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

Interagency Report 6887 - 2003 Edition

Government Smart Card Interoperability Specification

Version 2.1

Teresa Schwarzhoff
Jim Dray
John Wack
Eric Dalci
Alan Goldfine
Michaela Iorga

July 16, 2003

NIST Interagency Report 6887 - 2003 Edition

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 – 2003 Edition, 247 pages (July 2003)

THIS PAGE INTENTIONALLY LEFT BLANK.

Foreword

(a) This section is non-normative and is provided for informational purposes only.

(b) 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 implemented 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 U.S. Government Smart Card (GSC) program.

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

In May 2000, GSA awarded the *Smart Access Common ID Card* contracts to five prime contractors to provide smart 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 established the Architecture Working Group (formerly known as the Technical Working Group), which consists of representatives of the contract awardees and federal agencies. The AWG, chaired and led by NIST, developed the Government Smart Card Interoperability Specification (GSC-IS), version 1.0. This specification defined 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. The AWG subsequently updated version 1.0 and released 2.0.

(c) Change Management, Requirements Definition, 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 will be posted on the NIST smart card program web site (see Section (d)).

As additional language bindings to the Basic Services Interface (see Section 1.3) are developed, they will be added to the GSC-IS.

In the longer term, it is expected that the GSA-IAB will be the governing body for the identification of the U.S. Government's requirements. Major releases of the GSC-IS will be determined by the GSC-IAB. NISTIR 6887 will be submitted for formal standardization to the ANSI approved formal standards setting body for smart card technology.

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)).

(d) Testing for Conformance

NIST is developing a comprehensive conformance test program in support of the GSC program. Products available will be subject to a formal certification process to validate conformance to the requirements of

the GSC-IS. The goal of the conformance tests is to determine whether or not a given Government Smart Card product conforms with the GSC Specification. Qualified laboratories will perform operational conformance testing. The GSC-IAB Conformance Committee is chaired by GSA, with representatives from the federal agencies and GSA contract awardees.

NIST is working on user guidance for achieving conformance certification for the various elements of the GSC-IS framework. This guidance will be posted at **http://smartcard.nist.gov**

(e) NIST Government Smart Card Program Web Site

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

- General program descriptions and updates
- The current 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, composed of representatives from the public and private sectors; 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 GSC Architecture Work Group (formerly known as Technical Working Group) of the Government Smart Card Interagency Advisory Board are particularly recognized. Chaired by the National Institute of Standards and Technology, the AWG was responsible for reviewing the original Government Smart Card Interoperability Specification. The AWG has been a major contributor to the development of this new version of the Government Smart Card Interoperability Specification. Special recognition is extended to the AWG.

THIS PAGE INTENTIONALLY LEFT BLANK.

Table of Contents

Appendices

List of Appendices

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] and 7816-8 [ISO8]. 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 "virtual 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 challenges associated with smart card technology. The original version of the GSC-IS (version 1.0, August 2000) 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 GSA *Smart Access Common Identification Card* contract.

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 [ISO3] 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
- Support for both of the Container Data Models defined in Chapter 8 and the appropriate Appendices
- At least one language binding for BSI Services, as defined in the Appendices.

A smart card that claims conformance to the GSC-IS must support each of the following:

- The Architectural Model as it relates to smart cards, i.e., as defined in sections 1, 4, 5, and 6 of Chapter 2
- The Access Control Model, as defined in Chapter 3
- Either the file system card edge interface or the VM 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. The Access Control File and associated SEIWG string defined in Appendix C are mandatory for contact-type GSC cards, and the SEIWG container defined in Appendix G is mandatory for contactless GSC cards.

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 provides an interoperable set of smart card services to client applications is called the Service Provider Module (SPM). These services are logically divided into three modules that provide utility, secure data storage, and cryptographic services. Since an SPM generally will be implemented through a combination of hardware and software, the software component of the SPM is referred to as the Service Provider Software (SPS).
- **Card Command Level:** This level is concerned with the exact APDUs (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 SPM.

The SPM is a combination of both these levels and it includes:

- SPS, implementing both BSI and VCEI interfaces
- Smart card reader driver
- Smart card reader
- GSC-IS conformant smart card

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 defines Data Models (DM). These Data Models provide data portability across GSC-IS conformant 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 Capability Container (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 and are further described in Sections 2.2 - 2.8. All objects below the client application layer are components of the SPM.

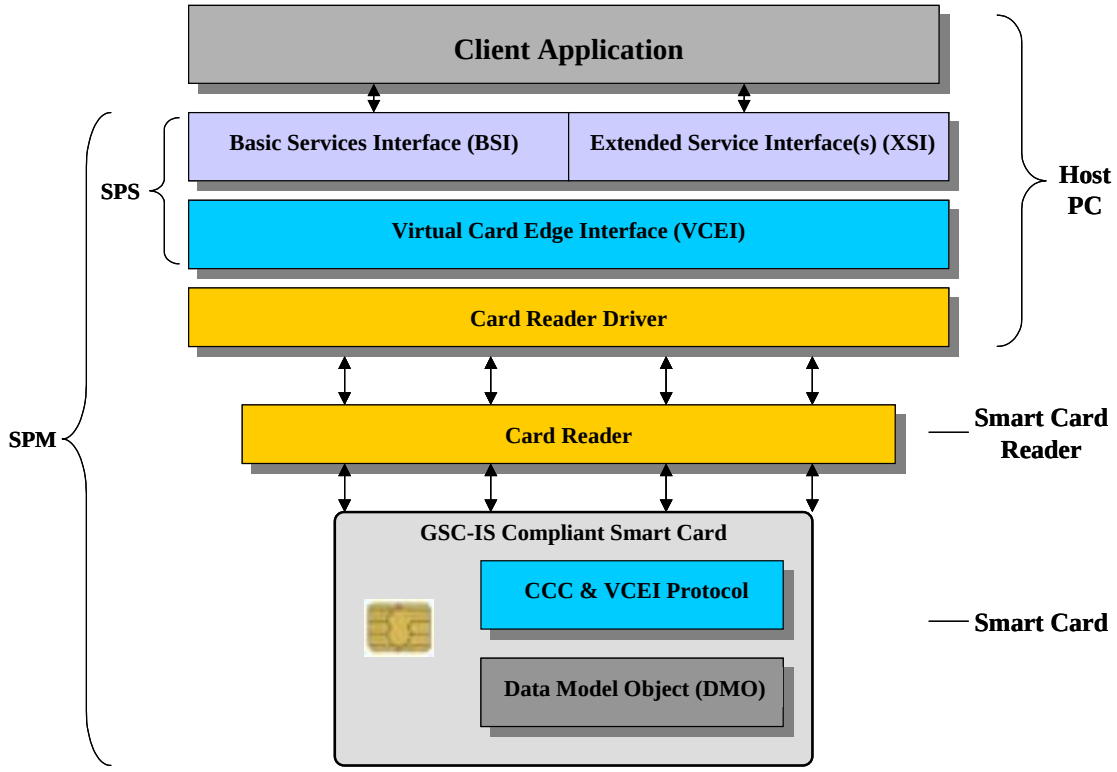


Figure 2-1: The GSC-IS Architectural Model

2.2 Basic Services Interface Overview

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.
- **Generic 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 [ISO4].
- **Cryptographic Provider Module:** Provides fundamental cryptographic services such as random number generation, authentication, and digital signature generation.

The capabilities of a given SPM depend on the smart card available to the SPM 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 SPM. 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 Overview

Because the BSI is not a complete operational interface, real world SPM implementations may 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 SPM implementations may support additional functionality outside the BSI domain. An SPM may therefore include an Extended Service Interface (XSI) that provides non-interoperable, but operationally required, functions. Since XSIs are implementation and application 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 Overview

ISO 7816-4 [ISO4] defines a hierarchical file system structure for smart cards. Smart cards that conform to ISO 7816-4 [ISO4] 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™ [JAVA] 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.

A virtual machine card is one that can be extended by loading executable programs after the card has been manufactured. 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 [ISO4] and 7816-8 [ISO8] standard.

The GSC-IS VCEI defines default sets of interoperable APDU level commands for both virtual machine and file system smart cards. The SPS of an SPM 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.

2.5 Roles of the BSI and VCEI

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 conformant SPS and any GSC conformant smart card. Similarly, standardization at the BSI layer establishes interoperability between any GSC conformant application and any GSC conformant 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-IS Data Model Overview

Each GSC-IS conformant smart card shall conform to a GSC-IS Data Model. GSC-IS Data Models define the set of containers and data elements within each container for cards supporting that Data Model. The GSC-IS defines two Data Models: the GSC Data Model (Appendix C) (formerly referred to as the J.8 Data Model in GSC-IS v1.0) and the U.S. Department of Defense Common Access Card Data Model (Appendix D). The following containers are mandatory in either Data Model:

- CCC for contact and contactless cards and
- Access control file with SEIWG [SEIW] string for contact cards or
- SEIWG container and SEIWG [SEIW] string for contactless cards.

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 are accessed through the Generic Container Provider Module of the BSI. Access to the containers are subject to the Access Control Rules (ACR) defined in Chapter 3.

This document uses the terms “file,” “container,” and “object” synonymously.

2.7 Card Capabilities Container Overview

Each GSC-IS conformant card shall carry a Card Capabilities Container. The CCC is one of the mandatory containers that must be present in all GSC-IS 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 are represented in a CCC.

The CCC allows each GSC-IS conformant 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.

2.8 Service Provider Software Overview

The SPS component of an SPM 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.

2.9 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 including:

- 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, conformant 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 SPM are subject to a set of Access Control Rules (ACR). ACRs are defined for each card service and default container when a GSC-IS conformant 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:

- **Always:** The corresponding service can be provided without restrictions.
- **Never:** The corresponding service can never be provided.
- **External Authenticate:** The corresponding service can be provided only after a `GET CHALLENGE` and subsequent `EXTERNAL AUTHENTICATE` APDUs.
- **PIN Protected:** The corresponding service is provided if and only if the verification code of the PIN associated with the service has been provided in the current card session.
- **PIN Always:** The corresponding service can be provided only if its associated PIN code has been verified immediately before each unique service request.
- **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.
- **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.
- **External Authenticate and PIN:** The two methods must be chained successfully before access to the service is granted. Order of the methods is not important.
- **PIN then External Authenticate:** The PIN presentation is followed by an External Authentication.
- **Secure Channel (GP):** The corresponding service can be provided only through a Secure Channel managed by a Global Platform [GLOB] Secure Messaging layer.
- **Update Once:** A target object can only be updated once during its lifetime.

- **Secure Channel (ISO):** The corresponding service can be provided through a Secure Channel managed by an ISO [ISO4],[ISO8] Secure Messaging layer.

BSI-level ACRs are a logical combination of primitive access methods. The BSI-level access methods and associated hexadecimal values are summarized in the Table 3-1. Hexadecimal values are assigned to the `unAccessMethodType` member of the `BSIAuthenticator` structure defined in Section 4.5.3.

Table 3-1: BSI Access Method Types

Access Method Type	Value	Meaning
`BSI_AM_XAUTH`	0x02	External Authentication.
`BSI_AM_SECURE_CHANNEL_GP`	0x04	Secure Channel (Global Platform)
`BSI_AM_PIN`	0x06	PIN code is required
`BSI_AM_SECURE_CHANNEL_ISO`	0x0B	Secure Channel (ISO 7816-4)

The BSI-level ACRs and associated hexadecimal values are summarized in Table 3-2. Hexadecimal values are returned in the `ACRType` member of the `BSIAcr` structure defined in Section 4.6.3. The `BSIAcr` structure is present in the members of the `GCacr` structure defined in Section 4.6.3 and the `CRYPTOacr` strucuture defined in Section 4.7.5.

Table 3-2: BSI Access Control Rule Types

Access Control Rule Type (ACRType)	Access Method List	Logical Relation between AMs	Value	Meaning
`BSI_ACR_ALWAYS`	–	–	0x00	No access control rule is required
`BSI_ACR_NEVER`	–	–	0x01	Operation is never possible
`BSI_ACR_XAUTH`	`BSI_AM_XAUTH`	–	0x02	External Authentication
`BSI_ACR_XAUTH_OR_PIN`	`BSI_AM_XAUTH, BSI_AM_PIN`	OR	0x03	The object method can be accessed either after an External Authentication or after a successful PIN presentation
`BSI_SECURE_CHANNEL_GP`	`BSI_AM_SECURE_CHANNEL_GP`	–	0x04	Secure Channel (Global Platform)
`BSI_ACR_PIN_ALWAYS`	`BSI_AM_PIN`	–	0x05	PIN must be verified immediately prior to service request
`BSI_ACR_PIN`	`BSI_AM_PIN`	–	0x06	PIN code is required
`BSI_ACR_XAUTH_THEN_PIN`	`BSI_AM_XAUTH, BSI_AM_PIN`	AND	0x07	External Authentication followed by a PIN presentation
`BSI_ACR_UPDATE_ONCE`	–	–	0x08	The target object can only be updated once during its lifetime

Access Control Rule Type (ACRType)	Access Method List	Logical Relation between AMs	Value	Meaning
BSI_ACR_PIN_THEN_XAUTH	BSI_AM_PIN, BSI_AM_XAUTH	AND	0x09	PIN presentation followed by External Authentication
Reserved for future use	–	–	0x0A	RFU
BSI_SECURE_CHANNEL_ISO	BSI_AM_SECURE_CHANNEL_ISO	–	0x0B	Secure Channel (ISO 7816-4)
BSI_ACR_XAUTH_AND_PIN	BSI_AM_XAUTH, BSI_AM_PIN	AND	0x0C	PIN presentation AND External Authentication in any order are required.
Reserved for future use	–	–	0x0D-0xFF	RFU

The External Authentication method shall conform with ISO 7816-4 [ISO4] and 7816-8 [ISO8]. 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 4.5.13) to allow applications to create a secure channel and operate inside this channel.

3.2 Determining Containers

Applications can retrieve the ACR that must be fulfilled to access a specific service or container. ACR retrieval processes 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()`.
- **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 Tables 3-3 and 3-4.

Table 3-3: 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 BSI_ACR_UPDATE_ONCE
gscBsiGcGetContainerProperties()	BSI_ACR_ALWAYS

Table 3-4: 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_PIN_ALWAYS BSI_ACR_XAUTH
gscBsiPkiGetCertificate()	BSI_ACR_ALWAYS BSI_ACR_PIN BSI_ACR_XAUTH
gscBsiGetCryptoProperties()	BSI_ACR_ALWAYS

Note: When using the `gscBsiPkiCompute()` function for signature operation, it is highly recommended that the implementation require `BSI_ACR_PIN_ALWAYS` for access control.

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 user executing the client application, the client application itself, and the smart

card. The GSC-IS ACRs are based on three general authentication mechanisms: PIN Verification, External Authentication, and Secure Messaging.

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

It is important to note that at the smart card level, the privileges are granted sequentially. 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()`.

Sections 3.3.1 through 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.

3.3.1 PIN Verification

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

- Establish a logical connection with the card through a call to the BSI's function `gscBsiUtilConnect()`.
- 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 (e.g., `gscBsiGcReadValue()` or `gscBsiPkiCompute()`), the ACR returned in the GCacr or `CRYPTOacr` structure for this service must be `BSI_ACR_PIN`.
- Call `gscBsiUtilAcquireContext()` with the `BSIAuthenticator` structures 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 `authValue` field and `accessMethodType` set to `BSI_ACR_PIN`.
- Access the desired smart card service through subsequent BSI calls.
- Call `gscBsiUtilReleaseContext()` to release the security context.

3.3.2 External Authentication

A typical BSI sequence of calls for an External Authentication:

- Establish a logical connection with the card through a call to `gscBsiUtilConnect()`.
- Retrieve the ACRs for a desired card service provider through a call to either `gscBsiGcGetContainerProperties()` 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 (e.g., `gscBsiGcReadValue()` or `gscBsiPkiCompute()`), the ACR returned in the `GCacr` or `CRYPTOacr` structure for this service must be `BSI_ACR_XAUTH.`

- 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 cryptogram by encrypting the random challenge using a symmetric External Authentication key. The client application may need to examine the `keyIDOrReference` member of the appropriate ACR returned in `GCacr` or `CRYPTOacr` to determine which External Authentication key it should use to encrypt the random challenge.
- The client application calls the BSI's `gscBsiUtilAcquireContext()` function passing the cryptogram computed in the previous step.
- The smart card decrypts the Authenticator using its External Authentication key, and verifies that the resulting plaintext value matches the original random challenge value.
- Access the desired smart card service through subsequent BSI calls.
- 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 that 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]

.

4. Basic Services Interface

4.1 Overview

An SPM must provide a BSI. Client applications communicate with the SPM through this interface. The SPS component of the SPM is directly responsible for implementing the BSI.

This chapter defines the BSI services, using notation similar to Interface Definition Language (IDL) which is referred to as pseudo IDL throughout this document. The set of services consists of 23 functions grouped into three functional modules as follows:

A Smart Card Utility Provider Module:

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

A Smart Card Generic Container Provider Module:

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

A Smart Card Cryptographic Provider Module:

- `gscBsiGetChallenge()`

- gscBsiSkiInternalAuthenticate()
- gscBsiPkiCompute()
- gscBsiPkiGetCertificate()
- gscBsiGetCryptoProperties()

All SPM implementations must provide the full set of 23 functions as specified in this chapter. Based on the capabilities available, a given function call may return a BSI_NO_CARDSERVICE or BSI_NO SPSSERVICE error message in case the SPM 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()
- gscBsiGcGetContainerProperties()
- 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 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 Chapter 3, Table 3-2, Table 3-3, and Table 3-4. Section 4.4 defines BSI return codes and Section 4.5 defines 23 functions of the BSI, using pseudo IDL.

4.2 Binary Data Encoding

BSI functions accept or return binary data, such as cryptograms. However, some of the BSI services may pass or get some ASCII or ASCII hexadecimal formatted data depending on the usage. In this case, each of the services involved must explicitly mention this and which of its parameter(s) is/are impacted.

4.3 Mandatory Cryptographic Algorithms

The following cryptographic algorithms and associated algorithm identifiers are mandatory for all GSC smart cards. These algorithm ID values are used as parameters at the BSI level.

- Algorithm Identifier “0x81”: `DES3-ECB`, with a double length key-size, 16 bytes.
- Algorithm Identifier “0xA3”: `RSA_NO_PAD`, the private key computation, 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 4.5.3 to 4.7.5), return 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 smart card associated with the specified card handle is not present.
`BSI_CARD_REMOVED`	0x0A	The smart card associated with the specified card handle has been removed.
`BSI_NO_SPSSERVICE`	0x0B	The SPS does not provide the requested service.
`BSI_IO_ERROR`	0x0C	Error encountered during input/output of the specified data.
–	0x0D	RFU
`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_BLOCKED`	0x11	The PIN is blocked.
–	0x012	RFU
`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 smart card before the timeout value expired.

Label	Return Code Hexadecimal Value	Meaning
BSI_TERMINAL_AUTH	0x15	The card reader has performed a successful authentication exchange with the smart 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_SC_LOCKED	0x19	The smart card associated with the specified card handle is under the exclusive transaction of another client application (see blocking mode in Section 4.5.6)
BSI_NOT_TRANSACTED	0x20	The current transaction has not ended.

4.5 Smart Card Utility Provider Module Interface Definition

Section 4.5.1 presents the pseudo IDL used to define the 23 functions of the BSI services.

4.5.1 Pseudo IDL Definition

Using a modified Backus-Naur notation, a definition for the pseudo IDL is presented as follows:

```
BSI_IDL_Definition: (BSI_Function_Unit, …)
BSI_Function_Unit:(
               Function_Prototype:
               (
                    [Return_Type],  // See below for possible values
                    Function_Name,
                    [Parameters*: (
                        Way: {“IN” | “OUT” | “INOUT”},
                        Parameter_Type, // See below for possible values
                        Parameter_Name
                                  )
                    ]
               )

(Return_Type | Paramater_Type) : Type

Type:   “unsigned long”
      | “string”
      | “boolean”
      | “short”
      | “sequence” +<Type> // represent a sequence of element of type “Type”
      | “GCacr”            // structure
      | “GCContainerSize”  // structure
      | “CRYPTOacr”        // structure
      | “BSIAuthenticator” // structure
      | “BSIAcr”           // structure
```

The types `GCacr`, `GCContainerSize`, `CRYPTOacr` and `BSIAuthenticator` are structure. The definition of a structure is as follows:

```
Struct_Definition: (Struct_Definition, …)
```

```
Struct_Definition: (
    “struct” structure_Name “{“
                Struct_Parameters*:
                 (
                   Parameter_Type,    // See above for possible values
                   Parameter_Name
                  )
                            “}”
                     )
```

4.5.2 Rules

A description of the symbols used is in Table 4-2.

Table 4-2: Description of Symbols

Symbol	Meaning
:	is composed of
[ ]	optional element
( )	includes or included in
,	separates elements
…	element repeats unspecified number of times
{ }	choose one from list
|	or, indicates choice of possibilities for element value
+	element is combined with preceding element
//	remainder of line contains comments
“”	contains a value
*	number of elements is zero or several

Tables 4-3 and 4-4 are the pseudo IDL to Java and pseudo IDL to C mappings for the different types specified above.

Table 4-3: Mapping Pseudo IDL to Java

IDL type	Java type
unsigned long	int
String	byte[] or Java.lang.String (depending on the format : binary, ASCII or ASCII hex.)
Boolean	boolean
octet (unsigned 8 bits type)	short
sequence + <Type>	<Type>[] or Vector of Type
Gcacr	class Gcacr
GCContainerSize	class GCContainerSize
CRYPTOacr	class CRYPTOacr
BSIAcr	Class BSIAcr

Table 4-4: Mapping Pseudo IDL to C

IDL type	C type
unsigned long	unsigned long
String	unsigned char *
Boolean	boolean
octet (unsigned 8 bits type)	unsigned char
sequence + <Type>	<Type>[] (for byte see below)
sequence<byte>	unsigned char *
Gcacr	struct Gcacr
Gctag	unsigned char
GCContainerSize	struct GCContainerSize
CRYPTOacr	struct CRYPTOacr
BSIAcr	struct BSIAcr
String (with n characters max, null terminated)	char[n]

4.5.3 gscBsiUtilAcquireContext()

Purpose: This function shall establish a session with a target container on the smart card by submitting the appropriate Authenticator in the `BSIAuthenticator` structure. For ACRs requiring external authentication (`XAUTH`), the `authValue` field of the `BSIAuthenticator` structure must contain a cryptogram calculated by encrypting a random challenge from **`gscBsiGetChallenge()`**. In cases where the card acceptance device authenticates the smart card, this function returns a `BSI_TERMINAL_AUTH` return code and the cryptogram is ignored.

For ACRs that require chained authentication such as `BSI_ACR_PIN_AND_XAUTH`, the calling application passes in the required authenticators in multiple `BSIAuthenticator` structures. In this example the calling application passes a PIN and the appropriate External Authentication cryptogram in two `BSIAuthenticator` structures. The client application must set the `accessMethodType` field of each `BSIAuthenticator` structure to match the type of authenticator contained in the structure. To satisfy an ACR of `BSI_ACR_PIN_AND_XAUTH`, the application would construct a sequence of two `BSIAuthenticators`: one containing a PIN and one containing an External Authentication cryptogram. The `BSIAuthenticator` structure containing the PIN would have an `accessMethodType` of `BSI_AM_PIN`, and the `BSIAuthenticator` structure containing the External Authentication cryptogram would have an `accessMethodType` of `BSI_AM_XAUTH.`

Prototype:

```
unsigned long gscBsiUtilAcquireContext(
   IN unsigned long              hCard,
   IN string                     AID,
   IN sequence<BSIAuthenticator> strctAuthenticator,
   IN unsigned long              authNb
);
```

Parameters:

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

`AID`: Target container AID value. The parameter shall be in ASCII hexadecimal format.

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

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

The `BSIAuthenticator` structure is defined as follows:

```
struct BSIAuthenticator {
   unsigned long        accessMethodType;
   unsigned long        keyIDOrReference;
```

```
    sequence<byte>        authValue;
};
```

Variables associated with the `BSIAuthenticator` structure:

`accessMethodType`: Access Method Type (see Table 3-1 in Section 3.1). This function does not support secure channel and will return a `BSI_BAD_PARAM` if this field is set to one of the secure channel authentication methods.

`keyIDOrReference`: Key identifier or reference of the authenticator. This is used to distinguish between multiple authenticators with the same Access Method Type.

`authValue`: Authenticator, can be an external authentication cryptogram or PIN. If the authenticator value is NULL, then the SPS is in charge of gathering authentication information and authenticating to the card.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_ACR_NOT_AVAILABLE
BSI_BAD_AUTH
BSI_CARD_REMOVED
BSI_PIN_BLOCKED
BSI_UNKNOWN_ERROR
BSI_TERMINAL_AUTH
BSI_BAD_PARAM
BSI_SC_LOCKED
```

4.5.4 gscBsiUtilConnect()

Purpose: Establish a logical connection with the smart 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            readerName,
   OUT unsigned long    hCard
);
```

Parameters:

hCard: Card connection handle.

readerName: Name of the reader that the smart card is inserted into. If this field is a NULL pointer, the SPS shall attempt to connect to the smart card in the first available reader, as returned by a call to the BSI's function **gscBsiUtilGetReaderList().** The reader name string shall be stored as ASCII encoded String. (See Section 4.2)

Return Codes:

BSI_OK
BSI_BAD_PARAM
BSI_UNKNOWN_READER
BSI_CARD_ABSENT
BSI_TIMEOUT_ERROR
BSI_UNKNOWN_ERROR

4.5.5 gscBsiUtilDisconnect()

Purpose: Terminate a logical connection to a smart card.

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

Parameters:

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

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

4.5.6 gscBsiUtilBeginTransaction()

Purpose: This function starts an exclusive transaction with the smart card referenced by `hCard`. When the transaction starts, all other applications are precluded from accessing the smart card while the transaction is in progress. Two types of calls can be made: a blocking transaction call and a non-blocking transaction call, with a boolean type parameter identifying which mode is called. In the non-blocking mode, the call will return immediately if another client has an active transaction lock. The returned error code will be `BSI_SC_LOCKED`. In the blocking mode, the call will wait indefinitely for any active transaction locks to be released. A transaction must be completed by a call to **`gscBsiUtilEndTransaction().`**

For single-threaded BSI implementations, it can be assumed that each application will be associated with a separate process. The same process that starts a transaction must also complete the transaction. For multi-threaded BSI implementations, it can be assumed that each application will be associated with a separate thread and/or process. The same thread that starts a transaction must also complete the transaction.

If this function is called by a thread that has already called **`gscBsiUtilBeginTransaction()`** but has not yet called **`gscBsiUtilEndTransaction(),`** it will return the error `BSI_NOT_TRANSACTED`.

If the SPS (Service Provider Software) does not support transaction locking, it should return the error code `BSI_NO_SPSSERVICE` in response to a call to **`gscBsiUtilBeginTransaction()`**.

Prototype:

```
unsigned long gscBsiUtilBeginTransaction(
   IN unsigned long     hCard
   IN boolean           blType
);
```

Parameters:

`hCard:`	Card communication handle returned from **`gscBsiUtilConnect()`**
`blType:`	Boolean specifying the type of transaction call (`blType` set to “true” in blocking mode. `blType` set to “false” in non-blocking mode).

`Return Codes:`

```
BSI_OK
BSI_BAD_HANDLE
BSI_UNKNOWN_ERROR
BSI_SC_LOCKED
BSI_NOT_TRANSACTED
BSI_NO_SPSSERVICE
```

4.5.7 gscBsiUtilEndTransaction()

Purpose: This function ends a previously started transaction, allowing other blocked applications to begin or resume interactions with the card.

If this function is called by a thread that has not yet called **`gscBsiUtilBeginTransaction(),`** it will return the error `BSI_NOT_TRANSACTED`.

If the SPS does not support transaction locking, it should return the error code `BSI_NO_SPSSERVICE` in response to a call to **`gscBsiUtilEndTransaction()`**.

Prototype:

```
unsigned long gscBsiUtilEndTransaction(
   IN unsigned long     hCard
);
```

Parameters:

`hCard:`	Card communication handle returned from **`gscBsiUtilConnect().`**

`Return Codes:`

```
BSI_OK
BSI_BAD_HANDLE
BSI_UNKNOWN_ERROR
BSI_NOT_TRANSACTED
BSI_NO_SPSSERVICE
```

4.5.8 gscBsiUtilGetVersion()

Purpose: Returns the BSI implementation version.

Prototype:

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

Parameters:

version:	The BSI and SPS version formatted as "major,minor,revision,build_number". The value for an SPS conformant with this version of the GSC-IS is "2,1,0,<build number>". The build number field is vendor/implementation dependent. The version name string shall be stored as ASCII encoded String. (See Section 4.2)

Return Codes:

```
BSI_OK
BSI_INSUFFICIENT_BUFFER
BSI_UNKNOWN_ERROR
```

4.5.9 gscBsiUtilGetCardProperties()

Purpose: Retrieves Card Capability Container ID and capability information for the smart card.

Prototype:

```
unsigned long gscBsiUtilGetCardProperties(
   IN unsigned long     hCard,
   INOUT sequence<byte> CCCUniqueID,
   OUT unsigned long    cardCapability
);
```

Parameters:

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

CCCUniqueID: Buffer for the Card Capability Container ID.

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

```
#define BSI_GCCDM      0x00000001
#define BSI_SKI        0x00000002
#define BSI_PKI        0x00000004
```

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_CARD_REMOVED
BSI_INSUFFICIENT_BUFFER
BSI_NO_CARDSERVICE
BSI_UNKNOWN_ERROR
BSI_SC_LOCKED
```

4.5.10 gscBsiUtilGetCardStatus()

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

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

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

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

4.5.11 gscBsiUtilGetExtendedErrorText()

Purpose: When a BSI function call returns an error, an application can make a subsequent call **`gscBsiUtilGetExtendedErrorText`** 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 unsigned long     hCard,
   OUT string           errorText
);
```

Parameters:

`hCard`:	Card connection handle from **`gscBsiUtilConnect()`**.
`errorText`:	A fixed length buffer containing an implementation specific error text string. The text string has a maximum length of 255 characters. 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`. The error text string shall be stored as ASCII encoded String. (See Section 4.2)

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_NO_TEXT_AVAILABLE
BSI_UNKNOWN_ERROR
```

4.5.12 gscBsiUtilGetReaderList()

Purpose: Retrieves the list of available readers.

Prototype:

```
unsigned long gscBsiUtilGetReaderList(
   INOUT sequence<string>  readerList
);
```

Parameters:

readerList:	Reader list buffer. The reader list is returned as a multi-string. The list of available readers shall be stored as ASCII encoded String. (See Section 4.2)

Return Codes:

```
BSI_OK
BSI_INSUFFICIENT_BUFFER
BSI_UNKNOWN_ERROR
```

4.5.13 gscBsiUtilPassthru()

Purpose: Allows a client application to send a "raw" ISO 7816-4 [ISO4] APDU through the BSI directly to the smart card and receive the APDU-level response.

Prototype:

```
unsigned long gscBsiUtilPassthru(
   IN unsigned long     hCard,
   IN sequence<byte>    cardCommand,
   INOUT sequence<byte> cardResponse
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
cardCommand:	The APDU to be sent to the smart card. That parameter must be in ASCII hexadecimal format.
cardResponse:	Pre-allocated buffer for the APDU response from the smart card. The response must include the status bytes SW1 and SW2 returned by the smart card. If the size of the buffer is insufficient, the SPS shall return truncated response data and the return code BSI_INSUFFICIENT_BUFFER. That parameter must be in ASCII hexadecimal format.

Return Codes:

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

4.5.14 gscBsiUtilReleaseContext()

Purpose: Terminate a session with the target container on the smart card.

Prototype:
```
unsigned long gscBsiUtilReleaseContext(
   IN unsigned long     hCard,
   IN sequence<byte>    AID
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
AID:	Target container AID value. The AID shall be stored as an ASCII hexadecimal string.

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

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 unsigned long     hCard,
   IN string            AID,
   IN octet             tag,
   IN sequence<byte>    value
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
AID:	Target container AID value. The parameter shall be in ASCII hexadecimal format.
tag:	Tag of data item to store.
value:	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_IO_ERROR
BSI_UNKNOWN_ERROR
BSI_SC_LOCKED
```

4.6.2 gscBsiGcDataDelete()

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

Prototype:

```
unsigned long gscBsiGcDataDelete(
   IN unsigned long     hCard,
   IN string            AID,
   IN octet             tag
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
AID:	Target container AID value. The parameter shall be in ASCII hexadecimal format.
tag:	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_SC_LOCKED
BSI_ACCESS_DENIED
BSI_IO_ERROR
BSI_UNKNOWN_ERROR
```

4.6.3 gscBsiGcGetContainerProperties()

Purpose: Retrieves the properties of the specified container.

Prototype:

```
unsigned long gscBsiGcGetContainerProperties(
   IN unsigned long     hCard,
   IN string            AID,
   OUT GCacr            strctGCacr,
   OUT GCContainerSize  strctContainerSizes,
   OUT string           containerVersion
);
```

Parameters:

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

AID: Target container AID value. The parameter shall be in ASCII hexadecimal format.

strctGCacr: Structure indicating access control conditions for all operations. The range of possible values for the members of this structure is defined in Table 3-2 (Section 3.1). The allowable ACRs for each function are listed in Table 3-3 (Section 3.2). `keyIDOrReference` contains the key identifier or reference for each access method contained in the ACR in order of appearance. `authNb` is the number of access methods logically combined in the ACR. `ACRID` is RFU and must be NULL (0x00).

```
struct GCacr {
   BSIAcr               createACR;
   BSIAcr               deleteACR;
   BSIAcr               readTagListACR;
   BSIAcr               readValueACR;
   BSIAcr               updateValueACR;
};

struct BSIAcr {
   unsigned long        ACRType;
   unsigned long        keyIDOrReference[MaxNbAM];
   unsigned long        AuthNb;
   unsigned long        ACRID;
};
```

strctContainerSizes: For Virtual Machine cards, the size (in bytes) of the container specified by **AID.** **maxNbDataItems** is the size of the T-Buffer, and **maxValueStorageSize** 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        maxNbDataItems;
   unsigned long        maxValueStorageSize;
```

```
}
```

containerVersion: Version of the container. The format of this value is application dependent. In cases where the smart card cannot return a container version, this byte sequence will be empty.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_CARD_REMOVED
BSI_SC_LOCKED
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 unsigned long        hCard,
   IN string               AID,
   INOUT sequence<octet>   tagArray
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
AID:	Target container AID value. The parameter shall be in ASCII hexadecimal format.
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_SC_LOCKED
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 unsigned long     hCard,
   IN string            AID,
   IN octet             tag,
   INOUT sequence<byte> value
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
AID:	Target container AID value. The parameter shall be in ASCII hexadecimal format.
tag:	Tag value of data item to read.
value:	Value associated with the specified tag. The client application must allocate the buffer.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_TAG
BSI_CARD_REMOVED
BSI_SC_LOCKED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_INSUFFICIENT_BUFFER
BSI_IO_ERROR
BSI_UNKNOWN_ERROR
```

4.6.6 gscBsiGcUpdateValue()

Purpose: Updates the Value associated with the specified Tag.

Prototype:

```
unsigned long gscBsiGcUpdateValue(
   IN unsigned long     hCard,
   IN string            AID,
   IN octet             tag,
   IN sequence<byte>    value
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
AID:	Target container AID value. The parameter shall be in ASCII hexadecimal format.
tag:	Tag of data item to update.
value:	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_SC_LOCKED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_NO_MORE_SPACE
BSI_IO_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 smart card as the first step of a challenge-response authentication protocol between the client application and the smart card. The client subsequently encrypts the challenge using a symmetric key and returns the encrypted random challenge to the smart card through a call to **`gscBsiUtilAcquireContext()`** in the `authValue` field of a `BSIAuthenticator` structure.

Prototype:

```
unsigned long gscBsiGetChallenge(
   IN unsigned long     hCard,
   IN string            AID,
   INOUT sequence<byte> challenge
);
```

Parameters:

`hCard:`	Card connection handle from **`gscBsiUtilConnect()`**.
`AID:`	Target container AID value. The parameter shall be in ASCII hexadecimal format.
`challenge:`	Random challenge returned from the smart card.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_CARD_REMOVED
BSI_SC_LOCKED
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 smart 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 smart card. `BSI_ACCESS_DENIED` is returned if the card reader fails to authenticate the smart card.

Prototype:

```
unsigned long gscBsiSkiInternalAuthenticate(
   IN unsigned long     hCard,
   IN string            AID,
   IN octet             algoID,
   IN sequence<byte>    challenge,
   INOUT sequence<byte> cryptogram
);
```

Parameters:

`hCard:`	Card connection handle from **`gscBsiUtilConnect()`**.
`AID:`	SKI provider module AID value. The parameter shall be in ASCII hexadecimal format.
`algoID:`	Identifies the cryptographic algorithm that the smart 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.
`challenge:`	Challenge generated by the client application and submitted to the smart card.
`cryptogram:`	The cryptogram computed by the smart card.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_BAD_ALGO_ID
BSI_CARD_REMOVED
BSI_SC_LOCKED
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 unsigned long     hCard,
   IN string            AID,
   IN octet             algoID,
   IN sequence<byte>    message,
   INOUT sequence<byte> result
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
AID:	PKI provider module AID value. The parameter shall be in ASCII hexadecimal format.
algoID:	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.
message:	The message digest to be signed.
result:	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_SC_LOCKED
BSI_ACCESS_DENIED
BSI_NO_CARDSERVICE
BSI_INSUFFICIENT_BUFFER
BSI_UNKNOWN_ERROR
```

4.7.4 gscBsiPkiGetCertificate()

Purpose: Reads the certificate from the smart card.

Prototype:

```
unsigned long gscBsiPkiGetCertificate(
   IN unsigned long     hCard,
   IN string            AID,
   INOUT sequence<byte> Certificate
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
AID:	PKI provider module AID value. The parameter shall be in ASCII hexadecimal format.
certificate:	Buffer containing the certificate.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_CARD_REMOVED
BSI_SC_LOCKED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_IO_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 unsigned long     hCard,
   IN string            AID,
   OUT CRYPTOacr        strctCRYPTOacr,
   OUT unsigned long    keyLen
);
```

Parameters:

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

`AID:` AID of the PKI provider. The parameter shall be in ASCII hexadecimal format.

`strctCRYPTOacr:` Structure indicating access control conditions for all operations. The `BSIAcr` structure is defined in Section 4.6.3. The range of possible values for the members of this structure are defined in Table 3-2 (Section 3.1), and the allowable ACRs for each function in Table 3-4 (Section 3.2). `keyIDOrReference` contains the key identifier or reference for each access method contained in the ACR in order of appearance. `authNb` is the number of access methods logically combined in the ACR. ACRID is RFU and must be NULL (0x00) in this version. Note that the `readValueACR` member maps to the `gscBsiPkiGetCertificate()` function.

```
struct CRYPTOacr {
   BSIAcr               getChallengeACR;
   BSIAcr               internalAuthenticateACR;
   BSIAcr               pkiComputeACR;
   BSIAcr               createACR;
   BSIAcr               deleteACR;
   BSIAcr               readTagListACR;
   BSIAcr               readValueACR;
   BSIAcr               updateValueACR;
};
```

`keyLen:` Length in bits of the private key managed by the PKI provider.

Return Codes:

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

THIS PAGE INTENTIONALLY LEFT BLANK.

5. Virtual Card Edge Interface

The Virtual Card Edge Interface includes two sets of APDU commands: (1) an ISO 7816-4 [ISO4] and 7816-8 [ISO8] conformant GSC-IS APDU set for use in conformant file system cards, and (2) a set of VM APDUs for use in VM cards. The card edge also consists of the CCC, which is a file located on each conformant smart card, and the GSC-IS APDU mapping mechanism.

The GSC-IS ISO-conformant APDU set can be implemented directly by conformant cards (such as in a conformant file system card or as a VM card applet). It is expected that some file system smart cards may use native APDU instruction sets that will differ from the GSC-IS APDU set. In those cases, an SPS must modify the ADPU set such that it conforms to the smart card's native APDU set. This is done using the GSC-IS APDU mapping mechanism described in Section 5.2 and in Chapter 6.

Sections 5.1 through 5.3 describe the GSC-IS APDU set, overview information on the procedures for mapping this APDU set to smart card-specific APDU sets, and the APDUs for VM cards only. Chapter 6 provides details on the rules and procedures for APDU translations according to the CCC grammar.

5.1 GSC-IS ISO Conformant APDUs

Table 5-1 shows the GSC-IS APDU set for file system and VM cards. The APDUs are conformant with ISO 7816-4 [ISO4] and 7816-8 [ISO8], however some values have been defined for cryptogram lengths and cryptographic algorithm identifiers. Additional behavior for the APDUs would be described in a smart card's CCC tuples using the descriptor code mechanisms. Support for secure messaging is not provided in this APDU set; as described in Section 3.3.3, secure messaging is implemented via the **`gscBsiUtilPassthru()`** mechanism in accordance with the Global Platform [GLOB] or ISO 7816-4 [ISO4].

Table 5-1: GSC-IS APDU Set

GSC-IS APDU Set	
Generic File Access APDUs	**GET RESPONSE**
	READ BINARY
	SELECT DF
	SELECT EF UNDER SELECTED DF
	SELECT FILE
	SELECT MASTER FILE (Root)
	UPDATE BINARY
Access Control APDUs	**EXTERNAL AUTHENTICATE**
	GET CHALLENGE
	INTERNAL AUTHENTICATE
	VERIFY

GSC-IS APDU Set	
Public Key Operations APDUs	**MANAGE SECURITY ENVIRONMENT**
	PERFORM SECURITY OPERATION

The APDUs are divided into three categories: Generic File Access, Access Control, and Public Key Operations. The ADPU commands and responses are structured as follows:

Table 5-2: APDU Command and Response Structure

Command APDU						
CLA	INS	P1	P2	L_c	Data Field	L_e

Response APDU		
Response	SW1	SW2

The terms described in Table 5-3 are used throughout this section.

Table 5-3: APDU Command and Response Structure

APDU Term	Description
CLA	Class byte
Data Field	String of bytes sent in the data field of the command
FC	Function code, used in the CCC grammar to identify the default APDU that is being mapped (see Chapter 6 for detailed information)
L_c	Number of bytes present in data field of the command
L_e	Maximum number of bytes expected in the data field of the response to the command
INS	Instruction byte; ISO 7816 defines a set of common commands, e.g., 'B0' is Read Binary
P1-P2	Instruction parameter 1 and 2
Response	String of bytes received in the data field of the response
SW1	Command processing status, i.e., the return code from the smart card
SW2	Command processing qualifier, supplies further information on SW1

5.1.1 Generic File Access APDUs

The APDUs in Table 5-4 are used to perform basic file access functions.

Table 5-4: Generic File Access APDUs

Generic File Access APDUs								
FC	**Card Function**	**CLA**	**INS**	**P1**	**P2**	**L_c**	**Data**	**L_e**
0x07	**GET RESPONSE**	0x00	0xC0	0x00	0x00	–	–	L_e
0x02	**READ BINARY**	0x00	0xB0	Off/H	Off/L	–	–	L_e
0x01	**SELECT DF**	0x00	0xA4	0x01	0x00 or 0x0C	0x02	File ID (2 bytes)	–
0x0D	**SELECT EF FILE UNDER SELECTED DF**	0x00	0xA4	0x02	0x00 or 0x0C	0x02	File ID (2 bytes)	–
0x0C	**SELECT FILE**	0x00	0xA4	0x00-0x03	0x00 or 0x0C	0x02	File ID (2 bytes)	–
0x0E	**SELECT MASTER FILE (Root)**	0x00	0xA4	0x03	0x00 or 0x0C	0x02	File ID (2 bytes)	–
0x03	**UPDATE BINARY**	0x00	0xD6	Off/H	Off/L	L_c	Data to Update	–

5.1.1.1 Get Response APDU

This APDU is used to read smart card results available from the completion of the previously executed APDU.

Command Message

Function Code	0x07
CLA	0x00
INS	0xC0
P1	0x00
P2	0x00
L_c	Empty
Data Field	Empty
L_e	Number of bytes to read in response

Response Message

Data Field returned in the Response Message

If the immediately preceding APDU has indicated that additional data is available, the data field of an immediately following Get Response APDU will contain this data.

Processing State returned in the Response Message

SW1	SW2	Meaning
61	XX	Normal processing, XX still available to read with subsequent Get Response
62	81	Part of returned data may be corrupted
67	00	Wrong length (incorrect L_e field)
6A	86	Incorrect parameters P1-P2
6C	XX	Wrong length (wrong L_e field; XX indicates the exact length)
90	00	Successful execution

5.1.1.2 Read Binary APDU

This APDU is used to read the currently selected transparent file. All access control operations necessary for reading the file must be completed before using this APDU.

Command Message

Function Code	0x02
CLA	0x00
INS	0xB0
P1	High-order byte of 2-byte offset
P2	Low-order byte of 2-byte offset
L_c	Empty
Data Field	Empty
L_e	Number of bytes to read

Response Message

Data Field returned in the Response Message

L_e number of bytes followed by the two-byte processing state.

Processing State returned in the Response Message

SW1	SW2	Meaning
62	81	Part of returned data may be corrupted
62	82	End of file reached before reading L_e bytes
67	00	Wrong length (wrong L_e field)
69	81	Command incompatible with file structure
69	82	Security status not satisfied
69	86	Command not allowed (no current EF)
6A	81	Function not supported
6A	82	File not found
6B	00	Wrong parameters (offset outside the EF)
6C	XX	Wrong length (wrong L_e field; XX indicates the exact length)
90	00	Successful execution

5.1.1.3 SELECT DF APDU

Sets the currently selected dedicated file to a dedicated file contained in the currently selected dedicated file.

Command Message

Function Code	0x01
CLA	0x00
INS	0xA4
P1	0x01 - Select child DF of current DF
P2	0x00 for response required, 0x0C for no response required
L_c	0x02
Data Field	2-byte File Identifier
L_e	Number of bytes returned

Response Message

Data Field returned in the Response Message

If P2 is set to 0x00, data is returned as per ISO 7816-4 [ISO4]. If P2 is set to 0x0C, no data is returned.

Processing State returned in the Response Message

SW1	SW2	Meaning
62	83	Selected file deactivated
62	84	File control information not formatted according to ISO 7816-4.
6A	81	Function not supported
6A	82	File not found
6A	86	Incorrect parameters P1-P2
6A	87	L_c inconsistent with P1-P2
90	00	Successful execution

5.1.1.3.1 Select EF Under Selected DF APDU

This APDU selects an Elementary File under the currently selected DF.

Command Message

Function Code	0x0D
CLA	0x00
INS	0xA4
P1	0x02 - Select child EF of current DF
P2	0x00 for response required, 0x0C for no response required
L_c	0x02
Data Field	2-byte File Identifier
L_e	Number of bytes returned

Response Message

Data Field returned in the Response Message

If P2 is set to 0x00, data is returned as per ISO 7816-4 [ISO4]. If P2 is set to 0x0C, no data is returned.

Processing State returned in the Response Message

SW1	SW2	Meaning
62	83	Selected file deactivated
62	84	File control information not formatted according to ISO 7816-4, Section 5.1.5
6A	81	Function not supported
6A	82	File not found
6A	86	Incorrect parameters P1-P2
6A	87	L_c inconsistent with P1-P2
90	00	Successful execution

5.1.1.4 Select File APDU

This APDU works as described in ISO 7816-4 [ISO4] to select the master file, a DF, or an EF.

Command Message

Function Code	0x0C
CLA	0x00
INS	0xA4
P1	See below
P2	0x00 for response required, 0x0C for no response required
L_c	Number of bytes in File Identifier, i.e., 2
Data Field	File Identifier
L_e	Empty

P1:
- 0x00 Explicit selection with Data Field; Data field must contain a valid File Identifier
- 0x01 Select child DF of current DF; Data Field must contain a valid File Identifier
- 0x02 Select child EF of current DF; Data Field must contain a valid File Identifier
- 0x03 Select parent DF of current DF; empty Data Field

Response Message

Data Field returned in the Response Message

If P2 is set to 0x00, data is returned as per ISO 7816-4 [ISO4]. If P2 is set to 0x0C, no data is returned.

Processing State returned in the Response Message

SW1	SW2	Meaning
62	83	Selected file deactivated
62	84	FCI not formatted according to ISO 7816-4 Section 5.1.5
6A	81	Function not supported
6A	82	File not found
6A	86	Incorrect parameters P1-P2
6A	87	L_c inconsistent with P1-P2
90	00	Successful execution

5.1.1.5 Select Master File APDU

This APDU selects the Master File or the root of a file system card directory structure.

Command Message

Function Code	0x0E
CLA	0x00
INS	0xA4
P1	0x03 - Select MF
P2	0x00 for response required, 0x0C for no response required
L_c	0x02
Data Field	File Identifier
L_e	Empty

Response Message

Data Field returned in the Response Message

If P2 is set to 0x00, data is returned as per ISO 7816-4 [ISO4]. If P2 is set to 0x0C, no data is returned.

Processing State returned in the Response Message

SW1	SW2	Meaning
62	83	Selected file deactivated
62	84	FCI not formatted according to ISO 7816-4 Section 5.1.5
6A	81	Function not supported
6A	82	File not found
6A	86	Incorrect parameters P1-P2
6A	87	L_c inconsistent with P1-P2
90	00	Successful execution

5.1.1.6 Update Binary APDU

This APDU is used to update the currently selected transparent file. All access control operations necessary for writing to the selected file must be completed before using this APDU.

Command Message

Function Code	0x03
CLA	0x00
INS	0xD6
P1	High-order byte of 2-byte offset
P2	Low-order byte of 2-byte offset
L_c	Number of bytes to update
Data Field	New data to be used to replace existing data
L_e	Empty

Response Message

Data Field returned in the Response Message

Empty.

Processing State returned in the Response Message

SW1	SW2	Meaning
63	CX	Successful updating after X retries, X=0 means no counter provided
65	81	Memory failure (unsuccessful updating)
67	00	Wrong length (wrong Lc field)
69	81	Command incompatible with file structure
69	82	Security status not satisfied
69	86	Command not allowed (no current EF)
6A	81	Function not supported
6A	82	File not found
6B	00	Wrong parameters (offset outside the EF)
90	00	Successful execution

5.1.2 Access Control APDUs

Table 5-5 shows the Access Control APDU set for file system and VM cards. The Access Control APDUs assume that the default cryptographic algorithm is `DES3-ECB`, with a double length key-size, 16 bytes.

Table 5-5: Access Control APDUs

Access Control APDUs								
FC	**Card Function**	**CLA**	**INS**	**P1**	**P2**	**L_c**	**Data**	**L_e**
0x0A	**EXTERNAL AUTHENTICATE**	0x00	0x82	AlgID	Key #	L_c	Cryptogram	–
0x05	**GET CHALLENGE**	0x00	0x84	0x00	0x00	–	–	L_e
0x09	**INTERNAL AUTHENTICATE**	0x00	0x88	AlgID	Key #	L_c	Challenge	L_e
0x08	**VERIFY**	0x00	0x20	0x00	CHV	L_c	Authentication data	–

Various smart cards perform external and internal authentication in similar but slightly different ways. The general methods used by the default GSC-IS APDU set are described below. To change the syntax and behavior of the default APDUs, the appropriate descriptor codes can be used in conjunction with command and response code tuples in the CCC as described in Chapter 6.

External Authentication Method:

1. The client application and the smart card share a secret key; the smart card may store the key in a key file.
2. The SPS instructs the smart card to issue an 8-byte challenge via the `GET CHALLENGE` APDU; the smart card returns the challenge to the SPS.
3. The client application encrypts the challenge with its secret key to produce a cryptogram.
4. The SPS sends the cryptogram to the smart card and possibly the key number via the `EXTERNAL AUTHENTICATE` APDU.
5. The smart card accesses the specified secret key, its saved copy of the challenge, and computes the same cryptogram and returns a status code to the SPS.
6. If the status code indicates that the cryptograms match, external authentication is successful.

Internal Authentication Method:

Step 1: PIN authentication

1. The client application and the smart card share a PIN; the smart card may store the PIN in a PIN file.
2. The SPS sends the PIN and the PIN number to the smart card via the `VERIFY` APDU.

3. The smart card accesses the specified PIN, compares it to the client application's PIN, and returns a status code to the SPS.

4. If the status code indicates that the PINs match, the smart card will permit the internal authentication to proceed.

Step 2: Internal Authentication

1. The client application and the smart card share a secret key; the smart card may store the key in a key file.

2. The client application computes an 8-byte challenge and sends this to the smart card along with the key number via the `INTERNAL AUTHENTICATION` APDU.

3. The smart card accesses the specified secret key, the challenge, and computes the same cryptogram.

4. The SPS retrieves the cryptogram in the response to the `INTERNAL AUTHENTICATION` APDU.

5. If the cryptograms match, internal authentication is successful.

Algorithm Identifiers for EXTERNAL and INTERNAL AUTHENTICATE APDUs:

ISO 7816-4 [ISO4] does not define algorithm identifiers for `EXTERNAL` and `INTERNAL AUTHENTICATE`, therefore this specification defines them in Table 5-6. If a smart card does not use the algorithm identifiers defined in Table 5-6, then the appropriate definitions of the `EXTERNAL` and `INTERNAL AUTHENTICATE` APDUs in the CCC command tuples will be required. If the smart card supports multiple cryptographic algorithms for this command, then successive tuples can be used to identify all the possible cryptographic algorithms and their corresponding P1 values.

Table 5-6: Algorithm Identifiers for Authentication APDUs

Algorithm Identifier	Algorithm-Mode	Key Length in Bits
0x00	Triple DES-ECB	128
0x01	Triple DES-CBC	128
0x02	DES-ECB	64
0x03	DES-CBC	64
0x04	RSA	512
0x05	RSA	768
0x06	RSA	1024
0x07	(Reserved for RSA 2048)	(2048)
0x08	AES-ECB	128
0x09	AES-CBC	128
0x0A	AES-ECB	192
0x0B	AES-CBC	192
0x0C	AES-ECB	256
0x0D	AES-CBC	256

Algorithm Identifier	Algorithm-Mode	Key Length in Bits
0x0E	RFU	–
0x0F	RFU	–

NOTE: High nibble of the Algorithm Identifier shall be zero.

5.1.2.1 External Authenticate APDU

This APDU is used in conjunction with the `GET CHALLENGE` APDU to authenticate a client application to the smart card. `GET CHALLENGE` would be issued first to cause the smart card to issue a random number, i.e., the challenge. The client application would encrypt the challenge and send the resultant cryptogram to the smart card via the `EXTERNAL AUTHENTICATE` APDU. The smart card would then decrypt it using the same algorithm as the client application and compare it to its internally stored copy of the challenge. If the cryptograms match, the client application is authenticated to the smart card. If the cryptograms do not match, the challenge is no longer valid.

Command Message

Function Code	0x0A
CLA	0x00
INS	0x82
P1	Algorithm Identifier – see Table 5-6
P2	0x00 for default key, 0x01 to 0x30 for key number
L_c	Length of data field
Data Field	Cryptogram
L_e	Empty

Response Message

Data Field returned in the Response Message

Empty.

Processing State returned in the Response Message

SW1	SW2	Meaning
63	00	No information given (Authentication failed)
63	CX	Authentication failed; X indicated number of further allowed retries
67	00	Wrong length (the Lc field is incorrect)
69	83	Authentication method blocked
69	84	Referenced data deactivated
69	85	Conditions of use not satisfied (the command is not allowed in this context)
6A	86	Incorrect parameters P1-P2
6A	88	Referenced data not found
90	00	Successful execution

5.1.2.2 Get Challenge APDU

This APDU is used to cause the smart card to generate a cryptographic challenge, e.g., a random number, for use in the subsequent security related procedure such as `EXTERNAL AUTHENTICATE`. The smart card saves a copy of the challenge internally until the completion of the security related procedure or an error occurs.

The challenge is valid only for the next APDU in the same card session.

Command Message

Function Code	0x05
CLA	0x00
INS	0x84
P1	0x00
P2	0x00
L_c	Empty
Data Field	Empty
L_e	Length in bytes of expected random challenge

Response Message

Data Field returned in the Response Message

If the APDU result indicates success, L_e number of bytes will be available to read from the smart card, i.e., the 8-byte challenge.

Processing State returned in the Response Message

SW1	SW2	Meaning
6A	81	Function not supported
6A	86	Incorrect parameters P1-P2
90	00	Successful execution

5.1.2.3 Internal Authenticate APDU

This APDU is used to authenticate the smart card to the client application. An 8-byte challenge is computed by the client application and then passed to the smart card via this command. Also passed are a key number and the cryptographic algorithm the smart card uses when encrypting the challenge. The smart card takes this information and encrypts the challenge according to the algorithm specified and the specified key and returns the resultant cryptogram. If the decrypted cryptogram from the smart card matches the initial challenge computed by the client application, the smart card is authenticated to the client application.

Command Message

Function Code	0x09
CLA	0x00
INS	0x88
P1	Algorithm Identifier – see Table 5-6
P2	0x00 for default key, 0x01 to 0x30 for key number
L_c	Length of data field
Data Field	Challenge
L_e	Length of expected cryptogram

Response Message

Data Field returned in the Response Message

The cryptogram.

Processing State returned in the Response Message

SW1	SW2	Meaning
69	84	Referenced data deactivated
69	85	Conditions of use not satisfied
6A	86	Incorrect parameters P1-P2
6A	88	Reference data not found
90	00	Successful execution

5.1.2.4 Verify APDU

This APDU is used to compare authentication data such as a password, key or PIN with corresponding authentication data on the smart card. The SPS sends the authentication data in this APDU and directs the smart card to compare it with authentication data on the smart card. The authentication data is passed unencrypted.

Command Message

Function Code	0x08
CLA	0x00
INS	0x20
P1	0x00
P2	0x00 for default key, 0x01 to 0x30 for key number
L_c	Length of data field
Data Field	Authentication data (i.e., password or PIN)
L_e	Empty

Note: If the Lc is 0x00 and the Data Field is empty, VERIFY returns the number of tries remaining on the referenced PIN.

Response Message

Data Field returned in the Response Message

Empty.

Processing State returned in the Response Message

SW1	SW2	Meaning
63	00	Verification failed
63	CX	Verification failed, X indicates the number of further allowed retries
69	83	Authentication method blocked
69	84	Referenced data deactivated
6A	86	Incorrect parameters P1-P2
6A	88	Reference data not found
90	00	Successful execution

5.1.3 Public Key Operations APDUs

Table 5-7 shows the public key operations APDUs for file system and VM cards. The default padding scheme for RSA is assumed to be `RSA_NO_PAD`. The computation is performed with the private key.

Table 5-7: Public Key Operations APDUs

Public Key Operations APDU								
FC	**Card Function**	**CLA**	**INS**	**P1**	**P2**	**L_c**	**Data**	**L_e**
0x05	**MANAGE SECURITY ENVIRONMENT**	0x00	0x22	0x41	0xB6	L_c	Key Reference information	–
0x0B	**PERFORM SECURITY OPERATION**	0x00	0x2A	0x9E	0x9A	L_c	Message digest to sign	L_e

5.1.3.1 Manage Security Environment APDU

This APDU is used to initiate the computation of a digital signature on a message by setting a digital signature template to be used by a subsequent `PERFORM SECURITY OPERATION` APDU.

Command Message

Function Code	0x05
CLA	0x00
INS	0x22
P1	0x41
P2	0xB6
L_c	L_c = Message length in bytes
Data Field	Key Reference information
L_e	Empty

Data Field: Key reference information, formatted as per ISO 7816-8 [ISO8].

Response Message

Data Field returned in the Response Message

Empty.

Processing State returned in the Response Message

SW1	SW2	Meaning
66	00	The Security Environment cannot be set
67	00	Wrong length (the Lc field incorrect)
6A	80	Invalid or missing tag, length or value in a Control Reference Data Object (CRDO)
6A	86	Incorrect parameters P1-P2
90	00	Successful execution

5.1.3.2 Perform Security Operation APDU

This APDU is used to initiate the computation of a digital signature on a message digest. This APDU responds with the computed signature.

Command Message

Function Code	0x0B
CLA	0x00
INS	0x2A
P1	0x9E
P2	0x9A
L_c	Length in bytes of message digest
Data Field	Message digest to sign
L_e	Length of response

Response Message

Data Field returned in the Response Message

The signed message digest.

Processing State returned in the Response Message

SW1	SW2	Meaning
67	00	Wrong length (the Lc field is incorrect)
69	81	Invalid file type
69	85	No preceding MSE-Set or previously specified key file is missing
69	87	Missing Secure Messaging Data Object
69	88	Incorrect Secure Message Data Object
6A	86	Incorrect parameters P1-P2
90	00	Successful execution
6C	XX	Wrong length (wrong L_e field; XX indicates the exact length)

5.2 Mapping Default APDUs to Native APDU Sets

For file system cards that contain a native APDU instruction set that differs from the GSC-IS default set, the SPS must implement a mapping mechanism to translate the default APDUs into the native APDUs in accordance with the information obtained from the CCC.

5.2.1 The CCC Command and Response Tuples

The CCC is a file that must be present on each conformant GSC-IS smart card. The CCC includes a set of tuples, which are 2-byte values that describe the differences in syntax between a file system card's native APDU set and the GSC-IS APDU set. Chapter 6 describes the contents of the CCC in more detail. Besides syntactical differences, the tuples also describe differences in APDU execution and data format. The codes used in the tuples to describe these differences are called Descriptor Codes.

As an example, Descriptor Codes can be used to indicate that a smart card's native `READ BINARY` APDU requires that offsets be on word boundaries as opposed to byte boundaries. Or, a smart card's native `EXTERNAL AUTHENTICATE` APDU may require 4 bytes of a cryptographic challenge whereas the default APDU requires 8 bytes. A descriptor code can be used to indicate that the SPS must build and send an APDU using a 4-byte cryptographic challenge.

A smart card with a native APDU instruction set identical to the GSC-IS APDU set would still contain a CCC, however the CCC would contain no tuples (and descriptor codes), since no APDU mapping would be necessary.

5.2.2 Native APDU Mapping and CCC Grammar

Each conformant SPS for file system cards must implement the translation or mapping mechanism to translate the default GSC-IS APDU set into a smart card's native APDU set both in syntax and in operation. The SPS performs this translation according to the rules of a CCC grammar associated with the set of tuples located in the smart card's CCC, described in more detail in Chapter 6.

The card edge interface for file system cards operates as follows:

1. A smart card vendor creates a CCC and loads it onto a smart card.

2. The SPS has knowledge of the default GSC-IS APDUs and how to translate them into a conformant card's native APDU set using the CCC grammar.

3. The smart card, when ready for use, is inserted into a reader.

4. The SPS's card edge locates and reads the contents of the CCC.

5. The SPS's card edge maps the default APDU set into the card's native set using the tuples in the CCC and the associated CCC grammar.

6. The SPS, when sending APDUs to the smart card, then uses the smart card's native ADPU set according to its rules of operation.

5.2.3 Detecting Card APDUs

The SPS can detect which of the default GSC-IS APDUs are available on a smart card according to the following rules:

1. If the APDU is defined in a capability tuple as not implemented (via Descriptor Code 0xFE, see Table 6-10), then the APDU is not available.

2. If the APDU is defined otherwise in one or more capability tuples, the APDU is available as defined.

3. If the APDU is not defined in any capability tuple, the APDU is assumed to be available and operates as described in this specification and in ISO 7816-4 [ISO4] and 7816-8 [ISO8].

The CCC optionally may contain a six-byte CARD APDUs bit-string for the purposes of informing the SPS which ISO 7816-4 [ISO4] and 7816-8 [ISO8] APDUs are available on the smart card. Each bit in the string, if set to 1, would indicate the presence of a corresponding APDU; a '0' would indicate the corresponding APDU is not present or is not to be used. The CARD APDUs string does not override any command tuples; however, if an APDU is described in command tuples but not in the CARD APDUs field, the command tuples are to be used. Table 5-8 shows bit positions and corresponding APDUs.

Table 5-8: CARD APDUs Values

Bit Position	7816-4 APDU
0	Reserved, Used for Shift Operation (see Section 6.4.2)
1	Select DF
2	Transparent Read (Binary)
3	Update Binary File
4	RFU
5	Manage Security Environment
6	Get Challenge
7	Get Response
8	Verify (CHV)
9	Internal Authenticate
10	External Authenticate
11	Perform Security Operation
12	Select File
13	Select EF (under current DF)
14	Select MF (root)
15	RFU

5.2.4 Default Status Code Responses

The default APDUs return status codes according to ISO 7816-4 [ISO4]. Non-ISO card-specific status codes can be mapped into a GSC-IS set of status code responses, shown in Table 5-9. As described in Chapter 6, the status codes can be mapped using the CCC grammar and status code tuples.

Table 5-9: GSC-IS 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

5.3 Card Edge Interface for VM Cards

The Card Edge Interface for VM Cards is made up of provider modules that provide three classes of services: generic container management services, symmetric key cryptographic services, and public (asymmetric) key cryptographic services. Each provider module may provide one or more class of service. These provider modules are implemented as on-card applets. For virtual machine cards, the terms “provider” and “applet” are synonymous.

Common interface methods that must be implemented by all providers are described first. The six APDUs listed in Table 5-12 must be implemented by all providers. The methods unique to each provider class are described in subsequent sections. Table 5-10 provides a summary of the APDUs implemented for the virtual machine card edge.

Table 5-10: Virtual Machine Card Edge APDUs

Virtual Machine APDU Set	
Common Interface Methods VM APDUs	**SELECT APPLET**
	SELECT OBJECT

Virtual Machine APDU Set	
	GET PROPERTIES
	GET ACR
	GET RESPONSE
	VERIFY PIN
Generic Container Provider VM APDUs	**READ BUFFER**
	UPDATE BUFFER
Symmetric Key Provider VM APDUs	**GET CHALLENGE**
	EXTERNAL AUTHENTICATE
	INTERNAL AUTHENTICATE
Public Key Provider VM APDUs	**READ BUFFER**
	UPDATE BUFFER
	PRIVATE SIGN/DECRYPT

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.

ACRs shall be coded as a single byte value (range 0x00 - 0xFF) as defined in Table 3-2.

5.3.2 Virtual Machine Card Edge General Error Conditions

Tables 5-11a and 5-11b apply to all virtual machine card edge APDUs:

Table 5-11a: Successful Conditions

Status bytes SW1 SW2	Meaning
61 LL	SW2 indicates the number of response bytes available
90 00	Normal ending of the command

Table 5-11b: General Error Conditions

Status bytes SW1 SW2	Meaning
62 00	Applet or instance logically deleted
63 CX	Authentication failed, X indicates the remaining tries
65 81	Memory failure
67 00	Incorrect parameter Lc
6C XX	Wrong length in Le parameter, SW2 indicates the exact length
69 82	Security status not satisfied
69 83	Authentication method blocked (ie. PIN code blocked)
69 85	Conditions of use not satisfied
69 99	Applet select failed
6A 80	Invalid parameters in command Data Field
6A 82	Applet or file not found
6A 84	Insufficient memory space to complete command
6A 86	Incorrect P1 or P2 parameter
6A 88	Referenced data not found
6D 00	Unknown instruction given in the command
6E 00	Wrong class given in the command
6F 00	Technical problem with no diagnostic given

5.3.3 Common Virtual Machine Card Edge Interface Methods

The common virtual machine APDUs are shown in Table 5-12.

Table 5-12: Common VM APDUs

Card Function	CLA	INS	P1	P2	L_c	Data	L_e
SELECT APPLET	0x00	0xA4	0x04	0x00	L_c	AID	–
SELECT OBJECT	0x00	0xA4	0x02	0x00	L_c	File ID	–
GET PROPERTIES	0x00	0x56	P1	0x00	L_c	Requested Tags	–
GET ACR	0x80	0x4C	P1	0x00	L_c	AID or Object ID	–
GET RESPONSE	0x00	0xC0	0x00	0x00	–	–	L_e
VERIFY PIN	0x00	0x20	0x00	0x00	L_c	PIN	–

5.3.3.1 Access Control

A fixed set of Access Control Rules (ACR) are assigned to the Common Virtual Machine Card Edge Interface APDU commands as defined in Table 5-13:

Table 5-13: ACRs assigned to the Common VM CEI

APDU	ACR
Get Properties	BSI_ACR_ALWAYS
Get ACR	BSI_ACR_ALWAYS
Get Challenge	BSI_ACR_ALWAYS
External Authenticate	BSI_ACR_ALWAYS
Get Response	–
Verify PIN	BSI_ACR_ALWAYS

5.3.3.2 Select Applet APDU

The command is used to select the instance of an applet using its AID.

Command Message

CLA	0x00
INS	0xA4
P1	0x04
P2	0x00
Lc	Length of the applet AID
Data Field	Applet AID (between 5 and 16 bytes in length).
Le	Empty

Response Message

Data field returned in the response message

Empty.

Processing state returned in the response message

If the applet is not found on the smart card, the ISO 7816-4 [ISO4] status condition: '6A82' is returned (status bytes SW1,SW2=0x6A,0x82). For other status conditions see section General Error Conditions in Section 5.3.2.

5.3.3.3 Select Object APDU

The command is used to select a container managed by an applet.

Command Message

CLA	0x00
INS	0xA4
P1	0x02
P2	0x00
L_c	Length of the object ID, 2 bytes.
Data Field	Object ID.
L_e	Empty

Response Message

Data field returned in the response message

Empty.

Status bytes returned in the response message

If the object is not found, the ISO 7816-4 [ISO4] status condition: '6A82' is returned (status bytes SW1=0x6A, SW2=0x82). For other status conditions see section General Error Conditions in Section 5.3.2.

5.3.3.4 Get Properties APDU

This command is used to retrieve applet instance properties of a currently selected applet.

Command Message

CLA	0x00
INS	0x56
P1	Requested properties information type
P2	0x00
Lc	If P1=0x02 then length of list of requested tags, else empty.
Data Field	If P1=0x02 then list of requested tags, else empty.
Le	Expected applet instance properties length

Reference control parameter P1

The reference control parameter P1 shall be used to indicate the type of requested properties information. The following P1 values are possible:

0x00: Get a GSC-IS v2.0 compatible properties response message. If this response cannot be supported by the smart card then an error (0x6A86) shall be returned.

0x01: Get all the properties.

0x02: Get the properties of the tags provided in list of tags in the command data field.

Data field sent in the command message

This field is present only when P1 is 0x02. In that case, this data field is composed of the list of tags to be requested from the applet instance (the tag values, 1 byte each, are chained).

Response Message

Data field returned in the response message when P1 is 0x00

The Data field returned in the response message contains the values of the following properties:

- Applet family (1 byte)
- Applet version (4 bytes)
- RFU byte
- RFU byte
- ID/CHV-applet AID length (1 byte)
- ID/CHV-applet AID (always 16 bytes padded with 0 if necessary) – AID of the ID/CHV applet instance that shall be used for Card Holder Verification (CHV)

- Key Set Version (1 byte)1
- Key Set Id (1 byte) 2
- T-Buffer length (2 bytes)
- V-Buffer length (2 bytes)
- X bytes of applet specific information and RFU to complement to 64 bytes.

Data field returned in the response message when P1 is 0x01 or 0x02

The data field returned in the response message contains the current value of all the properties when P1 is 0x01 or the current value of the requested properties when P1 is 0x02. The properties are returned in a single buffer containing a list of TLVs packed end-to-end according to the table below. The scope of these tags is specific to the properties object and should not be confused with the GSC and CAC data model tags.

Tag	Length	Value
0x01	5	**Applet Information**
		Applet Family (1 byte)
		Applet version (4 bytes)
0x40	1	**Number of objects managed by this instance**
0x50	11	**First TV-Buffer Object**
0x41	2	ObjectID (2 bytes)
0x42	5	Buffer Properties (5 bytes)
		Type of Tag Supported (1 byte)
		T-Buffer length (2 bytes): LSB, MSB
		V-Buffer length (2 bytes): LSB, MSB
		(Next TV-Buffer Object…)
0x50	11	**Last TV-Buffer Object**
0x41	2	ObjectID (2 bytes)
0x42	5	Buffer Properties (5 bytes)
		Type of Tag Supported (1 byte)
		T-Buffer length (2 bytes): LSB, MSB
		V-Buffer length (2 bytes): LSB, MSB
0x51	17	**First PKI Object**
0x41	2	ObjectID (2 bytes)
0x42	5	Buffer Properties
		Type of Tag Supported (1 byte)
		T-Buffer length (2 bytes): LSB, MSB
		V-Buffer length (2 bytes): LSB, MSB
0x43	4	PKI Properties
		Algorithm ID (1 byte)

[1] Key Set and Key Levels are applicable to v2.0 for backward compatibility.
[2] Key Set ID refers to the key number and the Key Level is used to indicate whether the referenced key is part of the READ or WRITE Key Set.

Tag	Length	Value
		Key Length Bytes / 8 (1024 bits -> 128 bytes-> 0x10) (1 byte)
		Private Key Initialized (1 byte)
		Public Key Initialized (1 byte)

Processing state returned in the response message

If the properties retrieval succeeds, SW1 = 0x61 and SW2 = size of next block of data available to read.

If P1 = 0x00 cannot be supported by the smart card, SW1 = 0x6A and SW2 = 86.

SW1	SW2	Meaning
61	LL	More data available, 0xLL specifying the size of next block to read.
6A	86	P1 or P2 parameter not supported.

For other status conditions see Table 5-11b.

5.3.3.5 Get ACR APDU

This command is used to retrieve Access Control Rule properties.

Command Message

CLA	0x80
INS	0x4C
P1	Reference Control Parameter P1
P2	0x00
Lc	If P1=0x00, 0x10, 0x20, or 0x21 then empty. If P1=0x01 then the length of the ACRID (0x01). If P1=0x11 then the length of the AID (<=0x10). If P1=0x12 then the length of object ID (0x02)
Data Field	If P1 = 0x00, 0x10, 0x20, 0x21 then empty. If P1=0x01 then the value of the ACRID. If P1=0x11 then the value of the AID. If P1=0x12 then the value of the object ID.
Le	Empty.

Reference control parameter P1

The reference control parameter P1 shall be used to indicate the type of requested ACR properties information. The following P1 values are possible:

0x00: All ACR table entries are to be extracted.

0x01: Only one entry of the ACR table is extracted based on ACRID.

0x10: All Applet/Object ACR table entries are to be extracted.

0x11: Only the entries of the Applet/Object ACR table for one applet are extracted based on applet AID.

0x12: Only one entry of the Applet/Object ACR table for an object is extracted based on object ID.

0x20: The Access Method Provider table is extracted.

0x21: The Service Applet table is extracted.

Data field sent in the command message

This field is present only when P1 is 0x11 or 0x12. If P1 equals 0x11, it contains the AID value of the applet for which the Applet/Object ACR table is to be extracted. If P1 equals 0x12, it contains the Object ID value of the object for which the Applet/Object ACR table is to be extracted.

Response Message

Data field returned in the response message

The following tables may be retrieved:

- **ACR table:** This table maps the Access Control Rule Type (`ACRType`) and Access Method information to the Access Control Rule Identifier (`ACRID`) for each Access Control Rule.
- **Applet/Object ACR table:** This table maps the service (INS code/P1 byte/P2 byte/1st data byte) to the `ACRID` for each container.
- **Access Method Provider table:** This table maps the Access Method Provider ID to the full `AID` for each Access Method Provider.
- **Service Applet table:** This table maps the Service Applet ID to the full `AID` for each Service Applet.

The data fields returned in the response message may contain all the entries for a table or only the requested ones depending on the command parameters.

The following entry is always returned and precedes any ACR table, Applet/Object ACR table or Authentication Method Provider table.

Table 5-14: Applet Information String

Tag	Length	Value
0x01	5	**Applet Family of** Access Control Applet **(ACA) (1 byte)**
		Applet version of ACA (4 bytes)

In addition to the common Applet Information entry the following entries are conditionally returned depending on the reference control parameter P1.

Data field returned in the response message when P1 is 0x00

The data field returned in the response message contains all the entries of the ACR table.

Table 5-15: ACR Table

Tag	Length	Value
0xA1	1	**Number of ACR entries (unique ACRID)**
0xA0	*	**First ACR entry (structured as follows)**
		ACRID of ACR entry (1 byte)
		ACRType (as defined in Table 3-2) (1 byte)
		Number of AccessMethods in this ACR (1 byte)
		First AccessMethodProviderID (1 byte)
		First keyIDOrReference (1 byte)
		(Next AccessMethod…)
		Last AccessMethodProviderID (1 byte)
		Last keyIDOrReference (1 byte)
0xA0	*	**(Next ACR entry …)**

* Denotes Variable length field

Data field returned in the response message when P1 is 0x01

The data field returned in the response message a single entry of the ACR table based on ACRID.

Table 5-16: Applet/Object ACR Table

Tag	Length	Value
0xA0	*	**ACR entry corresponding to ACRID sent**
		ACRID of ACR entry (1 byte)
		ACRType (as defined in Table 3-2) (1 byte)
		Number of AccessMethods in this ACR (1 byte)
		First AccessMethodProviderID (1 byte)
		First keyIDOrReference (1 byte)
		(Next AccessMethod…)
		Last AccessMethodProviderID (1 byte)
		Last keyIDOrReference (1 byte)

* Denotes Variable length field

Data field returned in the response message when P1 is 0x10

The data field returned in the response message contains all entries of the Applet/Object ACR table.

Table 5-17: Access Method Provider Table

Tag	Length	Value
0x81	1	**Number of applets managed by this ACA**
0x80	Length is 2 plus length of nested TLV fields 0x82	**Card Applet ACR structured as follows**
		Applet ID (1 byte)
		Number of objects managed by this applet (1 byte)
0x82	*	Card Object ACR structured as follows
		Card Object ID (2 bytes)
		INS1 Code (1 byte)
		INS1 Configuration Definition - 0000 0 $b_2b_1b_0$(1 byte) If b_0=1 then P1 byte is present. If b_1=1 then P2 byte is present. If b_2=1 then first data field byte is present.
		P1 Value – OPTIONAL (1 byte)
		P2 Value – OPTIONAL (1 byte)
		First Data Byte Value – OPTIONAL (1 byte)
		ACRID (1 byte)
		INSx …
0x82	*	(Next Card Object ACR…)
0x80	*	**(Next Card Applet ACR…)**

* Denotes Variable length field

Data field returned in the response message when P1 is 0x11

The data field returned in the response message contains the entries of the Applet/Object ACR table for a single applet based on AID.

Table 5-18: Service Applet Table

Tag	Length	Value
0x80	Length is 2 plus length of nested TLV fields 0x82	**Applet ACR table based on applet AID entered**
		Applet ID (1 byte)
		Number of objects managed by this applet (1 byte)
0x82	*	Card Object ACR structured as follows
		Card Object ID (2 bytes)
		INS1 Code (1 byte)
		INS1 Configuration Definition - 0000 0 $b_2b_1b_0$(1 byte) If b_0=1 then P1 byte is present. If b_1=1 then P2 byte is present. If b_2=1 then first data field byte is present.
		P1 Value – OPTIONAL (1 byte)
		P2 Value – OPTIONAL (1 byte)
		First Data Byte Value – OPTIONAL (1 byte)
		ACRID (1 byte)
		(INSx …)
0x82	*	(Next Card Object ACR…)

* Denotes Variable length field

Data field returned in the response message when P1 is 0x12

The data field returned in the response message contains the entry of the Applet/Object ACR table for a single object based on OID.

Table 5-19: Applet/Object ACR table for a Single Object

Tag	Length	Value
0x82	*	**Card Object ACR (structured as follows)**
		Card Object ID (2 bytes)
		INS1 Code (1 byte)
		INS1 Configuration Definition - 0000 0 $b_2b_1b_0$(1 byte) If b_0=1 then P1 byte is present. If b_1=1 then P2 byte is present. If b_2=1 then first data field byte is present.
		P1 Value – OPTIONAL (1 byte)
		P2 Value – OPTIONAL (1 byte)
		First Data Byte Value – OPTIONAL (1 byte)
		ACRID (1 byte)
		(INSx …)

* Denotes Variable length field

Data field returned in the response message when P1 is 0x20

The data field returned in the response message contains all the entries of the Access Method Provider table.

Table 5-20: Access Method Provider Table

Tag	Length	Value
0x91	1	**Number of AMP entries**
0x90	Length includes nested TLV structure 0x92	**AMP entry (structured as follows)**
		Access Method provider ID (short form) (1 byte)
0x92	*	Access Method provider AID
0x90	*	**(Next AMP entry…)**

Data field returned in the response message when P1 is 0x21

The data field returned in the response message contains all the entries of the Service Applet table.

Table 5-21: Service Applet Table

Tag	Length	Value
0x94	1	**Number of Applet entries**
0x93	*	**Applet entry (structured as follows)**
		Applet ID (short form) (1 byte)
0x92	*	Applet AID
0x93	*	**(Next Applet entry)**

* Denotes Variable length field

Processing state returned in the response message

If properties retrieval succeeds, SW1 = 0x61 and SW2 = size of next block of data available to read.

SW1	SW2	Meaning
61	LL	More data available, 0xLL specifying the size of next block to read.

For other status conditions see Table 5-11b.

5.3.3.6 Get Response APDU

The GET RESPONSE APDU is used to retrieve from the smart card the response message of the immediately preceding APDU in the case that this APDU has returned a processing state of 61xx indicating that a response message of xx bytes is available.

Command Message

CLA	0x00
INS	0xC0
P1	0x00
P2	0x00
L_c	Empty
Data Field	Empty
L_e	Number of bytes to read in response

Response Message

Data field returned in the response message

If the APDU result indicates success, L_e number of bytes will be available to read from the smart card.

Processing state returned in the response message

See Table 5-11b.

5.3.3.7 Verify PIN APDU

The VERIFY command is used to verify the global PIN code, or to check if the PIN code verification is required, or to check whether or not the PIN code has been already verified. The global PIN is a root level key.

Command Message

CLA	0x00
INS	0x20
P1	0x00
P2	0x00
Lc	0xNN (Effective PIN length, 0x00 indicates no PIN present)
Data Field	PIN code to be verified
Le	Empty

Note: The maximum effective PIN length is dependent on the card platform.

Data field sent in the command message

If the data length and the data field sent in the command message are empty (data field does not include a PIN code), the command corresponds to a PIN verify check command, and it is used to determine if the PIN code verification is necessary and whether or not the PIN code has been already verified.

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 blocked. If the verification succeeds, the PIN-verified flag value and the PIN-always flag value are both set to 0x01.

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 succeeds, SW1=0x90 and SW2=0x00.

If PIN verification fails, the status returned is SW1=0x63, SW2=0xCX where X is number of remaining PIN tries.

If PIN verify check command is submitted and PIN is already verified, SW1=0x90 and SW2=0x00, otherwise SW1 = 0x63, SW2 = 0xCX, where X = number of remaining PIN tries.

SW1	SW2	Meaning
90	00	PIN verification succeeds
63	CX	PIN not verified and X indicates the remaining tries
69	83	PIN code blocked
6A	88	No PIN code defined

5.3.4 Generic Container Provider Virtual Machine Card Edge Interface

Table 5-22 shows the Generic Container Provider VM APDUs. As described in Chapter 8, containers accessed by these APDUs are split into two buffers: a TL buffer containing Tag and associated Length values, and a V buffer containing the values identified by the corresponding Tags and Lengths.

Table 5-22: Generic Container VM APDUs

Card Function	CLA	INS	P1	P2	L_c	Data	L_e
READ BUFFER	0x80	0x52	Off/H	Off/L	0x02	Buffer and number bytes to read	–
UPDATE BUFFER	0x80, 0x84	0x58	Off/H	Off/L	L_c	Buffer and data to update	–

5.3.4.1 Update Buffer APDU

This command allows updating all or part of a buffer.

Command Message

CLA	0x80
INS	0x58
P1	Reference Control Parameter P1
P2	Reference Control Parameter P2
Lc	1+ Length of data to be updated
Data Field	Buffer (1 byte) + 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

SW1	SW2	Meaning
67	00	Invalid command data length
6A	86	Wrong P1/P2 (Try to update data out of the buffer)
6A	88	No corresponding buffer (invalid Buffer Type)

5.3.4.2 Read Buffer APDU

This command allows reading all or part of a buffer.

Command Message

CLA	0x80
INS	0x52
P1	Reference Control Parameter P1
P2	Reference Control Parameter P2
Lc	0x01 + 0x01 = 0x02
Data Field	Buffer type (1 byte value) followed by the data length to read (1 byte value)
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 smart card, according to the P1, P2 parameters (offset indicating from where to read data) or empty if `GET RESPONSE` command is required to receive data read from the smart card.

Processing state returned in the response message

If `READ BUFFER` command was successful, SW1=0x90 and SW2=0x00, any available data is returned in the data field of the response message. If command is successful and SW1=0x61, SW2 contains bytes remaining to be read from the smart card with subsequent `GET RESPONSE` commands.

SW1	SW2	Meaning
67	00	Invalid command data length
6A	86	Wrong P1/P2 (Try to update data out of the buffer)
6A	88	No corresponding buffer (invalid Buffer Type)

5.3.5 Symmetric Key Provider Virtual Machine Card Edge Interface

Table 5-23 shows the Symmetric Key Provider VM APDUs.

Table 5-23: Symmetric Key VM APDUs

Card Function	CLA	INS	P1	P2	L_c	Data	L_e
GET CHALLENGE	0x00	0x84	0x00	0x00	–	–	L_e
EXTERNAL AUTHENTICATE	0x00	0x82	AlgID	Key #	L_c	Cryptogram	–
INTERNAL AUTHENTICATE	0x00	0x88	AlgID	Key #	L_c	Challenge	L_e

5.3.5.1 Get Challenge APDU

The `GET CHALLENGE` command is the first step of the host authentication process and is followed immediately by the `EXTERNAL AUTHENTICATE` command. The computed challenge is valid only for the following `EXTERNAL AUTHENTICATE` APDU.

Command Message

CLA	0x00
INS	0x84
P1	0x00
P2	0x00
Lc	Empty
Data Field	Empty
Le	Challenge length

Response Message

Data field returned in the response message

The response message contains the challenge used later for authentication.

Processing state returned in the response message

See Table 5-11b.

Note: The computed challenge must be stored within the applet instance in order to evaluate the expected `EXTERNAL AUTHENTICATE` command. The client application shall encrypt the challenge received from the smart card using a cryptographic algorithm known by the smart card and the corresponding shared key. The cryptographic algorithm is `DES3-ECB` with a 16-byte key. The encrypted challenge shall then be submitted to the smart card using the `EXTERNAL AUTHENTICATE` command.

5.3.5.2 External Authenticate APDU

This `EXTERNAL AUTHENTICATE` command is a subset of the ISO 7816-4 [ISO4] standard command. The default cryptographic algorithm is `DES3-ECB` with double length key size (16 bytes) and an 8-byte challenge requested from the smart card using the `GET CHALLENGE` command just before the authentication command is submitted. This command is introduced to allow external authentication with different cryptographic algorithms selected through the P1 parameter and multiple key sets if same data is updated by different applications that do not desire to share their keys.

Command Message

CLA	0x00
INS	0x82
P1	Algorithm identifier and security level
P2	0x00 for default key, 0x01 to 0x30 for key number
L_c	Length of the cryptogram
Data Field	Cryptogram
L_e	Empty

P1: **0xAS** where **A** specifies the algorithm identifier using the 4-MSb of **P1** and **S** defines the secure messaging and command encryption as described in the table below, using the 4-LSb of the parameter

Table 5-6 contains the algorithm identifiers.

P1								Meaning of	
b8	b7	B6	b5	b4	b3	b2	b1	A (b8-b5)	S(b4-b1)
0	0	0	0	0	0	0	0	Default algorithm or already known	No secure messaging expected
0	0	0	0	0	0	0	1	Default algorithm or already known	Secure messaging C-MAC (Global Platform)
0	0	0	0	0	0	1	1	Default algorithm or already known	Command encryption and C-MAC (Global Platform)
–	–	–	–	0	0	0	0	Algorithm Identifier	No secure messaging expected
–	–	–	–	0	0	0	1	Algorithm Identifier	Secure messaging C-MAC (Global Platform)
–	–	–	–	0	0	1	1	Algorithm Identifier	Command encryption and C-MAC (Global Platform)

Response Message

Data field returned in the response message

Empty.

Processing state returned in the response message:

For specific status conditions see Table 5-11b.

5.3.5.3 Internal Authenticate APDU

This command is used to perform a challenge-response authentication.

Command Message

CLA	0x00
INS	0x88
P1	0x00 for the default DES3-ECB or Algorithm ID as defined in the CCC
P2	0x00 for default key, 0x01 to 0x30 for key number
Lc	Length of the subsequent data field
Data Field	Authentication related data (e.g. Challenge)
Le	0xLL Maximum number of bytes expected in response

Data field sent in the command message

The data field contains the data to be encrypted by the smart card using the selected key.

Response Message

Data field returned in the response message

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

Processing state returned in the response message

See Table 5-11b.

5.3.6 Public Key Provider Virtual Machine Card Edge Interface

The Public Key Provider VM APDU set consists of one APDU, the `PRIVATE SIGN/DECRYPT` APDU as detailed in Section 5.3.6.1.

5.3.6.1 Private Sign/Decrypt APDU

This command is used to perform an RSA signature or data decryption.

Command Message

CLA	0x80
INS	0x42
P1	0x00
P2	0x00
Lc	Data Field length
Data Field	Data to sign or decrypt
Le	Expected length of the signature/decryption

Data field sent in the command message

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

The data must be already padded before the message is sent.

Response Message

Data field returned in the response message

The data field in the response message contains the data signed or decrypted. The client application is responsible for any data padding.

Processing state returned in the response message

See Table 5-11b.

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. This mechanism is based on the GSC-IS Card Capability grammar. The differences between a smart card's APDU set and the standard APDU set defined by the VCEI are carried on the smart card in the CCC.

Each GSC-IS conformant smart card shall contain a CCC and support a standard procedure for accessing it as defined in Section 6.2. The contents of a CCC shall conform with the formal card capabilities grammar defined in this chapter.

Virtual Machine cards can be programmed to directly implement the VCEI APDU set. However, Virtual Machine cards shall still contain a CCC.

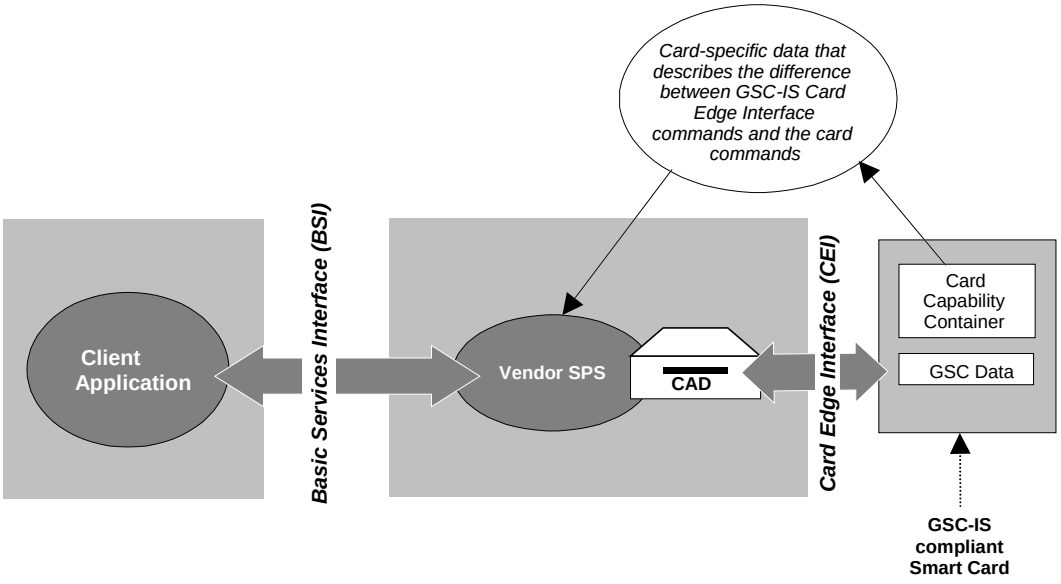


Figure 6-1: The Card Capability Container

Before the card-specific APDU definitions can be used to communicate with the smart card, the CCC must be read.

6.2 Procedure for Accessing the CCC

The CCC is designated by the Capabilities Application Identifier (AID: GSC-RID||DB00). The Universal AID of the smart card CCC shall be 0xA000000116DB00. The CCC shall be the default container of a CCC applet on a VM card. This container shall be selected by default when the CCC applet is selected.

The CCC is implemented as a transparent (binary) file on file system cards. The GSC CCC Elementary File (EF) shall be contained in the Master Directory (FID: "**0x3F00"**) and is designated by the Capabilities Application Identifier (AID: GSC-RID||DB00) as well as the FID: "**0xDB00"**.

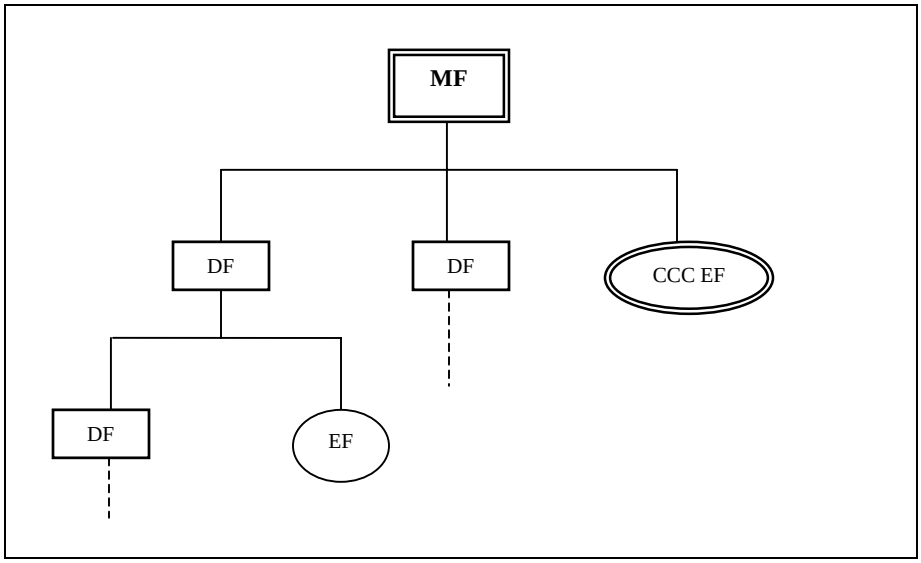


Figure 6-2: Location of the CCC Elementary File in a file system card

6.2.1 General CCC Retrieval Sequence

The CCC shall be stored under a known AID on Virtual Machine cards and a known FID under File system cards. The following CCC retrieval sequence shall be executed after an ATR (Answer-To-Reset)

to the smart card. The retrieval sequence is used to determine which card edge interface is implemented, virtual machine card edge or File system card edge and then to read the CCC. Once the ATR is successful, the SPS first attempts to retrieve the CCC using the procedure for Virtual machine cards. If this fails, the SPS then attempts to read the CCC using the file system card procedure. If that also fails, the SPS assumes that the smart card does not contain a CCC and is not GSC-IS conformant.

The procedure for the retrieval of the CCC is as follows:

1. The SPS sends a `SELECT APPLET` APDU to the smart card as shown in the following table:

CLA	INS	P1	P2	Lc	DATA
0x00	0xA4	0x04	0x00	Length of AID	AID

2. The CCC applet is selected on a VM card if the smart card returns the status bytes “0x9000” or “0x61LL” (“LL” indicates more data available). If not, the SPS then attempts to use the File system procedure to access the CCC as defined in steps 4-8.

3. A successful applet selection is followed by an attempt to read the CCC by sending a `READ BUFFER` APDU command as specified in the Card Edge Interface for VM cards. The `READ BUFFER` APDU is sent as follows:

CLA	INS	P1	P2	Lc	DATA
0x80	0x52	P1	P2	0x02	Buffer type +data length to read

Note 1 : Reference Control Parameter P1/P2 : See Card Edge interface for VM (Chapter 5, Section 5-43)

Note 2 : The first buffer to be read is the TL-Buffer (Buffer type = “0x01”), the second buffer to be read is the V-Buffer (Buffer type = “0x02).

Note 3 : The “data length to read” is application/vendor specific, but in practice it is advisable to set it to 64.

If no error status bytes are returned, the smart card will return the data read from the card with “0x9000” status byte to indicate complete completion or “0x61LL” to indicate that “LL” bytes are still available to read. The TL-Buffer and the V-Buffer shall be entirely read.

If an error status byte is returned and the card does not support the `READ BUFFER` APDU command, the SPS attempts to use the File system card edge by sending a `READ BINARY` APDU with CLA=”0x00” as defined in step 5. If this succeeds, the VM card is using the File system card edge APDUs. If this fails and the smart card does not support `READ BINARY` either, the smart card is not GSC-IS compliant.

4. For the file system card, the SPS sends a sequence of APDUs to the smart card until the CCC is successfully read. This sequence selects the Master File (MF) using its reserved FID value “0x3F00”, then the CCC Elementary File (EF) using its reserved FID value “0xDB00”, and then performs a binary read operation on that CCC Elementary File.

The SPS sends a `SELECT MF` APDU command as follows:

CLA	INS	P1	P2	Lc	DATA
TEST CLA	0xA4	0x03	0x00	0x02	0x3F00

Note 1 : The default TEST CLA values are: 0x00, 0xC0, 0xF0, 0x80, 0xBC, 0x01. The CLA value "0x00" is ISO 7816-4 [ISO4] conformant. The value "0x00" shall be the first to be tested. (Additional test values for CLA are: 0x90, 0xA0, 0xB0-0xCF.)

5. If the returned status byte is "0x6E00", the tested Class byte is not supported. The SPS loops back to step 4 and attempts the next CLA value.

6. If the returned status byte is "0x9000" or "0x61LL" ("LL" indicates more data available), then the command structure and CLA value are correct.

7. Once CLA has been determined, the SPS **selects (CCC) EF under MF** as follows:

CLA	INS	P1	P2	Lc	DATA
Determined CLA	0xA4	0x02	0x00	0x02	0xDB00

The CCC EF is selected if no error codes are returned.

8. Then to Read a binary file (with no secure messaging), the SPS uses the following `READ BINARY` APDU **on the selected CCC EF**:

CLA	INS	P1	P2	Le
Determined CLA	0xB0	Off/H	Off/L	Le

Note 2: P1, P2 and Le are as defined in Section 5.1.1.2

Note 3: SPS implementations should define a timeout value to avoid an infinite wait for a response from the smart 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.

6.2.2 Card Capabilities Container Structure

For a file system card, the Card Capability Container shall be an elementary file. The file consists of a string of SIMPLE Tag-Length-Value (TLV) data objects with no encoding, with the exception of fields that use structured SIMPLE TLV ("Application CardURL" and "Access Control Rule Table" fields).

For a VM card, the Card Capability Container shall be the default container (buffer) managed by the CCC applet. The internal format of that CCC container is defined in Section 8.2.

For both card types, the CCC is configured for `ALWAYS READ`. However, it is up to each implementer to define write/modify rules.

Table 6-1: CCC Fields

Card Capabilities Container		FID: 0xDB00	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	
Access Control Rule Table	0xF6	Variable – Multiple Objects	
CARD APDUs	0xF7	Fixed: 6 bytes	
Redirection Tag	0xFA	Variable	
Capability Tuples (CTs)	0xFB	Variable: Collection of 2 byte Tuples	
Status Tuples (STs)	0xFC	Variable: Collection of 3 byte Tuples	
Next CCC	0xFD	Application Card URL, 20 bytes or greater	
Optional Issuer Defined Objects	Issuer Defined	Variable	
Error Detection Code	0xFE	LRC	

6.3 CCC Fields

Sections 6.3.1 through 6.3.9 describe the CCC fields defined in Table 6-1. The smart 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.

6.3.1 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. This identifer 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) : ‘0x01’,
      javaCard                [1] BIT STRING SIZE(8) : ‘0x02’,
      Multos                  [2] BIT STRING SIZE(8) : ‘0x03’,
      JavaCardFS              [3] BIT STRING SIZE(8) : ‘0x04’,
      ...
}
```

`JavaCardFS` refers to a Java Card implementing the file system card edge defined in Chapter 5.

6.3.2 Capability Container Version Number

The Capability Container Version Number field describes the version of the card capability container. The field is of length one byte; the high order nibble of the byte describes the major version number, and the low order nibble of the byte describes the minor version number.

```
CapabilityContainerVersion ::= SEQUENCE {
      MajorVersion            BIT STRING SIZE(4),
      MinorVersion            BIT STRING SIZE(4)
}
```

For instance, for this version of the CCC, the high order nibble would contain the number 2, and the low order nibble would contain the number 1, to correspond to version 2.1.

6.3.3 Capability Grammar Version Number

The Capability Grammar Version Number field describes the version of the Card Capability Container grammar. The field is of length one byte; the high order nibble of the byte describes the major version number, and the low order nibble of the byte describes the minor version number.

```
CapabilityGrammarVersion ::= SEQUENCE {
      MajorVersion            BIT STRING SIZE(4),
      MinorVersion            BIT STRING SIZE(4)
}
```

For instance, for this version of the Card Capability Container grammar, the high order nibble would contain the number 2, and the low order nibble would contain the number 1, to correspond to version 2.1.

6.3.4 Applications CardURL Structure

The Card Capabilities Container may contain multiple instances of `ApplicationsCardURL` structures, each denoted by the tag value “0xF3”. 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).

The structure of the `ApplicationsCardURL` is denoted {T-L-{T1-L1-V1} … {Tn-Ln-Vn}} with a tag field followed by a length field encoding a number. If the number is not zero, then the value field of the constructed data object, called "template" in ISO/IEC 7816, consists of one or more SIMPLE TLV data objects, each one consisting of a tag field, a length field encoding a number and if the number is not zero, a value field.

6.3.5 PKCS#15

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

6.3.6 Registered Daa Model Number

The Registered Data Model Number indicates the registered Data Model in use by the smart card.

6.3.7 Access Cntrol Rules Table

The Access Control Rules Table allows Access Control Rules to be recorded only once in the card. The table definition is either stored directly in the CCC or in the Access Control Applet (ACA) of a VM card in which case the CCC has a reference to the AID of the Access Control Applet.

For additional information on structure format, see Section 6.3.4.

```
ACRTableOrAIDReference ::= CHOICE {
      acrTable          [0] ACRTable,
      acrTableAID       [1] STRING SIZE(16)
}

ACRTable ::= SEQUENCE {
      acrs                    SEQUENCE OF ACR,
      accessMethods           SEQUENCE OF AccessMethod,
      accessMethodProviders   SEQUENCE OF AccessMethodProvider
}

ACR ::= SEQUENCE {
      acrID                   BIT STRING SIZE(8),
      acrType                 BIT STRING SIZE(8),
      accessMethodIDs         SEQUENCE OF AccessMethodID
}

AccessMethodID ::= BIT STRING SIZE(8)

AccessMethod ::= SEQUENCE {
      accessMethodID          BIT STRING SIZE(8),
      accessMethodProviderID  BIT STRING SIZE(8),
      keyIDOrReference        BIT STRING SIZE(8)
}

AccessMethodProvider ::= SEQUENCE {
      accessMethodProviderID        BIT STRING SIZE(8),
      accessMethodProviderAID       STRING SIZE(16)
}
```

6.3.8 Card APDUs

The card capability container optionally may contain a 6-byte Card APDUs field for the purposes of informing the SPS which ISO 7816-4 [ISO4] and 7816-8 [ISO8] APDUs are available on the smart card. Each bit in the string, if set to 1, would indicate the presence of a corresponding APDU. The Card APDUs field is described in more detail in Section 5.2.3.

6.3.9 Reirection Tag

In the case an implementer 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, Data Model 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 DM can have any number of "redirection flags" to handle Tag level exceptions to the nominal DM.

6.3.10 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 smart 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 smart 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 VCEI's status codes and the status codes defined in ISO 7816-4 [ISO4].

Sections 6.3.11 through 6.3.14 describe the construction of tuples in more detail.

6.3.11 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` APDU, 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` APDU, or the CHV level for a `VERIFY PIN` APDU. Whether the V-byte is a constant value or a descriptor code is determined by the 7th bit (most significant 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 in Table 6-3.

Table 6-3: Parameter and Function Codes

Parameter Codes	
0x00	DATA
0x01	CLA
0x02	INS
0x03	P1
0x04	P2
0x05	P3*
0x06	Prefix
0x07	Suffix

Function Codes	
0x00	Reserved, Used for Shift Operation (see Section 6.4.2)
0x01	Select DF
0x02	Transparent Read (Binary)
0x03	Update Binary File
0x04	RFU
0x05	Manage Security Environment
0x06	Get Challenge
0x07	Get Response
0x08	Verify (CHV)
0x09	Internal Authenticate
0x0A	External Authenticate
0x0B	Perform Security Operation
0x0C	Select File
0x0D	Select EF (under current DF)
0x0E	Select MF (root)
0x0F	RFU

*Note : P3 is a Length (Lc or Le)

6.3.12 Prefix and Suffix Codes

Parameter codes 06 (hexadecimal) and 07 represent prefix and suffix commands respectively. These are commands (function codes) that must execute before or after the specified function code. For example, on some smart cards, a `GET RESPONSE` must succeed a cryptographic function, or a `VERIFY` must precede a `READ BINARY` with secure messaging.

6.3.13 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 6-10 presents a summary of all descriptor codes.

6.3.14 Status Tuples

The purpose of the Status Tuples is to map a smart 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. 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. Tables 6-4 through 6-6 describe the status tuple construction and status condition codes.

Table 6-4: Status Tuples

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

SW	
SW1	SW2

Table 6-5: 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

6.3.15 Next CCC Description

This field, if included, is used to point to another CCC container. The values in this next CCC container will override values in the current CCC or define new values and fields. The Next CCC field contains an `ApplicationsCardURL` structure, with minimum length of 20 bytes.

6.4 CCC Formal Grammar Definition

Using a modified Backus-Naur notation, a 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
)
```

6.4.1 Grammar Rules

A description of the symbols follows:

Symbol	Meaning
:	is composed of
[ ]	optional element
( )	includes or included in
,	separates elements
…	element repeats unspecified number of times
{ }	choose one from list
< >	element value must be given at execution time
|	or, indicates choice of possibilities for element value
+	element is combined with preceding element
//	remainder of line contains comments

In general, the word immediately preceding a colon is the name of the element, while the word to the right of the colon is the name of an element value that may be expected. A description of the element values is given as follows:

Element	Meaning
Function_code	value from function code table, always required when other elements are present
Class	value for the APDU CLA byte, when entered this is a constant
Instruction	value for APDU INS byte, when entered this is a constant
Extension	(see discussion about extended function code)
P_constant	value for the APDU P1 or P2 byte, when entered this is a constant
Code	code for parameter definition, the code must be in the descriptor table
Length	length of data element, when entered this is a constant
Data_type	code for the composition of the APDU Data field, must be in the descriptor table
Qual	Qualifier for CLA; only possible value is 0xFE to indicate command is not available

Note that all elements except `function_code` are essentially optional in a `command_unit`. The square brackets [ ] are used to emphasize that the enclosed optional elements can only be present if the preceding element is present.

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

- 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.
- The sequence of tuples is important and must be presented in the order defined by the formal grammar.
- Each command unit consists of a required function code and optional APDU elements.
- When present, the CLA element may have a constant value (and/or one qualifier code equal to 0xFE, which indicates the command is not available on the smart card).
- When present, the INS element must have a constant value.
- When present, the P1 element may optionally have a constant value and/or one/multiple definition code.
- When present, the P2 element may optionally have a constant value and/or one/multiple definition code.
- When present, the P3 element may have a constant value; P3 always refers to the length of the DATA element in the Command APDU or the length of the expected DATA element in the Response APDU (respectively Lc or Le).
- 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 [CCPG] card:

Table 6-6: Default vs. Schlumberger DF APDU

Select Dedicated File (DF)						
Card Type	**CLA**	**INS**	**P1**	**P2**	**P3**	**Data**
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 Table 6-7, where the C-Byte and V-Byte are built from the parameter, function, and descriptor codes given in the Table 6-3 and Table 6-10.

Table 6-7: 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	2 byte FID	8121

Table 6-7 shows the complete tuple sequence to define the `SELECT DF` command for the Cryptoflex[CCPG] 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 VCEI. The tuples required to define this `SELECT DF` command for the Cryptoflex card would be:

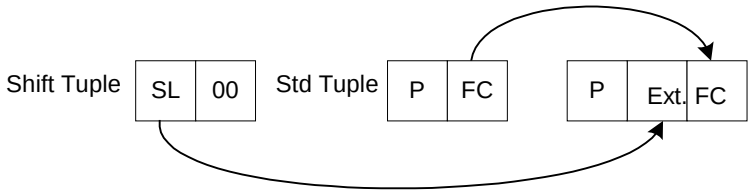
Table 6-8: Derived Select DF Tuple

#	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

6.4.2 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 smart 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. A diagram illustrating the mechanics is given in Figure 6-3.



SL: Shift Level; FC: Function Code; P:Parameter; Ext.: Extension

Figure 6-3: Shift Tuple Sequence (SL: shift level)

As an example of using the shift tuple, consider the following sequence of tuples in Table 6-9:

Table 6-9: Example of Extended Function Code

#	C-Byte			V-Byte	Description			Tuple
	S	P	FC		Function, Parm	V/D	Value/Descriptor	
1	0	7	8	1C	Verify, Suffix	V	"1C"	7817
2	0	1	0	00	Shift up 1	V	"00"	1000
3	0	1	7	00	Get Response, CLA	V	"00"	1700
4	0	2	7	C0	Get Response, INS	V	"C0"	27C0
5	0	5	7	12	Get Response, P3	V	"12"	5712
6	1	5	7	15	Get Response, P3	D	15	D715
7	1	0	7	FD	Get Response, Data	D	FD	87FD
8	1	0	7	38	Get Response, Data	D	38	8738
9	1	0	7	2F	Get Response, Data	D	2F	872F

The first two tuples have function code 08h indicating a `VERIFY` command, and give the value for the Data and Suffix parameters. In this case the suffix is a `GET RESPONSE` with an extended function code. The third tuple is used to set the current shift state. The function codes in the following tuples are logically or'ed with the shift tuple key, which is the C-byte of the shift tuple ("10" in the previous table) to create the extended function code 17h (result of 10h logically or'ed with 07h). This extended function code is then used to identify a new command that completely specifies a `GET RESPONSE` using the constant value "12" for P3. In this way a card and command-specific length can be specified for the `GET RESPONSE`.

Table 6-10: Descriptor Codes

Code	Meaning	Comments
0x00–0x0F	Execute Function Code	
0x11	Challenge	*Card Random Number: a designated number of random byte values generated by the smart card.*
0x12	Algorithm Identifier	
0x13	RFU	
0x14	RFU	
0x15	Length	
0x16	MSB of Offset	*The most significant byte of the file offset in bytes.*
0x17	LSB of Offset	*The least significant byte of the file offset in bytes.*
0x18	Key Level	*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.*
0x19	Key Identifier	*Key number*
0x1A	CHV Level	
0x1B	CHV Identifier	*CHV number on smart card*
0x1C	AID	*Application Identifier*
0x1D	EF	*The File ID of an Elementary File*
0x1E	SID	*The Security Identifier value used by Microsoft Windows.™*
0x1F	Parameter is not used	
0x20	RFU	*Reserved for future use*
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	
0x24	AES-ECB	*AES algorithm, mode ECB*
0x25	AES-CBC	*AES algorithm, mode CBC*
0x26	DES	*DES algorithm*
0x27	DES3_16	*Triple DES algorithm*
0x28	Plain Text	*un-encrypted ANSI text*
0x29	RFU	
0x2A	Pad Data with 0s	*The Data is padded at the end with low values to length of P3*
0x2B	PIN	*PIN value*
0x2C	2-byte Key File Identifier	*The 2-byte File Identifier of the file of the key being referenced.*
0x2D	PIN Type	*Pin Type*
0x2E	RFU	
0x2F	8 Byte Random Number	
0x30	Length + 6	*Length of data plus 6 bytes*
0x31	Length + 3	*Length of data plus 3 bytes*
0x32	Max Buffer Size	*Maximum buffer size in preceding data bytes*
0x33	n (modulus length)	*Used in the RSA algorithm*
0x34	Message	*Plain text message to be encrypted*
0x35	4 Byte Word	*Length or offset is given in words (one word = 4 bytes)*
0x36	Pad Data with FF	*Data padded at end with high values*

Code	Meaning	Comments
0x37	Length = SW2 (with SW1 =61)	*Length = low nibble of SW1-SW2 (61nn) from last response*
0x38	RFU	
0x39	RSA = 512	*RSA 512 bit algorithm using Chinese Reminder Theorem*
0x3A	RSA = 768	*RSA 768 bit algorithm using Chinese Reminder Theorem*
0x3B	RSA = 1024	*RSA 1024 bit algorithm using Chinese Reminder Theorem*
0x3C	Pad = FF at beginning	*Padding (FF) put at the beginning for the length of key to be 128 bytes*
0x3D	ANSI X9.31 Padding	
0x3E	Pad = 00(8)	*Data padded at the end with low values to the 8-byte boundary (ISO 9797.2 paragraph 5.1 method 1).*
0x3F	Pad = FF(128)	*Data padded at end with high values to total length of 128 bytes (PKCS#1)*
0x40	Pad = FF(Front)	
0x41	MD5 Header	
0x42	LSN Key Encoding	*Concatenate least significant nibbles of key. For example 8 byte key can be represented by 4 bytes.*
0x43	Terminal Random Number	*A designated number of random byte values generated on the terminal by the BSI.*
0x44	Key Level + Key	*Most significant bit is global/local flag*
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	RFU	
0x49	Block Length	
0x4A	TLV Format	
0x4B	Operation Mode	*Cryptographic operation modes*
0x4C	LOUD	*Length of useful data: the number of bytes in the data transmitted, without counting any padding or added bytes.*
0x4D	RFU	
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 DES3 encryption for a 16-byte key.*
0x4F	RFU	
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	Select child DF of current DF	*Descriptor code used to describe variation of the ISO Select file command for P1 (Function code "0x0C") See section 5.1.1.4*
0x53	Length + 8	*The number of bytes of data to be read or written plus 8.*
0x54	Select EF of current DF	*Descriptor code used to describe variation of the ISO Select file command for P1 (Function code "0x0C") See section 5.1.1.4*
0x55	Select parent DF of current DF	*Descriptor code used to describe variation of the ISO Select file command for P1 (Function code "0x0C") See section 5.1.1.4*
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 smart card, and the data bytes to be written.*

Code	Meaning	Comments
0x57	TLV Response 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 smart 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 smart card, and the data bytes read.*
0x5A	DES3_16-ECB	*Triple DES algorithm, 16 bytes key, ECB mode,*
0x5B	DES3_16-CBC	*Triple DES algorithm, 16 bytes key, CBC mode,*
0x5C	DES-ECB	*DES algorithm, mode ECB*
0x5D	DES-CBC	*DES algorithm, mode CBC*
0x5E	RSA = 2048	*RSA 2048 bit algorithm using Chinese Reminder Theorem*
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 smart 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 smart card, or a decrypted padded hash for a digital signature sent to the smart 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 smart card or response data received from the smart 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.*
0x70-0x99	RFU	
0xA0-0xDF	Implementation Dependent	

Code	Meaning	Comments
0xE0	Put Data Bytes	*Place Data Bytes (En) in data stream output to smart card*
0xE1-0xEF	En N Data Bytes	*En: Next n bytes are Data Bytes*
0xF0-0xFC	Reserved	
0xFD	Interpret Response	*Following descriptor bytes are used to interpret response*
0xFE	Command not available	*Command is not available on smart card*
0xFF	User Input Required	*Parameter value must be supplied by use/program*

7. Container Selection and Discovery

The GSC-IS architecture isolates client applications from the differences between virtual machine and file system cards. Virtual machine cards use AID to identify containers and file system cards use File IDs (FID) to identify files; containers and files fall under the category of "objects." An applet on a virtual machine card may manage one or more containers, whereas a directory on a file system may contain one or more files. Client applications must be able to locate the appropriate container or file, regardless of which applet or directory is required. These differences are abstracted by defining `ApplicationsCardURL` and Universal AID structures that are common to both virtual machine and file system cards. In this context the terms "container" and "file" and "object" are synonymous. The term "container" will be used preferentially throughout this section.

7.1 AID Abstraction: The Universal AID

Client applications use Universal AIDs to select generic containers and 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 selecting cryptographic service modules, Universal AIDs are constructed by concatenating the GSC RID value with the File ID of the desired cryptographic key file (symmetric or asymmetric). For example, the Universal AID of the Card Capabilities Container on a card that conforms to the GSC-IS Data Model (Appendix C) would be 0xA000000116DB00.

7.2 The CCC Universal AID and CCC Applet

As one of its first functions, an SPS must read the CCC from the smart card. The retrieval process for the CCC is detailed in Chapter 6. For virtual machine cards, the CCC shall be the default container of an applet whose Universal AID is known by client applications (RID+"DB00"). Therefore, selecting this applet makes the CCC the default selected object available to read.

7.3 The Applications CardURL

Before accessing a container on a smart card, client applications need a method for identifying the applet and directory information associated with the container. Therefore, all GSC conformant smart cards shall provide, in the CCC, an `ApplicationsCardURL` structure for each container present on the card. The `ApplicationsCardURL` structure is used to uniquely reference a container on a smart card by including its Universal AID and its associated applet or directory information. This structure also provides a mechanism for client applications to determine the ACRs and PIN and key labels associated with the given container.

`ApplicationsCardURL` structures are stored in the CCC as outlined in Chapter 6. For VM cards, the `pinID, AccessKeyInfo`, and `keyCryptoAlgorithm` fields must be present but are not applicable. The following ASN.1 sequence describes the structure of the `ApplicationsCardURL`:

```
ApplicationsCardURL ::=       SEQUENCE {
      Rid                     OCTET STRING SIZE(5),
      CardApplicationType,
      ObjectID                BIT STRING SIZE(16),
      ApplicationID           BIT STRING SIZE(16),
      AccessProfile,
      pinID                   BIT STRING SIZE(8),
      AccessKeyInfo,
      keyCryptoAlgorithm
}
```

```
CardApplicationType ::=       CHOICE {
      genericContainer        [0]   BIT STRING SIZE(8) : ‘0x01’,
      ski                     [1]   BIT STRING SIZE(8) : ‘0x02’,
      pki                     [2]   BIT STRING SIZE(8) : ‘0x04’
}

ObjectID ::=                  CHOICE {
      -- GSC data model definitions
      generalInfo             [0]   BIT STRING SIZE(16) : ‘0x2000’,
      proPersonalInfo         [1]   BIT STRING SIZE(16) : ‘0x2100’,
      accessControl           [2]   BIT STRING SIZE(16) : ‘0x3000’,
      login                   [3]   BIT STRING SIZE(16) : ‘0x4000’,
      cardInfo                [4]   BIT STRING SIZE(16) : ‘0x5000’,
      biometrics              [5]   BIT STRING SIZE(16) : ‘0x6000’,
      digitalSigCert          [6]   BIT STRING SIZE(16) : ‘0x7000’,
      -- CAC data model definitions
      personInstance          [7]   BIT STRING SIZE(16) : ‘0x0200’,
      benefitsInfo            [8]   BIT STRING SIZE(16) : ‘0x0202’,
      otherBenefits           [9]   BIT STRING SIZE(16) : ‘0x0203’,
      personnel               [10]  BIT STRING SIZE(16) : ‘0x0201’,
      loginInfo               [11]  BIT STRING SIZE(16) : ‘0x0300’,
      pkiCert                 [12]  BIT STRING SIZE(16) : ‘0x02FE’
      -- Common definitions
      SEIWG                   [13]  BIT STRING SIZE(16) : ‘0x0007’
}

AccessProfile ::= ACRList

ACRList ::=                   CHOICE {
      GCACRList,
      CryptoACRList
}

CryptoACRlist ::=             SEQUENCE  {
      listID                        BIT STRING SIZE(8) : ‘0x01’,
      getChallengeACRID             BIT STRING SIZE(8),
      internalAuthenticateACRID     BIT STRING SIZE(8),
      pkiComputeACRID               BIT STRING SIZE(8),
      readTagListACRID              BIT STRING SIZE(8),
      updatevalueACRID              BIT STRING SIZE(8),
      readvalueACRID                BIT STRING SIZE(8),
      createACRID                   BIT STRING SIZE(8),
      deleteACRID                   BIT STRING SIZE(8)
}

GCACRlist ::=                 SEQUENCE  {
      listID                        BIT STRING SIZE(8) : ‘0x02’,
      readTagListACRID              BIT STRING SIZE(8),
      updatevalueACRID              BIT STRING SIZE(8),
      readvalueACRID                BIT STRING SIZE(8),
      createACRID                   BIT STRING SIZE(8),
      deleteACRID                   BIT STRING SIZE(8)
}


AccessKeyInfo ::=          SEQUENCE  {
```

```
      keyFileID               BIT STRING SIZE(16),
      keyNumber               BIT STRING SIZE(8),
      keyCryptoAlgorithm
}

keyCryptoAlgorithm         CHOICE {
         DES3-16-ECB          [0] BIT STRING SIZE(8) : ‘0x00’,
         DES3-16-CBC          [1] BIT STRING SIZE(8) : ‘0x01’,
         DES-ECB              [2] BIT STRING SIZE(8) : ‘0x02’,
         DES-CBC              [3] BIT STRING SIZE(8) : ‘0x03’,
         RSA512               [4] BIT STRING SIZE(8) : ‘0x04’,
         RSA768               [5] BIT STRING SIZE(8) : ‘0x05’,
         RSA1024              [6] BIT STRING SIZE(8) : ‘0x06’,
         RSA2048              [7] BIT STRING SIZE(8) : ‘0x07’,
         AES128-ECB           [8] BIT STRING SIZE(8) : ‘0x08’,
         AES128-CBC           [9] BIT STRING SIZE(8) : ‘0x09’,
         AES192-ECB          [10] BIT STRING SIZE(8) : ‘0x0A’,
         AES192-CBC          [11] BIT STRING SIZE(8) : ‘0x0B’,
         AES256-ECB          [12] BIT STRING SIZE(8) : ‘0x0C’,
         AES256-CBC          [13] BIT STRING SIZE(8) : ‘0x0D’
}
```

7.4 Using the Applications CardURL Structure for Container Selection

The Universal AIDs associated with each data model are published in the appendices of this specification. When a client application attempts to first access a container, it will need to retrieve the `ApplicationsCardURL` structure that corresponds to that container's Universal AID, and use the information contained therein to access the container. This is done differently for file system and VM smart cards. The RID field contains the registered identifier [ISO5] data model.

7.5 File System Cards: Selecting Containers

The `ObjectID` field in the `ApplicationsCardURL` structure contains the two-byte File ID of the desired container. In the case of file system cards, the `ApplicationID` field will be the two-byte File ID of either the Master File or the Directory file within the Master File.

7.6 VM Cards: Selecting Containers and Applets

For VM cards, selecting the container is a two-part process. First, one retrieves the File ID for the desired container from the `ObjectID` field (as with file system cards). Secondly, one retrieves the AID of the applet that manages the container; that applet's AID is found in the `ApplicationID` field.

7.7 Using the Applications CardURL Structure for Identifying Access Control Rules

Identifying the access control rules associated with a specific container is straightforward after the container's associated `ApplicationsCardURL` structure is retrieved. The value of the `AccessProfile` field determines whether the following structure is a generic container ACR list (`GCACRlist`) or a cryptographic service modules ACR list (`CryptoACRlist`). Note that different access control rules can be associated with reading tags versus reading values.

THIS PAGE INTENTIONALLY LEFT BLANK.

8. Data Model

8.1 Data Model Overview

Data Models define a set of containers (files) and associated data elements in TLV format. The only mandatory containers are the CCC and Access Control File or SEIWG file. With the exception of the CCC and Access Control File, a GSC-IS conformant card may implement all, some, or none of the other containers associated with a Data Model. However, if the smart card uses any of the data elements defined in Data Model then it must use the container and TLV format specified by that Data Model for that element.

The SEIWG [SEIW] string is defined as the minimum interoperability mechanism for card holder authentication. The SEIWG strings and files are therefore mandatory for both contact and contactless GSC cards.

This specification defines two Data Models. The GSC Data Model was developed for version 1.0 of the GSC-IS (see Appendix C). The GSC 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).

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 0x01 if the card is using the GSC-IS Data Model defined in Appendix C, or a 0x02 if the card conforms to the CAC Data Model in Appendix D. Error Detection Codes (EDC) are only mandated for the GSC-IS Data Model.

8.2 Internal Tag-Length-Value Format

All container data elements are stored in SIMPLE-TLV format. Each SIMPLE-TLV data object shall consist of a tag field, a length field and an optional value field. For VM cards implementing the VM card edge interface, the SIMPLE-TLV format is split into a T-Buffer and V-Buffer. (See description in Section 8.4)

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 values "0x00" and "0xFF" are invalid for tag fields. 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 and have different meanings. 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 (LSB) - most significant byte (MSB) order, which encode an integer L with a value from 0 to 65,535.

If L is not zero, then the value field V shall consist of L consecutive bytes. If L is zero 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.

8.3 Structure and Length Values for Cards Requiring the File System Card Edge

The file system card edge requires containers to be implemented as a single file, i.e., one file comprises the container. The first TLV record of the container may optionally contain the length of the occupied space in the container as follows:

Container Byte 0: Tag = 0xEE

Container Byte 1: Length = 0x02

Container Byte 2: Least significant byte of length of occupied space

Container Byte 3: Most significant byte of length of occupied space

Container Byte 4: Next tag value

8.4 Structure and Length Values for Cards Requiring the Virtual Machine Card Edge

The virtual machine card edge is designed to interact with containers that are split into two buffers: the T-Buffer, for storing the tag and associated tag lengths, and the V-Buffer, for storing the values. The first two bytes of each buffer contain the length of the occupied space in the buffer in LSB-MSB format.

8.4.1 T-Buffer

The T-Buffer is constructed according to the TLV format (Section 8.2):

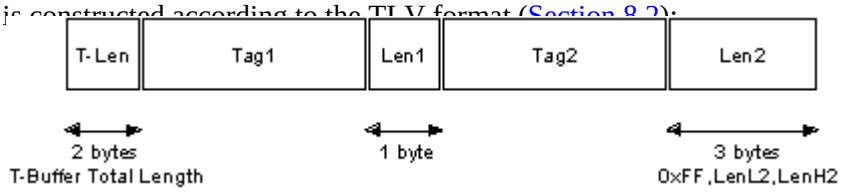


Figure 8-1: T-Buffer Format

8.4.1 V-Buffer

The V-Buffer is constructed as follows according to the TLV format:

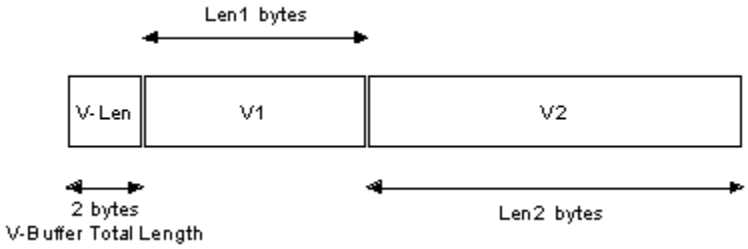


Figure 8-2: V-Buffer Format

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

[FIPS1] National Institute of Standards and Technology, Federal Information Processing Standard (FIPS) 140-2: Security Requirements for Cryptographic Modules, Decemeber 3, 2002

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

[ISO3] ISO/IEC 7816-3 1995(E): Electronic Signals and Transmission Protocols, http://www.iso.ch.

[ISO4] ISO/IEC 7816-4 1995(E): Interindustry Commands for Interchange

[ISO5] ISO/IEC 7816-5 1994-1996 (Amendment 1): Numbering system and registration procedure for application identifiers.

[ISO8] ISO/IEC 7816-8 1995(E): Interindustry Commands for a Cryptographic Toolbox

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

[1444] ISO/IEC 14443, Contactless integrated circuit(s) cards – Proximity cards - Parts 1 - 4

THIS PAGE INTENTIONALLY LEFT BLANK.

Appendix B—Informative References

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

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

[CCCG] GSC-IS CCC Grammar Tutorial, Jackson, Harry, 2001, http://smartcard.nist.gov/cccgrammartutorial.pdf

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

[CCPG] Cryptoflex Cards Programmer's Guide, www.cryptoflex.com

[SEIW] Physical Access Interoperability Working Group (PAIWG) *Technical Implementation Guidance, Final Draft v1.0, Smart Card Enabled Physical Access Control Systems* (dated 2 July 2003).

THIS PAGE INTENTIONALLY LEFT BLANK.

Appendix C—GSC Data Model

The RID for the GSC Data Model is 0xA000000116. The registered Data Model number is 0x01, and the Data Model version number is 0x01.

Note that the Access Control File and the SEIWG[SEIW] field contained therein are mandatory for this data model.

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

General Information File / Buffer		EF 2000	Always Read
Data Element (TLV)	**Tag**	**Type**	**Max. Bytes**
SSN Designator	1D	Variable	6
Error Detection Code	FE	LRC	1

Protected Personal Information File / Buffer		EF 2100	Verify CHV
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 (Note: File mandatory for contact cards)		EF 3000	Always Read
Data Element (TLV)	**Tag**	**Type**	**Max. Bytes**
SEIWG Data (Note: Field mandatory for contact cards)	30	Fixed	40*
PIN	31	Fixed Numeric	10
Domain (Facility / System ID)	32	Variable	8
Error Detection Code	FE	LRC	1

*The SEIWG data format is defined in [SEIW].

Login Information File / Buffer		EF 4000	Verify CHV
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

Card Information File / Buffer		EF 5000	Always Read
Data Element (TLV)	**Tag**	**Type**	**Max. Bytes**
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	Verify CHV
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

THIS PAGE INTENTIONALLY LEFT BLANK.

Appendix D—DoD Common Access Card (CAC) Data Model

D.1 CAC Data Model Specific

The RID for the all the files except the CCC in the CAC Data Model is 0xA000000079. The registered Data Model number is 0x02, and the Data Model version number is 0x01. The CCC RID is 0xA000000116.

The CAC containers are stored in SIMPLE-TLV format as per Chapter 8.

File/Buffer Description	FID	Maximum Length (Bytes)	Read Access Condition
Capability	DB00		Always Read
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
SEIWG	0007	41	Always Read

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

SEIWG File / Buffer (Note: File Mandatory for Contact Cards)		EF 0007	Always Read
Data Element (TLV)	**Tag**	**Type**	**Max. Bytes**
SEIWG Data	30	Fixed	40*
Error Detection Code	FE	LRC	1

*The SEIWG data format is defined in [SEIW].

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
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

THIS PAGE INTENTIONALLY LEFT BLANK.

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 23 C functions derived from the pseudo IDL specification (Chapter 4). The return codes for the functions are as defined in Section 4.4. The C binding 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    unsigned 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). This number includes the additional byte for the NULL terminator in the case where actual text is expected (e.g. Reader Name). 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. The first time to get the required buffer size (discovery call), and the second time to execute the function with the correct buffer size (execution call). However, only one call is possible if the client application is able to estimate the required buffer size. In that last case, the call is an execution call.

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 and 2 in Section E.3 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 3 in the Section E.3 for more information.

E.2.1 Variable Length String Data

Ten BSI function calls accept and/or return variable-length string data identified in Table E-1.

Table E-1: BSI functions using the discovery method

BSI function	Discovery buffer	Discovery length
gscBsiUtilGetVersion ()	*uszVersion	*unVersionLen
gscBsiUtilGetCardProperties ()	*uszCCCUniqueID	*unCCCUniqueIDLen
gscBsiUtilGetReaderList ()	*uszReaderList	*unReaderListLen
gscBsiUtilPassthru () (See Note in Section E.3.9)	*uszCardResponse	*unCardResponseLen
gscBsiGcReadTagList ()	*TagArray	*unNbTags
gscBsiGcReadValue ()	*uszValue	*unValueLen
gscBsiGetChallenge ()	*uszChallenge	*unChallengeLen
gscBsiSkiInternalAuthenticate ()	*uszCryptogram	*unCryptogramLen
gscBsiPkiCompute ()	*uszResult	*unResultLen
gscBsiPkiGetCertificate ()	*uszCertificate	*unCertificateLen

Each of these functions is invoked in the discovery mode by passing in a NULL value for the discovery buffer parameter. With the exception of **`gscBsiGcReadTagList ()`**, each of these returns (Discovery call) the size in bytes (including the NULL Terminator) of the buffer needed to store the return variable-length string data. The lone exception, **`gscBsiGcReadTagList ()`**, returns the number of tags in the tag array, so that the size of the array buffer needed is given by "*unNbTags * size of (GCtag)".

E.3 Discovery Mechanisms Code Samples

Following are three examples in C illustrating the discovery mechanism.

The three examples make the following assumptions:

```
- Application defined return codes SUCCESS & FAILURE
- ERROR_RETURN reports error and returns FAILURE
- Parameters AID and AID length are given
- PROCESS_READ_CERTIFICATE processes the read of the certificate
```

Example 1

```
{

// Discovers the correct size for the certificate buffer, allocates memory
and executes.

unsigned char *   pCert;           //Discovery buffer
unsigned long     unCertLen = 0;   //Discovery length
```

```
long              iRet;             //return code (“unsigned long” in the
Spec)

//First call : Discovery call
iRet = gscBsiPkiReadCertificate (hCard, usAID, unAIDLen, NULL, &unCertLen);

if (iRet != BSI_OK)
      ERROR_RETURN ("gscBsiPkiReadCertificate-discovery call", iRet);

if (unCertLen == 0)
      ERROR_RETURN ("Unexpected BSI_OK with unCertLen == 0", unCertLen);

//Memory allocation of the buffer with the returned length from first call
pCert = (unsigned char *) malloc (unCertLen * sizeof(unsigned char));

if (pCert==NULL)
      ERROR_RETURN ("Unable to allocate memory", unCertLen);
else

//Second call : Execution call
iRet = gscBsiPkiReadCertificate (hCard, usAID, unAIDLen, pCert, &unCertLen);
if (iRet != BSI_OK)
      {
      free (pCert);     // avoid memory leak!
      ERROR_RETURN ("gscBsiPkiReadCertificate-results call", iRet)
      }
else

PROCESS_READ_CERTIFICATE{…}

free (pCert);           // avoid memory leak!
return (SUCCESS);
}
```

Example 2

```
{

// Try default buffer first, if buffer is large enough normal execution
occurs, or if buffer is too small reacts by discovering the length and
executes.

unsigned char     usBuffer [ESTIMATED_CERT_SIZE];
unsigned char *   pCert  = usBuffer;                  //Discovery buffer
unsigned long     unCertLen = sizeof (usBuffer);      //Discovery length

long              iRet; //return code (“unsigned long” in the Spec)


//First call : Discovery call, or Execution call if buffer large enough
iRet = gscBsiPkiReadCertificate (hCard, usAID, unAIDLen, pCert, &unCertLen);

if (iRet==BSI_INSUFFICIENT_BUFFER)
      {
      pCert = (unsigned char *) malloc(unCertLen * sizeof(unsigned char));

      if (pCert==NULL)
            ERROR_RETURN ("Unable to allocate memory", unCertLen);

      //Second call : Execution call
      iRet = gscBsiPkiReadCertificate (hCard, usAID, unAIDLen, pCert,
&unCertLen);

      if (iRet != BSI_OK)
            free (pCert);     // avoid memory leak!
      }

if (iRet != BSI_OK)     // Works for either 1st or 2nd call!
      ERROR_RETURN ("gscBsiPkiReadCertificate", iRet);

PROCESS_READ_CERTIFICATE {…}

if (unCertLen > ESTIMATED_CERT_SIZE)
      free (pCert);           // avoid memory leak!

return (SUCCESS);

}
```

Example 3

```
{

// Use a buffer so large that discovery is never necessary.

unsigned char     usBuffer [REALLY_BIG_BUFFER];
unsigned char     *pCert            = usBuffer;       //Discovery buffer
unsigned long     unCertLen   = sizeof (usBuffer);    //Discovery length
long        iRet;             //return code (“unsigned long” in the Spec)

//First call: Execution call
iRet = gscBsiPkiReadCertificate (hCard, usAID, unAIDLen, pCert, &unCertLen);
if (iRet != BSI_OK)
      ERROR_RETURN ("gscBsiPkiReadCertificate", iRet);

PROCESS_READ_CERTIFICATE

return (SUCCESS);

}
```

E.4 Smart Card Utility Provider Module Interface Definition

E.4.1 gscBsiUtilAcquireContext()

Purpose: This function shall establish a session with a target container on 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 from **`gscBsiGetChallenge()`**. In cases where the card acceptance device authenticates the smart card, this function returns a `BSI_TERMINAL_AUTH` return code and the cryptogram is ignored.

For ACRs that require chained authentication such as `BSI_ACR_PIN_AND_XAUTH`, the calling application passes in the required authenticators in multiple `BSIAuthenticator` structures. In this example the calling application passes a PIN and the appropriate External Authentication cryptogram in two `BSIAuthenticator` structures. The client application must set the `unAccessMethodType` field of each `BSIAuthenticator` structure to match the type of authenticator contained in the structure. To satisfy an ACR of `BSI_ACR_PIN_AND_XAUTH`, the application would construct a sequence of two `BSIAuthenticators`: one containing a PIN and one containing an External Authentication cryptogram. The `BSIAuthenticator` structure containing the PIN would have an `unAccessMethodType` of `BSI_AM_PIN`, and the `BSIAuthenticator` structure containing the External Authentication cryptogram would have an `unAccessMethodType` of `BSI_AM_XAUTH.`

Prototype:

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

Parameters:

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

`uszAID`: Target container AID value. The parameter shall be in ASCII hexadecimal format.

`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 required list of authenticators 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 long        unAccessMethodType;
   unsigned long        unKeyIDOrReference;
   unsigned char        uszAuthValue
                        [BSI_AUTHENTICATOR_MAX_LEN];
   unsigned long        unAuthValueLen;
};
```

Variables associated with the `BSIAuthenticator` structure:

`unAccessMethodType:` Access Method Type (see Table 3-1 in Section 3.1).

`unKeyIDOrReference:` Key identifier or reference of the authenticator. This is used to distinguish between multiple authenticators with the same Access Method Type.

`uszAuthValue:` Authenticator, can be an external authentication cryptogram or PIN. If the authenticator value is NULL, then BSI is in charge of gathering authentication information and authenticating to the card.

`unAuthValueLen:` Authenticator value length in bytes.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_ACR_NOT_AVAILABLE
BSI_BAD_AUTH
BSI_CARD_REMOVED
BSI_SC_LOCKED
BSI_PIN_BLOCKED
BSI_TERMINAL_AUTH
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()**. The reader name string shall be stored as ASCII encoding String. (See Section 4.2)

unReaderNameLen: Length of the reader name in bytes.

Return Codes:

```
BSI_OK
BSI_BAD_PARAM
BSI_UNKNOWN_READER
BSI_CARD_ABSENT
BSI_TIMEOUT_ERROR
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 gscBsiUtilBeginTransaction()

Purpose: Starts an exclusive transaction with the smart card referenced by hCard. When the transaction starts, all other applications are precluded from accessing the smart card while the transaction is in progress. Two types of calls can be made with that function: a blocking transaction call and a non-blocking transaction call. A boolean type parameter identify which mode is called. In the non-blocking mode, the call will return immediately if another client has an active transaction lock. The returned error code will be `BSI_SC_LOCKED`. In the blocking mode, the call will wait indefinitely for any active transaction locks to be released. A transaction must be completed by a call to **`gscBsiUtilEndTransaction().`**

For single-threaded BSI implementations, it can be assumed that each application will be associated with a separate process. The same process that starts a transaction must also complete the transaction. For multi-threaded BSI implementations, it can be assumed that each application will be associated with a separate thread and/or process. The same thread that starts a transaction must also complete the transaction.

If this function is called by a thread that has already called **`gscBsiUtilBeginTransaction()`** but has not yet called **`gscBsiUtilEndTransaction()`** it will return the error `BSI_NOT_TRANSACTED`.

If the SPS (Service Provider Software) does not support transaction locking, it should return the error code `BSI_NO_SPSSERVICE` in response to a call to **`gscBsiUtilBeginTransaction()`**.

Prototype:

```
unsigned long gscBsiUtilBeginTransaction(
   IN unsigned long     hCard;
   IN boolean           blType;
);
```

Parameters:

hCard:	Card communication handle returned from **`gscBsiUtilConnect()`**
blType:	Boolean specifying the type of transaction call (blType set to “true” in blocking mode. blType set to “false” in non-blocking mode).

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_UNKNOWN_ERROR
BSI_SC_LOCKED
BSI_NOT_TRANSACTED
BSI_NO_SPSSERVICE
```

E.4.5 gscBsiUtilEndTransaction()

Purpose: Completes a previously started transaction, allowing other applications to resume interactions with the card.

If this function is called by a thread that has not yet called **`gscBsiUtilBeginTransaction()`** it will return the error `BSI_NOT_TRANSACTED`.

If the SPS (Service Provider Software) does not support transaction locking, it should return the error code `BSI_NO_SPSSERVICE` in response to a call to **`gscBsiUtilEndTransaction()`**.

Prototype:

```
unsigned long gscBsiUtilEndTransaction(
   IN unsigned long     hCard
);
```

Parameters:

`hCard:`	Card communication handle returned from **`gscBsiUtilConnect().`**

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_UNKNOWN_ERROR
BSI_NOT_TRANSACTED
BSI_NO_SPSSERVICE
```

E.4.6 gscBsiUtilGetVersion()

Purpose: Returns the BSI implementation version.

Prototype:
```
unsigned long gscBsiUtilGetVersion(
   INOUT unsigned char *   uszVersion,
   INOUT unsigned long *   punVersionLen
);
```

Parameters:

uszVersion:	The BSI and SPS version formatted as "major,minor,revision, build_number\0". The version text shall be stored as ASCII encoded String. (See Section 4.2)
punVersionLen:	Length of the version string.

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

Discovery Mode:

Parameters:

uszVersion:	Set to NULL.
punVersionLen:	Pointer to value containing the required buffer length to contain the version string, including a null terminator.

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

E.4.7 gscBsiUtilGetCardProperties()

Purpose: Retrieves ID and capability information for the card.

Prototype:
```
unsigned long gscBsiUtilGetCardProperties(
   IN UTILCardHandle       hCard,
   INOUT unsigned char *   uszCCCUniqueID,
   INOUT unsigned long *   punCCCUniqueIDLen,
   OUT unsigned long *     punCardCapability
);
```

Parameters:

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

uszCCCUniqueID: Buffer for the Card Capability Container ID, represented in ASCII Hexadecimal.

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

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

```
#define BSI_GCCDM         0x00000001
#define BSI_SKI           0x00000002
#define BSI_PKI           0x00000004
```

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

Discovery Mode:

Parameters:

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

uszCCCUniqueID: Set to NULL.

punCCCUniqueIDLen: Pointer to value containing the required buffer length for the CCC Unique ID string, including a null terminator.

punCardCapability: Can be set to NULL, unused in discovery.

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

E.4.8 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.4.9 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. The error text shall be stored as ASCII encoding String. (See Section 4.2)

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_NO_TEXT_AVAILABLE
BSI_UNKNOWN_ERROR
```

E.4.10 gscBsiUtilGetReaderList()

Purpose: Retrieves the list of available readers.

Prototype:
```
unsigned long gscBsiUtilGetReaderList(
   INOUT unsigned char * uszReaderList,
   INOUT unsigned long * punReaderListLen
);
```

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

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

Discovery Mode:

Parameters:

uszReaderList:	Set to NULL.
punReaderListLen:	Required buffer length for Reader list in bytes including all terminating '\0' characters.

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

E.4.11 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,
   INOUT unsigned char *   uszCardResponse,
   INOUT unsigned long *   punCardResponseLen
);
```

Parameters:

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

uszCardCommand: The APDU to be sent to the card. That parameter must be in ASCII hexadecimal format.

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. That parameter must be in ASCII hexadecimal format.

punCardResponseLen: 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_SC_LOCKED
BSI_UNKNOWN_ERROR
```

Discovery Mode (depending on usage):

Note: The discovery mechanism may cause the command APDU to be executed twice depending on the context of use.

The discovery mode is as follows:

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: Set to NULL.

punCardResponseLen: Length of the buffer required to contain the APDU response.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_SC_LOCKED
BSI_BAD_PARAM
BSI_CARD_REMOVED
BSI_UNKNOWN_ERROR
```

E.4.12 gscBsiUtilReleaseContext()

Purpose: Terminate a session with the target container on 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. The parameter shall be in ASCII hexadecimal format.
unAIDLen:	AID value length in bytes.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_CARD_REMOVED
BSI_SC_LOCKED
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. The parameter shall be in ASCII hexadecimal format.
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_SC_LOCKED
BSI_BAD_PARAM
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_NO_MORE_SPACE
BSI_TAG_EXISTS
BSI_IO_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. The parameter shall be in ASCII hexadecimal format.
unAIDLen:	AID value length in bytes.
ucTag:	Tag of data item to delete.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_SC_LOCKED
BSI_BAD_AID
BSI_BAD_PARAM
BSI_BAD_TAG
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_IO_ERROR
BSI_UNKNOWN_ERROR
```

E.5.3 gscBsiGcGetContainerProperties()

Purpose: Retrieves the properties of the specified 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 *     containerVersion
);
```

Parameters:

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

uszAID: Target container AID value. The parameter shall be in ASCII hexadecimal format.

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 3-2 (Section 3.1). The allowable ACRs for each function are listed in Table 3-3. `unKeyIDOrReference` contains the key identifier or reference for each access method contained in the ACR in order of appearance. `unAuthNb` is the number of access methods logically combined in the ACR. `ACRID` is RFU and must be NULL (0x00) in this version.

```
struct GCacr {
   BSIAcr              strctCreateACR;
   BSIAcr              strctDeleteACR;
   BSIAcr              strctReadTagListACR;
   BSIAcr              strctReadValueACR;
   BSIAcr              strctUpdateValueACR;
};

struct BSIAcr {
   unsigned long       unACRType;
   unsigned long       unKeyIDOrReference;
   unsigned long       unAuthNb;
   unsigned long       unACRID;
};
```

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;

}
```

containerVersion: 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_SC_LOCKED
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,
   INOUT Gctag *           TagArray,
   INOUT unsigned long *   punNbTags
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect().**
uszAID:	Target container AID value. The parameter shall be in ASCII hexadecimal format.
unAIDLen:	AID value length in bytes.
TagArray:	An array containing the list of tags for the selected container.
punNbTags:	Number of tags in TagArray.

Return Codes:

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

Discovery Mode:

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
uszAID:	Target container AID value.
unAIDLen:	AID value length in bytes.
TagArray:	Set to NULL.
punNbTags:	Number of tags which would be contained in a resulting TagArray.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_SC_LOCKED
BSI_BAD_AID
BSI_BAD_PARAM
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
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,
   INOUT unsigned char *    uszValue,
   INOUT unsigned long *    punValueLen
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect().**
uszAID:	Target container AID value. The parameter shall be in ASCII hexadecimal format.
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.
punValueLen:	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_SC_LOCKED
BSI_BAD_AID
BSI_BAD_PARAM
BSI_BAD_TAG
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_INSUFFICIENT_BUFFER
BSI_IO_ERROR
BSI_UNKNOWN_ERROR
```

Discovery Mode:

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:	Set to NULL.
punValueLen:	Size of the buffer required to hold the returned Value.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_SC_LOCKED
BSI_BAD_TAG
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_IO_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. The parameter shall be in ASCII hexadecimal format.
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_SC_LOCKED
BSI_BAD_AID
BSI_BAD_PARAM
BSI_BAD_TAG
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_NO_MORE_SPACE
BSI_IO_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,
   INOUT unsigned char *   uszChallenge,
   INOUT unsigned long *   punChallengeLen
);
```

Parameters:

`hCard:`	Card connection handle from **`gscBsiUtilConnect()`**.
`uszAID:`	Target container AID value. The parameter shall be in ASCII hexadecimal format.
`unAIDLen:`	AID value length in bytes.
`uszChallenge:`	Random challenge returned from the card.
`punChallengeLen:`	Length of random challenge in bytes.

Return Codes:

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

Discovery Mode:

Parameters:

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

uszChallenge:	Set to NULL.
punChallengeLen:	Length of buffer required to store returned random challenge in bytes.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_SC_LOCKED
BSI_BAD_AID
BSI_BAD_PARAM
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
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,
   INOUT unsigned char *   uszCryptogram,
   INOUT unsigned long *   punCryptogramLen
);
```

Parameters:

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

`uszAID:` SKI provider module AID value. The parameter shall be in ASCII hexadecimal format.

`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.

`punCryptogramLen:` Length of the cryptogram computed by the card in bytes.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_SC_LOCKED
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

Discovery Mode:

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:	Set to NULL.
punCryptogramLen:	Length of the buffer required to store the cryptogram computed by the card in bytes.

Return Codes:

BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_SC_LOCKED
BSI_BAD_ALGO_ID
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_TERMINAL_AUTH
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,
   INOUT unsigned char *   uszResult,
   INOUT unsigned long *   punResultLen
);
```

Parameters:

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

uszAID: PKI provider module AID value. The parameter shall be in ASCII hexadecimal format.

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.

punResultLen: Length of the signature buffer in bytes.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_SC_LOCKED
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
```

Discovery Mode:

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:`	Set to NULL.
`punResultLen:`	Length of the required signature buffer in bytes.

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_SC_LOCKED
BSI_BAD_AID
BSI_BAD_PARAM
BSI_BAD_ALGO_ID
BSI_CARD_REMOVED
BSI_ACCESS_DENIED
BSI_NO_CARDSERVICE
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,
   INOUT unsigned char *   uszCertificate,
   INOUT unsigned long *   punCertificateLen
);
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
uszAID:	PKI provider module AID value. The parameter shall be in ASCII hexadecimal format.
unAIDLen:	AID value length in bytes.
uszCertificate:	Buffer containing the certificate.
punCertificateLen:	Length of the certificate buffer in bytes.

Return Codes:
```
BSI_OK
BSI_BAD_HANDLE
BSI_SC_LOCKED
BSI_BAD_AID
BSI_BAD_PARAM
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_IO_ERROR
BSI_INSUFFICIENT_BUFFER
BSI_UNKNOWN_ERROR
```

Discovery Mode:

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
uszAID:	PKI provider module AID value.
unAIDLen:	AID value length in bytes.
uszCertificate:	Set to NULL.
punCertificateLen:	Length of the required certificate buffer in bytes.

Return Codes:
```
BSI_OK
BSI_BAD_HANDLE
BSI_SC_LOCKED
BSI_BAD_AID
BSI_BAD_PARAM
```

```
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_IO_ERROR
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 *  punKeyLen
);
```

Parameters:

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

uszAID: AID of the PKI provider. The parameter shall be in ASCII hexadecimal format.

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

strctCRYPTOacr: Structure indicating access control conditions for all operations. The `BSIAcr` structure is defined in Section E.5.3. The range of possible values for the members of this structure are defined in Table 3-2 (Section 3.1), and the allowable ACRs for each function in Table 3-4 (Section 3.2). `keyIDOrReference` contains the key identifier or reference for each access method contained in the ACR in order of appearance. `AuthNb` is the number of access methods logically combined in the ACR. ACRID is RFU and must be NULL (0x00) in this version. Note that the `strctReadValueACR` member maps to the **gscBsiPkiGetCertificate()** function.

```
struct CRYPTOacr {
   BSIAcr               strctGetChallengeACR;
   BSIAcr               strctInternalAuthenticateACR;
   BSIAcr               strctPkiComputeACR;
   BSIAcr               strctCreateACR;
   BSIAcr               strctDeleteACR;
   BSIAcr               strctReadTagListACR;
   BSIAcr               strctReadValueACR;
   BSIAcr               strctUpdateValueACR;
};
```

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

Return Codes:

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

THIS PAGE INTENTIONALLY LEFT BLANK.

Appendix F—Java Language Binding for BSI Services

This appendix defines the Java language binding, which comprises a set of classes and interfaces that provide the basic support for a Java implementation of a SPM as defined in the GSC-IS.

Similar to the pseudo IDL specification, the Java translation is logically grouped into three functional modules:

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

This appendix provides the required syntax and semantics of 23 methods that correspond to the 23 functions in Chapter 4. These methods are collectively defined in the `public interface` **`gov.gsc.interfaces.BSI`** (see Section F.1), and shall be implemented within a `public class` **`gov.gsc.classes.GSCBSI`**.

F.1 Interfaces and Classes

Based on the 23 methods described in this appendix (Section F.2 to Section F.4) the interface `gov.gsc.interfaces.BSI` is defined as follows:

```
public interface gov.gsc.interfaces.BSI
{
      public abstract void gscBsiUtilAcquireContext(
                                    int               hCard,
                                    String            AID,
                                    java.util.Vector  strctAuthenticator
      ) throws gov.gsc.classes.BSIException;

      public abstract int gscBsiUtilConnect(
                                    String      readerName
      ) throws gov.gsc.classes.BSIException;

      public abstract void gscBsiUtilDisconnect(
                                    int   hCard
      ) throws gov.gsc.classes.BSIException;

      public abstract void gscBsiUtilBeginTransaction(
                                    int         hCard
                                    boolean     blType
      )throws gov.gsc.classes.BSIException;

      public abstract void gscBsiUtilEndTransaction(
                                    int         hCard
      )throws gov.gsc.classes.BSIException;

      public abstract String gscBsiUtilGetVersion()
                      throws gov.gsc.classes.BSIException;

      public abstract CardProperties gscBsiUtilGetCardProperties(
                                    int   hCard
      ) throws gov.gsc.classes.BSIException;

      public abstract void gscBsiUtilGetCardStatus(
                                    int   hCard
      ) throws gov.gsc.classes.BSIException;

      public abstract String gscBsiUtilGetExtendedErrorText(
                                    int   hCard
      ) throws gov.gsc.classes.BSIException;

      public abstract java.util.Vector gscBsiUtilGetReaderList()
         throws gov.gsc.classes.BSIException;

      public abstract byte[] gscBsiUtilPassthru(
                                    int         hCard,
                                    byte[]      cardCommand
      ) throws gov.gsc.classes.BSIException;

      public abstract void gscBsiUtilReleaseContext(
                                    int               hCard,
                                    String      AID
```

```
) throws gov.gsc.classes.BSIException;

public abstract void gscBsiGcDataCreate(
                              int         hCard,
                              String      AID,
                              short       tag,
                              byte[]      value
) throws gov.gsc.classes.BSIException;

public abstract void gscBsiGcDataDelete(
                              int         hCard,
                              String      AID,
                              short       tag
) throws gov.gsc.classes.BSIException;

public abstract ContainerProperties gscBsiGcGetContainerProperties(
                              int         hCard,
                              String      AID
) throws gov.gsc.classes.BSIException;

public abstract short[] gscBsiGcReadTagList(
                              int         hCard,
                              String      AID
) throws gov.gsc.classes.BSIException;

public abstract byte[] gscBsiGcReadValue(
                              int         hCard,
                              String      AID,
                              short       tag
) throws gov.gsc.classes.BSIException;

public abstract void gscBsiGcUpdateValue(
                              int         hCard,
                              String      AID,
                              short       tag,
                              byte[]      value
) throws gov.gsc.classes.BSIException;

public abstract byte[] gscBsiGetChallenge(
                              int         hCard,
                              String      AID
) throws gov.gsc.classes.BSIException;

public abstract byte[] gscBsiSkiInternalAuthenticate(
                              int         hCard,
                              String      AID,
                              short       algoID,
                              byte[]      challenge
) throws gov.gsc.classes.BSIException;

public abstract byte[] gscBsiPkiCompute(
                              int         hCard,
                              String      AID,
                              short       algoID,
                              byte[]      message
) throws gov.gsc.classes.BSIException;
```

```
      public abstract byte[] gscBsiPkiGetCertificate(
                                    int         hCard,
                                    String      AID
      ) throws gov.gsc.classes.BSIException;

      public abstract CryptoProperties gscBsiGetCryptoProperties(
                                    int         hCard,
                                    String      AID
      ) throws gov.gsc.classes.BSIException;
}
```

F.1.1 - The Same `Class` **GSCBSI** Shall Also Implement The Following Interfaces:

- `gov.gsc.interfaces.BSIReturnCodes`
- `gov.gsc.interfaces.CryptographicAlgoID`
- `gov.gsc.interfaces.BSIAccessControlRules`
- `gov.gsc.interfaces.BSICardCapabilities`

F.1.1.1 - The interfaces are defined as follows:

```
public interface gov.gsc.interfaces.BSIReturnCodes
{
      public static final int BSI_OK                 = 0x00;

      public static final int BSI_ACCESS_DENIED      = 0x01;
      public static final int BSI_ACR_NOT_AVAILABLE  = 0x02;

      public static final int BSI_BAD_AID            = 0x03;

      public static final int BSI_BAD_ALGO_ID        = 0x04;
      public static final int BSI_BAD_AUTH           = 0x05;
      public static final int BSI_BAD_HANDLE         = 0x06;
      public static final int BSI_BAD_PARAM          = 0x07;
      public static final int BSI_BAD_TAG            = 0x08;
      public static final int BSI_CARD_ABSENT        = 0x09;
      public static final int BSI_CARD_REMOVED       = 0x0A;
      public static final int BSI_NO_SPSSERVICE      = 0x0B;
      public static final int BSI_IO_ERROR           = 0x0C;
      public static final int BSI_INSUFFICIENT_BUFFER = 0x0E;
      public static final int BSI_NO_CARDSERVICE     = 0x0F;
      public static final int BSI_NO_MORE_SPACE      = 0x10;
      public static final int BSI_PIN_LOCKED         = 0x11;
      //Note : 0x12 is RFU
      public static final int BSI_TAG_EXISTS         = 0x13;
      public static final int BSI_TIMEOUT_ERROR      = 0x14;
      public static final int BSI_TERMINAL_AUTH      = 0x15;
      public static final int BSI_NO_TEXT_AVAILABLE  = 0x16;
      public static final int BSI_UNKNOWN_ERROR      = 0x17;
      public static final int BSI_UNKNOWN_READER     = 0x18;
      public static final int BSI_SC_LOCKED          = 0x19;
      public static final int BSI_NOT_TRANSACTED     = 0x20;
```

```
}

public interface gov.gsc.interfaces.CryptographicAlgoID
{
      //Mandatory Cryptographic Algorithms (see Section 4.3)
      //Cryptographic algorithm computation on the private key,
      //Chinese Remainder Theory.
      public static final short RSA_NO_PAD = 0xA3;

      //DES3-ECB cryptographic algorithm with a double length
      //key-size of 16 bytes.
      public static final short BSI_DES3_ECB = 0x81;

      //DES3-CBC cryptographic algorithm with a double length
      //key-size of 16 bytes.
      public static final short BSI_DES3_CBC = 0x82;
}

public interface gov.gsc.interfaces.BSIAccessControlRules
{
      //BSI ACR Values as defined in the Table 3-1.
      public static final int BSI_ACR_ALWAYS          = 0x00;
      public static final int BSI_ACR_NEVER           = 0x01;
      public static final int BSI_ACR_XAUTH           = 0x02;
      public static final int BSI_ACR_XAUTH_OR_PIN    = 0x03;
      public static final int BSI_SECURE_CHANNEL_GP   = 0x04;
      public static final int BSI_ACR_PIN_ALWAYS      = 0x05;
      public static final int BSI_ACR_PIN             = 0x06;
      public static final int BSI_ACR_XAUTH_THEN_PIN  = 0x07;
      public static final int BSI_ACR_UPDATE_ONCE     = 0x08;
      public static final int BSI_ACR_PIN_THEN_XAUTH  = 0x09;
      //NOTE: 0x0A currently not used
      public static final int BSI_SECURE_CHANNEL_ISO  = 0x0B;
      public static final int BSI_XAUTH_AND_PIN       = 0x0C;
      //NOTE: RESERVED FOR FUTURE USED 0x0D-0xFF

}

public interface gov.gsc.interfaces.BSICardCapabilities
{
      public static final int BSI_GCCDM         = 0x00000001;
      public static final int BSI_SKI           = 0x00000002;
      public static final int BSI_PKI           = 0x00000004;
}
```

F.1.1.2 - All 23 methods throw a `BSIException` if an error occurred during execution. A `BSIException` shall be constructed using one of the eligible return code listed for every individual method.

The class `BSIException` is defined as follows:

```
public class gov.gsc.classes.BSIException
                                    extends java.lang.Exception
{
      //Methods inherited from class java.lang.Throwable
```

```
      //   fillInStackTrace, getLocalizedMessage, getMessage,
      //   printStackTrace, printStackTrace, printStackTrace,
      //   toString

      //Methods inherited from class java.lang.Object
      //   clone, equals, finalize, getClass, hashCode,
      //   notify, notifyAll, wait, wait, wait

      //All Implemented Interfaces:
      //     java.io.Serializable

      //FIELDS:
      protected int errorCode = 0;


      //CONSTRUCTORS:
      //Constructor specifying the error code value as
      //defined in the Table 4-1
      public BSIException(int error)
      {
         super();
         errorCode = error;
      }

      //Constructor specifying the error code value
      //and corresponding message as defined in the Table 4-1
      public BSIException(int error,
                          String msg)
      {
         super(msg);
         errorCode = error;
      }

      //ACCESSORS:
      //Gets error code
      //Returns: errorCode
      public int getErrorCode()
      { return errorCode; }
}
```

F.2 Smart Card Utility Provider Module Interface Definition

F.2.1 gscBsiUtilAcquireContext()

Purpose: This function shall establish a session with a target container on the smart card by submitting the appropriate Authenticator in the `BSIAuthenticator` structure. For ACRs requiring external authentication (`XAUTH`), the `authValue` field of the `BSIAuthenticator` structure must contain a cryptogram calculated by encrypting a random challenge from **`gscBsiGetChallenge()`**. In cases where the card acceptance device authenticates the smart card, this function returns a `BSI_TERMINAL_AUTH` return code and the cryptogram is ignored.

For ACRs that require chained authentication such as `BSI_ACR_PIN_AND_XAUTH`, the calling application passes in the required authenticators in multiple `BSIAuthenticator` structures. In this example the calling application passes a PIN and the appropriate External Authentication cryptogram in two `BSIAuthenticator` structures. The client application must set the `accessMethodType` field of each `BSIAuthenticator` structure to match the type of authenticator contained in the structure. To satisfy an ACR of `BSI_ACR_PIN_AND_XAUTH`, the application would construct a sequence of two `BSIAuthenticators`: one containing a PIN and one containing an External Authentication cryptogram. The `BSIAuthenticator` structure containing the PIN would have an `accessMethodType` of `BSI_AM_PIN`, and the `BSIAuthenticator` structure containing the External Authentication cryptogram would have an `accessMethodType` of `BSI_AM_XAUTH.`

Prototype:

```
public abstract void gscBsiUtilAcquireContext(
                     int                hCard,
                     String             AID,
                     java.util.Vector   strctAuthenticator
) throws gov.gsc.classes.BSIException;
```

Parameters:

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

`AID:` AID of the target service provider or container. The AID shall be stored as an ASCII hexadecimal string.

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

The `BSIAuthenticator` class is defined as follows:

```
public class gov.gsc.classes.BSIAuthenticator
{
   //FIELDS:
   protected int        accessMethodType;
```

```
    protected int        keyIDOrReference;
    protected byte[]     authValue;

    //CONSTRUCTORS:
    public BSIAuthenticator()
    {
         accessMethodType = 0;
         keyIDOrReference = 0;
         authValue = “”;
    }

    public BSIAuthenticator( int        amType,
                             int        keyIDOrRef,
                             byte[]     authVal )
    {
         accessMethodType = amType;
         keyIDOrReference = keyIDOrRef;
         authValue = authVal;
    }

    //ACCESSORS:
    public int getAccessMethodType()
    {   return accessMethodType; }

    public void setAccessMethodType(int type)
    {   accessMethodType = type; }

    public int getKeyIDOrReference()
    {   return keyIDOrReference; }

    public void setKeyIDOrReferece(int keyIDOrRef)
    {    keyIDOrReference = keyIDOrRef; }

    public byte[] getAuthValue()
    {   return authValue; }

    public void setAuthValue(byte[] auth)
    {   authValue = auth; }


}
```

The fields of the `BSIAuthenticator` class are:

`accessMethodType:` Access Method Type (see Table 3-1 in Section 3.1).

`keyIDOrReference:` Key identifier or reference of the authenticator. This is used to distinguish between multiple authenticators with the same Access Method Type.

`authValue:` Authenticator, can be an external authentication cryptogram or PIN. If the authenticator value is NULL, then BSI is in charge of gathering authentication information and authenticating to the card.

Return codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_ACR_NOT_AVAILABLE
BSI_SC_LOCKED
BSI_BAD_AUTH
BSI_CARD_REMOVED
BSI_PIN_BLOCKED
BSI_TERMINAL_AUTH
BSI_UNKNOWN_ERROR
```

F.2.2 gscBsiUtilConnect()

Purpose: Establish a logical connection with the card inserted 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:

```
public abstract int gscBsiUtilConnect(
                        String   readerName
) throws gov.gsc.classes.BSIException;
```

Parameter: **`readerName:`** 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().`** The Name of the reader shall be stored as ASCII encoding Strings. (See Section 4.2)

Return Value: **`hCard:`** Card connection handle.

Return codes:

```
BSI_OK
BSI_UNKNOWN_READER
BSI_CARD_ABSENT
BSI_TIMEOUT_ERROR
BSI_UNKNOWN_ERROR
```

F.2.3 gscBsiUtilDisconnect()

Purpose: Terminate a logical connection to a card.

Prototype:
```
public abstract void gscBsiUtilDisconnect(
                                int    hCard
) throws gov.gsc.classes.BSIException;
```

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

Return codes:
```
BSI_OK
BSI_BAD_HANDLE
BSI_CARD_REMOVED
BSI_UNKNOWN_ERROR
```

F.2.4 gscBsiUtilBeginTransaction()

Purpose: Starts an exclusive transaction with the smart card referenced by hCard. When the transaction starts, all other applications are precluded from accessing the smart card while the transaction is in progress. Two types of calls can be made with that function: a blocking transaction call and a non-blocking transaction call. A boolean type parameter identify which mode is called. In the non-blocking mode, the call will return immediately if another client has an active transaction lock. The returned error code will be `BSI_SC_LOCKED`. In the blocking mode, the call will wait indefinitely for any active transaction locks to be released. A transaction must be completed by a call to **`gscBsiUtilEndTransaction().`**

For single-threaded BSI implementations, it can be assumed that each application will be associated with a separate process. The same process that starts a transaction must also complete the transaction. For multi-threaded BSI implements, it can be assumed that each application will be associated with a separate thread and/or process. The same thread that starts a transaction must also complete the transaction.

If this function is called by a thread that has already called **`gscBsiUtilBeginTransaction()`** but has not yet called **`gscBsiUtilEndTransaction()`** it will return the error `BSI_NOT_TRANSACTED`.

If the SPS (Service Provider Software) does not support transaction locking, it should return the error code `BSI_NO_SPSSERVICE` in response to a call to **`gscBsiUtilBeginTransaction()`**.

Prototype:

```
public abstract void gscBsiUtilBeginTransaction(
                        int      hCard
                        boolean  blType
) throws gov.gsc.classes.BSIException;
```

Parameters:

hCard:	Card communication handle returned from **`gscBsiUtilConnect()`**
blType:	Boolean specifying the type of transaction call (blType set to “true” in blocking mode. blType set to “false” in non blocking mode).

Return Code:

```
BSI_OK
BSI_BAD_HANDLE
BSI_UNKNOWN_ERROR
BSI_SC_LOCKED
BSI_NOT_TRANSACTED
BSI_NO_SPSSERVICE
```

F.2.5 gscBsiUtilEndTransaction()

Purpose: Completes a previously started transaction, allowing other applications to resume interactions with the card.

If this function is called by a thread that has not yet called **`gscBsiUtilBeginTransaction()`** it will return the error `BSI_NOT_TRANSACTED`.

If the SPS (Service Provider Software) does not support transaction locking, it should return the error code `BSI_NO_SPSSERVICE` in response to a call to **`gscBsiUtilEndTransaction()`**.

Prototype:

```
public abstract void gscBsiUtilEndTransaction(
                          int      hCard
)  throws gov.gsc.classes.BSIException;
```

Parameters:

`hCard:`	Card communication handle returned from **`gscBsiUtilConnect().`**

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_UNKNOWN_ERROR
BSI_NOT_TRANSACTED
BSI_NO_SPSSERVICE
```

F.2.6 gscBsiUtilGetVersion()

Purpose: Returns the BSI implementation version.

Prototype:
```
public abstract String gscBsiUtilGetVersion(
)  throws gov.gsc.classes.BSIException;
```

Return Value:

version:	A String representing the BSI and SPS's version formatted as "major,minor,revision,build_number". The value for an SPS conformant with this version of the GSC-IS is "2,1,0,<build number>". The build number field is vendor/implementation dependent. The version shall be stored as ASCII encoded Strings. (See Section 4.2)

Return codes:
```
BSI_OK
BSI_UNKNOWN_ERROR
```

F.2.7 gscBsiUtilGetCardProperties()

Purpose: Retrieves ID and capability information for the card.

Prototype:
```
public abstract CardProperties gscBsiUtilGetCardProperties(
                int       hCard
) throws gov.gsc.classes.BSIException;
```

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

Return Value: **`cardProps:`** A **`CardProperties`** object defined as follows:

```
public class gov.gsc.classes.CardProperties
{
      //FIELDS:
      protected int        cardCapability;
      protected byte[]     CCCUniqueID;

      //CONSTRUCTORS:
      public CardProperties( int capability,
                          byte[] uniqueID)
      {
           cardCapability = capability;
           CCCUniqueID = uniqueID;
      }

      //ACCESSORS
      public int getCardCapability()
      {   return cardCapability; }

      public byte[]  getCCCUniqueID()
      {   return CCCUniqueID; }

      public void setCardCapability(int capability)
      {   cardCapability = capability; }

      public void setCCCUniqueID(byte[] id)
      {   CCCUniqueID = id; }
}
```

where the fields are described as follows:

`CCCUniqueID:` String for the Card Capability Container ID.

`cardCapability:` 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.

Return codes:
```
BSI_OK
BSI_BAD_HANDLE
BSI_CARD_REMOVED
BSI_SC_LOCKED
```

```
BSI_NO_CARDSERVICE
BSI_UNKNOWN_ERROR
```

F.2.8 gscBsiUtilGetCardStatus()

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

Prototype:

```
public abstract void gscBsiUtilGetCardStatus(
                int       hCard
) throws gov.gsc.classes.BSIException;
```

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

Return codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_CARD_REMOVED
BSI_UNKNOWN_ERROR
```

F.2.9 gscBsiUtilGetExtendedErrorText()

Purpose: When a BSI function call throws a `BSIException`, 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:

```
public abstract String gscBsiUtilGetExtendedErrorText(
                int       hCard
) throws gov.gsc.classes.BSIException;
```

Parameters:

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

Return Value:

`errorText:` 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`. The error text shall be stored as ASCII encoding Strings. (See Section 4.2)

Return Codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_NO_TEXT_AVAILABLE
BSI_UNKNOWN_ERROR
```

F.2.10 gscBsiUtilGetReaderList()

Purpose: Retrieves the list of available readers.

Prototype:
```
public abstract java.util.Vector gscBsiUtilGetReaderList()
                                 throws gov.gsc.classes.BSIException;
```

Return Value: **`vReaderList:`** Vector of Strings containing a list of the available readers. The Strings shall be in ASCII format.

Return codes:
`BSI_OK`
`BSI_UNKNOWN_ERROR`

F.2.11 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:

```
public abstract byte[] gscBsiUtilPassthru(
                           int      hCard,
                           byte[]   cardCommand
) throws gov.gsc.classes.BSIException;
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect().**
cardCommand:	An array of bytes representing the APDU to be sent to the card. The parameter must be in ASCII hexadecimal format.

Return Value:

cardResponse:	An array of bytes representing the APDU response from the card. The parameter must be in ASCII hexadecimal format. 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_SC_LOCKED
BSI_CARD_REMOVED
BSI_UNKNOWN_ERROR
```

F.2.12 gscBsiUtilReleaseContext()

Purpose: Terminate a session with the target container on the card.

Prototype:

```
public abstract void gscBsiUtilReleaseContext(
                int       hCard,
                String    AID
) throws gov.gsc.classes.BSIException;
```

Parameters:

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

AID: Target container AID value. The AID shall be stored as an ASCII hexadecimal string.

Return codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_SC_LOCKED
BSI_CARD_REMOVED
BSI_UNKNOWN_ERROR
```

F.3 Smart Card Generic Container Provider Module Interface Definition

F.3.1 gscBsiGcDataCreate()

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

Prototype:

```
public abstract void gscBsiGcDataCreate(
                int       hCard,
                String    AID,
                short     tag,
                byte[]    value
) throws gov.gsc.classes.BSIException;
```

Parameters:

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

`AID:` Target container AID value. The AID shall be stored as an ASCII hexadecimal string.

`tag:` Tag of data item to store.

`value:` Data value to store.

Return codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_PARAM
BSI_SC_LOCKED
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_NO_MORE_SPACE
BSI_TAG_EXISTS
BSI_IOERROR
BSI_UNKNOWN_ERROR
```

F.3.2 gscBsiGcDataDelete()

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

Prototype:

```
public abstract void gscBsiGcDataDelete(
                int       hCard,
                String    AID,
                short     tag
) throws gov.gsc.classes.BSIException;
```

Parameters:

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

AID: Target container AID value. The AID shall be stored as an ASCII hexadecimal string.

tag: Tag of data item to delete.

Return codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_BAD_TAG
BSI_SC_LOCKED
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_IO_ERROR
BSI_UNKNOWN_ERROR
```

F.3.3 gscBsiGcGetContainerProperties()

Purpose: Retrieves the properties of the specified container.

Prototype:

```
public abstract ContainerProperties
gscBsiGcGetContainerProperties(
              int       hCard,
              String    AID
) throws gov.gsc.classes.BSIException;
```

Parameters:

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

AID: Target container AID value. The AID shall be stored as an ASCII hexadecimal string.

Return Value: **containerProps:** A **ContainerProperties** object defined as follows:

```
public class gov.gsc.classes.ContainerProperties
{
      //FIELDS:
      protected GCacr strctGCacr;
      protected GCContainerSize strctContainerSizes;
      protected String containerVersion;

      //CONSTRUCTORS:
      public ContainerProperties()
      {
           strctGCacr = new GCacr();
           strctContainerSizes = new GCContainerSize();
           containerVersion = new
      byte[CONTAINER_VERSION_MAXLENGTH];
      }
      public ContainerProperties( GCacr acr,
                                  GCContainerSize sizes,
                        String version)
      {
           strctGCacr = acr;
           strctContainerSizes = sizes;
           containerVersion = version;
      }

      //ACCESSORS
      public GCacr getGCacr()
      {   return strctGCacr; }

      public GCContainerSize getGCContainerSize()
      {   return strctContainerSizes; }


      public String getContainerVersion()
      {   return containerVersion; }

      public void setGCacr(GCacr thisACR)
      {   strctGCacr = thisACR; }
```

```
      public void setGCContainerSize(GCContainerSize thisSize)
      {   strctContainerSizes = thisSize; }

      public void setContainerVersion(String thisVersion)
      {   containerVersion = thisVersion; }

}
```

where the fields are described as follows:

`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 3-2 (Section 3.1). The allowable ACRs for each function are listed in Table 3-3 (Section 3.2). `keyIDOrReference` contains the key identifier or reference for each access method contained in the ACR in order of appearance. `AuthNb` is the number of access methods logically combined in the ACR. ACRID is RFU and must be NULL (0x00) in this version.

The `class GCacr` is defined as follows:

```
public class gov.gsc.classes.GCacr
{
   //FIELDS:
   protected BSIAcr createACR;
   protected BSIAcr deleteACR;
   protected BSIAcr readTagListACR;
   protected BSIAcr readValueACR;
   protected BSIAcr updateValueACR;

   //CONSTRUCTORS
   public GCacr()
   {
        createACR = new BSIAcr();
        deleteACR = new BSIAcr();
        readTagListACR = new BSIAcr();
        readValueACR = new BSIAcr();
        updateValueACR = new BSIAcr();
   }
   public GCacr(BSIAcr c, BSIAcr d, BSIAcr rt, BSIAcr rv,
   BSIAcr u)
   {
        createACR = c;
        deleteACR = d;
        readTagListACR = rt;
        readValueACR = rv;
        updateValueACR = u;
   }
```

```
   //ACCESSORS
   public void setCreateACR(BSIAcr  i)
   {   createACR = i; }
   public void setDeleteACR(BSIAcr  i)
   {   deleteACR = i; }
   public void setReadTagListACR(BSIAcr  i)
   {   readTagListACR = i; }
   public void setReadValueACR(BSIAcr  i)
   {   readValueACR = i; }
   public void setUpdateValueACR(BSIAcr  i)
   {   updateValueACR = i; }
   public BSIAcr getCreateACR()
   {   return createACR; }
   public BSIAcr getDeleteACR()
   {   return deleteACR; }
   public BSIAcr getReadTagListACR()
   {   return readTagListACR; }
   public BSIAcr getReadValueACR()
   {   return readValueACR; }
   public BSIAcr getUpdateValueACR()
   {   return updateValueACR; }
}
```

The `class BSIAcr` is defined as follows:

```
public class gov.gsc.classes.BSIAcr
{
   //FIELDS:
   protected int ACRType;
   protected int[] keyIDOrReference;
   protected int authNb;
   protected int ACRID;


   //CONSTRUCTORS
   public GCacr()
   {
        ACRType = 0;
        keyIDOrReference = new int[MaxNbAM];
        authNb = 0;
        ACRID = 0;
   }
   public GCacr(int acrType, int[] keyIDOrRef, int authNum,
   int acrID)
   {
        ACRType = acrType;
        keyIDOrReference = keyIDOrRef;
        authNb = authNum;
        ACRID = acrID;
   }


   //ACCESSORS
   public void setACRType(int  i)
   {   ACRType = i; }
   public void setKeyIDOrReference(int[]  i)
```

```
   {   keyIDOrReference = i; }
   public void setAuthNb(int  i)
   {   authNb = i; }
   public void setACRID(int  i)
   {   ACRID = i; }
   public int getACRType()
   {   return ACRType; }
   public int[] getKeyIDOrReference()
   {   return keyIDOrReference; }
   public int getAuthNb()
   {   return authNb; }
   public int getACRID()
   {   return ACRID; }
}
```

strctContainerSizes: Object indicating the size (in bytes) of the container specified by the `AID`.

```
public class gov.gsc.classes.GCContainerSize
{
      protected int maxNbDataItems;
      protected int maxValueStorageSize;

      //CONSTRUCTORS
      public GCContainerSize ()
      {
          maxNbDataItems = 0;
          maxValueStorageSize = 0;
      }
      public GCContainerSize (int i, int s)
      {
          maxNbDataItems = i;
          maxValueStorageSize = s;
      }

      //ACCESSORS
      public void setMaxNbDataItems(int  i)
      {   maxNbDataItems = i; }
      public void setMaxValueStorageSize(int  i)
      {   maxValueStorageSize = i; }
      public int getMaxNbDataItems()
      {   return maxNbDataItems; }
      public int getMaxValueStorageSize()
      {   return maxValueStorageSize; }
}
```

Return codes:

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

F.3.4 gscBsiGcReadTagList()

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

Prototype:

```
public abstract short[] gscBsiGcReadTagList(
                int       hCard,
                String    AID
) throws gov.gsc.classes.BSIException;
```

Parameters:

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

AID: Target container AID value. The AID shall be stored as an ASCII hexadecimal string.

Return Value:

tagListArray: An array containing the list of tags for the selected container. The tags shall be of the type "short".

Return codes:

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

F.3.5 gscBsiGcReadValue()

Purpose: Returns the Value associated with the specified Tag.

Prototype:

```
public abstract byte[]     gscBsiGcReadValue(
                int        hCard,
                String     AID,
                short      tag
) throws gov.gsc.classes.BSIException;
```

Parameters:

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

AID: Target container AID value. The AID shall be stored as an ASCII hexadecimal string.

tag: Tag value of data item to read.

Return Value:

value: Data Value associated with the specified tag.

Return codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_SC_LOCKED
BSI_BAD_AID
BSI_BAD_TAG
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_IO_ERROR
BSI_UNKNOWN_ERROR
```

F.3.6 gscBsiGcUpdateValue()

Purpose: Updates the Value associated with the specified Tag.

Prototype:

```
public abstract void gscBsiGcUpdateValue(
                int       hCard,
                String    AID,
                short     tag,
                byte[]    value
) throws gov.gsc.classes.BSIException;
```

Parameters:

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

AID: Target container AID value. The AID shall be stored as an ASCII hexadecimal string.

tag: Tag of data item to update.

value: New Value of the data item.

Return codes:

```
BSI_OK
BSI_BAD_HANDLE
BSI_BAD_AID
BSI_SC_LOCKED
BSI_BAD_PARAM
BSI_BAD_TAG
BSI_CARD_REMOVED
BSI_NO_CARDSERVICE
BSI_ACCESS_DENIED
BSI_NO_MORE_SPACE
BSI_IO_ERROR
BSI_UNKNOWN_ERROR
```

F.4 Smart Card Cryptographic Provider Module Interface Definition

F.4.1 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 `authValue` instance field of the `BSIAuthenticator` object.

Prototype:

```
public abstract byte[]  gscBsiGetChallenge(
                Int       hCard,
                String    AID
) throws gov.gsc.classes.BSIException;
```

Parameters:

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

`AID:` Target container AID value. The AID shall be stored as an ASCII hexadecimal string.

Return Value:

`challenge:` An array of bytes representing a random challenge returned from the card.

Return codes:

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

F.4.2 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:

```
public abstract byte[]  gscBsiSkiInternalAuthenticate(
                int       hCard,
                String    AID,
                short     algoID,
                byte[]    challenge
) throws gov.gsc.classes.BSIException;
```

Parameters:

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

`AID:` SKI provider module AID value. The AID shall be stored as an ASCII hexadecimal string.

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

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

Return Value: **`cryptogram:`** The cryptogram computed by the card.

Return codes:

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

F.4.3 gscBsiPkiCompute()

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

Prototype:

```
public abstract byte[]  gscBsiPkiCompute(
                int       hCard,
                String    AID,
                short     algoID,
                byte[]    message
) throws gov.gsc.classes.BSIException;
```

Parameters:

hCard:	Card connection handle from **gscBsiUtilConnect()**.
AID:	PKI provider module AID value. The AID shall be stored as an ASCII hexadecimal string.
algoID:	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.
message:	The message digest to be signed.

Return Value:

result:	An array of bytes containing the signature.

Return codes:

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

F.4.4 gscBsiPkiGetCertificate()

Purpose: Reads the certificate from the card.

Prototype:

```
public abstract byte[]  gscBsiPkiGetCertificate(
                int       hCard,
                String    AID
) throws gov.gsc.classes.BSIException;
```

Parameters:

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

`AID:` PKI provider module AID value. The AID shall be stored as an ASCII hexadecimal string.

Return Value:

`certificate:` An array of bytes containing the certificate.

Return codes:

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

F.4.5 gscBsiGetCryptoProperties()

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

Prototype:

```
public abstract CryptoProperties gscBsiGetCryptoProperties(
                int          hCard,
                String       AID
) throws gov.gsc.classes.BSIException;
```

Parameters:

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

AID: AID of the PKI provider. The AID shall be stored as an ASCII hexadecimal string.

Return Value: **cryptoProps:** A CryptoProperties object defined as follows:

```
public class gov.gsc.classes.CryptoProperties
{
   // FIELDS
   protected CRYPTOacr strctCRYPTOacr;
   protected int keyLen;

   // CONSTRUCTORS
   public CryptoProperties()
   {
         strctCRYPTOacr = new CRYPTOacr();
         keyLen = 0;
   }

   public CryptoProperties( CRYPTOacr acr, int keylen)
   {
         strctCRYPTOacr = acr;
         keyLen = keylen;
   }

   // ACCESSORS
   public CRYPTOacr getCRYPTOacr()
   { return strctCRYPTOacr; }

   public int getKeyLen()
   { return keyLen; }

   public void setCRYPTOacr(CRYPTOacr thisACR)
   { strctCRYPTOacr = thisACR; }

   public void setKeyLen(int keylen)
   { keyLen = keylen; }
}
```

strctCRYPTOacr: Object indicating access control conditions for all operations. The `BSIAcr` structure is defined in Section F.3.3. The range of possible values for the instance fields of this object are defined in Table 3-2 (Section 3.1), and the allowable ACRs for each function in Table 3-4 (Section 3.2).

`keyIDOrReference` contains the key identifier or reference for each access method contained in the ACR in order of appearance. `authNb` is the number of access methods logically combined in the ACR. ACRID is RFU and must be NULL (0x00) in this version. Note that the `readValueACR` member maps to the **`gscBsiPkiGetCertificate()`** function.

```
public class gov.gsc.classes.CRYPTOacr
{
   //FIELDS:
   protected BSIAcr getChallengeACR;
   protected BSIAcr internalAuthenticateACR;
   protected BSIAcr pkiComputeACR;

   protected BSIAcr createACR;
   protected BSIAcr deleteACR;
   protected BSIAcr readTagListACR;
   protected BSIAcr readValueACR;
   protected BSIAcr updateValueACR;

   //CONSTRUCTORS
   public CRYPTOacr()
   {
         getChallengeACR = new BSIAcr();
         internalAuthenticateACR = new BSIAcr;
         pkiComputeACR = new BSIAcr;
         createACR = new BSIAcr();
         deleteACR = new BSIAcr();
         readTagListACR = new BSIAcr();
         readValueACR = new BSIAcr();
         updateValueACR = new BSIAcr();
   }

   public CRYPTOacr(BSIAcr ch, BSIAcr ia, BSIAcr pc, BSIAcr c,
   BSIAcr d, BSIAcr rt, BSIAcr rv, BSIAcr u)
   {
         getChallengeACR = ch;
         internalAuthenticateACR = ia;
         pkiComputeACR = pc;

         createACR = c;
         deleteACR = d;
         readTagListACR = rt;
         readValueACR = rv;
         updateValueACR = u;
   }

   //ACCESSORS
   public void setGetChallengeACR(int i)
   { getChallengeACR = i; }
   public void setInternalAuthenticateACR(int i)
   { internalAuthenticateACR = i; }
   public void setPkiComputeACR(int i)
```

```
   { pkiComputeACR = i; }
   public void setCreateACR(BSIAcr  i)
   {   createACR = i; }
   public void setDeleteACR(BSIAcr  i)
   {   deleteACR = i; }
   public void setReadTagListACR(BSIAcr  i)
   {   readTagListACR = i; }
   public void setReadValueACR(BSIAcr  i)
   {   readValueACR = i; }
   public void setUpdateValueACR(BSIAcr  i)
   {   updateValueACR = i; }
   public int getGetChallengeACR()
   { return getChallengeACR; }
   public int getInternalAuthenticateACR()
   { return internalAuthenticateACR; }
   public int getPkiComputeACR()
   { return pkiComputeACR; }
   public BSIAcr getCreateACR()
   {   return createACR; }
   public BSIAcr getDeleteACR()
   {   return deleteACR; }
   public BSIAcr getReadTagListACR()
   {   return readTagListACR; }
   public BSIAcr getReadValueACR()
   {   return readValueACR; }
   public BSIAcr getUpdateValueACR()
   {   return updateValueACR; }
}
```

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

Return codes:

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

THIS PAGE INTENTIONALLY LEFT BLANK.

Appendix G—Contactless Smart Card Requirements

This appendix defines the requirements for GSC contactless smart cards, in accordance with the decisions of the Government Smart Card Interagency Advisory Board's Physical Access Interoperability Working Group (PAIWG). Contactless smart cards are often used in physical access control applications, but may also be used in the same environment as contact type cards. These requirements must therefore satisfy the following design goals:

GSC contactless cards should provide a minimum interoperability mechanism for cardholder identification in both physical access control and contact card type environments. This cardholder identification mechanism should use the same card edge functions (APDUs) and Data Models as those defined for GSC contact cards, to ensure interoperability with middleware designed for GSC contact cards.

The minimum interoperability mechanism for cardholder identification is to read a Security Equipment Integration Working Group (SEIWG)[SEIW] string from a fixed location using APDUs defined in the GSC virtual card edge interface.

G.1 Card to Reader Interoperability

GSC contactless cards and readers shall conform to ISO 14443 Parts 1 through 4[1444]. Cryptographic functionality is not required, but GSC contactless cards that implement cryptography shall use FIPS approved cryptographic algorithms in FIPS 140-2 [FIPS1] validated modules.

G.2 Contactless Card Edge

GSC contactless cards shall support two ISO 7816-4 [ISO4] APDUs required to select the SEIWG container/file and read the SEIWG string; `READ BINARY` and `SELECT EF`.

Note: The return code 0x9000 indicates command success, all other return codes indicate failure. Additional information on the `READ BINARY` APDU and `SELECT EF` APDUs can be found in Sections 5.1.1.2 and 5.1.1.4, respectively.

The Master File shall be automatically selected when a GSC contactless file system card is powered up. The applet that manages the SEIWG container shall be automatically selected on a GSC contactless Virtual Machine card at power up. SELECT MF and SELECT AID APDUs are therefore not required.

GSC contactless cards may optionally support other APDUs. These additional APDUs should be taken from the GSC file system card edge definitions in Chapter 5, to achieve maximum interoperability with middleware written for GSC contact cards.

G.3 Data Model Requirements

The SEIWG string is stored in a separate mandatory container/file in TLV format (EF 0007). This is necessary because host applications operating in a physical access control environment must be able to retrieve SEIWG strings quickly from a fixed location, and because no Access Control Rules are imposed on SEIWG container read operations. For file system cards, this file shall be a transparent file.

SEIWG File / Buffer	EF 0007	Always Read

Data Element (TLV)	Tag	Type	Max. Bytes
SEIWG Data	30	Fixed	40*
Error Detection Code	FE	LRC	1

*The SEIWG data format is defined in [SEIW].

Only the FID component is mandated for the SEIWG File / Buffer. For container based implementations, the RID component of the AID is not defined by this specification.

All GSC contactless cards shall contain the SEIWG file as defined above as well as a valid Card Capability Container (see Chapter 6). Physical access control applications may elect not to read this container for reasons of efficiency. This is possible because GSC contactless cards directly implement a subset of the GSC virtual card edge interface and therefore require no APDU mapping.

Appendix H—Acronyms

ACA	Access Control Applet
ACR	Access Control Rule
AID	Application Identifier
ANSI	American National Standards Institute
APDU	Application protocol data unit
API	Applications Programming Interface
ASN.1	Abstract Syntax Notation One
ATR	Answer-to-Reset
b	Binary value
BSI	Basic Services Interface
CAD	Card Accepting Device
CCC	Card Capability Container
CEI	Card Edge Interface
CHV	Card Holder Verification
CLA	Class Byte of the Command Message
CT	Capability Tuple
DES	Data Encryption Standard
DES3	Triple 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
FID	File ID
FCI	File Control Information
GCA	Generic Container Applet

GSC	Government Smart Card, as defined in the Smart Access Common Identification Card Solicitation
GSC-IS	Government Smart Card Interoperability Specification
h	Hexadecimal value
IEC	International Electrotechnical Commission
INS	Instruction Byte of Command Message associated with the *T=0* and *T=1* protocol
ISO	International Organization for Standardization
LEN or Len	Length
LOUD	Length of useful data
LRC	Longitudinal Redundancy Check associated with the *T=1* protocol
LSB	Less Significant Byte
LSN	Least significant nibbles
MAC	Message Authentication Code
MSB	Most Significant Byte
MSE	Manage security environment command
OCF	Open Card Framework
P1(2)	Parameters used in the *T=0* and *T=1* protocol
PAIWG	Physical Access Interoperability Working Group
PC/SC	Personal Computer/Smart Card
PIN	Personal Identification Number
PKI	Public Key Infrastructure
PKCS	Public Key Cryptography Standards
RFU	Reserved for Future Use
RID	Registerd Application Provider Identifier
SEIWG	Security Enterprise Integration Working Group
SKI	Symmetric Key Interface

SPM	Service Provider Module
SPS	Service Provider Software
ST	Status Tuple
SW1(2)	Status Word1 (2)
T=0	Character-oriented asynchronous half duplex transmission protocol
T=1	Block-oriented asynchronous half duplex transmission protocol
TLV	Tag-Length-Value
USZ	Unsigned Zero-Terminated Character String
VCEI	Virtual Card Edge Interface
VM	Virtual Machine
VM CEI	Virtual Machine Card Edge Interface
XSI	Extended Service Interface(s)