable	4
Department Code	08	Variable	4
Position/Title	09	Variable	30
Building Name	10	Variable	30
Office Address 1	11	Variable	60
Office Address 2	12	Variable	60
Office City	13	Variable	50
Office State	14	Variable	20
Office ZIP	15	Variable	15
Office Country	16	Variable	4
Office Phone	17	Variable	15
Office Extension	18	Variable	4
Office Fax	19	Variable	15
Office Email	1A	Variable	60
Office Room Number	1B	Variable	6
Non-Government Agency	1C	Fixed Text	1
SSN Designator	1D	Variable	6
Error Detection Code	FE	LRC	1

Protected Personal Information File / Buffer		EF 2100	CHV Verify
Data Element (TLV)	**Tag**	**Type**	**Max. Bytes**
Social Security Number	20	Fixed Text	9
Date of Birth	21	Date (YYYYMMDD)	8
Gender	22	Fixed Text	1
Error Detection Code	FE	LRC	1

Access Control File / Buffer		EF 3000	Always Read
Data Element (TLV)	**Tag**	**Type**	**Max. Bytes**
SEIWG Data	30	Fixed	40
PIN	31	Fixed Numeric	10
Domain (Facility / System ID)	32	Variable	8
Error Detection Code	FE	LRC	1

Login Information File / Buffer		EF 4000	CHV Verify
Data Element (TLV)	**Tag**	**Type**	**Max. Bytes**
User ID	40	Variable	60
Domain	41	Variable	60
Password	42	Variable	20
Error Detection Code	FE	LRC	1

Card Information File / Buffer		EF 5000	Always Read
Data Element (TLV)	**Tag**	**Type**	**Max. Bytes**
Issuer ID	50	Variable	32
Issuance Counter	51	Variable	4
Issue Date	52	Date (YYYYMMDD)	8
Expiration Date	53	Date (YYYYMMDD)	8
Card Type	54	Variable	32
Demographic Data Load Date	55	Date (YYYYMMDD)	8
Demographic Data Expiration Date	56	Date (YYYYMMDD)	8
Card Security Code	57	Fixed Text	32
Card ID AID	58	Variable	32
Error Detection Code	FE	LRC	1

Biometrics – X.509 Certificate File / Buffer		EF6000	Always Read
Data Element (TLV)	**Tag**	**Type**	**Max. Bytes**
Template	60	Variable	512
Certificate	61	Variable	1500
Error Detection Code	FE	LRC	1

PKI – Digital Signature Certificates File / Buffer		EF 7000	CHV Verify
Data Element (TLV)	**Tag**	**Type**	**Max. Bytes**
Certificate	70	Variable	3000
Issue Date	71	Date (YYYYMMDD)	8
Expiration Date	72	Date (YYYYMMDD)	8
Error Detection Code	FE	LRC	1

The Security Enterprise Integration Working Group (SEIWG, [SEIWG]) data element within the Access Control File container as required by the SEIWG specification shall be of the Packed Binary Coded Decimal (BCD) character set. All other file containers and data elements shall use the American Standards Code for Information Interchange (ASCII) character set, except for certain fields such as public key certificates, biometric data, and PINs that may be application dependent.

THIS PAGE INTENTIONALLY LEFT BLANK.

Appendix D—CAC Data Model

The RID for the CAC Data Model is 0xA000000079.

File/Buffer Description	FID	Maximum Length (Bytes)	Read Access Condition
Capability	DB00		Always
Person Instance Container	0200	469	PIN or External Auth
Benefits Information Container	0202	19	PIN or External Auth
Other Benefits Container	0203	59	PIN or External Auth
Personnel Container	0201	141	PIN or External Auth
Login Information Container	0300	133	PIN or External Auth
PKI Certificate Container	02FE	2013	PIN Always

Person Instance File/Buffer	EF 0200	Always Read	
Data Element (TLV)	**Tag**	**Type**	**Max. Bytes**
Person First Name	01	Variable	40
Person Middle Name	02	Variable	40
Person Last Name	03	Variable	52
Person Cadency Name	04	Variable	8
Person Identifier	05	Fixed Text	30
Date of Birth	06	Date(YYYYMMDD)	16
Sex Category Code	07	Fixed Text	2
Person Identifier Type Code	08	Fixed Text	2
Blood Type Code	11	Fixed Text	4
DoD EDI Person Identifier	17	Fixed Text	20
Organ Donor	18	Fixed Text	2
Identification Card Issue Date	62	Date(YYYYMMDD)	16
Identification Card Expiration Date	63	Date(YYYYMMDD)	16
Date Demographic Data was Loaded on Chip	65	Date(YYYYMMDD)	16
Date Demographic Data on Chip Expires	66	Date(YYYYMMDD)	16
Card Instance Identifier	67	Fixed Text	2

Benefits Information File / Buffer	EF 0202	CHV Verify	
Data Element (TLV)	**Tag**	**Type**	**Max. Bytes**
Exchange Code	12	Fixed Text	2
Commissary Code	13	Fixed Text	2
MWR Code	14	Fixed Text	2
Non-Medical Benefits Association End Date	1B	Date(YYYYMMDD)	16
Direct Care End Date	1C	Date(YYYYMMDD)	16

Benefits Information File / Buffer		EF 0202	CHV Verify
Data Element (TLV)	**Tag**	**Type**	**Max. Bytes**
Civilian Health Care Entitlement Type Code	D0	Fixed Text	2
Direct Care Benefit Type Code	D1	Fixed Text	2
Civilian Health Care End Date	D2	Fixed Text	16

Other Benefits File / Buffer		EF 0203	Always Read
Data Element (TLV)	**Tag**	**Type**	**Max. Bytes**
Meal Plan Type Code	1A	Fixed Text	4

Personnel File / Buffer		EF 0201	Always Read
Data Element (TLV)	**Tag**	**Type**	**Max. Bytes**
DoD Contractor Function Code	19	Fixed Text	2
US Government Agency/Subagency Code	20	Fixed Text	8
Branch of Service Code	24	Fixed Text	2
Pay Grade Code	25	Fixed Text	4
Rank Code	26	Fixed Text	12
Personnel Category Code	34	Fixed Text	2
Non-US Government Agency/Subagency Code	35	Fixed Text	4
Pay Plan Code	36	Fixed Text	4
Personnel Entitlement Condition Code	D3	Fixed Text	4

Login Information File / Buffer		EF 0300	CHV Verify
Data Element (TLV)	**Tag**	**Type**	**Max. Bytes**
User ID	0x40	Variable	20
Domain	0x41	Variable	20
PasswordInfo	0x43	Fixed Text	1
ApplicationName	0x44	Variable	8
Error Detection Code	0xFE	LRC	1

PKI Certificate File / Buffer		EF 02FE	CHV Verify
Data Element (TLV)	**Tag**	**Type**	**Max. Bytes**
Certificate	0x70	Variable	1100
CertInfo	0x71	Fixed Text	1
MSCUID	0x72	Variable	38
Error Detection Code	0xFE	LRC	1

Appendix E—C Language Binding for BSI Services

This appendix defines the C language binding for the BSI services. This set of services consists of 21 C functions derived from the IDL specification (Chapter 4). The return codes for the functions are as defined in Section 4.4. Similar to the IDL specification, the C translation is grouped into three functional modules as follows:

- A Smart Card Utility Provider Module
- A Smart Card Generic Container Provider Module
- A Smart Card Cryptographic Provider Module.

E.1 Type Definitions for BSI Functions

The following type definitions are used by multiple BSI functions.

```
#typedef    long              UTILCardHandle
#typedef    unsigned char     GCtag
```

E.2 Parameter Format and Buffer Size Discovery Process

Many BSI function calls accept and/or return variable-length string data. The buffers that store the strings are paired with an integer value representing the number of bytes (the size of the buffer), including the `NULL` terminator. When a string/integer argument pair is passed as input parameters to a BSI function, the SPS shall check the length of the null terminated string against the length indicated by the paired integer. If the two do not agree, the BSI function shall return the error code `BSI_BAD_PARAM`.

Calling applications shall allocate buffers of sufficient size to hold string arguments returned by the BSI functions. The BSI shall provide a discovery mechanism to allow applications to determine required buffer size for returned data. To determine the required buffer size, the calling application must typically call the BSI function two times. However, only one call is possible if the client application is able to estimate the required buffer size.

The client application sets the pointer to the buffer that should be allocated for the returned arguments to NULL. This approach signals to the service that it must determine the buffer size required for the returned arguments and return this information in the corresponding paired integers. The client application then allocates buffers of the required size, sets the paired integers accordingly, and calls the BSI function a second time. The SPS must check the length integer against its previously cached value and, if the value contained in the length integer is greater than or equal to the required buffer length, it shall return the appropriate data in the buffers. See Example 1 in the following Section for additional information.

If an application knows or is able to estimate the required buffer size beforehand, it can shorten the process by making only one call. To do so, the application allocates buffers it believes to be of sufficient size to hold the data returned by the BSI function, sets the paired length integers accordingly, and calls the BSI function. The SPS shall check the length integer against the required value and, if it is greater than or equal to the required buffer length, it shall return the appropriate data in the buffers. If not, the BSI function shall return the `BSI_INSUFFICIENT_BUFFER` error code and the required buffer sizes in the respective paired length integers. See Example 2 in the following Section for more information.

E.3 Discovery Mechanisms Code Samples

Following are two examples in C of using the discovery mechanisms.

Example 1

```
unsigned char *pCertificate = NULL;
unsigned int unCertificateLen = 0;

if(gscBsiPkiGetCerticate(hCard,uszAID,unAIDLen,NULL,
                         &unCertificateLen) == BSI_OK)
{

      if(unCertificateLen > 0)
      {

            pCertificate = (unsigned char*)malloc(
            sizeof(unsigned char)*unCertificateLen);

            if(pCertificate)
            {
                  if(gscBsiPkiGetCertificate(hCard,
                                             uszAID,
                                             unAIDLen,
                                             pCertificate,
                                             &unCertificateLen)==BSI_OK)
                  {

                  //
                  //--- process the certificate ---
                  //

                  }

            }

      }

}
```

Example 2

```
unsigned char *pCertificate = NULL;
unsigned int unCertificateLen = ESTIMATED_CERT_SIZE;
int iRet = BSI_OK;

//...

pCertificate =(unsigned char*) malloc(
                  sizeof(unsigned char) * unCertificateLen);

if(pCertificate)
{

      iRet = gscBsiPkiGetCertificate(hCard,uszAID,unAIDLen,
                        pCertificate,&unCertificateLen);

      if(iRet == BSI_INSUFFICIENT_BUFFER)
      {

            free(pCertificate);

            // --- NOTE: The returned unCertificateLen gives
            // the correct length ---
            pCertificate = (unsigned char*) malloc(
            sizeof(unsigned char)*unCertificateLen);

            if(pCertificate)
            {
                  if(gscBsiPkiGetCertificate(hCard,
                                          uszAID,
                                          unAIDLen,
                                          pCertificate,
                                          &unCertificateLen)==BSI_OK)
                  {
                        //
                        //--- process the certificate ---
                        //
                  }

            }

      }

}
```

E.4 Smart Card Utility Provider Module Interface Definition

E.4.1 gscBsiUtilAcquireContext()

Purpose: This function shall establish a session with the smart card by submitting the appropriate Authenticator in the BSIAuthenticator structure. For ACRs requiring external authentication (XAUTH), the `uszAuthValue` field of the BSIAuthenticator structure must contain a cryptogram calculated by encrypting a random challenge.

Prototype:

```
unsigned long gscBsiUtilAcquireContext(
   IN UTILCardHandle    hCard,
   IN unsigned char     *uszAID,
   IN unsigned int      unAIDLen,
   IN BSIAuthenticator  *strctAuthenticator,
   IN unsigned int      unAuthNb
);
```

Parameters:

hCard: Card connection handle from **gscBsiUtilConnect()**.

uszAID: Target container AID value.

unAIDLen: AID value length in bytes.

strctAuthenticator: An array of structures containing the authenticator(s) specified by the ACR required to access a value in the container. The authenticator is returned by **gscBsiGcGetContainerProperties()**. The calling application is responsible for allocating this structure.

unAuthNb: Number of authenticator structures contained in **strctAuthenticator**.

The BSIAuthenticator structure is defined as follows. `BSI_AUTHENTICATOR_MAX_LEN` and `BSI_KEY_LENGTH` are implementation-dependent constants.

```
struct BSIAuthenticator {
   unsigned int         unACRtype;
   unsigned char        usAuthValue
                        [BSI_AUTHENTICATOR_MAX_LEN];
   unsigned int         unAuthValueLen;
   unsigned char        usKeyValue[BSI_KEY_LENGTH];
   boolean              bWriteKey;
};
```

Variables associated with the BSIAuthenticator structure:

unACRtype: Access Control Rule (see Table 3.1 in Section 3.1).

usAuthValue: Authenticator, can be an external authentication cryptogram or PIN.

unAuthValueLen: Authenticator value length in bytes.

usKeyValue: Cryptographic authentication key..

bWriteKey: At the smart card level, the *read* and *write* privileges are granted sequentially therefore each granted privilege and its acquired security context is associated with either a *read* or a *write* operation through this member of the BSIAuthenticator. If set to TRUE, this ACR controls write access. If set to FALSE, it controls read access.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_ACR_NOT_AVAILABLE
BSI_BAD_AUTH
BSI_CARD_REMOVED
BSI_PIN_LOCKED
BSI_UNKNOWN_ERROR
```

E.4.2 gscBsiUtilConnect()

Purpose: Establish a logical connection with the card in a specified reader.

Prototype:

```
unsigned long gscBsiUtilConnect(
   IN unsigned char     *uszReaderName,
   IN unsigned long     unReaderNameLen,
   OUT UTILCardHandle   *hCard
);
```

Parameters:

hCard:	Card connection handle.
uszReaderName:	Name of the reader that the card is inserted into. If this field is a NULL pointer, the SPS shall attempt to connect to the card in the first available reader, as returned by a call to the BSI's function **gscBsiUtilGetReaderList()**.
unReaderNameLen:	Length of the reader name in bytes.

Return Codes:

```
BSI_OK
BSI_BAD_PARAM
BSI_UNKNOWN_READER
BSI_CARD_ABSENT
BSI_UNKNOWN_ERROR
```

E.4.3 gscBsiUtilDisconnect()

Purpose: Terminate a logical connection to a card.

Prototype:
```
unsigned long gscBsiUtilDisconnect(
   IN UTILCardHandle    hCard
);
```

Parameters: **hCard**: Card connection handle from **gscBsiUtilConnect()**.

Return Codes:
```
BSI_OK
BSI_BAD_HANDLE
BSI_CARD_REMOVED
BSI_UNKNOWN_ERROR
```

E.4.4 gscBsiUtilGetVersion()

Purpose: Returns the BSI implementation version.

Prototype:

```
unsigned long gscBsiUtilGetVersion(
   OUT unsigned char    *uszVersion,
   IN/OUT unsigned long *unVersionLen
);
```

Parameters:

uszVersion:	The BSI and SCSPM version formatted as "major,minor,revision, build_number\0".
unVersionLen:	Length of the version string.

Return Codes:

```
BSI_OK
BSI_BAD_PARAM
BSI_INSUFFICIENT_BUFFER
BSI_UNKNOWN_ERROR
```

E.4.5 gscBsiUtilGetCardProperties()

Purpose: Retrieves version and capability information for the card.

Prototype:
```
unsigned long gscBsiUtilGetCardProperties(
   IN UTILCardHandle    hCard,
   OUT unsigned char    *uszCCCUniqueID,
   IN/OUT unsigned long *unUniqueIDLen,
   OUT unsigned long    *unCardCapability
);
```

Parameters:

hCard: Card connection handle from **gscBsiUtilConnect().**

uszCCCUniqueID: Buffer for the Card Capability Container version .

unUniqueIDLen: Length of the CCC Unique ID string (input). Length of the returned Card Unique ID string including the null terminator (output).

unCardCapability: Bit mask value defining the providers supported by the card. The bit masks represent the Generic Container Data Model, the Generic Container Data Model Extended, the Symmetric Key Interface, and the Public Key Interface providers respectively:

```
#define BSI_GCCDM         0x00000001
#define BSI_SKI           0x00000002
#define BSI_PKI           0x00000004
#define BSI_GCCDM_EXT     0x00000008
#define BSI_SKI_EXT       0x00000016
#define BSI_PKI_EXT       0x00000032
```

Return Codes:
```
BSI_OK
BSI_BAD_HANDLE
BSI_CARD_REMOVED
BSI_BAD_PARAM
BSI_INSUFFICIENT_BUFFER
BSI_NO_CARDSERVICE
BSI_UNKNOWN_ERROR
```

E.4.6 gscBsiUtilGetCardStatus()

Purpose: Checks whether a given card handle is associated with a card that is inserted into a powered up reader.

Prototype:
```
unsigned long gscBsiUtilGetCardStatus(
   IN UTILCardHandle    hCard
);
```

Parameters: **hCard**: Card connection handle from **gscBsiUtilConnect()**.

Return Codes:
```
BSI_OK
BSI_BAD_HANDLE
BSI_CARD_REMOVED
BSI_UNKNOWN_ERROR
```

E.4.7 gscBsiUtilGetExtendedErrorText()

Purpose: When a BSI function call returns an error, an application can make a subsequent call to this function to receive additional implementation specific error information, if available.

Prototype:

```
unsigned long gscBsiUtilGetExtendedErrorText(
   IN UTILCardHandle    hCard,
   OUT char             *uszErrorText[255]
);
```

Parameters:

hCard: Card connection handle **gscBsiUtilConnect()**.

uszErrorText: A fixed length buffer containing an implementation specific error text string. The text string is NULL-terminated, and has a maximum length of 255 characters including the NULL terminator. The calling application must allocate a buffer of 255 bytes. If an extended error text string is not available, this function returns a NULL string and BSI_NO_TEXT_AVAILABLE.

Return Codes:

BSI_OK
BSI_BAD_HANDLE
BSI_BAD_PARAM
BSI_NO_TEXT_AVAILABLE
BSI_UNKNOWN_ERROR

E.4.8 gscBsiUtilGetReaderList()

Purpose: Retrieves the list of available readers.

Prototype:
```
unsigned long gscBsiUtilGetReaderList(
   IN/OUT unsigned char *uszReaderList,
   IN/OUT unsigned long *unReaderListLen
);
```

Parameters:

uszReaderList: Reader list buffer. The reader list is returned as a multi-string, each reader name terminated by a '\0'. The list itself is terminated by an additional trailing '\0' character.

unReaderListLen: Reader list length in bytes including all terminating '\0' characters.

Return Codes:
```
BSI_OK
BSI_BAD_PARAM
BSI_INSUFFICIENT_BUFFER
BSI_UNKNOWN_ERROR
```

E.4.9 gscBsiUtilPassthru()

Purpose: Allows a client application to send a "raw" APDU through the BSI directly to the card and receive the APDU-level response.

Prototype:

```
unsigned long gscBsiUtilPassthru(
   IN UTILCardHandle    hCard,
   IN unsigned char     *uszCardCommand,
   IN unsigned long     unCardCommandLen,
   IN/OUT unsigned char *uszCardResponse,
   IN/OUT unsigned long *unCardResponseLen
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect().**
uszCardCommand:	The APDU to be sent to the card.
unCardCommandLen:	Length of the APDU string to be sent.
uzsCardResponse:	Pre-allocated buffer for the APDU response from the card. The response must include the status bytes SW1 and SW2 returned by the card. If the size of the buffer is insufficient, the SPS shall return truncated response data and the return code BSI_INSUFFICIENT_BUFFER.
unCardResponseLen:	Length of the APDU response. If the size of the uszCardResponse buffer is insufficient, the SPS shall return the correct size in this field.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_PARAM
BSI_INSUFFICIENT_BUFFER
BSI_CARD_REMOVED
BSI_UNKNOWN_ERROR
```

E.4.10 gscBsiUtilReleaseContext()

Purpose: Terminate a session with the card.

Prototype:

```
unsigned long gscBsiUtilReleaseContext(
   IN UTILCardHandle    hCard,
   IN unsigned char     *uszAID,
   IN unsigned long     unAIDLen
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect().**
uszAID:	Target container AID value.
unAIDLen:	AID value length in bytes.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_CARD_REMOVED
BSI_UNKNOWN_ERROR
```

E.5 Smart Card Generic Container Provider Module Interface Definition

E.5.1 gscBsiGcDataCreate()

Purpose: Create a new data item in {Tag, Length, Value} format in the selected container.

Prototype:

```
unsigned long gscBsiGcDataCreate(
   IN UTILCardHandle    hCard,
   IN unsigned char     *uszAID,
   IN unsigned long     unAIDLen,
   IN GCtag             ucTag,
   IN unsigned char     *uszValue,
   IN unsigned long     unValueLen
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect().**
uszAID:	Target container AID value.
unAIDLen:	AID value length in bytes.
ucTag:	Tag of data item to store.
uszValue:	Data value to store.
unValueLen:	Data value length in bytes.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_NO_MORE_SPACE
BSI_TAG_EXISTS
BSI_CREATE_ERROR
BSI_UNKNOWN_ERROR
```

E.5.2 gscBsiGcDataDelete()

Purpose: Delete the data item associated with the tag value in the specified container.

Prototype:

```
unsigned long gscBsiGcDataDelete(
   IN UTILCardHandle    hCard,
   IN unsigned char     *uszAID,
   IN unsigned long     unAIDLen,
   IN GCtag             ucTag
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect().**
uszAID:	Target container AID value.
unAIDLen:	AID value length in bytes.
ucTag:	Tag of data item to delete.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_BAD_TAG
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_DELETE_ERROR
BSI_UNKNOWN_ERROR
```

E.5.3 gscBsiGcGetContainerProperties()

Purpose: Retrieves the properties of the specified container. Access Control Rules are common to all data items managed by the selected container.

Prototype:

```
unsigned long gscBsiGcGetContainerProperties(
   IN UTILCardHandle    hCard,
   IN unsigned char     *uszAID,
   IN unsigned long     unAIDLen,
   OUT Gcacr            *strctGCacr,
   OUT GCContainerSize  strctContainerSizes,
   OUT unsigned char    *uszContainerVersion
);
```

Parameters:

hCard: Card connection handle from **gscBsiUtilConnect()**.

uszAID: Target container AID value.

unAIDLen: AID value length in bytes.

strctGCacr: Structure indicating access control conditions for all operations. The range of possible values for the members of this structure is defined in Table 3.1 (Section 3.1). The allowable ACRs for each function are listed in Table 3.2 (Section 3.2):

```
struct GCacr {
   unsigned long        unCreateACR;
   unsigned long        unDeleteACR;
   unsigned long        unReadTagListACR;
   unsigned long        unReadValueACR;
   unsigned long        unUpdateValueACR;
};
```

strctContainerSizes: For Virtual Machine cards, the size(in bytes) of the container specified by `uszAID. unMaxNbDataItems` is the size of the T-Buffer, and `unMaxValueStorageSize` is the size of the V-Buffer. For file system cards that cannot calculate these values, both fields of this structure will be set to 0.

```
struct GCContainerSize {
   unsigned long        unMaxNbDataItems;
   unsigned long        unMaxValueStorageSize;

}
```

uszContainerVersion: Version of the container. The format of this value is application dependent. In cases where the card cannot return a container version, this string will contain only the NULL terminator "\0".

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_UNKNOWN_ERROR
```

E.5.4 gscBsiGcReadTagList()

Purpose: Return the list of tags in the selected container.

Prototype:

```
unsigned long gscBsiGcReadTagList(
   IN UTILCardHandle    hCard,
   IN unsigned char     *uszAID,
   IN unsigned long     unAIDLen,
   IN/OUT Gctag         *TagArray,
   IN/OUT unsigned long *unNbTags
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect().**
uszAID:	Target container AID value.
unAIDLen:	AID value length in bytes.
TagArray:	An array containing the list of tags for the selected container.
unNbTags:	Number of tags in **TagArray**.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_INSUFFICIENT_BUFFER
BSI_UNKNOWN_ERROR
```

E.5.5 gscBsiGcReadValue()

Purpose: Returns the Value associated with the specified Tag.

Prototype:

```
unsigned long gscBsiGcReadValue(
   IN UTILCardHandle    hCard,
   IN unsigned char     *uszAID,
   IN unsigned long     unAIDLen,
   IN GCtag             ucTag,
   IN/OUT unsigned char *uszValue,
   IN/OUT unsigned long *unValueLen
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect().**
uszAID:	Target container AID value.
unAIDLen:	AID value length in bytes.
ucTag:	Tag value of data item to read.
uszValue:	Value associated with the specified tag. The caller must allocate the buffer.
unValueLen:	Size of the buffer allocated by the caller to hold the returned Value (input). Size of the Value returned (output).

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_BAD_TAG
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_INSUFFICIENT_BUFFER
BSI_READ_ERROR
BSI_UNKNOWN_ERROR
```

E.5.6 gscBsiGcUpdateValue()

Purpose: Updates the Value associated with the specified Tag.

Prototype:

```
unsigned long gscBsiGcUpdateValue(
   IN UTILCardHandle    hCard,
   IN unsigned char     *uszAID,
   IN unsigned long     unAIDLen,
   IN GCtag             ucTag,
   IN unsigned char     *uszValue,
   IN unsigned long     unValueLen
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect().**
uszAID:	Target container AID value.
unAIDLen:	AID value length in bytes.
ucTag:	Tag of data item to update.
uszValue:	New Value of the data item.
unValueLen:	Length in bytes of the new Value.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_BAD_TAG
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_NO_MORE_SPACE
BSI_UPDATE_ERROR
BSI_UNKNOWN_ERROR
```

E.6 Smart Card Cryptographic Provider Module Interface Definition

E.6.1 gscBsiGetChallenge()

Purpose: Retrieves a randomly generated challenge from the card as the first step of a challenge-response authentication protocol between the client application and the card. The client subsequently encrypts the challenge using a symmetric key and returns the encrypted random challenge to the card through a call to **`gscBsiUtilAcquireContext`**`()` in the `uszAuthValue` field of a BSIAuthenticator structure.

Prototype:

```
unsigned long gscBsiGetChallenge(
   IN UTILCardHandle    hCard,
   IN unsigned char     *uszAID,
   IN unsigned long     unAIDLen,
   IN/OUT unsigned char *uszChallenge,
   IN/OUT unsigned long *unChallengeLen
);
```

Parameters:

`hCard`:	Card connection handle from **`gscBsiUtilConnect()`**.
`uszAID`:	Target container AID value.
`unAIDLen`:	AID value length in bytes.
`uszChallenge`:	Random challenge returned from the card.
`unChallengeLen`:	Length of random challenge in bytes.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_INSUFFICIENT_BUFFER
BSI_UNKNOWN_ERROR
```

E.6.2 gscBsiSkiInternalAuthenticate()

Purpose: Computes a symmetric key cryptogram in response to a challenge. In cases where the card reader authenticates the card, this function does not return a cryptogram. In these cases a BSI_TERMINAL_AUTH will be returned if the card reader successfully authenticates the card. BSI_ACCESS_DENIED is returned if the card reader fails to authenticate the card.

Prototype:

```
unsigned long gscBsiSkiInternalAuthenticate(
   IN UTILCardHandle    hCard,
   IN unsigned char     *uszAID,
   IN unsigned long     unAIDLen,
   IN unsigned char     ucAlgoID,
   IN unsigned char     *uszChallenge,
   IN unsigned long     unChallengeLen,
   OUT unsigned char    *uszCryptogram,
   IN/OUT unsigned long *unCryptogramLen
);
```

Parameters:

hCard: Card connection handle from **gscBsiUtilConnect()**.

uszAID: SKI provider module AID value.

unAIDLen: AID value length in bytes.

ucAlgoID: Identifies the cryptographic algorithm that the card must use to encrypt the challenge. All conformant implementations shall, at a minimum, support DES3-ECB (Algorithm Identifier 0x81) and DES3-CBC (Algorithm Identifier 0x82). Implementations may optionally support other cryptographic algorithms.

uszChallenge: Challenge generated by the client application and submitted to the card.

unChallengeLen: Length of the challenge in bytes.

uszCryptogram: The cryptogram computed by the card.

unCryptogramLen: Length of the cryptogram computed by the card in bytes.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_BAD_ALGO_ID
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_TERMINAL_AUTH
BSI_INSUFFICIENT_BUFFER
BSI_UNKNOWN_ERROR
```

E.6.3 gscBsiPkiCompute()

Purpose: Performs a private key computation on the message digest using the private key associated with the specified AID.

Prototype:

```
unsigned long gscBsiPkiCompute(
   IN UTILCardHandle    hCard,
   IN unsigned char     *uszAID,
   IN unsigned long     unAIDLen,
   IN unsigned char     ucAlgoID,
   IN unsigned char     *uszMessage,
   IN unsigned long     unMessageLen,
   IN/OUT unsigned char *uszResult,
   IN/OUT unsigned long *unResultLen
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
uszAID:	PKI provider module AID value.
unAIDLen:	AID value length in bytes.
ucAlgoID:	Identifies the cryptographic algorithm that will be used to generate the signature. All conformant implementations shall, at a minimum, support RSA_NO_PAD (Algorithm Identifier 0xA3). Implementations may optionally support other algorithms.
uszMessage:	The hash of the message to be signed.
unMessageLen:	Length of hashed message to be signed, in bytes.
uszResult:	Buffer containing the signature.
unResultLen:	Length of the signature buffer in bytes.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_BAD_ALGO_ID
BSI_CARD_REMOVED
BSI_ACCESS_DENIED
BSI_NO_CARDSERVICE
BSI_INSUFFICIENT_BUFFER
BSI_UNKNOWN_ERROR
```

E.6.4 gscBsiPkiGetCertificate()

Purpose: Reads the certificate from the card.

Prototype:

```
unsigned long gscBsiPkiGetCertificate(
   IN UTILCardHandle    hCard,
   IN unsigned char     *uszAID,
   IN unsigned long     unAIDLen,
   OUT unsigned char    *uszCertificate,
   IN/OUT unsigned long *unCertificateLen
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
uszAID:	PKI provider module AID value.
unAIDLen:	AID value length in bytes.
uszCertificate:	Buffer containing the certificate.
unCertificateLen:	Length of the certificate buffer in bytes.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_READ_ERROR
BSI_INSUFFICIENT_BUFFER
BSI_UNKNOWN_ERROR
```

E.6.5 gscBsiGetCryptoProperties()

Purpose: Retrieves the Access Control Rules and private cryptographic key length managed by the PKI provider module.

Prototype:

```
unsigned long gscBsiGetCryptoProperties(
   IN UTILCardHandle    hCard,
   IN unsigned char     *uszAID,
   IN unsigned long     unAIDLen,
   OUT CRYPTOacr        *strctCRYPTOacr,
   OUT unsigned long    *unKeyLen
);
```

Parameters:

hCard: Card connection handle from **gscBsiUtilConnect()**.

uszAID: AID of the PKI provider.

unAIDLen: Length of the AID of the PKI provider, in bytes.

strctCRYPTOacr: Structure indicating access control conditions for all operations. The range of possible values for the members of this structure are defined in Table 3.1 (Section 3.1), and the allowable ACRs for each function in Table 3.3 (Section 3.2):

```
struct CRYPTOacr {
   unsigned long        unGetChallengeACR;
   unsigned long        unInternalAuthenticateACR;
   unsigned long        unPkiComputeACR;
   unsigned long        unReadCertificateACR;
};
```

unKeyLen: Length of the private key managed by the PKI provider.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_UNKNOWN_ERROR
```

Appendix F—Java Language Binding for BSI Services

This appendix defines the Java language binding for the BSI services. This set of services consists of 21 Java functions derived from the IDL specification (Chapter 4). Similar to the IDL specification, the Java translation is grouped into three functional modules as follows:

- A Smart Card Utility Provider Module
- A Smart Card Generic Container Provider Module
- A Smart Card Cryptographic Provider Module.

F.1 Smart Card Utility Provider Module Interface Definition

F.1.1 gscBsiUtilAcquireContext()

Purpose: This function shall establish a session with the smart card by submitting the appropriate Authenticator in the BSIAuthenticator object. For ACRs requiring external authentication (XAUTH), the uszAuthValue instance variable of the BSIAuthenticator object must contain a cryptogram calculated by encrypting a random challenge.

Prototype:

```
int gscBsiUtilAcquireContext(
   IN int               hCard,
   IN String            uszAID,
   IN Vector            strctAuthenticator
);
```

Parameters:

hCard: Card connection handle from **gscBsiUtilConnect()**.

uszAID: AID of the target service provider or container.

strctAuthenticator: Vector of BSIAuthenticator objects containing the authenticator(s) specified by the ACR required to access a value in the container. The authenticator is returned by **gscBsiGcGetContainerProperties()**. The calling application is responsible for constructing this object.

The BSIAuthenticator class is defined as follows:

```
public class BSIAuthenticator {
   protected int        unACRtype;
   protected String     uszAuthValue;
   protected String     uszKeyValue;
   protected boolean    bWriteKey;

   //CONSTRUCTORS:
   public BSIAuthenticator()
   public BSIAuthenticator(
        int             acrtype,
        String          authV,
        boolean         thisAccess
   )

   //ACCESSORS:
   public int           getACRtype()
   public void          setACRtype(int type)
   public String        getAuthValue()
   public void          setAuthValue(String value)
   public boolean       getWriteKey()
   public void          setWriteKey(boolean b)
}
```

The instance variables of the BSIAuthenticator class are:

unACRtype:	Access Control Rule (see Table 3.1 in Section 3.1).
uszAuthValue:	Authenticator, can be an external authentication cryptogram or PIN.
uszKeyValue:	Cryptographic authentication key.
bWriteKey:	At the smart card level, the *read* and *write* privileges are granted sequentially therefore each granted privilege and its acquired security context is associated with either a *read* or a *write* operation through this instance variable of the BSIAuthenticator. If set to TRUE, this ACR controls write access. If set to FALSE, it controls read access.

Return Codes:

```
BSI_OK
BSI_BAD_AID
BSI_BAD_HANDLE
BSI_ACR_NOT_AVAILABLE
BSI_BAD_AUTH
BSI_CARD_REMOVED
BSI_PIN_LOCKED
BSI_UNKNOWN_ERROR
```

F.1.2 gscBsiUtilConnect()

Purpose: Establish a logical connection with the card in a specified reader. `BSI_TIMEOUT_ERROR` will be returned if a connection cannot be established within a specified time. The timeout value is implementation dependent.

Prototype:

```
int gscBsiUtilConnect(
   IN String            uszReaderName,
   OUT int              hCard
);
```

Parameters:

hCard:	Card connection handle.
uszReaderName:	Name of the reader that the card is inserted into. If this field is an empty String, the SPS shall attempt to connect to the card in the first available reader, as returned by a call to the BSI's function **gscBsiUtilGetReaderList()**

Return Codes:

```
BSI_OK
BSI_UNKNOWN_READER
BSI_CARD_ABSENT
BSI_TIMEOUT_ERROR
BSI_UNKNOWN_ERROR
```

F.1.3 gscBsiUtilDisconnect()

Purpose: Terminate a logical connection to a card.

Prototype:
```
int gscBsiUtilDisconnect(
   IN int                hCard
);
```

Parameters: **hCard**: Card connection handle from **gscBsiUtilConnect()**.

Return Codes:
```
BSI_OK
BSI_BAD_HANDLE
BSI_CARD_REMOVED
BSI_UNKNOWN_ERROR
```

F.1.4 gscBsiUtilGetVersion()

Purpose: Returns the BSI implementation version.

Prototype:

```
int gscBsiUtilGetVersion(
   OUT String           uszVersion
);
```

Parameters:

uszVersion:	The BSI and SCSPM's version formatted as "major,minor,revision, build_number\0". The value for an SCSPM compliant with this version of the GSC-IS is "2,0,0,<build number>\0". The build number field is vendor/implementation dependent.

Return Codes:

```
BSI_OK
BSI_UNKNOWN_ERROR
```

F.1.5 gscBsiUtilGetCardProperties()

Purpose: Retrieves version and capability information for the card.

Prototype:

```
int gscBsiUtilGetCardProperties(
   IN int               hCard,
   OUT String           uszCCCUniqueID,
   OUT int              unCardCapability
);
```

Parameters:

hCard: Card connection handle from **gscBsiUtilConnect().**

uszCCCUniqueID: Buffer for the Card Capability Container version .

unCardCapability: Bit mask value defining the providers supported by the card. The bit masks represent the Generic Container Data Model, the Generic Container Data Model Extended, the Symmetric Key Interface, and the Public Key Interface providers respectively:

```
public interface BSICardCapabilities
{
   public final static int BSI_GCCDM         = 0x00000001;
   public final static int BSI_SKI           = 0x00000002;
   public final static int BSI_PKI           = 0x00000004;
   public final static int BSI_GCCDM_EXT     = 0x00000008;
   public final static int BSI_SKI_EXT       = 0x00000010;
   public final static int BSI_PKI_EXT       = 0x00000020;
}
```

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_UNKNOWN_ERROR
```

F.1.6 gscBsiUtilGetCardStatus()

Purpose: Checks whether a given card handle is associated with a card that is inserted into a powered up reader.

Prototype:
```
int gscBsiUtilGetCardStatus(
   IN int                hCard
);
```

Parameters: **hCard**: Card connection handle from **gscBsiUtilConnect()**.

Return Codes:
```
BSI_OK
BSI_BAD_HANDLE
BSI_CARD_REMOVED
BSI_UNKNOWN_ERROR
```

F.1.7 gscBsiUtilGetExtendedErrorText()

Purpose: When a BSI function call returns an error, an application can make a subsequent call to this function to receive additional error information from the card reader driver layer, if available. Since the GSC-IS architecture accommodates different card reader driver layers, the error text information will be dependent on the card reader driver layer used in a particular implementation. This function must be called immediately after the error has occurred.

Prototype:

```
int gscBsiUtilGetExtendedErrorText(
   IN int                hCard,
   OUT String            uszErrorText
);
```

Parameters:

hCard: Card connection handle **gscBsiUtilConnect()**.

uszErrorText: A String of maximum 255 characters including the null terminator, containing an implementation specific error text. If an extended error text string is not available, this function returns an empty string and BSI_NO_TEXT_AVAILABLE.

Return Codes:

BSI_OK
BSI_BAD_HANDLE
BSI_NO_TEXT_AVAILABLE
BSI_UNKNOWN_ERROR

F.1.8 gscBsiUtilGetReaderList()

Purpose: Retrieves the list of available readers.

Prototype:

```
int gscBsiUtilGetReaderList(
   OUT Vector                vReaderList
);
```

Parameters:

vReaderList:	Vector of Strings containing a list of the available readers. The Vector shall be constructed by the calling application, using the default constructor.

Return Codes:

```
BSI_OK
BSI_UNKNOWN_ERROR
```

F.1.9 gscBsiUtilPassthru()

Purpose: Allows a client application to send a "raw" APDU through the BSI directly to the card and receive the APDU-level response.

Prototype:

```
int gscBsiUtilPassthru(
   IN int                hCard,
   IN String             uszCardCommand,
   OUT String            uszCardResponse
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect().**
uszCardCommand:	The APDU to be sent to the card.
uzsCardResponse:	The APDU response from the card. The response must include the status bytes SW1 and SW2 returned by the card.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_PARAM
BSI_CARD_REMOVED
BSI_UNKNOWN_ERROR
```

F.1.10 gscBsiUtilReleaseContext()

Purpose: Terminate a session with the card.

Prototype:

```
int gscBsiUtilReleaseContext(
   IN int                hCard,
   IN String             uszAID
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect().**
uszAID:	Target container AID value.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_CARD_REMOVED
BSI_UNKNOWN_ERROR
```

F.2 Smart Card Generic Container Provider Module Interface Definition

F.2.1 gscBsiGcDataCreate()

Purpose: Create a new data item in {Tag, Length, Value} format in the selected container.

Prototype:

```
int gscBsiGcDataCreate(
   IN int               hCard,
   IN String            uszAID,
   IN byte              ucTag,
   IN String            uszValue
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect().**
uszAID:	Target container AID value.
ucTag:	Tag of data item to store.
uszValue:	Data value to store.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_NO_MORE_SPACE
BSI_TAG_EXISTS
BSI_CREATE_ERROR
BSI_UNKNOWN_ERROR
```

F.2.2 gscBsiGcDataDelete()

Purpose: Delete the data item associated with the tag value in the specified container.

Prototype:
```
int gscBsiGcDataDelete(
   IN int               hCard,
   IN String            uszAID,
   IN byte              ucTag
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
uszAID:	Target container AID value.
ucTag:	Tag of data item to delete.

Return Codes:
```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_TAG
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_DELETE_ERROR
BSI_UNKNOWN_ERROR
```

F.2.3 gscBsiGcGetContainerProperties()

Purpose: Retrieves the properties of the specified container. ACRs are common to all data items managed by the selected container.

Prototype:

```
int gscBsiGcGetContainerProperties(
   IN int               hCard,
   IN String            uszAID,
   OUT GCacr            strctGCacr,
   OUT GCContainerSize  strctContainerSizes,
   OUT String           uszContainerVersion
);
```

Parameters:

hCard: Card connection handle from **gscBsiUtilConnect()**.

uszAID: Target container AID value.

strctGCacr: Object indicating access control conditions for all operations. The range of possible values for the instance variables of this object is defined in Table 3.1 (Section 3.1). The allowable ACRs for each function are listed in Table 3.2 (Section 3.2):

The GCacr class is defined as follows:

```
public class GCacr
{
   protected int unCreateACR;
   protected int unDeleteACR;
   protected int unReadTagListACR;
   protected int unReadValueACR;
   protected int unUpdateValueACR;

   //CONSTRUCTORS
   public GCacr()
   public GCacr(int c, int d, int rt, int rv, int u)

   //ACCESSORS
   public void setCreateACR(int  i)
   public void setDeleteACR(int  i)
   public void setReadTagListACR(int  i)
   public void setReadValueACR(int  i)
   public void setUpdateValueACR(int  i)
   public int getCreateACR()
   public int getDeleteACR()
   public int getReadTagListACR()
   public int getReadValueACR()
   public int getUpdateValueACR()
}
```

strctContainerSizes: The size (in bytes) of the container specified by uszAID.

```
public class GCContainerSize
{
      protected int unMaxNbDataItems;
      protected int unMaxValueStorageSize;

   //CONSTRUCTORS
      public GCContainerSize ()
      public GCContainerSize (int i, int s)

      //ACCESSORS
      public void setMaxNbDataItems(int  i)
      public void setMaxValueStorageSize(int  i)
      public int getMaxNbDataItems ()
      public int getMaxValueStorageSize ()
}
```

uszContainerVersion: Version of the container. The format of this value is application dependent.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_UNKNOWN_ERROR
```

F.2.4 gscBsiGcReadTagList()

Purpose: Return the list of tags in the selected container.

Prototype:

```
int gscBsiGcReadTagList(
   IN int               hCard,
   IN String            uszAID,
   OUT Vector           TagArray
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect().**
uszAID:	Target container AID value.
TagArray:	A Vector containing the list of tags for the selected container.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_UNKNOWN_ERROR
```

F.2.5 gscBsiGcReadValue()

Purpose: Returns the Value associated with the specified Tag.

Prototype:

```
int gscBsiGcReadValue(
   IN int               hCard,
   IN String            uszAID,
   IN byte              ucTag,
   OUT String           uszValue
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect().**
uszAID:	Target container AID value.
ucTag:	Tag value of data item to read.
uszValue:	Value associated with the specified tag.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_TAG
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_READ_ERROR
BSI_UNKNOWN_ERROR
```

F.2.6 gscBsiGcUpdateValue()

Purpose: Updates the Value associated with the specified Tag.

Prototype:

```
int gscBsiGcUpdateValue(
   IN int               hCard,
   IN String            uszAID,
   IN byte              ucTag,
   IN String            uszValue
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect().**
uszAID:	Target container AID value.
ucTag:	Tag of data item to update.
uszValue:	New Value of the data item.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_BAD_TAG
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_NO_MORE_SPACE
BSI_UPDATE_ERROR
BSI_UNKNOWN_ERROR
```

F.3 Smart Card Cryptographic Provider Module Interface Definition

F.3.1 gscBsiGetChallenge()

Purpose: Retrieves a randomly generated challenge from the card as the first step of a challenge-response authentication protocol between the client application and the card. The client subsequently encrypts the challenge using a symmetric key and returns the encrypted random challenge to the card through a call to **gscBsiUtilAcquireContext()** in the uszAuthValue field of a BSIAuthenticator object.

Prototype:

```
int gscBsiGetChallenge(
   IN int                hCard,
   IN String             uszAID,
   OUT String            uszChallenge
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
uszAID:	Target container AID value.
uszChallenge:	Random challenge returned from the card.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_UNKNOW_ERROR
```

F.3.2 gscBsiSkiInternalAuthenticate()

Purpose: Computes a symmetric key cryptogram in response to a challenge. In cases where the card reader authenticates the card, this function does not return a cryptogram. In these cases a `BSI_TERMINAL_AUTH` will be returned if the card reader successfully authenticates the card. `BSI_ACCESS_DENIED` is returned if the card reader fails to authenticate the card.

Prototype:

```
int gscBsiSkiInternalAuthenticate(
   IN int               hCard,
   IN String            uszAID,
   IN byte              ucAlgoID,
   IN String            uszChallenge,
   OUT String           uszCryptogram
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
uszAID:	SKI provider module AID value.
ucAlgoID:	Identifies the cryptographic algorithm that the card must use to encrypt the challenge. All conformant implementations shall, at a minimum, support `DES3-ECB` (Algorithm Identifier 0x81) and `DES3-CBC` (Algorithm Identifier 0x82). Implementations may optionally support other cryptographic algorithms.
uszChallenge:	Challenge generated by the client application and submitted to the card.
uszCryptogram:	The cryptogram computed by the card.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_BAD_ALGO_ID
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_UNKNOWN_ERROR
```

F.3.3 gscBsiPkiCompute()

Purpose: Performs a private key computation on the message digest using the private key associated with the specified AID.

Prototype:

```
int gscBsiPkiCompute(
   IN int               hCard,
   IN String            uszAID,
   IN byte              ucAlgoID,
   IN String            uszMessage,
   OUT String           uszResult
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
uszAID:	PKI provider module AID value.
ucAlgoID:	Identifies the cryptographic algorithm that will be used to generate the signature. All conformant implementations shall, at a minimum, support RSA_NO_PAD (Algorithm Identifier 0xA3). Implementations may optionally support other algorithms.
uszMessage:	The message digest to be signed.
uszResult:	Buffer containing the signature.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_BAD_ALGO_ID
BSI_CARD_REMOVED
BSI_ACCESS_DENIED
BSI_NO_CARDSERVICE
BSI_UNKNOWN_ERROR
```

F.3.4 gscBsiPkiGetCertificate()

Purpose: Reads the certificate from the card.

Prototype:

```
int gscBsiPkiGetCertificate(
   IN int                hCard,
   IN String             uszAID,
   OUT String            uszCertificate
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
uszAID:	PKI provider module AID value.
uszCertificate:	Buffer containing the certificate.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_READ_ERROR
BSI_UNKNOWN_ERROR
```

F.3.5 gscBsiGetCryptoProperties()

Purpose: Retrieves the Access Control Rules associated with the PKI provider module.

Prototype:

```
int gscBsiGetCryptoProperties(
   IN int               hCard,
   IN String            uszAID,
   OUT CRYPTOacr        strctCRYPTOacr
);
```

Parameters:

hCard: Card connection handle from **gscBsiUtilConnect()**.

uszAID: AID of the PKI provider.

strctCRYPTOacr: Object indicating access control conditions for all operations. The range of possible values for the instance variables of this object are defined in Table 3.1 (Section 3.1), and the allowable ACRs for each function in Table 3.3 (Section 3.2):

```
public class CRYPTOacr
{
   protected int unGetChallengeACR;
   protected int unInternalAuthenticateACR;
   protected int unPkiComputeACR;
   protected int unReadCertificateACR;

   //CONSTRUCTORS
   public CRYPTOacr()
   public CRYPTOacr(int ch, int ia, int pc, int rv)

   //ACCESSORS
   public void setGetChallengeACR(int i)
   public void setInternalAuthenticateACR(int i)
   public void setPkiComputeACR(int i)
   public void setReadCertificateACR(int i)
   public int getGetChallengeACR()
   public int getInternalAuthenticateACR()
   public int getPkiComputeACR()
   public int getReadCertificateACR()
}
```

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_UNKNOWN_ERROR
```

Appendix G—Descriptor Code Derivations and Extended Descriptions

This appendix discusses how the default descriptor codes are derived and provides extended descriptions for each code. In the following descriptions, refer to Figure G-1 for details on how specific values were derived and/or calculated.

0x11 Challenge

Card Random Number: a designated number of random byte values generated by the card.

0x13 6 MSB of Cryptogram

The 6 most significant bytes of a cryptogram, generated by encrypting a random number with a designated key: DES encryption for 8-byte keys and 3DES encryption for 16-byte keys.

0x14 3 LSB of Cryptogram

The 3 least significant bytes of a cryptogram, generated by encrypting a random number with a designated key: DES encryption for 8-byte keys and 3DES encryption for 16-byte keys.

0x6 MSB of Offset

The most significant byte of the file offset in bytes.

0x7 LSB of Offset

The least significant byte of the file offset in bytes.

0xD EF

The File ID of a file.

0x1E System ID

The system ID value used by Microsoft Windows.

0x21 2 Byte FID

The 2-byte File Identifier of the file being accessed.

0x22 Short FID

The 5 least significant bits of the 2-byte File Identifier of the file being accessed.

0x23 File Name

0x29 3 LSB MAC

MAC Calculation 1: *m*: The final result is truncated to the 24 least significant bits.

0x2C Key File Identifier

The 2-byte File Identifier of the file of the key being referenced.

0x2D 3 LSB MAC

MAC Calculation 2: Uses default values for all parameters.

0x2E 4 MSB of Cryptogram

The 4 most significant bytes of a cryptogram, generated by encrypting a random number with a designated key: DES encryption for 8-byte keys and 3DES encryption for 16-byte keys.

0x2F 8 Byte Random Number

0x38 4 LSB of Cryptogram

The 4 least significant bytes of a cryptogram, generated by 3DES encrypting a terminal random number with a temporary session key computed per descriptor code 0x4F.

0x3E Pad=00(8)

Data padded at the end with low values to the 8-byte boundary (ISO 9797 paragraph 5.1 method 1).

0x43 Terminal Random Number

A designated number of random byte values generated on the terminal by the BSI.

0x45 Key File Short ID

The 5 least significant bits of the 2-byte File Identifier of the file of the key being referenced.

0x46 MSB of Offset in Words

The most significant byte of the file offset in 4 byte words.

0x47 LSB of Offset in Words

The least significant byte of the file offset in 4 byte words.

0x48 Chaining Information

0x49 Block Length

0x4A TLV Format

0x4B Operation Mode

0x4C LOUD

Length of useful data: the number of bytes in the data transmitted, without counting any padding or added bytes.

0x4D 6 MSB CBC Cryptogram

The 6 MSBs of a DES CBC mode cryptogram.

0x4E 8 Byte Cryptogram

The cryptogram is generated by encryption of an 8-byte random number with a designated key, with DES encryption for an 8-byte key and 3DES encryption for a 16-byte key.

0x4F Session Key 1

The temporary session key is a 16-byte 3DES key composed of Key A, the 8 most significant bytes, and Key B, the 8 least significant bytes. They are generated as follows:

- An 8-byte random number is created from the 4 most significant bytes of a terminal random number, followed by the 4 most significant bytes of a card random number.
- Key A is generated by 3DES encrypting the random number using the 16-byte read or write key.
- Key B is generated by 3DES encrypting the random number using a 16-byte key where the first and last halves 16-byte read or write key are interchanged. That is, the 8 most significant bytes of the key are the 8 least significant bytes of the read or write key and the 8 least significant bytes of the key are the 8 most significant bytes of the read or write key.

0x50 Length + X

The number of bytes to be read or written plus X, where X is the smallest value such that Length + 3 + X is evenly divisible by 8.

0x51 Pad with X 0xFF Bytes

Pad data to be read or written with X 0xFF bytes where X is defined in descriptor code 0x50.

0x52 6 MSB of MAC

MAC Calculation 3:

- *K*: The MAC encryption key is the 8-byte or 16-byte read or write key.
- *A*: The MAC encryption algorithm is 64-bit block cipher algorithm, DES for a 8-byte key and 3DES for a 16-byte key (ISO 9797, paragraph 3.1.3).
- *Init*: The initial block is the 64-bit card random number.
- D_1: APDU bytes INS, P1, P2 followed by the LOUD byte, followed by the first 4 bytes of the data. If there are less than 4 bytes of data, the block is padded to 64 bits with 0xFF bytes.
- D_q: The last block of data is padded as needed to 64 bits with 0xFF bytes.
- *m*: The final result is truncated to the 48 most significant bits.

0x53 Length + 8

The number of bytes of data to be read or written plus 8.

0x54 8 byte MAC

MAC Calculation 4:

- *K*: The MAC encryption key is the 8 most significant bytes of the 16-byte read or write key.
- *Init*: The initial block is a 64-bit card random number for writing, or a 64-bit terminal random number for reading.
- D_1: APDU bytes CLA, INS, P1, P2, P3 padded to 64 bits with null bytes (ISO 9797 paragraph 5.1 method 1).
- *OP*: The optional process uses the MAC optional process derived key K_1 to decrypt the last block, then uses MAC encryption key *K* to encrypt that result (ISO 9797 paragraph A.1 optional process 1).
- K_1: The MAC optional process derived key is the 8 least significant bytes of the 16-byte read or write key.
- *m*: The final result is not truncated (the MAC is the entire 64 bits).

0x55 Session Key 2

The temporary session key is an 8-byte DES key generated as follows:

An 8-byte random number is created from the 4 most significant bytes of a card random number, followed by the 4 most significant bytes of a terminal random number. The session key is generated by encrypting the random number using the read or write key, with DES encryption for an 8-byte read or write key and 3DES encryption for a 16-byte read or write key.

0x56 TLV Command Data for Update Binary

Insert the tag byte 0x81, the length byte representing the number of data bytes to be written to the card, and the data bytes to be written.

0x57 TLV Response Data for Update Binary

Interpret as the tag byte 0x99, the length byte 0x02, and two data bytes representing ISO 7816-4 status bytes SW1 and SW2.

0x58 TLV Command Data for Read Binary

Insert the tag byte 0x97, the length byte 0x01, and a byte representing the number of bytes to be read from the card.

0x59 TLV Response Data for Read Binary

Interpret as the tag byte 0x81, the length byte representing the number of data byte read from the card, and the data bytes read.

0x5A 4 MSB of MAC (TLV Command MAC 1)

MAC Calculation 5:

- *K*: The MAC encryption key is the 8-byte session key previously derived per descriptor code 0x55.
- *Init*: The initial block contains the most significant byte of the card random number, followed by the next most significant byte of the card random number incremented by 1, followed by 6 null bytes.
- D_1: APDU bytes CLA, INS, P1, P2, followed by a 0x80 byte and padded to 64 bits with null bytes (ISO 9797 paragraph 5.1 method 2).
- $D_{2\text{-}q}$: The data is in TLV form as defined in descriptor code 0x56 for write or 0x58 for read.
- D_q: The last block of data is followed by a 0x80 byte padded as needed to 64 bits with null bytes (ISO 9797 paragraph 5.1 method 2).
- *m*: The final result is truncated to the 32 most significant bits. Insert the tag byte 0x8E, the length byte 0x04, and the 4 MAC bytes calculated above.

0x5B 4 MSB of MAC (TLV Response MAC 1)

MAC Calculation 6:

- *K*: The MAC encryption key is the 8-byte session key previously derived per descriptor code 0x55.
- *Init*: The initial block contains the most significant byte of the card random number, followed by the next most significant byte of the card random number incremented by 2, followed by 6 null bytes.
- $D_{1\text{-}q}$: The data is in TLV form as defined in descriptor code 0x57 for write or 0x59 for read.
- D_q: The last block of data is followed by a 0x80 byte padded as needed to 64 bits with null bytes (ISO 9797 paragraph 5.1 method 2).
- *m*: The final result is truncated to the 32 most significant bits. Interpret as the tag byte 0x8E, the length byte 0x04, and the 4 MAC bytes calculated above.

0x5C 4 MSB of MAC (TLV Command MAC 2)

MAC Calculation 7:

- *K*: The MAC encryption key is the 8-byte session key previously derived per descriptor code 0x55.
- *Init*: The initial block contains the most significant byte of the card random number, followed by the next most significant byte of the card random number incremented by 1, followed by 6 null bytes.

- *OE*: The optional encryption is performed on the initial block using the MAC encryption key *K*.
- D_1: APDU bytes CLA, INS, P1, P2, followed by a 0x80 byte and padded to 64 bits with null bytes (ISO 9797 paragraph 5.1 method 2).
- $D_{2\text{-}q}$: The data is in TLV form as defined in descriptor code 0x56 for write or 0x58 for read.
- D_q: The last block of data is followed by a 0x80 byte padded as needed to 64 bits with null bytes (ISO 9797 paragraph 5.1 method 2).
- *m*: The final result is truncated to the 32 most significant bits. Insert the tag byte 0x8E, the length byte 0x04, and the 4 MAC bytes calculated above.

0x5D 4 MSB of MAC (TLV Response MAC 2)

MAC Calculation 8:

- *K*: The MAC encryption key is the 8-byte session key previously derived per descriptor code 0x55.
- *Init*: The initial block contains the most significant byte of the card random number, followed by the next most significant byte of the card random number incremented by 2, followed by 6 null bytes.
- *OE*: The optional encryption is performed on the initial block using the MAC encryption key *K*.
- $D_{1\text{-}q}$: The data is in TLV form as defined in descriptor code 0x57 for write or 0x59 for read.
- D_q: The last block of data is followed by a 0x80 byte padded as needed to 64 bits with null bytes (ISO 9797 paragraph 5.1 method 2).
- *m*: The final result is truncated to the 32 most significant bits. Interpret as the tag byte 0x8E, the length byte 0x04, and the 4 MAC bytes calculated above.

0x5E 8 Byte MAC 2

MAC Calculation 9:

- *K*: The MAC encryption key is the 8 most significant bytes of the 16-byte read or write key.
- *A*: The MAC encryption algorithm is 64-bit block cipher algorithm, DES for a 8-byte key and 3DES for a 16-byte key (ISO 9797, paragraph 3.1.3).
- *Init*: The initial block is the 64-bit card random number.
- D_1: APDU bytes INS, P1, P2, followed by the LOUD byte, followed by the most significant and least significant bytes of the 2-byte File Identifier for the file being read from or written two, and padded to 64 bits with null bytes (ISO 9797 paragraph 5.1 method 1).
- D_q: The last block of data is padded as needed to 64 bits with 0xFF bytes.

- *m*: The final result is not truncated (the MAC is the entire 64 bits).

0x5F Key Number << 1

The number of the designated key is shifted 1 bit to the left (equal to multiplying the key number by 2).

0x60 Key Level Flag

If the designated key is at the current level (local) insert the byte 0x80; otherwise, if the key is at the root level (global) insert the byte 0x00.

0x61 Length + #Padding

The length of the data transmitted plus the number of padding bytes required to fill the designate block size: 64 bytes for an RSA 512-bit key, 96 bytes for an RSA 768-bit key, and 128 bytes for an RSA 1024-bit key.

0x62 Length of RSA Response

The response length is the same as the padded length of data sent to the card in an RSA Compute command.

0x63 RSA Response Data

Interpret as the return data from an RSA Compute command: a digital signature computed for a padded hash sent to the card, or a decrypted padded hash for a digital signature sent to the card.

0x64 Pad Hashed Data (PKCS#1)

MD5 hash: append to data 18 header bytes:
(0x10,0x04,0x00,0x05,0x05,0x02,0x0D,0xF7,0x86,0x48,0x86,0x2A,0x08,0x06,0x0C,0x30,0x20,0x30);
SHA-1 hash: append to data 15 header bytes:
(0x14,0x04,0x00,0x05,0x1A,0x02,0x03,0x0E,0x2B,0x05,0x06,0x09,0x30,0x21,0x30).

For all these hash algorithms, after appending the designated header bytes, append one 0x00 byte, followed by a variable number of 0xFF bytes followed by two bytes (0x01,0x00); the number of 0xFF bytes appended brings the total number of bytes, data plus padding, to the same length as that of the PKI key (64 bytes for a 512-bit key, 96 bytes for a 768-bit key, 128 bytes for a 1024-bit key).

0x65 Swap Data Bytes

The data bytes (either command data sent to the card or response data received from the card) are swapped, so that for N bytes, the 1st swapped byte is the Nth data byte, the 2nd swapped byte is the N-1st and so forth, until the Nth swapped byte is the 1st data byte.

0x66 TLV Key ID

Insert the tag byte 0x84, the length byte 0x01, and a byte representing the key identifier of the key used in the PKI computation.

0x67 TLV Hash Algorithm ID

Insert the tag byte 0x80, the length byte 0x01, and a byte representing the algorithm used to hash the data being signed: 0x32 for MD5 or 0x12 for SHA-1.

0x68 Key Length Padded Hash Data

The first byte of the data is a value equal to the length of the PKI key being used, followed by the 0x00 byte, followed by the swapped padded hashed data bytes, with padding per descriptor byte 0x65 and swapping per descriptor byte 0x64.

0x69 Key Length + 2

The value is the length of the PKI key being used plus 2.

0xA0-0x DF Implementation Dependant

0xE0 Output Data Bytes

Place Data Bytes (En) in data stream output to card.

0xE1-0x Ef Ex X Data Bytes

Next x bytes equal data constants.

0xF0-0x FC Reserved

0xFD Interpret Response

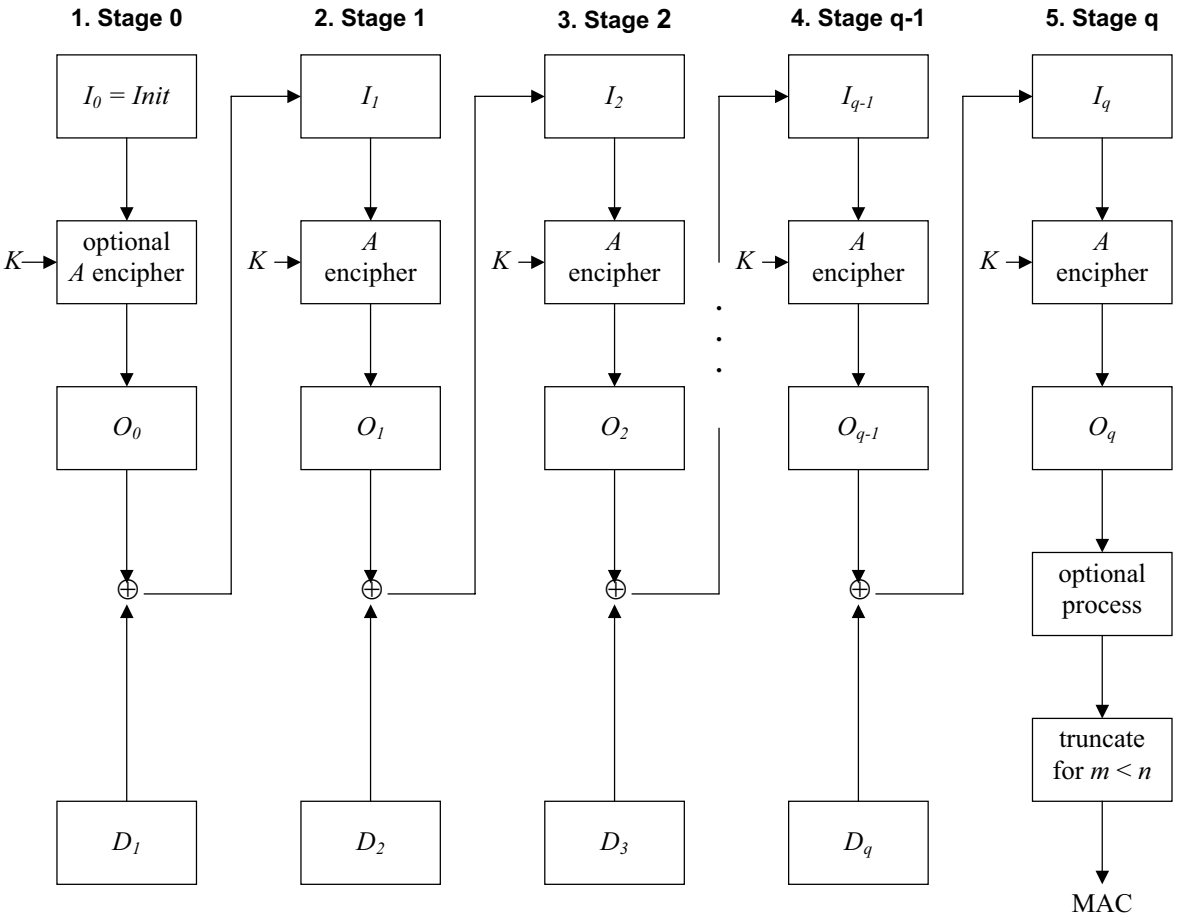
Following descriptor bytes are used to interpret response.

0xFE Command Not Available

Command is not available on card.

0xFF User Input Required

Parameter value must be supplied by user/program.



Legend:

I_i	input block ($i = 0,1,\ldots,q$)
Init	initial buffer ($= I_0$)
A	*n*-bit block cipher algorithm
O_i	output block ($i = 0,1,\ldots,q$)
K	key
D_i	data block ($i = 1,\ldots,q$)
$\oplus$	exclusive or

Init, the initial buffer, is defined in a card-specific manner.
D_1, the initial data block, is defined in a card-specific manner, and may include one or more of the initial bytes of transmitted data. D_2 through D_q contain the remaining bytes of transmitted data, divided into *n*-bit blocks, with padding of the last block, D_q, as needed.
m, the cryptogram truncation bits, may be either most or least significant bits.

Figure G-1: The MAC Calculation From ISO 9797, modified for ICC Card Secure Messaging

For the MAC Calculations 1-9, following the MAC Calculation schema diagrammed in G-1, the default values are:

- *n*: The MAC encryption block size is 64 bits.
- *K:* The MAC encryption key is the 16-byte session key previously derived per descriptor code 0x4F.
- *A*: The MAC encryption algorithm is 3DES 64-bit block cipher algorithm (ISO 9797, paragraph 3.1.3).
- *Init*: The initial block is filled to 64 bits with null bytes.
- *OE*: There is no optional encryption performed on the initial block.
- D_1: APDU bytes CLA, INS, P1, P2, P3 followed by the first 3 bytes of the data. If there are less than 3 bytes of data, the block is padded to 64 bits with null bytes (ISO 9797 paragraph 5.1, method 1).
- D_q: The last block of data is padded as needed to 64 bits with null bytes (ISO 9797 paragraph 5.1, method 1).
- *OP*: There is no optional process performed on the final block.
- *m*: The final result is truncated to the 24 most significant.

Appendix H—Acronyms

3DES	Triple Data Encryption Standard(also **DES3**)
ACK	Acknowledgment
ACR	Access Control Rule
AID	Application Identifier
ANSI	American National Standards Institute
APDU	Application protocol data unit
API	Applications Programming Interface
ATR	Answer-to-Reset
BSI	Basic Services Interface
C-APDU	Command APDU
CAPI	Cryptographic Applications Programming Interface
CCC	Card Capability Container
CEI	Card Edge Interface
CHV	Card Holder Verification
CLA	Class Byte of the Command Message
CLK	Clock
CSP	Cryptographic Service Provider
CT	Capability Tuple
CWI	Character Waiting Time Integer
CWT	Character Waiting Time
DAD	Destination Node Address
DES	Data Encryption Standard
DES3-CBC	Triple Data Encryption Standard in Cipher Block Chaining mode
DES3-ECB	Triple Data Encryption Standard in Electronic Codebook mode

EDC	Error Detection Code
ETU	Elementary Time Unit
FID	File ID
GCA	Generic Container Applet
GND	Ground
GSC	Government Smart Card, as defined in the Smart Access Common Identification Card Solicitation.
GSC-IS	Government Smart Card Interoperability Specification
HLSI	High Level Service Interface
HSM	Hardware Security Module
ICC	Integrated Circuit Card
IEC	International Electrotechnical Commission
IFD	Interface Device
IFS	Information Field Size associated with the *T=1* protocol.
IFSC	Information Field Size for the ICC associated with the *T=1* protocol.
IFSD	Information Field Size for the terminal associated with the *T=1* protocol.
IFSI	Information Field Size Integer associated with the *T=1* protocol.
INF	Information field associated with the *T=1* protocol.
INS	Instruction Byte of Command Message associated with the *T=0* and *T=1* protocol.
ISO	International Organization for Standardization
LEN	Length
LOUD	Length of useful data.
LRC	Longitudinal Redundancy Check associated with the *T=1* protocol.
MAC	Message Authentication Code
MSB	Most Significant Bit
NAD	Node address associated with the *T=1* protocol.

NAK	Negative ACK
OCF	Open Card Framework
P1(2)	Parameters used in the *T=0* and *T=1* protocol.
PCB	Protocol Control Byte
PC/SC	Personal Computer/Smart Card
PIN	Personal Identification Number
PKI	Public Key Infrastructure
PTS	Protocol Type Selection
PKCS	Public Key Cryptography Standards
R-APDU	Response APDU
RFU	Reserved for Future Use
RST	Reset
R-TPDU	Response TPDU
SAD	Source Node address associated with the T=1 protocol.
SCSPM	Smart Card Service Provider Module
SEIWG	Security Enterprise Integration Working Group
SKI	Symmetric Key Interface
SPI	Service Provider Interface
SPS	Service Provider Software
ST	Status Tuple
SW1(2)	Status Byte 1 (2)
T=0	Character-oriented asynchronous half duplex transmission protocol
T=1	Block-oriented asynchronous half duplex transmission protocol
TAL	Terminal Application Layer
TCK	Check Character

TLV	Tag-Length-Value
TPDU	Transport Protocol Data Unit
TTL	Terminal Transport Layer
USZ	Unsigned Zero-Terminated Character String
VCC	Supply Voltage
VCEI	Virtual Card Edge Interface
VM	Virtual Machine
VM CEI	Virtual Machine Card Edge Interface
VPP	Programming Voltage
WI	Waiting Time Integer
WTX	Waiting Time Extension
XSI	Extended Service Interface(s)