Interoperability Specification for ICCs and Personal Computer Systems

Part 9. IFDs with Extended Capabilities

Apple Computer, Inc.

Axalto

Gemplus SA

Infineon Technologies AG

Ingenico SA

Microsoft Corporation

Philips Semiconductors

Toshiba Corporation

Revision 2.01.01

September 2005

Revision History

Revision	Issue Date	Comments
2.00.10	May 28, 2004	Spec 2.0 Final Draft
2.01.00	June 23, 2005	Final Release
2.01.01	September 29, 2005	Changed Schlumberger to Axalto

Contents

1 System Architecture

The general architecture of the Interoperability Specification is described in detail in Part 1 of this document and is summarized following. This part deals with the management of IFDs with some extended capabilities such as secure PIN entry or user interface functionality. IFDs with extended capabilities are also called in this specification enhanced IFDs. To manage such type of functionality, this part describes a specific element of the architecture, the IFD Service Provider (IFDSP), indicated by the shaded area of the figure.

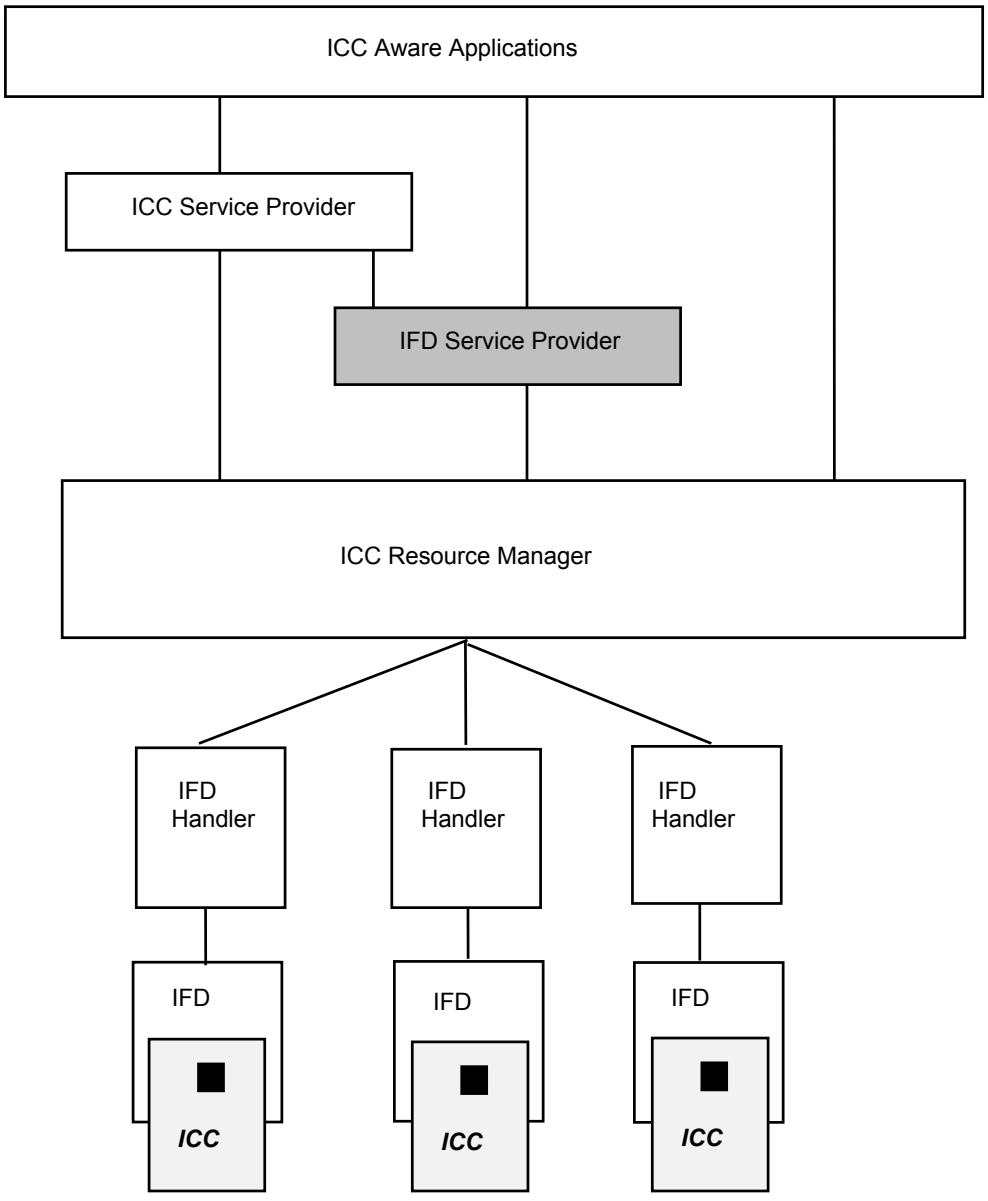


Figure 1-1 PC/SC Architecture

2 Application Context

2.1 Introduction

In many cases, some IFDs have some extra-capabilities other than just APDU communication. Such capabilities may be relative to some security requirements (e.g. secure PIN entry, biometric components, cryptography, …) or to some ergonomic reasons (e.g. use of the display/keypad features of a device). Many other capabilities may be found in different types of devices but only a limited number of capabilities is specified in this document. However, they can be included within the architecture described further on and as an extension of this specification.

2.2 Definition

An application context is a list of requirements on the IFD side in order to execute a particular ICC aware application or ICC Service Provider. It is associated to a set of functionality present in a given IFD. Some examples of requirements are:

- presence of trusted components in the IFD,
- required functionality list of the IFD —which can imply functions that shall not be supported (example: user entry other than PIN entry not supported)—,
- permitted PIN processing APDU commands,
- IFD authentication,
- sharing of physical/logical components of the IFD.

This application context can be defined for a particular card application or for a set of card applications. It may be defined by the card issuer or the on-card application developer. For example, an application context may be defined by a payment or health care system.

In order to run a particular application context, the IFD has to support this context. In most of the cases, this context will have to be implemented by the IFD supplier as it references the particular characteristic of an IFD. The way to implement an application context and to verify that a given IFD is compliant with an application context is outside of these specifications. For this purpose, some other rules might be defined by the entity defining an application context.

2.3 Association with IFDs

Some query functions are present at the Resource Manager level in order to test the support of an application context and to get an application context list for different IFDs. Only the support of an application context can be queried but not its presence. That means that an IFD may support an application context but it might not be currently available in the IFD. It might be downloaded to the IFD when necessary that is when the application context will have to be used. This download process is outside of these specifications and should be transparent to the application.

Each IFD which supports application contexts offers new functionalities for ICC Service Providers and ICC Aware application. These new functionalities which correspond to a new API appears at the IFD Service Provider level as interfaces. For a same interface, different application contexts may exist and for a given application context, different

interfaces may exist.

2.4 Resource Locking

The reservation and the locking of some IFD resources are also defined within the application context and they are implemented in the IFD subsystem and/or IFDSP depending on the security level to be achieved and on the intelligence of the IFD (see further on the logical devices). This locking defines, for example, if, for an IFD, two sessions with two application contexts identical or not can be opened at the same time. A typical reason where sharing could not be enable is because of the security. In this case, the application context may define, for example, that the entire physical device be locked as long as the execution within the application context is not completed. In some other cases, the logic itself may imply locking. For example, it does not make sense that a display be used from one side and the keypad used by another application.

3 Logical Devices

In a concrete implementation, an IFD handler may be capable to deal at the same time with several physical IFDs (same model or not). Therefore, an IFD handler may expose several IFD devices. But, in a case of multi-slot readers or enhanced readers, an IFD handler can expose more logical devices.

3.1 Slot Logical Devices and Multi-slot ICC Readers

Some of the smart card readers have more than one ICC slot. The main purpose is to have in the same physical device the user card and one or several Security Application Modules (SAMs). Some examples of applications are in the Electronic Purse domain with the customer card and the merchant card or in health care with the patient card and the physician card.

The handling of these different slots shall be treated as if there were different physical devices or mono-slot IFDs. Therefore, each slot is referenced by a different *ReaderName*. These slot logical devices have the interface detailed in the appendix A part 3 of this document. From now, to keep the current terminology, a slot is identified to a reader and in an IFD, which is a physical device unit, several slots or readers may coexist.

3.2 Functional Logical Devices

If an IFD has some extended capabilities, it may expose other devices than just slot devices. Depending on the flexibility and the security, one or several functional logical devices can be exposed. The main role of these functional logical devices is to enable the IFDSP via the Resource Manager to lock independently different types of functionality.

This second type of logical device, the functional logical device, can be in direct relation with a physical part of an enhanced IFD such as a display functional logical device or a keypad functional logical device. It can also regroup several physical parts such as a PinPad functional logical device, which may use a display and a keypad or represent a subset of the functionalities of a physical part. The functional logical device unlike the slot device supports only a generic communication channel.

This notion of functional logical device layout should be transparent to the ICC-aware application or ICCSP. It is only a mean for the IFDSP to lock through the Resource Manager a functional unit and to deal in a proprietary way with the communication to the IFD to manage the extended capabilities. The application has only an interface view of the IFDSP.

The example of configuration given in Figure 3-1 is only an implementation example. Indeed, with the same extended types of functionality with a display and a keypad, the logical devices decomposition could be totally different for the first pictured IFD. There can be only two slot logical devices and a global functional logical device. The only reason to have multi-functional logical devices is to give to the IFDSP more freedom about the locking of the devices through the Resource Manager. This may or may not enable some applications to access at the same time different types of functionality of an enhanced IFD.

As these configurations are completely independent of the way an ICC-aware

application or ICCSP uses an IFDSP and as only the proprietary implementation of the IFDSP (IFD dependent) use the particular decomposition into logical devices, an interoperability between different enhanced readers with the same types of extended functionality can occur as long as the same interfaces are exposed by the different IFDSPs.

To simplify resource sharing implementation, in the revision 2.0 of this specification, functional logical devices are always associated to a slot logical device. Therefore, doing a connection to a functional logical device must be done with a *ReaderName* and it is not possible to lock separately functional logical devices and also to lock them independently of a slot logical device.

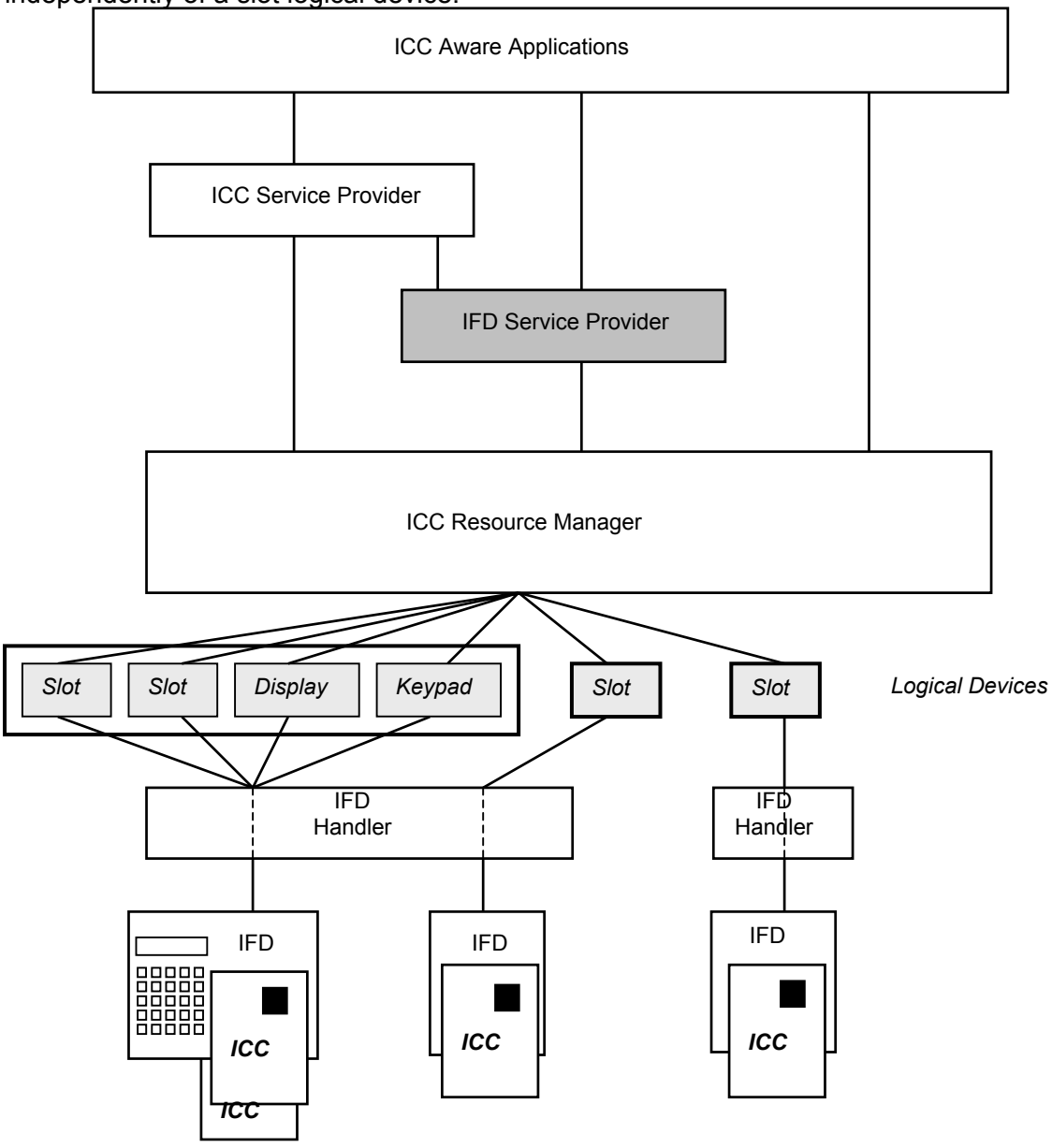


Figure 3-1 Logical Device Configuration Example

4 IFD Service Provider

The IFD Service Provider exposes the extra capabilities of an IFD and in such is delivered by the IFD supplier. The IFD Service Provider is an optional component. It communicates to the IFD handler via the Resource Manager. This communication channel is proprietary to the IFD supplier as long as it respects the interface of the Resource Manager.

As for the ICC Service Provider, different interfaces may be supported by the IFD Service Provider. A mandatory interface is described below as well as some optional interfaces that an IFD may be capable to support. The types of optional interfaces depend on the functionalities of an IFD, on the security to be achieved. However, to be interoperable between different IFDs supporting the same functionality specified hereafter, the IFD Service Provider should provide the ad hoc interfaces detailed in this part.

4.1 Functional Definition

This section describes the functional interfaces of the IFDSP. The interface is described in terms of object classes, and methods on object instances, along with required parameters and expected return values. Implementations may alter the naming conventions and parameters as required to adapt to specific environments, but shall conform to the functional interfaces defined herein.

4.1.1 Syntax

The syntax used in describing the IFDSP is based on common procedural language constructs. Data types are described in terms of common C-language due to its widespread use. The following table lists specific conventions and pre-defined values used in this document.

4.1.2 Data types

Type	Characteristic
BYTE	unsigned char, a 8-bit value
USHORT	unsigned short, a 16-bit value
BOOL	short, a 16-bit value
DWORD	unsigned long, a 32-bit value
STR	char array (string)
GUID	unsigned char[16], a 128-bit unique identifier to reference a component
IID	unsigned char[16], a 128-bit unique identifier to reference an interface
ACID	unsigned char[16], a 128-bit unique identifier to reference an application context
RESPONSECODE	long, signed 32-bit value
HANDLE	unsigned long, a 32-bit quantity

VOID	unspecified data type whose interpretation is context-specific.

Arrays of these basic data types are indicated by [ ]. For example BYTE[ ] indicates an array of BYTE values of unspecified length. BYTE[4] indicates an array of BYTE values with four elements.

Data structures are indicated using C-language “struct” type definitions. The following example defines a data structure consisting of a BYTE and DWORD value that is referenced using the SAMPLE_STRUCT identifier.

```
typedef struct {
   BYTE ByteValue
   DWORD DwordValue
} SAMPLE_STRUCT ;
```

4.1.3 Calling Conventions

The interface to the IFDSP is defined in terms of methods associated with one-high level object. Methods are invoked by referencing a named method within the context of an object instance. How the object is referenced is not specified, because this may vary by implementation. Methods require zero or more parameters and return information using a simple data type and optional output parameters.

For example,

```
RESPONSECODE MethodA(
   IN       DWORD    DwordValue
   IN OUT   BYTE     ByteValue
   OUT      BYTE     OutValue
}
```

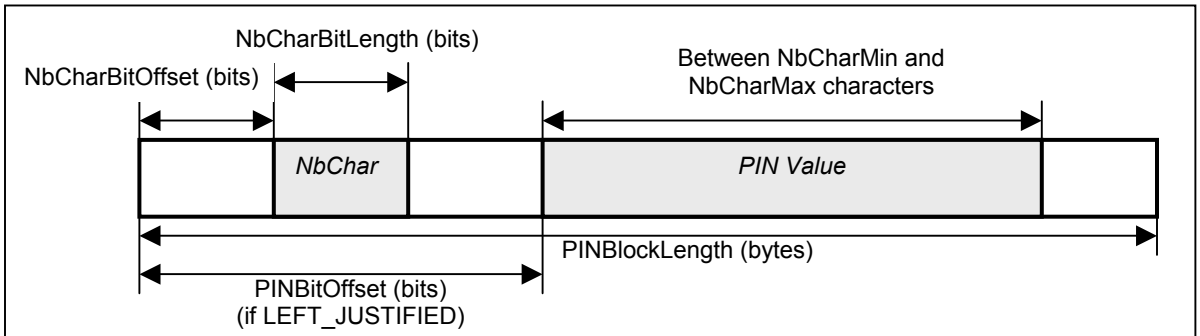
defines a method with 3 parameters which returns a RESPONSECODE value. It has two input parameters (DwordValue and ByteValue) and returns additional information in two output parameters (ByteValue and OutValue).

4.1.4 Data structure

4.1.4.1 PINBLOCK

The structure PINBLOCK is used to define the format of the PIN for the SECUREPIN interface.

```
typedef struct {
   DWORD Attributes;  // refers to the PIN overwritting
                      format as defined by the
                      PINATTRIBUTE constants. Can be
                      combined with a binary OR
   USHORT PINBlockLength; //length of the PINBlock in bytes
   USHORT NbCharMin;  // minimum number of characters to be
                      a valid PIN value
   USHORT NbCharMax;  // maximum number of characters to be
                      a valid PIN value
   USHORT PINBitOffset;  // position in bits in the APDU
                      data field to overwrite with the
                      formatted PIN. It is a position
                      from the left if it is
                      LEFT_JUSTIFIED and a position from
                      right for RIGHT_JUSTIFIED
   USHORT NbCharBitLength; // size in bits of the "number of
                        characters of the entered user
                        PIN" to overwrite in the APDU at
                        nbCharBitOffset
   USHORT NbCharBitOffset; // position in bits from the left
                        side in the APDU data field to
                        overwrite the number of
                        characters of the entered user
                        PIN
} PINBLOCK;
```



PINBLOCK Parameters

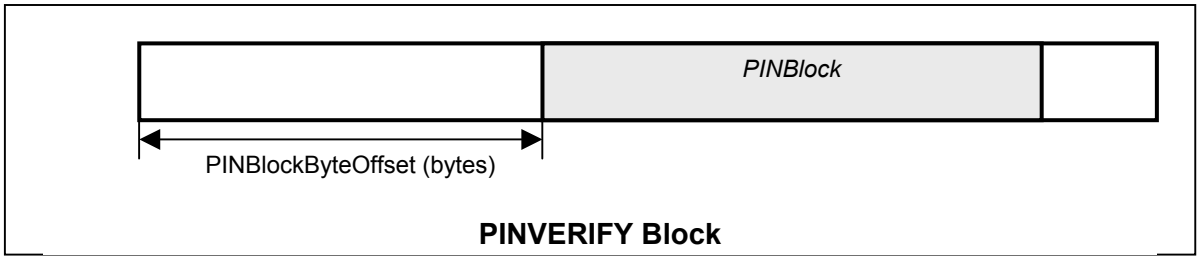
An example of use is given in the paragraph 4.1.8.1.

Remark: NbCharBitLength should be coherent with NbCharMax.

4.1.4.2 PINVERIFY

The structure PINVERIFY is used to define the format of the PIN for the SECUREPIN interface.

```
typedef struct {
   PINBLOCK PINBlockFormat; // formatting parameters of the
                            PIN block
   USHORT PINBlockByteOffset; // position in bytes in the
                              APDU data field to overwrite with
                              the PIN block
} PINVERIFY;
```



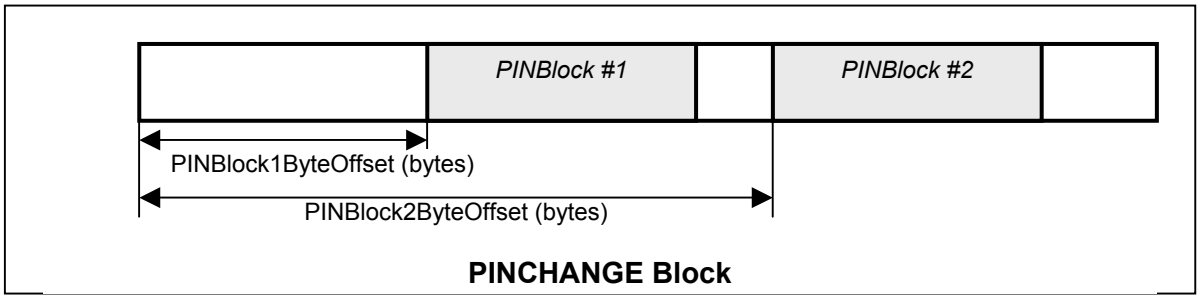
PINVERIFY Block

4.1.4.3 PINCHANGE

The structure PINCHANGE is used to define the format of the PIN for the SECUREPIN interface.

```
typedef struct {
   PINBLOCK PINBlockFormat; // formatting parameters of the
                            PIN block
   USHORT PINBlock1ByteOffset;   // position in bytes of the
                            PIN Block #1 which corresponds to
                            the current PIN (the max value of
                            USHORT corresponds to no
                            PINBlock#1). This block is
                            optional.
   USHORT PINBlock2ByteOffset;   // position in bytes of the
                            PIN Block #2 which corresponds to
                            the new PIN

} PINCHANGE;
```



PINCHANGE Block

4.1.5 Defined constants

Parameter	Symbol	Comments
PINATTRIBUTE		Bit values which can be combined in a bit mask
	NUMERIC	PIN is only composed of digits.
	ALPHANUMERIC	PIN can be composed of digits and letters.
	UPPER_ALPHANUMERIC	PIN can be composed of digits and uppercase letters.
	ASCII	Characters are coded in ASCII format.
	UNICODE	Characters are coded in UNICODE format.
	BCD	Characters (only digits) are coded in BCD format.
	LEFT_JUSTIFIED	PIN is left-justified.
	RIGHT_JUSTIFIED	PIN is right-justified.

Parameter	Symbol	Comments
PIN_INPUT_FEEDBACK		
	KEY_FIRST	The user has pressed a key. This is sent only the first time and this used by in the BEGIN_END event capability mode.
	KEY_CHARACTER	A valid character has been entered.
	KEY_CORRECTION	A correction key has been pressed.
	KEY_VALIDATION	The validation key has been pressed (end of PIN entry).
	KEY_INVALID	An invalid character has been entered.
	OP_OLD_PIN	For a PIN Change operation, indicates that the old PIN needs to be entered.
	OP_NEW_PIN	For a PIN Change operation, indicates that the new PIN needs to be entered.
	OP_CONFIRMATION	For a PIN Change operation, indicates that the new PIN needs to be re-entered for confirmation.
FEEDBACKCAPABILILITY		
	IFD	The IFD handles automatically the feedback for the PIN entry, usually via its display
	IFDSP	The IFD cannot handle the PIN entry feedback and this has to be handled by the IFDSP on the PC via a GUI for example.
	NONE	The PIN entry feedback should not be handled by the IFDSP
EVENTCAPABILITY		
	NONE	The IFD does not return any information about PIN entry
	BEGIN_END	The IFD can notify when the

Parameter	Symbol	Comments
		user has begin to enter the PIN and when this has been ended
	KEYSTROKE	The IFD can notify during the PIN entry when a key is pressed

4.1.6 Error codes

Error, Warning, and Failure codes.

Symbol	Meaning
SCARD_S_SUCCESS	No error was encountered.
SCARD_E_INSUFFICIENT_BUFFER	The data buffer to receive returned data is too small for the returned data.
SCARD_E_INVALID_HANDLE	The supplied handle was invalid.
SCARD_E_INVALID_PARAMETER	One or more of the supplied parameters' could not be properly interpreted.
SCARD_E_INVALID_TARGET	Configuration startup information is missing or invalid.
SCARD_E_INVALID_VALUE	One or more of the supplied parameters' values is not valid.
SCARD_E_NO_MEMORY	Not enough memory available to complete this command.
SCARD_E_NO_SMARTCARD	The operation requires an ICC, but no ICC is currently in the device.
SCARD_E_READER_UNAVAILABLE	The specified IFD is not currently available for use.
SCARD_E_SHARING_VIOLATION	The ICC cannot be accessed because of other connections outstanding.
SCARD_E_SYSTEM_CANCELLED	The action was cancelled by the system, presumably to log off or shut down.
SCARD_E_TIMEOUT	The user-specified timeout value has expired.
SCARD_E_UNSUPPORTED_INTERFACE	The specific interface is not supported in this service provider
SCARD_E_UNSUPPORTED_CONTEXT	The specific application context is not supported by the IFD subsystem
SCARD_E_METHOD_NOT_ALLOWED	The method cannot be used with the current application context
SCARD_E_UNKNOWN_CARD	The specified ICC name is not recognized.
SCARD_E_UNKNOWN_READER	The specified IFD name is not recognized.
SCARD_F_COMM_ERROR	An internal communications error has been detected.
SCARD_F_INTERNAL_ERROR	An internal consistency check failed.
SCARD_F_UNKNOWN_ERROR	An internal error has been detected, but the source is unknown.

Symbol	Meaning
SCARD_W_REMOVED_CARD	The card has been removed, so further communication is not possible. This error may be cleared by the SCardReconnect service.
SCARD_W_RESET_CARD	The card has been reset, so any shared state information is invalid. This error may be cleared by the ScardReconnect service.
SCARD_W_UNPOWERED_CARD	Power has been removed from the card, so further communication is not possible. This error may be cleared by the ScardReconnect service.

4.1.7 Required Interface

The following is a generic description of the standard interface that must be supported by compliant IFD Service Provider. This description is suitable for implementation in a variety of languages on a variety of systems.

4.1.7.1 Class ENHANCEDIFD

4.1.7.1.1 Properties

HANDLE hContext //handle to the Resource Manager. Set to NULL at object //creation.

4.1.7.1.2 Methods

ENHANCEDIFD()

Creates an instance of the ENHANCEDIFD class and returns a reference to the calling module. The type of this object is implementation specific.

~ENHANCEDIFD()

Deletes an instance of the ENHANCEDIFD class. If hContext is valid, this method calls the Detach() method before destroying the object.

RESPONSECODE AttachByReader(
IN STR ReaderName
)

Opens a connection to the IFD Service Provider. The *ReaderName* parameter defines which reader the extended functions are relative to. The main purpose of this function is to perform a validity check and allocate the necessary communication resources.

This *Attach* function should be used when the extended function does not concern directly an operation with a smart card. Otherwise, the method *AttachByHandle* should be used.

RESPONSECODE AttachByHandle(
IN HANDLE hCard
)

Opens a connection to the IFD Service Provider. The hCard parameter fixes the reader to be used for the extended functions and more particularly the card inserted in the reader. This function should be used when the extended functions are in relation with the smart card inserted and when probably some communication with the smart card will occur to carry out the specific functions. This insures to the application that the smart card is not removed during the operation.

RESPONSECODE Detach()

Releases the communication context associated with the hContext property and resets hContext to NULL.

RESPONSECODE SelectContext(
IN ACID ApplicationContext //Unique Identifier of an Application Context
IN DWORD TimeOut //Time-out to get the context in miliseconds

)

Selects a given *ApplicationContext* for the current instance of ENHANCEDIFD. If the *ApplicationContext* is not supported by the IFD subsystem, a SCARD_E_UNSUPPORTED_CONTEXT error is returned. If some IFD resources are not available (for example taken by another application context opened), the method waits at most for *TimeOut* for allocation liberation.

Only one context can be selected at one time for the instance of ENHANCEDIFD.

RESPONSECODE ReleaseContext()

Deselects the current Application context. All the interface objects created within the previous Application Context are not anymore valid.

RESPONSECODE Cancel()

Cancels the current pending SelectContext request.

RESPONSECODE CreateSecurePIN(
OUT SecurePIN aSecurePIN
)

RESPONSECODE CreateDisplay(
OUT Display aDisplay
)

RESPONSECODE CreateUserConfirmation(
OUT UserConfirmation aUserConfirmation
)

RESPONSECODE CreateUserEntry(
OUT UserEntry aUserEntry
)

RESPONSECODE CreateInterface(
IN IID IFDSPInterface //Unique identifier of the interface object to be created
OUT VOID InterfaceObject
)

These five methods either return a SCARD_SUCCESS, meaning that the requested object was successfully created or a SCARD_E_UNSUPPORTED_INTERFACE, meaning that the requested interface is not supported by that Service Provider with the current Application context (an application context has to be selected previously).

4.1.8 Optional Interfaces

The following is a generic description of the optional interfaces defined for compliant IFD Service Providers. If an IFD Service Provider exposes secure PIN, display, user confirmation or user entry service, they should be exposed through these interfaces. If they do not support this functionality, then these interfaces need not be supported. Some other methods (and interfaces) may be specified in the future such as *GetOnlinePIN*, *SubmitBiometrics*, *AcquireBiometrics*, *GetOnlineBiometrics*.

In the following interfaces, a method may be unavailable for a certain application context due to security reasons for example. In this case, the SCARD_E_METHOD_NOT_ALLOWED error is returned by the method.

4.1.8.1 Class SECUREPIN

This class is intended to be an interface to the device's PIN entry functionality. The security level provided is a function of the IFD subsystem and IFDSP implementations. This class definition does not provide any additional security.

RESPONSECODE GetCapabilities(

OUT	DWORD PINAttributes	// PIN Attributes capabilities as defined by the PINATTRIBUTE constants (bit set to 1 when feature is supported)
OUT	USHORT Feedback	// Feedback Capability as defined by the FEEDBACKCAPABILITY constants
OUT	USHORT Event	// Event Capability as defined by the EVENTCAPABILITY constants
OUT	BOOL PINEntryNotification	// indicates the capability for an IFD to inform the user about PIN entry

)

Returns the capabilities of the SECUREPIN service as far as feedback from the PIN entry is concerned. For Feedback, only IFD or IFDSP value can be returned. If the IFD cannot handle the feedback, it is mandatory that the IFDSP provide a default feedback. It is up to the service caller to use it or not.

PINEntryNotification is TRUE if the IFD will indicate that the user has to enter a PIN during the *Verify* and *Change* methods. If PINEntryNotification is FALSE, the application has to handle the user notification, for example via the DISPLAY class.

RESPONSECODE SetFeedback(

IN	USHORT FeedbackMethod	// Defines the feedback method via the FEEDBACKCAPABILITY constants
IN	VOID EventCallback	// Callback Function Pointer with an event to indicate when a key is pressed.

)

If the feedback capability is IFD, only IFD feedback method can be used otherwise either NONE or IFDSP can be used. The EventCallback method is used for the feedback event notification according to the EVENTCAPABILITY of the IFD.

The prototype of the *EventCallback* method is:

RESPONSECODE EventCallback(

IN	PIN_INPUT_FEEDBACK Event // Event type

)

If EVENTCAPABILITY is BEGIN_END only KEY_VALIDATION and the KEY_FIRST are received for the 'KEY_…' constants. If EVENTCAPABILITY is KEYSTROKE, all 'KEY_…' are received except KEY_FIRST. If EVENTCAPABILITY is NONE, no PINFEEDBACK is received.

RESPONSECODE Verify(

```
IN      SCARD_IO_HEADER SendPci // Send protocol structure
IN      BYTE[] SendBuffer       // Predefined APDU in which the IFD fills in the APDU
IN      PINVERIFY PinFormat // Structure which specify how to format the PIN code.
IN      DWORD TimeOut       // time-out for the user entry in milliseconds
OUT     SCARD_IO_HEADER RecvPci // Receive protocol structure
OUT     BYTE[] RecvBuffer       // Data buffer for ResponseAPDU
OUT     DWORD RecvLength    // Length of RecvBuffer
```

)

This method handles the PIN entry, builds the PINVERIFY block and sends the block to the ICC. If the feedback handling is to be used, the *SetFeedback* method shall be called first.

The *Verify* method overwrites the data field of the APDU (after the bytes CLA, INS, P1, P2, Lc) with the PINVERIFY block formatted with the *PinFormat* rules.

For example, for a PIN value of 12345 to be inserted in the APDU '*00 20 00 00 09 66 FF FF FF FF FF FF FF FF*' with *PINBlockByteOffset* set to 1 byte and *PINBlock.PINBitOffset* set to 4 bits

- with *PINBlock.NbCharBitLength* = 0 bits
  - with *PINBlock.Attributes* set to (BCD | RIGHT_JUSTIFIED), the resulting APDU to be sent is '*00 20 00 00 09 66 FF FF FF FF FF 12 34 5F*'
  - with *PINBlock.Attributes* set to (BCD | LEFT_JUSTIFIED), the resulting APDU to be sent is '*00 20 00 00 09 66 F1 23 45 FF FF FF FF FF*'
- with *PINBlock.NbCharBitLength* = 4 bits and *PINBlock.NbCharBitOffset* = 4 bits
  - with *PINBlock.Attributes* set to (BCD | RIGHT_JUSTIFIED), the APDU to be sent is '*00 20 00 00 09 66 F5 FF FF FF FF 12 34 5F*'.
  - with *PINBlock.Attributes* set to (BCD | LEFT_JUSTIFIED), the APDU to be sent is '*00 20 00 00 09 66 F5 12 34 5F FF FF FF FF*'.

RESPONSECODE Change(

```
IN      SCARD_IO_HEADER SendPci // Send protocol structure
IN      BYTE[] SendBuffer       // Predefined APDU in which the IFD fills in the APDU
IN      PINCHANGE PinFormat         // Structure which specify how to format the PIN
                                    code.
IN      DWORD TimeOut       // time-out for the user entry in milliseconds
OUT     SCARD_IO_HEADER RecvPci // Receive protocol structure
OUT     BYTE[] RecvBuffer       // Data buffer for ResponseAPDU
OUT     DWORD RecvLength    // Length of RecvBuffer
```

)

Submits to the ICC a PIN code change, given the APDU to send to the smart card and the PIN format. For this operation, the current PIN may have to be entered. The formatting of the PIN is similar to the *Verify* method except that

two PIN blocks may be used for optionally the current PIN and for the new PIN. The feedback handling has to be set through the *SetFeedback* method.

The *Change* method overwrites the data field of the APDU with the PINCHANGE block formatted with the *PinFormat* rules.

RESPONSECODE Cancel()

Cancels the current pending *Verify* or *Change* request.

Warning: the IFD subsystem as defined in Part 3 should filter the APDU through a definition given in the Application Context in order to make sure that the APDU to be sent to the ICC is really a VERIFY APDU (determined by CLA and INS of the APDU) or CHANGE APDU command. If this is not done, there is a potential security risk where an application could use a WRITE BINARY APDU, instead, in order to write in clear the PIN in a free zone of the ICC and, then, with a READ BINARY it could be possible for the application to retrieve the user PIN. This is also the case for the Change method.

4.1.8.2 Class DISPLAY

Only basic character display is considered in this service, as it is the most common interoperable capability. Addressing in the virtual screen is used for devices with scrolling capabilities (virtual dimensions greater than physical dimensions). If a device does not support it, the physical and the virtual dimensions should be identical.

RESPONSECODE ClearDisplay()

Erases the virtual text screen content.

RESPONSECODE GetCharacterDisplayResolution(

OUT	USHORT nPhysicalColumnCount	// Number of columns for the physical screen
OUT	USHORT nPhysicalRowCount	// Number of rows for the physical screen
OUT	USHORT nVirtualColumnCount	// Number of columns for the virtual screen
OUT	USHORT nVirtualRowCount	// Number of rows for the virtual screen

)

A non-proportional font is assumed in this display service. If it is not the case, the screen size should be returned with the larger character taken into account.

RESPONSECODE DisplayMessage(

IN	STR Message	// Message in Unicode
IN	USHORT x	// x coordinate in the virtual screen
IN	USHORT y	// y coordinate in the virtual screen
IN	DWORD MinTime	// Minimum time for the message to be displayed in milliseconds

)

Automatic wrapping is not performed by this method. Scrolling if supported should be handled directly by the device (automatically or manually). The origin (0,0) is at the upper left.

4.1.8.3 Class USERCONFIRMATION

RESPONSECODE GetConfirmation(
IN DWORD TimeOut // Time-out for the user confirmation in milliseconds
OUT BOOL Confirmation // True or false
)

The IFD enters in a mode where the user has to validate or cancel via a validation or cancel key for example.

RESPONSECODE Cancel()

Cancels the current pending *GetConfirmation* request.

4.1.8.4 Class USERENTRY

Warning: there is a potential security hazard if the SECUREPIN interface and the USERENTRY are supported. Indeed, a fake PIN entry request might be made by an application through the USERENTRY interface.

RESPONSECODE GetString(
IN DWORD TimeOut // Time-out for the user entry in milliseconds
OUT STR UserData // Return from the User Entry in Unicode
)

The IFD enters in a mode where a user can enter some information via a keyboard for example. This operation may be ended by a validation key.

RESPONSECODE Cancel()

Cancels the current pending *GetString* request.

4.2 Examples

This paragraph gives an example for an ICC-aware application or an ICCSP of a typical use of an IFDSP.

An application A requires that an application context $ACID_{ac}$ be implemented in the IFD and the application needs to use the interface defined by the IID_i (this interface is used through the IFDSP).

The application can wait for a card in the slots which support this $ACID_{ac}$ (the list of the readers can be determined through the *ListReaders* and then *Control* with *IFD_List_Contexts* for each reader via the Resource Manager) or check when the card has been inserted if the slot can support the $ACID_{ac}$ (through *Control* with *IFD_List_Contexts* via the Resource Manager). In the same manner, through *Control* with *IFD_List_Interfaces* via the Resource Manager the application can check that the reader supports the interface IID_i.

Let's say that the card of A has been inserted in the right slot *ReaderName* and that a connection to the card has already been established (with the handle *hCard*) and that some APDUs have been transmitted to the card. Now, the application needs to use the generic interface IID_i:

```
ResourceManager.Control(GET_IFDSP, &IFDSP_ID) //get the GUID
                                  // of the principal IFDSP
IFDSP=CreateObject(IFDSP_ID)      //OS-Specific
IFDSP.AttachByHandle(hCard)
IFDSP.SelectContext(ACID_ac,timeout)
OBJ_i=IFDSP.CreateInterface(IID_i)  //OS-Specific
OBJ_i.Method1()                   // Method of the
                                  // interface GUID_i
(...)
IFDSP.Detach()
```

The following figure shows two possible implementations of an IFDSP. The first one is decomposed into two components: a Principal which is referenced by the Resource Manager under *IFDSP_ID* and which implements the interface ENHANCEDIFD, and the class $Class_i$ referenced by the $GUID_{IFDSP-i}$ which implements the interface IID_i. In the second version, there is only one global IFDSP which implements the interface ENHANCEDIFD and the interface IID_i. During the *IFDSP.CreateInterface()* in the second type of implementation, the object returned will be in fact the same object, i.e. *IFDSP*. For the first implementation, it is an instance of $Class_i$ which is returned. This way, via the *CreateInterface* method, the application can be really independent of the real structure and implementation of the IFDSP.

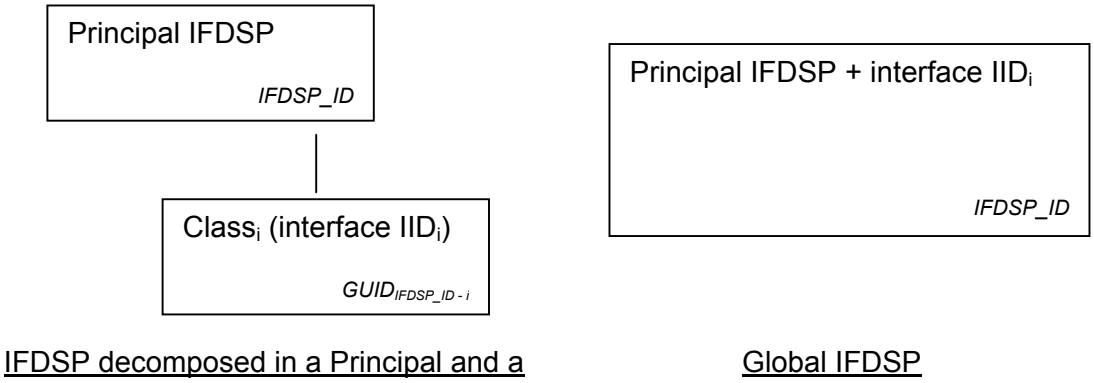


Figure 4-1 Examples of IFDSP Structure

5 GUID References

5.1 PC/SC Application Contexts

Four application context examples are described within this part 9:

PCSC_DISPLAY Application Context:

- The DISPLAY interface is required and at least the access to the display is reserved to the application.

PCSC_DISPLAY_USERENTRY Application Context:

- The DISPLAY and USERENTRY interface are required and at least the access to the display and keypad is reserved to the application.

PCSC_STANDARD Application Context:

- A secure keypad is required in order that the PIN be directly transmitted to the smartcard without leaving the IFD.
- The permitted PIN processing commands for the verify command and referenced by ISO 7816-4 are with an instruction byte 0x20 and the class byte is included in the ranges 0x00-0x0F, 0x80-0x8F, 0x90-0x9F, 0xA0-0xAF and 0xB0-0xCF.
- The SECUREPIN interface is required.
- A general user entry such as the USERENTRY interface is not allowed when this application context is selected. However, DISPLAY and USERCONFIRMATION interfaces are allowed. This is to prevent a fake PIN entry.

PCSC_ENHANCED Application Context:

- A secure keypad is required in order that the PIN be directly transmitted to the smartcard without leaving the IFD.
- The permitted PIN processing commands for the verify command and referenced by ISO 7816-4 are with an instruction byte 0x20 and the class byte is included in the ranges 0x00-0x0F, 0x80-0x8F, 0x90-0x9F, 0xA0-0xAF and 0xB0-0xCF.
- The SECUREPIN interface is required.
- It is not possible to use other interfaces, which involve “transparent use” of the keyboard and the display of the IFD, such as DISPLAY, USER CONFIRMATION or USERENTRY interfaces. This is to prevent fake PIN entry and fake information on display.

Application Context	ACID
PCSC_DISPLAY	`e2eb96e4-4984-11d3-83fc-0080c7e29271`
PCSC_DISPLAY_USERENTRY	`e3866666-4984-11d3-83fc-0080c7e29271`
PCSC_STANDARD	`e2ebeee0-4984-11d3-83fc-0080c7e29271`

	0080c7e29271
PCSC_ENHANCED	e3848560-4984-11d3-83fc-0080c7e29271

5.2 Interface IDs

These are the IIDs of the interfaces defined in this part 9.

Interface	IID
SECUREPIN	2d49d340-4984-11d3-83fc-0080c7e29271
DISPLAY	2eb1eec0-4984-11d3-83fc-0080c7e29271
USERCONFIRMATION	2f82fa60-4984-11d3-83fc-0080c7e29271
USERENTRY	30ba41e0-4984-11d3-83fc-0080c7e29271