Interoperability Specification for ICCs and Personal Computer Systems

Part 5. ICC Resource Manager Definition

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
2.00.4	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 defined by this specification is described in detail in Part 1 and is summarized below. This Part deals with one specific element of this architecture, the ICC Resource Manager, indicated by the shaded area of the figure.
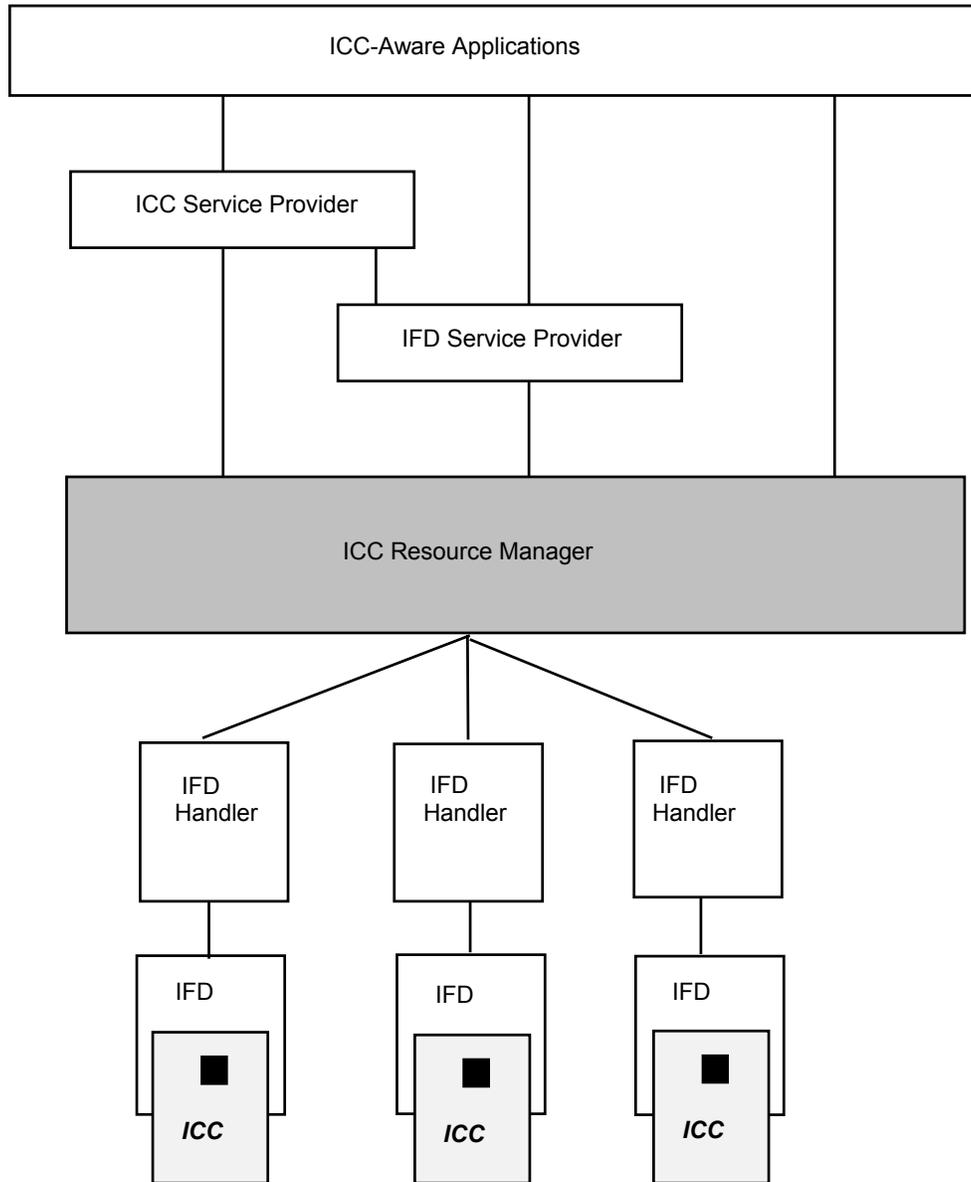

Figure 1-1. General Architecture

2   Theory of Operation

2.1   Functional Overview

The ICC Resource Manager is a key component of the PC/SC Workgroup's architecture. It is responsible for managing the other ICC-relevant resources within the system and for supporting controlled access to IFDs and, through them, individual ICCs. The ICC Resource Manager is assumed to be a system-level component of the architecture. It must be present and will most likely be provided by the operating system vendor. There should be only a single ICC Resource Manager within a given system.

The ICC Resource Manager solves three basic problems in managing access to multiple IFDs and ICCs.

First, it is responsible for identification and tracking of resources. This includes:
- Tracking installed IFDs and making this information accessible to other applications.
- Tracking known ICC types, along with their associated Service Providers and supported Interfaces, and making this information accessible to other applications.
- Tracking ICC insertion and removal events to maintain accurate information on available ICCs within the IFDs.

Second, it is responsible for controlling the allocation of IFD resources (and hence access to ICCs) across multiple applications. It does this by providing mechanisms for attaching to specific IFDs in shared or exclusive modes of operations.

Finally, it supports transaction primitives on access to services available within a given ICC. This is extremely important because current ICCs are single-threaded devices that often require execution of multiple commands to complete a single function. Transactions allow multiple commands to be executed without interruption, ensuring that intermediate state information is not corrupted.

2.2   Implementation Considerations

The ICC Resource Manager will typically be written by the system provider as part of an implementation of this specification targeted to a given environment (PC plus Operating System). It will be directly involved in all transactions between ICC-Aware Applications and ICCs. It is expected to gain direct control over all IFD Handlers, and hence over all IFDs, on a given system at system boot time. It is then responsible for controlling access to these resources by PC applications.

The ICC Resource Manager is a privileged component in the sense that it controls access to physical devices and is involved in the movement of commands and data between an application and an ICC. Hence, it is critically important that it be designed to ensure a logical separation between data streams associated with different processes. It should be implemented in a manner that maintains the same degree of security and protection afforded by the base Operating System. In short, addition of this component to an environment should not create new vulnerabilities.

2.3   User Interface Elements

In conjunction with the ICC Resource Manager, it is expected that certain User Interface (UI) elements will be provided. This ensures consistency within a given environment and avoids duplicative efforts by IFD and/or ICC vendors in defining and implementing this functionality.

First, the UI for controlling and managing IFD and ICC resources should be provided. This UI is intended to expose resource management services provided by the ICC Resource Manager for administrative control purposes. Through this UI, it shall be possible for the user to:

- Install and/or remove IFDs and ICC type definitions on the system.
- Enumerate installed IFDs and ICC types.
- Control properties associated with the IFDs:
  - User-defined "friendly" names
  - User-defined IFD "groups"
  - Access to IFD configuration tools (supplied by the vendor)
- Control properties associated with the ICC types.
  - User-defined "friendly" names
  - ATR strings and masks used to identify the ICC type
  - Associated Service Provider
  - Associated Interfaces

Second, the UI should be defined so as to allow the user to resolve potential runtime resource conflicts. Typically these will involve situations where the ICC Resource Manager is unable to uniquely determine the ICC type within a given IFD, or multiple ICC types fit within a given application-supplied set of criteria. In either case, it is likely the application will request that the user select from among the possible ICC types based on the ICC he is actually using, or intends to use. This resolution is necessary to ensure that the application selects the appropriate Service Provider to use.

This UI won't necessarily be implemented as a part of the ICC Resource Manager. An implementation as a "common dialog," or provision of a standard UI template, are both appropriate mechanisms for making this functionality available to application developers.

2.4   Installation and Configuration

As discussed previously, the ICC Resource Manager plays a key role in defining and enforcing logical separation between data associated with multiple applications. Hence, it must be installed with security characteristics commensurate with the role within the basic platform operating system. The ICC Resource Manager must have security privileges sufficient to control all IFD Handlers within the system and to restrict access by ICC Service Providers to specific IFD Handlers. It is highly desirable that it be installed in a manner that requires special privilege (for example, "administrator" access) to install, remove, or modify the executable image and associated data files.

For the ICC Resource Manager to do its job, it must be aware of the available IFD devices installed in the system. It is expected that each IFD will be introduced to the

Resource Manager as part of the installation process, creating the required entries in the Resource Manager data base.

Prior to an ICC becoming available through the ICC Resource Manager, it must first be introduced to the system. This will typically be done through an ICC setup utility provided by the ICC manufacturer. This utility may come with the ICC on a floppy disk, or may be available on a Web site, and so on. The setup utility must provide three pieces of information about the card:

- Its ATR string, and a mask, used as an aid in identifying the ICC
- A reference to the Service Provider associated with this ICC type
- A "friendly name" for the ICC, to be used in identifying the ICC to the user

2.5 Runtime Considerations

The Architecture defined in Part 1 provides several means by which an ICC-Aware Application or an ICC Service Provider can form a connection to an ICC:

- The Application may access whatever ICC is present within a given IFD. The ICC Resource Manager provides a service (SCARDCOMM.Connect()) to connect to whatever ICC resides in a given IFD. This is the simplest form of establishing communication with an ICC.
- The Application may search for a specific ICC inserted into a given IFD. When an Application wishes to interact with an ICC, it identifies the ICC by its friendly name, and specifies a collection of IFDs of interest. The ICC Resource Manager searches the IFDs for any ICCs matching the specified type(s) and returns status information to the Application. The ICC Resource Manager never interacts directly with an ICC beyond obtaining the ATR string; however, it supplies sufficient information for the Application to be able to walk the user through locating the ICC. This results in mapping the request to a specific IFD, to which further I/O is directed.
- The Application may search within a given collection of IFDs for any ICC supporting a given set of ICC Interfaces. This is similar to the previous case, but any named ICC that supports all listed interfaces is considered a match.

In attempting to locate an ICC type(s) within a collection of IFDs, there are several possible outcomes:

- No such IFD(s) is known to the ICC Resource Manger. This is an error condition.
- No such ICC type is known to the ICC Resource Manager. This is an error condition.
- No matching ICC is currently in any of the target IFDs, and none of the target IFDs are available to put it in.
- No matching ICC is currently in any of the target IFDs, but the returned subset of the target IFDs contains ICCs that are not in use, so the user may be prompted to remove any of them.
- No matching ICC is currently in any of the target IFDs, but the returned subset of the target IFDs does not contain ICCs, so the user may be prompted to insert the desired ICC into any of them.

If a matching ICCs is found in the returned subset of the target IFDs, then two possibilities exist. First, the ICC may resolve uniquely to a known ICC type in which case

the application knows which Service Provider to use in communicating with the ICC.. Second, the ICC may not be resolvable to a single ICC type. In this case, the Application may apply further filters in an attempt to determine the ICC type, or use the common UI (see Section 2.3) to prompt the user to select exactly which ICC is desired.

Note that an IFD may already be opened for exclusive use by another Application, so access to an IFD containing the ICC of interest is not guaranteed.

3 Functional Description

In this section, we describe the functional interface exposed by the ICC Resource Manager. This is described in terms of object Classes, and methods on object instances of those Classes, along with required parameters and expected return values. The interface definition is language and system independent. Implementations may alter the naming conventions and parameters as required to adapt to their environment. For example, some implementations in C may wish to use "pointer" data types for some parameters. Such implementation-level decisions are consistent with the intent of this specification provided that the required functionality is supported.

Note that the interface definitions use terminology that reflects common usage. For example, ICCs are generally referred to as "cards" and related constants are prefixed by "SCARD" (short for smart card). IFDs are referred to as "readers". This is a somewhat arbitrary decision, but was motivated by the desire to minimize potential symbol conflicts with deployed API functions. These conflicts are more likely if simple 3-letter acronyms, such as "ICC" are used.

3.1 Syntax

The syntax used in describing the ICC Resource Manager interface is based on common procedural language constructs. Data types are described in terms of pseudo types similar to C-language types, due to its widespread use. The following describes specific conventions and predefined values used in this document.

3.1.1 Data Types

The following data types are used in defining the Resource Manager interface:
- BYTE == unsigned char, an 8-bit value
- USHORT == unsigned short, a 16-bit value
- BOOL == short, signed 16-bit value
- DWORD == unsigned long, a 32-bit value
- STR == char array (string)
- GUID == unsigned char[16], a 128-bit unique identifier
- RESPONSECODE == long , signed 32-bit value
- HANDLE == a 32-bit value
- VOID == unspecified data type whose interpretation is context dependent

Objects which are handling arrays of the basic data types are indicated by "[]". Those objects in general provide methods to query the length of the included array. For example, "BYTE[]" indicates an object which includes an array of BYTE values of unspecified length. "BYTE[4]" indicates an object with an array of BYTE values with the length four. The goal behind this was to give the possibility to use this specification also for programming languages which are handling data generally as objects like Java.

Data structures are indicated using the "structure" type definition. The following defines a data structure consisting of a BYTE and DWORD value, which is referenced using the SAMPLE_STRUCT identifier.

```
structure {
    BYTE        ByteValue
    DWORD  DwordValue
}  SAMPLE_STRUCT;
```

3.1.2 Calling Conventions

The interface to the ICC Resource Manager is defined in terms of methods associated with the exposed objects. Methods are invoked by referencing a named method within the context of an object instance. How the object is referenced is not specified as this may vary by implementation. Methods require zero or more parameters and return information using a simple data type and, optionally, output parameters. For example:

RESPONSECODE MethodA(
    IN   DWORD DwordValue
    IN OUT BYTE ByteValue
    OUT   BYTE OutValue
)

This example defines a method with three parameters that returns a RESPONSECODE value. It has two input parameters (DwordValue and ByteValue) and returns additional information in two output parameters (ByteValue and OutValue).

3.1.3 Defined Constants

The following constants are defined. For the purposes of defining the ICC Resource Manager interface, these are defined by symbol only. It is the responsibility of the implementers to assign specific manifest constants suitable for use within their environment. These should all be defined as DWORD quantities.

Parameter	Symbol	Comments
Access Mode Flags		Used to indicate mode of access to a card.
	SCARD_SHARE_SHARED	Application is willing to share access to card with other applications.
	SCARD_SHARE_EXCLUSIVE	Application requires exclusive access to the card.
	SCARD_DIRECT	Application requires connection to reader whether or not card is present. Implies exclusive access.
Protocol Identifier		These define the protocols used in communication with the card. These must be defined such that a protocol maps to a specific bit position so that multiple protocols may be specified by combining them with a bitwise OR operation.

Parameter	Symbol	Comments
	SCARD_PROTOCOL_UNDEFINED	
	SCARD_PROTOCOL_DEFAULT	Provides hint to reader that it should use default communication parameters to establish communication with the card.
	SCARD_PROTOCOL_OPTIMAL	Provides hint to reader that it should attempt to negotiate optimal communications settings with the card.
	SCARD_PROTOCOL_T0	ISO/IEC 7186 T=0 protocol.
	SCARD_PROTOCOL_T1	ISO/IEC 7816 T=1 protocol.
	SCARD_PROTOCOL_RAW	
Card Disposition		Used to indicate the desired disposition of the card following a Transaction or when a connection is terminated.
	SCARD_LEAVE_CARD	Don't alter card state.
	SCARD_RESET_CARD	Reset the card.
	SCARD_UNPOWER_CARD	Unpower and terminate access to the card.
	SCARD_EJECT_CARD	Eject the card from the reader.
	SCARD_CONFISCATE_CARD	Used to indicate that a sophisticated commercial reader should move the card to the confiscation bin and not return it to the user.
Card/Reader State		Used to indicate the state of the card in the reader and the current protocol status of the card. For the latter, this is used to specify whether the card is willing to negotiate a new protocol or not as defined in ISO/IEC 7816.
	SCARD_ABSENT	No card is in the reader.
	SCARD_PRESENT	A card is in the reader.
	SCARD_SWALLOWED	A card is in the reader and is properly positioned for operation.
	SCARD_POWERED	A card is in the reader and has been powered.
	SCARD_NEGOTIABLEMODE	The card is capable of negotiating

Parameter	Symbol	Comments
		a new protocol setting.
	SCARD_SPECIFICMODE	The card is in a specific protocol mode and a new protocol may not be negotiated.
Context Scope		Used to designate the scope of access desired within a given Resource Manager communication context.
	SCARD_SCOPE_USER	Operate within the user's scope.
	SCARD_SCOPE_TERMINAL	Operate within the scope of a terminal.
	SCARD_SCOPE_SYSTEM	Operate within the system-wide scope.

3.1.4 Response Codes

The following table lists the defined RESPONSECODE data types. For the purposes of defining the Resource Manager interface, these are defined by symbol only. It is the responsibility of the implementers to assign specific manifest constants suitable for use within their environment.

A RESPONSECODE is returned by all methods associated with the ICC Resource Manager. The returned value provides a primary notification of success or failure at the attempt to execute the requested operation. These codes do not replace or supplement the card command response codes defined by ISO/IEC 7816. Those codes are passed through to the application by the protocol methods.

Symbol	Meaning
SCARD_S_SUCCESS	No error was encountered.
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_VALUE	One or more of the supplied parameters' values could not be properly interpreted.
SCARD_E_CANCELLED	The action was cancelled by an SCARDTRACK::Cancel or SCARDCOMM::Cancel request
SCARD_E_NO_MEMORY	Not enough memory available to complete this command
SCARD_E_INSUFFICIENT_BUFFER	The data buffer to receive returned data is too

Symbol	Meaning
	small for the returned data
SCARD_E_UNKNOWN_READER	The specified reader name is not recognized
SCARD_E_TIMEOUT	The specified timeout value has expired
SCARD_E_SHARING_VIOLATION	The ICC cannot be accessed because of other connections outstanding
SCARD_E_NO_SMARTCARD	The operation requires an ICC, but no ICC is currently in the device
SCARD_E_UNKNOWN_CARD	The specified ICC name is not recognized
SCARD_E_PROTO_MISMATCH	The requested protocols are incompatible with the protocol currently in use with the ICC
SCARD_E_NOT_READY	The IFD or ICC is not ready to accept commands
SCARD_E_SYSTEM_CANCELLED	The action was cancelled by the system, presumably to log off or shut down
SCARD_E_NOT_TRANSACTED	An attempt was made to end a non-existent transaction
SCARD_E_READER_UNAVAILABLE	The specified IFD is not currently available for use
SCARD_W_UNSUPPORTED_CARD	The reader cannot communicate with the card, due to ATR configuration conflicts. This error may be cleared by the SCARDCOMM::Reconnect service.
SCARD_W_UNRESPONSIVE_CARD	The card is not responding to a reset. This error may be cleared by the SCARDCOMM::Reconnect service.
SCARD_W_UNPOWERED_CARD	Power has been removed from the card, so that further communication is not possible. This error may be cleared by the SCARDCOMM::Reconnect service.
SCARD_W_RESET_CARD	The card has been reset, so any shared state information is invalid. This error may be cleared by the SCARDCOMM::Reconnect service.
SCARD_W_REMOVED_CARD	The ICC has been removed, so that further communication is not possible
Additional codes may be added in future revisions	

3.2   Required Services

The following is a generic description of the services supported by the ICC Resource Manager. The purpose of each service is defined, along with required parameters and return codes. This description is suitable for implementation in a variety of languages on a variety of systems.

3.2.1   Class RESOURCEMANAGER

Communication with the ICC Resource Manager may occur only within a well-defined Context. A RESOURCEMANAGER object provides the methods necessary to create and manage this Context. This Context may be released at any time. However, implementations shall automatically release the Context on object destruction if not previously released.

A Context is referenced by a 32-bit handle, *hContext*. This handle shall be set to NULL to indicate the absence of a Context. It shall be an implementation-defined, 32-bit, non-NULL quantity when referencing a valid Context.

3.2.1.1   Properties

    HANDLE hContext   // Handle to a communications Context associated with
                                  // an SCARDMANAGER object instance (see below).
                                  // Set to NULL on object creation.

3.2.1.2   Methods

    RESOURCEMANAGER()

    This creates an instance of the RESOURCEMANAGER class and returns a reference to the calling application. The type of this object reference is implementation dependent.

    ~RESOURCEMANAGER ()

    This deletes an instance of the RESOURCEMANAGER. If the *hContext* is valid, then this method shall call *ReleaseContext()* prior to destroying the object.

    RESPONSECODE EstablishContext (
    IN   DWORD Scope,   // A scope indicator (see below)
    IN   DWORD Reserved1, // Reserved for future use to allow privileged
                            // administrative programs to act on behalf of
                            // another user
    IN   DWORD Reserved2 // Reserved for future use to allow privileged
                              // administrative programs to act on behalf of
                            // another terminal
)

    Creates Context to be used in subsequent communication with the ICC Resource Manager. If successful, a handle to the Context is stored in the *hContext* property (32-bit value of type HANDLE) of the SCARDMANAGER object.

The Scope parameter is intended to allow a caller to designate the "security" context that this Resource Manager Context operates within. Normally, this is associated with the user on whose behalf the calling process is running, and operations will be restricted to devices that the user is allowed to access. Alternately, one could request the security context appropriate for a specific terminal or the system as a whole.

RESPONSECODE ReleaseContext()

This method releases the Context associated with the *hContext* property and sets *hContext* to NULL. An error is returned if *hContext* is invalid.

3.2.2 Class RESOURCEDB

A principal responsibility of the ICC Resource Manager is maintenance of global information on the ICCSPs, cards, and readers known to the system. It is responsible for maintaining a database of this information. A RESOURCEDB object provides methods for managing this database of information including the ability to insert, delete, create associations between resources, and query the database. This database is considered a private information repository for the Resource Manager and may be implemented in any convenient manner. In general, this object would be accessed only by system administrative routines that manage the addition or removal of resources.

RESOURCEDB is derived from RESOURCEDBQUERY, (described in the next section,) and inherits its methods. These define the query interface to the resource database.

3.2.2.1 Properties

private RESOURCEMANAGER resmgr   // Reference to a
                                       // RESOURCEMANAGER object

3.2.2.2 Methods

RESOURCEDB(
    IN   RESOURCEMANAGER resmgr
)

This creates an instance of the RESOURCEDB class and returns a reference to the calling application. An object instance will be created only if a reference to a valid RESOURCEMANAGER object is provided.

~RESOURCEDB()

This deletes an instance of the RESOURCEDB. The object referenced by the *resmgr* property is unaffected.

RESPONSECODE IntroduceReader(
    IN   STR ReaderName   // Friendly name to be associated with the
                            // reader
    IN   STR DeviceName   // System-specific name of the reader device.
                            // Should conform to OS-specific naming
                            // conventions.
)

Adds a new card reader to the Resource Manager database, associating it with the given *ReaderName* and a system-specific *DeviceName*. In general, this function would be used by the set-up program for the associated reader device driver. Any attempts to define duplicate reader names are detected and result in an error.

RESPONSECODE ForgetReader(
    IN   STR ReaderName   // Friendly name associated with reader
)

Removes the reader from the Resource Manager database. The reader is also removed from any groups to which it is currently assigned. If the reader isn't found, this returns an error.

RESPONSECODE IntroduceReaderGroup(
    IN   STR GroupName   // Group name to be added to the
                                 // Resource Manager database
)

Supports defining a new card reader group within the Resource Manager database. Any attempts to define duplicate group names are detected and result in an error. This function is expected to be called only from system administrative routines.

RESPONSECODE ForgetReaderGroup(
    IN   STR GroupName   // Group name to be removed from the Resource
                                 // Manager database
)

Supports removing a card reader group within the resource database. All readers within the group are removed from the group first, and then the group removed. This function is expected to be called only from system administrative routines.

RESPONSECODE AddReaderToGroup(
    IN   STR ReaderName   // Name of Reader
    IN   STR GroupName   // Group name to which Reader is to be added
)

Assigns a reader to a reader group. If the group or the reader isn't defined in the Resource Manager database, this returns an error.

RESPONSECODE RemoveReaderFromGroup(
    IN   STR ReaderName   // Name of Reader
    IN   STR GroupName   // Group name from which Reader is to be
                                 // removed
)

Removes a reader from a group. If the reader or group isn't defined in the Resource Manager database, this returns and error.

RESPONSECODE IntroduceCardType(
IN	STR CardName	// Friendly name for the card being introduced
IN	BYTE[] ATRRefVal	// ATR reference value to match in determining card // type
IN	BYTE[] ATRMask	// Mask which is logically AND'd with an card ATR // prior to comparison with the ATR reference value
IN	BYTE[] ProviderId	// Unique identifier for the primary service provider, // encoding is system specific
IN	GUID[] Interfaces	// List of unique identifiers for the card Interfaces // supported by the card

)

Adds a new card type to the Resource Manager database along with an identifying ATR reference value and associated Mask; a reference to the ICCSP(s); and associated card interface identifiers. This service checks for duplicate card names and returns an error if a duplicate is detected. In general, this function would be used by the set-up program for the associated ICCSP.

RESPONSECODE ForgetCardType(
IN	STR CardName	// Friendly name for the card to be removed

)

Removes information on a specific card type from the Resource Manager database. This service does not remove the associated ICCSP. If the *CardName* is not known within the Resource Manager database, this returns an error.

3.2.3 Class RESOURCEQUERY

A RESOURCEQUERY object supports retrieval of resource information maintained by the ICC Resource Manager including groups, readers, and card types. For card types, information on associated service providers (ICCSPs) and supported interfaces may also be retrieved. This information is based on the contents of the Resource Manager database and is guaranteed to be correct only at the time the query is made.

3.2.3.1 Properties

private RESOURCEMANAGER resmgr   // Reference to a
                                                                  // RESOURCEMANAGER object

3.2.3.2 Methods

RESOURCEQUERY(
IN	RESOURCEMANAGER resmgr	

)

This creates an instance of the RESOURCEQUERY class and returns a reference to the calling application. An object instance will be created only if a reference to a valid RESOURCEMANAGER object is provided.

~RESOURCEQUERY()

This deletes an instance of the RESOURCEQUERY. The object referenced by the *resmgr* property is unaffected.

RESPONSECODE ListReaderGroups(
OUT   STR[] Groups   // Array of strings containing the Group names
)

Returns a list of the group names known to the system, that is, defined in the Resource Manager database.

RESPONSECODE ListReaders(
IN   STR[] Groups   // Array of strings containing Group names of
    // interest
OUT   STR[] Readers   // Array of strings containing Readers within the
    // Groups
)

Returns a list of the readers assigned to one or more groups. Any invalid group names are ignored.

RESPONSECODE ListCardTypes(
IN   BYTE[] ATR   // ATR string to compare against known card types;
    // maybe NULL
IN   GUID[] Interfaces   // Array of GUIDs associated with the desired
    // interfaces
OUT   STR[] Cards   // Array of strings containing Card Types
)

Returns a list of card types that match the supplied ATR string or interface list. The parameter *ATR* may be NULL and/or the interface list may be empty. In this event all cards match those criteria. Matching for the ATR is determined by examining the Resource Manager database record for each card type and performing a BYTE-wise comparison of the card type ATR against the supplied *ATR* AND'd with the card type mask value. If the supplied *ATR* is the NULL string, then all card types will match. A card matching the interface list must support all supplied *Interfaces*.

RESPONSECODE GetProviderId(
IN   STR CardName   // Friendly name for a Card Type
OUT   BYTE[] ProviderId   // Reference to the Primary SP(s)
)

Returns a reference to the Primary ICCSP associated with a given card type. If the *CardName* is not known within the Resource Manager database, this returns an error.

RESPONSECODE ListInterfaces(
```
IN      STR CardName      // Friendly name for a Card Type
OUT     GUID[] Interfaces  // Array of GUIDs associated with the supported
                           // Interfaces
)
```

Returns a list of the GUIDs associated with the card Interfaces supported by a given card type. If the *CardName* is not known within the Resource Manager database, this returns an error.

3.2.4 Class SCARDTRACK

This object encapsulates functionality that supports determination of the presence or absence of specific Card Types within the available Readers. This information is made available based on selection criteria provided by the calling application.

The SCARDTRACK methods use the following common data structure as a parameter:

```
structure {
    STR     Reader;          // reader name
    VOID    UserData;        // user defined data
    DWORD       CurrentState; // current state of reader at time of call
    DWORD       EventState;   // state of reader after state change
} SCARD_READERSTATE;
```

Where the fields have the following meanings:

Reader Supplies the friendly name of the name of the reader being monitored.

UserData This field is reserved for arbitrary application-supplied data.

CurrentState This field supplies the current state of the reader, based on the calling application's knowledge. This field can take on any of the following values, or in combination, as a bit mask:

SCARD_STATE_UNAWARE The application is unaware of the current state, and would like to know. The use of this value results in an immediate return from state transition-monitoring services. This is represented by all bits set to zero.

SCARD_STATE_IGNORE The application is not interested in this reader, and it should not be considered during monitoring operations. If this bit value is set, all other bits are ignored.

SCARD_STATE_UNAVAILABLE The application believes that this reader is not available for use. If this bit is set, then all the following bits are ignored.

SCARD_STATE_EMPTY The application believes that there is not a card in the reader. If this bit is set, all the following bits are ignored.

SCARD_STATE_PRESENT The application believes that there is a card in the reader.

SCARD_STATE_ATRMATCH The application believes that there is a card in the reader with an ATR matching one of the target cards. If this bit is set, *SCARD_STATE_PRESENT* is assumed.

SCARD_STATE_EXCLUSIVE The application believes that the card in the reader is allocated for exclusive use by another application. If this bit is set, *SCARD_STATE_PRESENT* is assumed.

SCARD_STATE_INUSE The application believes that the card in the reader is in use by one or more other applications, but may be connected to in shared mode. If this bit is set, *SCARD_STATE_PRESENT* is assumed.

EventState This field receives the current state of the reader, as determined by the Resource Manager. This field can take on any of the following values, or in combination, as a bit mask:

SCARD_STATE_IGNORE The application requested that this reader be ignored. No other bits will be set.

SCARD_STATE_CHANGED This implies that there is a difference between the state input by the calling application, and the current state. When this bit is set, the application may assume a significant state change has occurred on this reader.

SCARD_STATE_UNKNOWN This implies that the given reader name is not recognized by the Resource Manager. If this bit is set, then *SCARD_STATE_CHANGED* will also be set.

SCARD_STATE_UNAVAILABLE This implies that the actual state of this reader is not available. If this bit is set, then all the following bits are clear.

SCARD_STATE_EMPTY This implies that there is no card in the reader. If this bit is set, all the following bits will be clear.

SCARD_STATE_PRESENT This implies that there is a card in the reader.

SCARD_STATE_ATRMATCH This implies that there is a card in the reader with an ATR matching one of the target cards. If this bit is set, *SCARD_STATE_PRESENT* will also be set. This bit is returned only by the *LocateCard()* method.

SCARD_STATE_EXCLUSIVE This implies that the card in the reader is allocated for exclusive use by another application. If this bit is set, *SCARD_STATE_PRESENT* will also be set.

> **SCARD_STATE_INUSE** This implies that the card in the reader is in use by one or more other applications, but may be connected to in shared mode. If this bit is set, *SCARD_STATE_PRESENT* will also be set.

3.2.4.1 Properties

private RESOURCEMANAGER resmgr   // Reference to a
                                            // RESOURCEMANAGER object

3.2.4.2 Methods

SCARDTRACK(
    IN   RESOURCEMANAGER resmgr
)

This creates an instance of the SCARDTRACK class and returns a reference to the calling application. An object instance will be created only if a reference to a valid RESOURCEMANAGER object is provided..

~SCARDTRACK()

This deletes an instance of SCARDTRACK. The object referenced by the *resmgr* property is unaffected.

RESPONSECODE LocateCards(
    IN   STR[] Cards   // Array of strings giving the names of the Card
                                     // Types of interest
    IN OUT SCARD_READERSTATE[] ReaderStates // Array of READERSTATE
                                     // structures for readers of interest
)

Returns immediately with information concerning the status of the reader(s) specified in the *SCARD_READERSTATE* data structures. The value of the *EventState* parameter for each reader indicates whether a card matching one of the card types indicated in the first parameter is present. To block pending insertion of the desired card type(s), use the *GetStatusChange()* method.

Unknown card types are ignored. If an unknown reader is specified, then an error is returned.

RESPONSECODE GetStatusChange(
    IN OUT SCARD_READERSTATE[] ReaderStates // Array of READERSTATE
                                     // structures for readers of interest
    IN   DWORD Timeout   // Time-out value in milliseconds
)

The method will block until there is a status change in one of the Readers specified in the *SCARD_READERSTATE* data structures or the specified time-out period expires. A *Timeout* value of INFINITE (which is defined system dependent) is used to indicate that the calling application is willing to wait forever. A *Timeout* value of zero is used to indicate that the method should return immediately. Once this returns, the application can

determine which readers have undergone a state change, and the new state, by examining the *SCARD_READERSTATE* structures. If an unknown reader is specified, then an error is returned.

Note that this method will provide information on when cards are removed or inserted into one of the specified readers. It does not indicate anything about card types that may be in a specific reader. This behavior ensures that the calling application is made aware of all changes across the readers of interest, making it possible to present appropriate UI.

RESPONSECODE Cancel()

The method provides a means to terminate outstanding actions (blocked requests) within this Object context. Within this version of the interface, only the *GetStatusChange()* method may be cancelled.

3.2.5 Class SCARDCOMM

This object encapsulates a communication interface to a specific card or reader. It provides methods for managing the connections, controlling transactions, sending and receiving commands, and determining card state information.

The command interfaces are designed to simplify interaction with a card using the protocols required by this specification. In particular, this document describes the usage of the protocols ISO/IEC T=0 and T=1. In addition, a "raw" mode is supported, which may be used to support arbitrary data exchange protocols (such as T=14) for special-purpose requirements.

3.2.5.1 Properties

private RESOURCEMANAGER resmgr // Reference to a RESOURCEMANAGER
                                // object

private HANDLE hCard   // Handle to a Context associated with communication
                           // to a specific card and/or Reader. Set to NULL on
                           // object creation. This context is established using the
                           // Connect()method and destroyed using Disconnect().

3.2.5.2 Methods

SCARDCOMM(
    RESOURCEMANAGER resmgr
)

This creates an instance of the SCARDCOMM class and returns a reference to the calling application. A valid RESOURCEMANAGER object reference must be supplied upon creation. Subsequent connections to a specific card or reader require a valid Context.

~SCARDCOMM()

This deletes an instance of SCARDCOMM. If the *hCard* is valid, then this method shall call Disconnect() prior to destroying the object. The object referenced by *resmgr* is not affected by this operation.

RESPONSECODE Connect(
```
IN      STR ReaderName    // Friendly name for a Reader
IN      DWORD Flags        // Desired access mode information
IN      DWORD PreferredProtocols // Card communications protocols that may
                              be used
OUT     DWORD ActiveProtocol     // Protocol actually in use
)
```

Opens a connection to the card located in the reader identified by the *ReaderName* parameter. This connection is established in the context of an existing SCARDMANAGER object communication context pointed to by the *hContext* property. If *hContext* is not valid, then an error is returned.

The *Flags* parameter is used to indicate three things. First, it indicates whether the connection is opened for shared or exclusive access. If the requested mode is unavailable, an error is returned. Second, it indicates whether the caller desires a "direct" connection to the reader. Direct mode implies that a connection will be established, even if a card is not present. Finally, it is a hint to the reader indicating whether it should use default reader-card communication settings to attempt to optimize those settings if the card is still in negotiable mode as defined by ISO/IEC 7816.

Communication with the card will be initialized only for one of the valid protocols identified by the *PreferredProtocol* bit mask. If none of the requested protocols is available, then an error is returned. The actual protocol in use is returned in the *ActiveProtocol* parameter.

If this method is successful, a communication context between the application and the card is created. A reference to this context is set in the object's *hCard* property. This is considered a "general" context and may be used for all methods in this Object. If this method fails, *hCard* will be NULL.

RESPONSECODE Reconnect(
```
IN     DWORD Flags                // desired access mode
IN     DWORD PreferredProtocols // card communications protocols
                               // which may be used
IN     DWORD Initialization       // Specify card initialization to be performed
OUT    DWORD ActiveProtocol       // protocol actually in use
)
```

Reopens a connection to the card associated with a valid context referenced by the *hCard* property (must have been created set by calling the *Connect()* method). If *hCard* is invalid, then an error is returned.

This method may be used to clear an error condition preventing access to the card. For example, if the card is reset by another application, then

further attempts to execute methods against that card will fail with a warning being returned. This method allows the application to acknowledge the reset notification and continue operation.

The *Initialization* parameter allows to specify a desired action on the card as a side effect of the Reconnect. This may include performing a warm or cold reset by specifying either the SCARD_RESET_CARD or SCARD_UNPOWER_CARD standard disposition codes. This parameter is ignored if the Reconnect is applied to a connection that would normally return one of the return codes SCARD_W_RESET_CARD, SCARD_W_REMOVED_CARD and may be others dependent of the target platform. This must be done to prevent race conditions in applications.

It may also use this method to change the current access modes via the *Flags* parameter.

RESPONSECODE Disconnect(
```
IN      DWORD Disposition        // Desired Card disposition action
)
```

Disconnects from the card associated with the *hCard* property. If *hCard* is invalid, then an error is returned. The *Disposition* parameter indicates the desired action to perform on disconnect and includes the ability to:

- Leave the Card
- Reset the card.
- Power down and close the card.
- Eject the card.
- Confiscate the card

Any application may reset the card, even in shared access mode.

RESPONSECODE Status(
```
OUT     STR[] Reader             // Friendly name of the connected reader
OUT     DWORD State              // Current status the connection
OUT     DWORD ActiveProtocol     // protocol actually in use
OUT     BYTE Atr[]               // ATR data buffer
)
```

Returns the status from the connection associated with the *hCard* property. If *hCard* is invalid, then an error is returned. The parameter *Reader* returns the friendly names of the connected reader. The *State* parameter returns the current reader state. If there is a card inserted, the parameter *ActiveProtocol* returns the negotiated communication protocol and the parameter *Atr* returns the ATR string from the card.

The State parameter can take on any of the following values:

> **SCARD_UNKNOWN** - This value implies the driver is unaware of the current state of the reader. This also implies that the reader handler is not able to communicate with the corresponding reader.

SCARD_ABSENT - This value implies there is no card in the reader.

SCARD_PRESENT - This value implies there is a card present in the reader, but that it has not been moved into position for use.

SCARD_SWALLOWED - This value implies there is a card in the reader in position for use. The card is not powered.

SCARD_POWERED - This value implies there is power being provided to the card, but the reader handler is unaware of the mode of the card.

SCARD_NEGOTIABLE - This value implies the card has been reset and is awaiting PTS negotiation.

SCARD_SPECIFIC - This value implies the card has been reset and specific communication protocols have been established.

RESPONSECODE BeginTransaction()

This method initiates a logical transaction against the card associated with the context referenced by *hCard*. If *hCard* is invalid, this method will return an error. If successful, this blocks other applications from accessing the card, allowing the calling application to perform a sequence of operations against the card with assurance that any intermediate state information will remain valid.

At present, only a single outstanding transaction may exist for a given card. If a transaction is underway when an application calls this service, then this method will block pending completion of the prior transaction(s). Application requests for transactions are serviced on a first-in, first-out basis. The application may call the *Cancel()* method to terminate a pending request.

RESPONSECODE EndTransaction(
IN   DWORD Disposition   // Desired Card disposition action
)

This method ends a logical Transaction sequence against the card associated with the context referenced by *hCard*. If *hCard* is invalid, this method will return an error. If successful, this frees the transaction and the next pending transaction, if any, is serviced.

Upon termination, the action indicated by *Disposition* is performed if possible. This may be:
- Leave the card
- Reset the card.
- Power down and close the card.
- Eject the card.

Any application may reset the card, even in shared access mode. Power down and eject actions will succeed only if the application has exclusive access to the card.

RESPONSECODE Cancel()

The method provides a means to terminate outstanding actions (blocked requests) within this object context. Within this version of the interface, only the *BeginTransaction()* method may be cancelled.

RESPONSECODE Transmit(

```
IN       SCARD_IO_HEADER SendPci        // Send protocol structure
IN       BYTE[] SendBuffer              // Data buffer for send data
IN OUT  SCARD_IO_HEADER RecvPci         // Receive protocol structure
IN OUT  BYTE[] RecvBuffer               // Data buffer for receive data
OUT      DWORD RecvLength               // Length of received data
)
```

This method sends data to the card, associated with the context referenced by *hCard* and expects to receive data back from the card. The protocol which should be used therefore is encoded in the SCARD_IO_HEADER structure.

The data buffer for the returned data must be adequate to hold the maximum amount of data that may be returned. This service may return fewer than the expected number of bytes if the InterByte or InterBlock time-outs are exceeded. The actual number of bytes returned is indicated by the *RecvLength* parameter.

The protocol type and other required information is passed to the *Transmit()* method using an SCARD_IO_HEADER structure. The definition of this structure depends on the protocol type, which is always encoded at the beginning of the structure.

The SCARD_IO_HEADER structure is defined as:

```
structure {
    DWORD                    Protocol;
    DWORD                    Length;
} SCARD_IO_HEADER;
```

Where:

Protocol - Identifies the protocol in use based on the defined protocol constants (see 3.1.3).

Length - Contains the length of the tSCARD_IO_HEADER structure.

RESPONSECODE Control(
IN	DWORD ControlCode	// Vendor-defined control code
IN	BYTE[] InBuffer	// Input data buffer
IN OUT	BYTE[] OutBuffer	// Output data buffer
OUT	DWORD OutBufferLength	// Length of data in output data buffer

)

This method supports direct communication with the Reader device. Its primary intent is to provide a mechanism to communicate with vendor-defined features. It is the responsibility of the vendor to define *ControlCode* values and input/output data associated with these features.

The *ControlCode*, *InBuffer* data and *OutBuffer* are send directly to the reader device driver. The response from the device driver is returned in *OutBuffer* and the valid data size indicated in *OutBufferLength*.

RESPONSECODE GetReaderCapabilities (
IN OUT	DWORD Tag	// Value of tag associated with attribute to retrieve
OUT	BYTE[] Buffer	// Data returned

)

This method returns a reader attribute value associated with the supplied *Tag* parameter (see Part 3 for defined values). If the tag is unknown, an error is returned.

RESPONSECODE SetReaderCapabilities (
IN	DWORD Tag	// Value of tag associated with attribute to retrieve
IN	BYTE[] Buffer	// Data returned

)

This method sets a reader attribute value. The attribute to be set is identified by the *Tag* parameter (see Part 3 for defined values) and the value to set is passed in *Buffer*. If the attribute does not exist or cannot be set, or the provided value is illegal an error is returned.