WP31 - Interface Device Service

D31.2 - Interface and Module Specification and Documentation

Document Identification	
Date	01.05.2013
Status	Final
Version	1.0

Related SP / WP	SP3 / WP31	**Document Reference**	D31.2
Related Deliverable(s)	D22.x	**Dissemination Level**	PU
Lead Participant	TUD	**Lead Author**	Moritz Horsch
Contributors	Moritz Horsch (TUD), Pouyan Sepehrdad (TUD), Christoph Busold (TUD), Tobias Wich (ECS), Detlef Hühnlein (ECS), Frank-Michael Kamm (G&D), Daniel Albert (G&D), Detlef Houdeau (IFAG), Peter Lipp (TUG), Christof Rath (TUG)	**Reviewers**	Jan Camenisch (IBM), Thomas Groß (UNEW)

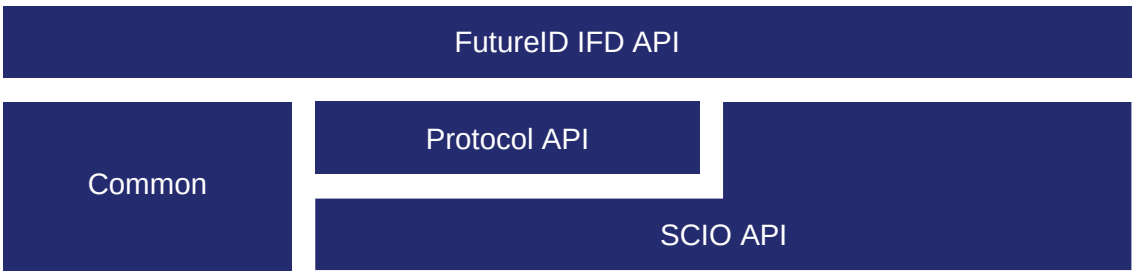
This document is issued within the frame and for the purpose of the FutureID project. This project has received funding from the European Unions Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 318424

Abstract

The FutureID project builds a comprehensive, flexible, privacy-aware and ubiquitously usable identity management infrastructure for Europe. It integrates existing eID technology, trust infrastructures, emerging federated identity management services, and modern credential technologies. It creates a user-centric system for the trustworthy and accountable management of identity claims.

One important aspect of achieving that goal is to support existing credentials like smart cards, secure elements, and hardware/software tokens. The Interface Device (IFD) service provides a common interface for communication to such credentials. Therefore, users, developers, and applications must not consider how such credentials are integrated or addressed. The IFD abstracts from the specific interfaces and physical properties like contactless interfaces. This provides platform independency and interoperability for applications.

The IFD interface and module specification focus on desktop environments supporting PC/SC [1] and mobile environments using Android [2] devices equipped with Universal Serial Bus (USB) [3], Near Field Communication (NFC) [4] [5], and the Open Mobile API [6]. Mobile Trusted Module (MTM) and Trusted Platform Module (TPM) are not considered in the IFD service. MTM are currently not implemented by any devices and TPM does not fit into the ISO/IEC 7816 [7] based architecture of the IFD.



As depicted in the Figure above the IFD provides an Application Programming Interface (API) to access the service. The IFD includes a Common module which contains generic data structures and provides convenience functions for applications. The Protocol API provides an interface for protocols to establish a secure channel between the IFD and connected devices. The Smart Card Interface Input Output Application Programming Interface (SCIO API) provides an interface for common smart card operations.

The IFD API is based on ISO/IEC 24727-4 [8] and TR-03112-6 [9], because they are . The functions of the IFD service for the FutureID client mainly equal to the TR-03112 specification. Only a few additions are made to fulfill the requirements and archive the desired functionality.

Document Information

History

Version	Date	Author	Changes
0.1	19.11.2012	Moritz Horsch	Added section introduction
0.11	14.02.2013	Moritz Horsch	Added section interfaces specification
0.12	01.02.2013	Moritz Horsch	Added section architecture
0.13	18.02.2013	Moritz Horsch	Added section requirement
0.14	28.02.2013	Moritz Horsch	Added section protocol functions
0.15	04.03.2013	Moritz Horsch	Added section protocols
0.16	07.03.2013	Moritz Horsch	Added section proxy support
0.2	22.03.2013	Moritz Horsch	Updated changes from Tobias Wich, Frank-Michael Kamm, and Daniel Albert
0.21	22.03.2013	Moritz Horsch	Added section documentation
0.22	25.03.2013	Moritz Horsch	Added section abstract
0.23	26.03.2013	Moritz Horsch	Added section conclusion
0.24	03.04.2013	Moritz Horsch	Updated changes from Detlef Hühnlein
0.25	04.04.2013	Moritz Horsch	Updated changes from Detlef Houdeau
0.26	08.04.2013	Moritz Horsch	Updated changes from Pouyan Sepehrdad and Christoph Busold
0.27	18.04.2013	Moritz Horsch	Added section limitations
0.3	19.04.2013	Moritz Horsch	Finalized document for review
0.4	29.04.2013	Moritz Horsch	Updated changes from Jan Camenisch
0.5	30.04.2013	Moritz Horsch	Updated changes from Thomas Groß
1.0	01.05.2013	Moritz Horsch	Finalized document

Table of Contents

Abstract 1

Document Information 2

Table of Contents 3

1. Introduction 5
1.1 Scope 5
1.2 Outline 5
1.3 Terminology 6
1.3.1 Key Words 6
1.3.2 Abbreviations and Notations 6

2. Preliminaries 7
2.1 MTM 7
2.2 TPM 7
2.3 Programming language 7
2.4 Platforms 7
2.5 Bindings 7

3. Architecture 8
3.1 IFD API 9
3.2 Protocol API 9
3.3 SCIO API 10
3.4 Platforms 10
3.4.1 Desktop Environment 10
3.4.2 Mobile Environment 11
3.5 Proxy Support 12
3.5.1 Architecture 12
3.5.2 Fundamentals 13
3.5.3 Implementation 14

4. Interface Specification 16
4.1 IFD API 16
4.1.1 Card terminal functions 16
4.1.2 Card functions 16
4.1.3 User interaction functions 17
4.1.4 IFD-Callback-Interface for card terminal events 17
4.1.5 Protocol functions 18
4.2 Protocol API 21
4.2.1 Protocol 21
4.2.2 ProtocolFactory 21
4.3 SCIO API 21

5. Protocols 22
5.1 Password Authenticated Connections Establishment (PACE) 22

6. Limitations 24
6.1 GetIFDCapabilities 24
6.2 BeginTransaction / EndTransaction 24
6.3 Transmit 24

7. Documentation 25
7.1 IFD API 25
7.2 Open Mobile API 25
7.3 PC/SC 25
7.4 SICCT 25
7.5 Java Smart Card I/O 25

8. Conclusion 26

9. Bibliography 27

A. Appendix 29
A.1 XML Definitions 29
A.1.1 EstablishChannel 29
A.1.2 DestroyChannel 29
A.1.3 DIDAuthenticationDataType 30
A.1.4 PACEInputType 30
A.1.5 PACEOutputType 30
A.2 Communication Handles 31

1. Introduction

1.1 Scope

The FutureID project builds a comprehensive, flexible, privacy-aware and ubiquitously usable identity management infrastructure for Europe. It integrates existing eID technology, trust infrastructures, emerging federated identity management services, and modern credential technologies. It creates a user-centric system for the trustworthy and accountable management of identity claims.

One important aspect of achieving that goal is to support existing credentials like smart cards, secure elements, and hardware/software tokens. The Interface Device (IFD) service provides a common interface for communication to such credentials. Therefore, users, developers, and applications must not consider how such credentials are integrated or addressed. The IFD abstracts from the specific interfaces and physical properties like contactless interfaces. This provides platform independency and interoperability for applications.

The scope of this document is to provide an interface and module specification as well as a documentation of the IFD service of the FutureID client according to the requirements described in D31.1 [10].

1.2 Outline

This document is structured as follows: Section **Error! Reference source not found.** considers the requirements from deliverable D31.1 [10]. Section 3 describes the architecture of the IFD service concerning the desktop and mobile environment. Section 4 specifies the interface of the IFD service. Section 5 focuses on the integration of protocols in the IFD architecture. Section 6 describes limitations regarding the Open Mobile API. Section 7 provides a list of references for further documentation of the IFD service and Section 8 concludes the interface and module specification.

1.3 Terminology

1.3.1 Key Words

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [11].

1.3.2 Abbreviations and Notations

APDU	Application Protocol Data Unit
API	Application Programming Interface
BIOS	Basic Input/Output System
CAN	Card Access Number
CAR	Certification Authority Reference
CCID	Circuit(s) Cards Interface Devices
CHAT	Certificate Holder Authorization Template
IFD	Interface Device
JSR	Java Specification Request
MRZ	Machine Readable Zone
MTM	Mobile Trusted Module
NFC	Near Field Communication
PACE	Password Authenticated Connections Establishment
PC/SC	Personal Computer/Smart Card
PIN	Personal Identification Number
PUK	Personal Unblocking Key
RFC	Request for Comments
SAL	Service Access Layer
SCIO	Smart Card Interface Input Output
SICCT	Secure Interoperable Chip Card Terminal
TEE	Trusted Execution Environment
TPM	Trusted Platform Module
URI	Uniform Resource Identifier
USB	Universal Serial Bus

2. Preliminaries

Based on the requirements from the deliverable D31.1 [10] some design and architecture decisions were made, which are described in the following Section. This Section clarifies why certain requirements are not considered and mentions some preliminaries for the interface and module specification of the IFD service.

2.1 MTM

We do not take MTM [12] into account, because currently none of the existing mobile devices implement this specification. It is also not expected, that this configuration would be implemented in the near future in Europe.

2.2 TPM

TPM [13] is specified for desktop environments. Round about 70 to 80% of the current sold computer, notebooks etc. are equipped with TPMs. The TPM hardware module is inactive by default and must be activated through the BIOS. Therefore, we can only assume a small amount of accessible TPMs in practice.

Furthermore, the TPM uses different interfaces and data structures in comparison to APDU-based smart cards according to ISO/IEC 7816 [7]. Hence, we will not include the TPM into the IFD architecture. The TPM should be put in a separate component located at the same level as the IFD and therefore directly accessible by the eID service or on top of the eID service. Hence, the integration of the TPM should be considered in WP 3.2 and WP 3.5.

2.3 Programming language

Implementations in Task 31.3 and 31.4 will be done in Java. The description of the IFD service in this document, in particular the architecture in Section 3, may focus on a Java implementation. Nevertheless, the specification of the IFD interface in Section 4 will be platform-independent and can be implemented by any programming language.

2.4 Platforms

We focus on desktop environments supporting PC/SC [1] and mobile environments using Android [2] supporting USB [3], NFC [4] [5], and the Open Mobile API [6].

2.5 Bindings

The IFD service will neither provide an implementation of SOAP [14] and PAOS [15] nor a specific interface for such protocols. The IFD will provide a common interface that can be implemented by any programming languages. Task 31.3 will provide a Java-based implementation of the IFD interface. Transport protocols based on SOAP, PAOS or REST should be supported by the Dispatcher, which will be developed in WP 3.6.

3. Architecture

This Section provides a detailed description of the architecture of the IFD service. Details about the bindings, device interfaces, and proxy functionality are described in the corresponding subsections.

As described in deliverable D31.1 [10] the IFD service should provide a generalized interface for smart cards and card terminals based on ISO/IEC 27427-4 [8] and TR-03112-6 [9]. To provide interoperability we adopt the functions from TR-03112-6 and do not change any functions. Additional functionality like the support of protocols, through a trusted channel can be established, are realised by adding new functions. This so-called IFD Application Programming Interface (API) provides a generalized interface for other application to simple access card terminals, smart cards, and secure elements. As depicted in Figure 1 the IFD API is located on top of the IFD architecture. The functions of the IFD API are described in Section 4.

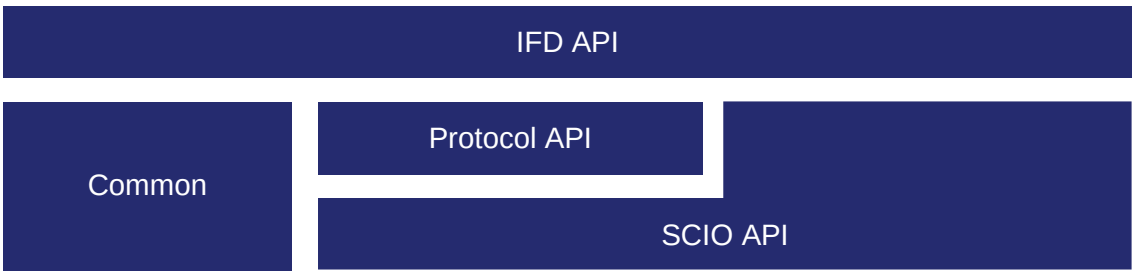


Figure 1: IFD architecture

The existing eID solutions may use various protocols; therefore, the IFD service specifies an interface to implement and integrate different protocols. Protocols running at the IFD service focus on secure channel establishment between the IFD and the connect device (e.g., smart card). Those protocols are, for instance, password-based to perform a user authentication, usually by a PIN, and ensure that only the legitimate user can use the ID card. The secure channel should protect the data transmission between the IFD and the connected device from being eavesdropped, in particular, if a contactless interface is used. The protocols are executed by a particular IFD API call (cf. Section 0).

The IFD API and the Protocol API use the SCIO API (Smart Card Interface Input Output, SCIO) to access the connected devices like card terminals, smart cards, and secure elements. The SCIO API provides an interface for common smart card operations and abstracts from the particular interfaces technology like PC/SC [1], NFC [4] [5] or SICCT [16].

In addition, the IFD includes a Common module which contains generic data structures and provides convenience functions.

3.1 IFD API

According to the requirements of the deliverable D31.1 the IFD API provides an interface according to ISO/IEC 24727-4 [8] and TR-03112-6 [9]. We adopt the functions from TR-03112-6 and added new function to provide the protocol functionality. The API is specified in Section 4.

In addition, the IFD service should support different bindings. In particular, SOAP [14] and PAOS [15] should be supported. As depicted in Figure 2, the IFD API will be implemented by a particular programming language to provide a language-specific interface. Figure 2 shows the Task 31.3 will provide a Java-based implementation of the IFD API. The interface can be directly accessed by applications or bindings for message transport (e.g., between the client and the server) like SOAP or PAOS. The integration of further bindings can be realised by implementing the language-specific interface. It is recommended that the language-specific interface is used by a Dispatcher (cf. WP 3.6) that encapsulates transport protocols like SOAP or PAOS.

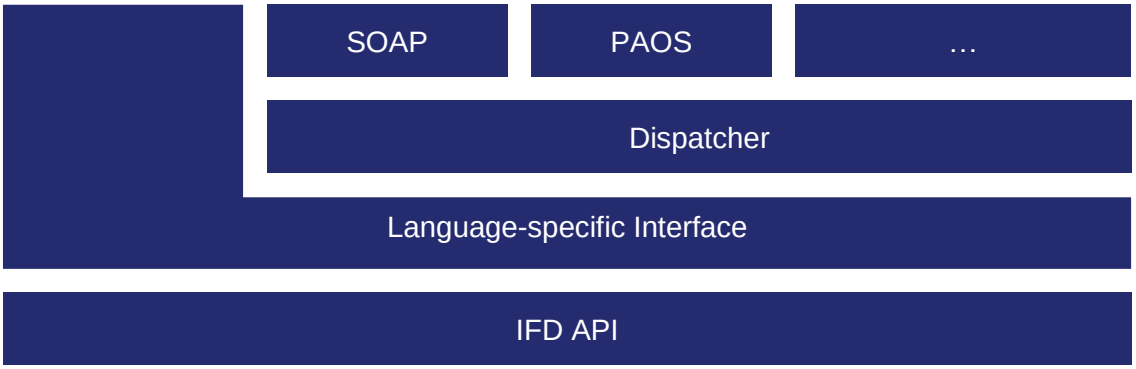


Figure 2: Bindings

3.2 Protocol API

The Protocol API provides an interface and data structures for protocols, which establish a secure channel or perform user authentication between the IFD service and connected devices. Those protocols are, for instance, password-based protocols to perform a user authentication, usually by a PIN, and ensure that only the legitimate user can use the ID card. The secure channel should protect the data transmission between the IFD and the connected device (e.g., an ID card) from being eavesdropped, in particular, if a contactless interface is used.

The protocols are selected by an identifier. Each protocol must have a unique identifier in the form of an URI. The protocols are executed by an `EstablishChannel` call through the IFD API (cf. Section 0).

For instance, the Password Authenticated Connections Establishment (PACE) [17] protocol can be integrated into the IFD service to secure the connection between the IFD and the

German identity card. Note that PACE in combination with the contactless card interface as part of the security architecture of an eID card program is today only used in Germany. By end of 2012 in Germany round 20 million new eID cards were issued.

3.3 SCIO API

The SCIO API (Smart Card Interface Input Output, SCIO) provides an interface for common smart card operations like selecting files and transmitting data. In a Java-based implementation of the IFD service the SCIO API will be mapped to the Java Smart Card I/O API (JSR 268 [18]). Java Smart Card I/O defines a Java API for communication with smart card using ISO/IEC 7816-4 [8] APDUs.

3.4 Platforms

This Section focuses on differences of technologies and configurations of mobile and desktop devices and illustrates the IFD architecture for both environments. Desktop computers are typically equipped with USB [3] interfaces and card terminals and smart cards are addressed by PS/SC [1] or SICCT [16]. Whereby, mobile devices typically equipped with NFC [4] [5] and may be USB [3]. Contactless smart cards, for instance, can be addressed via a contactless card reader using PC/SC on a desktop and NFC on mobile devices. To fulfil those different technologies on desktop and mobiles devices we provide two different architectures for the IFD. Nevertheless, only the interface to access the particular interface technology (e.g., PC/SC vs. NFC) differs on both platforms. Therefore, we only must adopt that part of the IFD service and can keep the IFD API, Protocol API, and SCIO API remain the same.

3.4.1 Desktop Environment

Figure 3 illustrates the architecture of the IFD on desktop platforms usually running Windows, Linux or Mac OS. The IFD service integrates the PC/SC interface and allows the integration of further interfaces. The PC/SC interface allows accessing the majority of card readers and smart cards. The implementation of the IFD service for desktop environments will use the Java Smart Card I/O API (JSR 268 [18]) as the SCIO API.

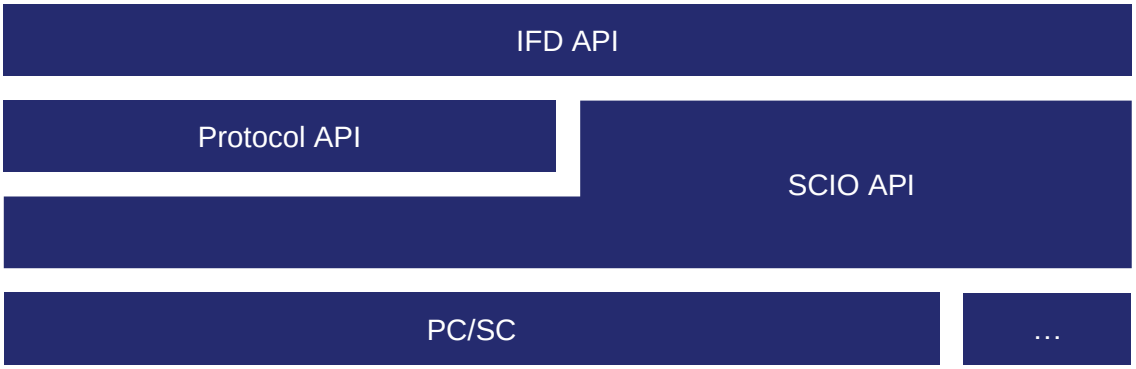


Figure 3: IFD architecture on desktop platforms

3.4.2 Mobile Environment

In the mobile environments we focus on the integration of the Open Mobile API [6] as illustrated in Figure 4. The implementation of the IFD service for desktop environments will use the Java Smart Card I/O API (JSR 268 [18]) as the SCIO API and the Transport API (cf. [6], Section 6) of the Open Mobile API. The Open Mobile API allows accessing the NFC interface and CCID-based [19] card terminals via the "plugin terminal" functionality[1].

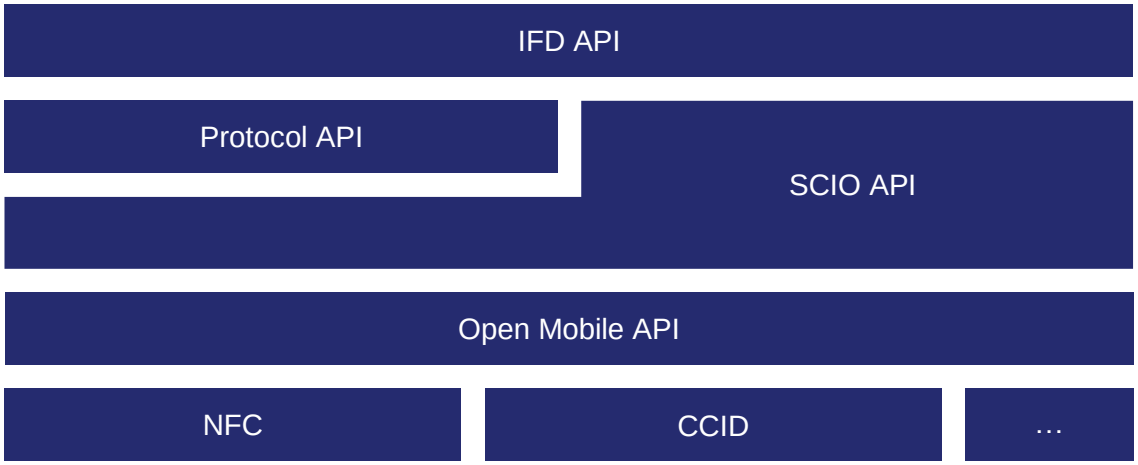


Figure 4: IFD architecture on mobile platforms

[1] https://code.google.com/p/seek-for-android/wiki/AddonTerminal

Another solution for the mobile environment could be the integration of the Android NFC API[2] and a port of the PC/SC-lite[3] library. Both can be extended to implement the Java Smart Card I/O API as illustrated in Figure 5. To keep the architecture as simple as possible we plan to access the NFC interface and external card readers via the Open Mobile API as illustrated in Figure 4.

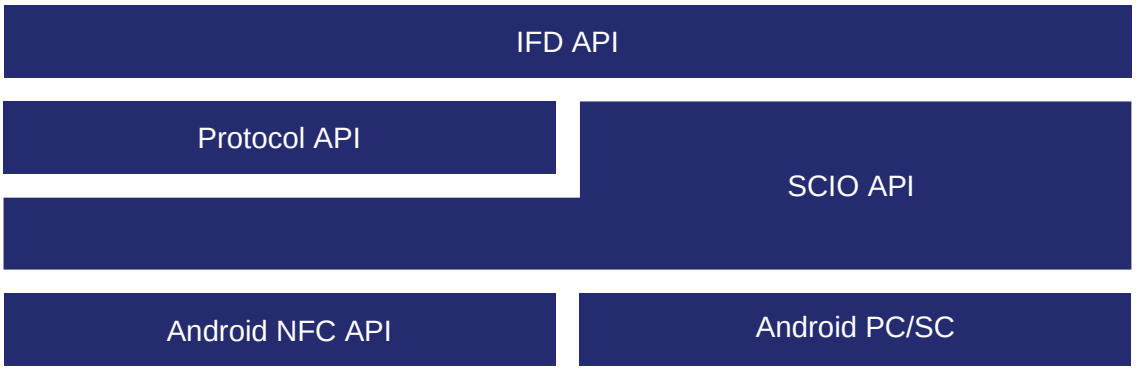


Figure 5: IFD architecture on mobile platforms using Android NFC and PC/SC API

Mobile devices are may equipped with a Trusted Execution Environment (TEE) [20], which provides a secure area to process code, data, and resources. It supports, among other things, securing the user interface, access control for resources, and execution of trusted software. To make a TEE available for the FutureID client it should be integrated as an IFD device. The strong similarity between TEE and smart cards regarding the use of APDUs for data and command exchange should allow a convenient integration.

3.5 Proxy Support

Some devices may be equipped with multiple interfaces, for instance, USB and NFC. Hence, smart cards can be accessed via an attached card terminal or directly via NFC. To support multiple interfaces simultaneously an additional meta-layer must encapsulate different IFD interfaces and must forward messages to the respective IFDs.

3.5.1 Architecture

The IFD Proxy component will encapsulate interface-specific IFDs (in short Sub-IFDs) as depicted in Figure 6. Both types of IFDs, the IFD Proxy and the Sub-IFDs, will provide the API as specified in Section 4. For external components there will be no difference between a Sub-IFD and the IFD Proxy. The functionality of the IFD Proxy will be provided in software.

[2] http://developer.android.com/guide/topics/connectivity/nfc/index.html
[3] http://pcsclite.alioth.debian.org

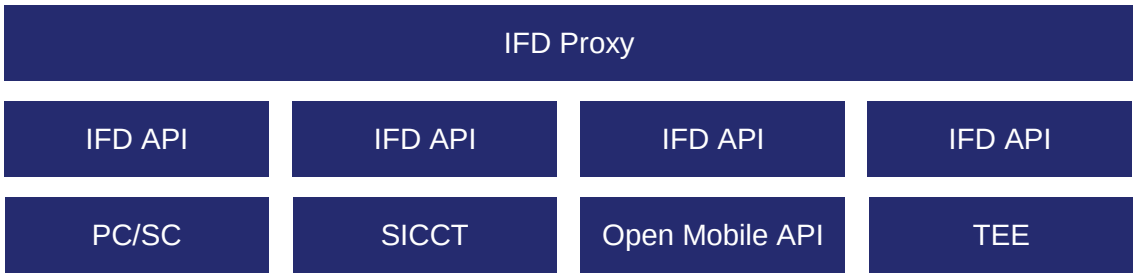


Figure 6: IFD Proxy

3.5.2 Fundamentals

This Section gives a brief overview of component addressing through different handles to provide the necessary background for the implementation details in the next Section 3.5.3. These handles are used to address the different components. In particular, if multiple card terminals or smart cards simultaneously used the handles guaranties that each devices can be clearly address. The relationship between the components and the different handles are illustrated in Figure 7. See also TR-03112-4 [21] (Page 14, Figure 2).

Applications address a Service Access Layer (SAL) by a `ChannelHandle`. An IFD is initialized by the `EstablishContext` function that returns a `ContextHandle` which is used to address the respective IFD in further API calls. Using the `ListIFDs` or `Wait` function a list of available card terminal can be received. Each card terminal is addressed by an `IFDName`. Using the `Connect` function (containing a `ContextHandle`, an `IFDName`, and a `SlotIndex`) a connection to a smart card can be established. The smart card is then addressed by a `SlotHandle`.

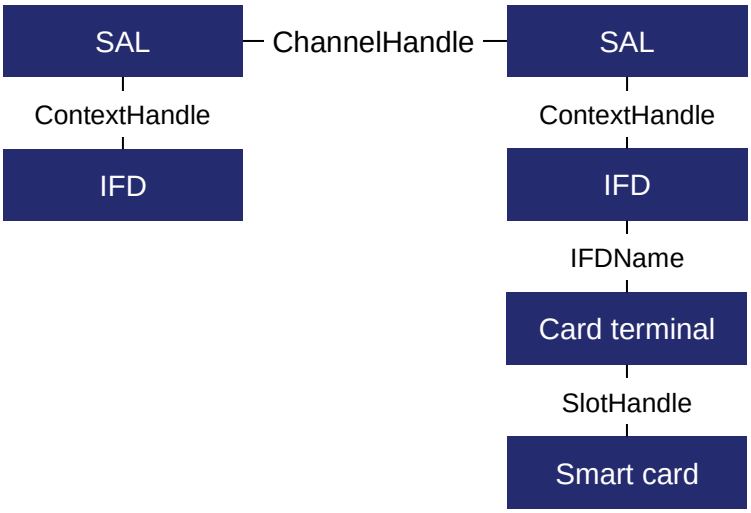


Figure 7: Component Addressing

3.5.3 Implementation

As mentioned in Section 3.5.2 each card terminal, which is attached to an IFD, is addressed by an `IFDName`. This is in general the name of the product, e.g., “MyCompany Card Reader XYZ” and it is provided by the particular interface like PC/SC[4]. Such an `IFDName` might not be unique; hence an additional parameter will be added to each Sub-IFD. The parameter acts as a suffix for the `IFDNames` associated with the corresponding IFD. A PC/SC-based IFD, for instance, with an attached card terminal named “MyCompany Card Reader XYZ” and the suffix “PCSCIFD” should extend the `IFDName` to “MyCompany Card Reader XYZ_PCSCIFD”. Each Sub-IFD must take care of that mapping.

The `IFDName` is only necessary for some IFD API functions (cf. Section A.2). If such a function is called on the IFD Proxy the request must be forwarded to the corresponding Sub-IFD. The IFD Proxy must determine the suffix of the `IFDName` used in the request and must determine the respective IFD associated with that suffix.

The behaviour of the IFD Proxy concerning the different API functions is described in the following:

FUNCTION	ACTION
`EstablishContext`	The IFD Proxy MUST call the `EstablishContext` function on each underlying Sub-IFD. If at least one Sub-IFD responses without an error the Proxy MUST return a positive `EstablishContextResponse`, otherwise the Proxy MUST return an error.
`ReleaseContext`	The IFD Proxy MUST call the function on each Sub-IFD. Errors occurring on underlying IFDs SHOULD be ignored.
`ListIFDs`	The IFD Proxy MUST call the `ListIFDs` function on all Sub-IFDs. The `IFDNames` containing in the `ListIFDsReponse` from the Sub-IFDs MUST be merged and returned as a list in a single `ListIFDsReponse`. Errors occurring on underlying IFDs SHOULD be ignored.
`GetIFDCapabilities`	The IFD Proxy MUST determine the Sub-IFD based on the suffix used in the `IFDName` and MUST call the function on the selected Sub-IFD. The Proxy MUST forward the corresponding response. Errors occurring on underlying IFDs SHOULD be ignored.
`GetStatus`	See `GetIFDCapabilities`.
`Wait`	If the `Wait` parameters contain the `IFDStatus` element the Sub-IFD SHOULD be determined by the `IFDName` contained in the `IFDStatus`. The response MUST be forwarded. If the `IFDStatus` is not present the `Wait` function MUST be called on each Sub-IFD. After

[4] See `javax.smartcardio.CardTerminal.getName()` [18]

	the first response from one Sub-IFD the Proxy MUST call the `Cancel` function on all remaining Sub-IFDs.
`Cancel`	See `GetIFDCapabilities`.
`ControlIFD`	See `GetIFDCapabilities`.
`Connect`	See `GetIFDCapabilities`. The `SlotHandle` contained in the `ConnectReponse` SHOULD be stored to determine the matching Sub-IFD in further API calls.
`Disconnect`	See `BeginTransaction`. The stored `SlotHandle` SHOULD be deleted.
`BeginTransaction`	The Sub-IFD SHOULD be determined using the stored `SlotHandle` or the Proxy SHOULD call the function on each Sub-IFD and errors with the `ResultMinor` value `invalidSlotHandle` MUST be ignored. The result of the function call from the designated Sub-IFD MUST be forwarded.
`EndTransaction`	See `BeginTransaction`.
`Transmit`	See `BeginTransaction`.
`VerifyUser`	See `BeginTransaction`.
`ModifyVerificationData`	See `BeginTransaction`.
`Output`	See `GetIFDCapabilities`.
`SignalEvent`	
`EstablishChannel`	See `BeginTransaction`.
`DestroyChannel`	See `BeginTransaction`.

4. Interface Specification

This Section describes the different interfaces of the IFD service. Namely, it specifies the IFD, Protocol, and SCIO API. The functionality of the IFD service can be access by other applications thought the IFD API. It provides functions for card terminals, smart cards (or secure elements), user interaction, and protocol establishment. The Protocol API specifies an interface for protocols which should be integrated into the IFD service. The SCIO API encapsulates common smart card / secure element functions like data transmission.

4.1 IFD API

The specification of the IFD API is based on ISO/IEC 24727-4 [8] and TR-03112-6 [9]. This specification only considers adjustment and additions to TR-03112-6.

4.1.1 Card terminal functions

The following functions are used as described in TR-03112-6. A detail description can be found in TR-03112-6.

FUNCTION	DESCRIPTION
`EstablishContext`	Opens a session with the IFD service and returns a `ContextHandle` to address the IFD instance in future.
`ReleaseContext`	Terminates a session with the IFD service.
`ListIFDs`	Returns a list of available card terminals.
`GetIFDCapabilities`	Returns information of a card terminal.
`GetStatus`	Determines the current status of the card terminal.
`Wait`	Registers an event callback (e.g., new smart card is inserted).
`Cancel`	Cancels a `Wait` call.
`ControlIFD`	Sends (proprietary) commands to a connected card terminal.

4.1.2 Card functions

The following functions are used as described in TR-03112-6. A detail description can be found in TR-03112-6.

FUNCTION	DESCRIPTION
`Connect`	Establishes a connection to a card and returns a `SlotHandle` to address the connection in future.
`Disconnect`	Destroys a connection to a smart card.
`BeginTransaction`	Starts a transaction through which several commands can be sent to the card.
`EndTransaction`	Ends a transaction.
`Transmit`	Sends a set of APDUs.

4.1.3 User interaction functions

The following functions are used as described in TR-03112-6. A detail description can be found in TR-03112-6.

FUNCTION	DESCRIPTION
`VerifyUser`	Verifies the user by means of a PIN.
`ModifyVerificationData`	Modifies the verification data (i.e. changes a PIN).
`Output`	Can be used to control the output units of a card terminal.

4.1.4 IFD-Callback-Interface for card terminal events

The following functions are used as described in TR-03112-6. A detail description can be found in TR-03112-6.

FUNCTION	DESCRIPTION
`SignalEvent`	Can be used to inform applications about card terminal events.

4.1.5 Protocol functions

This Section describes functions to support protocols in the IFD service. This functionality is not yet part of ISO/IEC 24727-4 [8] or TR-03112-6 [9], but will may be subject of a corresponding amendment. The XML Schemas are attached in Section A.1.

4.1.5.1 EstablishChannel

The `EstablishChannel` function (cf. Section A.1.1) can be used to establish a secure channel between the IFD service and a connected device. The connection is addressed by the `SlotHandle`, which is returned by a successful execution of the `Connect` function. The protocol used for channel establishment is selected through the `protocol` attribute in the `AuthenticationProtocolData` element. This element includes protocol-specific data defined by certain protocols (cf. Section 5).

REQUEST

Name	`EstablishChannel`	
Description	The `EstablishChannel` function establishes a channel between the IFD service and a connected device addressed by the given `SlotHandle`. The `AuthenticationProtocolData` contains protocol-specific data.	
Parameters		
	Name	`SlotHandle`
	Description	Addresses an established connection between the IFD service and a device (cf. `Connect` [9]).
	Name	`AuthenticationProtocolData`
	Description	Protocol-specific data for channel establishment. The `AuthenticationProtocolData` (cf. Section A.1.2) is defined as an open type depending on a specific protocol performed to establish the channel.

RESPONSE

Name	`EstablishChannelReponse`		
Description	The `EstablishChannelReponse` is returned in response to `EstablishChannel` and includes status information and protocol-specific data.		
Parameters			
	Name	`Result`	
	Description	Contains status information and if necessary additional error details.	
	Parameters		
		Name	`ResultMajor`
		Description	The following values are defined for the `ResultMajor` element: ▪ `/resultmajor#ok` The operation executed successfully. ▪ `/resultmajor#error` The operation could not be satisfied due to an error. Details are indicated through the `ResultMinor` and `ResultMessage` element.
		Name	`ResultMinor`
		Description	One of the following `ResultMinor` values SHOULD be returned when the `ResultMajor` value is `error`. ▪ /resultminor/ifdl/common#invalidSlotHandle ▪ /resultminor/ifdl/common#cancellationByUser ▪ /resultminor/ifdl/common#timeoutError ▪ /resultminor/ifdl/common#unknownError ▪ /resultminor/ifdl/protocol#passwordSuspended ▪ /resultminor/ifdl/protocol#passwordBlocked ▪ /resultminor/ifdl/protocol#passwordError ▪ /resultminor/ifdl/protocol#passwordDeactivated ▪ /resultminor/ifdl/protocol#authenticationFailed
		Name	`ResultMessage`
		Description	A message that MAY contain more detailed information about the error.
	Name	`AuthenticationProtocolData`	
	Description	Protocol-specific data defined by the protocol.	

4.1.5.2 DestroyChannel

The `DestroyChannel` function (cf. Section A.1.2) destroys a channel which was established by an `EstablishChannel` execution. The established channel between the IFD service and the device is addressed by the given `Slothandle`.

REQUEST

Name	`DestroyChannel`	
Description	The `DestroyChannel` function destroys an established channel. The channel is addressed by the `SlotHandle`.	
Parameters		
	Name	`SlotHandle`
	Description	Addresses an established connection between the IFD service and a device (cf. `Connect` [9]).

RESPONSE

Name	`DestroyChannelReponse`		
Description	The `DestroyChannelReponse` is returned in response to `DestroyChannel` and includes status information.		
Parameters			
	Name	`Result`	
	Description	Contains status information and if necessary additional error details.	
	Parameters		
		Name	`ResultMajor`
		Description	The following values are defined for the `ResultMajor` element: ▪ `/resultmajor#ok` The operation executed successfully. ▪ `/resultmajor#error` The operation could not be satisfied due to an error. Details are indicated through the `ResultMinor` and `ResultMessage` element.
		Name	`ResultMinor`
		Description	One of the following `ResultMinor` values SHOULD be returned when the `ResultMajor` value is `error`. ▪ /resultminor/ifdl/common#invalidSlotHandle
		Name	`ResultMessage`
		Description	A message which MAY contain more detailed information about the error.

4.2 Protocol API

This Section describes the Protocol API. Protocols that should be integrated into the IFD service must implement that API.

4.2.1 Protocol

The interface `Protocol` defines functions for IFD protocols which establish a channel. Each protocol must implement the `establish` function to perform the protocol. In addition, the interface specifies functions to apply and remove Secure Messaging from the APDU according to ISO/IEC 7816-4.

Methods and Description

```
establish(EstablishChannel request, Dispatcher dispatcher, UserConsent
gui) : EstablishChannelResponse
```

Perform protocol and thereby set up a secure messaging channel.

```
applySM(byte[] commandAPDU) : byte[]
```

Filter function to perform secure messaging after the protocol has been established. Apply secure messaging encryption to APDU.

```
removeSM(byte[] responseAPDU) : byte[]
```

Filter function to perform secure messaging after the protocol has been established. Remove secure messaging encryption from APDU.

4.2.2 ProtocolFactory

Each protocol must implement the `ProtocolFactory` to create a protocol instance for a connection. The `getProtocol()` function must return a unique identifier that represents the respective protocol.

Methods and Description

```
createInstance() : Protocol
```

Create an instance of the protocol.

```
getProtocol() : String
```

Returns the URI of the protocol.

4.3 SCIO API

The IFD service uses the Java Smart Card I/O as the SCIO API. Details can be found in the JSR 256 [18].

5. Protocols

This section focus on protocols running at the IFD service to establish a secure channel between the IFD and the connect device. Those protocols are, for instance, password-based to perform a user authentication, usually by a PIN, and ensure that only the legitimate user can use the ID card. The secure channel should protect the data transmission between the IFD and the connected device (e.g., an ID card) from being eavesdropped, in particular, if a contactless interface is used.

This Section contains protocol-specific `AuthenticationProtocolData` used in the messages `EstablishChannel` and `EstablishChannelReponse` (cf. Section 0).

5.1 Password Authenticated Connections Establishment (PACE)

The PACE [17] protocol is used in the security architecture of the German Identity Card. It performs a user authentication based on a PIN and establishes a secure channel between the IFD and the ID card to protect the contactless interface. This Section describes the PACE-specific `AuthenticationProtocolData`. The XML Schemas are listed in Section A.1.4 and A.1.5. See TR-03110 [17] for detailed information about the protocol.

REQUEST

Name	`PACEInputType`	
Description	Used as `AuthenticationProtocolData` in the `EstablishChannel` to establish the PACE protocol.	
Parameters		
	Name	`PINID`
	Description	Specifies the type of the PIN (MRZ, PIN, PUK, and CAN).
	Name	`PIN`
	Description	Contains the PIN (or MRZ, PUK, CAN respectively).
	Name	`CHAT`
	Description	Contains the CHAT (Certificate Holder Authorization Template).
	Name	`CertificateDescription`
	Description	Contains the description of the terminal certificate.

RESPONSE

Name	`PACEOutputType`	
Description	Used as `AuthenticationProtocolData` in the `EstablishChannelResponse`.	
Parameters		
	Name	`RetryCounter`
	Description	Contains the retry counter of the PIN, i.e., number of remaining attempts.
	Name	`EFCardAccess`
	Description	Contains the file content of the file `EF.CardAccess`.
	Name	`CARcurr`
	Description	Contains the current CAR (Certificate Authority Reference).
	Name	`CARprev`
	Description	Contains the previous CAR (Certificate Authority Reference).
	Name	`IDPICC`
	Description	Contains the chip identifier.

Card Terminal with keypad

The `PIN` element of the `PACEInputType` MUST be empty to indicate the usage of the card terminal's keypad. Note that the capabilities of the card reader can be obtained by calling the `GetIFDCapabilities` function.

6. Limitations

The integration of the Open Mobile API in the mobile environment causes some limitations which are described in this Section. The limitations result from unsupported functionality by the API. These limitations do not occur on the desktop environment using PC/SC or SICCT.

6.1 GetIFDCapabilities

The Open Mobile API does not support the functionality of getting information about connect devices, for instance, if a card terminal is equipped with a display or keypad. Therefore, the `GetIFDCapabilities` will not provide the intended functionality.

Information about the functionality of connected devices may be derived from the `IFDName` received from the `ListIFDs` function. But is approach should be used with caution.

6.2 BeginTransaction / EndTransaction

The Open Mobile API does not support establishing a transaction channel as indented by the `BeginTransaction` and `EndTransaction` function. Therefore, transactions cannot be used.

6.3 Transmit

The Open Mobile API does not support “channel management” APDUs. The restrictions on the set of commands that can be sent are

- MANAGE_CHANNEL commands are not allowed,
- SELECT by DF Name (P1=04) are not allowed, and
- CLA bytes with channel numbers are de-masked.

7. Documentation

This Section provides a brief list of references for further information about the technologies used in the IFD service.

7.1 IFD API

The IFD API provides a common interface for communication to various devices like smart cards and secure elements. Further details can be found in ISO/IEC 24727-4 [8] and TR-03112-6 [9].

7.2 Open Mobile API

The Open Mobile API is an API that allows applications on mobile devices to access various kinds of secure elements in a standardized way. Details can be found in the SIMalliance specification [6].

7.3 PC/SC

Personal Computer/Smart Card (PC/SC) defines a common interface to access Integrated Circuit Cards (ICC) from within different computing environments. Details can be found in the PC/SC Workgroup Specifications [1].

7.4 SICCT

Secure Interoperable Chip Card Terminal (SICCT) defines a generic concept for application-independent card terminals which is based on established card terminal and ICC standards including ISO/IEC 7816. Details can be found in the SICCT specification [16].

7.5 Java Smart Card I/O

Java Smart Card I/O defines a Java API for communication with smart cards according ISO/IEC 7816-4 [8] APDUs. Details can be found in the JSR 268 [18].

8. Conclusion

The IFD service provides an interface for communication to arbitrary devices like smart cards and secure elements. Applications can access such devices without considering the particular interface technology.

The IFD service has a sophisticated architecture that allows easily adaption of different technologies and platforms. The protocol architecture allows simply integrating additional protocols and further interface technologies are supported through the proxy functionality.

The IFD includes a Common module which contains generic data structures and provides convenience functions. The Protocol API provides an interface for protocols to establish a channel between the IFD and connected devices and the SCIO API provides an interface for common smart card operations.

The IFD API is based on the existing standards ISO/IEC 24727-4 and TR-03112-6 and provides extensions to support protocols for channel establishment. This facilitates an easy introduction and wide adaption of the interface specification.

The integration of the Open Mobile API causes some limitations. Therefore, not all functions of the API can be completely implemented. With regard to the mobile environment and possible use case that seems to be negligible. Note the described alternatives in the corresponding Sections.

9. Bibliography

[1] PC/SC Workgroup, *PC/SC Workgroup Specifications,* Version 2.01.11, Part 1 - 10, 2012.

[2] Open Handset Alliance, *Android,* http://www.android.com.

[3] USB Implementers Forum, *Universal Serial Bus Revision 3.0 Specification,* http://www.usb.org/developers/docs/, 2011.

[4] ISO/IEC, *Information technology - Telecommunications and information exchange between systems - Near Field Communication - Interface and Protocol (NFCIP-1),* International Standard, ISO/IEC 18092, 2003.

[5] ISO/IEC, *Information technology - Telecommunications and information exchange between systems - Near Field Communication Interface and Protocol -2 (NFCIP-2),* International Standard, ISO/IEC 21481, 2005.

[6] SIMalliance, *Open Mobile API specification,* Version 2.03, 2012.

[7] ISO/IEC, *Identification cards - Integrated circuit cards,* International Standard, ISO/IEC 7816, Part 1 - 13,15, 2004 - 2011.

[8] ISO/IEC, *Identification cards - Integrated circuit card programming interfaces - Part 4: Application programming interface (API) administration,* International Standard, ISO/IEC 24727-4, 2008.

[9] Bundesamt für Sicherheit in der Informationstechnik (BSI), *eCard-API-Framework - IFD-Interface,* Version 1.1.2, Part 6, 2012.

[10] FutureID, *D31.1 - Requirements report,* 2013.

[11] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels,* RFC 2119, 1997.

[12] Trusted Computing Group, *TCG Mobile Trusted Module Specification,* Version 1.0, 2010.

[13] Trusted Computing Group, *TPM Main,* Version 1.2, Part 1 - 3, 2011.

[14] B. Don, E. David, K. Gopal, L. Andrew and M. Noah, Simple Object Access Protocol (SOAP) 1.1, 2000.

[15] Liberty Alliance Project, *Liberty Reverse HTTP Bindin for SOAP Specification,* Version 2.0, http://www.projectliberty.org/liberty/content/download/909/6303/file/liberty-paos-v2.0.pdf.

[16] TeleTrusT Deutschland e.V., *SICCT - Secure Interoperable ChipCard Terminal,* Version 1.6, 2009.

[17] Bundesamt für Sicherheit in der Informationstechnik (BSI), *Advanced Security Mechanisms for Machine Readable Travel Documents,* Technical Guideline TR-03110, Version 2.10, Part 1 - 3, https://www.bsi.bund.de/ContentBSI/EN/Publications/Techguidelines/TR03110/BSITR03110.html, 2012.

[18] Oracle Corporation, *Java Smart Card I/O API,* JSR 268, http://jcp.org/en/jsr/detail?id=268, 2006.

[19] USB Device Working Group, *CCID - Specification for Integrated Circuit(s) Cards Interface Devices,* Revision 1.1, http://www.usb.org/developers/devclass_docs/DWG_Smart-Card_CCID_Rev110.pdf, 2005.

[20] GlobalPlatform, *Trusted Execution Environment (TEE) Specifications,* Version 1.0, http://www.globalplatform.org/specificationsdevice.asp, 2011 - 2012.

[21] Bundesamt für Sicherheit in der Informationstechnik (BSI), *eCard-API-Framework - ISO 24727-3-Interface,* Technical Guideline TR-03112-4, Version 1.1.2, 2012.

[22] Bundesamt für Sicherheit in der Informationstechnik (BSI), *eCard-API-Framework,* Technical Guideline TR-03112, Part 1 - 7, Version 1.1.2, https://www.bsi.bund.de/ContentBSI/Publikationen/TechnischeRichtlinien/tr03112/index_htm.html.

[23] A. Robert and K. John, Liberty Reverse HTTP Binding for SOAP Specification, Liberty Alliance Specification, Version 2.0, 2006.

A. Appendix

A.1 XML Definitions

A.1.1 EstablishChannel

```
<element name="EstablishChannel">
 <complexType>
  <complexContent>
   <extension base="iso:RequestType">
    <sequence>
     <element name="SlotHandle" type="iso:SlotHandleType" />
     <element name="AuthenticationProtocolData" type="iso:DIDAuthenticationDataType" />
    </sequence>
   </extension>
  </complexContent>
 </complexType>
</element>

<element name="EstablishChannelResponse">
 <complexType>
  <complexContent>
   <extension base="iso:ResponseType">
    <sequence>
     <element name="AuthenticationProtocolData" maxOccurs="1" minOccurs="0"
      type="iso:DIDAuthenticationDataType" />
    </sequence>
   </extension>
  </complexContent>
 </complexType>
</element>
```

A.1.2 DestroyChannel

```
<element name="DestroyChannel">
 <complexType>
  <complexContent>
   <extension base="iso:RequestType">
    <sequence>
     <element name="SlotHandle" type="iso:SlotHandleType" maxOccurs="1" minOccurs="1" />
    </sequence>
   </extension>
  </complexContent>
 </complexType>
</element>

<element name="DestroyChannelResponse">
 <complexType>
  <complexContent>
   <extension base="iso:ResponseType">
    <sequence />
   </extension>
  </complexContent>
 </complexType>
</element>
```

A.1.3 DIDAuthenticationDataType

See TR-03112 [22] (file ISO24727-3.xsd).

```
<complexType name="DIDAuthenticationDataType">
 <complexContent>
  <extension base="anyType">
   <attribute name="Protocol" type="anyURI" use="required" />
  </extension>
 </complexContent>
</complexType>
```

A.1.4 PACEInputType

```
<complexType name="PACEInputType">
 <complexContent>
  <restriction base="iso:DIDAuthenticationDataType">
   <sequence>
    <element name="PinID" type="iso:ByteType" />
    <element name="CHAT" type="hexBinary" maxOccurs="1" minOccurs="0"/>
    <element name="PIN" type="string" maxOccurs="1" minOccurs="0" />
    <element name="CertificateDescription" type="hexBinary" maxOccurs="1" minOccurs="0" />
   </sequence>
  </restriction>
 </complexContent>
</complexType>
```

A.1.5 PACEOutputType

```
<complexType name="PACEOutputType">
 <complexContent>
  <restriction base="iso:DIDAuthenticationDataType">
   <sequence>
    <element name="RetryCounter" type="nonNegativeInteger" maxOccurs="1" minOccurs="0" />
    <element name="EFCardAccess" type="hexBinary" maxOccurs="1" minOccurs="0" />
    <element name="CARcurr" type="hexBinary" maxOccurs="1" minOccurs="0" />
    <element name="CARprev" type="hexBinary" maxOccurs="1" minOccurs="0" />
    <element name="IDPICC" type="hexBinary" maxOccurs="1" minOccurs="0" />
   </sequence>
  </restriction>
 </complexContent>
</complexType>
```

A.2 Communication Handles

The following Table shows the elements that are present in the different API functions.

	ChannelHandle	ContextHandle	IFDName	Slot	SlotHandle
EstablishContext	■	□	□	□	□
ReleaseContext	□	■	□	□	□
ListIFDs	□	■	□	□	□
GetIFDCapabilities	□	■	■	□	□
GetStatus	□	■	■	□	□
Wait	□	■	■*	□	□
Cancel	□	■	■	□	□
ControlIFD	□	■	■	□	□
Connect	□	■	■	■	□
Disconnect	□	□	□	□	■
BeginTransaction	□	□	□	□	■
EndTransaction	□	□	□	□	■
Transmit	□	□	□	□	■
VerifyUser	□	□	□	□	■
ModifyVerificationData	□	□	□	□	■
Output	□	■	■	□	□
SignalEvent	□	□	□	□	□
EstablishChannel	□	□	□	□	■
DestroyChannel	□	□	□	□	■

* The `IFDName` may be available if the `IFDStatus` element is present.