TCG Storage Architecture Core Specification

Specification Version 2.01
Revision 1.00

August 5, 2015

Contact: admin@trustedcomputinggroup.org

TCG

PUBLISHED

Disclaimers, Notices, and License Terms

Revision History

Version	Date	Description
Version 1.00, Revision 0.9	24 May 2007	First published Draft
Version 2.00, Revision 1.00	20 April 2009	First Final publication
Version 2.00, Revision 2.00	4 November 2011	Numerous clarifications of behavior to support other released Storage specifications.
Version 2.01, Revision 1.00	5 August 2015	Removed section 3.3.7.3.2, minor editorial clarifications in section 3.3.7.3, and updated reference [2] to point to latest specification version 1.04.

Figures

Tables

1 Introduction

1.1 Scope and Audience

Begin Informative Content

The TCG Storage specifications are intended to provide a comprehensive architecture for putting selected features of Storage Devices under policy-driven access control. The capabilities of the Storage Device are able to be configured to conform to the policies of the trusted platform. The controlled features include access to secure storage areas and the life cycle state of the Storage Device as a Trusted Peripheral (TPer). This document also serves as a specification for TPers where that is deemed appropriate.

The intended audience for this document is Storage Device manufacturers and developers that wish to tie trusted Storage Devices into trusted platforms.

End Informative Content

1.2 Key Words

Key words are used to signify the requirements in the specification. The key words “SHALL,” "SHALL NOT," “SHOULD,” "SHOULD NOT," “MAY,” and “OPTIONAL” are used in this document. These key words are to be interpreted as described in [8]

The key word "OBSOLETE" is used to indicate that the designated methods, tables, or values that may have been defined in previous standards are not defined in this standard and SHALL NOT be reclaimed for other uses in future standards. However, some degree of functionality may be required for items designated as OBSOLETE to provide for backward compatibility.

Invocation of methods defined as OBSOLETE may result in an error status method response returned by devices conforming to this specification.

Tables and values defined as OBSOLETE may result in an error status method response returned by devices conforming to this specification when attempts to reference those tables or values are made.

Bits, bytes, fields, and values identified as “Reserved” are set aside for future standardization. Their use and interpretation MAY be specified by future versions of this or other standards. A reserved bit, byte, field, or value SHALL be cleared to zero where applicable, or in accordance with a future version of this standard. A communicator SHALL NOT check reserved bits, bytes, fields, or values.

Values identified as “Reserved for TCG” SHOULD NOT be used by implementers in a manner other than that required by the TCG specifications.

1.3 References

[1] Trusted Computing Group (TCG), “Storage Work Group Use Case White Paper – v 1.0”

[2] Trusted Computing Group (TCG), “TCG Storage Interface Interactions Specification“, Version 1.04

[3] Trusted Computing Group (TCG), “TCG Storage Protection Mechanisms for Secrets”, Version 1.00

[4] Advanced Access Content System (AACS), "Introduction and Common Cryptographic Elements", Revision 0.91

[5] [ANSI INCITS 452-2008], “Information technology - AT Attachment 8 - ATA/ATAPI Command Set (ATA8-ACS)“

[6] [INCITS T10/1731-D], “Information technology - SCSI Primary Commands - 4 (SPC-4)“

[7] Internet Engineering Task Force (IETF), "Character Mnemonics & Character Sets" (RFC 1345)

[8] Internet Engineering Task Force (IETF), "Key words for use in RFCs to Indicate Requirement Levels" (RFC 2119)

[9] Internet Engineering Task Force (IETF), " Augmented BNF for Syntax Specifications: ABNF" (RFC 5234)

[10] National Institute of Standards and Technology (NIST), "Secure Hash Standard", FIPS Publication 180-2

[11] National Institute of Standards and Technology (NIST), "Digital Signature Standard (DSS)", FIPS Publication 186-2

[12] National Institute of Standards and Technology (NIST), "Advanced Encryption Standard (AES)", FIPS Publication 197

[13] National Institute of Standards and Technology (NIST), "The Keyed-Hash Message Authentication Code (HMAC)", FIPS Publication 198

[14] National Institute of Standards and Technology (NIST), "Recommendation for Block Cipher Modes of Operation - Methods and Techniques", NIST Special Publication 800-38A

[15] National Institute of Standards and Technology (NIST), "Recommendation for Block Cipher Modes of Operation –The CMAC Mode for Authentication", NIST Special Publication 800-38B

[16] National Institute of Standards and Technology (NIST), "Recommendation for Block Cipher Modes of Operation – The CCM Mode for Authentication and Confidentiality", NIST Special Publication 800-38C

[17] National Institute of Standards and Technology (NIST), "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST Special Publication 800-38D

[18] RSA Laboratories, "PKCS #1: RSA Cryptography Standard (v 2.1)"

[19] Standards for Efficient Cryptography, "SEC2: Recommended Elliptic Curve Domain Parameters", Version 1.0

1.4 Terminology

1.4.1 Global Terminology

Table 01 Global Terminology

Term	Definition
Access Control Element (ACE)	A Boolean expression of authorities.
Access Control List (ACL)	List of ACEs.
Admin SP	The SP that is used in the issuance of other SPs, and provides information about the state of SPs on the TPer as well as the TPer itself.
Authority	An Authority associates a credential with an authentication operation.
Data Types	Encoding format of data. Data is encoded in different ways depending on the context in which the data is being used (stream encoding, table encoding, etc.)
Host Application	Software that communicates with the TPer.

Term	Definition
IF-SEND	An interface command, such as the ATA (T13) TRUSTED SEND or SCSI (T10) SECURITY PROTOCOL OUT command used to transmit data from the host to the TPer.
IF-RECV	An interface command, such as the ATA (T13) TRUSTED RECEIVE or SCSI (T10) SECURITY PROTOCOL IN command used by the host to retrieve data from TPer.
Issuance	The act of creating an SP on a TPer from one or more templates.
MAC	Message Authentication Code
Messaging	Session communications are by messages defined by a messaging protocol. Messages from a Host convey remote method calls on an SP and other messages return the results.
Method	A Method is a remote procedure call to an SP that initiates an action on the SP.
Object	Any row of an Object Table.
Persistent State	This is the content of tables, and exists through power cycles, resets, and spin up/spin down cycles.
Personalization	The act of configuring an issued SP.
Platform Host	A collection of one or more Host Application resources that utilizes or provides a specific service or set of services.
Read command	See: Read user data
Read user data	An operation requested by the host to transfer user data to the host
Secure Messaging	Session communications that support message confidentiality, message integrity/authenticity, or both.
Security Subsystem Class (SSC)	Identifies the components from the Core Specification that are Mandatory, Optional, Excluded, or Not Required for a particular class of security subsystem.
Security Provider (SP)	A collection of Tables and Methods with access control.
Security Identifier (SID)	The authority that represents the TPer owner.
Session	A temporary information exchange that occurs between a host application and an SP, and that is established at a certain point in time and closed at a later point in time. All communications with SPs occur within sessions.
Storage Device (SD)	A Storage Device is any device that provides digital storage services.
Storage Media	Storage Media refers to the non-volatile or persistent storage in a Storage Device.
Storage Work Group (SWG)	One of the TCG working groups whose purpose is to define security building blocks for the Storage Device.
Stream Encoding	The encoding mechanism as defined in section 3.2.2.
SymK	Convenient notation for symmetric key (shared secret) cryptography.

Term	Definition
Table	The basic data structures within an SP. Tables store persistent SP state defined in this specification.
Template	Templates are sets of tables and methods, grouped by feature, from which SPs are created.
TPer	A Trusted Peripheral.
Transaction	A series of one or more method invocations grouped to enable atomicity and state rollback by the host application to a pre-defined point. Methods are invoked either within or outside of transactions.
Transient State	State of an SP that does not persist past the end of a session. This includes authentication state of authorities, changes made in a Read-Only session, or changes made within an uncommitted transaction.
Trusted Commands	Interface protocol commands (IF-SEND or IF-RECV) used to communicate with an SP.
Unique Identifier (UID)	Unique 8-byte identifier that identifies objects within tables, tables, methods, and the SP itself. UIDs are unique within an SP, but not across SPs.
User data	Data that may be transferred between the host and the device using read commands and write commands
Write command	See: Write user data
Write user data	An operation requested by the host to modify user data, which may include transferring data from the application client to the device

2 Trusted Storage Device Architecture

2.1 Architecture Overview

Begin Informative Content

The TCG Storage Architecture supports use cases and threat models developed for the TCG Storage use cases (see [1]). Peripherals based on this architecture are called Trusted Peripherals or TPers.

End Informative Content

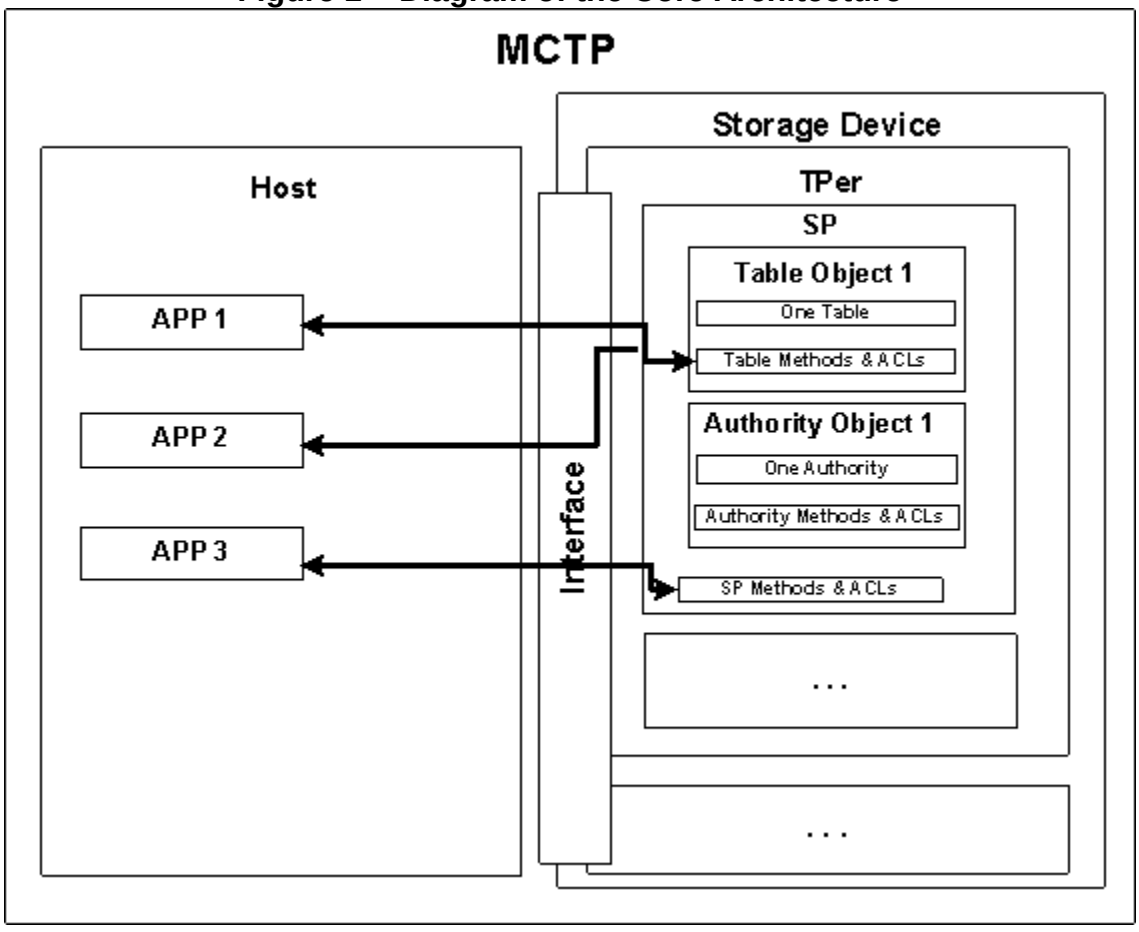
2.2 Architecture Components

Begin Informative Content

The architecture is illustrated in Figure 1, which shows a single Multicomponent Trusted Platform (MCTP) with one Trusted Peripheral (TPer). An MCTP supports 1 or more TPers. Figure 1 shows just one example. Other possibilities include multiple hosts communicating with a single Storage Device/TPer, a single host communicating with multiple Storage Devices/TPers, etc.

End Informative Content

Figure 1 Diagram of the Core Architecture



2.2.1 Multicomponent Trusted Platform (MCTP)

Begin Informative Content

A Multicomponent Trusted Platform (MCTP) describes a platform in which one or more hosts, applications, peripherals, or devices participate in the trust state. In an MCTP, various host applications communicate with the TPer through a peripheral interface such as ATA or SCSI.

End Informative Content

2.2.2 Host

Begin Informative Content

For the purposes of this specification, a Host is the entity that initiates ATA (T13) TRUSTED SEND/RECEIVE commands or SCSI (T10) SECURITY PROTOCOL IN/OUT commands under Security Protocol `0x01-0x06`.

End Informative Content

2.2.2.1 Host Applications

Begin Informative Content

Host applications initiate sessions to communicate with a TPer in order to create, query or change the persistent state of the TPer data structures.

End Informative Content

2.2.3 Trusted Peripheral (TPer)

Begin Informative Content

The Trusted Peripheral (TPer) resides in the Storage Device. The TPer manages trusted storage-related functions and data structures. Two main aspects to the TPer use cases as they pertain to the TCG Storage Architecture are:

a. **Data confidentiality and access control over TPer features and capabilities:** TPer functions and capabilities are built upon policy driven setup and the use of cryptographic access control over TPer content. Such features and capabilities include access controlled readable and writeable data areas, and access control to built-in firmware functions or hardware functions in the TPer. It is possible for a single trusted host application to gain exclusive access to subsets of these features and capabilities. The protection provided by this exclusive access extends to confidentiality of instructions and data in transit between the trusted host application (or a TPM it uses) and the TPer.

b. TPers and Hosts bilateral enrollment and connection: Enrollment establishes the conditions under which data/instruction connections are established between TPers and hosts. The access control conditions for enrollment could be different than those for connection. The data/instruction consequences of a failure to be enrolled or connected MAY be different for different TPers and hosts. The permissions/authorizations required for enrollment and connection of a TPer with a host could be different than the permissions/authorizations required for enrollment and connection of a Host with a TPer.

The TCG Storage Architecture provides for a system of tables where the content and meaning of the table entries are potentially different for different types of Storage Devices with different features and capabilities.

This TCG Storage Architecture's access control system scales with the available Storage Device resources. Storage Device resources include processor performance, memory space, and media capacity. TPer data tables, methods, and capabilities are able to be fixed (and limited) or host application-definable up to the limit of the Storage Device's available resources.

End Informative Content

2.2.4 Security Providers (SPs)

The TPer MAY contain one or more Security Providers (SPs). A Security Provider is a set of tables and methods that control the persistent trust state of the SP and MAY participate in control of the persistent trust state of the TPer. Each SP SHALL have its own storage, functional scope, and security domain.

Begin Informative Content

A Security Provider supports specific TPer functionality. SPs support functions such as authentication, secured attribute-value storage, disk encryption/decryption, backup, time stamping, and event logging. SPs are created by the manufacturer during Storage Device creation, or through the Issuance process (see SP Issuance section 2.3.2).

A Security Provider provides a way for the host and manufacturer to define which TCG functions are performed; who has access to these functions; how the TPer and SPs communicate with the Host; when these events are permitted; and when the events are logged.

A Security Provider is made up of the following components:

a. **Tables**. The two types of tables are described in Section 3.2.5. Tables consist of rows and columns.

b. **Table content** is the persistent state information of the SP.

c. **Methods**. Method operations include functions such as: table additions, table deletion, table read access, and table backup.

d. **Authorities** specify passwords or cryptographic proofs required to become authenticated within a session to the SP.

e. **Access Control Lists (ACLs) and Access Control Elements (ACEs)** bind methods to the authorities that are permitted to invoke them.

End Informative Content

2.3 Core Architecture Operations

2.3.1 Host <–> TPer Communication Infrastructure

Begin Informative Content

The Host communicates with SPs using interface commands generically known as "Trusted Commands." Trusted Commands are interface-specific protocols (i.e. T10 SECURITY PROTOCOL IN/OUT Protocol `0x01-0x06`, or T13 TRUSTED SEND/RECEIVE Protocol `0x01-0x06`). This specification defines the payload content of those commands.

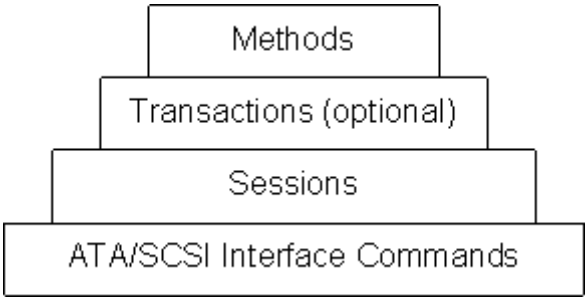
The SP communication protocol that defines the contents of Trusted Commands uses a layered communication system consisting of the following elements:

a. **Methods**.

b. **Transactions**.

c. **Sessions**.

Figure 2 Communications Infrastructure



The only way to communicate with an SP is via a session. Only the host is able to open a session. Methods are invoked within sessions.

Normally, when the methods and associated responses are completed, the host closes the session. Other interface-specific commands (i.e. ATA/SCSI) are able to be interleaved among IF-SEND/IF-RECV commands at any time.

Secure Messaging enables the host and TPer to pass encrypted or integrity protected messages (methods and their associated responses) during sessions. Message encryption is recommended but not required. When secure messaging is in use, it is done regardless of and in addition to any encryption done on the communications channel.

In the simplest communications case the host is just the platform host to which the TPer is directly attached or attached over a network. The host application could also be some other platform host that communicates with the immediate platform host, which then relays the session stream to the TPer over a network. In another case, the TPer could be wirelessly connected to a host application, or part of a SAN and connected to multiple hosts. The TPer could be directly attached to the platform host, and connected to multiple Host Applications either also directly attached to that platform, attached remotely, or both.

End Informative Content

If the device is capable, one or more Read-Only sessions MAY be established simultaneously to a single SP. Typically, changes made to an SP during a Read-Only session SHALL NOT persist past the end of that session. Non-transient changes that persist past the end of a session are noted, where applicable. A case of a non-transient change permitted in a Read-Only session is automatic forensic logging, if enabled.

Read-Write Sessions MAY or MAY NOT alter persistent state information (table content). A Read-Write session (one which has the capability of making non-transient changes to an SP) SHALL be unable to run simultaneously with any other sessions to the same SP.

2.3.2 SP Issuance & Personalization Overview

Begin Informative Content

When TPers are capable of SP issuance, special resources called templates are required. Templates define the initial tables and methods upon which new SPs are based when issued.

The Base Template provides to SPs the tables and methods required for authentication and access control management. SPs are built from a combination of templates, and always include at least a subset of the Base Template. Some templates that extend the capabilities provided by the Base Template are: Admin Template, Clock Template, Crypto Template, Locking Template, and Log Template.

All SPs incorporate at least a subset of the Base Template's tables and methods.

Personalization is the customization of a newly created SP. The primary purposes of personalization are modification of the SP's initial table data and/or the administrative authority on that specific SP, as well as creating additional authorities and customization of the default access control settings. Personalization typically refers to the initial customization of an SP, but the personalization process continues throughout the life of an SP.

End Informative Content

2.3.3 Security Subsystem Classes Overview

Begin Informative Content

The Core Specification defines the set of TCG-related functions supportable by a TPer. However, every TPer is not required to support all functionality defined in this specification. There are multiple "classes" of Core Specification compliance, called Security Subsystem Classes (SSCs). Each Security Subsystem Class specification is a companion document to the Core Specification.

Security Subsystem Classes explicitly define the minimum acceptable Core Specification capabilities of a TPer in a specific "class".

Security Subsystem Classes define only TCG-related functionality. TPer attributes such as host interface type, storage capacity, data rates, and seek times are not key Security Subsystem Class attributes, though TPer resources such as available memory, storage capacity, and processing power influence which Security Subsystem Class(es) a TPer supports.

End Informative Content

A TPer MAY have only some of the capabilities (tables, methods, access controls, etc.) defined in this Core Specification and MAY include additional capabilities through table definitions and/or methods. A Security Subsystem Class SHALL NOT replace a capability called out in the Core Specification with the same capability implemented in different tables, methods, and access controls.

2.3.4 Preliminary Architectural Components

This section identifies a series of architectural components in this specification that are to be considered as preliminary. Implementations of any these elements as defined in this specification MAY NOT be compliant with either SSCs or future versions of this specification.

a. **Secure session start up.** This applies to challenge response authentication and key exchange that occurs during session startup. In addition, if two SP Authorities refer to the same ResponseExch authority, and therefore use the same public-private keypair for encrypting the HostSessionKey to be sent to the TPer,, then there is an escalation replay attack possible where one authority can successfully replay the commands of another.

b. **Session Timeouts, Flow Control, and Session Reliability in regards to control sessions.** This includes, but is not limited to, session timeouts, acknowledgements, negative acknowledgements, transmission timeouts, packet sequence numbers, and credit exchange.

c. **The Log Template and related logging functionality.** In addition to the Log Template, this applies to components in other templates that relate to logging, such as those in the Base Template's `Authority` and `AccessControl` tables. Methods to manage the logging functionality presented in the `AccessControl` table are also TBD.

d. **The Clock Template and functionality related to timekeeping.** In addition to the Clock Template, this applies to components in other templates that relate to timekeeping, such as those in the Base Template's `Authority` table.

e. **Admin SP discovery mechanisms related to TemplateID columns.** This also applies to the `TemplateID` column in tables in Templates other than the Admin Template.

f. **Default ACL values for access control associations when a new table or object is created.** This affects the values are placed in the ACL columns of the `AccessControl`

table when a new table or object is created and associated rows are created in the `Table`, `Column`, `ACE`, and `AccessControl` tables.

g. **Mechanisms for retrieval of meta-ACL column values from the `AccessControl` table.**

h. **The Crypto Template and functionality related to host-requested on-device cryptographic operations.** This includes on-device encryption, decryption, etc. This does not include the `Random` method.

i. **Certificate related components, including the `Certificates` table and the `PresentCertificate` column of the `Authority` table.**

j. **Issuance related components, including the `IssueSP` method.**

3 Architecture Elements

3.1 Architecture Elements Overview

Begin Informative Content

This section introduces global TCG storage-related document format, data structures, and functional behavior.

End Informative Content

3.2 Data Structure Descriptions

3.2.1 Document Data Formats

Begin Informative Content

This specification defines three distinct but closely related data models:

a. **Tables:** Data stored in tables is of a maximum fixed size.

b. **Messaging**: Data moving across the interface is encoded into byte streams. These streams carry encodings for method calls, parameters, and results, as well as other control information.

c. **Exposition Pseudo-code**: This provides a C-like representation of methods and table structure and contents. The definition of the exposition pseudo-code is in section 3.2.1.2.

Data is encoded in different ways depending on the context in which the data is being used. One data context is data stored in tables. Another data context is data crossing the interface in messaging – this is called “Stream Encoding”.

This section introduces the different basic data types, provides a brief introduction on how these types are used, and shows how they are displayed in this document. See Section 3.2.2 for additional details regarding data types and data type Stream Encoding.

End Informative Content

3.2.1.1 Table Definition Format

Begin Informative Content

Each table in this specification is defined in a manner that follows the format described in this section.

A table's structure follows the format that appears in Table 03.

End Informative Content

The description table column "Column Number" identifies the number assigned to that column, which is unique within that table and is used to address the column in methods and other tables. See Table 02 for how Column Numbers are assigned. Column numbers assigned in TCG specs will be assigned from the range of numbers reserved for TCG usage.

Table 02 Column Number Assignment

Column Number Range	Description
0x00-0xFFFEFFFF	Reserved for TCG usage
0xFFFF0000-0xFFFFFFFF	Vendor Unique

The description table column "Column Name" identifies the name of the column. This is the name assigned to that column in the `Column` table, a Base Template table that stores metadata about each column in each table in an SP.

The description table column "IsUnique" identifies whether that column is required to be part of the unique set of column values for that table (See 3.2.5.4).

The description table column "Type" identifies the format of the data stored in that column. The definition of the type itself is found in the `Type` table, a Base Template table that stores metadata about each type used in an SP. Each type used in this specification is defined in 5.1.3.

Each column being described is defined in its own subsection that follows the description table.

Table 03 Foo Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	ID		uid
0x01	Username		name
0x02	SerialNumber		uinteger_4

3.2.1.2 Method Signature Pseudo-code

Begin Informative Content

Method signatures are pseudo-code representations of TCG methods, which are used to describe method parameters, types and snippets of code without having to use the byte encodings directly.

End Informative Content

In this document, MethodName is the UID of the method being invoked and:

a. Session Manager method calls are written as follows, where “SMUID” is `0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xFF: SMUID.MethodName[ <Parameters> ]`

b. SP method calls are written as follows, where "ThisSP" is `0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01`: `ThisSP.MethodName[ <Parameters> ]`

c. Table method calls are written as follows, where TableUID is the UID of the table (see 3.2.5.3) upon which the method is being invoked: `TableUID.MethodName[ <Parameters> ]`

d. Object methods are written as follows, where ObjectUID is the UID of the object (see 3.2.5.3) upon which the method is being invoked: `ObjectUID.MethodName[ <Parameters> ]`

For example:

a. Invoking the `Properties` method: `SMUID.Properties[ <Parameters> ]`

b. Invoking an SP method: `ThisSP.Random[ <Parameters> ]`

c. Adding an entry to a log table: `SomeLogTableUID.AddLog[ <Parameters> ]`

d. Encrypting host data: `C_AES_128ObjectUID.Encrypt[ <Parameters> ]`

A method signature example is displayed in this document as:

```
<InvokingID>.<MethodName>[
       Required Parameter(s),
       Optional Parameter(s)
       ]
=>
[ Result ]
```

The InvokingID (the table or object UID to which the method applies) and MethodName (the method's `UID` column value as it is defined in an SP's `MethodID` table) appear first in the signature. The parameters follow, enclosed in "list" delimiters ("[" and "]"). The "=>" is a separator between the method parameters and the method results. The method results are displayed within the "list" delimiters ("[" and "]").

Methods are made up of two kinds of parameters: required and optional.

a. In the pseudo-code signature, required parameters are given names for ease of reference. The right-hand portion of the parameter is the type, interface (see 3.2.2) or abstract (see 5.1.4), to be supplied for that parameter.

   Required parameters are formatted as follows:

   ```
   a. Parameter-Name : Parameter-type
   ```

b. In the pseudo-code method signature, optional parameters are given in the form of Named values (see 3.2.1.3), and are submitted to the method invocation as Named values. The right-hand portion of the parameter is the type, interface (see 3.2.2) or abstract (see 5.1.4), that is supplied for that parameter. The name supplied to the parameter is expositional, provided for ease of reference.

   Optional parameters are formatted as follows:

   ```
   b. Parameter-Name = Parameter-type
   ```

The result portion of a method's signature is formatted similarly to the above required and optional parameters, using the same conventions for results required to be returned for successful method invocations ("required results") and results returned only in certain situations ("optional results").

The pseudo-code method signatures utilize the following key symbols:

a. **Equals sign ("=")** – Any appearance of "=" in a method's parameter list or result list (including in abstract type definitions) indicates the required use of an interface Named value, where the type of the required value is to the right of the "=". The "=" is not represented in the streamed method data, but indicates that the name and the value are encompassed by the Named value indicator tokens (see 3.2.2.3.2.1).

b. **Colon (":")** – When represented in abstract types or method signatures, a colon indicates that the string to the left of the colon is only a pseudo-code identifier associated with the type to the right of the colon. The type to the right of the colon is the type of the value to be transmitted on the interface.

c. **Separating brackets ("[", "]")** – Square brackets in method signatures are used to mark places in the stream where List tokens (see 3.2.2.2) are used to encapsulate values. Brackets are required to be present in the streamed method invocation, and are represented in the stream by list encoding tokens.

d. **"list"** – The word "list" is used to indicate that the bracketed grouping immediately following is a list (see 3.2.2.3.2.2). Ellipses ("…") in pseudo-code method signatures are used to indicate that multiples of the immediately preceding type appears within the list (e.g. list [type ...]). Note that in some contexts, a list MAY be required to be empty or to contain only a single element. Neither the word "list" nor the ellipses affect the streamed method data.

e. **Commas (",")** – Commas in the pseudo-code method signatures are used to separate items in a list, options in a typeOr value, or to separate parameters, and do not affect the streamed method data.

f. **Curly braces ("{", "}")** – Curly braces are used to provide additional information regarding the type that precedes them (e.g. specifying a specific type of UID reference for a uidref type) or to encapsulate the options for a typeOr value, and do not affect the streamed method data.

3.2.1.3 Messaging Data Types

For stream encoding, because of the manner in which data is encoded and transferred across the interface, the types used in method parameter and result values are described using two basic types:

a. Byte-string values are a sequence of n bytes that are used to represent strings, blobs, bit vectors, etc.

b. N length integer values are whole numbers that are either signed or unsigned.

Due to the nature of method parameters and results, there are two additional constructs defined for messaging that serve as grouping mechanisms for the basic types: Named values and List values.

a. **Named values**. The name (a byte-string/integer/uinteger value) followed by its value (any messaging type, i.e. byte-string values, N length signed or unsigned integer values, list values, or Named values).

b. **List values**. Zero or more values of some type, grouped into an ordered list. List tokens are used to encapsulate method parameters and method results.

Named values and List values serve multiple uses. One use of Named values is to identify optional method parameters in stream encoding. List tokens are used to encapsulate method parameters or to separate the InvokingID/MethodID from the method parameters in the stream encoding. For more information on stream encoding, see 3.2.2.

3.2.1.4 Type Checking

Begin Informative Content:

It is reasonable to consider the parameter list of each method call as a struct with both required and optional member types. Since this is the case, whenever a particular method is received, the TPer is able to check the types of the received parameters to ensure they match the expected types for that method's signature.

For methods that have dynamic parameter requirements (such as the `Get` and `Set` methods), it is necessary to consider the composition of the table upon which the method is operating. Using the `Set` method as an example, the method parameters include identifiers for columns and the values to be assigned to each of those columns. Because the definition of a table is known and fixed, the TPer is able to treat each table as a struct (for the purposes of type checking), with components equivalent to the columns of that table.

With the knowledge of the columns that make up the table/object upon which the method is operating, as well as the type of each of those columns, the TPer is able to initially determine if the value sent is of the correct type for each column. The TPer is able to accomplish this without having to perform strong type checking on whether or not the value is valid for actual assignation to that column (i.e. the TPer is able to initially verify that a particular parameter is a uinteger without having to determine if its size is within bounds for the column).

End Informative Content

3.2.2 Data Stream Encoding

Begin Informative Content

The messaging model provides for stream encoding of multiple remote procedure calls and multiple responses, with the purpose of permitting large data blocks to be broken up and submitted in parts, for the parts to be acted on, and for the results to be returned in parts. This streaming model permits results to be asynchronously returned before all the parts are received.

This section details how values and control markers are encoded into byte sequences for transport over session streams (byte streams).

End Informative Content

There are no predefined limits on the size or length of these data streams. An SSC or TPer implementation MAY limit the maximum size of encoded values.

3.2.2.1 Data Types

As introduced in 3.2.1.3, messaging data is encoded using two basic types of values combined with two grouping mechanisms that are applied to those two basic types. Combined, these four types are able to represent all of the basic and derived data types.

a. **Integers:** Integer values are used to represent numbers, Booleans, and enumerations. The implementation is free to use other representations in other circumstances, converting as necessary. Sign representation for signed integers is two's complement.
b. **Bytes:** These are sequences of bytes and are used to represent strings, cryptographic keys, bit-vector encoded sets, blobs, etc.
c. **List:** Zero or more values of any type, grouped into an ordered list**.** All items in the list must be of the same type.
d. **Named:** The name (a byte-string/integer/uinteger value) followed by its value (any messaging type). A Named value attaches an identifier to some other value (ex. size=32).

3.2.2.2 Endianness

The endianness of integers transmitted across the interface is big endian.

3.2.2.3 Tokens

Values of the four basic types are packaged into tokens, each of which is a TLV (tag, length, value) sequence of bits that specifies a single data value.

Table 04 Token Types

Byte 0	Byte 1	Byte 2	Byte 3	Hex	Acronym	Meaning
0 *S* *d<5..0>*				00..7F		Tiny atom
1 0 *B* *S* *n<3..0>*				80..BF		Short atom
1 1 0 *B* *S* *n<10..0>*				C0..DF		Medium atom
1 1 1 0 0 0 *B* *S* *n<23..16>*	*n<15..8>*	*n<7..0>*		E0..E3		Long atom
				E4..EF	TCG Reserved	
1 1 1 1 0 0 0 0				F0	SL	Start List
1 1 1 1 0 0 0 1				F1	EL	End List
1 1 1 1 0 0 1 0				F2	SN	Start Name
1 1 1 1 0 0 1 1				F3	EN	End Name
				F4..F7	TCG Reserved	
1 1 1 1 1 0 0 0				F8	CALL	Call
1 1 1 1 1 0 0 1				F9	EOD	End of Data
1 1 1 1 1 0 1 0				FA	EOS	End of session
1 1 1 1 1 0 1 1				FB	ST	Start transaction
1 1 1 1 1 1 0 0				FC	ET	End transaction
				FD..FE	TCG Reserved	
1 1 1 1 1 1 1 1				FF	MT	Empty atom

The Token Types identified in Table 04 are divided into 3 subgroups:

a. Simple Tokens - Atoms: tiny, short, medium, long, and empty atoms

b. Sequence Tokens: Start List, End List, Start Name, and End Name

c. Control Tokens: Call, End of Data, End of Session, Start Transaction, End Transaction

Tokens `0xE4-0xEF`, `0xF4-0xF7` and `0xFD-0xFE` are reserved for use by TCG.

An SSC MAY define support for only a subset of the available tokens, as well as the behavior of the TPer when unsupported tokens are transmitted by the host.

3.2.2.3.1 Simple Tokens – Atoms Overview

Atoms are used to encode data of various sizes and types. Atoms MAY be tiny atoms, which are one byte in length; short atoms which have a 1-byte header and contain up to 15 bytes of data; medium atoms which have a 2-byte header and contain up to 2047 bytes of data; or long atoms which have a 4-byte header and which contain up to 16,777,215 bytes of data.

Tiny atoms only represent integers, whereas short, medium, and long atoms are used to represent integers or bytes (with the “B” bit set).

A continued value is used to represent a long byte sequence when the total length is not known in advance. A continued value is represented by a sequence of two or more atoms.

Each atom in a continued value MAY be a short atom, medium atom, or long atom. The BS bits are set to 11b for all atoms except the last atom, for which the BS bits are set to 10b. All representations of continued values are considered equivalent encodings of the same value.

Integer and uinteger values SHOULD be encoded using the shortest possible atom.

3.2.2.3.1.1 Tiny atoms

Tiny atom header and data are all contained in eight bits.

Table 05 Tiny Atom Description

Header+Data								
Tiny atom	sign	data						
0	S	*d*	*d*	*d*	*d*	*d*	*d*	

The encoding is as follows:

Table 06 Tiny Atom Encoding

Tiny Atom indicator	This bit is set to 0b to indicate the atom is a tiny atom
Sign indicator	***Value Interpretation*** 0b The data is treated as unsigned integer data. 1b The data is treated as a signed integer.
Data bits	These represent the data value, an unsigned value in the range of 0...63 or a signed value in the range of –32...31. The interpretation is based on the setting of the sign bit.

3.2.2.3.1.2 Short atoms

Short atoms consist of a one-byte header and between 0 and 15 bytes of data.

Table 07 Short Atom Description

Header (1 byte)				Data
Short Atom	byte/ integer	sign/ continued	length	(0...15 bytes)

Header (1 byte)					Data
Short Atom	byte/ integer	sign/ continued	length		(0...15 bytes)
1	0	B	S	*n n n n*	*d* ... *d*

The encoding is as follows:

Table 08 Short Atom Encoding

Short Atom indicator	These two bits are set to 10b to indicate the atom is a short atom.
Byte/integer indicator	***Value*** ***Interpretation*** 0b The data bytes represent an integer value and the S bit indicates if that value is signed. 1b The data bytes represent a byte sequence and the S bit indicates whether or not this value is continued into another atom.
Sign/continued indicator	***Value*** ***Interpretation*** 0b The interpretation of the data depends on the byte/integer indicator bit. B==0b The data is treated as unsigned integer data. B==1b The data is either the complete byte sequence, or the final segment of a continued byte sequence. 1b The interpretation of the data depends on the byte/integer indicator bit. B==0b The data is treated as signed integer data. B==1b The data is a non-final segment of a multi-byte continued value.
Length	These bits specify the length of the following data byte sequence. The permitted range is from 0 to 15, inclusive.

A length of 0 SHALL only be permitted for non-continued bytes tokens The encoding of a 0-length byte value is displayed in Table 09.

Table 09 0-Length Byte Encoding

Header (1 byte)				
Short Atom	byte/ integer	sign/ continued	length	
1	0	1	0	*0 0 0 0*

A 0-length byte value is encoded using only 1 byte: 1 0 1 0 0 0 0 0. This value would be encoded in the token stream as `0xA0`.

3.2.2.3.1.3 Medium atoms

Medium atoms consist of a two-byte header, and between 1 and 2047 bytes of data.

Table 10 Medium Atom Description

Header (2 bytes)						Data
0				1		...
Medium Atom	byte/ integer	sign/ continued	length			(1..2047 bytes)
1 1 0	B	S	*n n n n n n n n n n n*			*d* ... *d*

The encoding is as follows:

Table 11 Medium Atom Encoding

Medium Atom indicator	These three bits are set to 110b to indicate the atom is a medium atom.
Byte/integer indicator	***Value*** ***Interpretation*** 0b The data bytes represent an integer value and the S bit indicates if that value is signed. 1b The data bytes represent a byte sequence and the S bit indicates whether or not this value is continued into another atom.
Sign/continued indicator	***Value*** ***Interpretation*** 0b The interpretation of the data depends on the byte/integer indicator bit. B==0b The data is treated as unsigned integer data. B==1b The data is either the complete byte sequence, or the final segment of a continued byte sequence. 1b The interpretation of the data depends on the byte/integer indicator bit. B==0b The data is treated as signed integer data. B==1b The data is a non-final segment of a multi-byte continued value.
Length	These bits specify the length of the following data byte sequence. The value 0 is not a legal value. The permitted range is up to 2047.

3.2.2.3.1.4 Long atoms

Long atoms consist of a four-byte header, and between 1 and 16M-1 bytes of data.

Table 12 Long Atom Description

Header (4 bytes)																										Data		
0								1				2				3										...		
Long Atom				reserved		byte/ integer	sign/ continued	Length																		(1..16,777,215 bytes)		
1	1	1	0	0	0	B	S	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*n*	*d*	...	*d*

The encoding is as follows:

Table 13 Long Atom Encoding

Long Atom indicator	These four bits are set to 1110b to indicate the atom is a long atom.
reserved	These bits are reserved and SHALL be set to 0b.
Byte/integer indicator	***Value*** ***Interpretation*** 0b The data bytes represent an integer value and the S bit indicates if that value is signed. 1b The data bytes represent a byte sequence and the S bit indicates whether or not this value is continued into another atom.

Sign/continued indicator	***Value*** ***Interpretation*** 0b The interpretation of the data depends on the byte/integer indicator bit. B==0b The data is treated as unsigned integer data. B==1b The data is either the complete byte sequence, or the final segment of a continued byte sequence. 1b The interpretation of the data depends on the byte/integer indicator bit. B==0b The data is treated as signed integer data. B==1b The data is a non-final segment of a multi-byte continued value.
Length	These bits specify the length of the following data byte sequence. The value 0 is not a legal value. The permitted range is up to 16,777,215.

3.2.2.3.1.5 Empty Atom

The Empty atom is one byte consisting of eight 1 bits.

Table 14 Empty Atom Description

Header (1 byte)							
Empty Atom							
1	1	1	1	1	1	1	1

The Empty atom MAY appear at any point in the stream encoding where any other atom is able to appear, including between the atoms of a continued value and after the End of Session, and it SHALL be ignored.

Begin Informative Content

The Empty atom does not encode values. The Empty atom allows other values in the data subpacket contained in that part of the stream to be aligned with multi-byte boundaries for efficiency. It also allows areas of a fixed buffer to be filled with a value that is able to be safely ignored.

End Informative Content

3.2.2.3.2 Sequence Tokens

Composite values, such as Named values and lists, are represented by a sequence of tokens.

3.2.2.3.2.1 Named

Named values have the expositional form `name=value` and are used to represent a name-value pair. A Named value is a sequence of tokens: a `Start Name` token (SN), followed by a non-continued byte-string/uinteger/integer value that specifies the name, followed by any value (including list or a Named value), followed by an `End Name` token (EN).

3.2.2.3.2.2 List

Lists are ordered sequences of elements of the form `[e1,e2,...,ei]`. List elements MAY be tokens, lists, or Named values. A list is encoded as a `Start List` token (SL) followed by a sequence of zero or more elements followed by an `End List` token (EL).

3.2.2.3.3 Control Tokens

Control tokens are single byte tokens that are used to specify special actions.

3.2.2.3.3.1 Call (CALL)

This token is used to indicate the start of a method invocation.

3.2.2.3.3.2 End of Data (EOD)

This token is used to signal the end of the parameters, or the result, of a method invocation. This token is used in message streams by both the host and the SP.

3.2.2.3.3.3 End of Session (EOS)

The host application utilizes this token to signal to the SP that it is ending the session. The SP responds to this token with an End of Session token of its own in its response stream.

3.2.2.3.3.4 Start Transaction (ST)

The host application utilizes this token to open a transaction.

When the host begins a transaction, the Start Transaction token is sent by the host to the SP along with the status, a uinteger, required for that transaction control token. The status supplied by the host with the Start Transaction token SHOULD be a `0x00`, and SHALL be ignored by the TPer.

When the SP delivers its response to the host application's message, the SP's message SHALL mirror that of the host by including Start Transaction tokens in the corresponding places in the message stream. The TPer SHALL supply the status of the Start Transaction request. If the host sends a non-zero status code with the Start Transaction token, the device SHALL respond with a status code of `0x00`, unless the transaction was unable to start.

If the host transmits a Start Transaction token that causes the transaction nesting limit to be exceeded, the TPer SHALL abort the session (see Properties Section for details on the transaction nesting limit). If for any reason the TPer is unable to start a transaction as requested by the host, the TPer SHALL abort the session.

Table 15 Start Transaction Status Codes

Start Transaction Status Code (uinteger)	Meaning
0x00	Success
>0x00	Reserved

3.2.2.3.3.5 End Transaction (ET)

The host application utilizes this token to commit or abort the associated open transaction level.

When the host ends the transaction, the End Transaction token is sent by the host to the SP along with the uinteger status required by the host for that transaction control token.

When the SP delivers its response to the host application's message, the SP's message SHALL mirror that of the host by including End Transaction tokens in the equivalent places in the message stream along with the actual status of the End Transaction request.

The host SHOULD send a status code of `0x00` or `0x01` with an End Transaction token. A status code of `0x00` signals to the device that the host is committing that transaction level. A status code of `0x01` signals to the device that the host is aborting that transaction level and the TPer SHALL abort that transaction level.

If the host sends a status code of `0x00`, the device SHALL attempt to commit that transaction level, and SHALL return either `0x00` in the case of a successfully committed transaction or `0x01` in the case of an unsuccessfully committed transaction.

If the host sends a status code with an End Transaction token that the device does not support, the device SHALL abort the transaction and return a status code of `0x01`.

Host delivery of the End Transaction token with a status code other than `0x00` signals that the host is aborting the transaction. The TPer SHALL abort that transaction level.

SP delivery of the End Transaction token with a status code other than `0x00` signals that the SP aborted the transaction.

Table 16 End Transaction Status Codes

End Transaction Status Code (uinteger)	Meaning
0x00	Commit
0x01	Abort
>0x01	Reserved

3.2.2.3.4 Out of Order Control Tokens

In cases where the host transmits out of order control tokens the TPer SHOULD abort the session. These cases include (but are not limited to):

a. Multiple consecutive control tokens of the same type where this repetition is not permitted. This includes the Call, End of Data, and End of Session tokens.
b. Out of order control tokens

Any tokens encoded after an End of Session token SHALL be ignored by the TPer.

3.2.2.4 Invalid and Unexpected Tokens

3.2.2.4.1 Invalid Tokens

An invalid token is a token that is not supported by the TPer's current communications configuration.

The list of invalid tokens is as follows:

1. A token whose size is greater than the TPer's MaxIndTokenSize property.
2. A token whose size is greater than the TPer's MaxAggTokenSize property.
3. A continued token, if the TPer does not support continued tokens (the ContinuedTokens property is reported by the TPer as FALSE).
4. Transaction Control tokens, if the TPer does not support transactions (the MaxTransactionLimit property is omitted from the TPer's Properties method response).
5. Unsupported simple tokens (ie if an SSC does not require support for certain simple tokens).
6. A TCG Reserved token.

When an invalid token appears in a communication from the host, the TPer SHALL behave as follows:

1. For regular sessions the TPer SHALL abort the session associated with the Packet that contained the violating token. Results for methods that were completed before the violating token was encountered SHALL be sent to the host.
2. For control sessions the TPer SHALL stop processing the packet where the violating token occurs, and ignore the remainder of the packet. Results for methods completed before the violating token was encountered SHALL be sent to the host.

3.2.2.4.2 Unexpected Tokens

An unexpected token is an otherwise valid and supported token that, based on the construction of a Subpacket payload, is not a token of the type that is expected, by the construction of the Subpacket payload, to occupy a particular position in that payload.

Begin Informative Content

For example, in the construction of a method invocation, the beginning of the Subpacket payload (discounting empty atoms) has the following structure:

1. Call – the Call token
2. InvokingID – a byte token
3. MethodID – a byte token
4. StartList – the StartList token

If the beginning of the Subpacket payload was sent by the host as follows: Call token + uinteger token + byte token + StartList token; then the uinteger token that appears between Call and the byte token would be unexpected, as the expected token is a byte token.

End Informative Content

Given no other encoding errors in the Subpacket payload, or any other layer of the protocol stack, the result of inclusion of an unexpected token SHALL be on e of the following, based on the conditions stated in each item:

1. For a regular session, abort the session (Note that the TPer is always free to abort the session at any time for any reason). For a control session, ignore the remainder of the packet. Results for methods that were completed before the violating token was encountered SHALL be sent to the host.
2. If the unexpected token appears outside of a method invocation, at any position in the subpacket other than after a Call token and before the next End of Data token after that Call token, abort the session for a regular session or, for a control session, ignore the remainder of the packet. Results for methods that were completed before the violating token was encountered SHALL be sent to the host.
3. If the unexpected token appears within a method invocation after the Call token but before a the first StartList token after that Call token (ie in the method header) for a method invoked within a regular session, the method fails and the TPer responds with an empty method result list and a method status of NOT_AUTHORIZED. If the unexpected token appears within a method invocation after the Call token but before the first StartList token after that Call token for a method invoked within a control session, the method SHALL be ignored.
4. If the unexpected token appears within a method invocation after parameter StartList and before the End of Data token (ie in the method parameter list), the method fails and the TPer responds with a method result list that MAY be empty, and a method status of INVALID_PARAMETER. This includes type mismatches for parameter values.
5. If the unexpected token appears between the StartList token that marks the beginning of the Status List, and the EndList token that marks the end of the Status List, abort the session for a regular session or, for a control session, ignore the remainder of the packet. Results for methods that were completed before the violating token was encountered SHALL be sent to the host. Note that the status list is a list of unsigned integers, so the appearance of a signed integer, or any other non-uinteger token, is unexpected.

If empty atoms are supported, then they SHALL NOT be unexpected tokens.

3.2.3 ComPackets, Packets & Subpackets

Begin Informative Content

The low-level interface transport layer handles the retransmission of damaged or incomplete commands. Secure messaging, detailed in later sections of this specification, permits the host application to secure its data from malicious attack, not to address hardware and low-level transport issues. (Similarly with the session start up protocol, hashing is intended to detect tampering.)

The payloads of ComPackets convey tokenized byte streams (method calls, parameters, results, and status codes) and other control information, such as ACKs and NAKs.

End Informative Content

3.2.3.1 Format

A ComPacket is the primary unit of communication transmitted as the payload of an interface command. An interface command payload SHALL hold only one ComPacket. A ComPacket SHALL NOT span multiple interface commands. A ComPacket MAY contain zero or more packets in its payload.

A Packet is associated with a particular session and MAY hold zero or more subpackets.

A Subpacket MAY hold zero or more tokens. Tokens MAY span multiple subpackets and multiple packets. However, subpackets SHALL NOT span multiple packets, and packets SHALL NOT span multiple ComPackets.

3.2.3.2 ComPacket Format

Table 17 ComPacket Format

Bit / Byte	7	6	5	4	3	2	1	0
0	(MSB)							
1				Reserved				
2								
3								(LSB)
4	(MSB)							
5				ComID				(LSB)
6	(MSB)							
7				ComID Extension				(LSB)
8	(MSB)							
9				OutstandingData				
10								
11								(LSB)
12	(MSB)							
13				MinTransfer				
14								
15								(LSB)
16	(MSB)							
17				Length (n)				
18								
19								(LSB)
If Length > 0, 20 to n + 19	Payload							

3.2.3.2.1 ComPacket Header Fields

3.2.3.2.1.1 Reserved

The values in this field are reserved.

This field SHOULD be set to zero and SHALL be ignored by both host and TPer.

3.2.3.2.1.2 ComID

The value in this field is the ComID of this ComPacket (see 3.3.2).

3.2.3.2.1.3 ComID Extension

The value in this field is the ComID Extension of this ComPacket (see 3.3.3.1)

3.2.3.2.1.4 OutstandingData

For ComPackets sent by the TPer to the Host, this field contains the total number of bytes that the TPer has available for the host on this ComID. This value is based on the data available in the TPer at the point in time when the ComPacket is transmitted to the host by the TPer.

This total SHALL NOT include the data being transferred in the current ComPacket. This total SHALL include Compacket/Packet/Subpacket overhead. If the TPer has no additional data for this ComID, this value SHALL be `0x0000_0000`. If the TPer has more than `0xFFFF_FFFF` bytes for this ComID, this value SHALL be `0xFFFF_FFFF`. If the TPer is still processing a response but no additional data is ready yet, this value SHALL be `0x0000_0001`.

For ComPackets sent by the Host to the TPer, this field is reserved and SHOULD contain `0x0000_0000`, and SHALL be ignored by the TPer.

3.2.3.2.1.5 MinTransfer

For ComPackets sent by the TPer to the Host, this field contains the minimum number of bytes that the host SHALL request on this ComID in order to transfer a packet for any session associated with this ComID. This value is based on the data available in the TPer at the point in time when the ComPacket is sent by the TPer.

This value SHALL include Compacket/Packet/Subpacket overhead. If the TPer has no additional data for this ComID, or if the TPer has no minimum requirement, this value SHALL be `0x0000_0000`. The host application that manages this ComID SHOULD request at least MinTransfer bytes on the next IF-RECV command that it sends for this ComID.

For ComPackets sent by the Host to the TPer, this field is reserved and SHOULD contain `0x0000_0000`, and SHALL be ignored by the TPer.

3.2.3.2.1.6 Length

This field value is the number of bytes in the ComPacket payload.

3.2.3.2.2 ComPacket Payload Fields

3.2.3.2.2.1 Data

This field contains a sequence of one or more packets.

3.2.3.3 Packet Format

Each packet is made up of the fixed fields noted in this section to allow acknowledgements, negative acknowledgements, and/or data to be included in a single packet.

Table 18 Packet Format

Byte \ Bit	7	6	5	4	3	2	1	0
0	(MSB)							
1								
2								
3				Session (0-3 = TSN, 4-7 = HSN)				
4								
5								

Byte \ Bit	7	6	5	4	3	2	1	0
6								
7								(LSB)
8	(MSB)							
9				SeqNumber				
10								
11								(LSB)
12	(MSB)							
13				Reserved				(LSB)
14	(MSB)							
15				AckType				(LSB)
16	(MSB)							
17				Acknowledgement				
18								
19								(LSB)
20	(MSB)							
21				Length (n)				
22								
23								(LSB)
If Length > 0, 24 to n + 23	Payload							

3.2.3.3.1 Packet Header Fields

3.2.3.3.1.1 Session

This field identifies the session number associated with this packet. The session number is composed of two uinteger_4 values – the TPer session number and the Host session number (Session = TPerSN concatenated with the HostSN). The TPer Session Number is sent first; the Host Session Number is second. Consequently, the same session number is used for communications between both parties.

3.2.3.3.1.2 SeqNumber

This is an incrementing counter that starts at 1 and increments until 2^{32}-1, which identifies the number of the packet within the session and defines the ordering of transmitted packets.

If packet numbering is supported, the message recipient SHALL ignore a packet with an equal or lower SeqNumber value than any previously acted-upon packet. In addition, wrapping of the SeqNumber SHALL result in the session being automatically aborted.

Each communicator SHALL maintain multiple SeqNumber counts, including that of the last packet acknowledged, the next packet expected, and the last packet transmitted.

3.2.3.3.1.3 Reserved

The values in this field are reserved.

This field SHOULD be set to zero and SHALL be ignored by both host and TPer.

3.2.3.3.1.4 AckType

This field identifies the usage of the Acknowledgement field.

a. This SHALL be `0x0001` if the Acknowledgement field contains a packet acknowledgement (ACK).

b. This SHALL be 0x0002 if the Acknowledgement field contains a packet negative acknowledgement (NAK).

c. This SHALL be 0x0000 if no packets are being acknowledged or negative acknowledged, and the value of the Acknowledgement field SHALL be zeroes.

3.2.3.3.1.5 Acknowledgement

The meaning of this field is determined by the value of the AckType field.

a. If the value of the AckType field is 0x0001, then this number SHALL be the SeqNumber of the last packet successfully received by the receiver.

b. If the value of the AckType field is 0x0002, then this SHALL be the SeqNumber of the packet at which the receiver wishes the sender to begin retransmission. Generally, the receiver puts a value of the last known good packet received plus one.

    i. For AckType field value of 0x0002, the communicator SHALL NOT NAK a SeqNumber less than or equal to the last ACKed SeqNumber.

c. If the AckType field is 0x0000, then the value of this field SHALL be zeroes.

3.2.3.3.1.6 Length

This field identifies the number of bytes in the Payload field.

3.2.3.3.2 Packet Payload Fields

3.2.3.3.2.1 Data

This field contains a sequence of one or more subpackets.

3.2.3.4 Subpacket Formats

Begin Informative Content

Subpackets are used to package data for transmission between the host and the TPer, as well as to exchange credits between communicators. The different types of Subpackets are enumerated in Table 19

End Informative Content

Table 19 Subpacket Types

Subpacket Type	Kind Field Value
Data	0x0000
Credit Control	0x8001

3.2.3.4.1 Data Subpacket Format

Table 20 Data SubPacket Format

Byte \ Bit	7	6	5	4	3	2	1	0
0	(MSB)							
1								
2								
3				Reserved				
4								
5								(LSB)
6	(MSB)							
7				Kind				(LSB)
8	(MSB)							
9								
10				Length (n)				
11								(LSB)
If Length > 0, 12 to (n+(-n modulo 4)) + 11	Payload							

3.2.3.4.1.1 Data Subpacket Header Fields

3.2.3.4.1.1.1 Reserved

The values in this field are reserved.

This field SHOULD be set to zero and SHALL be ignored by both host and TPer.

3.2.3.4.1.1.2 Kind

This field identifies the type of the subpacket. For data subpackets, this field is set to zeroes.

3.2.3.4.1.1.3 Length

The field identifies the number of bytes in the Data portion of the subpacket Payload. This value does not include the length of the Pad portion of the Payload.

3.2.3.4.1.2 Data Subpacket Payload Fields

3.2.3.4.1.2.1 Data

This field contains the stream tokenization.

3.2.3.4.1.2.2 Pad

The pad field ensures that the boundaries between subpackets (and therefore packets) are aligned to 4-byte boundaries. The number of pad bytes SHALL be (-Subpacket.Length modulo 4). This field SHALL be zeroes.

Begin Informative Content

The receiver of a Subpacket is able to unambiguously determine how many bytes of real data there are by examining the Length field in the Subpacket header.

Note that the standard C library does not handle the modulo operation with negative numbers properly.

End Informative Content

3.2.3.4.2 Credit Control Subpacket Format

For information on the use of Credit Control Subpackets, see Flow Control in Section 3.3.8.

Table 21 Credit Control Subpacket

Byte \ Bit	7	6	5	4	3	2	1	0
0	(MSB)							
1								
2								
3				Reserved				
4								
5								(LSB)
6	(MSB)							
7				Kind				(LSB)
8	(MSB)							
9								
10				Length				
11								(LSB)
12	(MSB)							
13								
14				Credit				
15								(LSB)

3.2.3.4.2.1 Credit Control Subpacket Header Fields

3.2.3.4.2.1.1 Reserved

The values in this field are reserved.

This field SHOULD be set to zero and SHALL be ignored by both host and TPer.

3.2.3.4.2.1.2 Kind

This field identifies the type of the subpacket. For Credit Control Subpackets, this field is set to `0x8001`.

3.2.3.4.2.1.3 Length

The field identifies the number of bytes in the in the Credit Control Subpacket payload. This is always `0x00000004` for a subpacket of this type.

3.2.3.4.2.2 Credit Control Subpacket Payload Fields

3.2.3.4.2.2.1 Credit

This field identifies the number of bytes to credit. This is an additional number of bytes that the receiver of the Credit Control Subpacket MAY be send to the stream (see 3.3.8.2).

3.2.3.5 Secure Messaging Packet Format

Begin Informative Content

Secure messaging enables confidentiality of the packet payload and integrity/authenticity of the entire packet (including header). Secure messaging comes in three types:

a. Confidential Messaging – this provides encryption on the message being transmitted. Confidential Messaging prevents the packet contents from being read by an intruder between the packet source and destination.

b. Integrity/Authenticity Checking – this provides the ability to detect corruption and/or tampering with packets in a session.

c. Confidential Messaging with Integrity/Authenticity Checking – this provides encryption on the message being transmitted and the added ability to detect corruption and/or tampering with packets in a session.

End Informative Content

A secure messaging packet SHALL be used when encryption or integrity/authenticity checking (or both) is enabled for a session. The format of the Secure Messaging Packet follows that defined in Table 18. The contents of the Secure Messaging Packet payload field are displayed in Table 22 and Table 23.

Table 22 Secure Messaging Packet – Payload Field

Field	Type
IV	bytes
SecureData	bytes
MAC	bytes

Table 23 Secure Messaging Packet Payload– SecureData Field

Field	Type
DataLength	uinteger
Data	bytes
Pad	bytes

3.2.3.5.1 Secure Messaging Packet Header Fields

3.2.3.5.1.1 Session

See Section 3.2.3.3.1.1.

3.2.3.5.1.2 SeqNumber

See Section 3.2.3.3.1.2.

3.2.3.5.1.3 Reserved

See Section 3.2.3.3.1.3.

3.2.3.5.1.4 AckType

See Section 3.2.3.3.1.4.

3.2.3.5.1.5 Acknowledgement

See Section 3.2.3.3.1.5.

3.2.3.5.1.6 Length

See Section 3.2.3.3.1.6.

3.2.3.5.2 Secure Messaging Packet Payload Fields

This section describes the fields presented in Table 22.

3.2.3.5.2.1 Initialization Vector (IV)

This field contains the Initialization Vector (IV) input for the selected encryption or integrity checking mode. For GCM, GMAC, and CCM, the IV is 8 bytes long and SHALL contain a unique value with each encryption invocation. A simple algorithm is for the sender to use the sequence number as the IV.

For AES-CBC encryption, the IV SHALL contain a random 16-byte value.

For all other modes, the IV SHALL have zero length.

3.2.3.5.2.2 SecureData

This field contains the encrypted and/or integrity-protected data being transmitted in the packet.

3.2.3.5.2.2.1 DataLength

This field is the length of the SecureData's Data field, in bytes.

3.2.3.5.2.2.2 Data

This field contains the encrypted or integrity-checked subpackets

3.2.3.5.2.2.3 Pad

This field contains any necessary padding required to fulfill the alignment constraints for the encryption mode in use. For AES-CBC encryption, the length of the Pad field SHALL include a number of padding bytes such that the total length of the Data field plus the Pad field is congruent to zero mod 16. For GCM and CCM, there is no required padding.

The value for pad bytes SHALL be `0x00`.

3.2.3.5.2.3 Message Authentication Code (MAC)

This field contains a message authentication code that protects the integrity of the packet. The MAC SHALL encompass the entire Packet header, including the Reserved field, IV, and, for encrypted data, the ciphertext (the value of the SecureData field, which is made up of the Data Length, Data, and Pad fields), or, for unencrypted data, the unencrypted SecureData field.

3.2.4 Methods

Begin Informative Content

This section describes the syntax and encoding of method calls.

End Informative Content

3.2.4.1 Method Syntax

A method invocation is made up of the following parts:

1. Method Header – The method header is made up of the InvokingID and the MethodID, and identifies what method is being called and on what the method is operating.
    1. **InvokingID** – This is the 8-byte UID of the table, object, or SP upon which the method is being invoked.
        a. For SP methods invoked within a session, the InvokingID SHALL be `0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01`, which is used to signify “this SP”.
        b. For methods invoked at the Session Manager Layer, the InvokingID SHALL be `0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xFF`, known as the "SMUID".
        c. For other methods, this is the 8-byte UID of the table or object upon which the method is being invoked.

2. **MethodID** – This is the 8-byte UID of the method being invoked.
    a. For methods invoked within a session, this SHALL be the UID column value of the object that represents the methed as assigned in the `MethodID` table.
    b. For Session Manager Layer methods, this SHALL be the UID as assigned in Table 241. There SHALL NOT be rows in the `MethodID` table that represent these methods.

2. **Method Parameters** – This is a list of the parameters submitted to the method. These parameters MAY be one of two types.
    1. **Required parameters** – These parameters are required to be submitted to the method invocation. These parameters SHALL appear first in the method invocation, ahead of any optional parameters, and SHALL be submitted in the order in which they are listed in a method's signature as defined in this specification.
    2. **Optional parameters** – These parameters SHALL NOT be required to be submitted to the method invocation. Optional parameters that are submitted to a method invocation SHALL be submitted after all required parameters, and SHALL appear in the order defined in this specification.
        a. Optional parameters are submitted to the method invocation as Named value pairs. The Name portion of the Named value pair SHALL be a uinteger. Starting at zero, these uinteger values are assigned based on the ordering of the optional parameters as defined in this document.
        b. The first optional parameter in a method signature SHALL be represented by the "name" zero (`0x00`) in the Named value pair when that method is invoked, and SHALL thus have the format "`0x00 = value`" when that method is invoked.
        c. Each optional parameter in a method signature after the first SHALL be represented by the uinteger of the previous optional parameter indicated in the method's signature incremented by one. Thus, the second optional parameter in an invocation of a particular method SHALL have the format "`0x01 = value`".

3.2.4.2 Method Encoding

A method invocation is made up of a sequence of tokens that are sent from the application to the TPer, and from the TPer to the host for Session Manager method responses, as follows:

1. **Call token** – A Call token is transmitted to indicate that a method invocation is to follow.
2. **Method Header –** This is the encoding of the InvokingID and the MethodID. This value is:
    1. **InvokingID –** The InvokingID is a bytes token representing the 8-byte value that is the first part of the Method Header being transmitted.
    2. **MethodID –** The MethodID is a bytes token representing the 8-byte value that is the second part of the Method Header being transmitted.
3. **Parameters –** The parameters are submitted to a method invocation as a list. The parameter list follows this format:
    1. **Start List token –** This identifies the beginning of the list of parameters.
    2. **Required parameters –** This is the set of parameters that are required to be sent to a method. The encoding of required parameters is dependent on the type associated with that parameter, as defined by the method signature and the context in which the method is being invoked.
    3. **Optional parameters** – this is the set of zero or more Named value pairs that MAY be sent to the method to represent the method's optional parameters. Each optional parameter SHALL be made up of the following parts:
        1. The Start Name token, which indicates the start of this optional parameter.

2. The encoded name, which in the case of optional parameters is a uinteger.
3. The encoded value. The encoding of the parameter value is dependent on the type associated with that parameter, as defined by the method signature and the context in which the method is being invoked.
4. The End Name token, which indicates the end of this optional parameter

4. **End List token** – this identifies the end of the list of parameters.

4. **End Of Data token –** The End of Data token is transmitted to indicate that the method invocation is ended.
5. **Status Code List** – This is the status list, a list of values of type uinteger, which contains the status codes expected from the host's invocation of the method. These status values are encoded using List tokens.
    1. The first value in the list SHALL be `0x00` for a method that the host expects to complete properly. For a method that the host wishes to abort, the host SHALL NOT include a value that is `0x00` as the first value in the status list, which SHALL cause the TPer to abort processing on that method and return that non-`0x00` value as the first value in the status list.
    2. The second and third values in the status list are reserved, and are defined in this specification to be `0x00` and `0x00` and SHOULD be ignored by the TPer.

Except for the Session Manager methods, each method call SHALL have a response that is a sequence of tokens that are sent from the TPer to the host as follows.

1. **Start List token –** This identifies the beginning of the list of results.
2. **Output Results –** This is zero or more token sequences that represent the response to the method, as defined in the method signature.
3. **End List token –** This identifies the end of the list of results.
4. **End Of Data –** The End of Data token is transmitted to indicate that the result list has ended.
5. **Status List** – This is the status list, a list of values of type uinteger, which contains the status codes expected from the host's invocation of the method. These status values are encoded using List tokens.
    i. If the host invoked the method with a status list whose first uinteger was `0x00`, then the first value in the status list SHALL always be the status of the method, as described in 5.1.5. If the host invoked the method with a status list whose first byte was not `0x00`, then the first value in the status list SHALL contain the same value that was sent by the host in the first uinteger of the host's status list.
    ii. The second and third values in the list are uintegers reserved for use by the TCG, and are defined in this specification to be `0x00` and `0x00` and SHOULD be ignored by the host.
    iii. Additional values MAY be returned in the status list, as long as the first three values in the status list are returned as required by this specification.

Method responses SHALL be returned for all method invocations or method invocation attempts within a session. Responses for method invocation attempts of methods not recognized by the TPer or that result in some other failure condition MAY return an empty method result (the output result is an empty list) and an error code. Unrecognized method invocation attempts outside of Regular sessions SHALL be ignored by the TPer – in these cases, no response is sent.

Session Manager protocol layer method invocations that are recognized but fail SHALL result in the normal response format for that method, accompanied by an error status code. Session startup methods that fail in this way SHALL have returned the expected method response, but that method

SHALL have only the identifying parameters (Host, SP) and an error status code. If the identifying parameters (particularly the Host parameter) are invalid (i.e. of the incorrect type), the TPer MAY ignore the method.

The TPer MAY begin sending the response as soon as enough parameters have been received to prepare a response.

3.2.4.3 Method Result Retrieval Protocol

A method is invoked by tokenizing the method call and its parameters as described in previous sections, using the token encoding format and Subpacket-Packet-ComPacket format. The host sends the ComPacket to the TPer in an IF-SEND command. Multiple IF-SEND commands MAY be required to encompass the entirety of a method invocation or series of method invocations and their related data.

The host then polls the TPer by transmitting IF-RECV commands. When the TPer has packaged its response, it transmits the tokenized results to the host in the payload of an IF-RECV command. Multiple IF-RECV commands MAY be required to retrieve all of the results of a particular method invocation or series of method invocations.

For additional information on the operation of the IF-SEND and IF-RECV commands, see the descriptions for those commands as detailed in the appropriate interface specifications.

3.2.5 Tables

Tables SHALL be stored in SP-specific parts of the secure storage area of the TPer. The SP-related secure storage area(s) of a TPer SHALL only be accessible via the host interface-specific IF-SEND and IF-RECV commands. Table content SHALL NOT, unless otherwise stated, be part of the User Addressable Logical Block Address space on the Storage Device and therefore is not affected by the partitioning or formatting of the Storage Device by the host operating system.

Begin Informative Content

All persistent data for SPs are stored in tables – the only data for an SP that persists past the end of a session is the data that is stored in tables. Tables survive operations on user-areas, such as reformatting.

A table is defined as a grid with columns and addressable rows. At each column and row intersection there is a cell. All the cells in a column have the same type. The column types for a host-created table are specified at table creation.

For some SSCs, the number of rows in a table whose size is not specified is completely determined when it is created (additional rows are not able to be allocated), but other SSCs define tables whose size is not specified with a dynamically allocable number of rows. If an SSC permits additional rows to be added to a table, then the number of rows specified at table creation is the initial number of rows allocated for that table.

End Informative Content

A table name or table column name MAY be up to 32 bytes in length. By convention, the names assigned in this document consist of ASCII characters, the first of which is a letter and others are letters, digits or underscores. Adjacent underscores do not occur. All names are case sensitive.

Within an SP, tables MAY be created and deleted. For each table, rows MAY be created and deleted (except within a Byte table – see 3.2.5.1), but columns are created only when the table is created. Tables MAY contain zero or more rows. A specific Security Subsystem Class MAY disallow the creation of any of these.

Each SP has a set of metadata tables (such as the `Table` table, `Column` table, etc.) that describes all the tables of the SP including the metadata tables themselves.

Access control provides a means to limit the methods that MAY be successfully invoked on tables, or particular rows or cells of tables.

Some table columns represent control points for functionality provided by an SP, either based on the templates incorporated into the SP, or on the underlying TPer implementation. If the functionality represented by a particular column or set of columns as defined in this Specification is not provided by an SP, then access to the table columns that represent that functionality MAY be restricted.

3.2.5.1 Kinds of Tables

There are two kinds of tables:

a. **Byte table.** Byte tables provide raw data storage. A byte table has one unnamed column of type `bytes_1`. The address of the first row in a byte table is 0. Upon creation, the value of all cells in a byte table SHALL be `0x00`. The rows of a byte table SHALL NOT be allocated or freed (i.e. via `CreateRow` or `DeleteRow`). Byte table rows are addressed by row number.

b. **Object table.** Object tables provide storage for data that binds a set of methods and access controls to that data. When a table is created it SHALL be allocated a fixed number of fixed-size columns. Zero or more columns are designated as the unique set of values (see 3.2.5.4).

For Object tables:

a. A newly created table is initially empty and rows SHALL be created using the `CreateRow` method, before they are usable.

b. There is always a `UID` column of type UID. In object tables, rows are addressed by UID.

3.2.5.2 Objects

Begin Informative Content

An object is any row of an object table. The particular object type is defined by the object table in which the object occurs. The columns of the object table define the contents of each object in it.

For a specific SP, there are methods on the SP itself, methods that act on the tables and have the whole table as their possible scope, and methods for each of the objects within the SP. Object-specific ACLs are applied to the methods capable of manipulating that object's data (see 3.4.2).

End Informative Content

3.2.5.3 Unique Identifiers (UIDs)

Each object table has a column named `UID`. This column contains an 8-byte unique identifier for that row. Each row has an SP-wide unique value in this column. This value is never shared with another row, and is never reused by that SP. The TPer SHALL guarantee that UIDs are unique across the entire SP anytime that a UID is generated, and that UIDs SHALL NOT be re-used even if an object is deleted and the UID is no longer in use.

The `UID` column is present to provide anti-spoofing capability, and to provide a means to address these rows. New UIDs are assigned when rows are created and old values are discarded when rows are deleted. If all UIDs have been used, no more rows are able to be created.

Each table is also represented by a UID. A table's UID is derived from the UID of that table in the `Table` table. The `Table` table is an object table in which each row is a table descriptor object that stores metadata about the associated table.

The bytes in a UID SHALL be utilized as follows:

a. The first four bytes of a table row's UID SHALL be the "containing table" portion of the UID and the last four bytes SHALL be assigned in a TPer-specific manner.

b. UIDs of tables SHALL be assigned as follows:

   i. The UIDs of table descriptor objects (the table's row in the `Table` table) SHALL be `0x00 0x00 0x00 0x01 XX XX XX XX`, where `XX XX XX XX` represents the values assigned by the TPer to that object's UID, or assigned by this specification or an

SSC for pre-defined tables. For example, The `Table` table's UID SHALL be `0x00 0x00 0x00 0x01 0x00 0x00 0x00 0x01`

ii. The UID used to reference the actual table (rather than that table's row in the `Table` table) SHALL be `XX XX XX XX 0x00 0x00 0x00 0x00`, where `XX XX XX XX` are the last four bytes of the UID from that table's row in the `Table` table. Four `0x00`'s as the last four bytes of a UID that does not have four `0x00`'s at the beginning are references to a table.

iii. All object UIDs SHALL have their high four bytes be the high four bytes of the containing table's UID. So, references to rows in a table are assigned UIDs based on the UID of the containing table. For instance, references to the rows in table `XX XX XX XX 0x00 0x00 0x00 0x00` are assigned UIDs `XX XX XX XX yy yy yy yy` where the first four bytes of the containing table UID and of the row are the same.

All UIDs with their first four bytes equal to `0x00 0x00 0x00 0x00` are reserved for use by the TCG and SHALL NOT be assigned by the TPer.

When necessary to refer to the SP with a UID, as when an SP method is invoked, a UID of `0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01` is reserved to signify "this SP".

For each table defined in this specification, UIDs with last four bytes between `0x00 0x00 0x00 0x01` and `0x00 0x01 0x00 0x00` SHALL be reserved for use by the TCG.

A NULL UID reference is all zeroes (`0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00`). This is used to indicate that no object is being referenced.

3.2.5.4 Unique Column Value Combinations

In addition to the `UID` column, an object table MAY also have one or more columns designated by the host (for host-created tables) or by the specification (for tables specified in this document) as required to be unique.

If a table has a column or set of columns defined as unique, then each row of the table SHALL have a value or combination of values in the indicated column(s) that is unique within the table for those column values. When more than one column is marked as participating in this uniqueness requirement, each of these columns participate in the unique value ("multi-column unique value").

The TPer is not required to keep rows of the table sorted by these unique values.

3.2.6 Templates

Begin Informative Content

This document covers the following Templates:

a. **Base Template:** Provides the tables and methods common for all SPs.

b. **Admin Template**: Provides administrative control over other SPs and the TPer settings as a whole, and control over Issuance of new SPs.

c. **Clock Template:** Contains tables and methods specialized for forensic and cryptographic clocks.

d. **Crypto Template:** Contains functional extensions to the Base SP cryptographic and procedural capabilities.

e. **Locking Template:** Provides tables and methods for storage encryption/decryption and read/write lock state control.

~~f.~~ **Log Template:** Contains tables and methods specialized to forensic logging.

End Informative Content

3.3 Interface Communications

Begin Informative Content

The TCG Storage Architecture Core Specification describes the architecture and main command set in an interface protocol-independent way. The implementation of this specification on various interfaces does have some differences (see [2]).

This section abstracts out the common features of these commands that serve as a requirement for an interface protocol to implement the present specification. These sections address communications on protocols `0x01` and `0x02` only (see Table 25).

The following assumptions are made regarding the interface commands:

a. The interface commands have two parts: (1) a command block and (2) a data payload. Each host interface protocol has its own minimum payload size. The payload size is not related to the 'logical block size' of the user data on the medium of the Storage Device. See the definitions of IF-SEND and IF-RECV for details.

b. There is at least one command in the interface protocol that transfers data from the host to the Storage Device. These commands are called IF-SEND.

c. There is at least one command in the interface protocol that transfers data from the Storage Device to the host. These commands are called IF-RECV.

The abstracted command block of the interface commands are described in the format defined in Table 24.

The mapping of the IF-SEND and IF-RECV commands to specific interface protocol commands are described in [2].

End Informative Content

Table 24 Interface Command – Command Block

Command	Either IF-SEND or IF-RECV.
Protocol ID	Between `0x01` and `0x06` (see Table 25)
Transfer Length: at least 2 bytes (the length of this field varies by host interface)	The amount of data to be transferred.
ComID	The ComID to be used, for Protocol IDs `0x01`, `0x02`, `0x06` (see 3.3.2 and 3.3.3).

Table 25 Protocol IDs

ID	Description
0x00	See [6]
0x01	Defined in this document
0x02	Defined in this document
0x03	Reserved for TCG
0x04	Reserved for TCG
0x05	Reserved for TCG

0x06	Reserved for TCG
All others	See [6]

IF-SEND and IF-RECV commands to a Protocol ID between 0x00 and 0x06 that the TPer does not support SHALL result in the IF command failing with a status of Invalid Security Protocol ID Parameter (see [2]).

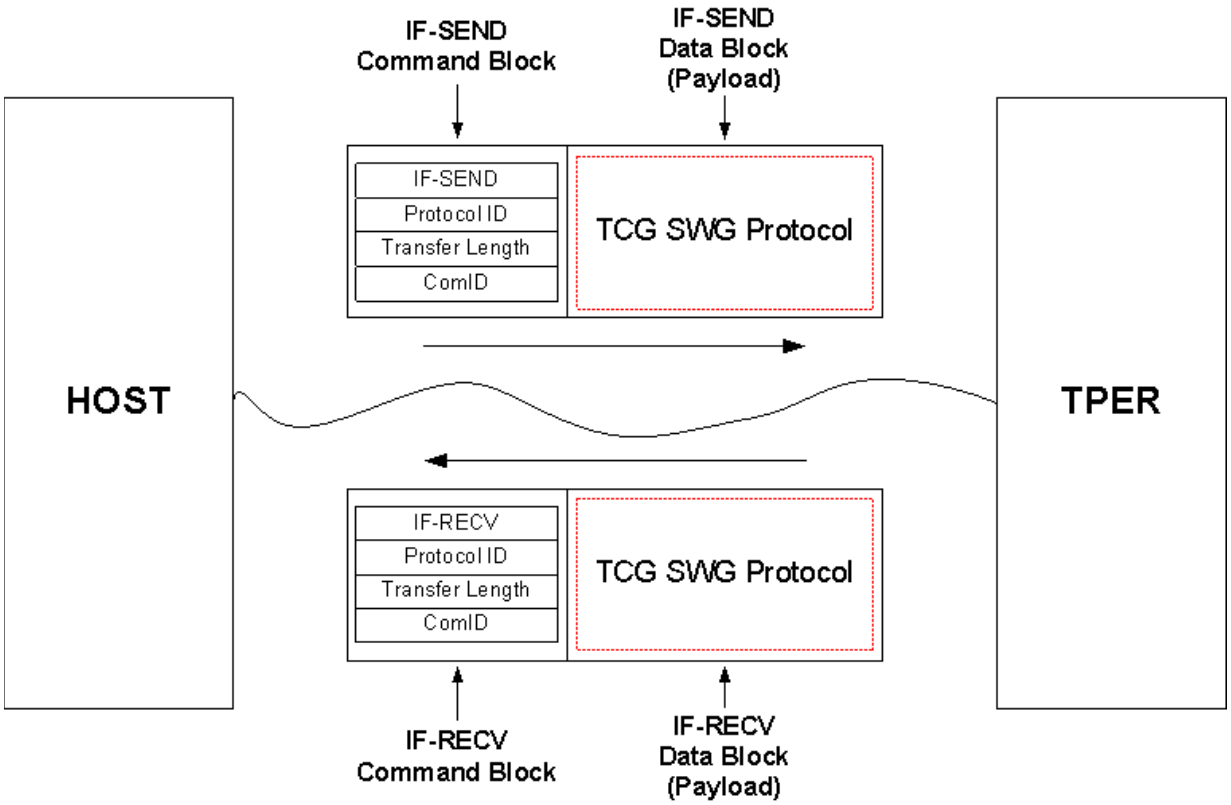
3.3.1 Communicating With the TPer Through the Interface Protocol

Begin Informative Content

The communication between the Host and the TPer takes place through the use of IF-SEND and IF-RECV as illustrated in Figure 3 . Most of the useful communication between a Host and a TPer is encapsulated in the payload of these commands.

End Informative Content

Figure 3 TPer-Host Communication



3.3.2 The ComID

Begin Informative Content

The ComID is used to select the correct response data for the host. The ComID allows the TPer to identify the caller of the IF-RECV command and appropriately populate the payload for the command.

For dynamic assignment of ComIDs, in order to open a session with a particular SP on a TPer, the host application starts by requesting a ComID from the TPer if it doesn't already have one that's active.

The TPer then issues a ComID to the host application. Once the host application has a unique ComID, the host is able to initiate the process of starting a session.

Once the session is started, the TPer associates the session number with the ComID. In this way, when an IF-RECV is sent to the TPer using Protocol ID of `0x01`, the TPer is able to respond with a payload containing only the packets for the session numbers associated with the ComID. This allows for multiple applications to be simultaneously communicating with the TPer without interfering with one another.

In some situations it is useful to allow for a single entity, called the Host Session Manager, to manage the TPer communications for a set of different applications running on the host.

To the TPer, communication with a single host application is no different than communication with a Session Manager that acts as an intermediary for multiple host applications with which the TPer is communicating.

End Informative Content

To enable a single host application to manage communications for multiple other applications, multiple sessions MAY be opened with a single ComID. All the sessions opened with a given ComID SHALL be associated with it.

An application MAY open a single session to the TPer for itself, multiple sessions for itself, multiple sessions for one or more other applications, multiple sessions for itself and one or more other applications, or any other combination.

When an IF-RECV is sent to the TPer using a particular ComID, the TPer SHALL respond by putting packets from the sessions associated with the ComID into the response. If there are more pending responses from the various sessions associated with the ComID than fits the IF-RECV, it is up to the TPer to determine which packets to include.

Begin Informative Content

The number of packets/subpackets that are included in the response is a function of the amount of available responses, the transfer length of the command, and the flow control mechanism. The amount of data still remaining to be retrieved and the minimum transfer length required to retrieve at least one packet, at the time the ComPacket was generated, is reported in the ComPacket header.

End Informative Content

3.3.3 ComID Management

Begin Informative Content

A mechanism is required to enable dynamic management of ComIDs so as to minimize the chances of two host applications using the same ComID in the rare occasions in which there are ComID conflicts. Support for dynamic ComID management is SSC-specific.

End Informative Content

ComIDs SHALL be assigned based on the allocation presented in Table 26.

Table 26 ComID Assignments

ComID	Description
0x0000	Reserved
0x0001	Level 0 Device Discovery
0x0002-0x07FF	Reserved for TCG
0x0800-0x0FFF	Vendor Unique
0x1000-0xFFFF	ComID management (Protocol ID=0x01, and 0x02) -

ComID	Description
	these are “non-reserved” ComIDs.

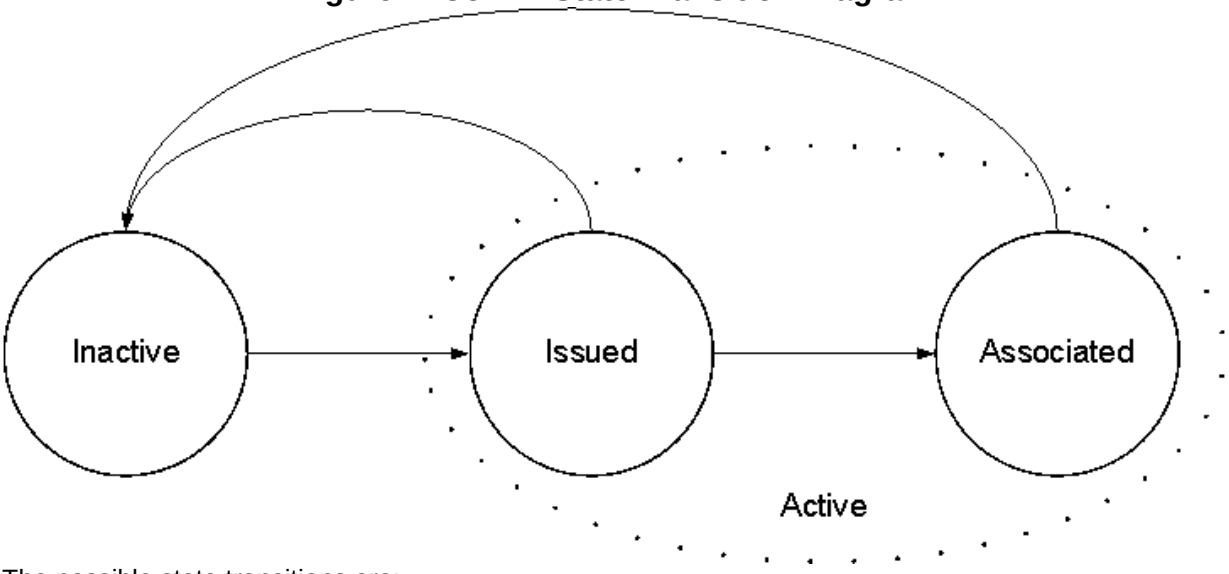
The lower 4096 out of the possible ComIDs SHALL be reserved – 0-2047 are reserved for TCG use/assignment, and 2048-4095 are reserved as vendor-unique. The other, non-reserved ComIDs SHALL be used for multiplexing the TPer responses to IF-RECVs.

A ComID SHALL be in one of the following three states:

1. **Inactive**: The ComID has not been assigned to anyone since the last hardware reset or power cycle, or because the ComID was retired due to all sessions on the ComID being closed.
2. **Issued**: The ComID has been issued (it was returned to the host during a successful completion of a GET_COMID command) but no sessions have been started using this ComID.
3. **Associated**: One or more open sessions are associated with the ComID.

ComIDs that are either in the Issued state or the Associated state are considered Active. The state diagram in Figure 4 shows these states and the possible transitions among them.

Figure 4 ComID State Transition Diagram



The possible state transitions are:

a. **Inactive to Issued**: A ComID SHALL transition from the Inactive state to the Issued state when it is returned to the host during a successful execution of the GET_COMID command.
b. **Issued to Associated**: A ComID SHALL transition from the Issued state to the Associated state once a session is open using that ComID. This occurs at the point when session startup has successfully completed.
c. **Issued to Inactive**: A ComID SHALL transition from Issued to Inactive when any one of the following conditions hold:
    i. There is a hardware reset or power cycle.
    ii. The host does not start a session using the ComID within MaxComIDTime from the ComID being issued. MaxComIDTime defines a limit on the amount of time a ComID is able to exist in the Issued state without an active session. A TPer's MaxComIDTime value is retrieved using the `Properties` method. Support for MaxComIDTime is SSC-dependent.
d. **Associated to Inactive**: A ComID SHALL transition from Associated to Inactive when any

of the following conditions are met:

i. There is a hardware reset or power cycle.
ii. After all sessions associated with the ComID are closed, and no session startup activities are in progress.

In order to minimize the possibility of conflict, the ComID issuance mechanism SHALL have the following two characteristics:

a. A ComID that is in an active state SHALL NOT be issued again. That is, only ComIDs that are in the inactive state SHALL be returned to the host as a response to the GET_COMID command.

b. The TPer SHALL issue ComIDs in a sequential manner (wrapping around cyclically as needed).

In addition to the above transitions, the TPer MAY transition a ComID to the Inactive state at any time for any reason.

3.3.3.1 Extended ComID

Begin Informative Content

Despite all the mechanisms in place, there is always the possibility that some application holds on to its ComID for an extended period of time and not recognize that the ComID has become inactive and (possibly) subsequently issued to another application. Since there are only 61440 normal non-reserved ComIDs, the probability of this occurring is not small enough to be neglected. To help deal with this issue the TPer makes use of Extended ComIDs.

End Informative Content

Extended ComIDs SHALL be 4 bytes long and have the first 2 bytes equal to the ComID. The second 2 bytes make up the ComID Extension.

The MSB of the ComID is the first byte (MSB) of the Extended ComID, and the LSB of the ComID is the second byte of the Extended ComID. The TPer arbitrarily generates the remaining 2 bytes (the ComID Extension) every time a ComID is issued. The GET_COMID command returns the 4-byte Extended ComID to the host. There MAY be many Extended ComIDs associated with the same ComID over the life of the TPer. The ComID Extension associated with reserved ComIDs (0-4095) SHALL always be `0x0000`.

The ComID Extension value of `0xFFFF` is reserved to indicate that the host has attempted to communicate using an inactive ComID.The ComID Extension for ComIDs that are not dynamically assigned by the TPer SHALL be set to `0x0000`.

The Extended ComID SHALL be in one of the following states

1. **Inactive**: The associated ComID is in the inactive state.
2. **Issued**: The Extended ComID has been issued (it was returned to the host during a successful completion of a GET_COMID command) but no sessions have been started using the associated ComID.
3. **Associated**: One or more open sessions were open with the ComID. These sessions are said to be associated with the Extended ComID.
4. **Invalid**: The Extended ComID has not been issued since the last power cycle/reset, or has become inactive and there exists another Extended ComID with the same associated ComID in one of the active states (Issued or Associated).

The Extended ComID is used to determine if an application is using a conflicting ComID, i.e., if the ComID the application is using has become inactive and subsequently assigned to another application. When this happens, the application's Extended ComID SHALL be invalid. When the application makes an inquiry to the TPer using the Extended ComID, the TPer SHALL respond with an indication that the Extended ComID is invalid.

When the TPer receives a ComPacket (via IF-SEND) that contains a ComPacket with an invalid Extended ComID, the TPer SHALL ignore and discard the payload of the ComPacket.

When the host receives a ComPacket (via IF-RECV) that contains an unexpected Extended ComID, this is an indication to the host that it is using an invalid Extended ComID and that the ComID is being used by another host or application. The host SHOULD assume that any sessions it had open on that ComID have been aborted. To resume communications with the TPer, the host SHALL acquire a new ComID.

3.3.3.2 IF-SEND to Inactive or Unsupported Reserved ComID

If the host sends an IF-SEND command to the TPer with a ComID value in the non-reserved range (see Table 26), and the ComID is in the Inactive state:

a. If the TPer supports dynamic ComID allocation, the TPer SHALL:
    i. Accept all data in the payload of the IF-SEND command and complete the command normally with good status (provided there are no other errors which would cause the command to abort at the interface level)
    ii. Ignore and discard the entire payload of the IF-SEND command.
b. If the TPer does not support dynamic ComID allocation, the TPer SHALL:
    i. Report “Other Invalid Command Parameter”[2] **OR**
    ii. Perform the action described above for TPers that support dynamic ComID allocation.

If the host sends an IF-SEND command to the TPer with a ComID value in the reserved range(see Table 26), and the ComID is not supported by the TPer, the TPer SHALL:

a. Report “Other Invalid Command Parameter” [2].

3.3.3.3 IF-RECV to Inactive or Unsupported Reserved ComID

If the host sends an IF-RECV command to the TPer with a ComID value in the non-reserved range (see Table 26), and the ComID is in the Inactive state:

a. If the TPer supports dynamic ComID allocation, the TPer SHALL:
    i. Respond to the IF-RECV with a zero-length ComPacket (a ComPacket header only) in the IF-RECV payload. The fields in the ComPacket header SHALL contain:
        1. ExtendedComID = {<ComID from SP_Specific field of CDB>, `0xFFFF`}
            a. Note: The value of `0xFFFF` in bits 15 through 0 of the ExtendedComID field is an indication to the host that the ComID it is attempting to use is inactive, and that it should not expect to receive any data on that ComID.
        2. OutstandingData = `0x00000000`
        3. MinTransfer = `0x00000000`
        4. Length = `0x00000000`
    ii. Complete the command normally with good status (provided there are no other errors which would cause the command to abort at the interface level)
b. If the TPer does not support dynamic ComID allocation, the TPer SHALL:
    i. Report “Other Invalid Command Parameter”[2] **OR**
    ii. Perform the action described above for TPers that support dynamic ComID allocation.

If the host sends an IF-RECV command to the TPer with a ComID value in the reserved range (see Table 26), and the ComID is not supported by the TPer, the TPer SHALL:

a. Report “Other Invalid Command Parameter”[2].

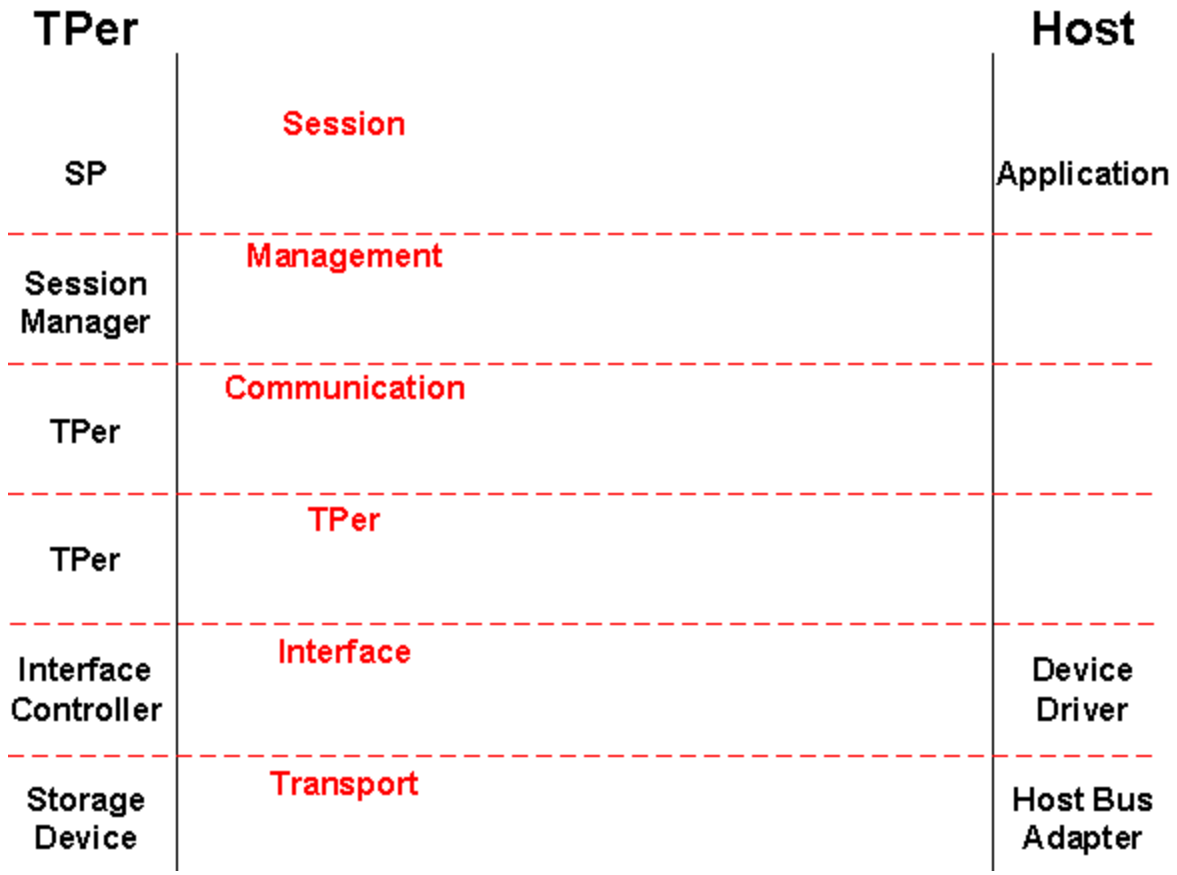
3.3.4 Protocol Layers

Begin Informative Content

In order to describe the overall process for establishing communication with the TPer and initiating a session to an SP, it is necessary to partition the protocol stack into layers. The commands in each layer differ in the amount of functionality available. The lower level allows only one-way communication (TPer to host) and uses only simplistic byte field responses. The higher layers have two-way communication and use packets and methods.

Figure 5 depicts the protocol layers.

Figure 5 TPer-Host Communication Protocol Layers



a. **Session layer**: This layer is entered when a session is successfully established between the host application and an SP in the TPer. Most of the commands and functionality specified in the TCG Storage Architecture Core Specification operate in this layer. Payloads in this layer are packetized and tokenized.

b. **Management layer**: This layer deals with establishing a session between an SP and a host application. Payloads in this layer are packetized and tokenized.

c. **Communication (Com) layer**: In this layer the host application already has an assigned ComID that is used for establishing two-way communication. It is a bidirectional communication/control layer. This layer is used for management of ComIDs and dealing with error conditions and other Storage Device management issues.

d. **TPer layer**: This is the first entry point to the TPer. This is a "one-way" communication layer. That is, only IF-RECV commands are dealt with in this layer. The host application does not have a ComID yet. There is a set of reserved ComIDs that is used to invoke special commands at this layer.

e. **Interface layer**: This portion of the stack contains the protocol for allowing the host to control a specific Storage Device. The interface protocol must support IF-SEND and IF-RECV, i.e., have commands with the properties that are required for these TCG commands.

f. **Transport layer**: This portion of the stack is responsible for transporting the data from one particular host to one particular Storage Device and vice-versa. An example is Fibre Channel.

End Informative Content

3.3.4.1 Transport Layer

Begin Informative Content

This layer of the protocol stack is responsible for transmitting the data from one particular host to one particular Storage Device and vice-versa. There are no specific interactions with this layer described in the TCG Core Specification. The only requirement is that this layer interact with the Interface layer in such a way as to guarantee that the order of commands sent from a single host to a single Storage Device are preserved.

End Informative Content

3.3.4.2 Interface Layer

The commands at this layer are the IF-SEND and the IF-RECV commands. The interface controller on the Storage Device SHALL identify these commands and send them to the TPer level.

All commands that map to IF-SEND and all the commands that map to IF-RECV that have the protocol ID field in the set {`0x01, 0x02, 0x03, 0x04, 0x05, 0x06`} SHALL be sent to the TPer.

3.3.4.3 TPer Layer

Begin Informative Content

This is the entry point into the TPer. This layer has very limited functionality. Commands at this layer are designed to be used without ComIDs. In particular, the command used to request a ComID, GET_COMID, is dealt with in this layer.

The only commands dealt with in this layer are IF-RECV commands with some specific reserved ComIDs and protocol ID settings. All other commands are passed up to the Communication Layer.

End Informative Content

The commands specified in the TPer layer and in the communication layer SHALL utlilize Protocol ID = `0x02`.

3.3.4.3.1 GET_COMID

The command block for the GET_COMID command is defined in Table 27. The payload of the GET_COMID command is defined in Table 28.

Table 27 GET_COMID Command Block

FIELD	VALUE

Command	IF-RECV
Protocol ID	02
Transfer Length	00 01
ComID	00 00

Table 28 GET_COMID Payload

BYTE	FIELD	VALUE
0 to 3	Extended ComID	Allocated ComID

a. The first 4 bytes of the payload SHALL be the Extended ComID. The first two bytes of the Extended ComID are the ComID. If the TPer is not able to assign a new ComID for any reason it SHALL return all zeroes in the Extended ComID field.

b. The TPer SHALL NOT assign the value of `0xFFFF` as the ComID Extension.

c. See [2] for padding requirements.

3.3.4.4 Communication Layer

Begin Informative Content

The Communcation Layer provides a mechanism for two-way communication between the host application and the TPer. The primary purpose of the communication at this layer is to manage the allocated ComID and to verify the validity of the allocated ComID.

Communication at this layer occurs using IF-SEND and IF-RECV commands using Protocol ID `0x02`. The host must have a ComID that has been assigned by the TPer using the GET_COMID command available at the TPer Layer.

If the host application uses a ComID that is not valid or has become invalid since its last usage (see 3.3.3.1), the host application may query the TPer at this layer to retrieve ComID state without raising exceptions on lower layers such as the Interface or TPer layers. This allows host applications to verify validity of ComIDs without disturbing the operation of the TPer.

End Informative Content

3.3.4.4.1 Communication Layer Protocol

Begin Informative Content

The commands for communication with the TPer at this layer are as follows:

a. HANDLE_COMID_REQUEST: IF-SEND to ComIDs with the caller's Extended ComID passed as the first 4 bytes of the payload.

b. GET_COMID_RESPONSE: IF-RECVs on ComIDs previously allocated by the TPer.

See 3.3.4.7 for information about the usage of these commands.

End Informative Content

3.3.4.5 Management Layer

Begin Informative Content

Commands dealt with in this layer are IF-SEND and IF-RECV with Protocol ID = `0x01` and with a valid active ComID.

This is the first layer that makes use of tokenized and packetized payloads. Communications in this layer occur between the TPer Session Manager (TSM) and the Host Session Manager (HSM). All communications happen within Control Sessions.

The Control Session associated with a particular ComID starts as soon as the ComID is issued. When the ComID becomes inactive, the Control Session is terminated. The flow control for the Control Session is performed in the same manner as the flow control for Regular Sessions, with the difference that the communication is between the TSM and the HSM and these entities are responsible for the flow control.

One of the main tasks of this layer is to manage the startup of Regular Sessions. During this process, the TSM and the HSM assign the TSN and the HSN that compose the SN for the Session to be created.

When the process is initiated the HSM assigns an HSN (i.e. newHSN). The HSM has the opportunity to make sure newHSN is different from any other HSNs in use by other sessions managed by it, though this is not required.

End Informative Content

Once the TSM processes the `StartSession` method and returns the `SyncSession` response, the Regular Session SHALL be considered open for the case of sessions that do not require challenge-response and/or key exchange. For sessions that require challenge-response and/or key exchange, the Regular Session SHALL be considered open when the TSM finishes processing the `StartTrustedSession` and has prepared the `SyncTrustedSession` response.

3.3.4.6 Session Layer

In this layer all communications SHALL occur within Regular Sessions.

3.3.4.7 Communication Layer Commands

For any given ComID, the host is expected to issue HANDLE_COMID_REQUEST and GET_COMID_RESPONSE commands in pairs. Consecutive GET_COMID_RESPONSE commands SHALL return data corresponding to the last HANDLE_COMID_REQEUST received by the TPer. The response MAY be regenerated by the TPer at the time of receipt of the command.

3.3.4.7.1 HANDLE_COMID_REQUEST

This command is used to inquire about or manage the state of the ComID previously allocated by the TPer. The command block for the HANDLE_COMID_REQUEST command is defined in Table 29.

Table 29 HANDLE_COMID_REQUEST Command Block

FIELD	VALUE
Command	IF-SEND
Protocol ID	02
Transfer Length	00 01
ComID	Allocated ComID

The payload sent by the host to the TPer, at the minimum, consists of the 4-byte Extended ComID and a Request code. Additional fields MAY be required for some request codes. Currently two request codes are defined: VERIFY_COMID_VALID and STACK_RESET.

3.3.4.7.2 GET_COMID_RESPONSE

This command is used to retrieve the response of the TPer to a previous HANDLE_COMID_REQUEST command. The command is sent to the ComID for which the status is requested. The command block for the GET_COMID_RESPONSE command is defined in Table 30

Table 30 GET_COMID_RESPONSE Command Block

FIELD	VALUE
Command	IF-RECV
Protocol ID	02
Transfer Length	00 01
ComID	Request_ComID

The Transfer Length is the amount of data that the TPer SHALL send in response to the command. If the actual length of the response data is smaller, then the TPer SHALL pad the data with zeros. If the actual length of the response data is larger, then the TPer SHALL only send the requested amount of data.

Bytes 10-11 of the payload contain the length of the response data. The host MAY use this information to repeat the response command with a transfer length that fits the available data.

3.3.4.7.3 No Response Available

If no response is currently available to the GET_COMID_RESPONSE command, "No Response Available" is returned. "No Response Available" is defined in Table 31.

Table 31 No Response Available

BYTES	FIELD	VALUE
0 to 3	Extended ComID	Allocated ComID
4 to 7	Request Code	00 00 00 00
8 to 9	Reserved	00 00
10 to 11	Available Data Length in bytes	00 00
12 to TRNSFLEN - 1	Reserved	00

3.3.4.7.4 VERIFY_COMID_VALID

On receiving this request, the TPer checks if the ComID sent in the payload matches any of the ComIDs currently active in the TPer.

The command is delivered in the payload of the HANDLE_COMID_REQUEST command. The response is reported in the payload of the next GET_COMID_RESPONSE command sent to the requested ComID.

The VERIFY_COMID_VALID command is defined in Table 32. TRNSFLEN is defined as number of bytes transferred via the interface.

Table 32 VERIFY_COMID_VALID Request

BYTES	FIELD	VALUE
0 to 3	Extended ComID	Allocated ComID
4 to 7	Request Code	00 00 00 01

BYTES	FIELD	VALUE
8 to TRNSFLEN - 1	Reserved	zero

The payload built by the TPer in response to the VERIFY_COMID_VALID command is defined in Table 33.

Table 33 VERIFY_COMID_VALID Command Response

BYTE	FIELD	VALUE
0 to 3	Extended ComID	Requested ComID
4 to 7	Request Code	00 00 00 01
8 to 9	Reserved	00 00
10 to 11	Available Data Length in bytes	00 22
12 to 15	Current state of Extended ComID	00 00 00 00 = Invalid, 00 00 00 01 = Inactive, 00 00 00 02 = Issued, 00 00 00 03 = Associated
16 to 25	Absolute time of allocation relative to last reset of TPer	10 byte format – see Table 34.
26 to 35	Absolute time of expiry relative to last reset of TPer	10 byte format – see Table 34.
36 to 45	Time since last reset of TPer	10 byte format – see Table 34.
46 to TRNSFLEN - 1	Reserved	00

The Extended ComID field value is the ComID being verified.

If the TPer does not support a real-time clock, the Time values in the VERIFY_COMID_VALID response (bytes 16 to 45) SHALL be all zeroes. If the TPer supports a real-time clock, the fields that report the time SHALL use the following format described in Table 34.

Table 34 Date Values

Value	Type	Range
Year	uinteger_2	1970 to 9999
Month	uinteger_1	1 to 12
Day	uinteger_1	1 to 31
Hour	uinteger_1	0 to 23
Minute	uinteger_1	0 to 59
Second	uinteger_1	0 to 59
Fraction	uinteger_2	0 to 999
Reserved	uinteger_1	0

If the current state of the ComID is reported as Invalid or Inactive, only the time since last reset of the TPer is valid in the data payload. If the ComID state is reported as Issued, or Associated, the time of expiry SHALL be greater than the time since last reset of the TPer.

3.3.4.7.5 STACK_RESET

This command is used to reset that state of the synchronous protocol stack (see 3.3.10).

The command is delivered in the payload of the HANDLE_COMID_REQUEST command. The ComID field identifies the ComID being reset. The response to the STACK_RESET is retrieved using the GET_COMID_RESPONSE command.

The STACK_RESET command is defined in Table 35.

TRNSFLEN is defined as number of bytes transferred via the interface. Reserved bytes SHOULD be set to zero and SHALL be ignored by both host and device.

The device SHALL return an “Invalid Transfer Length parameter on IF-SEND” TPer Error[2] if less than 8 bytes are sent to the device.

Depending on the SSC, if the ComID value in the IF-SEND for the HANDLE_COMID_REQUEST command represents a non Active ComID, the device SHALL respond as described in 3.3.3.2.

Table 35 STACK_RESET Command Request

BYTES	FIELD	VALUE
0 to 3	Extended ComID	Allocated ComID
4 to 7	Request Code	00 00 00 02
8 to TRNSFLEN - 1	Reserved	00

Once received, the TPer SHALL reset the protocol stack for the ComID value defined in bytes 0-3 of the command block payload. While resetting the stack, the Tper SHALL NOT process any command for that ComID received via an IF-SEND on Protocol ID 0x01. A Security Protocol stack reset results in:

1. All open sessions for that ComID SHALL be aborted. `CloseSession` methods SHALL NOT be prepared by the TPer;
2. All uncommitted transactions SHALL be aborted;
3. All pending session startup activities occurring on that ComID SHALL be aborted;
4. All TCG command and response buffers SHALL be invalidated for that ComID;
5. All related method processing occurring on that ComID SHALL be aborted;
6. The protocol stack SHALL reset to its initial state for that ComID only;
7. All communications properties (set via `Properties` method) and ComID associated properties for that ComID SHALL be reset to their default values;
8. No notification of these events SHALL be sent to the host.

The response SHALL be returned via the GET_COMID_RESPONSE (IF-RECV) command. The STACK_RESET command response payload is defined in Table 36.

If the STACK_RESET is still processing and another HANDLE_COMID_REQUEST is received, the STACK_RESET SHALL complete but a response for that STACK_RESET command SHALL NOT be available.

Table 36 STACK_RESET Command Response

BYTES	FIELD	VALUE
0 to 3	Extended ComID	Allocated ComID
4 to 7	Request Code	00 00 00 02
8 to 9	Reserved	00 00
10 to 11	Available Data Length in bytes	00 04
12 to 15	Success/Failure	00 00 00 00/ 00 00 00 01
16 to TRNSFLEN - 1	Reserved	00

Success (`0x00000000`) indicates that the protocol stack has been reset for the specified ComID. Failure (`0x00000001`) indicates tha the protocol stack has not been reset for the specified ComID.

The response SHALL be cleared from the response buffer if one of the following conditions is true:

a. The host retrieves the entire response via the GET_COMID_RESPONSE command;
b. The device is hard-reset or power-cycled.
c. Another HANDLE_COMID_REQUEST is made for that ComID.

The device SHALL return “No Response Available” if:

a. No HANDLE_COMID_REQUEST command preceded the GET_COMID_RESPONSE command;
b. An error is detected in the HANDLE_COMID_REQUEST command payload.

If no Handle_ComID_Request was sent, the Extended ComID field SHALL contain zeroes.

The "No Response Available" payload is defined in Table 31.

The device SHALL return “Pending” if:

a. The host retrieves the command result via the GET_COMID_RESPONSE command while the stack reset is in progress for that specific ComID.

The "Pending" payload is defined in Table 37.

Table 37 STACK_RESET Pending

BYTES	FIELD	VALUE
0 to 3	Extended ComID	Allocated ComID
4 to 7	Request Code	00 00 00 02
8 to 9	Reserved	00 00
10 to 11	Available Data Length in bytes	00 00
12 to TRNSFLEN - 1	Reserved	00

3.3.5 Capability Discovery

Discovery is a process that provides a way for the Host to examine the SD's configurations and capabilities.

There are three levels of discovery:

a. **Level 0:** This discovery level discloses basic SD status and configuration. This discovery request is sent as an IF-RECV command (see 3.3.6).

b. **Level 1:** This discovery level discloses basic TPer capabilities via the `Properties` method (see 5.2.2.1)

c. **Level 2:** This discovery level uses the `Get` method (see 5.3.3.6) to retrieve table cell values under access control as defined by the ACLs in each SP's `AccessControl` table (see 5.3.2.7), and the associated ACEs in each SP's `ACE` table (see 5.3.2.8).

3.3.6 Level 0 Discovery

The Level 0 Discovery command provides a host with some basic information about TPer capabilities, both current and potential. More detailed information is obtainable through SP operations (see 3.3.7).

3.3.6.1 IF-SEND Command

IF-SEND command, with

Protocol ID = `0x01`
ComID = `0x0001`
Transfer Length= (any length)

There is no IF-SEND command defined for Level 0 Discovery, so the TPer SHALL transfer all of the data from the host, SHALL discard it, and return 'good' status to the host.

3.3.6.2 IF-RECV Command

IF-RECV command, with

Protocol ID = `0x01`
ComID = `0x0001`
Transfer Length = maximum length of the Level 0 Discovery response data that the host elects to receive.

This IF-RECV command MAY be processed at any time, without regard to sessions or prior authentication.

If the Transfer Length is less than the size of the Level 0 Discovery response data that is available, the TPer SHALL return the requested amount of data, even if it is truncated.

If the Transfer Length is greater than the size of the Level 0 Discovery response data, the device shall pad according to the rules specified in [6] and [5].

The Level 0 Discovery response data (see Table 38) consists of a header field and zero or more variable length feature descriptors. A TPer SHALL NOT include feature descriptors for features that it does not implement. The data is not packetized.

Table 38 Level 0 Discovery Response Data Format

Byte \ Bit	7	6	5	4	3	2	1	0
0 – 47	Level 0 Discovery header (see Table 39)							
48 – n	Feature Descriptor(s) (see 3.3.6.3)							

Table 39 Level 0 Discovery Header Format

Bit / Byte	7	6	5	4	3	2	1	0
0	(MSB)	Length of Parameter Data						
1								
2								
3								(LSB)
4	(MSB)	Data Structure Major Version						
5								(LSB)
6	(MSB)	Data Structure Minor Version						
7								(LSB)
8	(MSB)	Reserved						
…								
15								(LSB)
16	(MSB)	Vendor Unique						
…								
47								(LSB)

3.3.6.2.1 Length of parameter data

Indicates the total number of bytes that are valid in the Level 0 Discovery header and all of the feature descriptors returned, not including this field.

3.3.6.2.2 Data Structure Major Version

This is the Major Version number of the Data Structure format of the Level 0 Discovery header returned. The value of this field SHALL be `0x0000`.

This value SHALL be incremented when non-backwards compatible changes are made to the header or to the format of the feature descriptors.

3.3.6.2.3 Data Structure Minor Version

This is the Minor Version number of the Data Structure format of the Level 0 Discovery header returned. The value of this field SHALL be `0x0001`.

This value SHALL be incremented when backwards compatible changes are made to Header or to the format of the feature descriptors.

3.3.6.2.4 Vendor Unique

These bytes are vendor specific.

3.3.6.3 Features - Overview

A feature is a set of capabilities that MAY be implemented in a TPer. A Host MAY discover the capabilities and properties of a TPer by examining its feature descriptors. Features that are implemented by a TPer SHALL be indicated by the presence of a feature descriptor.

The feature descriptors SHALL be returned in the Level 0 Discovery response data in order of increasing feature code values. Features that are not implemented SHALL NOT be returned.

Table 40 contains the list of defined feature codes.

Table 40 Feature Codes

Feature Code	Feature Name	Description
0000h	Reserved	
0001h	TPer feature	See 3.3.6.4
0002h	Locking feature	See 3.3.6.5
0003h – 00FFh	Reserved	
0100h – 03FFh	SSCs	
0400h - BFFFh	Reserved	
C000h - FFFFh	Vendor Unique	Vendor specific features

All feature descriptors SHALL conform to the general format defined in Table 41.

Table 41 Feature Descriptor Template Format

Bit / Byte	7	6	5	4	3	2	1	0
0	(MSB)	Feature Code						
1								(LSB)
2	Version				Reserved			
3	Length							
4 – n	Feature Dependent Data							

3.3.6.3.1.1 Feature Code

The Feature Code field SHALL identify a feature (see Table 42) implemented by the TPer.

3.3.6.3.1.2 Version

The Version field describes the format of the data returned. Future versions of a feature SHOULD be backward compatible; incompatible changes SHOULD be included in a different feature.

3.3.6.3.1.3 Length

The Length field indicates the length of the Feature Dependent Data (in bytes) that follow this header. This field SHALL be an integral multiple of 4.

3.3.6.4 TPer Feature

This information reports support for various TPer parameters. This mandatory feature SHALL always be returned in the Level 0 Discovery response.

These parameters indicate whether the TPer supports a variety of features. Having a given “support’ flag true does not imply that the feature is required or enabled. Actually enabling a feature MAY require setting of host properties by invoking the `Properties` method.

The Feature Code value for the TPer feature is `0x0001`.

Table 42 TPer Feature Descriptor

Bit / Byte	7	6	5	4	3	2	1	0

Bit / Byte	7	6	5	4	3	2	1	0
0	(MSB)	Feature Code						
1								(LSB)
2	Version				Reserved			
3	Length							
4	Reserved	ComID Mgmt Supported	Reserved	Streaming Supported	Buffer Mgmt Supported	ACK/NAK Supported	Async Supported	Sync Supported
5 - 15	Reserved							

The Feature Code field SHALL be set to `0x0001`.

The Version field SHALL be set to `0x01`.

The Length field SHALL be set to `0x0C`.

3.3.6.4.1 Sync Supported

SyncSupported SHALL be set to one if the TPer supports the Synchronous Protocol (see 3.3.10), otherwise SyncSupported SHALL be cleared to zero.

3.3.6.4.2 Async Supported

AsyncSupported SHALL be set to one if the TPer supports the Asynchronous Protocol, otherwise AsyncSupported SHALL be cleared to zero.

3.3.6.4.3 ACK/NAK Supported

ACK/NAKSupported SHALL be set to one if the TPer supports transmission ACK/NAK flow control (see 3.3.8) or communications, otherwise ACK/NAKSupported SHALL be cleared to zero.

3.3.6.4.4 BufferMgmt Supported

BufferMgmtSupported SHALL be set to one if the TPer supports buffer management flow control (see3.3.8.2) for communications, otherwise BufferMgmtSupported SHALL be cleared to zero.

3.3.6.4.5 Streaming Supported

StreamingSupported SHALL be set to one if the TPer supports data stream encoding (see 3.2.2 and 3.2.3), otherwise StreamingSupported SHALL be cleared to zero.

3.3.6.4.6 ComID Management Supported

SHALL be set to one if the TPer supports ComID management using Protocol ID `0x02` (see 3.3.3), otherwise SHALL be cleared to zero.

3.3.6.5 Locking Feature

This information indicates support for an issued Locking template. This mandatory feature SHALL always be returned in the Level 0 Discovery response.

The Feature Code value for the Locking feature is `0x0002`.

Table 43 Locking Feature Descriptor

Bit / Byte	7	6	5	4	3	2	1	0
0	(MSB)	Feature Code						
1								(LSB)
2	Version				Reserved			
3	Length							
4	Reserved		MBR Done	MBR Enabled	Media Encryption	Locked	Locking Enabled	Locking Supported
5 - 15	Reserved							

The Feature Code field SHALL be set to `0x0002`.

The Version field SHALL be set to `0x01`.

The Length field SHALL be set to `0x0C`.

3.3.6.5.1 LockingSupported

LockingSupported SHALL be set to one if the TPer supports the Locking template; otherwise LockingSupported SHALL be set to zero.

3.3.6.5.2 LockingEnabled

LockingEnabled SHALL be set to one if an SP that incorporates the Locking template is in any state other than nonexistent; otherwise LockingEnabled SHALL be set to zero.

3.3.6.5.3 Locked

Locked SHALL be set to one if LockingEnabled is set to one, and one or more LBA ranges in the Locking table have either (`ReadLockEnabled`=True and `ReadLocked`=True) or (`WriteLockEnabled`=True and `WriteLocked`=True); otherwise Locked SHALL be set to zero.

3.3.6.5.4 MediaEncryption

MediaEncryption SHALL be set to one if the TPer supports media encryption; otherwise MediaEncryption SHALL be set to zero.

3.3.6.5.5 MBREnabled

MBREnabled SHALL be set to one if `LockingEnabled` is set to one, and the `MBRControl` and `MBR` tables are implemented, and that the `MBRControl` table's `Enable` column has a value of "True"; otherwise MBREnabled SHALL be set to zero.

3.3.6.5.6 MBRDone

MBRDone SHALL be set to one if MBREnabled is set to one, and the `MBRControl` table's `Done` column has a value of "True"; otherwise MBRDone SHALL be set to zero.

3.3.6.6 Common SSC feature information

This information is supplied as part of every reported SSC feature.

Table 44 Common SSC Information

Bit / Byte	7	6	5	4	3	2	1	0

Bit Byte	7	6	5	4	3	2	1	0
0 - 15	Reserved for common SSC parameters							

3.3.7 Sessions, Methods, and Transactions

3.3.7.1 Sessions

Begin Informative Content

There are two types of sessions:

a. Regular Sessions (or just Sessions): These are communication channels between a host application and an SP.

b. Control Sessions: These are between the TPer Session Manager (TSM) and the Host Session Manager (HSM).

The Host Session Manager is an abstract entity that represents the peer, on the host side, of the TPer Session Manager. The HSM could be an application that is routing traffic to several applications on the host or it could simply be a module in a given application that deals with establishing sessions with the TPer.

End Informative Content

All communications with an SP occurs within sessions. A session SHALL be started by a host and successfully ended by a host.

Normally the host application ends a session when it has finished its communication, but either the TPer or the host MAY abort a session at any time for any reason (see 3.3.7.1.5).

For a specific SP there MAY be any number of Read-Only sessions active simultaneously, but only one Read-Write session with a particular SP SHALL be open at a time. Read-Only and Read-Write sessions are mutually exclusive.

The existence of Read-Only sessions, the maximum number of simultaneous Read-Only sessions that are able to be opened to any SP, and/or the total number of open sessions available to a TPer SHALL be defined by Security Subsystem Class.

Except as noted, explicit changes to an SP made during a Read-Only session SHALL NOT be made permanent, even when the session closes successfully. Indirect changes, such as PIN blocking, log updates, etc., are noted where appropriate, and SHALL remain persistent.

3.3.7.1.1 Regular Sessions

Each Regular Session is identified by a distinct Session Number (SN). The SN is an 8-byte quantity composed of two subparts: the TPer Session Number (TSN) and the Host Session Number (HSN), each of which has 4 bytes. This is the value used in a packet's Session field.

SN = (TSN, HSN)

The HSN is assigned by the HSM. The HSM MAY assign HSNs in such a way as to make them unique for all of its communications with one or more TPers.

The TSN is assigned by the TSM. The TSM SHALL guarantee that all Regular Sessions associated with a particular ComID are assigned a different TSN. In addition, the TSM SHALL NOT assign any TSN in the range 0 to 4095 to a regular session. These TSNs are reserved by TCG for special sessions, of which the control session (0) is the only one currently defined.

3.3.7.1.2 Additional details regarding session startup can be found in 3.3.7.1.4 and 5.3.4.1.4.Control Sessions

All Session Manager Layer Methods SHALL be transmitted in packets where Packet.Session = `0x00000000_00000000`.

Session Manager layer methods are:

a. `Properties`
b. `StartSession`
c. `SyncSession`
d. `StartTrustedSession`
e. `SyncTrustedSession`
f. `CloseSession`

Once a session has started (the session startup protocol has completed successfully), data is able to be transmitted for that newly started session. The Packet.Session for that session SHALL be the concatenation of the TSN and HSN (see 3.3.7.1.1), where HSN is initially transmitted in the `StartSession` method and TSN is initially transmitted in the `SyncSession` method.

The life cycle of the Control Session is tied to the life cycle of the ComID, in that the Control Session associated with a particular ComID starts as soon as the ComID is issued. When the ComID is retired, the Control Session is terminated. The flow control for the Control Session is performed in the same manner as the flow control for Regular Sessions, with the difference that the communication is between the TSM and the HSM and these entities are responsible for the flow control.

There SHALL be only one Control Session per ComID.

3.3.7.1.3 Session Manager Protocol Layer

Begin Informative Content

The Session Manager Layer (see 3.3.4) is a special protocol layer session on any TPer with SPs. It is the communications channel used by host applications to start and manage sessions with SPs, to inform the TPer of the host's communications capabilities, and to inquire about TPer communication characteristics. The Session Manager protocol layer does not provide a session “to” any SP – it provides a communications control session. The method calls available on the Session Manager Layer are identified in section 5.2.

End Informative Content

Although method invocations on the Session Manager layer SHALL NOT change permanent state on the TPer, some method invocations MAY have side effects that occur outside of the normal method invocation process, such as logging or PIN retry counts. In cases where these changes occur – for example, logging a `StartSession` method call success or failure – the change SHALL occur on the SP to which the method call was attempted.

Method calls on the Session Manager Layer are formatted/encoded the same as on any other session. Due to the asynchronous nature of session startup and TPer communications, all of Session Manager layer methods’ responses are formatted as method calls, so that the host is able to identify responses to methods it has invoked.

The Session Manager Layer control session for a given ComID SHALL always be open. The TPer SHALL ignore End of Session and Transaction Control tokens sent to the control session, and SHALL not echo those tokens back to the host.

Methods invoked on the control session with an InvokingID that is not the SMUID SHALL be ignored/discarded by the TPer. Methods invoked on the control session with a MethodID that is not a

control session method, or a control session method that the TPer does not support, SHALL be ignored/discarded by the TPer.

3.3.7.1.4 Starting Sessions

Successful session startup depends upon three independent requirements:

1. The TPer and the requested SP having sufficient resources.
2. Successful negotiation of exchange keys if secure messaging with key exchange is required.
3. The required authentication is successful. (one of the following):
    a. Host authenticates to SP
    b. SP authenticates to Host
    c. Both of the above
    d. None of the above (No authentication)

Sessions are started with either a two or four method exchange on the Session Manager protocol layer:

`StartSession`
`SyncSession`
`StartTrustedSession` (optional)
`SyncTrustedSession` (required if `StartTrustedSession` is used)

Because of the asynchronous nature of session startup and other Session Manager layer traffic, the `StartSession`/`StartTrustedSession` responses (`SyncSession`/`SyncTrustedSession`, respectively) are formatted as method calls back to the host.

The authorities used during session startup determine the secure messaging and authentication requirements.

a. HostExchangeAuthority: The authority that references the Host's Exchange Key – used for exchange of session keys, provides implicit authentication
b. HostSigningAuthority: The authority that references the Host's Signing Key for challenge/response authentication, or the host's C_PIN credential for password authentication – used for authenticating the host; and, for challenge/response authentication, provides session startup method integrity.
c. SPExchangeAuthority: The authority that references the SP's Exchange Key – used for exchange of session keys, provides implicit authentication
d. SPSigningAuthority: The authority that references the SP's Signing Key – used for authenticating the SP to the host and session startup method integrity, provides explicit authentication

These authorities are already known to the SP.

Host authorities and SP authorities enable mutual authentication between the host and the TPer. Host authorities, if used, are passed in the `StartSession` method call. SP authorities are authorities that MAY be referenced in the Host authorities' `Authority` table rows. The ability to specify authorities in the `StartSession` method call, coupled with the linking of authorities in the `Authority` table, provides a large and diverse set of possible session protocols, including secure messaging. It is the initial selection of authorities by the host that determines which protocol is to be followed.

When the host makes the `StartSession` method call it knows which SPExchangeAuthority and SPSigningAuthority (if any) the SP uses. Those MAY be the root authorities in a certificate chain whose ultimate effective authority, as represented by the chained-down certificate, the host does not know. This is why the SP MAY return certificates to the host as part of `SyncSession`.

If a HostSigningAuthority or SPSigningAuthority requires a Challenge-Response, as is the case for all PuK, SymK, and HMAC authorities, or if secure messaging is to be used (or both), then the `StartSession` and `SyncSession` methods SHALL be followed by the `StartTrustedSession` and `SyncTrustedSession` methods.

An authority (HostExchangeAuthority, SPExchangeAuthority, HostSigningAuthority, or SPSigningAuthority) that is also a Public Key Authority (an Authority with public key credentials--PuK) MAY have additional information supplied for it in the form of a certificate or certificate chain. In this case the Effective Authority (the one responding to the challenge) SHALL be the tail PuK of that chain.

The effective authority is transient to the session. It is necessary to create a new authority on the SP (in a Read-Write session) if that authority is to persist on the SP past the end of that session.

All authorities that participate in the successful startup of a session SHALL be authenticated for that session.

3.3.7.1.5 Ending Sessions

The Host or TPer is free at any time to end a session in which it is participating, but only the host SHALL end the session successfully.

The session SHALL NOT be considered successfully closed until the party receiving the end of session request has responded indicating whether or not it was able to comply with the session ending request. Thus, a session is successfully ended when the TPer receives an End of Session token (see section 3.2.2.2) from the host and prepares a response with an End of Session token, and when transmission acknowledgement for ending the session has been performed as noted in Section 3.3.9.5 (if transmission acknowledgement is in use).

The host SHOULD NOT encode additional tokens after the End of Session token in a subpacket. Additional tokens encoded after an End of Session token SHALL be ignored by the TPer.

When a session closes, TPer resources that had been reserved for use with that session SHALL be released. The release of resources is not dependent on whether the session closed successfully or unsuccessfully – the end of the session releases the resources.

Sessions end unsuccessfully (abort) in a number of ways. These include (but are not limited to):

a. If the TPer detects any violation of flow control.

b. If the host does not (or is unable to) send any additional packets to the TPer, and sends no other communications, the TPer would time out while waiting for the communication from the host.

c. One of the communicators reached its implementation-specific limit on the number of times it re-sends a packet (due to negative acknowledgements or transmission timeouts while waiting for acknowledgement).

If a session is ended in the middle of the transmission of a method call or its parameters, then the method call SHALL be aborted in addition to the session being aborted. This is considered a fatal session error indicating a communication synchronization error (or worse).

An aborted session causes the following to occur:

1. All uncommitted transactions SHALL be aborted.
2. All method processing for that session SHALL be aborted;
3. The TPer MAY transmit a CloseSession method on the Session Manager layer.

When a session is aborted, open transactions within that session SHALL be aborted, and any method currently executing SHALL fail in its entirety.

The `CloseSession` method allows the TPer to notify the host that it has aborted a session. The TPer MAY send a `CloseSession` method on the Session Manager layer when it aborts a session. This is done by the TPer to notify the host that the TPer is ending the session.

Hardware resets and power cycles SHALL cause all open sessions to abort.

The host is able to abort a session by sending an End of Session token to the TPer.

3.3.7.1.6 Session Timeouts

The session timeout is used to limit the lifetime of a session. A session timeout is associated with every session and is specified in milliseconds.

The session timeout is a property of the session and is derived from three sources.

a. DefaultSessionTimeout : A value in the `Properties` method response.

b. SPSessionTimeout : A column in the `SPInfo` table.

c. SessionTimeout : An optional parameter in the `StartSession` method call used to open the session to the SP.

The TPer and the Host both maintain a timer associated with every active session. The timer starts when a session is successfully opened to an SP. Depending on the type of session started, this occurs when the tokens for the `SyncSession` or the `SyncTrustedSession` method call are built by the TPer and made available to the host.

The TPer MAY impose conditions on maximum and minimum timeouts supported by the device depending on hardware and other design considerations. These are indicated in the `Properties` method response values MaxSessionTimeout and MinSessionTimeout. These limits apply to all of the three timeouts listed above.

A column in the `SPInfo` table contains the SP default timeout. Modification of this value SHALL take effect on all future sessions opened on the SP.

If no value is specified for the SessionTimeout parameter of the `StartSession` method, then the SP's default value, stored in the `SPInfo` table's `SPSessionTimeout` column, SHALL be used. If no value exists as an SP default (i.e. the `SPSessionTimeout` column value is zero), then the TPer default (as reported in the `Properties` method response, DefSessionTimeout) SHALL be used.

A value of zero is permitted for the SessionTimeout parameter. The value is only permitted if the TPer's property response for MaxSessionTimeout is zero and the SP's SPSessionTimeout value is zero. Otherwise, the method SHALL fail with a `SyncSession` status of INVALID_PARAMETER.

The TPer MAY abort the session any time after the session's lifetime exceeds the session timeout value. The session is considered to have been closed / terminated when the last status token sent by the TPer is picked out of the output buffer by the host, or when the TPer releases all the resources (including the output buffer) for the session.

Session timeout SHALL NOT apply to the Session Manager Layer control session since it is always open. The time taken to complete the Session Manager Layer exchange to successfully start a session is not in the scope of this feature.

3.3.7.2 Methods

Begin Informative Content

Methods are remote procedure calls that operate on tables or SPs, and are called within a regular session to an SP, or within a control session in the case of Session Manager Layer methods. The caller passes a list of parameter values to the method and the method returns a list of result values followed by a status list, the first value of which is the status code response to the method invocation. Method calls, their parameters, and their results are all sent and received over session streams. Each session to an SP has at least two streams of bytes onto which data is encoded. One stream goes from the host to the SP, and the other comes from the SP to the host. Each stream operates asynchronously from all other streams, unless the Synchronous Interface Communications Protocol is in use (see section 3.3.10).

Typical host method calls send all their parameters/data to the SP before trying to read any of the results, but the SP is free to generate results incrementally as it consumes its parameters. The host is similarly free to try to read SP results while sending parameters. The TPer implementation determines how synchronous or asynchronous to be, so long as the semantics of the method call(s) are not compromised.

End Informative Content

A well-formed method call SHALL consist of the following steps:

1 The host tells the SP the method it wants to call.
2 The host sends a list of parameters to the SP.
3 The method is processed in the SP.
4 The method results are returned from the SP to the host.

Steps 2-4 MAY be repeated when input and output are incrementally streamed.

Within a given session at most one method SHALL be active at a time. If a method is unable to be processed completely, it SHALL fail and none of the direct changes made by the method take effect.

For information on method syntax, see 3.2.4.1. For information on method encoding, see 3.2.4.2.

3.3.7.3 Transactions

Begin Informative Content

Transactions are used to provide a clean model for how changes to an SP are to take effect. They also provide an easy way for host applications to handle error recovery.

End Informative Content

If a session is aborted, any open transactions SHALL be aborted.

Changes are successfully committed and made persistent (to the media, made visible to subsequent sessions on the same SP, etc.) in 2 ways:

a. When a method is invoked outside of a transaction, and resolves successfully, changes made by that method SHALL be committed and made persistent immediately.

b. When a method is invoked inside of a transaction or set of nested transactions, changes made by that method SHALL be committed and made persistent when the top-level transaction is committed.

Changes made within a transaction SHALL be visible within that transaction. For instance, modification of a table value within a transaction would result in the new value being returned by a `Get` method invoked within that transaction. Those changes SHALL be made persistent when the top-level transaction is committed. If the transaction is aborted, those changes SHALL be rolled back.

Changes that affect other aspects of the TPer (i.e. hardware settings) SHALL occur when associated changes are successfully committed. This means that changes made during transactions that affect the state of the device, such as changing media encryption keys or read/write lock state, SHALL NOT occur until the changes are successfully committed.

Some changes MAY occur as exceptions to transactional rollback (i.e. logging), and SHALL commit immediately even if they occur inside of a transaction or as a side effect of a method invocation that has failed.

An aborted transaction SHALL only occur for one of two reasons:

a. At the request of the host, by host-transmission of the End Transaction token with a status other than `0x00`.

b. If an error occurs while committing the transaction (i.e. the host sends End Transaction with a status of `0x00`, but the TPer encounters some kind of error while committing the transaction to media).

Specific transaction-related control tokens in the session stream, defined in the 3.2.2, serve to indicate transaction start and end points. If a transaction control token is received at a point in the session stream that occurs within a method invocation, the TPer SHALL abort the session.

All transactions consist of the following steps:

1. The transaction is opened.
2. Zero or more method calls are made.
3. The transaction is either aborted or committed.

If a transaction is aborted all SP state SHALL be reset ("rolled-back") to its value at the time the transaction was opened (e.g., authentication state), unless otherwise noted in this specification (e.g., logging). A transaction SHALL only be committed at the request of the host application. The TPer SHALL only commit or abort a transaction upon receipt of an End Transaction token from the host except in the case when a session is aborted.

The failure or success of the methods encapsulated in the transaction SHALL NOT directly affect whether or not the host is able to commit the transaction, but committing a transaction in which method invocations have not succeeded MAY leave the SP in an intermediate (and potentially unrecoverable) state. A failed method within a transaction SHALL NOT affect the state of the transaction or the state of the SP within the transaction, unless otherwise noted (e.g., logging, PIN tries count, etc.).

The TPer SHALL guarantee that a transaction completely commits to media (persists) or completely aborts. This means that the TPer SHALL arrange that if a power cycle, reset, or other event occurs in the middle of a commit, when the TPer recovers the commit is either finished or all the changes are aborted. This guarantees SP consistency and prevents power-off or reset attacks.

3.3.7.3.1 Nested Transactions

A session MAY include nested transactions. The maximum number of transactions that MAY be nested is Security Subsystem Class-specific, and SHALL be specified in response to the `Properties` method invocation in the MaxTransactionLimit property if transactions are supported. If the TPer does not support transactions, the Properties method response SHALL NOT contain the MaxTransactionLimit property.

Nested transactions SHALL abort or commit relative to their parent transaction. In the case of an aborted transaction, the SP state SHALL be rolled back to the point where the transaction was started, unless otherwise noted in this specification (e.g. logging). This is true whether or not the transaction is nested. In the case of a commit, the nested transaction's changes SHALL become part of its parent transaction, as if the nested transaction boundaries had never been established.

A commit of a nested transaction does not make a commit that necessarily persists since the parent transaction is not yet ended. All transactions SHALL be committed before data is written to the SP.

3.3.8 Stream Flow Control

3.3.8.1 Introduction

Begin Informative Content

Flow control ensures that when data is sent from a source to a destination that the destination has enough buffer space to receive it. There are two kinds of flow control: Interface and Stream data.

Interface flow control is involved in moving IF-SEND or IF-RECV commands across an interface between a host and TPer (see [2]).

Stream data flow control is used to keep a Host or TPer from overwhelming the other party with data during a session.

End Informative Content

3.3.8.2 Buffer Management

Begin Informative Content

Flow control is used to keep a Host or TPer from overwhelming the other party with data during a session. The exchange of credits permits data to be moved from one communicator to the other.

Before session data is able to be sent, the receiver needs to notify the sender that it is ready to receive data and how much data it is able to receive. This is done by sending a Credit Control Subpacket in the direction opposite that of the data.

End Informative Content

As data in the receive buffers of the communicators is consumed and space released, additional Credit Control Subpackets MAY be sent.

The InitialCredit parameters of the `StartSession` and `SyncSession` methods provide each communicator in a session the opportunity to provide an initial amount of credits for use when the session successfully starts. If either of these values is omitted, then once a session has been successfully started, the communicator that omitted the value from the InitialCredit parameter of its session startup method SHALL send to the other communicator a credit subpacket announcing its available session buffer space.

Credit values are byte counts for the payload of data subpackets and do not include packet or subpacket headers/overheads. Packets containing only ACK/NAK information, or only Credit Control Subpackets, MAY be sent at any time regardless of how much credit the sender has.

The sender SHALL NOT send more data than it has credits from the receiver. As the sender transmits data, the amount of transmitted data is subtracted from the total credits that had been provided to the sender. This identifies the amount of data that MAY still be sent without receiving additional buffer credits.

As the receiver consumes data, the receiver MAY notify the sender that it is able to receive additional data. This is done by transmitting a Credit Control Subpacket identifying how much additional buffer space the sender is able to utilize. The number of bytes of data that are able to be sent to that session SHALL be increased by the value of each credit received. When a communicator transmits data, the amount of data sent SHALL be subtracted from the credit total.

If buffer management is supported, credit subpackets SHALL be exchanged after ComID acquisition, so that the host and TPer are able to exchange methods/responses on the control session. This credit only applies to the control session for that ComID. Credit subpackets SHALL also be exchanged immediately after session startup within the new session, unless values are posted in the InitialCredit parameters, in which case additional credit subpackets are optional at that time.

Otherwise, Credit Control Subpackets SHOULD be sent infrequently and be bundled with other traffic, in order to minimize interface overhead. Either communicator in a session MAY send Credit Control Subpackets as frequently as in every packet, or when a threshold is reached (e.g. the unreported credit is more than some percentage of the buffer size).

Violating flow control is one reason either side MAY abort a session.

3.3.9 Session Reliability

3.3.9.1 Introduction

Begin Informative Content

Session Reliability provides resilience against lost, duplicated, or deleted packets.

End Informative Content

3.3.9.2 Transmission Acknowledgement

If the TPer supports transmission acknowledgement and the host has informed the TPer (via invocation of the `Properties` method) that it also supports transmission acknowledgement, each packet sent from

the TPer to the Host (or vice-versa) for a given session SHALL have a sequence number (SeqNumber) that corresponds to the number of packets that have been sent by that communicator since the start of the session. The first packet in a session SHALL have a SeqNumber value of 1.

If transmission acknowledgement is supported, each packet with SeqNumber N SHALL be acknowledged by the receiver. Once the sender receives an acknowledgement for data contained in packets up to packet N, the sender is able to safely discard the data for packets with SeqNumber N and lower.

Packets that contain only ACK or NAK information SHALL NOT require an ACK/NAK response from the receiver. These packets SHALL still have an appropriate SeqNumber field value. Packets for sessions that are not protected by secure messaging that do not require ACK/NAK SHALL be those packets with a Length field value of zero and a corresponding empty Data field value. Packets that are protected by secure messaging that do not require ACK/NAK SHALL be those packets with a DataLength field value of zero and an empty Data field (The IV and MAC fields MAY still contain values).

When a communicator sends a packet that contains only ACK/NAK information, it SHALL still keep that packet (for possible re-transmission) until either it or a later packet is ACKed. This is because the receiver MAY NAK that packet in the case of loss/corruption, and in this case the packet SHALL be retransmitted.

After receiving a packet, the receiver SHOULD send an ACK within the TransTimeout period so that the sender does not re-send un-ACK'ed packets.

3.3.9.3 Transmission Negative Acknowledgement

If the receiver detects data gap in the SeqNumbers of received packets, the receiver SHALL send a negative-acknowledgement packet (NAK) with the SeqNumber of the packet at which the receiver wishes the sender to begin retransmission. The receiver puts a value of the SeqNumber of the last known good packet (N) received plus one. This automatically acknowledges all previous packets with SeqNumbers less than or equal to N. The receiver SHALL NOT NAK a SeqNumber less than or equal to the last ACKed SeqNumber. If the TPer receives a NAK for a SeqNumber less than or equal to the last ACKed SeqNumber, it SHALL abort the session. Negative acknowledgement serves to notify the sender that a retransmission of packet N+1, etc. is needed.

Upon dispatch of the NAK, the sender of the NAK SHALL discard all packets with SeqNumbers N+ 2 and higher, since the sender is expected to retransmit these. The NAK SHALL NOT be re-transmitted due to receiving packets with SeqNumber containing a value other than N+1, because packets with values greater than N+1 could have already been 'in flight' when that NAK was sent. Retransmission of the NAK is dependent on the transmission timeout value for the session, not on subsequent receipt of additional data.

Retransmitted packets SHALL be sent with no modifications or additions, including packet headers.

3.3.9.4 Transmission Timeouts

The transmission timeout is set during the exchange of session startup methods `StartSession` and `SyncSession`. The transmission timeout for a session SHALL take effect after session startup has successfully completed. Both communicators share the same transmission timeout value.

The transmission timeout in effect for control sessions is the transmission timeout reported in the `Properties` method.

The sender MAY provide, in the `StartSession` method, a value for the TransTimeout parameter. The communicator that transmits the `SyncSession` method MAY include a value for the TransTimeout parameter. If so, that communicator's timeout value SHALL be larger than the `StartSession` TransTimeout value, and SHALL be the transmission timeout value in use for the session being started. In either case, the TransTimeout value SHALL be greater than or equal to the MinTransTimeout value and smaller than or equal to the MaxTransTimeout parameter reported in the `Properties` method response. If neither communicator includes a value for the TransTimeout parameter, the DefTransTimeout value, as reported in the `Properties` method response, SHALL be used.

If the sender detects a missing acknowledgment by means of a timeout, the sender SHALL retransmit the data from the last valid acknowledgment. If the sender still receives no acknowledgement after a timeout period, the sender SHALL retransmit the same packet with no modifications or additions, including packet headers. This retransmission repeats up to an implementation-specific number of times. Thereafter, the sender SHALL terminate the session, i.e. no more data is able to be be transmitted for this session (the session times out at some point and is closed by the receiver).

SSCs that require support for transmission timeouts SHALL define a minimum required value for retransmission repeats.

3.3.9.5 Closing a Session

If transmission acknowledgement is supported, when the host transmits a data subpacket that contains the End of Session token, the host SHOULD NOT immediately assume the session has successfully closed.

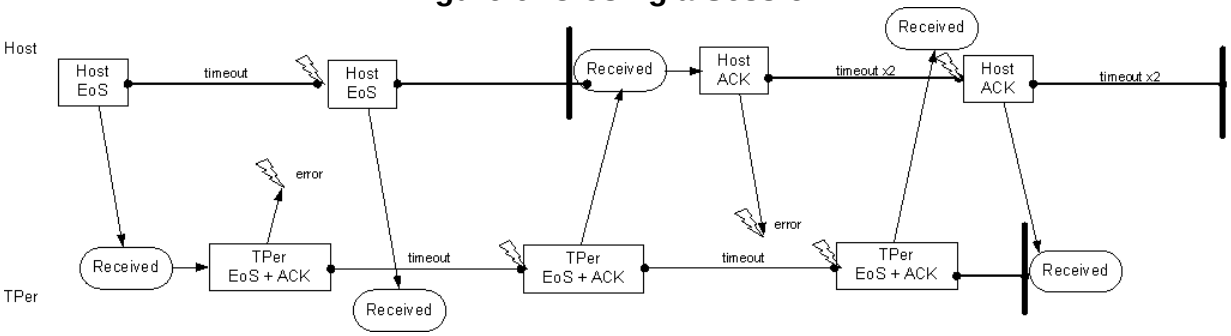
The host SHOULD wait for the TPer to both transmit the TPer's own data subpacket that contains the End of Session token and to ACK the host's packet that contained the End of Session token. The host SHOULD then ACK the TPer's data subpacket that contains the End of Session token.

The host SHOULD follow the normal rules of the ACK mechanism. If the transmission timeout period expires and the host has not received the TPer's ACK of its packet containing the End of Session token, the host SHOULD retransmit that packet and continue doing so until it receives an ACK or it reaches its timeout retransmission limit.

If the TPer has not received, by the end of the transmission timeout period, the host's ACK of the TPer's packet containing the End of Session token, the TPer SHALL retransmit that packet, and continue retransmitting it until it receives an ACK from the host or until it reaches its implementation-defined timeout retransmission limit.

Once the TPer has received the packet containing the host's ACK of the packet the TPer transmitted containing the data subpacket with the End of Session token, the TPer SHALL consider the session to be closed.

Figure 6 Closing a Session



3.3.10 Synchronous Interface Communications

3.3.10.1 Introduction

Begin Informative Text

The communications protocol stack as described in this specification enables a fully asynchronous exchange of data between host and TPer. Using the communications stack in this manner is a matter of arbitrarily interleaving IF-SEND commands with IF-RECV commands.

Asynchronous communications allows the host to transmit methods and data to the TPer without having to retrieve the results of those methods before sending additional methods, and enables the TPer to return method results upon request at arbitrary boundaries. Flow control provides a mechanism for buffer management to occur as data is successfully transmitted and received.

However, for some hosts and devices, these mechanisms are too complex and require more processing capability and code space than is available. For these situations, the synchronous communications protocol stack is tailored to better meet the capabilities of host and TPer.

For instance, fixed buffer sizes coupled with restrictions on the relationship between the exchange of IF-SEND and IF-RECV commands negates the need for flow control for buffer management.

End Informative Text

3.3.10.2 Interface Commands

This section defines the requirements imposed on the exchange of IF-SEND and IF-RECV commands.

3.3.10.2.1 Restrictions

The restrictions imposed on the exchange of IF-SEND and IF-RECV commands are as follows:

1. Any number of non IF-SEND/IF-RECV commands MAY be interleaved with IF-SEND/IF-RECV commands.
2. The normal communications state of an Associated ComID SHALL be to await receipt of an IF-SEND command for that ComID.
    a. While awaiting receipt of an IF-SEND interface command, any received IF-SEND command for that ComID SHALL be accepted (provided it meets the other requirements such as not exceeding the maximum supported length).
    b. Once the entire command payload has been received, the TPer SHALL return an interface status to the host.
    c. Any IF-RECV command received for the Associated ComID awaiting receipt of an IF-SEND command SHALL return to the host a ComPacket with a Length field value of zero, an OutstandingData field value of zero, and a MinTransfer field value of zero. This signals to the host that there is no pending response data to retrieve.
3. After an IF-SEND command has been received, a command completion without error has been returned, and the payload has been decoded without an error, the TPer SHALL NOT accept another IF-SEND command for that ComID until the host has retrieved the entire response via IF-RECV(s).
    a. Any subsequently received IF-SEND commands for the specified ComID SHALL be aborted at the interface level. The interface status for this action SHALL be "Synchronous Protocol Violation" (see [2]).
    b. If the TPer has not sufficiently processed the command payload and prepared a response, any IF-RECV command for that ComID SHALL receive a ComPacket with a Length field value of zero (no payload), an OutstandingData field value of `0x01`, and a MinTransfer field value of zero.
    c. If the TPer has sufficiently processed the command payload and prepared a response, an IF-RECV command that requests a transfer length less than the amount of response data the TPer has prepared SHALL reply with a ComPacket with a Length field value of zero (no payload) and OutstandingData value of total bytes currently available, and MinTransfer field value of the minimum request required to transfer a packet.
    d. The SSC MAY additionally require that each method response be retrieved separately (along with Control Tokens as determined by the TPer), via multiple IF-RECV commands. For these SSCs, if all responses have not been retrieved:

i. If additional responses are available, and the host has requested a transfer length less than the minimum transfer required, the TPer SHALL respond to an IF-RECV command with OutstandingData value of total bytes currently available; and MinTransfer field value of the minimum request required to transfer a packet.

ii. If additional responses are available and the host has requested a sufficient transfer length, the TPer SHALL respond to an IF-RECV with OutstandingData = `0x00`, `0x01`, or the amount of total bytes currently available; and MinTransfer field value of zero or the minimum request required to transfer a packet.

iii. If no additional responses are prepared but more are to come, the TPer SHALL respond to an IF-RECV command with OutstandingData field value of `0x01` and MinTransfer field value of Zero. Table 45 summarizes the values of the Length, OutstandingData, and MinTransfer fields of the packets returned to the host by the TPer in response to an IF-RECV command.

A summary of the values for ComPacket fields is displayed in Table 45.

Table 45 IF-RECV ComPacket Field Values Summary

IF-RECV	Length Field Value	OutstandingData Field Value	MinTransfer Field Value
Response(s) to come, no Response(s) available	0x00	0x01	0x00
Response ready, insufficient transfer length request	0x00	Total bytes currently available	The minimum request required to transfer a packet
Response, additional Response(s) available	Data Length	Additional bytes available, not including the data transferred in the current ComPacket.	The minimum request required to transfer the next packet
Response, additional Response(s) to come, no Response(s) available	Data Length	0x01	0x00
Response, all Response(s) returned – no further data	Data Length	0x00	0x00
All Response(s) returned – no further data	0x00	0x00	0x00

3.3.10.3 Synchronous Communications Restrictions

This section defines additional restrictions specific to the Synchronous Communications protocol.

a. Methods SHALL NOT span ComPackets. In the case where an incomplete method is submitted, if the TPer is able to identify the associated session, then that session SHALL

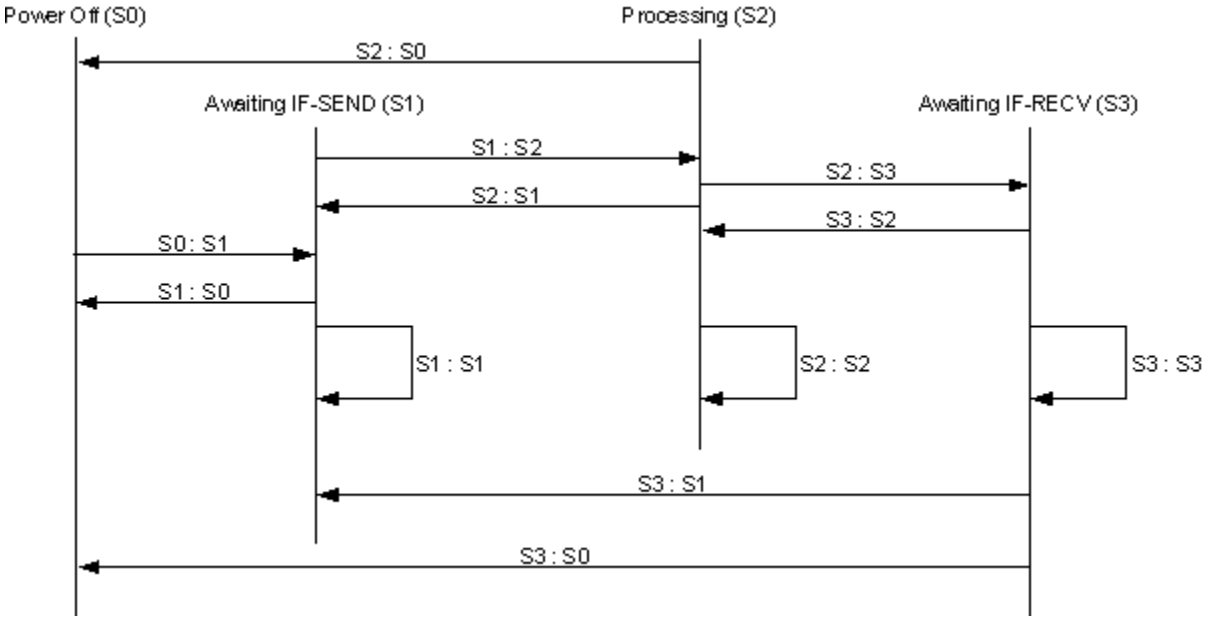
be aborted and a `CloseSession` MAY be prepared for delivery on that ComID's control session.

b. The synchronous exchange of interface commands SHALL only apply to IF-SEND/IF-RECV commands exchanged on Protocol ID `0x01`, but SHALL NOT apply to Level 0 Discovery.

3.3.10.4 State Transition Diagram

The state transitions for the exchange of IF-SEND and IF-RECV commands are delineated in Figure Figure 7 The states used are defined in 3.3.10.5, and the state transitions are defined in 3.3.10.6.

Figure 7 Synchronous Communications State Transition Diagram



3.3.10.5 State Descriptions

This section defines the states used in Figure 7

State "Power-Off (S0)" – In this state, power is removed from the TPer and it is completely unresponsive.

State "Awaiting IF-SEND (S1)" – In this state, the TPer command interface is ready and there are no outstanding IF-SEND/IF-RECV commands for the specified ComID. A command is "outstanding" if it has entered the "Processing" or "Awaiting IF-RECV" state. A command is not considered "outstanding" if it is in the TPer command queue awaiting initial processing by the device.

a. While in this state, if IF-SEND is received or dequeued with the ComID for this state machine, the TPer MAY request command payload transfer and SHALL return interface status to the host.

b. While in this state, if IF-RECV is received or dequeued with the ComID for this state machine, the TPer SHALL return a response ComPacket the specified ExtendedComID with the Length, OutstandingData, and MinTransfer fields set per "All Response(s) returned – no further data" defined in Table 45.

State "Processing (S2)" – In this state, the TPer has begun processing the payload of an IF-SEND command.

a. While in this state, the TPer SHALL terminate any received or dequeued IF-SEND commands. The interface status for this action SHALL be "Synchronous Protocol Violation" (see [2]).

b. While in this state, the TPer SHALL return a response ComPacket for any received or dequeued IF-RECV commands for the specified ExtendedComID with the Length, OutstandingData, and MinTransfer fields set per “Response(s) to come, no Response(s) available” defined in Table 45.

State “Awaiting IF-RECV (S3)” – The TPer has completely processed the TCG data payload and has the associated TCG response ready for retrieval by the host.

a. While in this state, if IF-RECV is received or dequeued with the ComID for this state machine and a transfer length less than the amount of response data staged for the ComID, the TPer SHALL return a response ComPacket for the specified ExtendedComID with the Length, OutstandingData, and MinTransfer fields set per “Response ready, insufficient transfer length request” defined in Table 45.

b. While in this state, the TPer SHALL terminate any received or dequeued IF-SEND command. The interface status for this action SHALL be "Synchronous Protocol Violation" (see [2]).

3.3.10.6 State Transitions

This section defines the state transitions for each valid ComID as presented in Figure 7

S0:S1 – This transition occurs automatically when the TPer is powered on.

S1:S0 – This transition occurs when the TPer is powered off.

S1:S1 – This transition occurs when:

a. The TPer receives an interface initiated TCG reset (see [2]), or

b. The TPer receives a Protocol Stack Reset Command for the ComID of this state machine (see 3.3.4.7.5), or

c. The TPer detects an error in a received IF-SEND payload that prevents the TPer from resolving an intended session for the IF-SEND command payload, or other error that prevents the TPer from processing the command (see 3.3.10.7), or

d. The TPer receives an IF-RECV command for this ComID (see the "Awaiting IF-SEND" state description in 3.3.10.5).

S1:S2 – This transition occurs when an IF-SEND command with the ComID associated with this state machine is received or dequeued and successfully completes data transfer of the command payload.

S2:S0 – This transition occurs when the TPer is powered off.

S2:S1 – This transition occurs when:

a. The TPer receives an interface initiated TCG reset (see [2]), or

b. The TPer receives a Protocol Stack Reset Command for the ComID of this state machine (see 3.3.4.7.5), or

c. The TPer detects an error in the IF-SEND payload that prevents the TPer from resolving an intended session for the IF-SEND command payload (see 3.3.10.7), or

d. If there is no action to be taken as a result of the received command, such as when the IF-SEND command payload is a ComPacket with

    a. no payload, or

    b. one or more Packets all of which have no payload, or

c. The IF-SEND command payload is a ComPacket with one or more Packets all of which have SubPackets that either have no payload or have a payload that consists entirely of Empty Atoms.

S2:S2 – This transition occurs when:

a. The TPer receives an IF-SEND for this ComID (see the "Processing" state description in 3.3.10.5), or

b. The TPer receives an IF-RECV for this ComID and processing has not completed to the point where data is available for retrieval by the host (see the "Processing" state description in 3.3.10.5).

S2:S3 – This transition occurs when the TPer has completely processed the contents of the IF-SEND command and has a complete response available for retrieval by the host. A separate response MAY be generated for each method in the IF-SEND.

S3:S0 – This transition occurs when the TPer is powered off.

S3:S1 – This transition occurs when the TPer receives:

a. An interface initiated TCG reset (see [2]), or

b. A Protocol Stack Reset Command for the ComID of this state machine (see 3.3.4.7.5), or

c. An IF-RECV able to retrieve the entire response resulting from the IF-SEND, or

d. An IF-RECV for the last of multiple responses resulting from the IF-SEND.

S3:S2 – This transition occurs when the TPer receives an IF-RECV able to retrieve a response resulting from an IF-SEND but still has additional responses to process from that IF-SEND.

S3:S3 – This transition occurs when:

a. The TPer receives an IF-SEND for this ComID (see the "Awaiting IF-RECV" state description in 3.3.10.5), or

b. An IF-RECV is received or dequeued with the ComID for this state machine and a transfer length less than the amount of response data staged for the ComID (see the "Awaiting IF-RECV" state description in 3.3.10.5), or

c. The TPer receives an IF-RECV able to retrieve a response resulting from an IF-SEND, and additional responses are still available. The TPer SHALL a response ComPacket for the specified ExtendedComID with the Length, OutstandingData, and MinTransfer fields set per “Response, additional Response(s) available” defined in Table 45.

3.3.10.7 Error Handling

This section defines the manner in which violations of the restrictions on Interface Command payloads SHALL be handled by the TPer.

a. If a violation of packet structure occurs such that the TPer is unable to resolve a valid Session ID in an IF-SEND command, or if the restriction violation occurs due to violations of packet requirements, the TPer SHALL ignore the entire packet and SHALL immediately transition to the state of awaiting an IF-SEND command.

b. If a violation of packet structure occurs such that the TPer is able to resolve the Session ID, the TPer SHALL close that session and MAY prepare for transmission the `CloseSession` method for retrieval by the host.

c. The device SHALL abort at the interface level any IF-SEND command whose transfer length (in bytes) is greater than the reported MaxComPacketSize for the corresponding ComID. The interface status for this action SHALL be "Invalid Transfer Length parameter on IF-SEND" (see [2]).

d. For SSCs that require that entire method responses be retrieved, if data generated in response to any single method in an IF-SEND command (together with required communications overhead) does not fit entirely within the TPer's response buffer, the device SHALL NOT return any part of that method response and SHALL instead return an empty results list with a status code of RESPONSE_OVERFLOW in the status list. Additionally, the TPer SHALL continue processing methods and control tokens that had been sent in that command payload (if any).

3.4 SP Operation Descriptions

This section defines the operational and access control model for SPs.

3.4.1 General SP Guidelines

3.4.1.1 Admin SP

Begin Informative Content

The Admin SP maintains information about other SPs and the TPer as a whole and enables creation of other SPs under issuance control.

End Informative Content

There SHALL be exactly one Admin SP on every TPer that has SPs or that is able to have SPs issued. If present, the Admin SP SHALL NOT be able to be deleted, disabled, or frozen. The Admin SP SHALL have the name "`Admin`."

3.4.1.2 SPs

SPs are created by integrating portions of one or more of the templates supported by a TPer (as identified in the Admin SP).

A template includes the following:

a. Each template SHALL have a different name.

b. Templates define a set of table and method definitions. These definitions are used to define the initial tables and methods that MAY be included in an instance of that template.

c. A maximum instance count. A maximum instance count of zero means no limit. At any time there SHALL be no more than this number of SPs based on this template instantiated within the TPer.

An SP includes the following:

a. A name – Each SP SHALL have a different name.

b. A set of tables – Tables SHOULD be stored in a non-user addressable storage area on the TPer.

c. A set of methods – The supported methods define the operations that MAY be performed on the SP and the SP's tables.

All SPs SHALL be created from at least the Base Template. The Base Template is combined with zero or more other template(s) to create an SP, though the number of SPs that instantiate a particular template MAY be limited by the template's maximum instance count. The number of Base Template instantiations permitted in a particular TPer by definition limits the number of SPs that MAY be issued for that TPer.

An SP MAY incorporate only a subset of the entire set of tables and methods provided by each template that makes up the SP.

3.4.2 Access Control

Begin Informative Content

This section introduces the concepts utilized to permit and restrict operations within an SP.

End Informative Content

3.4.2.1 Overview

Begin Informative Text

Access control limits the methods that are able to be processed on an SP, a table, or on specific rows and columns of a table.

Permission to process a method is governed by which secrets the method's invoker has proven that it knows. The secrets and their public parts are called Credentials. The operation for proving knowledge of a secret is called an Authentication Operation. The actual proving of knowledge of a secret is called Authentication.

End Informative Text

Authentication in this document describes either Explicit Authentication, which typically occurs as a result of password validation or challenge/response; or Implicit Authentication, which occurs as a result of implicitly proving knowledge of a secret, such as during session key exchange.

An authority SHALL be considered authenticated in either type of authentication scenario - the terms explicit and implicit are descriptive and SHALL NOT limit the authentication or capabilities of an authority.

Begin Informative Text

An authority is used by the host application to represent a person, a role, a program agent, etc. These are distinctions of meaning to the application, not to the SP.

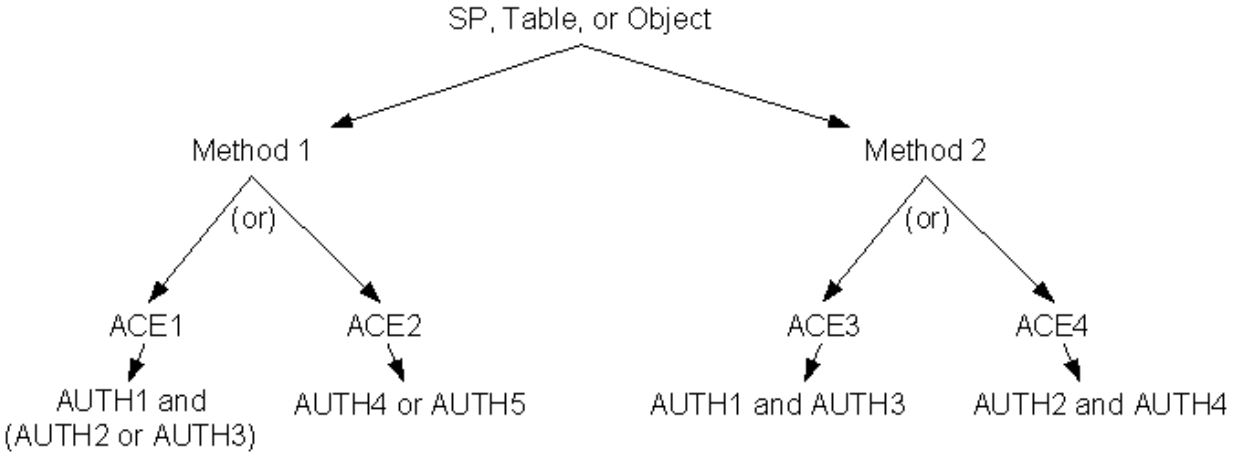
Access Control is specified in layers. The top layer of the mechanism is Access Control Lists (ACLs). ACLs are lists of Access Control Elements (ACEs). This layering gives the host a way to delegate control of an ACL, via control of its ACEs, to various independent entities.

ACEs are Boolean combinations of authorities. This permits the ACE to express cross-certification or other forms of restriction.

End Informative Text

When an authority is authenticated, its value in an ACE Boolean expression SHALL be True. If the authority has not been authenticated, its value in an ACE Boolean expression SHALL be False.

Figure 8 Access Control



3.4.2.2 Authorities

An authority is an object in the `Authority` table. An authority is one of two kinds: Individual or Class. Each individual authority MAY be a member of one class authority. A class authority MAY be a

member of one class authority. A class authority SHALL NOT refer directly to a credential. An individual authority specifies one credential and one operation that uses that credential.

Begin Informative Content

Class authorities are a convenient way to allow an ACE to be set on a method without enumerating all the individual authorities that authorize that method. This means that the individual authorities that belong to that class authority are able to be changed without having to change any of the ACEs that refer to the class authority.

End Informative Content

A class authority SHALL be authenticated when any member of the class is authenticated. Class authorities SHALL NOT be directly authenticated.

- A credential is an object in a Credential table. All credential tables have a name that starts with "C_". A credential table SHALL have at least one column that stores a secret. It MAY also have “public” parts, which contain information such as public keys and certificates. A particular credential MAY have only some of its columns filled in. For example, if only a public key and certificates validating that public key are known, then the private key columns are unused (zeroes in these columns indicate that this information is not present).

Authentication to an authority SHALL occur within a session or during session startup, and SHALL apply only to that session. All authorities that participate in successful session startup are authenticated for that session. During a session the host MAY make any number of `Authenticate` method invocations. There MAY be Security Subsystem Class-defined TPer and per session limits on the maximum number of authorities that MAY be authenticated at any one time.

A set of authorities is defined by this specification. The AdminExch authority, of the class Admins, is one such pre-defined authority. Every SP has an AdminExch authority at time of issuance. SSCs MAY specify authorities in addition to these, or MAY restrict the use of the authorities specified in this document.

For details regarding the Admins authority and other pre-defined authorities, refer to 5.3.4.1.2.

3.4.2.3 ACEs and ACLs

Begin Informative Content

ACEs apply to methods on an SP, on a particular table in an SP, or on arbitrary parts of a particular table in an SP, down to the granularity of a single table cell.

With ACEs as the building blocks of ACLs, each ACE is able to have separate managerial control. For example, a host authenticated with one authority creates a table and gives another authority control of some of the ACEs on that table. This allows flexible, fine-grained management of access.

End informative Content

The minimum and maximum number of ACEs in an ACL and the minimum and maximum number of authorities in an ACE are SSC-specific. Every SSC SHALL at least stipulate the minimum.

3.4.3 SP Issuance, Personalization, and Operational State

Begin Informative Content

Issuance is the cryptographically controlled creation of SPs from templates. Issuance occurs within a session to a TPer's Admin SP, and is achieved by demonstrating knowledge of the secrets required to authorize the creation of new SPs and then, for each new SP, creating a unique credential for the Admin authority on that SP.

End Informative Content

Issuance of a new SP SHALL be complete when the top-level transaction that contains the method invocation is successfully committed or, if the method was invoked outside of a transaction, once the TPer has processed the method and prepared a response. SPs SHALL be created using the templates

specified during issuance. Once an SP is issued, it is not possible to add functionality to the SP from additional templates.

Begin Informative Content

Personalization follows Issuance. The AdminExch authority on the new SP accomplishes personalization by opening a session to the issued SP, creating new tables (in addition to the tables that were provided by the templates), provisioning those tables, creating and configuring new authorities, and setting the access controls on the SP's methods. Personalization is an ongoing process that occurs during the entire life of an SP.

End Informative Content

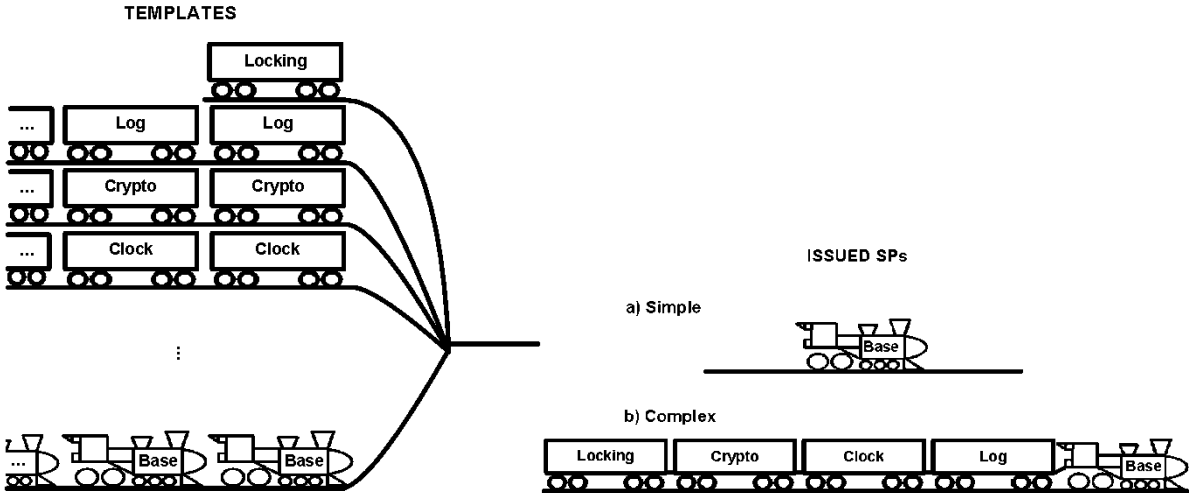
3.4.3.1 Issuing an SP

Begin Informative Content

Issuing an SP is similar to building a train (see Figure 9 below). Every train (SP) must have an engine (Base Template). Additional cars (other Templates) providing additional capabilities are able to be added at the time of issuance. In the simplest case, an SP is issued from just the Base Template (see part 'a'). In more complex cases several templates are used.

End Informative Content

Figure 9 Issuance



4 Life Cycle of SPs

4.1 Life Cycle of SPs Overview

Begin Informative Content

Each SP in a TPer has its own life cycle state. This section defines the various life cycle states and the transitions that an SP makes between them.

Life cycle applies to each individual SP. The life cycle state of the TPer as a whole emerges from life cycle states of individual SPs.

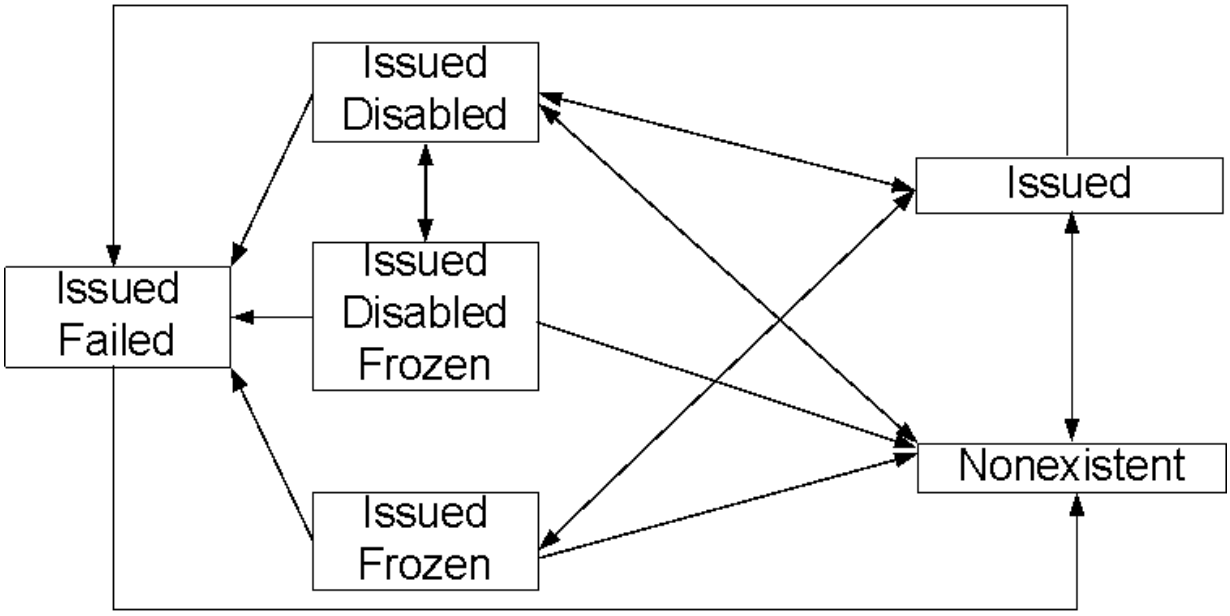
End Informative Content

Life cycle states are recorded in the `LifeCycleState` column of the Admin SP's `SP` table. This column identifies the SP's current state. The value of this column SHALL be changed by the TPer whenever an SP's life cycle state changes. The value of the Admin SP object's `LifeCycleState` column SHALL only be Issued.

Access control on reading the SPs available in a TPer, and the life cycle states of those SPs, SHALL be readable by the Anybody authority on the Admin SP.

4.2 Life Cycle States

Figure 10 Life Cycle State Transitions



The following list details the states depicted in Figure 10 .

a. **Nonexistent**: The Nonexistent state is a theoretical state that describes the condition of an SP before it has been instantiated, or after it has been deleted.

b. **Issued**: The Issued state is the standard operational state of an SP, and defines the initial required access control settings of an SP based on the templates incorporated into the SP and as defined by this specification and/or a supported SSC, prior to personalization. .

c. **Issued-Disabled**: This state occurs after an SP has been issued, when the value of the `Enabled` column of the SP's `SPInfo` table is False.

d. **Issued-Frozen**: This state occurs after an SP has been issued, when the value of the `Frozen` column of the Admin SP's `SP` table is True.

e. **Issued-Disabled-Frozen**: This state occurs after an SP has been issued, when both the value of the `Frozen` column of the Admin SP's `SP` table is True and the value of the `Enabled` column of the SP's `SPInfo` table is False.

f. **Failed**: The Failed state describes the condition where the SP has experienced an unrecoverable write failure; physical read error for the hidden (SP) space; or other unrecoverable failure that prevents access to TCG related functionality and data structures (i.e. the SP is unable to accept method invocations).

4.3 Life Cycle State Transitions

This section identifies and describes the possible transitions between life cycle states as depicted in Figure 10 .

a. Nonexistent/Issued

i. An SP transitions from Nonexistent to Issued when successful invocation of the `IssueSP` method causes the SP to be created. The SP SHALL be created in this state if the SP is operational and if the value of the `IssueSP` method's parameter Enabled was True.

ii. An SP transitions from Issued to Nonexistent when that SP is deleted.

b. Any State/Issued-Failed

i. An SP MAY transition into the Issued-Failed state if an unrecoverable write error or other failure occurs. The TPer SHALL control entry to this state.

ii. The Failed state is a terminal state. The only exit available from the Failed state is to the theoretical Nonexistent state, by invoking `Delete` on the SP's object in the Admin SP's `SP` table.

c. Issued/Issued-Disabled

i. An SP is transitioned from the Issued state into the Issued-Disabled state by setting the value of the `Enabled` column of the SP's `SPInfo` table to False.

ii. An SP is transitioned from the Issued-Disabled state to the Issued state by setting the value of the `Enabled` column of the SP's `SPInfo` table to True.

d. Issued/Issued-Frozen

i. An SP transitions from the Issued state into the Issued-Frozen state by setting the value of the `Frozen` column of the SP's object in the Admin SP's `SP` table to True.

ii. An SP transitions from the Issued-Frozen state into the Issued state by setting the value of the `Frozen` column of the SP's object in the Admin SP's `SP` table to False.

e. Issued-Disabled/Issued-Disabled-Frozen

i. An SP is transitioned from the Issued-Disabled state to the Issued-Disabled-Frozen state by setting the value of the `Frozen` column of the SP's object in the Admin SP's `SP` table to True.

ii. An SP is transitioned from the Issued-Disabled-Frozen state to the Issued-Disabled state by setting the value of the `Frozen` column of the SP's object in the Admin SP's `SP` table to False.

f. Issued-Disabled/Nonexistent

i. The SP MAY be deleted, and thus enter the Nonexistent state, by successful invocation of the `DeleteSP` method from within a session to the SP, or by successful invocation of the `Delete` method on the SP's object in the Admin SP's `SP` table.

g. Issued-Frozen/Nonexistent

i. The SP MAY be deleted, and thus enter the Nonexistent state, by successful invocation of the `Delete` method on the SP's object in the Admin SP's `SP` table.

h. Issued-Disabled-Frozen/Nonexistent

i. The SP MAY be deleted, and thus enter the Nonexistent state, by successful invocation of the `Delete` method on the SP's object in the Admin SP's `SP` table.

4.4 Default Authorities

The initial authorities defined in this specification that MAY affect the life cycle states are defined for:

a. Base Template (Table 209) – the Admins Authority (SP owner) and Makers Authority.

b. Admin Template (Table 216) – In addition to the Base Template Authorities, the Issuing (and related) authorities, and the SID (TPer Owner) authority.

These are the only authorities that are within the scope of the specification. Additional authorities MAY be defined in a Security Subsystem Class; during SP personalization and operational use, as required and permitted by the access control settings defined here; or both.

4.5 State Behaviors

4.5.1 Issued

Behavior of an SP in the Issued state is described in the Template Reference sections, and specifically in the sections of the templates of which the SP has been constructed. Access control settings in those sections apply at the point when an SP has been Issued and before personalization occurs.

4.5.2 Issued-Disabled

If the Log template has been issued into the SP, logging in the SP's default log table MAY reflect at least the successful or unsuccessful use of the disabling and enabling functions, any failed session attempts, and failed attempts to invoke the `DeleteSP` method, dependant on personalization.

Template-specific information related to disabling of an SP that includes that template is found in the template's reference section in this document.

In the Issued-Disabled state, only a host application that is able to authenticate to the necessary access controls SHALL have the ability to re-enable the SP. Only method invocations related directly to re-enabling the SP are successful (access control requirements SHALL still be fulfilled).

Only the following method invocations to the disabled SP SHALL function (fulfilling appropriate access control requirements SHALL be required):

a. `Authenticate`

b. `Set` on the `Enabled` column of the `SPInfo` table. Access control requirements SHALL be met as normal.

c. `DeleteSP` – Access control requirements SHALL be met as normal.

In addition, the disabled state SHALL NOT affect control session methods, and session startup methods SHALL operate as normal.

The TPer owner or an authorized authority SHALL still have the ability to invoke the `Delete` method within a session to the Admin SP in order to delete the disabled SP.

All method invocations, other than those specifically identified in this section, invoked within a session to an SP in the Issued-Disabled state, SHALL fail with the SP_DISABLED status code.

4.5.3 Issued-Frozen

If the Log template has been issued into the SP, logging in the SP's default log table MAY reflect failed session startup attempts, dependent on personalization.

Attempts to open sessions to an SP in the Issued-Frozen state SHALL fail with status SP_FROZEN.

4.5.4 Issued-Disabled-Frozen

If the Log template has been issued into the SP, logging in the SP's default log table MAY reflect failed session startup attempts, dependent on personalization.

Attempts to open sessions to an SP in the Issued-Frozen state SHALL fail with status SP_FROZEN.

4.5.5 Failed

When an SP is in the Failed state, session startup methods to the SP SHALL respond with an error status SP_FAIL and session startup SHALL NOT be able to complete.

The TPer owner or an authorized authority MAY invoke the `Delete` method within a session to the Admin SP in order to delete the failed SP.

5 SP Reference

5.1 Globally Applicable SP Values

Begin Informative Content

The following sections define variables, functions, constants, or any system attribute that applies to any SP.

End Informative Content

5.1.1 Column Types Overview

The following are the primitive data types used for column types as defined by the specification. How these primitive values are stored in a table cell is implementation dependent.

a. **integer**. Signed integer. To differentiate among the type sizes, a size identifier is specified with the type, i.e., a one-byte integer is denoted as `integer_1, etc`.

b. **uinteger**. Unsigned integer. To differentiate among the type sizes, a size identifier is specified with the type, i.e. a one-byte integer is denoted as `uinteger_1,` etc.

c. **bytes**. A fixed size sequence of bytes that is used to represent any type of data such as strings, blobs, bit vectors, time/dates, etc. To differentiate among the type sizes, a size identifier is specified with the type, i.e. a one-byte bytes type is denoted as bytes_1, etc.

d. **bytes{max=n}**. A variable size sequence of bytes. To differentiate among the type sizes, a size identifier is specified with the type, i.e. a one-byte max bytes type is denoted as max_bytes_1, etc. Invocation of the `Get` method on a table cell with this type of value SHALL return the exact sequence of bytes, with the same token length, as was originally set.

The value of a `Type` object's `Format` column SHALL indicate the structure and required values of that type. The parsing of the value of this column is defined in a general manner using the following rules in ABNF (see [9]). Additional specific information is provided after the notation. In the `Format` column, the Format code and the table_kind value SHALL be encoded as a uinteger_2. All other values are encoded as indicated.

```
Type = Base_Type / Simple_Type / Enumeration_Type / Alternative_Type / List_Type /
Restricted_Reference_Type / General_Reference_Type / Named_Value_Type / Struct_Type
/ Set_Type

table_kind                      = 1/2

Base_Type                       = 0
Simple_Type                     = 1 bytes_8 uinteger_2
Enumeration_Type                = 2 1*(uinteger_2 uinteger_2)
Alternative_Type                = 3 2*bytes_8
List_Type                       = 4 uinteger_2 bytes_8
Restricted_Reference_Type       = 5/6 1*bytes_8
General_Reference_Type          = 7/8/9
General_Reference_Table_Type    = 10 table_kind
Named_Value_Name_Type           = 11 1*32bytes bytes_8
Name_Value_Integer_Type         = 12 integer_2 bytes_8
Name_Value_Uinteger_Type        = 13 uinteger_2 bytes_8
Struct_Type                     = 14 1*bytes_8
Set_Type                        = 15 1*(uinteger_2 uinteger_2)
```

a. **Base_Type.** The `Base_Type` format describes the most basic types. Other types are created using the Base Types as building blocks. The Base Types SHALL NOT be used directly. Base Types SHALL always have a `Size` column value of 0 in the `Type` table.

a. 0 – this is the Format code indicating that this is a Base_Type.

b. **Simple_Type.** The `Simple_Type` format defines an instance of one of the `Base_Type` types. The `Simple_Type` always includes a uinteger in the format column, which defines the size for that instance of that `Simple_Type`.

   a. 1 – this is the Format code indicating that this is a Simple_Type.

   b. bytes_8 – this SHALL be a uidref to a Type object that is a Base_Type.

   c. uinteger_2 – this is the size of this instantiation of the Base_Type.

c. **Enumeration_Type**. This is a n unsigned integer in a specific range.

   a. 2 – this is the Format code indicating this is an Enumeration_Type.

   b. 1*(uinteger_2 uinteger_2) – this is a number of pairs of values of uinteger_2 that represent the supported ranges of values in the enumeration.

   If a non-contiguous range of values is supported, the `Format` column SHALL contain a number of uinteger_2 pairs to identify all of the supported values.

   a. An invocation of the `CreateRow` method SHALL contain only a single pair of uinteger_2 values.

   b. Pseudo-code example: enum {0..2} represents a range of 0 to 2 inclusive.

d. **Alternative_Type**. This is a value that SHALL be an element of one of the specified types. The `Alternative_Type` format defines a union with the uinteger specifying the number of member types and followed by that many uidref{TypeObjectUID} references to the member types.

   a. 3 – this is the Format code indicating that this is an Alternative_Type

   b. 2*bytes_8 – this is a number of 2 or more uidrefs to different Type objects, other than Base_Types, that identify the options available for this type.

   Pseudo-code example: typeOr{boolean,uinteger_4,bytes_7}

e. **List_Type**. This is a sequence of values of the same type. The maximum number of elements is specified. The elements of the list are not required to be provided in any specified order. The elements of the list SHALL be returned to the host (with the `Get` method, for example) in the order in which they were received by the TPer.

   a. 4 – this is the format code indicating that this is a List_Type.

   b. uinteger_2 – this is the maximum number of elements that make up the list.

   c. bytes_8 – this SHALL be a uidref to a Type object, other than a Base_Type, that indicates the type of the elements of the list.

   Pseudo-code example: list[10]{boolean} is a list of `boolean` values, with a maximum of 10 elements.

f. **Restricted_Reference_Type**. A reference to a row SHALL be contained in a specific table or group of tables. The reference is to a physical row number (5) or a UID (6) within the table. The value of a ref is the uinteger row number for a byte table. The value of a uidref is a UID from the `UID` column of an object table. A uidref value of `0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00`, called the NULL UID, serves as a “null pointer”.

   a. 5/6 – these are the format codes indicating that this is a Restricted_Reference_Type

      i. 5 – this format code indicates that this type SHALL be the row number contained in one of the indicated byte tables.

ii. 6 – this format code indicates that this type SHALL be the UID of an object contained in one of the indicated object tables.

b. 1*bytes_8 – this is 1 or more UIDs that SHALL be to different Table UIDs that identify the tables within which the row number or uidref SHALL exist.

In this example, TableName is the name of the referenced table:

a. Pseudo-code example: uidref{ <TableName>ObjectUID }

b. Pseudo-code example: ref{ <TableName>ObjectUID }

g. **General_Reference_Type**. This is a reference to a row of some byte table, to the UID of some object, or to the UID of some table. The `General_Reference_Type` format defines a physical row number of a byte table (7), a uid of some object (8), or a uid of some table (9). The UID reserved to represent "this SP" is encompassed by a `General_Reference_Type` of 8. The value of a ref is the uinteger row number for a byte table. The value of a uidref is a UID from the `UID` column of an object table. A uidref value of `0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00`, called the NULL UID, serves as a "null pointer".

a. 7/8/9 – these are the format codes indicating that this is a General_Reference_Type.

i. 7 – this format code indicates that this type SHALL be the physical row number of a byte table.

ii. 8 – this format code indicates that this type SHALL be the UID of some object.

iii. 9 – this format code indicates that this type SHALL be the UID of some table.

Pseudo-code example: uidref{*}

Pseudo-code example: ref{*}

h. **General_Reference_Table_Type.** This is a reference to a a specific kind of table, either byte or object.

a. 10 – this format code indicates that this is a General_Reference_Table_Type.

b. table_kind (1/2) – this identifies whether the type value is the UID of an object table or the UID of a byte table.

i. 1 – this table_kind value indicates that the type value SHALL be the UID of an object table.

ii. 2 – this table_kind value indicates that type the SHALL be the UID of a byte table.

i. **Named_Value_Name_Type.** This is a Named value pair where the Name in the pair is a max_bytes_32, and the value is a uidref to the required type of the value.

a. 11 – this format code indicates that this is a Named_Value_Name_Type.

b. max_bytes_32 – this is a string with a maximum length of 32 characters that SHALL be the name submitted with the value.

c. bytes_8 – this SHALL be a uidref to a Type object, other than a Base_Type, that indicates the type of the value to be submitted.

j. **Named_Value_Integer_Type.** This is a Named value pair where the Name in the pair is an integer_2, and the value is a uidref to the required type of the value.

a. 12 – this format code indicates that this is a Named_Value_Name_Type.

b. integer_2 – this is a signed integer that SHALL be the name submitted with the value.

c. bytes_8 – this SHALL be a uidref to a Type object, other than a Base_Type, that indicates the type of the value to be submitted.

k. **Named_Value_Uinteger_Type.** This is a Named value pair where the Name in the pair is a uinteger_2, and the value is a uidref to the required type of the value.

   a. 13 – this format code indicates that this is a Named_Value_Name_Type.

   b. uinteger_2 – this is an unsigned integer that SHALL be the name submitted with the value.

   c. bytes_8 – this SHALL be a uidref to a Type object, other than a Base_Type, that indicates the type of the value to be submitted.

l. **Struct_Type.** This is a combination of different Named value types. The `Struct_Type` format indicator is followed by the number of elements and then uidrefs to the rows in the `Type` table that represent each of those elements. Name-value pairs in structs represent optional components. These MAY be excluded when passing that struct as a method parameter. When used as a column type, the size SHALL account for inclusion of all of a struct's components.

   a. 14 – this format code indicates that this is a Struct_Type

   b. 1*bytes_8 – this is a number of uidrefs to different Type objects, other than Base_Types, that identify the components of this type.

   Named value types in a struct SHALL all be different uidrefs and SHALL all be defined to utilize different names.

   If an element of a Struct is supplied when the Struct is referenced (for instance, in a method parameter), then that element SHALL appear in the order identified for that Struct in the `Type` table.

   For a Struct made up of Named value parameters A, B, C, and D, if the Struct is referenced, as in a method parameter, if element A is supplied then it SHALL be supplied first within the Struct. Other correct element orderings include:

   a. ExampleStruct [A, C, D]

   b. ExampleStruct [B, D]

   c. ExampleStruct [A, D]

   Invalid element orderings include:

   a. ExampleStruct [D, C]

   b. ExampleStruct [B, C, A]

   c. ExampleStruct [B, A, D, C]

m. **Set_Type**. A set of unsigned integers in a specific range. The `Set_Type` format defines the range of the valid elements of the set, where the first integer is the start value of the valid elements of the set and the second integer is the end value. The type itself is not limited to only a single selection from among the choices defined, as in the `Enumeration_Type`. The `Set_Type` provides the host the ability to select more than one of the options. Each SHALL appear only once in the Set. The Set MAY hold any amount of selections, from zero to the number of selections.

   a. 15 – this format code indicates that this is a Set_Type

   b. 1*(uinteger_2 uinteger_2) – this is a number of pairs of values of uinteger_2 that represent the supported ranges of values in the set.

      i. If a non-contiguous range of values is supported, the `Format` column SHALL contain a number of uinteger_2 pairs to identify all of the supported values.

ii. An invocation of the `CreateRow` method SHALL contain only a single pair of uinteger_2 values.

Pseudo-code example: Set{0..2} – Valid values for this set are made up of the following = {}, {0}, {1}, {2}, {0,1}, {0,2}, {1,2}, {0,1,2}.

5.1.2 Types Encoding

Certain column types used in messaging as method parameters (particularly in the `Set` method) utilize the interface grouping mechanisms (Named and List values) to provide clarity regarding the scope of the transmitted values.

a. Simple types – values of this type require no special handling in the messaging stream.

b. Enumeration types – values of this type require no special handling in the messaging stream.

c. Alternative types – values of this type are encoded in `Get` and `Set` methods as follows:

   a. The Alternative column type is handled similarly to a Named value in a parameter list. The Named value grouping tokens are used (SN and EN tokens, which represent "StartName" and "EndName" respectively). The Name for the pair is the last four bytes of the UID ("half_uid") of the value's `Type` object. The value in the Named value is the value of the option being set to or retrieved from the column.

   Example: When setting a 16-byte key value to the `Key` column of the `K_AES_128` table, the value would be encoded as:

   F2 A400000202 D0100001020304050607080 90A0B0C0D0E0F F3

d. List type – values of this type are encoded as follows:

   a. The List column type is handled in the same way a parameter list is handled, by using the interface List value grouping tokens (F0 and F1 tokens, which represent "[" and "]" respectively) to enclose the values in the list.

   Example: F0 tokenized_value tokenized _value tokenized _value F1

e. Restricted Reference types – values of this type require no special handling in the messaging stream.

f. General Reference types – values of this type require no special handling in the messaging stream.

g. Named value types – values of this type are encoded as follows:

   a. Values of this type are handled in the same way a Named value in a parameter list is handled, by using the Named value grouping tokens (SN and EN tokens, which represent "StartName" and "EndName" respectively) to enclose the name-value pair.

   Example: F2 tokenized_name tokenized_value F3

h. Struct value types – Structs allow the creation of composite types by combining Named value types and other types. Values of the struct type are made up of either optional Named value types, or other types that are required to be supplied. The optional types MAY NOT be included when sending values for a struct. Values of this type are encoded as follows:

   a. The struct itself is delimited using the List value grouping tokens (F0 and F1 tokens, which represent "[" and "]" respectively) to enclose the values in the struct. The Named values that make up the values stored in the struct are each grouped using the interface Named value-grouping tokens (SN and EN tokens, which represent "StartName" and "EndName" respectively) to enclose each name-value pair.

Example: F0 F2 tokenized_name tokenized_value F3 F2 tokenized_name tokenized_value F3 F1

i. Set value types – values of this type are encoded as follows:

   a. The Set column type is handled in the same way that the List type is handled, by using the interface List value grouping tokens (F0 and F1 tokens, which represent "[" and "]" respectively) to enclose the values in the Set.

   Example: F0 tokenized_value tokenized_value F1

5.1.3 Column Types

This section describes each of the column types in the Template Reference sections of the Core Specification. The `UID`, `Name`, and `Format` columns identify the column values of the `Type` table. These values SHALL comprise the `Type` table for every SP, prior to any personalization. These types SHALL NOT be able to be changed or deleted by the host.

Included in this section are descriptions of the column types for each column of each table defined in this specification, as well as descriptions of each of the component types of the column types. Component types are types that have entries in the `Type` table, but are not referenced directly as column types. They are used to make up other types that do represent column types.

The `UID` column in the description table in each section SHALL be the UID for that type.

The `Name` column specifies the name for that type.

The `Format` column identifies the structure of the associated type. The first value in the `Format` column is the name of that type's Format code. Additional values listed in the column are determined by the type's format code. For readability, the names of `Type` objects are used in place of their UID, and commas are used to separate values.

An asterisk (*) in any of the descriptive tables indicates SSC-specific or implementation-specific values.

5.1.3.1 AC_element

An AC_element is a list type made up of ACE_expressions. The size of the AC_element list is implementation-dependant. A minimum size restriction MAY be defined by an SSC.

Table 46 AC_element

UID	Name	Format
00 00 00 05 00 00 08 01	AC_element	List_Type, *, ACE_expression

5.1.3.2 ACE_columns

This Set type identifies the columns to which an ACE applies. The values are: 0=Column0, 1=Column1, 2=Column2, etc. Each value in the set maps to a "Column Number". The size of the set is SSC/implementation dependant based on the maximum number of columns allowed in a table. For tables created from templates, the table descriptions in this specification indicate the ordering of the columns, such that the first column listed in a table description is "Column0", the second is "Column1", etc. For object tables created using the `CreateTable` method, the `UID` column SHALL be Column Number 0, the first column defined in the Columns parameter of `CreateTable` SHALL be Column Number 1, etc.

Table 47 ACE_columns

UID	Name	Format

UID	Name	Format
00 00 00 05 00 00 1A 03	ACE_columns	Set_Type, 0, *

5.1.3.3 ACE_expression

This is an alternative type where the options are either a uidref to an `Authority` object or one of the boolean_ACE (AND = 0 and OR = 1) options. This type is used within the AC_element list to form a postfix Boolean expression of Authorities.

Table 48 ACE_expression

UID	Name	Format
00 00 00 05 00 00 06 01	ACE_expression	Alternative_Type, Authority_object_ref, boolean_ACE

5.1.3.4 ACE_object_ref

This type describes a uidref to an object contained in the `ACE` table.

Table 49 ACE_object_ref

UID	Name	Format
00 00 00 05 00 00 0C 04	ACE_object_ref	Restricted_Reference_Type{6}, uidref{ACETableUID}

5.1.3.5 ACL

The ACL type is a list of uidrefs to `ACE` objects. The length of the list, and therefore the number of ACEs that MAY be included in a single Access Control List, is SSC/implementation dependant.

Table 50 ACL

UID	Name	Format
00 00 00 05 00 00 08 02	ACL	List_Type, *, ACE_object_ref

5.1.3.6 adv_key_mode

This enumeration type defines the behavior of the `NextKey` column.

Table 51 adv_key_mode

UID	Name	Format
00 00 00 05 00 00 04 0F	adv_key_mode	Enumeration_Type, 0, 7

The enumeration values are associated with key behaviors as defined in Table 52.

Table 52 adv_key_mode Enumeration Values

Enumeration Value	Behavior
0	Wait for AdvKey_Req
1	Auto-advance keys
2-7	Reserved

5.1.3.7 attr_flags

This set type describes the types of attributes available for the `AttributeFlags` column of the `Column` table.

Table 53 attr_flags

UID	Name	Format
00 00 00 05 00 00 1A 04	attr_flags	Set_Type, 0, 31

The set values are associated with column behaviors as defined in Table 54.

Table 54 attr_flags Set Values

Set Value	Behavior
0	Get Not Permitted
1	Set Not Permitted
2-31	Reserved

5.1.3.8 auth_method

This enumeration type is used to represent the authentication methods that MAY be used to authenticate authorities (see 5.3.4.1.3).

Table 55 auth_method

UID	Name	Format
00 00 00 05 00 00 04 08	auth_method	Enumeration_Type, 0, 23

The enumeration values are associated with authentication methods as defined in Table 56.

Table 56 auth_method Enumeration Values

Enumeration Value	Authentication Method
0	None
1	Password
2	Exchange
3	Sign

Enumeration Value	Authentication Method
4	SymK
5	HMAC
6	TPerSign
7	TPerExchange
8-23	Reserved

5.1.3.9 Authority_object_ref

The Authority_object_ref type describes a uidref to an object in the `Authority` table.

Table 57 Authority_object_ref

UID	Name	Format
00 00 00 05 00 00 0C 05	Authority_object_ref	Restricted_Reference_Type{6}, uidref {AuthorityTableUID}

5.1.3.10 boolean

The boolean column type is an enumeration used to represent True or False.

Table 58 boolean

UID	Name	Format
00 00 00 05 00 00 04 01	boolean	Enumeration_Type, 0, 1

The enumeration values are associated as defined in Table 59.

Table 59 boolean Enumeration Values

Enumeration Value	Associated Value
0	False
1	True

5.1.3.11 boolean_ACE

This enumeration is used to identify the Boolean operators "And", "Or", and "Not".

Table 60 boolean_ACE

UID	Name	Format
00 00 00 05 00 00 04 0E	boolean_ACE	Enumeration_Type, 0, 2

The enumeration values are associated with Boolean operators as defined in Table 61.

Table 61 boolean_ACE Enumeration Values

Enumeration Value	Operator
0	And
1	Or
2	Not

5.1.3.12 byte_row_ref

Type used for referencing a row in a byte table.

Table 62 byte_row_ref

UID	Name	Format
00 00 00 05 00 00 0F 01	byte_row_ref	General_Reference_Type {7}

5.1.3.13 byte_table_ref

This is a reference type that SHALL be used specifically for uidrefs to byte tables. When performing type checking, as part of that type checking the TPer SHALL validate that this uidref is to a table that is a byte table.

Table 63 byte_table_ref

UID	Name	Format
00 00 00 05 00 00 10 01	byte_table_ref	General_Reference_Table_Type, 2

5.1.3.14 bytes

This type represents the bytes base type, and is used to represent a value made up of a fixed-size sequence of bytes.

Table 64 bytes

UID	Name	Format
00 00 00 05 00 00 00 02	bytes	Base_Type

5.1.3.15 bytes_4

This is a bytes type with a size requirement of 4.

Table 65 bytes_4

UID	Name	Format
00 00 00 05 00 00 02 38	bytes_4	Simple_Type, bytes, 4

5.1.3.16 bytes_12

This is a bytes type with a size requirement of 12.

Table 66 bytes_12

UID	Name	Format
00 00 00 05 00 00 02 01	bytes_12	Simple_Type, bytes, 12

5.1.3.17 bytes_16

This is a bytes type with a size requirement of 16.

Table 67 bytes_16

UID	Name	Format
00 00 00 05 00 00 02 02	bytes_16	Simple_Type, bytes, 16

5.1.3.18 bytes_20

This is a bytes type with a size requirement of 20.

Table 68 bytes_20

UID	Name	Format
00 00 00 05 00 00 02 36	bytes_20	Simple_Type, bytes, 20

5.1.3.19 bytes_32

This is a bytes type with a size requirement of 32.

Table 69 bytes_32

UID	Name	Format
00 00 00 05 00 00 02 05	bytes_32	Simple_Type, bytes, 32

5.1.3.20 bytes_48

This is a bytes type with a size requirement of 48.

Table 70 bytes_48

UID	Name	Format
00 00 00 05 00 00 02 37	bytes_48	Simple_Type, bytes, 48

5.1.3.21 bytes_64

This is a bytes type with a size requirement of 64.

Table 71 bytes_64

UID	Name	Format
00 00 00 05 00 00 02 06	bytes_64	Simple_Type, bytes, 64

5.1.3.22 Certificates_object_ref

The Certificates_object_ref type describes a uidref to an object in the `Certificates` table.

Table 72 Certificates_object _ref

UID	Name	Format
00 00 00 05 00 00 0C 06	Certificates_object_ref	Restricted_Reference_Type{6}, uidref {CertificatesTableUID}

5.1.3.23 clock_kind

This enumeration type is used to define the type of clock currently active.

Table 73 clock_kind

UID	Name	Format
00 00 00 05 00 00 04 0B	clock_kind	Enumeration_Type, 0, 3

The enumeration values are associated as defined in Table 74.

Table 74 clock_kind Enumeration Values

Enumeration Value	Associated Value
0	Timer
1	Low
2	High
3	LowAndHigh

5.1.3.24 clock_time

This is a struct type made up of name-value pairs, and is used to represent time. Any value not supplied is treated as 0.

If the host has supplied a trusted time since powerup, that time is used; otherwise a monotonic counter is used.

The clock_time type represents times in either Generalized Time or UTC Time. Using this type to represent UTC Time requires 0's (zeroes) in fields where Generalized time requires a value but UTC Time does not (i.e. 2006 in UTC Time would be represented as 0006). Per the definition for the component types, the names for these name-value types are `0x00` (for the Year), `0x01` (for the Month),

0x02 (for the Day), 0x03 (for the Hour), 0x04 (for the Minute), 0x05 (for the Seconds), and 0x06 (for the Fraction).

Table 75 clock_time

UID	Name	Format
00 00 00 05 00 00 18 05	clock_time	Struct_Type, Year, Month, Day, Hour, Minute, Seconds, Fractoin

5.1.3.25 Column_object _ref

The Column_object _ref type describes a uidref to an object in the Column table.

Table 76 Column_object _ref

UID	Name	Format
00 00 00 05 00 00 0C 07	Column_object_ref	Restricted_Reference_Type{6}, uidref {ColumnTable_UID}

5.1.3.26 cred_object_uidref

The cred_object_uidref type is a restricted reference type that SHALL be used specifically for uidrefs to credential objects. When performing type checking, as part of that type checking the TPer SHALL validate that this uidref is to an object in a credential (C_*) table.

In the Format column of Table 77, the * is used to indicate the entire range of that particular type of credential table.

Table 77 cred_object_uidref

UID	Name	Format
00 00 00 05 00 00 0C 0B	cred_object_uidref	Restricted_Reference_Type{6}, uidref {C_PINTableUID}, uidref {C_AES_*TableUID}, uidref {C_RSA_*TableUID}, uidref{C_EC_*TableUID}, uidref{C_HMAC_*TableUID}

5.1.3.27 date

The date type represents the date portion of the time from the system clock. This is a set of name-value pairs, with the names 0x00 (for the Year), 0x01 (for the Month), and 0x02 (for the Day).

Table 78 date

UID	Name	Format
00 00 00 05 00 00 18 04	date	Struct_Type, Year, Month, Day

5.1.3.28 Day

Name-value pair that has a Name of "2" and takes day_enum as the value.

Table 79 Day

UID	Name	Format
00 00 00 05 00 00 14 03	Day	Name_Value_Uinteger_Type, 2, day_enum

5.1.3.29 day_enum

Used in association with the Day name-value pair.

Table 80 day_enum

UID	Name	Format
00 00 00 05 00 00 04 18	day_enum	Enumeration_Type, 1, 31

5.1.3.30 enc_supported

This enumeration type is used to define the types of user data encryption supported by the TPer.

Table 81 enc_supported

UID	Name	Format
00 00 00 05 00 00 04 1D	enc_supported	Enumeration_Type, 0, 15

The enumeration values are associated as defined in Table 82.

Table 82 enc_supported Enumeration Values

Enumeration Value	Associated Value
0	None
1	Media Encryption
2-15	Reserved

5.1.3.31 feedback_size

This uinteger type represents the feedback sizes for AES used in CFB mode. If AES Mode is CFB, this SHALL be between 1 and the block length.

Table 83 feedback_size

UID	Name	Format

UID	Name	Format
00 00 00 05 00 00 02 14	feedback_size	Simple_Type, uinteger, 2

5.1.3.32 Fraction

Name-value pair that has a Name of "6" and takes fraction enum as the value.

Table 84 Fraction

UID	Name	Format
00 00 00 05 00 00 14 07	Fraction	Name_Value_Uinteger_Type, 6, fraction_enum

5.1.3.33 fraction_enum

Used in association with the Fraction name-value pair.

Table 85 fraction_enum

UID	Name	Format
00 00 00 05 00 00 04 1C	fraction_enum	Enumeration_Type, 0, 999

5.1.3.34 gen_status

This set type is used to identify the general status of the re-encryption process.

Table 86 gen_status

UID	Name	Format
00 00 00 05 00 00 1A 02	gen_status	Set_Type, 0, 63

The enumeration values are associated as defined in table Table 87. Values 0-31 are valid for the `PAUSED` state, value 32-63 are valid for the `PENDING` state (see 5.7.3.3).

Table 87 gen_status Enumeration Values

Column Value	Associated Value	Meaning
0	None	
1	`pending_tper_error`	Last `ReEncryptState` value was `PENDING` AND a `TPer_Error_Detect` condition was detected
2	`active_tper_error`	Last `ReEncryptState` value was `ACTIVE` AND a `TPer_Error_Detect` condition was detected
3	`active_pause_requested`	Last `ReEncryptState` value was `ACTIVE` AND `PAUSE_req` was detected

Column Value	Associated Value	Meaning
4	pend_pause_requested	Last ReEncryptState value was PENDING AND a PAUSE_req value was detected
5	pend_reset_stop_detect	A reset condition AND its associated ContOnReset configuration does not allow re-encryption to continue AND last state was PENDING
6	key_error	ReEncryptState value was PENDING AND valid keys were not found in any C_* table OR insufficient access control granted for reading C_* table.
7 to 31	reserved	
32	wait_AvailableKeys	keys are not available
33	wait_for_TPer_resources	TPer_Ready condition is not True
34	active_reset_stop_detect	A reset condition AND its associated ContOnReset configuration does not allow re-encryption to continue AND last ReEncryptState value was ACTIVE
34-63	reserved	

5.1.3.35 hash_protocol

This enumeration type determines the hash algorithm to be used when creating a digital signature.

Table 88 hash_protocol

UID	Name	Format
00 00 00 05 00 00 04 0D	hash_protocol	Enumeration_Type, 0, 15

The enumeration values are associated as defined in Table 89.

Table 89 hash_protocol Enumeration Values

Enumeration Value	Associated Value
0	None
1	SHA 1
2	SHA 256
3	SHA 384
4	SHA 512
5-15	Reserved

5.1.3.36 Hour

Name-value pair that has a Name of "3" and takes hour_enum as the value.

Table 90 Hour

UID	Name	Format
00 00 00 05 00 00 14 04	Hour	Name_Value_Uinteger_Type, 3, hour_enum

5.1.3.37 hour_enum

Used in association with the Hour name-value pair.

Table 91 hour_enum

UID	Name	Format
00 00 00 05 00 00 04 19	hour_enum	Enumeration_Type, 0, 23

5.1.3.38 integer

This is the base type used to represent a signed integer.

Table 92 integer

UID	Name	Format
00 00 00 05 00 00 00 04	integer	Base_Type

5.1.3.39 integer_1

This is an integer type with a size limit of 1 byte.

Table 93 integer_1

UID	Name	Format
00 00 00 05 00 00 02 10	integer_1	Simple_Type, integer, 1

5.1.3.40 integer_2

This is an integer type with a size limit of 2 bytes.

Table 94 integer_2

UID	Name	Format
00 00 00 05 00 00 02 15	integer_2	Simple_Type, integer, 2

5.1.3.41 key_128

This is an alternative type, with options for various key sizes.

Table 95 key_128

UID	Name	Format
00 00 00 05 00 00 06 02	key_128	Alternative_Type, bytes_16, bytes_32

5.1.3.42 key_256

This is an alternative type, with options for various key sizes.

Table 96 key_256

UID	Name	Format
00 00 00 05 00 00 06 03	key_256	Alternative_Type, bytes_32, bytes_64

5.1.3.43 keys_avail_conds

This enumeration describes the conditions required to assert `KeysAvailable` in the `Locking` table.

Table 97 keys_avail_conds

UID	Name	Format
00 00 00 05 00 00 04 10	keys_avail_conds	Enumeration_Type, 0, 7

The enumeration values are associated as defined in Table 98.

Table 98 keys_avail_conds Enumeration Values

Enumeration Value	Associated Value
0	None
1	Authentication of an authority with `Set` access to any of the `ReadLocked`, `WriteLocked`, `ReadLockEnabled` or `WriteLockEnabled` columns for that LBA range
2-7	Reserved

5.1.3.44 lag

A struct made up of 2 uinteger_2 name-value types, used to define the lag when setting time. The two types represent seconds and fraction of seconds. The names required, as defined by the component types, are `0x05` ("Seconds") for the first value and `0x06` ("Fraction") for the second. The "Fraction" value is a number of milliseconds.

Table 99 lag

UID	Name	Format

UID	Name	Format
00 00 00 05 00 00 18 02	lag	Struct_Type, Seconds, Fraction

5.1.3.45 last_reenc_stat

This enumeration identifies the last attempted re-encryption step.

Table 100 last_reenc_stat

UID	Name	Format
00 00 00 05 00 00 04 11	last_reenc_stat	Enumeration_Type, 0, 7

The enumeration values are associated as defined in Table 101.

Table 101 last_reenc_stat Enumeration Values

Enumeration Value	Associated Value
0	Success
1	Read Error
2	Write Error
3	Verify Eror
4-7	Reserved

5.1.3.46 life_cycle_state

This enumeration is used to represent the current life cycle state of the SP.

Table 102 life_cycle_state

UID	Name	Format
00 00 00 05 00 00 04 05	life_cycle_state	Enumeration_Type, 0, 15

The enumeration values are associated as defined in Table 103.

Table 103 life_cycle_state Enumeration Values

Enumeration Value	Associated Value
0	Issued
1	Issued-Disabled
2	Issued-Frozen
3	Issued-Disabled-Frozen
4	Issued-Failed

Enumeration Value	Associated Value
5-7	Unassigned
8-13	Reserved for SSC Usage
14-15	Unassigned

5.1.3.47 LogList_object_ref

The LogList_object_ref type describes a uidref to an object in the `LogList` table.

Table 104 LogList_object_ref

UID	Name	Format
00 00 00 05 00 00 0C 0D	LogList_object_ref	Restricted_Reference_Type{6}, uidref {LogListTableUID}

5.1.3.48 log_row_ref

This type SHALL be used specifically for rows in Log tables. When performing type checking, as part of that type checking the TPer SHALL validate that this is the uid of a row in a `Log` table.

The * in the `Format` column of Table 105 indicates that other Log tables besides the default log MAY exist in a particular SP, and that the `Format` column value for this type also includes those.

Table 105 log_row_ref

UID	Name	Format
00 00 00 05 00 00 0C 0A	log_row_ref	Restricted_Reference_Type {6}, uidref{LogTableUID}, *

5.1.3.49 log_select

This enumeration is used to identify the scope of the logging for an access control association or authority authentication.

Table 106 log_select

UID	Name	Format
00 00 00 05 00 00 04 0C	log_select	Enumeration_Type, 0, 3

The enumeration values are associated as defined in Table 107.

Table 107 log_select Enumeration Values

Enumeration Value	Associated Value
0	None
1	LogSuccess
2	LogFail
3	LogAlways

5.1.3.50 max_bytes

This is the base type that is used to represent a bytes value that is equal to or less than the size specified for the type instance.

Table 108 max_bytes

UID	Name	Format
00 00 00 05 00 00 00 03	max_bytes	Base_Type

5.1.3.51 max_bytes_32

This is a max bytes type that provides a maximum size of 32.

Table 109 max_bytes_32

UID	Name	Format
00 00 00 05 00 00 02 0D	max_bytes_32	Simple_Type, max_bytes, 32

5.1.3.52 max_bytes_64

This is a max bytes type that provides a maximum size of 64.

Table 110 max_bytes_64

UID	Name	Format
00 00 00 05 00 00 02 0E	max_bytes_64	Simple_Type, max_bytes, 64

5.1.3.53 mediakey_obj_uidref

This is a restricted reference type that SHALL be used specifically for uidrefs to media encryption key objects (in the `K_*` tables). When performing type checking, as part of that type checking the TPer SHALL validate that this uidref is to an object in a media encryption key table.

Table 111 mediakey_obj_uidref

UID	Name	Format
00 00 00 05 00 00 0C 0C	mediakey_object_uidref	Restricted_Reference_Type{6}, uidref {K_AES_128TableUID}, uidref {K_AES_256TableUID}

5.1.3.54 MethodID_object _ref

The MethodID_object _ref type describes a uidref to an object in the `MethodID` table.

Table 112 MethodID_object _ref

UID	Name	Format
00 00 00 05 00 00 0C 03	MethodID_object_ref	Restricted_Reference_Type{6}, uidref {MethodIDTableUID}

5.1.3.55 messaging_type

This enumeration is used to describe the options for selecting secure messaging.

Table 113 messaging_type

UID	Name	Format
00 00 00 05 00 00 04 04	messaging_type	Enumeration_Type, 0, 255

The enumeration values and their associations defined in Table 179.

5.1.3.56 Minute

Name-value pair that has a Name of "" and takes minute_enum as the value.

Table 114 Minute

UID	Name	Format
00 00 00 05 00 00 14 05	Minute	Name_Value_Uinteger_Type, 4, minute_enum

5.1.3.57 minute_enum

Used in association with the Minute name-value pair.

Table 115 minute_enum

UID	Name	Format
00 00 00 05 00 00 04 1A	minute_enum	Enumeration_Type, 0, 59

5.1.3.58 Month

Name-value pair that has a Name of "1" and takes month_enum as the value.

Table 116 Month

UID	Name	Format
00 00 00 05 00 00 14 02	Month	Name_Value_Uinteger_Type, 1, month_enum

5.1.3.59 month_enum

Used in association with the Month name-value pair.

Table 117 month_enum

UID	Name	Format

UID	Name	Format
00 00 00 05 00 00 04 17	month_enum	Enumeration_Type, 1, 12

5.1.3.60 name

This max bytes type, with a size limitation of 32, is used to represent names.

Table 118 name

UID	Name	Format
00 00 00 05 00 00 02 0B	name	Simple_Type, max_bytes, 32

5.1.3.61 object_ref

Type used for referencing an object in an object table.

Table 119 object_ref

UID	Name	Format
00 00 00 05 00 00 0F 02	object_ref	General_Reference_Type {8}

5.1.3.62 padding_type

This enumeration is used to identify the type of padding used with RSA encryption. RSAES-PKCS1-v1_5 or RSAES-OAEP (see [18]) SHALL be used for RSA encryption. RSASSA-PKCS1-v1_5 or RSASSA-PSS (see [18]) SHALL be used for RSA signing.

Table 120 padding_type

UID	Name	Format
00 00 00 05 00 00 04 06	padding_type	Enumeration_Type, 0, 15

The enumeration values are associated as defined in Table 121.

Table 121 padding_type Enumeration Values

Enumeration Value	Associated Value
0	None
1	None
2	RSAES-PKCS1-v1_5
3	RSAES-OAEP
4	RSASSA-PKCS1-v1_5
5-15	Reserved

5.1.3.63 password

This max bytes type, with a size limitation of 32, is used in the C_PIN table.

Table 122 password

UID	Name	Format
00 00 00 05 00 00 02 0C	password	Simple_Type, max_bytes, 32

5.1.3.64 protect_types

This set is used to identify the protection mechanisms in operation when a column is identified as hidden.

Table 123 protect_types

UID	Name	Format
00 00 00 05 00 00 1A 05	protect_types	Set_Type, 0, 255

The empty set indicates that keys are not hidden. The values of the set are all applied to the protected value. The set values are assigned in [3].

5.1.3.65 reencrypt_request

This enumeration is used to identify the host re-encryption request value.

Table 124 reencrypt_request

UID	Name	Format
00 00 00 05 00 00 04 13	reencrypt_request	Enumeration_Type, 1, 16

The enumeration values are associated as defined in 5.7.2.2.14.

5.1.3.66 reencrypt_state

This enumeration type identifies the present re-encryption state for an LBA range.

Table 125 reencrypt_state

UID	Name	Format
00 00 00 05 00 00 04 14	reencrypt_state	Enumeration_Type, 1, 16

The enumeration values are associated as defined in Table 126.

Table 126 reencrypt_state Enumeration Values

Enumeration Value	Associated Value
1	Idle
2	Pending
3	Active
4	Completed
5	Paused
6-16	Reserved

5.1.3.67 reset_types

This Set type identifies the various TCG reset options available.

Table 127 reset_types

UID	Name	Format
00 00 00 05 00 00 1A 01	reset_types	Set_Type, 0, 31

The Set values are associated as defined in Table 128.

Table 128 reset_types Set Values

Set Value	Associated Value
0	Power Cycle
1	Hardware
2	HotPlug
3-15	Reserved
16-31	Vendor Unique

5.1.3.68 Seconds

Name-value pair that has a Name of "5" and takes seconds_enum as the value.

Table 129 Seconds

UID	Name	Format
00 00 00 05 00 00 14 06	Seconds	Name_Value_Uinteger_Type, 5, seconds_enum

5.1.3.69 seconds_enum

Used in association with the Seconds name-value pair.

Table 130 seconds_enum

UID	Name	Format
00 00 00 05 00 00 04 1B	seconds_enum	Enumeration_Type, 0, 59

5.1.3.70 SPTemplates_object _ref

The SPTemplates_object _ref type describes a uidref to an object in the `SPTemplates` table.

Table 131 SPTemplates_object _ref

UID	Name	Format
00 00 00 05 00 00 0C 01	SPTemplates_object_ref	Restricted_Reference_Type{6}, uidref{SPTemplatesTableUID}

5.1.3.71 SSC

This is a list of names used to represent the SSCs that a TPer supports.

Table 132 SSC

UID	Name	Format
00 00 00 05 00 00 08 03	SSC	List_Type, *, name

5.1.3.72 symmetric_mode

Defines the mode to be used with an AES credential.

Table 133 symmetric_mode

UID	Name	Format
00 00 00 05 00 00 04 0A	symmetric_mode	Enumeration_Type, 0, 23

The enumeration values are associated as defined in Table 134.

Table 134 symmetric_mode Enumeration Values

Enumeration Value	Associated Value
0	ECB
1	CBC
2	CFB
3	OFB

Enumeration Value	Associated Value
4	GCM
5	CTR
6	CCM
7	XTS
8	LRW
9	EME
10	CMC
11	XEX
12-23	Reserved

5.1.3.73 symmetric_mode_media

Defines the modes availableto be used with AES for user data encryption.

Table 135 symmetric_mode_media

UID	Name	Format
00 00 00 05 00 00 04 03	symmetric_mode_media	Enumeration_Type, 0, 23

The enumeration values are associated as defined in Table 134.

Table 136 symmetric_mode_media Enumeration Values

Enumeration Value	Associated Value
0	ECB
1	CBC
2	CFB
3	OFB
4	GCM
5	CTR
6	CCM
7	XTS
8	LRW
9	EME
10	CMC
11	XEX
12-22	Reserved
23	Media Encryption

5.1.3.74 table_kind

Defines the kinds of tables.

Table 137 table_kind

UID	Name	Format
00 00 00 05 00 00 04 15	table_kind	Enumeration_Type, 1, 8

The enumeration values are associated as defined in Table 138.

Table 138 table_kind Enumeration Values

Enumeration Value	Table Type
1	Object
2	Byte
3-8	Reserved

5.1.3.75 table_or_object_ref

This alternative type defines a reference to either the uid of a table or the uid of some object, or the UID of "ThisSP".

Table 139 table_or_object_ref

UID	Name	Format
00 00 00 05 00 00 06 06	table_or_object_ref	Alternative_Type, object_ref, table_ref

5.1.3.76 Table_object _ref

The Table_object _ref type describes a uidref to an object in the `Table` table.

Table 140 Table_object _ref

UID	Name	Format
00 00 00 05 00 00 0C 09	Table_object_ref	Restricted_Reference_Type{6}, uidref {TableTableUID}

5.1.3.77 table_ref

Type used for referencing a table.

Table 141 table_ref

UID	Name	Format
00 00 00 05 00 00 0F 03	table_ref	General_Reference_Type {9}

5.1.3.78 Template_object _ref

The Template_object _ref type describes a uidref to an object in the Admin SP's `Template` table.

Table 142 Template_object _ref

UID	Name	Format
00 00 00 05 00 00 0C 08	Template_object_ref	Restricted_Reference_Type{6}, uidref {TemplateTableUID}

5.1.3.79 type_def

The type_def type describes the format of the `Type` table's `Format` column. The value in the `Format` column of this type SHALL be encoded and parseable based on the notation description of the type formats (see 5.1.1).

Table 143 type_def

UID	Name	Format
00 00 00 05 00 00 02 03	type_def	Simple_Type, max_bytes, *

5.1.3.80 Type_object _ref

The Type_object _ref type describes a uidref to an object in the `Type` table.

Table 144 Type_object _ref

UID	Name	Format
00 00 00 05 00 00 0C 02	Type_object_ref	Restricted_Reference_Type{6}, uidref {TypeTableUID}

5.1.3.81 uid

This is the type used for the `UID` column of object tables.

Table 145 uid

UID	Name	Format
00 00 00 05 00 00 02 09	uid	Simple_Type, bytes, 8

5.1.3.82 uinteger

This is the base type that is used to represent an unsigned integer.

Table 146 uinteger

UID	Name	Format
00 00 00 05 00 00 00 05	uinteger	Base_Type

5.1.3.83 uinteger_1

This is a uinteger type with a size restriction of 1 byte.

Table 147 uinteger_1

UID	Name	Format
00 00 00 05 00 00 02 11	uinteger_1	Simple_Type, uinteger, 1

5.1.3.84 uinteger_128

This is a uinteger type with a size restriction of 128 bytes.

Table 148 uinteger_128

UID	Name	Format
00 00 00 05 00 00 02 12	uinteger_128	Simple_Type, uinteger, 128

5.1.3.85 uinteger_2

This is a uinteger type with a size restriction of 2 bytes.

Table 149 uinteger_2

UID	Name	Format
00 00 00 05 00 00 02 15	uinteger_2	Simple_Type, uinteger, 2

5.1.3.86 uinteger_20

This is a uinteger type with a size restriction of 20 bytes.

Table 150 uinteger_20

UID	Name	Format
00 00 00 05 00 00 02 16	uinteger_20	Simple_Type, uinteger, 20

5.1.3.87 uinteger_21

This is a uinteger type with a size restriction of 21 bytes.

Table 151 uinteger_21

UID	Name	Format
00 00 00 05 00 00 02 17	uinteger_21	Simple_Type, uinteger, 21

5.1.3.88 uinteger_24

This is a uinteger type with a size restriction of 24 bytes.

Table 152 uinteger_24

UID	Name	Format
00 00 00 05 00 00 02 18	uinteger_24	Simple_Type, uinteger, 24

5.1.3.89 uinteger_256

This is a uinteger type with a size restriction of 256 bytes.

Table 153 uinteger_256

UID	Name	Format
00 00 00 05 00 00 02 19	uinteger_256	Simple_Type, uinteger, 256

5.1.3.90 uinteger_28

This is a uinteger type with a size restriction of 28 bytes.

Table 154 uinteger_28

UID	Name	Format
00 00 00 05 00 00 02 1A	uinteger_28	Simple_Type, uinteger, 28

5.1.3.91 uinteger_30

This is a uinteger type with a size restriction of 30 bytes.

Table 155 uinteger_30

UID	Name	Format
00 00 00 05 00 00 02 1B	uinteger_30	Simple_Type, uinteger, 30

5.1.3.92 uinteger_36

This is a uinteger type with a size restriction of 36 bytes.

Table 156 uinteger_36

UID	Name	Format
00 00 00 05 00 00 02 1F	uinteger_36	Simple_Type, uinteger, 36

5.1.3.93 uinteger_4

This is a uinteger type with a size restriction of 4 bytes.

Table 157 uinteger_4

UID	Name	Format
00 00 00 05 00 00 02 20	uinteger_4	Simple_Type, uinteger, 4

5.1.3.94 uinteger_48

This is a uinteger type with a size restriction of 48 bytes.

Table 158 uinteger_48

UID	Name	Format
00 00 00 05 00 00 02 23	uinteger_48	Simple_Type, uinteger, 48

5.1.3.95 uinteger_64

This is a uinteger type with a size restriction of 64 bytes.

Table 159 uinteger_64

UID	Name	Format
00 00 00 05 00 00 02 24	uinteger_64	Simple_Type, uinteger, 64

5.1.3.96 uinteger_66

This is a uinteger type with a size restriction of 66 bytes.

Table 160 uinteger_66

UID	Name	Format
00 00 00 05 00 00 02 27	uinteger_66	Simple_Type, uinteger, 66

5.1.3.97 uinteger_8

This is a uinteger type with a size restriction of 8 bytes.

Table 161 uinteger_8

UID	Name	Format
00 00 00 05 00 00 02 25	uinteger_8	Simple_Type, uinteger, 8

5.1.3.98 verify_mode

This enumeration type defines the verification operation the TPer SHALL perform during the re-encryption process after a sector has been written with the new encryption key.

Table 162 verify_mode

UID	Name	Format
00 00 00 05 00 00 04 12	verify_mode	Enumeration_Type, 0, 7

The enumeration values are associated as defined in Table 163.

Table 163 verify_mode Enumeration Values

Enumeration Value	Associated Value
0	No verify
1	Verify enabled
2-7	Reserved

5.1.3.99 Year

Name-value pair that has a Name of "0" and takes year_enum as the value.

Table 164 Year

UID	Name	Format
00 00 00 05 00 00 14 01	Year	Name_Value_Uinteger_Type, 0, year_enum

5.1.3.100 year_enum

Used in association with the Year name-value pair.

Table 165 year_enum

UID	Name	Format
00 00 00 05 00 00 04 16	year_enum	Enumeration_Type, 1970, 9999

5.1.4 Abstract Types

Begin Informative Content

Abstract types are representations of grouped interface types, or interface types that have limits on their legal values, that are used specifically for encoding method parameters. These representations are used primarily for documentation purposes, as part of the pseudo-code method signatures, to simplify the description of those methods.

Abstract types do not affect the operation or regular encoding of a method, nor are they used as column types or represented in the Type table (though they resemble some of these types in structure, name, or both). The primary goals of the abstract type constructs are to simplify the pseudo-code description of the methods themselves, and to provide insight into grouping using the List and Named value tokens introduced previously.

End Informative Content

5.1.4.1 Name Representations in Abstract Type Named Value Components

Named values used in abstract types SHALL be encoded in the messaging stream using the rules described in this section.

a. The name in the Named values that represent Named value components of method parameters in a method invocation SHALL be a uinteger. Starting at zero, these uinteger values are assigned based on the ordering of the components of these abstract types.
    a. The first component of one of these grouped types SHALL be represented by the "name" zero (`0x00`) in the Named value pair when that method is invoked, and thus has the format "`0x00` = value" when that method is invoked.
    b. Each subsequent component in the grouped type after the first SHALL be represented by the uinteger of the previous component, as indicated in the method's signature or the abstract type definition, incremented by one. Thus, the second component of such a type in an invocation of a particular method has the format "`0x01` = value". Components of such types are not required to be sent in a method invocation, but if sent must appear in the order specified.
    c. For each subsequent relevant type grouping in the method invocation, if such exists, the components SHALL be numbered restarting at `0x00`.

5.1.4.2 Abstract Type Definitions

Begin Informative Content

The following sections describe the pseudo-code parameters that each of these abstract types represent when they appear in a pseudo-code method signature.

End Informative Content

5.1.4.2.1 access_control_list

An access_control_list is a list of uidrefs to objects in the `ACE` table. The length of the list is implementation/SSC-specific.

Format:

```
[ uidref ... ]
```

5.1.4.2.2 boolean

This abstract type is similar to an enumeration column type, and has a valid range of the integer 0 to the integer 1, where 0 is used to represent "False" and 1 is used to represent "True".

Format:

```
uinteger
```

In the messaging stream, "False" is represented as `0x00` and "True" is represented as `0x01`.

5.1.4.2.3 cell_block

This type represents a grouping of Named values that are used to identify a portion of a table. In messaging, this grouping is enclosed by List value delimiters, and each component is enclosed by Named value delimiters.

The name of each component is a uinteger representing the positioning of that component within the grouping, as defined by this specification. The associated number also appears in the appropriate component description.

Because this is a group of Named values, its separate components are optional. However, there are default requirements if components are omitted, and certain requirements for the values assigned to these components depending on the context in which the method is invoked. These requirements are as follows:

a. Table – this Named value has the Name "`0x00`" and a value that is a uid to a table.
    a. If the value with Name "`0x00`" is omitted, then the operation defaults to the table upon which the method was invoked.
    b. Table SHALL be omitted if the method was invoked on an object. If the method is invoked on an object and the value with the name "`0x00`" is included in the method parameterization, then the method SHALL fail.
    c. Table SHALL be omitted for an invocation of the `Get` method on a table. If the method is invoked on a byte or object table and the value with the name "`0x00`" is included in the method parameterization, then the method SHALL fail.
b. startRow – this Named value has the Name "`0x01`". This Named value type is assigned one of two values – either a uid of an object or a RowNumber that corresponds to the RowNumber value of a bytes table row. Only one of these two values SHALL appear in the messaging stream. The "typeOr" identifier and accompanying curly brackets ("{", "}") in the format description below have no effect on the values as represented in the message.
    a. If the value with Name "`0x01`" is omitted and the method is invoked on a byte table, then the operation defaults to the first row of that byte table.
    b. If the method is invoked on an object table, the value "`0x01`" SHALL be the uid of the object upon which the method is intended to operate. If the value with Name "`0x01`" is omitted in this case, then the method invocation SHALL fail.
        i. If the uid of the object does not belong to the table upon which the method was invoked, the method invocation SHALL fail.
    c. If the method is invoked on an object and the value with the name "`0x01`" is included in the method parameterization, then the method SHALL fail.
c. endRow – this Named value has the Name "`0x02`". This Named value type is a uinteger that corresponds to the RowNumber value of a byte table row.
    a. If the value with Name "`0x02`" is omitted and the method is invoked on a byte table, then the operation defaults to the last row of the table.
    b. If the method is invoked on an object or object table and the value with the name "`0x02`" is included in the method parameterization, then the method SHALL fail.
d. startColumn – this Named value has the Name "`0x03`". This Named value type has a uinteger value that indicates the column number of the cellblock's start column.

a. If the value with Name "0x03" is omitted, then the operation defaults to the first column of the table or object.

b. If the value with Name "0x03" is included in the method parameterization, and the method is invoked on a byte table, then the method SHALL fail.

e. endColumn – this Named value has the Name "0x04". This Named value type has a uinteger value that indicates the column number of the cellblock's end column.

a. if the value with Name "0x04" is omitted, then the operation defaults to the last column of the table or object.

b. If the value with Name "0x04" is included in the method parameterization, and the method is invoked on a byte table, then the method SHALL fail.

Format:

```
[ Table = uidref, startRow = typeOr { UID : uidref, Row : uinteger }, endRow =
uinteger, startColumn = uinteger, endColumn = uinteger ]
```

5.1.4.2.4 clock_kind

This type is similar to the column type of the same name, and represents the type of clock time that has been set, and is a return value of the GetClock method.

The possible values returned are as follows:

a. If the currently active clock kind is "Timer", the returned value is 0x00.

b. If the currently active clock kind is "Low", the returned value is 0x01.

c. If the currently active clock kind is "High", the returned value is 0x02.

d. If the currently active clock kind is "LowAndHigh", the returned value is 0x03.

Format:

```
uinteger
```

5.1.4.2.5 clock_time

This type represents a grouping of Named values that are used to identify time values, and is similar to the column type of the same name. In messaging, this grouping is enclosed by List value delimiters, and each component is enclosed by Named value delimiters.

The name of each component is a uinteger representing the positioning of that component within the grouping, as defined by this specification. The associated number also appears in the appropriate component description.

Because this is a group of Named values, its separate components are optional. Components that are omitted are considered to have a value of 0.

The components are as follows:

a. Year – this Named value has the Name "0x00" and a value that is implicitly defined as being of uinteger of size 2. This Named value abstract type represents the year in a timestamp. Valid values are unsigned integers ranging from 1970 to 9999

b. Month – this Named value has the Name "0x01" and a value that is implicitly defined as being of uinteger of size 2. This Named value abstract type represents the month in a timestamp. Valid values are unsigned integers ranging from 1 to 12, which correspond to the months of the year as follows:

a. January = 1 (0x01)

b. February = 2 (0x02)

c. March = 3 (0x03)
d. April = 4 (0x04)
e. MAY = 5 (0x05)
f. June = 6 (0x06)
g. July = 7 (0x06)
h. August = 8 (0x08)
i. September = 9 (0x09)
j. October = 10 (0x0A)
k. November = 11 (0x0B)
l. December = 12 (0x0C)

c. Day – this Named value has the Name "0x02" and a value that is implicitly defined as being of uinteger of size 1. This Named value abstract type represents the day of the month in a timestamp. Valid values are unsigned integers ranging from 1 to 31.

d. Hour – this Named value has the Name "0x03" and a value that is implicitly defined as being of uinteger size 1. This Named value abstract type represents the hour of the day in a timestamp. Valid values are unsigned integers ranging from 0 to 23.

e. Minute – this Named value has the Name "0x04" and a value that is implicitly defined as being of uinteger size 1. This Named value abstract type represents the minute of the hour in a timestamp. Valid values are unsigned integers ranging from 0 to 59.

f. Seconds – this Named value has the Name "0x05" and a value that is implicitly defined as being of uinteger size 1. This Named value abstract type represents the second of the minute in a timestamp. Valid values are unsigned integers ranging from 0 to 59.

g. Fraction – this Named value has the Name "0x06" and a value that is implicitly defined as being of uinteger size 2. This Named value abstract type represents fractions of a second in a timestamp, measured in milliseconds. Valid values are unsigned integers ranging from 0 to 999.

Format:

```
[ Year = uinteger, Month = uinteger, Day = uinteger, Hour = uinteger, Minute = uinteger, Second = uinteger, Fraction = uinteger ]
```

5.1.4.2.6 columns

This is a list of two lists of Named values, where the List value delimiters enclose the entire list and both subordinate lists, and the Named value delimiters enclose each component of each subordinate list.

The name of each component is a uinteger representing the positioning of that component within the grouping, as defined by this specification. The associated number also appears in the appropriate component description.

The Named values in both subordinate lists represent column names and their associated types. Each Name portion of the Named value SHALL be the host-supplied name of a column to be created in the new table, and the associated value is the uidref to the type to be assigned for that column.

The ordering of and within the subordinate lists determines the ordering of the columns and the unique column combination in the newly created table. The first subordinate list contains the columns whose combination of values is required to be unique within the table. The columns described within that list are ordered first. The name associated with this Named value type is "0x00".

The second subordinate list contains the rest of the columns of the table. The columns described within the second subordinate list are ordered according to their order in the list, all of which come after the

columns defined in the first subordinate list. The name associated with this Named value type is "0x01".

For Byte tables, the external grouping SHALL be empty. For tables with no host-assigned unique column combination, the first subordinate list SHALL be empty. For tables with no host assigned non-unique columns, the second list SHALL be empty. For tables with no host assigned columns, both lists SHALL be empty.

Format:

```
      [ IsUnique = [ ColumnName = uidref { TypeUID } ... ], IsColumn = [ ColumnName = uidref { TypeUID } ... ] ]
```

Byte table format pseudo-code example:

```
      [ ]
```

Object table with no unique column combination pseudo-code example:

```
      [ IsUnique = [ ]  IsColumn = [ ColumnName1 = uidref1 ColumnName2 = uidref2 ColumnName3 = uidref3 ] ]
```

5.1.4.2.7 date

This type represents a grouping of Named values that are used to identify time values, and is similar to the column type of the same name. In messaging, this grouping is enclosed by List value delimiters, and each component is enclosed by Named value delimiters.

The name of each component is a uinteger representing the positioning of that component within the grouping, as defined by this specification. The associated number also appears in the appropriate component description.

Because this is a group of Named values, its separate components are optional. Components that are omitted are considered to have a value of 0.

The components are as follows:

a. Year – this Named value has the Name "0x00" and a value that is implicitly defined as being of uinteger of size 2. This Named value abstract type represents the year in a timestamp. Valid values are unsigned integers ranging from 1970 to 9999

b. Month – this Named value has the Name "0x01" and a value that is implicitly defined as being of uinteger of size 2. This Named value abstract type represents the month in a timestamp. Valid values are unsigned integers ranging from 1 to 12, which correspond to the months of the year as follows:

    a. January = 1 (0x01)

    b. February = 2 (0x02)

    c. March = 3 (0x03)

    d. April = 4 (0x04)

    e. MAY = 5 (0x05)

    f. June = 6 (0x06)

    g. July = 7 (0x06)

    h. August = 8 (0x08)

    i. September = 9 (0x09)

    j. October = 10 (0x0A)

    k. November = 11 (0x0B)

l. December = 12 (0x0C)

c. Day – this Named value has the Name "`0x02`" and a value that is implicitly defined as being of uinteger of size 1. This Named value abstract type represents the day of the month in a timestamp. Valid values are unsigned integers ranging from 1 to 31.

Format:

```
[ Year = uinteger, Month = uinteger, Day = uinteger ]
```

5.1.4.2.8 hash_protocol

This abstract type is similar to an enumeration column type, and is used to identify a selected hash algorithm. This type has valid values in the range of integers from 0-15. These integers have the following values:

a. 0 = none

b. 1 = SHA 1

c. 2 = SHA 256

d. 3 = SHA 384

e. 4 = SHA 512

f. 5-15 = reserved

Format:

```
uinteger
```

In the messaging stream, these values SHALL be represented as follows:

a. 0x00 represents none

b. 0x01 represents SHA 1

c. 0x02 represents SHA 256

d. 0x03 represents SHA 384

e. 0x04 represents SHA 512

f. 0x05 – 0x0F are reserved.

5.1.4.2.9 key_size

This abstract type is used for the AdminExch parameter of the `IssueSP` method, and enables the host to select from supplying either a bytes_16 or a bytes_32 value to represent the size of the exchange key being submitted to the newly created SP.

Only one of these two values appears in the messaging stream. The "typeOr" identifier and accompanying curly brackets ("{", "}") have no effect on the values as represented in the message.

Format

```
typeOr { AES_128 : bytes_16, AES_256 : bytes_32 }
```

In the message stream itself, the value is one of the following:

a. `bytes_16`

b. `bytes_32`

5.1.4.2.10 lag

This type represents a grouping of two Named value pairs, used to describe seconds and milliseconds, and is similar to the column type of the same name. The components are encapsulated with the

interface type List value delimiters ("[", "]"). Each of the components is encapsulated with the Named value delimiters. The components are optional.

The name of each component is a uinteger representing the positioning of that component within the grouping, as defined by this specification. The associated number also appears in the appropriate component description.

The components are as follows:

a. Seconds – this component is a Named value pair with a Name of "0x00" and a value of uinteger. This value has an implicit size requirement of 2.

b. Milliseconds – this component is a Named value pair with a Name of "0x01" and a value of uinteger. This value has an implicit size requirement of 2.

Format:

```
[ Seconds = uinteger, Milliseconds = uinteger ]
```

5.1.4.2.11 name

This type is a representation of the max bytes type, and in most methods in which it is used it is assigned to parameters that are associated with a table's Name column or CommonName column. As such, it has an implicit size restriction of 32 bytes.

Format:

```
bytes
```

5.1.4.2.12 package

This abstract type is a grouping of Named value pairs that are used to describe the contents of a package retrieved from a TPer using the GetPackage method, or sent to the TPer with the SetPackage method. The components are encapsulated with interface type List value delimiters ("[", "]").

The name of each component is a uinteger representing the positioning of that component within the grouping, as defined in this specification. The associated number also appears in the appropriate component description.

The components are defined as follows:

a. Key - this component is a Named value pair with a Name of "0x00" and a value of bytes. It represents the key material from the invoking credential. If a WrappingKey was supplied to the GetPackage method, then this key material is encrypted using the WrappingKey credential. The WrappingKey credential MAY be a symmetric key or the public key of a public/private key pair. When retrieving the key material from credentials that store key information in multiple columns, any of those columns that are empty or uninitialized SHALL return as 0x00 for uinteger type columns and 0x00 for bytes type columns.

b. Purpose – this component is a Named value pair with a Name of "0x01" and a value of package_purpose. This is the value of the Purpose parameter of the GetPackage method invocation.

c. Date – this component is a Named value pair with a Name of "0x02" and a value of date. This is the value of the Date parameter of the GetPackage method invocation. This component is omitted if the Date parameter was not supplied to the GetPackage method invocation.

d. Log – this component is a Named value pair with a Name of "0x03" and a value of bytes. This is the value of the Log parameter of the GetPackage method invocation. This component is omitted if the Log parameter was not supplied to the GetPackage method invocation.

e. MAC – this component is a Named value pair with a Name of "0x04" and a value of bytes. This is the hash of the package contents (except this component) and is signed by the SigningKey credential identified in the GetPackage method invocation. The hash protocol used to create the hash is identified in the Hash column of the SigningKey credential. The value of this component is the signature of a private key if a public key credential is specified, or an HMAC if a symmetric key credential is specified.

Format:

```
[ Key = bytes, Purpose = package_purpose, Date = date, Log = bytes, MAC = bytes ]
```

5.1.4.2.13 package_purpose

This abstract type is similar to an enumeration column type, and is used to identify a selected purpose for the package in which it is being included. This type has valid values in the range of integers from 1-32. These integers have the following values:

a. 1 = Issuance
b. 2 = Key Wrapping
c. 3 = Backup
d. 4-32 = reserved

Format:

```
uinteger
```

In the messaging stream, these values SHALL be represented as follows:

a. 0x00 is reserved
b. 0x01 represents Issuance
c. 0x02 represents Key Wrapping
d. 0x03 represents Backup
e. 0x04 – 0x20 are reserved

5.1.4.2.14 row_address

This abstract type is used to describe a parameter that is either a uinteger that indicates the address within a bytes table, or a uidref of an object within an object table.

Only one of these two values appears in the messaging stream. The "typeOr" identifier and accompanying curly brackets ("{", "}") have no effect on the values as represented in the message.

Format

```
typeOr { RowAddress : uinteger, UIDAddress : uidref }
```

In the message stream itself, the value is one of the following:

a. `uinteger`
b. `uidref`

5.1.4.2.15 row_data

This type represents a list of lists of Named values. Each interior list represents a row, so there are multiple interior lists (a list of lists). The Named values represent column numbers and the values to be associated with them as defined by this specification or the column ordering requirements of the table when it was created. The value SHALL be of the type defined, as represented by the notation "<type of column>".

The number of interior lists (i.e. the number of rows that MAY be represented by this type "at one time") MAY be limited by SSC or implementation.

Format:

```
[ [ ColumnNumber = <type of column> ... ] ... ]
```

5.1.4.2.16 table_kind

This abstract type is similar to an enumeration column type, and is used to represent table types in the Table table. This type has valid values in the range of integers from 1-2. These integers have the following values:

a. 1 = Object

b. 2 = Byte

Format:

```
uinteger
```

In the messaging stream, these values are represented as follows:

a. 0x01 represents Object

b. 0x02 represents Byte

5.1.4.2.17 table_sizes

This abstract type defines a grouping of pairs of values that are table object uidrefs and the size associated with that particular table. The grouping is a list of uidrefs and uintegers. The set of values are encapsulated by List value delimiters ("[", "]"). Inside the delimiters is a series of one or more pairs of values. The first value in each pair is a uidref to a table descriptor object and the second value in each pair is a uinteger that describes the number of rows that MAY be additionally created for that table.

Format:

```
[ [ uidref {TableObjectUID}, uinteger ] ... ]
```

Pseudo-code example:

```
[ [ uidref1 uinteger1 ] [ uidref2 uinteger2 ] [ uidref3 uinteger3 ] ]
```

5.1.4.2.18 uidref

The uidref abstract type represents a uid of an object, table, or ThisSP that is expressed using a bytes type with a size of 8, and corresponds to an object or table's `UID` column value.

In the pseudo-code method signatures, the uidref abstract type is often followed by curly brackets ("{", "}") that are used to define the limitation of a valid value for that uidref. These valid values are typically represented as requiring an object of a specific type. Limitations expressed with curly brackets have no effect on the appearance of the associated uid value as it appears in the message stream.

Because this abstract type describes the inclusion of a uid, it represents a bytes value that has an implicit size restriction, and that value SHALL always be 8 bytes long.

Format:

```
bytes
```

5.1.5 Method Status Codes

Begin Informative Content

SP method calls invoke specific operations and receive associated status. The following sections identify and define the status codes that are returned by the TPer in response to method invocations and other operations. Table 166 identifies the value associated with each of these status codes.

End Informative Content

Table 166 Status Codes

Name	Value
SUCCESS	0x00
NOT_AUTHORIZED	0x01
OBSOLETE	0x02
SP_BUSY	0x03
SP_FAILED	0x04
SP_DISABLED	0x05
SP_FROZEN	0x06
NO_SESSIONS_AVAILABLE	0x07
UNIQUENESS_CONFLICT	0x08
INSUFFICIENT_SPACE	0x09
INSUFFICIENT_ROWS	0x0A
INVALID_PARAMETER	0x0C
OBSOLETE	0x0D
OBSOLETE	0x0E
TPER_MALFUNCTION	0x0F
TRANSACTION_FAILURE	0x10
RESPONSE_OVERFLOW	0x11
AUTHORITY_LOCKED_OUT	0x12
FAIL	0x3F

5.1.5.1 SUCCESS

This status SHALL be returned when a method is processed completely and without error by the TPer.

5.1.5.2 NOT_AUTHORIZED

This response is returned whenever an attempt is made to invoke a method for which the host does not have authorization.

Unless otherwise noted in a method's description, this status code SHALL be returned whenever there is no row in the `AccessControl` table to represent the InvokingID/MethodID combination, or when there is a row but the `ACL` for the InvokingID/MethodID combination has not been satisfied.

This status code SHALL be returned in response to the `GetACL` method if there is no `AccessControl` row that represents the InvokingID/MethodID combination as parameterized in the `GetACL` method, or when the combination is present in the `AccessControl` table but the `GetACLACL` has not been satisfied. This status code SHALL be returned if the `GetACL` method invocation is performed with an InvokingID other than that of the `AccessControl` table.

This status code SHALL be returned in response to the AddACE method if there is no AccessControl row that represents the InvokingID/MethodID combination as parameterized in the AddACE method, or when the combination is present in the AccessControl table but the AccACEACL has not been satisfied.

This status code SHALL be returned in response to the RemoveACE method if there is no AccessControl row that represents the InvokingID/MethodID combination as parameterized in the RemoveACE method, or when the combination is present in the AccessControl table but the RemoveACEACL has not been satisfied.

This status code SHALL be returned in response to the DeleteMethod method if there is no AccessControl row that represents the InvokingID/MethodID combination as parameterized in the DeleteMethod method, or when the combination is present in the AccessControl table but the DeleteMethodACL has not been satisfied.

This status code SHALL be returned as the status code of the SyncSession method if the authority referenced in the preceding StartSession method's HostSigningAuthority parameter has an Operation column value of Password, and the StartSession method's HostChallenge parameter value does not match the value required by the HostSigningAuthority parameter.

5.1.5.3 SP_BUSY

This status is returned as the status code of the SyncSession method if an attempt is made to open a Read-Write session to an SP when any other session to that SP is already open, or when an attempt is made to open a Read-Only session to an SP with which a Read-Write session is already open.

5.1.5.4 SP_FAILED

This status MAY be returned if an attempt is made to open a session to an SP that is in the Failed life cycle state (see 4.2).

5.1.5.5 SP_DISABLED

This status MAY be returned if a method is invoked from within a session to an SP that is in the Issued-Disabled state (see 4.2), and the method is not permitted because of the limitations placed on SP operation by the state behavior.

5.1.5.6 SP_FROZEN

This status SHALL be returned as the status of the SyncSession response when the host attempts to start a session to an SP that is in the Issued-Frozen or Issued-Disabled-Frozen state (see 4.2).

5.1.5.7 NO_SESSIONS_AVAILABLE

This status is returned if an attempt is made to open a session on a TPer on which the maximum number of concurrent sessions available for use are already being used.

5.1.5.8 UNIQUENESS_CONFLICT

This occurs when a conflict between objects is created due to the attempt to create a second object with a unique column combination that is already in use by another object. For instance, this status MAY be received when attempting to create a table, when a table already exists with the Name-CommonName-TemplateID combination submitted in the CreateTable invocation.

5.1.5.9 INSUFFICIENT_SPACE

This status is returned if an attempt is made to:

a. Create an SP and there is insufficient space on the TPer to create the new SP
b. Create a table and there is insufficient space in the SP to create the new table
c. Create more rows in a table than is permitted by the TPer or by the table's size settings.

Note that it is possible that re-invoking the method and requesting a smaller size for the SP or table MAY enable the method to then complete properly.

5.1.5.10 INSUFFICIENT_ROWS

This status MAY be returned if an attempt is made to create a table or object, but the associated metadata or support table rows (i.e., the `Table`, `Column`, `AccessControl`, or `ACE` tables) are not able to be created to support the new object or table.

5.1.5.11 INVALID_PARAMETER

This status is returned if a method invocation has any invalid parameters or parameter values, and is applicable to any parameter inside the invoked method's parameter list, unless otherwise indicated or another status code is directly applicable to the method failure.

There are many situations in which this error could be returned. Some of the specific situations where this could occur are:

a. Columns specified in the `CreateRow` method invocation are not part of the table definition.
b. If an attempt is made to set a cell to a value larger (or smaller) than that cell's type allows, or attempts to set a value of a type different than that of the column.
c. If an incorrect credential type is parameterized.
d. A parameterized value is of the incorrect type for that method.
e. One of the context-related restrictions defined for an abstract type is violated (see 5.1.4.2).
f. A parameterized value is larger or smaller than the value required by that method invocation
    a. An example of this MAY occur if the TransTimeout parameter value submitted in a StartSession method invocation is larger than the TPer's MaxTransTimeout property.

This status code SHALL be sent as the `SyncSession` method status code if the preceding `StartSession` method's HostSigningAuthority parameter is a class authority.

5.1.5.12 TPER_MALFUNCTION

This status is returned when some operational failure has occurred within the TPer that has caused the method invocation to fail.

5.1.5.13 TRANSACTION_FAILURE

This status is returned when a method fails due to an error in the transactional context in which it was invoked. An example of this is if a TPer is unable to process within the transaction the amount of data supplied as a parameter of the method, which under other circumstances the TPer would be able to process. The TPer in this case would return this status code to indicate that the method failed due to the transactional context, not due to a problem with the method invocation itself.

Multiple consecutive method invocations that result in this status code indicate a failure in the transactional context that MAY result in the entire transaction being uncommittable.

5.1.5.14 RESPONSE_OVERFLOW

This status is returned when a method fails if the method response and associated protocol overhead do not fit entirely within the response buffer.

5.1.5.15 AUTHORITY_LOCKED_OUT

This status MAY be returned as the status code of the `SyncSession` method or in response to the `Authenticate` method under one of the following conditions:

1) If an authority with the `Operation` column value of Password is being authenticated and its associated `C_PIN` object has a `Tries` column value equal to its `TryLimit` column value, and the `TryLimit` column is not set to 0; or
2) If the `Uses` column of the authority being authenticated has reached the value of its `Limit` column, and the `Uses` column is not set to 0.

5.1.5.16 FAIL

This status is returned when a method fails in a manner for which none of the other failure statuses apply.

5.2 Session Manager Methods

5.2.1 Overview

Begin Informative Content

Session Manager protocol layer methods permit a host to retrieve information about a TPer without having to start a session and provide the methods required to enable session startup.

Due to the nature of the Session Manager protocol layer methods, the responses to methods at this protocol layer are formatted as methods from the TPer to the host. In the case of multiple method invocations by a host to a TPer on the Session Manager layer, this mechanism allows the host to identify the method to which a response is directed.

End Informative Content

Session Manager methods SHALL be invoked using an InvokingID of SMUID, which is the reserved UID `0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xFF`.

The UIDs used to invoke Session Manager Methods are defined in Table 241.

5.2.2 TPer Properties Method

5.2.2.1 Properties (Method)

The `Properties` method is a control session method used by the host to provide its communication properties to the TPer, and to retrieve the communication properties of the TPer. The purpose of the `Properties` method is to permit the host and the TPer to exchange the information about their respective communications capabilities required for session startup and maintenance, without the need to first start a session.

Properties are maintained on a per-ComID basis in both the host and the TPer. The HostProperties parameter is used to describe the communications capabilities that the host possesses, and apply to any sessions started using the ComID associated with this `Properties` method invocation once the TPer has processed the method and prepared a response.

```
SMUID.Properties[ HostProperties = list [ name = value ... ] ]
=>
SMUID.Properties[ Properties : list [ name = value ... ], HostProperties = list [
name = value ... ] ]
```

5.2.2.1.1 HostProperties

This parameter is a list of name/value pairs that MAY be submitted when invoking the `Properties` method. This is a list of the communications capabilities that the host is able to support on communications it receives.

5.2.2.1.2 Properties Response

Because of the session-less nature of the Session Manager protocol layer, and the possible different ordering of responses to Session Manager layer methods, the response to a `Properties` method invocation is itself formatted as a `Properties` method invocation so as to be identifiable as the response to the `Properties` method.

5.2.2.1.2.1 Properties

This is a list of property names and values that represent the communications capabilities of the TPer.

5.2.2.1.2.2 HostProperties

If the host includes the HostProperties parameter to the `Properties` method invocation, then this portion of the method result SHALL include the communications limitations and capabilities that the TPer SHALL use for messages sent from the TPer to the Host.

5.2.2.2 Retrieving Properties

The TPer SHALL return all property name/value pairs for capabilities that it supports. For capabilities not supported by the TPer (for instance, Read-Only sessions), the associated property name/value pair (in this case, MaxReadSessions) SHALL be omitted from the TPer's response.

The TPer MAY also respond with additional name/value pairs other than those specified in this document.

The order of the name/value pairs returned by the TPer is not specified.

For the name/value pairs returned by the TPer, the TPer SHALL return values for the associated names as described in Table 167 or in the associated SSC (the values in the SSC have precedence) for all capabilities supported. The values returned SHALL apply to all sessions started with the currently associated ComID.

Table 167 Properties Method Response

Property	Type	Description	Applicable To
MaxMethods	uinteger	The maximum number of method invocations per Subpacket that the communicator can accept. If the TPer supports the Asynchronous Communication Protocol (the TPer's Asynchronous property is TRUE), then the TPer's MaxMethods SHALL be 0 (no limit). If the Host supports the Asynchronous Communication Protocol (the Host sets its Asynchronous property to TRUE), then the host SHOULD also set its MaxMethods property to 0 (no limit). The TPer SHALL ignore the Host's MaxMethods property if both the Host and the TPer have the Asynchronous property set to TRUE.	Host Property and TPer Property
MaxSubpackets	uinteger	Identifies the maximum number of subpackets that the communicator SHALL accept in a single Packet. A value of 0 indicates no limit.	Host Property and TPer Property
MaxPacketSize	uinteger	The maximum size of a packet (including both data and header), in bytes, that the communicator is able to receive. This value SHALL be at least 1004 (1024 - (ComPacket Header Size)). A value of 0 indicates no limit.	Host Property and TPer Property
MaxPackets	uinteger	Identifies the maximum number of packets that the communicator is able to accept in a single ComPacket. A value of 0 indicates no limit.	Host Property and TPer Property

Property	Type	Description	Applicable To
MaxComPacketSize	uinteger	The maximum size of an IF Command payload in bytes (includes both the ComPacket header and payload) that the communicator is able to receive. This value SHALL be at least 1024. A value of 0 indicates no limit.	Host Property and TPer Property
MaxResponseComPacketSize	uinteger	The maximum length of an IF Command payload, in bytes, that the communicator is able to generate. A value of 0 indicates no limit.	Host Property and TPer Property
MaxSessions	uinteger	The maximum number of simultaneous sessions supported by the TPer across all ComIDs. A value of 0 indicates no limit.	TPer Property
MaxReadSessions	uinteger	The maximum number of simultaneous Read-Only sessions to any one SP supported by the TPer. A value of 0 indicates no limit.	TPer Property
MaxIndTokenSize	uinteger	The maximum size of a token (in bytes) in a single subpacket that the communicator is able to accept. Token size refers to both the token header and data. This value SHALL be at least 968. A value of 0 indicates no limit.	Host Property and TPer Property
MaxAggTokenSize	uinteger	The maximum aggregate size of a continued token, after all individual parts of that token are combined, that the communicator is able to accept. Token size refers to both the token header and data. This value SHALL be at least 968. A value of 0 indicates no limit.	Host Property and TPer Property
MaxAuthentications	uinteger	The maximum number of simultaneously authenticated individual authorities per session that the TPer is able to support. A value of 0 indicates no limit.	TPer Property
MaxTransactionLimit	uinteger	The maximum number of concurrently open transactions that the TPer is able to support in a single session. A value of 0 indicates no limit.	TPer Property
DefSessionTimeout	uinteger	The session timeout length (in milliseconds) used by the TPer by default. A value of 0 indicates no limit.	TPer Property
MaxSessionTimeout	uinteger	The longest supported session timeout length (in milliseconds) supported by the TPer. A value of 0 indicates no limit.	TPer Property
MinSessionTimeout	uinteger	The shortest supported session timeout length (in milliseconds) supported by the TPer. A value of 0 indicates session timeouts are not supported.	TPer Property

Property	Type	Description	Applicable To
DefTransTimeout	uinteger	The transmission timeout length (in milliseconds) used by the TPer by default. A value of 0 indicates no limit.	TPer Property
MaxTransTimeout	uinteger	The longest transmission timeout length (in milliseconds) permitted by the TPer. A value of 0 indicates no limit.	TPer Property
MinTransTimeout	uinteger	The shortest transmission timeout length (in milliseconds) permitted by the TPer. A value of 0 indicates transmission timeouts are not supported.	TPer Property
MaxComIDTime	uinteger	The timeout length (in milliseconds) used by the TPer after it has assigned a ComID. The ComID SHALL transition to Inactive after this much time has elapsed. A value of 0 indicates no limit.	TPer Property
ContinuedTokens	boolean	**TRUE**: The communicator supports continuted tokens. **FALSE**: The communicator does not support continued tokens.	Host Property and TPer Property
SequenceNumbers	boolean	**TRUE**: The communicator supports Packet sequence numbers. **FALSE**: The communicator does not support Packet sequence numbers.	Host Property and TPer Property
AckNak	boolean	**TRUE**: The communicator supports the Packet ACK/NAK protocol. **FALSE**: The communicator does not support the Packet ACK/NAK protocol.	Host Property and TPer Property
Asynchronous	boolean	**TRUE**: The communicator supports the Asynchronous Communication Protocol. **FALSE**: The communicator does not support theAsynchronous Communication Protocol.	**Host Property and TPer Property**

5.2.2.3 Setting HostProperties

If the method is invoked with the optional HostProperties parameter, the list of name/value pairs that the TPer MAY support is the list of properties inTable 168.

These values MAY be submitted in any order by the host. Not all values are required to be submitted. Subsequent submission of these values (in a subsequent invocation of the `Properties` method) SHALL supersede values submitted to previous invocations of the `Properties` method for that ComID. Submitted values, if applicable, SHALL only apply to sessions started after the submission of those values, and not to sessions that are already open on that ComID.

The TPer uses these host properties when it is constructing responses to be transmitted to the host. The host MAY omit properties as necessary, depending on the host's communications capabilities. If the host omits a property, the value of that property SHALL NOT change from its current value. If the host specifies a value for a property that does not meet the minimum requirement as defined in Table 168, then the TPer SHALL use the minimum value defined in Table 168 in place of the value supplied by the host.

These values reflect the cumulative modifications of all processed `Properties` methods for the associated ComID.

If the host sends the HostProperties parameter in its `Properties` invocation, the TPer SHALL respond with ALL host properties it supports, with their current values, in the HostProperties parameter of its method response.

If a host includes property parameters to the `Properties` method invocation that the TPer does not support, the TPer SHALL ignore those parameters, and SHALL NOT return them in its response.

It is the host's responsibility to insure that `Properties` method invocations have processed prior to invocation of any session startup methods that rely on those invocations. Values for HostProperties at session startup rely on the `Properties` method invocations that have been processed by the TPer.

5.2.2.4 Communications Minimums

When a ComID is first allocated, the TPer assumes some minimum communications capabilities of the host until it receives a successful `Properties` method from the host. Similarly, the host assumes some minimum communications capability of the TPer until the host successfully receives the results of the `Properties` method from the TPer.

Invocation of the `Properties` method is optional. Communications MAY occur using just the minimum communications capability.

Table 168 shows all the properties that affect the behavior of hosts and TPers when sending ComPackets/Subpackets/Packets to the other. These are the properties that need to be considered for minimum communications capability between the two. The host's initial assumption about the TPer and the TPer's initial assumption about the host are listed for each of the properties.

Table 168 Communications Initial Assumptions

Property Name	Initial Host Assumption About TPer	Initial TPer Assumption About Host
MaxSubpackets	1	1
MaxPacketSize	1024 – (ComPacket Header size) = 1004	1024 – (ComPacket Header size) = 1004
MaxPackets	1	1
MaxComPacketSize	1024	1024
MaxIndTokenSize	1024 - (ComPacket Header Size) - (Packet Header Size) - (Subpacket Header Size) = 1024 - 20 - 24 - 12 = 968	1024 - (ComPacket Header Size) - (Packet Header Size) - (Subpacket Header Size) = 1024 - 20 - 24 - 12 = 968
MaxAggTokenSize	1024 - (ComPacket Header Size) - (Packet Header Size) - (Subpacket Header Size) = 1024 - 20 - 24 - 12 = 968	1024 - (ComPacket Header Size) - (Packet Header Size) - (Subpacket Header Size) = 1024 - 20 - 24 - 12 = 968
MaxMethods	1	1
ContinuedTokens	False	False
SequenceNumbers	False	False
AckNAK	False	False
Asynchronous	False	False

The values listed in Table 168 are the minimum values that TPers and hosts SHALL support. SSCs MAY impose minimums that are greater than the values listed above.

SSCs MAY redefine the initial assumptions that the host and TPer make about each other. In such cases, the initial assumptions the host makes about the TPer are based on the supported SSC the host discovers during Level 0 Discovery.

The host invokes `Properties` with the optional HostProperties parameter to inform the TPer of its capabilities. The result from the TPer tells the host the TPer's capabilities. All communication from that

point can use the mutually discovered capabilities. However, the initial invocation of `Properties` SHALL be encoded using the minimum assumptions outlined above in Table 168, or the minimum assumptions defined by the TPer's SSC. The TPer MAY format its response to the `Properties` invocation using the host's capabilities it received in the HostProperties parameter.

If the host attempts to set a property in the HostProperties parameter that is less than the initial assumed value, the TPer SHALL ignore the property value and the initial assumed value SHALL be used. In the Properties method response, the TPer SHALL report the initial assumed value.

If a particular property is not returned by the TPer, then the host SHALL NOT change its assumption about the TPer's capabilities related to that property. Likewise, if the host does not send a particular property in the HostProperties parameter, the TPer SHALL NOT change its assumption about the host's capabilities related to that property.

5.2.2.4.1 Communication Rules Based on TPer Properties and Host Properties

This section defines the rules for communication based on the TPer Properties and the Host Properties. These rules SHALL be enforced on a per-ComID basis, as hosts on different ComIDs may set different host properties.

Begin Informative Content

When communicating on statically allocated ComIDs, it is possible for the TPer's knowledge of the HostProperties to be reset without the host's knowledge (e.g. due to a TCG Hardware reset or a TCG Power Cycle reset). In this case, the TPer's knowledge of the host's communication properties will be reset to the initial assumed values shown in Table 168. This could adversely affect the performance of sessions that the host opens on the statically allocated ComID after the reset occurs. To prevent such performance degredation, it is the host's responsibility to invoke `Properties` with the HostProperties parameter prior to each invocation of `StartSession` on statically allocated ComIDs.

This problem does not occur when using dynamically allocated ComIDs, because dynamically allocated ComIDs become inactive when the TPer is reset. The host receives an indication that the ComID is inactive if it attempts further communication on that ComID. Therefore, the host needs to invoke `Properties` with the HostProperties parameter only once per dynamically allocated ComID.

End Informative Content

5.2.2.4.1.1 MaxSubpackets

The host SHOULD NOT send a Packet that contains more Subpackets than the value of the TPer's MaxSubpackets property. If the host sends a Packet that contains too many Subpackets, the TPer SHALL abort the session associated with the Packet. In the case of too many Subpackets in a Control Session Packet, the TPer SHALL discard and ignore the Packet.

The TPer SHALL NOT send a Packet that contains more Subpackets than the value of the host's MaxSubpackets property.

5.2.2.4.1.2 MaxPacketSize

The host SHOULD NOT send a Packet whose size (including Packet header) exceeds the value of the TPer's MaxPacketSize property. If the host sends a Packet that is too large, the TPer SHALL abort the session associated with the Packet. In the case of a Packet that is too large on the Control Session, the TPer SHALL discard and ignore the Packet.

The TPer SHALL NOT send a Packet whose size (including Packet header) exceeds the value of the host's MaxPacketSize property.

5.2.2.4.1.3 MaxPackets

The host SHOULD NOT send a ComPacket that contains more Packets than the value of the TPer's MaxPackets property. If the host sends a ComPacket that contains too many Packets, the TPer MAY ignore the extra Packets.

The TPer SHALL NOT send a ComPacket that contains more Packets than the value of the host's MaxPackets property.

5.2.2.4.1.4 MaxComPacketSize

The host SHOULD NOT send a ComPacket whose size (including ComPacket header) exceeds the value of the TPer's MaxComPacketSize property. If the host attempts to send a ComPacket that is too large, the TPer SHALL abort the IF-SEND command as described in the "Invalid Transfer Length parameter on IF-SEND" section of the appropriate interface section of [2].

The TPer SHALL NOT send a ComPacket whose size (including ComPacket header) exceeds the value of the host's MaxComPacketSize property.

5.2.2.4.1.5 MaxIndTokenSize

The host SHOULD NOT send an individual token whose size (including token header) is greater than the TPer's MaxIndTokenSize property. If the TPer encounters a token that is too long, the TPer's response is defined in section 5.2.2.4.3.

The TPer SHALL NOT send an individual token whose size (including token header) is greater than the host's MaxIndTokenSize property.

5.2.2.4.1.6 MaxAggTokenSize

The host SHOULD NOT send an aggregate token whose size (including token header) is greater than the TPer's MaxAggTokenSize property. If the TPer encounters a token that is too long, the TPer's response is defined in section 5.2.2.4.3.

The TPer SHALL NOT send an aggregate token whose size (including token header) is greater than the host's MaxAggTokenSize property.

5.2.2.4.1.7 MaxMethods

The host SHOULD NOT send a Data Subpacket that contains more method invocations that the value of the TPer's MaxMethods property. If the host sends a Data Subpacket that contains too many method invocations, the TPer MAY abort the session associated with the Packet. Results for methods that were completed before the violating method invocation was encountered SHALL be sent to the host. In the case of too many method invocations in a Data Subpacket of a Control Session Packet, the TPer MAY ignore the extra method invocations.

The TPer SHALL NOT send a Data Subpacket that contains more method responses than the value of the Host's MaxMethods property, unless both the Host and the TPer have the Asynchronous property set to TRUE. If both the Host and the TPer have the Asynchronous property set to TRUE, the TPer SHALL ignore the Host's MaxMethods property. Note that the TPer only sends method invocations on the Control Session.

5.2.2.4.1.8 ContinuedTokens

If the TPer's ContinuedTokens property is TRUE, the host MAY send continued tokens to the TPer. Otherwise, the host SHOULD NOT send continued tokens to the TPer. If the TPer encounters a continued token when its ContinuedTokens property is FALSE, the TPer's response is defined in section3.2.2.4.

If the host's ContinuedTokens property is TRUE, the TPer MAY send continued tokens to the host. Otherwise, the TPer SHALL NOT send continued tokens to the host.

If a communicator's ContinuedTokens property is FALSE, then that communicator's MaxAggTokenSize property value SHALL be ignored by the other communicator.

5.2.2.4.1.9 SequenceNumbers

If both the host's SequenceNumbers property and the TPer's SequenceNumbers property are TRUE:

a. The TPer SHALL generate sequence numbers for Packets sent to the host.
b. The TPer SHALL check the sequence numbers of Packets received from the host.
c. The host SHOULD generate sequence numbers for Packets sent to the TPer.
d. The host SHOULD check the sequence numbers of Packets received from the TPer.

If either the host's SequenceNumbers property or the TPer's SequenceNumbers property are FALSE:

a. The TPer SHALL put a value of `0x00000000` in the SeqNumber field for Packets sent to the host.
b. The TPer SHALL ignore the sequence numbers of Packets received from the host.
c. The host SHOULD put a value of `0x00000000` in the SeqNumber field for Packets sent to the TPer.
d. The host SHOULD ignore the sequence numbers of Packets received from the TPer.

Sequence Numbers SHALL be supported for transmission acknowledgement, MAY be supported for secure messaging, and MAY be supported otherwise

5.2.2.4.1.10 AckNak

If both the host's AckNak property and the TPer's AckNak property are TRUE:

a. The TPer SHALL use the Transmission Acknowledgement protocol.
b. The host SHOULD use the Transmission Acknowledgement protocol. If it does not, the TPer will not be able to discard the packets it has sent to the host, causing a transmit buffer overflow and a session abort.

If either the host's AckNak property or the TPer's AckNak property are FALSE:

a. The TPer SHALL NOT use the Transmission Acknowledgement protocol, and SHALL ignore the AckType and Acknowledgement fields on all packets received from the host. The TPer SHALL put a value of `0x0000` in the AckType field and a value of `0x00000000` in the Acknowledgement field for Packets sent to the host.
b. The host SHOULD NOT use the Transmission Acknowledgement protocol, and SHOULD ignore the AckType and Acknowledgement fields on all packets received from the TPer. The host SHOULD put a value of `0x0000` in the AckType field and a value of `0x00000000` in the Acknowledgement field for Packets sent to the TPer.

5.2.2.4.1.11 Asynchronous

If both the host's Asynchronous property and the TPer's Asynchronous property are TRUE:

a. The TPer SHALL use the Asynchronous Communication protocol.
b. The TPer SHALL generate Credit Control Subpackets for informing the host how much data can be sent to the TPer.
c. The TPer SHALL accept Credit Control Subpackets from the host, and SHALL only send as much data as it has credit to send.
d. The host SHOULD use the Asynchrounous Communication protocol.
e. The host SHOULD generate Credit Control Subpackets for informing the TPer how much data can be sent to the host. If it does not, the TPer will eventually stop sending data to the host after it uses all of the initial credit it was granted during session start up.
f. The host SHOULD accept Credit Control Subpackets from the TPer, and SHOULD only send as much data as it has credit to send. If it sends more data than it has credit to send, the TPer MAY abort the session.

If either the host's Asynchronous property or the TPer's Asynchronous property are FALSE:

a. The TPer SHALL use the Synchronous Communication protocol.
b. The TPer SHALL NOT generate Credit Control Subpackets.
c. The TPer SHALL ignore Credit Control Subpackets from the host.
d. The host SHOULD use the Synchronous Communication protocol. If it does not, it will likely cause Synchronous Protocol Violations on the TPer, and possible session aborts.
e. The host SHOULD NOT generate Credit Control Subpackets.
f. The host SHOULD ignore Credit Control Subpackets from the TPer.

5.2.2.4.2 AckNak and SequenceNumbers Dependency

If the TPer's AckNak property is TRUE, then its SequenceNumbers property SHALL also be true.

If the host invokes `Properties` with the HostProperties parameter, and sets its AckNak property to TRUE but its SequenceNumbers property to FALSE, and if the TPer supports those host properties, then the TPer SHALL treat both the host's AckNak and SequenceNumbers properties as FALSE, and shall return FALSE for both properties in the HostProperties parameter of the `Properties` method response. If the TPer does not support those host properties, it SHALL ignore them, and SHALL NOT list them in the HostProperties parameter of the `Properties` method response.

5.2.2.4.3 TPer Response for Invalid or Unexpected Token

See 3.2.2.4 for the TPer's response if an invalid or unexpected token is received in the message stream.

5.2.2.4.4 Interaction with TCG Reset Events

TCG Hardware Resets and TCG Power Cycle Resets SHALL cause the TPer's knowledge of the host's communications capabilities, on all ComIDs, to be reset to the initial minimum assumptions defined in this document or in the TPer's SSC definition. Other TCG reset events SHALL NOT cause the TPer's knowledge of the host's communications capabilities to be reset.

5.2.2.4.5 Interaction with TCG Protocol Stack Reset

Receiving a TCG Protocol Stack Reset SHALL cause the TPer's knowledge of the host's communications capabilities, on the ComID receiving the reset command, to be reset to the initial minimum assumptions defined in this document or in the TPer's SSC definition.

5.2.3 Session Startup Methods

Begin Informative Content

This section describes the methods used to start a session. For information on session startup, and how authorities interact during session startup, see section 5.3.4.1.4.

For details on using the session startup methods with Elliptic Curve parameters and EC-MQV or EC-DH, see section 5.3.4.1.11 and 5.3.4.1.12 respectively.

End Informative Content

5.2.3.1 StartSession Method

```
SMUID.StartSession [
HostSessionID : uinteger,
SPID : uidref {SPObjectUID},
Write : boolean,
HostChallenge = bytes,
HostExchangeAuthority = uidref {AuthorityObjectUID},
HostExchangeCert = bytes,
HostSigningAuthority = uidref {AuthorityObjectUID},
HostSigningCert = bytes,
SessionTimeout = uinteger,
TransTimeout = uinteger,
InitialCredit = uinteger,
SignedHash = bytes ]
=>
SMUID.SyncSession [ see SyncSession definition in 5.2.3.2]
```

5.2.3.1.1 HostSessionID

The **HostSessionID** parameter in the `StartSession` invocation is the host-side session number assigned and used by the host to identify this session. All further invocations in this series of method invocations and responses use this host-assigned session number in the HostSessionID parameter. This is the number that becomes the HSN portion of the packet header Session field (see 3.2.3.3 and 3.3.7.1).

5.2.3.1.2 SPID

The **SPID** parameter in the `StartSession` invocation is the uid of the SP with which the host is attempting to start a session. This is the uid of the SP's object in the Admin SP's `SP` table.

5.2.3.1.3 Write

The **Write** parameter determines the type of session that is being started. This value SHALL be True when a Read-Write session is requested and False when a Read-Only session is requested.

5.2.3.1.4 HostChallenge

If the Signing Authority (identified in the **HostSigningAuthority** parameter) has an Operation column value of Password in the Authority table and references a C_PIN credential, then the **HostChallenge** parameter is used by the host to submit a password for authentication. Otherwise, this parameter is used to submit a nonce to the SP that, during secure session startup, returns a response based on the HostChallenge value and the authentication requirements of the Signing Authority.

5.2.3.1.5 HostExchangeAuthority

The **HostExchangeAuthority** identifies the authority whose credential is used to exchange keys with the SP.

5.2.3.1.6 HostExchangeCert

The **HostExchangeCert** parameter provides the certificate associated with the credential to be used with the HostExchangeAuthority.

5.2.3.1.7 HostSigningAuthority

For challenge/response authentication, the **HostSigningAuthority**'s credential is used to formulate the response to the SP's challenge. The HostSigningAuthority parameter identifies the authority whose credential is used to sign the method hash (sent in the SignedHash parameter), and to sign the SPChallenge value sent in the `SyncSession` method invocation.. For password authentication, the HostSigningAuthority's credential is used to verify the password sent in the HostChallenge parameter.

5.2.3.1.8 HostSigningCert

The optional **HostSigningCert** parameter provides attestation to the HostSigningAuthority's credential.

5.2.3.1.9 SessionTimeout

The **SessionTimeout** parameter is used to allow the host to provide a requested timeout value for the session.

The value, in milliseconds, SHOULD be less than the TPer's MaxSessionTimeout property, greater than the TPer's MinSessionTimeout property (see 5.2.2.1), and less than the value of the `SPSessionTimeout` column in the SP's `SPInfo` table. If the parameter value is outside of these limits, the method invocation SHALL fail.

5.2.3.1.10 TransTimeout

The **TransTimeout** parameter is used to allow the host to provide a requested timeout value for acknowledgement.

The value, in milliseconds, SHOULD be less than the TPer's MaxTransTimeout property and greater than the TPer's MinTransTimeout property (these values are reported as the results of the `Properties` method (see 5.2.2.1). If the parameter value is outside of these limits, the method invocation SHALL fail.

If this capability is supported and no value is specified for this parameter, then the TPer's default value (identified as the DefaultTransTimeout response to the `Properties` mehod), SHALL be used as the transmission timeout value. For more information on the transmission timeout mechanism, see 3.3.9.4.

5.2.3.1.11 InitialCredit

The **InitialCredit** parameter enables the host to provide an amount of credits to the TPer for use in data exchange once the session has been successfully opened. For more information on the buffer management/flow control mechanism, see 3.3.8.2.

5.2.3.1.12 SignedHash

The optional **SignedHash** parameter of each session startup method is present if hashing is required by the Control Authority for that communicator (see 5.3.4.1.4). This is a signed hash of all the other parameters to the method, other than the SignedHash parameter. The purpose of this is to provide integrity during session startup, prior to the point when secure messaging takes effect.

The Host Control Authority identifies the hash type and signing type if hashing has been called out on messages from the host to the SP (see 5.3.4.1.7).

5.2.3.2 SyncSession Method

The `SyncSession` is returned by the TPer in response to invocation of the `StartSession` method by the host.

```
SMUID.StartSession [ see StartSession definition in 5.2.3.1]
=>
SMUID.SyncSession [
HostSessionID : uinteger,
SPSessionID : uinteger,
SPChallenge = bytes,
SPExchangeCert = bytes,
SPSigningCert = bytes,
TransTimeout = uinteger,
InitialCredit = uinteger,
SignedHash = bytes ]
```

5.2.3.2.1 HostSessionID

The **HostSessionID** parameter in the `SyncSession` invocation SHALL be the same as that in the `StartSession` invocation.

5.2.3.2.2 SPSessionID

The **SPSessionID** parameter in the `SyncSession` invocation is the TPer side session number, which is assigned by the TPer. All further invocations in this series of method invocations and responses use this TPer-assigned session number in the SPSessionID parameter.

This is the number that becomes the TSN portion of the packet header Session field (see 3.2.3.3 and 3.3.7.1).

5.2.3.2.3 SPChallenge

The **SPChallenge** parameter value is sent if the `StartSession` invocation includes a HostSigningAuthority that directly invokes a signing credential. Otherwise, this parameter is omitted.

5.2.3.2.4 SPExchangeCert

The **SPExchangeCert** is the certificate for the credential referenced by the SP exchange authority that MAY be referenced by the parameterized HostSigningAuthority specified in the `StartSession` invocation.

5.2.3.2.5 SPSigningCert

The optional **SPSigningCert** is the certificate for the credential referenced by the SP signing authority that MAY be referenced by the parameterized HostSigningAuthority specified in the `StartSession` invocation.

5.2.3.2.6 TransTimeout

The **TransTimeout** parameter in the `SyncSession` method is used by the TPer to report the Timeout value it SHALL use. This parameter is used to allow the TPer to provide a transmission timeout value for acknowledgement larger than that requested by the host.

This optional parameter SHALL be greater than or equal to the value of the TransTimeout parameter of the `StartSession` method, unless the TransTimeout parameter of `StartSession` contained a value that was greater than the TPer's MaxTransTimeout property, in which case the `SyncSession` method SHALL indicate a failure result.

The TransTimeout parameter value (measured in milliseconds) SHALL be less than the TPer's MaxTransTimeout property and greater than the TPer's MinTransTimeout property (see 5.2.2.1).

If this capability is supported and no value is specified for this parameter in either the `StartSession` or `SyncSession` methods, then the TPer's default value (identified as the DefTransTimeout response to the `Properties` mehod), SHALL be used as the transmission timeout value. For more information on the transmission timeout mechanism, see 3.3.9.4.

5.2.3.2.7 InitialCredit

The **InitialCredit** parameter enables the TPer to provide an amount of credits to the host for use in data exchange once the session has been successfully opened. For more information on the buffer management/flow control mechanism, see 3.3.8.2.

5.2.3.2.8 SignedHash

The **SignedHash** of the `SyncSession` method, if present, is the hash of the method's parameter's signed by the response signing credential that is the credential referred to by the SPSigningAuthority.

The SP Control Authority, if referenced by the Host Control Authority, identifies the hash type and signing type if hashing has been called out on messages from the SP to the host (see 5.3.4.1.7).

5.2.3.3 StartTrustedSession Method

The `StartTrustedSession/SyncTrustedSession` method exchange, if needed, SHALL occur after the `StartSession/SyncSession` method exchange. If invoked at any other time, the attempted method invocation SHALL return an error result.

```
SMUID.StartTrustedSession [
HostSessionID : uinteger,
SPSessionID : uinteger,
HostResponse = bytes,
HostEncryptSessionKey = bytes,
HostIntegritySessionKey = bytes,
SignedHash = bytes ]
=>
SMUID.SyncTrustedSession [See SyncTrustedSession definition in 5.2.3.4]
```

5.2.3.3.1 HostSessionID

The **HostSessionID** parameter in the `StartTrustedSession` invocation SHALL be the same as that in the `StartSession` invocation.

5.2.3.3.2 SPSessionID

The **SPSessionID** parameter in the `StartTrustedSession` invocation is the TPer side session number, which was assigned by the TPer and delivered to the host in the `SyncSession` method.

5.2.3.3.3 HostResponse

The **HostResponse** is included if the `SyncSession` method contained an SPChallenge parameter. The value of the HostResponse parameter is dictated by the credential of the HostSigningAuthority.

5.2.3.3.4 HostEncryptSessionKey

The **HostEncryptSessionKey** is the session keyset generated by the host and encrypted with the key used for exchange with the SP (see Session Startup (section 5.3.4.1.4) for more information). This session keyset is used in secure messaging to encrypt packets sent from the host to the SP.

5.2.3.3.5 HostIntegritySessionKey

The **HostIntegritySessionKey** is the session keyset generated by the host and encrypted with the key used for exchange with the SP. This session keyset is used to create a MAC of the data sent from the host to the SP (if required), to aid in integrity assurance.

5.2.3.3.6 SignedHash

The optional **SignedHash** parameter of each session startup method is present if hashing is required by the Control Authority for that communicator (see 5.3.4.1.4). This is a signed hash of all the other parameters to the method, other than the SignedHash parameter. The purpose of this is to provide integrity during session startup, prior to the point when secure messaging begins.

The Host Control Authority identifies the hash type and signing type if hashing has been called out on messages from the host to the SP (see 5.3.4.1.7).

5.2.3.4 SyncTrustedSession Method

The `SyncTrustedSession` method is returned by the TPer in response to invocation of the `StartTrustedSession` method by the host.

```
SMUID.StartTrustedSession [ See StartTrustedSession definition in 5.2.3.3]
=>
SMUID.SyncTrustedSession [
HostSessionID : uinteger,
SPSessionID : uinteger,
SPResponse = bytes,
SPEncryptSessionKey = bytes,
SPIntegritySessionKey = bytes,
SignedHash = bytes ]
```

5.2.3.4.1 HostSessionID

The **HostSessionID** parameter in the `SyncTrustedSession` invocation SHALL be the same as that in the `StartSession` invocation.

5.2.3.4.2 SPSessionID

The **SPSessionID** parameter in the `SyncTrustedSession` invocation is the TPer side session number, which was assigned by the TPer and delivered to the host in the `SyncSession` method.

5.2.3.4.3 SPResponse

A value is submitted in the SPResponse parameter if the `StartSession` method contained a HostChallenge parameter value. The response is dictated by the `Operation` column value and credential of the SPSigningAuthority.

5.2.3.4.4 SPEncryptSessionKey

The **SPEncryptSessionKey** is the session keyset generated by the SP and encrypted with the key used for exchange with the host (see Session Startup (section 5.3.4.1.4) for more information). This session keyset is used in secure messaging to encrypt packets sent from the SP to the host.

5.2.3.4.5 SPIntegritySessionKey

The **SPIntegritySessionKey** is the session keyset generated by the host and encrypted with the key used for exchange with the host. This session keyset is used to create a MAC of the data sent from the SP to the host (if required), to aid in integrity assurance.

5.2.3.4.6 SignedHash

The optional **SignedHash** parameter of each session startup method is present if hashing is required by the Control Authority for that communicator (see 5.3.4.1.4). This is a signed hash of all the other parameters to the method, other than the SignedHash parameter. The purpose of this is to provide integrity during session startup, prior to the point when secure messaging begins.

The SP Control Authority, if referenced by the Host Control Authority, identifies the hash type and signing type if hashing has been called out on messages from the SP to the host (see 5.3.4.1.7).

5.2.3.5 CloseSession Method

This method SHALL only be transmitted by the TPer. The TPer MAY transmit this method to notify the host that it is aborting the session identified in the `CloseSession` method, as well as all open un-committed transactions and methods undergoing processing (see 3.3.7.1.5).

```
SMUID.CloseSession [
RemoteSessionNumber : uinteger,
LocalSessionNumber : uinteger ]
```

5.2.3.5.1 RemoteSessionNumber

This is the portion of the session number assigned by the host (i.e. the HSN portion of the packet header Session field for the aborted session).

5.2.3.5.2 LocalSessionNumber

This is the portion of the session number assigned by the TPer (i.e. the TSN portion of the packet header Session field for the aborted session).

5.3 Base Template

5.3.1 Overview

The Base Template defines a common set of tables and methods, a subset of which SHALL be incorporated into all SPs.

5.3.1.1 Base Template Tables and Methods Overview

Begin Informative Content

Base Template tables are categorically divided into the following groups:

a. General metadata tables – store an SP's self-descriptive information, such as SP identification, size, and version numbers.
b. *Table and method metadata tables – store data about the tables and methods that make up* this SP.
c. Access control tables – define authorities, the secrets and authentication methods those authorities require, and the access control associations that permit method operation.
d. Credential tables – define available encryption/decryption algorithms and authentication mechanisms, and also store associated secrets or keys.

Base Template methods are divided into the following groups:

a. Basic table – enable creation of tables, addition and deletion of rows to tables, and modification of table cell values.
b. Access control – define which authorities are permitted to successfully invoke which methods and modify ACLs.

End Informative Content

5.3.2 Data Structures

5.3.2.1 General Metadata Group - SPInfo (Object Table)

The `SPInfo` table of each SP contains information about the SP, and a copy of some relevant information from the Admin SP. This table SHALL have exactly one row.

The `SPID` of the `SPInfo` table and the `GUDID` of the `TPerInfo` table in the Admin SP form an sp_guid that uniquely identifies the SP.

Table 169 SPInfo Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	SPID		uid
0x02	Name		name
0x03	Size		uinteger_8
0x04	SizeInUse		uinteger_8
0x05	SPSessionTimeout		uinteger_4
0x06	Enabled		boolean

5.3.2.1.1 UID

This is the unique identifier of this row of the `SPInfo` table.

This column SHALL NOT be modifiable by the host.

5.3.2.1.2 SPID

This is the unique identifier of this SP as assigned in the Admin SP's SP table.

This column SHALL NOT be modifiable by the host.

5.3.2.1.3 Name

This is the name of the SP. This SHALL be the same as the name recorded for this SP in the Admin SP's SP table.

This column SHALL NOT be modifiable by the host.

5.3.2.1.4 Size

This defines the total space allocated for the SP at creation, in bytes. This value SHALL be the same as the value of the Bytes column in the SP's object in the Admin SP's SP table.

This column SHALL NOT be modifiable by the host.

5.3.2.1.5 SizeInUse

This value is the amount of the allocated space that is in use (for tables), in bytes.

This column SHALL NOT be modifiable by the host.

5.3.2.1.6 SPSessionTimeout

This is the length of timeout interval (in milliseconds) that this SP uses by default.

5.3.2.1.7 Enabled

The value of this column identifies whether the SP is enabled or disabled. The column value is True if the SP is enabled, False if the SP is disabled.

Initial access control over modification of this column SHALL permit only the SP Owner (i.e. the Admins class authority) to disable or reenable this SP.

When the value of this column is False, the operation of the SP is modified according to 5.3.5.1.

As soon as the method invocation that changes this column value to False completes successfully, even inside of a transaction, the SP SHALL be considered disabled.

5.3.2.2 General Metadata Group - SPTemplates (Object Table)

Begin Informative Content

The SPTemplates table is an object table that identifies the component templates used to form the SP.

End Informative Content

There SHALL be one row in this table for each template used to create the SP, including a row for the Base Template (for all SPs), and one for the Admin Template in the Admin SP's SPTemplates table.

Table 170 SPTemplates Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	TemplateID		Template_object_ref
0x02	Name		name
0x03	Version		bytes_4

5.3.2.2.1 UID

This is the unique identifier of this row of the `SPTemplates` table.

This column SHALL NOT be modifiable by the host.

5.3.2.2.2 TemplateID

The value of the `TemplateID` column is the UID assigned to this template in the Admin SP's `Template` table.

This column SHALL NOT be modifiable by the host.

5.3.2.2.3 Name

This is the name of the template used as a component in the creation of this SP. This SHALL be the same as the value recorded in `Name` column of the Admin SP's `Template` table for the associated template.

This column SHALL NOT be modifiable by the host.

5.3.2.2.4 Version

The value of the `Version` column refers to TCG defined versions of templates. For devices compliant with the template versions defined in this specification, the 4-byte value SHALL be `0x00 0x00 0x00 0x02`.

This column SHALL NOT be modifiable by the host.

5.3.2.3 Table and Method Metadata Group - Table (Object Table)

Begin Informative Content

The `Table` table contains one row for each table descriptor object, which store metadata about each of the tables in the SP.

End Informative Content

In the `Table` table of every SP, there SHALL be a row for each table that exists in that SP. Each of these rows SHALL have a `CommonName` column value. Each table at issuance SHALL have a `CommonName` column value that is the name of the template from which that table was issued – the template name is the name from the associated row in the Admin SP's `Template` table.

The `Table` table in the Admin SP includes a row for each table that the TPer supports, in addition to a row for each table that exists in the Admin SP.

Table 171 Table Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	TemplateID	Yes	Template_object_ref

Column Number	Column Name	IsUnique	Column Type
0x04	Kind		table_kind
0x05	Column		Column_object_ref
0x06	NumColumns		uinteger_4
0x07	Rows		uinteger_4
0x08	RowsFree		uinteger_4
0x09	RowBytes		uinteger_4
0x0A	LastID		uid
0x0B	MinSize		uinteger_4
0x0C	MaxSize		uinteger_4

5.3.2.3.1 UID

This is the unique identifier of this row of the `Table` table.

This column SHALL NOT be modifiable by the host.

5.3.2.3.2 Name

This is the name of the table.

This column SHALL NOT be modifiable by the host for tables that are created during issuance.

5.3.2.3.3 CommonName

This is a name that MAY be shared among multiple table descriptor objects.

This column SHALL NOT be modifiable by the host for tables that are created during issuance.

5.3.2.3.4 TemplateID

In the Admin SP, this column is used to identify the template to which this table belongs and indicates a table that is not present in the Admin SP. In SPs other than the Admin SP, the value of this column SHALL be zeroes. See 5.4.4.1 for details.

This column SHALL NOT be modifiable by the host.

5.3.2.3.5 Kind

This value indicates the type of table – either object or byte.

This column SHALL NOT be modifiable by the host.

5.3.2.3.6 Column

This is a reference to the `Column` table row of this table's first column. For byte tables this value SHALL be the null uid.

This column SHALL NOT be modifiable by the host.

5.3.2.3.7 NumColumns

This value indicates the number of columns in the table. For byte tables this SHALL be `0x01`.

This column SHALL NOT be modifiable by the host.

5.3.2.3.8 Rows

This value indicates the actual number of rows that have been created for the table.

This column SHALL NOT be modifiable by the host.

5.3.2.3.9 RowsFree

This value indicates the number of unused rows in the table out of those allocated for use.

This column SHALL NOT be modifiable by the host.

5.3.2.3.10 RowBytes

This value is the number of bytes in each row of the table. This is the total number of bytes utilized by each table row, and SHALL include bytes devoted to overhead for system columns, type identification, etc.

This column SHALL NOT be modifiable by the host.

5.3.2.3.11 LastID

For object tables, this value is the most recent uid assigned to an object in that table. For byte tables, this value SHALL be the null uid.

This column SHALL NOT be modifiable by the host.

5.3.2.3.12 MinSize

This is the number of rows initially requested for this table. The table is able to contain at least this many rows. This column is user-settable (access control permitting). For more information see 5.3.4.2.1.

5.3.2.3.13 MaxSize

This is a host-defined maximum number of rows that MAY exist in this table. The table SHALL never have more than this many rows, although the TPer is not required to guarantee that the table can grow to MaxSize rows.

This column is user-settable (access control permitting), but the TPer MAY prevent the value in this column from being changed. A value of 0 indicates no host-defined limit of rows that MAY be created in this table.

5.3.2.4 Table and Method Metadata Group - Column (Object Table)

The `Column` table SHALL have one row for every column of every object table. Byte tables SHALL NOT have representative columns in the `Column` table.

The SP implementation is free to have hidden system columns in any table, as long as those columns do not interfere with host operations, including the operation of any methods invoked on that table. These columns SHALL NOT be recorded in the `Column` table.

The `Column` table in the Admin SP includes a row for each column that the TPer supports, in addition to a row for each column that exists in the Admin SP.

Table 172 Column Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name		name
0x02	CommonName		name
0x03	Type		Type_object_ref
0x04	IsUnique		boolean
0x05	ColumnNumber		uinteger_4
0x06	Transactional		boolean_def_true
0x07	Next		Column_object_ref

Column Number	Column Name	IsUnique	Column Type
0x08	AttributeFlags		attr_flags

5.3.2.4.1 UID

This is the unique identifier of this row of the `Column` table.

This column SHALL NOT be modifiable by the host.

5.3.2.4.2 Name

This is the name of the column.

This column SHALL NOT be modifiable by the host for tables that are created during issuance.

5.3.2.4.3 CommonName

This is a name that MAY be shared among multiple table descriptor objects. The value of the `CommonName` column for rows that exist upon issuance is the name of the template (from the `SPTemplates` table) to which that column belongs.

This column SHALL NOT be modifiable by the host for tables that are created during issuance.

5.3.2.4.4 Type

The value of this column identifies the type formatting the column's data.

This column SHALL NOT be modifiable by the host.

5.3.2.4.5 IsUnique

The value of this column indicates whether the column participates in the unique column combination for the table. The value of this column is True if this column is, or is part of, the unique columns for the table. If the value of this column is False, this column is not a part of the table's unique columns.

This column SHALL NOT be modifiable by the host.

5.3.2.4.6 ColumnNumber

The value of this column identifies the number of the column, and by extension, the ordering of columns in the containing table.

This column SHALL NOT be modifiable by the host.

5.3.2.4.7 Transactional

This value indicates whether the column is subject to transactional rollback.

If the value of the `Transactional` column is False, then modifications to this column take effect immediately, even if the method invocation that modifies the column is included in a transaction that has not yet resolved. Changes to the column are not rolled back if the transaction containing the modification is aborted. The value of this column for user-created table columns SHALL be True.

This column SHALL NOT be modifiable by the host.

5.3.2.4.8 Next

This is a reference to the row of the `Column` table that represents the next column in this column's table. If this is the last column in the containing table, then the value of this column is the NULL UID.

This column SHALL NOT be modifiable by the host.

5.3.2.4.9 AttributeFlags

Identifies globally assigned attributes for this table column, such as whether or not the `Get` and `Set` methods are globally permitted for this column regardless of access controls.

In the case of tables created by the host post-issuance, the value of this column for resulting additional rows in the Column table SHALL be the empty set. System-managed columns (such as the UID column) of those tables SHALL have a value of {1} for that column in the corresponding Column table row.

This column SHALL NOT be modifiable by the host.

5.3.2.5 Table and Method Metadata Group - Type (Object Table)

Begin Informative Content

The Type table stores the format and metadata for all of the column types used in the SP. The host adds host-defined types by invoking the CreateRow method on the Type table.

The Type table values that represent the built-in types, as well as all those types pre-defined in this specification, are found in 5.1.1.

End Informative Content

Any of the types predefined in the Core Specification MAY be included by default in the table for an SP. The default contents of the table are SSC-specific.

No user-defined types SHALL be removed by the Delete or DeleteRow methods unless the TPer is able to verify that no column of that type is currently in use.

Types are often constructed of other types. The TPer SHALL prevent modification or removal of a type object upon which another type is dependent.

The TPer SHALL prevent type recursion.

Table 173 Type Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	Format		type_def
0x04	Size		uinteger_2

5.3.2.5.1 UID

This is the unique identifier of this row in the Type table.

This column SHALL NOT be modifiable by the host.

5.3.2.5.2 Name

This is the name of the type.

This column SHALL NOT be modifiable by the host.

5.3.2.5.3 CommonName

This is a name that MAY be shared by multiple types.

This column SHALL NOT be modifiable by the host.

5.3.2.5.4 Format

The value of this column describes the format of the data type. For details, see the format specification, section 5.1.1.

This column SHALL NOT be modifiable by the host.

5.3.2.5.5 Size

This is the size (in bytes) needed to store a value of this type. The value of the `Size` column includes any necessary overhead (such as for bytes{max=<n>}, for tagging a value of an `Alternative_Type`, etc. The TPer calculates the value of this column. It is an error for the host to specify a value for this column in the `CreateRow` method invocation. This value SHALL be 0 for a base type (integer, uinteger, bytes, max bytes).

This column SHALL NOT be modifiable by the host.

5.3.2.6 Table and Method Metadata Group - MethodID (Object Table)

This table associates method names and uids. Access control SHALL permit this table to be read with the use of the Anybody authority, and SHALL prevent this table from being modified. In the `MethodID` table of every SP, there SHALL be a row for each method that MAY be invoked within a session to that SP.

The `Name-CommonName-TemplateID` column value combination SHALL be unique for each row in the table.

Table 174 MethodID Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	TemplateID	Yes	Template_object_ref

5.3.2.6.1 UID

This is the unique identifier of this row in the `MethodID` table. This is also the uid value used to invoke the method.

This column SHALL NOT be modifiable by the host.

5.3.2.6.2 Name

This is the name of the method.

For `MethodID` objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.6.3 CommonName

This is a name that MAY be shared by multiple methods.

Each row that exists at issuance SHALL have a `CommonName` column value that is the name of the template from which it was issued. This is the name of the template from the Admin SP's `Template` table.

For MethodID objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.6.4 TemplateID

In the Admin SP, this column is used to identify the template to which this method belongs and indicates a method that is not present in the Admin SP. In SPs other than the Admin SP, the value of this column SHALL be zeroes. See 5.4.4.1 for details.

This column SHALL NOT be modifiable by the host.

5.3.2.7 Table and Method Metadata Group - AccessControl (Object Table)

Begin Informative Content

The AccessControl table contains SP/method, table/method, and object/method access control associations and logging settings, and each access control association's related meta-ACL access requirements and meta-ACL logging settings.

End Informative Content

New rows SHALL NOT be created in or deleted from the AccessControl table directly (i.e. via CreateRow, Delete, or DeleteRow).

New rows are created in the AccessControl table as a side effect whenever a table is created or when a row in an object table is created. New rows added to the AccessControl table in this way SHALL NOT cause additional new rows to be added to the AccessControl table.

AccessControl table rows MAY be deleted through the use of the DeleteMethod method. AccessControl table rows associated with a particular object or table SHALL be removed whenever that table or object is deleted.

Table 175 AccessControl Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	InvokingID	Yes	table_or_object_ref
0x02	MethodID	Yes	MethodID_object_ref
0x03	CommonName		name
0x04	ACL		ACL
0x05	Log		log_select
0x06	AddACEACL		ACL
0x07	RemoveACEACL		ACL
0x08	GetACLACL		ACL
0x09	DeleteMethodACL		ACL
0x0A	AddACELog		log_select
0x0B	RemoveACELog		log_select
0x0C	GetACLLog		log_select
0x0D	DeleteMethodLog		log_select
0x0E	LogTo		LogList_object_ref

5.3.2.7.1 UID

This is the unique identifier of this row in the AccessControl table.

This column SHALL NOT be modifiable by the host.

5.3.2.7.2 InvokingID

This is the uidref to the SP/Table/Object portion of this access control association.

This column SHALL NOT be modifiable by the host.

5.3.2.7.3 MethodID

This is the unique identifier of the method portion of this access control situation, and is the same as the method's UID column value in the MethodID table.

This column SHALL NOT be modifiable by the host.

5.3.2.7.4 CommonName

This is a name that MAY be shared among multiple access control associations. The value for this column when a row is created is the empty string.

This column SHALL NOT be modifiable by the host.

5.3.2.7.5 ACL

This is the access control list for this SP/method, table/method, or object/method combination. This column is modified/accessed via the methods `GetACL`, `RemoveACE`, and `AddACE`. This column SHALL NOT be modifiable directly via the `Set` method.

5.3.2.7.6 Log

This column identifies the logging conditions when this method is invoked on this SP/table/object. The conditions indicate whether logging is performed when the method succeeds, fails, both, or neither. This column SHALL be disregarded if the Log Template has not been issued into the SP, and SHOULD be set to 0.

5.3.2.7.7 AddACEACL

This column holds the access control list that controls invocation of the `AddACE` method on the `ACL` column of this row in the `AccessControl` table.

5.3.2.7.8 RemoveACEACL

This column holds the access control list that controls invocation of the `RemoveACE` method on the `ACL` column of this row in the `AccessControl` table.

5.3.2.7.9 GetACLACL

This column holds the access control list that controls invocation of the `GetACL` method on the `ACL` column this row in the `AccessControl` table.

5.3.2.7.10 DeleteMethodACL

This column holds the access control list that controls invocation of the `DeleteMethod` method on the access control association represented by this row in the `AccessControl` table.

5.3.2.7.11 AddACELog

This column identifies the conditions under which logging of the `AddACE` method invocation on this access control association occurs. This column SHALL be disregarded if the Log Template has not been issued into the SP, and SHOULD be set to 0.

5.3.2.7.12 RemoveACELog

This column identifies the conditions under which logging of the `RemoveACE` method invocation on this access control association occurs. This column SHALL be disregarded if the Log Template has not been issued into the SP, and SHOULD be set to 0.

5.3.2.7.13 GetACLLog

This column identifies the conditions under which logging of the `GetACL` method invocation on this access control association occurs. This column SHALL be disregarded if the Log Template has not been issued into the SP, and SHOULD be set to 0.

5.3.2.7.14 DeleteMethodLog

This column identifies the conditions under which logging of the `DeleteMethod` method invocation on this access control association occurs. This column SHALL be disregarded if the Log Template has not been issued into the SP, and SHOULD be set to 0.

5.3.2.7.15 LogTo

This column value is a uidref to a LogList object. Log entries for this access control association are added to the Log table associated with that LogList object. This column SHALL be disregarded if the Log Template has not been issued into the SP, and SHOULD be set to the NULL UID.

5.3.2.8 Table and Method Metadata Group - SecretProtect (Object Table)

This table column is used by the host to identify the key protection mechanism(s), if any, in use by the storage device to "hide" the device's media encryption key material/secrets.

Table 176 SecretProtect Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Table		Table_object_ref
0x02	ColumnNumber		uinteger_4
0x03	ProtectMechanisms		protect_types

5.3.2.8.1 UID

This is the unique identifier of this row of the `SecretProtect` table.

This column SHALL NOT be modifiable by the host.

5.3.2.8.2 Table

This is a uid ref to an object in the `Table` table. This represents the table that, in conjunction with the column value identified in the `ColumnNumber` column, is protected using the mechanism(s) specified in the `ProtectMechanisms` column.

This column SHALL NOT be modifiable by the host.

5.3.2.8.3 ColumnNumber

This is a column number. This number represents a column in table (as identified in the `Table` column), which is protected using the mechanism(s) specified in the `ProtectMechanisms` column.

This column SHALL NOT be modifiable by the host.

5.3.2.8.4 ProtectMechanisms

This column identifies the type of key protection used by the storage device to hide key material/secrets. The protection mechanisms identified in this column SHALL all be applied to protection of the associated table column value.

This column SHALL NOT be modifiable by the host.

5.3.2.9 Access Control Metadata Group - ACE (Object Table)

The `ACE` table SHALL have one row for each access control element that MAY be referenced in the `AccessControl` table's `ACL` column.

Table 177 ACE Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name

Column Number	Column Name	IsUnique	Column Type
0x03	BooleanExpr		AC_element
0x04	Columns		ACE_columns

5.3.2.9.1 UID

This is the unique identifier of this row in the `ACE` table.

This column SHALL NOT be modifiable by the host.

5.3.2.9.2 Name

This is the name of the ACE.

For ACE objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.9.3 CommonName

This is a name that MAY be shared by multiple ACE objects.

For ACE objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.9.4 BooleanExpr

This column value is a Boolean expression of Authorities that authorizes the ACE if the expression evaluates to True. If the conditions described in this access control element are True, then the ACE is considered authenticated.

5.3.2.9.5 Columns

This column value dentifies the columns to which this ACE applies. An empty set indicates that this ACE applies to all columns in the table (when referenced in the ACL of a method that supports column restrictions).

The value for the `Columns` column SHALL be applicable to the table upon which a method requiring authentication of this ACE is being invoked. If the method is not column-dependant (e.g. the `Next` method), this column is ignored.

5.3.2.10 Access Control Metadata Group - Authority (Object Table)

Begin Informative Content
A row of the `Authority` Table is called an authority. An authority is a specific use of a credential and, possibly, other authorities. A class authority is an authority object referenced by multiple individual authorities and does not use a credential.
End Informative Content

Table 178 Authority Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	IsClass		boolean
0x04	Class		Authority_object_ref
0x05	Enabled		boolean
0x06	Secure		messaging_type
0x07	HashAndSign		hash_protocol
0x08	PresentCertificate		boolean

Column Number	Column Name	IsUnique	Column Type
0x09	Operation		auth_method
0x0A	Credential		cred_object_uidref
0x0B	ResponseSign		Authority_object_ref
0x0C	ResponseExch		Authority_object_ref
0x0D	ClockStart		date
0x0E	ClockEnd		date
0x0F	Limit		uinteger_4
0x10	Uses		uinteger_4
0x11	Log		log_select
0x12	LogTo		LogList_object_ref

5.3.2.10.1 UID

This is the unique identifier of this row in the `Authority` table.

This column SHALL NOT be modifiable by the host.

5.3.2.10.2 Name

This is the name of the authority.

For Authority objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.10.3 CommonName

This is a name that MAY be shared by multiple authorities.

For Authority objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.10.4 IsClass

This column identifies whether or not this is an individual authority or a class authority.

If True, this row is a class authority. If False, this row is an individual authority.

This column SHALL NOT be modifiable by the host.

5.3.2.10.5 Class

The `Class` column identifies the authority class of which an authority object is a member. Class authorities MAY be members of another class authority. However, this SHALL only be valid if it extends to one level. Class authorities are not permitted to be members of a class authority that is already a member of another class authority. The TPer SHALL enforce this requirement.

The value of this column SHALL be a NULL UID reference if the authority is not a member of a class.

5.3.2.10.6 Enabled

This column identifies whether the authority is enabled, thus identifying if the authority object is authenticatable. When this value is True, this authority is enabled.

5.3.2.10.7 Secure

The `Secure` column identifies the type of secure messaging (if any) that is required by this authority, and identifies the size of the key(s) that SHALL be generated during secure session startup if secure messaging is required. A value of “None” indicates secure messaging is not required. The value of this column SHALL be enforced when any attempt is made to authenticate this authority, including the use of the `Authenticate` method. If the conditions are not met when authentication is attempted, that authentication SHALL fail.

The options for the `Secure` column, which are the options defined for the messaging_type type, are identified in Table 179.

Table 179 Secure Column Values

Column value	Algorithm	Secure Messaging Type
0	None	None
1	HMAC_SHA_256	Integrity only
2	HMAC_SHA_384	Integrity only
3	HMAC_SHA_512	Integrity only
4	RSASSA-PKCS1-v1_5 _1024	Integrity only
5	RSASSA-PKCS1-v1_5 _2048	Integrity only
6	RSASSA-PKCS1-v1_5 _3072	Integrity only
7	RSASSA-PSS_1024	Integrity only
8	RSASSA-PSS_2048	Integrity only
9	RSASSA-PSS_3072	Integrity only
10	ECDSA_256_SHA_256	Integrity only
11	ECDSA_384_SHA_384	Integrity only
12	ECDSA_512_SHA_512	Integrity only
13	CMAC_128 with 128-bit MAC	Integrity only
14	CMAC_256 with 128-bit MAC	Integrity only
15	GMAC_128 with 128-bit MAC and 96-bit IV	Integrity only
16	GMAC_256 with 128-bit MAC and 96-bit IV	Integrity only
17-63	RESERVED	Integrity only
64	AES_CBC_128	Confidentiality only
65	AES_CBC_256	Confidentiality only
66-128	RESERVED	Confidentiality only
129	AES_CBC_128 with HMAC_SHA_256	Integrity and Confidentiality
130	AES_CBC_256 with HMAC_SHA_256	Integrity and Confidentiality
131	AES_CBC_256 with HMAC_SHA_384	Integrity and Confidentiality
132	AES_CBC_256 with HMAC_SHA_512	Integrity and Confidentiality
133	AES_CCM_128 with 128-bit MAC	Integrity and Confidentiality
134	AES_CCM_256 with 128-bit MAC	Integrity and Confidentiality
135	AES_GCM_128 with 128-bit MAC	Integrity and Confidentiality
136	AES_GCM_256 with 128 bit MAC	Integrity and Confidentiality
137-255	RESERVED	Integrity and Confidentiality

Note that the IV size for both the CCM and GCM modes is 12-bytes. The lower 8-bytes are directly provided within the secure message. The upper 4-bytes of the IV are taken from the last 4-bytes of the EncryptSessionKey parameters of the `StartTrustedSession/SyncTrustedSession` method pair (see [16] for CCM and [17] for GCM). The EncryptSessionKey parameters of the `StartTrustedSession/SyncTrustedSession` method pair SHALL be 4-bytes longer for the CCM and

GCM modes to accommodate the 4-bytes used as 'salt' within the IV. For CMAC see [15]. For GMAC see [17]. For RSASSA-PKCS1-v1_5 and RSASSA-PSS see [18].

5.3.2.10.8 HashAndSign

The value of the `HashAndSign` column determines if hashing and signing of session startup method parameters are required. If the value of this column is other than “None”, a signed hash of the session startup method parameters SHALL be used during session startup.

The value of the `Operation` column and the type of the credential referenced in the `Credential` column (and the hash protocol identified in that credential) determine the type of the hashing and signing.

`HashAndSign` is only enforced for a particular authority during session startup. Otherwise, this attribute SHALL be ignored (for instance, during an `Authenticate` method invocation). For additional information see 5.3.4.1.4.

5.3.2.10.9 PresentCertificate

The value of this column indicates if a certificate needs to be supplied with an authority at session startup. If the value of the `PresentCertificate` column is True, the authority is a public key authority, and the credential contains a certificate or certificate chain, then a certificate or certificate chain associated with this authority SHALL be sent as a parameter of the session startup protocol. If any of those conditions is False, no certificate is required to be sent.

5.3.2.10.10 Operation

The value of this column identifies the operation (see 5.3.4.1.3) to perform with the associated credential (e.g., Exchange, Sign, SymK, HMAC, Password, None).

5.3.2.10.11 Credential

The value of the `Credential` column identifies the specific credential object to be used with this authority. For a class authority, the value of this column SHALL be zeroes a NULL UID reference.

5.3.2.10.12 ResponseSign

This column identifies the signing authority with which the SP SHALL respond during session startup. This column value MAY be self-referential. The value of the `ResponseSign` column identifies the authority with which the TPer SHALL respond in the `SyncSession` method of the session startup method exchange, as the SP Signing Authority. If the value of this column is the NULL UID, then no SP Signing Authority is used for initiating that session.

5.3.2.10.13 ResponseExch

This column identifies the exchange authority with which the SP SHALL respond during session startup. This MAY be self-referential. The value of the `ResponseExch` column identifies the authority with which the TPer SHALL respond in the `SyncSession` method of the session startup method exchange, as the SP Exchange Authority. If the value of this column is the NULL UID, then no SP Exchange Authority is used for initiating that session.

5.3.2.10.14 ClockStart

This value identifies the date on which this authority becomes valid.

An authority is automatically valid starting on the date defined in the `ClockStart` column if the TPer supports this capability. A value of either all zeroes or an empty struct indicates no start date, and the authority SHALL be authenticatable until the date in the `ClockEnd` column is reached. Attempts to authenticate an authority before the `ClockStart` column value date has been reached SHALL fail.

The values in the ClockStart column's date struct SHALL be complete and valid or the authority SHALL NOT be authenticatable.

If the Clock Template has not been issued with this SP, then the value of this column SHOULD be disregarded, and SHALL be set to an empty struct. Any authority with a non-zero ClockStart date SHALL NOT be authenticatable if the ClockTime table's TrustMode column is "Timer".

See 5.5 for additional details on the Clock Template.

5.3.2.10.15 ClockEnd

This value identifies the date on which this authority expires/becomes invalid.

An authority is automatically invalid starting on the date defined in the ClockEnd column if the TPer supports this capability. A value of either all zeroes or an empty struct indicates no end date, and the authority's ability to be authenticated SHALL NOT expire. Attempts to authenticate an authority after the ClockEnd column value date has been passed SHALL fail.

The values in the ClockStart column's date struct SHALL be complete and valid or the authority SHALL NOT be authenticatable.

If the Clock Template has not been issued with this SP, then the value of this column SHOULD be disregarded, and SHALL be set to an empty struct. Any authority with a non-zero ClockEnd date SHALL NOT be authenticatable if the ClockTime table's TrustMode column is "Timer".

See 5.5 for additional details on the Clock Template.

5.3.2.10.16 Limit

The Limit column defines a limit on the number of times that an authority MAY be authenticated, either explicitly or implicitly. This value represents the maximum number of total successful authentications with this authority, including session start-up invocations and Authenticate method invocations. A value of 0 SHALL mean no limit.

5.3.2.10.17 Uses

This column defines the total number of successful authentications made with this authority, including both successful session start-up invocations and Authenticate method invocations. The value of the Uses column identifies the number of times an authority has been successfully authenticated.

If the value of Uses is equal to or greater than the value of Limit for this authority and the value of the Limit column is not 0, then this authority SHALL NOT be authenticatable.

This value SHALL NOT be subject to transactional rollbacks.

5.3.2.10.18 Log

The value of the Log column identifies when uses of this authority (i.e., authentications and authentication attempts) are logged.

If the Log Template has not been issued into the SP, then this column SHALL be disregarded and SHOULD be set to zero.

5.3.2.10.19 LogTo

This column value is a uidref to a LogList object. Log entries for this access control association are added to the Log table associated with that LogList object. This column SHALL be disregarded if the Log Template has not been issued into the SP, and SHOULD be set to the NULL UID.

5.3.2.11 Access Control Metadata Group - Certificates (Object Table)

Table 180 Certificates Table Description

Column Number	Column Name	IsUnique	Column Type

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	CertData		byte_table_ref
0x04	CertSize		uinteger_4

5.3.2.11.1 UID

This is the unique identifier of this row in the `Certificates` table.

This column SHALL NOT be modifiable by the host.

5.3.2.11.2 Name

This is the name of the certificate.

For `Certificates` objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.11.3 CommonName

This is a name that MAY be shared by multiple certificates.

For `Certificates` objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.11.4 CertData

This is the uidref to the byte table that holds the certificate data for this `Certificates` object.

5.3.2.11.5 CertSize

The value of this column is the number of bytes actually used in the certificate.

5.3.2.12 Credential Table Group - C_PIN (Object Table)

The `C_PIN` table contains one row for each password credential.

The `C_PIN` object with `UID=0x00 0x00 0x00 0x0B 0x00 0x00 0x00 0x01` and `Name="SID"` is the default SID object.

Table 181 C_PIN Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	PIN		password
0x04	CharSet		byte_table_ref
0x05	TryLimit		uinteger_4
0x06	Tries		uinteger_4
0x07	Persistence		boolean

5.3.2.12.1 UID

This is the unique identifier of this row in the `C_PIN` table.

This column SHALL NOT be modifiable by the host.

5.3.2.12.2 Name

This is the name of the C_PIN object.

For C_PIN objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.12.3 CommonName

This is a name that MAY be shared by multiple C_PIN objects.

For C_PIN objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.12.4 PIN

This is the bytes value to which authentication attempts on this C_PIN object are matched.

5.3.2.12.5 CharSet

This is a reference to the byte table that holds the character set used for TPer-generated PIN column values created using the GenKey method.

If the value of this column is a NULL UID reference, then the default character set is used with the GenKey method.

See 5.3.4.1.1.1 for a description of the use of this column.

5.3.2.12.6 TryLimit

The value of this column is the maximum number of failed authentication attempts that are able to be made using this C_PIN object.

The default value of the TryLimit column when a new C_PIN object is created is 0. The value 0 in this column indicates that there is no limit on the number of authentication attempts for that object.

For more information on the uses of this column see 5.3.4.1.1.2.

5.3.2.12.7 Tries

This column identifies the current number of failed authentication attempts using this C_PIN object.

For more information on the uses of this column see 5.3.4.1.1.2.

5.3.2.12.8 Persistence

The value of this column identifies if value of Tries column is persistent through power cycles.

5.3.2.13 Credential Table Group - C_RSA_1024 (Object Table)

Table 182 C_RSA_1024 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	Format		padding_type
0x04	Pu_Exp		uinteger_128
0x05	Mod		uinteger_128
0x06	Pr_Exp		uinteger_128
0x07	P		uinteger_64
0x08	Q		uinteger_64

Column Number	Column Name	IsUnique	Column Type
0x09	Dmp1		uinteger_64
0x0A	Dmq1		uinteger_64
0x0B	Iqmp		uinteger_64
0x0C	Hash		hash_protocol
0x0D	ChainLimit		int_1_def_0
0x0E	Certificate		Certificates_object_ref

5.3.2.13.1 UID

This is the unique identifier of this row in the `C_RSA_1024` table.

This column SHALL NOT be modifiable by the host.

5.3.2.13.2 Name

This is the name of the `C_RSA_1024` object.

For `C_RSA_1024` objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.13.3 CommonName

This is a name that MAY be shared among `C_RSA_1024` objects.

For `C_RSA_1024` objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.13.4 Format

This column defines the type of padding used with RSA encryption.

5.3.2.13.5 Pu_Exp

The value of this column is the RSA Public Exponent.

5.3.2.13.6 Mod

The value of this column is the RSA Public Modulus.

5.3.2.13.7 Pr_Exp

The value of this column is the RSA Private Exponent.

A value of `0x0` in this column indicates that there is no value, or that the column has not been initialized.

5.3.2.13.8 P

The value of this column is the p prime from the key generation.

A value of `0x0` in this column indicates that there is no value, or that the column has not been initialized.

5.3.2.13.9 Q

The value of this column is the q prime from the key generation.

A value of `0x0` in this column indicates that there is no value, or that the column has not been initialized.

5.3.2.13.10 Dmp1

The value of this column is d mod (p-1) (often known as dmp1).

A value of `0x0` in this column indicates that there is no value, or that the column has not been initialized.

5.3.2.13.11 Dmq1

The value of this column is d mod (q-1) (often known as dmq1).

A value of `0x0` in this column indicates that there is no value, or that the column has not been initialized.

5.3.2.13.12 Iqmp

The value of this column is (1/q) mod p (often known as iqmp).

A value of `0x0` in this column indicates that there is no value, or that the column has not been initialized.

5.3.2.13.13 Hash

If an authority object that references this `C_RSA_1024` object has a `HashAndSign` column value of True, this column identifies the hash algorithm to create the session startup method parameter MAC to be signed by this credential.

5.3.2.13.14 ChainLimit

This column value identifies the chaining limit for using a chained down key from the base certificate. A value of –1 indicates no limit. A value of 0 indicates no chain.

5.3.2.13.15 Certificate

This is a reference to a `Certificates` object, which if needed identifies a chained set of unencoded X.509 certificates to prove an ancestor authority.

5.3.2.14 Credential Table Group - C_RSA_2048 (Object Table)

Table 183 C_RSA_2048 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	Format		padding_type
0x04	Pu_Exp		uinteger_256
0x05	Mod		uinteger_256
0x06	Pr_Exp		uinteger_256
0x07	P		uinteger_128
0x08	Q		uinteger_128
0x09	Dmp1		uinteger_128
0x0A	Dmq1		uinteger_128
0x0B	Iqmp		uinteger_128
0x0C	Hash		hash_protocol
0x0D	ChainLimit		int_1_def_0
0x0E	Certificate		Certificates_object_ref

5.3.2.14.1 UID

This is the unique identifier of this row in the `C_RSA_2048` table.

This column SHALL NOT be modifiable by the host.

5.3.2.14.2 Name

This is the name of the C_RSA_2048 object.

For C_RSA_2048 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.14.3 CommonName

This is a name that MAY be shared among C_RSA_2048 objects.

For C_RSA_2048 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.14.4 Format

This column defines the type of padding used with RSA encryption.

5.3.2.14.5 Pu_Exp

The value of this column is the RSA Public Exponent.

5.3.2.14.6 Mod

The value of this column is the RSA Public Modulus.

5.3.2.14.7 Pr_Exp

The value of this column is the RSA Private Exponent.

A value of 0x0 in this column indicates that there is no value, or that the column has not been initialized.

5.3.2.14.8 P

The value of this column is the p prime from the key generation.

A value of 0x0 in this column indicates that there is no value, or that the column has not been initialized.

5.3.2.14.9 Q

The value of this column is the q prime from the key generation.

A value of 0x0 in this column indicates that there is no value, or that the column has not been initialized.

5.3.2.14.10 Dmp1

The value of this column is d mod (p-1) (often known as dmp1).

A value of 0x0 in this column indicates that there is no value, or that the column has not been initialized.

5.3.2.14.11 Dmq1

The value of this column is d mod (q-1) (often known as dmq1).

A value of 0x0 in this column indicates that there is no value, or that the column has not been initialized.

5.3.2.14.12 Iqmp

The value of this column is (1/q) mod p (often known as iqmp).

A value of 0x0 in this column indicates that there is no value, or that the column has not been initialized.

5.3.2.14.13 Hash

If an authority object that references this C_RSA_2048 object has a HashAndSign column value of True, this column identifies the hash algorithm to create the session startup method parameter MAC to be signed by this credential.

5.3.2.14.14 ChainLimit

This column value identifies the chaining limit for using a chained down key from the base certificate. A value of –1 indicates no limit. A value of 0 indicates no chain.

5.3.2.14.15 Certificate

This is a reference to a `Certificates` object, which if needed identifies a chained set of unencoded X.509 certificates to prove an ancestor authority.

5.3.2.15 Credential Table Group - C_AES_128 (Object Table)

Table 184 C_AES_128 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	Key		bytes_16
0x04	Mode		symmetric_mode
0x05	FeedbackSize		feedback_size
0x06	ResidualData		bytes_16
0x07	Hash		hash_protocol

5.3.2.15.1 UID

This is the unique identifier of this row in the `C_AES_128` table.

This column SHALL NOT be modifiable by the host.

5.3.2.15.2 Name

This is the name of the `C_AES_128` object.

For `C_AES_128` objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.15.3 CommonName

This is a name that MAY be shared by multiple `C_AES_128` objects.

For `C_AES_128` objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.15.4 Key

This column stores the key associated with this `C_AES_128` object.

5.3.2.15.5 Mode

This column value defines the encryption mode with which this credential SHALL be used.

5.3.2.15.6 FeedbackSize

This column defines the feedback size for CFB mode, and SHALL be ignored for all other modes.

5.3.2.15.7 ResidualData

The value in this column provides the IV for use with the `Encrypt`/`Decrypt` method (unless the IV parameter in the `EncryptInit`/`DecryptInit` method is invoked).

The value in the `ResidualData` column provides the IV for the `Encrypt`/`Decrypt` method (unless the IV parameter in the `EncryptInit`/`DecryptInit` method is invoked). The TPer then sets this value based on the last block encrypted by the `Encrypt` method or last block decrypted by the `Decrypt` method.

Subsequent method invocations use this column value as its IV. The value set to this column during `Encrypt/Decrypt` operations is dependent on this object's mode, as defined in Table 185.

5.3.2.15.8 Hash

The value of this column defines the hash protocol to be used with this credential.

Table 185 C_AES_128/C_AES_256 ResidualData Column Values After Encrypt/Decrypt/EncryptFinalize/DecryptFinalize

Mode	Column Value
ECB	All 00's
CBC	The ciphertext of the last block encrypted/decrypted
CFB	The (128 – FeedbackSize) LSBs of the last input to the AES cipher function, concatenated with the ciphertext of the last block encrypted/decrypted
OFB	The last output block of the AES cipher function
CTR	The last input block to the AES cipher function + 1
GCM	The last input block to the AES cipher function + 1
CCM	The last input block to the AES cipher function + 1

5.3.2.16 Credential Table Group - C_AES_256 (Object Table)

Table 186 C_AES_256 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	Key		bytes_32
0x04	Mode		symmetric_mode
0x05	FeedbackSize		feedback_size
0x06	ResidualData		bytes_16
0x07	Hash		hash_protocol

5.3.2.16.1 UID

This is the unique identifier of this row in the `C_AES_256` table.

This column SHALL NOT be modifiable by the host.

5.3.2.16.2 Name

This is the name of the `C_AES_256` object.

For `C_AES_256` objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.16.3 CommonName

This is a name that MAY be shared by multiple `C_AES_256` objects.

For C_AES_256 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.16.4 Key

This column stores the key associated with this C_AES_256 object.

5.3.2.16.5 Mode

This column value defines the encryption mode with which this credential SHALL be used.

5.3.2.16.6 FeedbackSize

This column defines the feedback size for CFB mode, and SHALL be ignored for all other modes..

5.3.2.16.7 ResidualData

The value in the ResidualData column provides the IV for the Encrypt/Decrypt method (unless the IV parameter in the EncryptInit/DecryptInit method is invoked). The TPer then sets this value based on the last block encrypted by the Encrypt method or last block decrypted by the Decrypt method. Subsequent method invocations use this column value as its IV. The value set to this column during Encrypt/Decrypt operations is dependent on this object's mode, as defined in Table 185.

5.3.2.16.8 Hash

The value of this column defines the hash protocol to be used with this credential.

5.3.2.17 Credential Table Group - C_EC_160 (Object Table)

Table 187 C_EC_160 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	p		uinteger_20
0x04	r		uinteger_20
0x05	b		uinteger_20
0x06	x		uinteger_20
0x07	y		uinteger_20
0x08	alpha		uinteger_20
0x09	u		uinteger_20
0x0A	v		uinteger_20
0x0B	Hash		hash_protocol
0x0C	ChainLimit		integer_1
0x0D	Certificate		Certificates_object_ref

5.3.2.17.1 UID

This is the unique identifier of this row in the C_EC_160 table.

This column SHALL NOT be modifiable by the host.

5.3.2.17.2 Name

This is the name of the C_EC_160 object.

For C_EC_160 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.17.3 CommonName

This is a name that MAY be shared by multiple C_EC_160 objects.

For C_EC_160 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.17.4 p

Modulus

5.3.2.17.5 r

Order of the curve

5.3.2.17.6 b

Curve coefficient ($y^2=x^3–3x+b \bmod p$)

5.3.2.17.7 x

Base point x-coordinate

5.3.2.17.8 y

Base point y-coordinate

5.3.2.17.9 alpha

Private key

5.3.2.17.10 u

Public key x-coordinate: (u, v) = α (x,y)

5.3.2.17.11 v

Public key y-coordinate: (u, v) = α (x,y)

5.3.2.17.12 Hash

The value of this column identifies the hash type used for ECDSA (message digesting), for ECDH and ECMQV (key derivation), and for creation of the MAC of session startup methods if a referencing authority has a HashAndSign column value of True.

5.3.2.17.13 ChainLimit

This column value identifies the chaining limit for using a chained down key from the base certificate. A value of –1 indicates no limit. A value of 0 indicates no chain.

5.3.2.17.14 Certificate

This is a reference to a Certificates object, which if needed identifies a chained set of unencoded X.509 certificates to prove an ancestor authority.

5.3.2.17.15 Values for C_EC_160

Table 188 represents the set of elliptic curve domain parameters as specified in [4]. The entries p, r, b, x and y are represented in decimal format. These are example values for a curve that MAY be used with the C_EC_160 table. These values are set as the default values for the associated columns when a new row is created in the C_EC_160 table and when values for those columns are not specified at table creation. These default values are not represented by a Type table entry; the TPer SHALL be required to keep track of these values and set them as defaults for new objects, as necessary.

Table 188 AACS Values for C_EC_160

Column	Value
p	900812823637587646514106462588455890498729007071
r	900812823637587646514106555566573588779770753047
b	366394034647231750324370400222002566844354703832
x	264865613959729647018113670854605162895977008838
y	518410759548831625104133927451689362961878086 97

5.3.2.18 Credential Table Group - C_EC_192 (Object Table)

Table 189 C_EC_192 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	p		uinteger_24
0x04	r		uinteger_24
0x05	b		uinteger_24
0x06	x		uinteger_24
0x07	y		uinteger_24
0x08	alpha		uinteger_24
0x09	u		uinteger_24
0x0A	v		uinteger_24
0x0B	Hash		hash_protocol
0x0C	ChainLimit		integer_1
0x0D	Certificate		Certificates_object_ref

5.3.2.18.1 UID

This is the unique identifier of this row in the `C_EC_192` table.

This column SHALL NOT be modifiable by the host.

5.3.2.18.2 Name

This is the name of the `C_EC_192` object.

For `C_EC_192` objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.18.3 CommonName

This is a name that MAY be shared by multiple `C_EC_192` objects.

For `C_EC_192` objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.18.4 p

Modulus

5.3.2.18.5 r

Order of the curve

5.3.2.18.6 b

Curve coefficient ($y^2=x^3–3x+b \bmod p$)

5.3.2.18.7 x

Base point x-coordinate

5.3.2.18.8 y

Base point y-coordinate

5.3.2.18.9 alpha

Private key

5.3.2.18.10 u

Public key x-coordinate: (u, v) = α (x,y)

5.3.2.18.11 v

Public key y-coordinate: (u, v) = α (x,y)

5.3.2.18.12 Hash

The value of this column identifies the hash type used for ECDSA (message digesting), for ECDH and ECMQV (key derivation), and for creation of the MAC of session startup methods if a referencing authority has a `HashAndSign` column value of True.

5.3.2.18.13 ChainLimit

This column value identifies the chaining limit for using a chained down key from the base certificate. A value of –1 indicates no limit. A value of 0 indicates no chain.

5.3.2.18.14 Certificate

This is a reference to a `Certificates` object, which if needed identifies a chained set of unencoded X.509 certificates to prove an ancestor authority.

5.3.2.18.15 Values for C_EC_192

Table 190 represents the set of elliptic curve domain parameters that is the fixed set known as P-192 in [11] and secp192r1 in [19]. The entries p, r, b, x and y represented in that table are example values for a curve that MAY be used with the `C_EC_192` table. These values are set as the default values for the associated columns when a new row is created in the `C_EC_192` table and when values for those columns are not specified at table creation. These default values are not represented by a `Type` table entry; the TPer SHALL be required to keep track of these values and set them as defaults for new objects, as necessary.

Table 190 FIPS P-192 Values for C_EC_192

Column	Value
p	FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFFFF FFFFFFFF = $2^{192} – 2^{64} – 1$
r	FFFFFFFF FFFFFFFF FFFFFFFF 99DEF836 146BC9B1 B4D22831
b	64210519 E59C80E7 0FA7E9AB 72243049 FEB8DEEC C146B9B1
x	188DA80E B03090F6 7CBF20EB 43A18800 F4FF0AFD 82FF1012
y	07192B95 FFC8DA78 631011ED 6B24CDD5 73F977A1 1E794811

5.3.2.19 Credential Table Group - C_EC_224 (Object Table)

Table 191 C_EC_224 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	p		uinteger_28
0x04	r		uinteger_28
0x05	b		uinteger_28
0x06	x		uinteger_28
0x07	y		uinteger_28
0x08	alpha		uinteger_28
0x09	u		uinteger_28
0x0A	v		uinteger_28
0x0B	Hash		hash_protocol
0x0C	ChainLimit		integer_1
0x0D	Certificate		Certificates_object_ref

5.3.2.19.1 UID

This is the unique identifier of this row in the C_EC_224 table.

This column SHALL NOT be modifiable by the host.

5.3.2.19.2 Name

This is the name of the C_EC_224 object.

For C_EC_224 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.19.3 CommonName

This is a name that MAY be shared by multiple C_EC_224 objects.

For C_EC_224 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.19.4 p

Modulus

5.3.2.19.5 r

Order of the curve

5.3.2.19.6 b

Curve coefficient ($y^2=x^3–3x+b \bmod p$)

5.3.2.19.7 x

Base point x-coordinate

5.3.2.19.8 y

Base point y-coordinate

5.3.2.19.9 alpha

Private key

5.3.2.19.10 u

Public key x-coordinate: (u, v) = α (x,y)

5.3.2.19.11 v

Public key y-coordinate: (u, v) = α (x,y)

5.3.2.19.12 Hash

The value of this column identifies the hash type used for ECDSA (message digesting), for ECDH and ECMQV (key derivation), and for creation of the MAC of session startup methods if a referencing authority has a `HashAndSign` column value of True.

5.3.2.19.13 ChainLimit

This column value identifies the chaining limit for using a chained down key from the base certificate. A value of –1 indicates no limit. A value of 0 indicates no chain.

5.3.2.19.14 Certificate

This is a reference to a `Certificates` object, which if needed identifies a chained set of unencoded X.509 certificates to prove an ancestor authority.

5.3.2.19.15 Values for C_EC_224

Table 192 represents the set of elliptic curve domain parameters that is the fixed set known as P-224 in [11] and secp224r1 in [19]. The entries p, r, b, x and y represented in that table are example values for a curve that MAY be used with the `C_EC_224` table. These values are set as the default values for the associated columns when a new row is created in the `C_EC_224` table and when values for those columns are not specified at table creation. These default values are not represented by a `Type` table entry – the TPer SHALL be required to keep track of these values and set them as defaults for new objects, as necessary.

Table 192 FIPS P-224 Values for C_EC_224

Column	Value
p	FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 00000000 00000000 00000001 = $2^{224} - 2^{96} + 1$
r	FFFFFFFF FFFFFFFF FFFFFFFF FFFF16A2 E0B8F03E 13DD2945 5C5C2A3D
b	B4050A85 0C04B3AB F5413256 5044B0B7 D7BFD8BA 270B3943 2355FFB4
x	B70E0CBD 6BB4BF7F 321390B9 4A03C1D3 56C21122 343280D6 115C1D21
y	BD376388 B5F723FB 4C22DFE6 CD4375A0 5A074764 44D58199 85007E34

5.3.2.20 Credential Table Group - C_EC_256 (Object Table)

Table 193 C_EC_256 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	p		uinteger_36
0x04	r		uinteger_36
0x05	b		uinteger_36
0x06	x		uinteger_36
0x07	y		uinteger_36
0x08	alpha		uinteger_36
0x09	u		uinteger_36
0x0A	v		uinteger_36
0x0B	Hash		hash_protocol
0x0C	ChainLimit		integer_1
0x0D	Certificate		Certificates_object_ref

5.3.2.20.1 UID

This is the unique identifier of this row in the `C_EC_256` table.

This column SHALL NOT be modifiable by the host.

5.3.2.20.2 Name

This is the name of the `C_EC_256` object.

For `C_EC_256` objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.20.3 CommonName

This is a name that MAY be shared by multiple `C_EC_256` objects.

For `C_EC_256` objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.20.4 p

Modulus

5.3.2.20.5 r

Order of the curve

5.3.2.20.6 b

Curve coefficient ($y^2=x^3–3x+b \bmod p$)

5.3.2.20.7 x

Base point x-coordinate

5.3.2.20.8 y

Base point y-coordinate

5.3.2.20.9 alpha

Private key

5.3.2.20.10 u

Public key x-coordinate: (u, v) = α (x,y)

5.3.2.20.11 v

Public key y-coordinate: (u, v) = α (x,y)

5.3.2.20.12 Hash

The value of this column identifies the hash type used for ECDSA (message digesting), for ECDH and ECMQV (key derivation), and for creation of the MAC of session startup methods if a referencing authority has a `HashAndSign` column value of True.

5.3.2.20.13 ChainLimit

This column value identifies the chaining limit for using a chained down key from the base certificate. A value of –1 indicates no limit. A value of 0 indicates no chain.

5.3.2.20.14 Certificate

This is a reference to a `Certificates` object, which if needed identifies a chained set of unencoded X.509 certificates to prove an ancestor authority.

5.3.2.20.15 Values for C_EC_256

Table 194 represents the set of elliptic curve domain parameters that is the fixed set known as P-256 in [11] and secp256r1 in [19]. The entries p, r, b, x and y represented in that table are example values for a curve that MAY be used with the `C_EC_256` table. These values are set as the default values for the associated columns when a new row is created in the `C_EC_256` table and when values for those columns are not specified at table creation. These default values are not represented by a `Type` table entry; the TPer SHALL be required to keep track of these values and set them as defaults for new objects, as necessary.

Table 194 FIPS P-256 Values for C_EC_256

Column	Value
p	FFFFFFFF 00000001 00000000 00000000 00000000 FFFFFFFF FFFFFFFF FFFFFFFF = $2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
r	FFFFFFFF 00000000 FFFFFFFF FFFFFFFF BCE6FAAD A7179E84 F3B9CAC2 FC632551
b	5AC635D8 AA3A93E7 B3EBBD55 769886BC 651D06B0 CC53B0F6 3BCE3C3E 27D2604B
x	6B17D1F2 E12C4247 F8BCE6E5 63A440F2 77037D81 2DEB33A0 F4A13945 D898C296
y	4FE342E2 FE1A7F9B 8EE7EB4A 7C0F9E16 2BCE3357 6B315ECE CBB64068 37BF51F5

5.3.2.21 Credential Table Group - C_EC_384 (Object Table)

Table 195 C_EC_384 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	p		uinteger_48
0x04	r		uinteger_48
0x05	b		uinteger_48
0x06	x		uinteger_48
0x07	y		uinteger_48
0x08	alpha		uinteger_48
0x09	u		uinteger_48
0x0A	v		uinteger_48
0x0B	Hash		hash_protocol
0x0C	ChainLimit		integer_1
0x0D	Certificate		Certificates_object_ref

5.3.2.21.1 UID

This is the unique identifier of this row in the C_EC_384 table.

This column SHALL NOT be modifiable by the host.

5.3.2.21.2 Name

This is the name of the C_EC_384 object.

For C_EC_384 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.21.3 CommonName

This is a name that MAY be shared by multiple C_EC_384 objects.

For C_EC_384 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.21.4 p

Modulus

5.3.2.21.5 r

Order of the curve

5.3.2.21.6 b

Curve coefficient ($y^2=x^3–3x+b \bmod p$)

5.3.2.21.7 x

Base point x-coordinate

5.3.2.21.8 y

Base point y-coordinate

5.3.2.21.9 alpha

Private key

5.3.2.21.10 u

Public key x-coordinate: (u, v) = α (x,y)

5.3.2.21.11 v

Public key y-coordinate: (u, v) = α (x,y)

5.3.2.21.12 Hash

The value of this column identifies the hash type used for ECDSA (message digesting), for ECDH and ECMQV (key derivation), and for creation of the MAC of session startup methods if a referencing authority has a `HashAndSign` column value of True.

5.3.2.21.13 ChainLimit

This column value identifies the chaining limit for using a chained down key from the base certificate. A value of –1 indicates no limit. A value of 0 indicates no chain.

5.3.2.21.14 Certificate

This is a reference to a `Certificates` object, which if needed identifies a chained set of unencoded X.509 certificates to prove an ancestor authority.

5.3.2.21.15 Values for C_EC_384

Table 196 represents the set of elliptic curve domain parameters that is the fixed set known as P-384 in [11] and secp384r1 in [19]. The entries p, r, b, x and y represented in that table are example values for a curve that MAY be used with the `C_EC_384` table. These values are set as the default values for the associated columns when a new row is created in the `C_EC_384` table and when values for those columns are not specified at table creation. These default values are not represented by a `Type` table entry; the TPer SHALL be required to keep track of these values and set them as defaults for new objects, as necessary.

Table 196 FIPS P-384 Values for C_EC_384

Column	Value
p	FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFFFF 00000000 00000000 FFFFFFFF = $2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
r	FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF C7634D81 F4372DDF 581A0DB2 48B0A77A ECEC196A CCC52973
b	B3312FA7 E23EE7E4 988E056B E3F82D19 181D9C6E FE814112 0314088F 5013875A C656398D 8A2ED19D 2A85C8ED D3EC2AEF
x	AA87CA22 BE8B0537 8EB1C71E F320AD74 6E1D3B62 8BA79B98 59F741E0 82542A38 5502F25D BF55296C 3A545E38 72760AB7
y	3617DE4A 96262C6F 5D9E98BF 9292DC29 F8F41DBD 289A147C E9DA3113 B5F0B8C0 0A60B1CE 1D7E819D 7A431D7C 90EA0E5F

5.3.2.22 Credential Table Group - C_EC_521 (Object Table)

Table 197 C_EC_521 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	p		uinteger_66
0x04	r		uinteger_66
0x05	b		uinteger_66
0x06	x		uinteger_66
0x07	y		uinteger_66
0x08	alpha		uinteger_66
0x09	u		uinteger_66
0x0A	v		uinteger_66
0x0B	Hash		hash_protocol
0x0C	ChainLimit		integer_1
0x0D	Certificate		Certificates_object_ref

5.3.2.22.1 UID

This is the unique identifier of this row in the `C_EC_521` table.

This column SHALL NOT be modifiable by the host.

5.3.2.22.2 Name

This is the name of the `C_EC_521` object.

For `C_EC_521` objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.22.3 CommonName

This is a name that MAY be shared by multiple `C_EC_521` objects.

For `C_EC_521` objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.22.4 p

Modulus

5.3.2.22.5 r

Order of the curve

5.3.2.22.6 b

Curve coefficient ($y^2=x^3–3x+b \bmod p$)

5.3.2.22.7 x

Base point x-coordinate

5.3.2.22.8 y

Base point y-coordinate

5.3.2.22.9 alpha

Private key

5.3.2.22.10 u

Public key x-coordinate: (u, v) = α (x,y)

5.3.2.22.11 v

Public key y-coordinate: (u, v) = α (x,y)

5.3.2.22.12 Hash

The value of this column identifies the hash type used for ECDSA (message digesting), for ECDH and ECMQV (key derivation), and for creation of the MAC of session startup methods if a referencing authority has a `HashAndSign` column value of True.

5.3.2.22.13 ChainLimit

This column value identifies the chaining limit for using a chained down key from the base certificate. A value of –1 indicates no limit. A value of 0 indicates no chain.

5.3.2.22.14 Certificate

This is a reference to a `Certificates` object, which if needed identifies a chained set of unencoded X.509 certificates to prove an ancestor authority.

5.3.2.22.15 Values for C_EC_521

Table 198 represents the set of elliptic curve domain parameters is the fixed set known as P-521 in [11] and secp521r1 in [19]. The entries p, r, b, x and y represented in that table are example values for a curve that MAY be used with the `C_EC_521` table. These values are set as the default values for the associated columns when a new row is created in the `C_EC_521` table and when values for those columns are not specified at table creation. These default values are not represented by a `Type` table entry; the TPer SHALL be required to keep track of these values and set them as defaults for new objects, as necessary.

Table 198 FIPS P-521 Values for C_EC_521

Column	Value
p	01FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF = 2^{521} - 1
r	01FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFA 51868783 BF2F966B 7FCC0148 F709A5D0 3BB5C9B8 899C47AE BB6FB71E 91386409
b	0051 953EB961 8E1C9A1F 929A21A0 B68540EE A2DA725B 99B315F3 B8B48991 8EF109E1 56193951 EC7E937B 1652C0BD 3BB1BF07 3573DF88 3D2C34F1 EF451FD4 6B503F00
x	00C6 858E06B7 0404E9CD 9E3ECB66 2395B442 9C648139 053FB521 F828AF60 6B4D3DBA A14B5E77 EFE75928 FE1DC127 A2FFA8DE 3348B3C1 856A429B F97E7E31 C2E5BD66
y	0118 39296A78 9A3BC004 5C8A5FB4 2C7D1BD9 98F54449 579B4468 17AFBD17 273E662C 97EE7299 5EF42640 C550B901 3FAD0761 353C7086 A272C240 88BE9476 9FD16650

5.3.2.23 Credential Table Group - C_EC_163 (Object Table)

Table 199 C_EC_163 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	k1		uinteger_1
0x04	k2		uinteger_1
0x05	k3		uinteger_1
0x06	r		uinteger_21
0x07	a		uinteger_1
0x08	b		uinteger_21
0x09	x		uinteger_21
0x0A	y		uinteger_21
0x0B	alpha		uinteger_21
0x0C	u		uinteger_21
0x0D	v		uinteger_21
0x0E	Hash		hash_protocol
0x0F	ChainLimit		integer_1
0x10	Certificate		Certificates_object_ref

5.3.2.23.1 UID

This is the unique identifier of this row in the `C_EC_163` table.

This column SHALL NOT be modifiable by the host.

5.3.2.23.2 Name

This is the name of the `C_EC_163` object.

For `C_EC_163` objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.23.3 CommonName

This is a name that MAY be shared by multiple `C_EC_163` objects.

For `C_EC_163` objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.23.4 k1

High non-leading, non-constant term of irreducible pentanomial

5.3.2.23.5 k2

Middle non-leading, non-constant term of irreducible pentanomial

5.3.2.23.6 k3

Low non-leading, non-constant term of irreducible pentanomial

5.3.2.23.7 r

Order of the curve

5.3.2.23.8 a

Curve coefficient ($y^2 + xy = x^3 + ax^2 + b$), SHALL be zero or one

5.3.2.23.9 b

Curve coefficient ($y^2 + xy = x^3 + ax^2 + b$)

5.3.2.23.10 x

Base point x-coordinate

5.3.2.23.11 y

Base point y-coordinate

5.3.2.23.12 alpha

Private key

5.3.2.23.13 u

Public key x-coordinate: (u, v) = α (x,y)

5.3.2.23.14 v

Public key y-coordinate: (u, v) = α (x,y)

5.3.2.23.15 Hash

The value of this column identifies the hash type used for ECDSA (message digesting), for ECDH and ECMQV (key derivation), and for creation of the MAC of session startup methods if a referencing authority has a `HashAndSign` column value of True.

5.3.2.23.16 ChainLimit

This column value identifies the chaining limit for using a chained down key from the base certificate. A value of –1 indicates no limit. A value of 0 indicates no chain.

5.3.2.23.17 Certificate

This is a reference to a `Certificates` object, which if needed identifies a chained set of unencoded X.509 certificates to prove an ancestor authority.

5.3.2.23.18 Values for C_EC_163

Table 200 represents the set of elliptic curve domain parameters that is the fixed set known as K-163 in [11] and sect163k1 in [19]. The entries k1, k2, k3, r, a, b, x and y represented in that table are example values for a curve that MAY be used with the `C_EC_163` table. These values are set as the default values for the associated columns when a new row is created in the `C_EC_163` table and when values for those columns are not specified at table creation. These default values are not represented by a `Type` table entry; the TPer SHALL be required to keep track of these values and set them as defaults for new objects, as necessary.

Table 200 FIPS K-163 Values for C_EC_163

Column	Value
k1	07
k2	06
k3	03
r	04 00000000 00000000 00020108 A2E0CC0D 99F8A5EF
a	01
b	00 00000000 00000000 00000000 00000000 00000001
x	02 FE13C053 7BBC11AC AA07D793 DE4E6D5E 5C94EEE8
y	02 89070FB0 5D38FF58 321F2E80 0536D538 CCDAA3D9

5.3.2.24 Credential Table Group - C_EC_233 (Object Table)

Table 201 C_EC_233 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	k		uinteger_2
0x04	r		uinteger_30
0x05	a		uinteger_1
0x06	b		uinteger_30
0x07	x		uinteger_30
0x08	y		uinteger_30
0x09	alpha		uinteger_30
0x0A	u		uinteger_30
0x0B	v		uinteger_30
0x0C	Hash		hash_protocol
0x0D	ChainLimit		integer_1
0x0E	Certificate		Certificates_object_ref

5.3.2.24.1 UID

This is the unique identifier of this row in the C_EC_233 table.

This column SHALL NOT be modifiable by the host.

5.3.2.24.2 Name

This is the name of the C_EC_233 object.

For C_EC_233 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.24.3 CommonName

This is a name that MAY be shared by multiple C_EC_233 objects.

For C_EC_233 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.24.4 k

Non-leading, non-constant term of irreducible trinomial

5.3.2.24.5 r

Order of the curve

5.3.2.24.6 a

Curve coefficient ($y^2 + xy = x^3 + ax^2 + b$), SHALL be zero or one

5.3.2.24.7 b

Curve coefficient ($y^2 + xy = x^3 + ax^2 + b$)

5.3.2.24.8 x

Base point x-coordinate

5.3.2.24.9 y

Base point y-coordinate

5.3.2.24.10 alpha

Private key

5.3.2.24.11 u

Public key x-coordinate: (u, v) = α (x,y)

5.3.2.24.12 v

Public key y-coordinate: (u, v) = α (x,y)

5.3.2.24.13 Hash

The value of this column identifies the hash type used for ECDSA (message digesting), for ECDH and ECMQV (key derivation), and for creation of the MAC of session startup methods if a referencing authority has a HashAndSign column value of True.

5.3.2.24.14 ChainLimit

This column value identifies the chaining limit for using a chained down key from the base certificate. A value of –1 indicates no limit. A value of 0 indicates no chain.

5.3.2.24.15 Certificate

This is a reference to a Certificates object, which if needed identifies a chained set of unencoded X.509 certificates to prove an ancestor authority.

5.3.2.24.16 Values for C_EC_233

Table 202 represents the set of elliptic curve domain parameters that is the fixed set known as K-233 in [11] and sect233k1 in [19]. The entries k, r, a, b, x and y represented in that table are example values for a curve that MAY be used with the C_EC_233 table. These values are set as the default values for the associated columns when a new row is created in the C_EC_233 table and when values for those columns are not specified at table creation. These default values are not represented by a Type table entry; the TPer SHALL be required to keep track of these values and set them as defaults for new objects, as necessary.

Table 202 FIPS K-233 Values for C_EC_233

Column	Value
k	4A (= 74 in decimal)
r	0080 00000000 00000000 00000000 00069D5B B915BCD4 6EFB1AD5 F173ABDF
a	00
b	0000 00000000 00000000 00000000 00000000 00000000 00000000 00000001
x	0172 32BA853A 7E731AF1 29F22FF4 149563A4 19C26BF5 0A4C9D6E EFAD6126
y	01DB 537DECE8 19B7F70F 555A67C4 27A8CD9B F18AEB9B 56E0C110 56FAE6A3

5.3.2.25 Credential Table Group - C_EC_283 (Object Table)

Table 203 C_EC_283 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	k1		uinteger_1
0x04	k2		uinteger_1
0x05	k3		uinteger_1
0x06	r		uinteger_36
0x07	a		uinteger_1
0x08	b		uinteger_36
0x09	x		uinteger_36
0x0A	y		uinteger_36
0x0B	alpha		uinteger_36
0x0C	u		uinteger_36
0x0D	v		uinteger_36
0x0E	Hash		hash_protocol
0x0F	ChainLimit		integer_1
0x10	Certificate		Certificates_object_ref

5.3.2.25.1 UID

This is the unique identifier of this row in the C_EC_283 table.

This column SHALL NOT be modifiable by the host.

5.3.2.25.2 Name

This is the name of the C_EC_283 object.

For C_EC_283 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.25.3 CommonName

This is a name that MAY be shared by multiple C_EC_283 objects.

For C_EC_283 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.25.4 k1

High non-leading, non-constant term of irreducible pentanomial

5.3.2.25.5 k2

Middle non-leading, non-constant term of irreducible pentanomial

5.3.2.25.6 k3

Low non-leading, non-constant term of irreducible pentanomial

5.3.2.25.7 r

Order of the curve

5.3.2.25.8 a

Curve coefficient ($y^2 + xy = x^3 + ax^2 + b$), SHALL be zero or one

5.3.2.25.9 b

Curve coefficient ($y^2 + xy = x^3 + ax^2 + b$)

5.3.2.25.10 x

Base point x-coordinate

5.3.2.25.11 y

Base point y-coordinate

5.3.2.25.12 alpha

Private key

5.3.2.25.13 u

Public key x-coordinate: $(u, v) = \alpha (x,y)$

5.3.2.25.14 v

Public key y-coordinate: $(u, v) = \alpha (x,y)$

5.3.2.25.15 Hash

The value of this column identifies the hash type used for ECDSA (message digesting), for ECDH and ECMQV (key derivation), and for creation of the MAC of session startup methods if a referencing authority has a HashAndSign column value of True.

5.3.2.25.16 ChainLimit

This column value identifies the chaining limit for using a chained down key from the base certificate. A value of –1 indicates no limit. A value of 0 indicates no chain.

5.3.2.25.17 Certificate

This is a reference to a Certificates object, which if needed identifies a chained set of unencoded X.509 certificates to prove an ancestor authority.

5.3.2.25.18 Values for C_EC_283

Table 204 represents the set of elliptic curve domain parameters that is the fixed set known as K-283 in [11] and sect283k1 in [19]. The entries k1, k2, k3, r, a, b, x and y represented in that table are example values for a curve that MAY be used with the C_EC_283 table. These values are set as the default values for the associated columns when a new row is created in the C_EC_283 table and when values for those columns are not specified at table creation. These default values are not represented by a Type table entry; the TPer SHALL be required to keep track of these values and set them as defaults for new objects, as necessary.

Table 204 FIPS K-283 Values for C_EC_283

Column	Value
k1	0C (= 12 in decimal)
k2	07
k3	05
r	01FFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFE9AE 2ED07577 265DFF7F 94451E06 1E163C61
a	00
b	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001
x	0503213F 78CA4488 3F1A3B81 62F188E5 53CD265F 23C1567A 16876913 B0C2AC24 58492836
y	01CCDA38 0F1C9E31 8D90F95D 07E5426F E87E45C0 E8184698 E4596236 4E341161 77DD2259

5.3.2.26 Credential Table Group – C_HMAC_160 (Object Table)

Table 205 C_HMAC_160 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	Key		bytes_20
0x04	Hash		hash_protocol

5.3.2.26.1 UID

This is the unique identifier of this row in the C_HMAC_160 table.

This column SHALL NOT be modifiable by the host.

5.3.2.26.2 Name

This is the name of the C_HMAC_160 object.

For C_HMAC_160 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.26.3 CommonName

This is a name that MAY be shared by multiple C_HMAC_160 objects.

For C_HMAC_160 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.26.4 Key

The value of the Key column of this table holds key material to be used with an HMAC authentication operation, or a host-invoked HMAC operation as enabled by the Crypto Template.

5.3.2.26.5 Hash

The value of the Hash column identifies the hash protocol to be used with this HMAC credential when this credential is referenced by an authority and used for authentication. The value of this column is ignored for host-invoked HMAC operations.

See [13] for details on matching key size to hash protocol selection.

5.3.2.27 Credential Table Group – C_HMAC_256 (Object Table)

Table 206 C_HMAC_256 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	Key		bytes_32
0x04	Hash		hash_protocol

5.3.2.27.1 UID

This is the unique identifier of this row in the C_HMAC_256 table.

This column SHALL NOT be modifiable by the host.

5.3.2.27.2 Name

This is the name of the C_HMAC_256 object.

For C_HMAC_256 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.27.3 CommonName

This is a name that MAY be shared by multiple C_HMAC_256 objects.

For C_HMAC_256 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.27.4 Key

The value of the Key column of this table holds key material to be used with an HMAC authentication operation, or a host-invoked HMAC operation as enabled by the Crypto Template.

5.3.2.27.5 Hash

The value of the Hash column identifies the hash protocol to be used with this HMAC credential when this credential is referenced by an authority and used for authentication. The value of this column is ignored for host-invoked HMAC operations.

See [13] for details on matching key size to hash protocol selection.

5.3.2.28 Credential Table Group – C_HMAC_384 (Object Table)

Table 207 C_HMAC_384 Table Description

Column Number	Column Name	IsUnique	Column Type

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	Key		bytes_48
0x04	Hash		hash_protocol

5.3.2.28.1 UID

This is the unique identifier of this row in the C_HMAC_384 table.

This column SHALL NOT be modifiable by the host.

5.3.2.28.2 Name

This is the name of the C_HMAC_384 object.

For C_HMAC_384 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.28.3 CommonName

This is a name that MAY be shared by multiple C_HMAC_384 objects.

For C_HMAC_384 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.28.4 Key

The value of the Key column of this table holds key material to be used with an HMAC authentication operation, or a host-invoked HMAC operation as enabled by the Crypto Template.

5.3.2.28.5 Hash

The value of the Hash column identifies the hash protocol to be used with this HMAC credential when this credential is referenced by an authority and used for authentication. The value of this column is ignored for host-invoked HMAC operations.

See [13] for details on matching key size to hash protocol selection.

5.3.2.29 Credential Table Group – C_HMAC_512 (Object Table)

Table 208 C_HMAC_512 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	Key		bytes_64
0x04	Hash		hash_protocol

5.3.2.29.1 UID

This is the unique identifier of this row in the C_HMAC_512 table.

This column SHALL NOT be modifiable by the host.

5.3.2.29.2 Name

This is the name of the C_HMAC_512 object.

For C_HMAC_512 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.29.3 CommonName

This is a name that MAY be shared by multiple C_HMAC_512 objects.

For C_HMAC_512 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.3.2.29.4 Key

The value of the Key column of this table holds key material to be used with an HMAC authentication operation, or a host-invoked HMAC operation as enabled by the Crypto Template.

5.3.2.29.5 Hash

The value of the Hash column identifies the hash protocol to be used with this HMAC credential when this credential is referenced by an authority and used for authentication. The value of this column is ignored for host-invoked HMAC operations.

See [13] for details on matching key size to hash protocol selection.

5.3.3 Methods

This section details the methods provided to an SP by the Base Template.

5.3.3.1 SP Method Group - DeleteSP (SP Method)

This method is used to delete the SP to which the DeleteSP method has been invoked (see 5.3.4.4).

```
ThisSP.DeleteSP[]
=>
[  ]
```

5.3.3.1.1 DeleteSP Result

5.3.3.1.1.1 Result

- The DeleteSP method returns an empty list. Success or failure of the method invocation is determinable based on the status code returned in response to the method invocation.

5.3.3.2 Basic Table Method Group - CreateTable (SP Method)

This method is used to create a new table in an SP (see 5.3.4.2.1).

```
ThisSP.CreateTable [
       NewTableName : name,
       Kind : table_kind,
       GetSetACL : access_control_list,
       Columns : columns,
       MinSize : uinteger,
       MaxSize = uinteger,
       HintSize = uinteger,
       CommonName = name]
=>
[ UID : uid, Rows : uinteger ]
```

5.3.3.2.1 NewTableName

The NewTableName parameter is the name for this table. The NewTableName-CommonName combination SHALL be unique within the Table table.

5.3.3.2.2 Kind

The Kind parameter identifies the table's type (object or byte).

5.3.3.2.3 GetSetACL

GetSetACL is the list of ACE object uids placed in the AddACEACL, RemoveACEACL, GetACLACL, and DeleteMethodACL columns of the AccessControl table rows that represent the methods available on the new table (see 5.3.4.2.3).

5.3.3.2.4 Columns

The Columns parameter defines the columns of the new table. For byte tables this parameter SHALL be an empty list.

5.3.3.2.5 MinSize

The MinSize parameter is used to define the initial number of rows allocated for the new table.

5.3.3.2.6 MaxSize

The optional MaxSize parameter defines the host-requested maximum number of rows that MAY be created for the table. If this parameter is included when creating a byte table, the method invocation SHALL fail with INVALID_PARAMETER.

5.3.3.2.7 HintSize

The optional HintSize parameter is used to suggest a number of rows to be created for the table. If this parameter is included when creating a byte table, the method invocation SHALL fail with INVALID_PARAMETER.

5.3.3.2.8 CommonName

The CommonName parameter is the CommonName column value for this table's object in the Table table, as well as for all associated objects that get created in the Column table.. The NewTableName-CommonName parameter value combination SHALL be unique within the Table table.

5.3.3.2.9 CreateTable Result

5.3.3.2.9.1 UID

This is the UID column value that is assigned to the newly created table in the Table table.

5.3.3.2.9.2 Rows

This value is the number of rows allocated for usage for the table.

5.3.3.2.10 Fails

a. If a table with the specified Name/CommonName column values already exists.
b. If there isn't space in the SP for the new table.
c. If metadata/support tables (i.e. Table, Column, AccessControl, and ACE) are not all able to create all required rows to support this table.

5.3.3.3 Basic Table Method Group - Delete (Object Method)

Successful invocation of this method deletes the object upon which this method was invoked. See 5.3.4.2.4 for information on deleting table rows and 5.3.4.2.5 for information on deleting tables.

If invoked on an SP object (a row in the Admin SP's SP table), the SP is deleted (see 5.4.4.2).

```
ObjectUID.Delete[ ]
=>
[   ]
```

5.3.3.3.1 Delete Result

5.3.3.3.1.1 Result

The Delete method returns an empty list. Success or failure of the method invocation is determinable based on the status code returned in response to the method invocation. The object and all of its associated objects in other tables SHALL be deleted, or the method SHALL return FAIL status and none of those items SHALL be deleted.

5.3.3.3.2 Fails

a. If the object does not exist.

5.3.3.4 Basic Table Method Group - CreateRow (Table Method)

This method inserts one or multiple rows into a table. This method is not available on byte tables. The list of uidrefs returned is the list of all UIDs of the rows created (see 5.3.4.2.3).

```
TableUID.CreateRow [
       Row : row_data ]
=>
[ Result : list [ uidref ... ] ]
```

5.3.3.4.1 Row

The Row parameter identifies the values to be stored in the columns indicated in the parameter for each row created.

5.3.3.4.2 CreateRow Result

5.3.3.4.2.1 Result

The result of the CreateRow method is a list containing the UID column values assigned to each of the newly created rows in the table.

5.3.3.4.3 Fails

a. When the table is full (i.e. MaxSize of the table was reached).
b. If a row where the unique column value combination already exists that is the same as that requested in the method
c. Columns specified are not part of table definition.
d. Attempts to create more rows than are able to be allocated
e. If all required associated rows are not able to be created in all related tables (i.e. the Table, AccessControl, Column, and ACE tables)

5.3.3.5 Basic Table Method Group - DeleteRow (Table Method)

This method is used to delete table rows. This method SHALL NOT be able to be successfully invoked on byte tables. See 5.3.4.2.4 for information on deleting table rows and 5.3.4.2.5 for information on deleting tables.

```
TableUID.DeleteRow [
       Rows : list [ uidref ... ] ]
=>
[ Result : boolean ]
```

5.3.3.5.1 Rows

The Rows parameter consists of a list of uids that represent each of the rows to be deleted from the table.

5.3.3.5.2 DeleteRow Result

5.3.3.5.2.1 Result

The `DeleteRow` method returns an empty list. Success or failure of the method invocation is determinable based on the status code returned in response to the method invocation. The object and all of its associated objects in other tables SHALL be deleted, or the method SHALL return FAIL status and none of those items SHALL be deleted.

5.3.3.5.3 Fails

a. If the addressed row does not exist.

5.3.3.6 Basic Table Method Group - Get (Table and Object Method)

This method is used to fetch the values of selected table cells (see 5.3.4.2.2).

```
TableUID.Get [
ObjectUID.Get [
       Cellblock : cell_block ]
=>
[ Result : typeOr { Bytes : bytes, RowValues : list [ ColumnNumber = Value ... ] } ]
```

5.3.3.6.1 Cellblock

The Cellblock parameter defines the scope of the data that the method is attempting to retrieve by identifying the cells on which the method should operate.

5.3.3.6.2 Get Result

5.3.3.6.2.1 Bytes

This is the value returned if the method is invoked on a byte table. If multiple row values are returned from a byte table, values SHALL be returned from the lowest numbered row to the highest numbered row.

5.3.3.6.2.2 RowValues

This value is returned if the method is invoked on an object table. This is a list of Named value pairs representing the columns returned for the object identified in the method invocation. Each Named value pair consists of a name that represents the column, identified by column number. The value of each Named value pair is the value of the indicated column.

Column name-value pairs SHALL be returned in the order in which they are listed in the `Column` table.

5.3.3.6.3 Fails

a. If table/object doesn't exist.
b. If the object method's Cellblock parameter contains row values or a table value.
c. If the method is invoked on a Byte table and has column values in the Cellblock parameter.
d. If the any of the Cellblock parameter values are out of bounds for the table upon which it was invoked.

5.3.3.7 Basic Table Method Group - Set (Table and Object Method)

This method is used to change the values of selected table cells (see 5.3.4.2.6).

```
TableUID.Set [
ObjectUID.Set [
       Where = row_address,
       Values = typeOr { Bytes : bytes, RowValues : list [ ColumnNumber = <type of
column> ... ] } ]
=>
[ ]
```

5.3.3.7.1 Where

This parameter identifies the location of the cells whose values the method is attempting to change.

If the `Set` method is invoked on an object, the Where parameter SHALL be omitted, or the method SHALL fail and return an error status code.

If the `Set` method is invoked on an object table, the Where parameter SHALL be the UID option, or the method SHALL fail and return an error status code.

If the `Set` method is invoked on a byte table and the Where parameter is included in the invocation, the Where parameter SHALL be the Row option, or the method SHALL fail and return an error status code.

5.3.3.7.1.1 UID

For `Object.Set`, if a value for the Where parameter is included in the method invocation, the method SHALL fail and return an error status code.

For `Table.Set` on an object table, "Where = { UID }" indicates the row upon which the operation is taking place. Invocations of `Table.Set` on an object table without a value for the Where parameter SHALL fail and return an error status code.

5.3.3.7.1.2 Row

For `Table.Set` on byte tables, "Where = { Row }" identifies the byte address (i.e. RowNumber) where the `Set` method is to begin operating.

If the Where parameter is omitted, the `Set` method's operation begins at the first row of the byte table.

5.3.3.7.2 Values

This parameter contains the values to be set to the indicated table cells.

If the Set method is invoked on an object or an object table, the Values parameter SHALL be the RowValues option, or the method SHALL fail and return an error status code.

If the Set method is invoked on a byte table, the Values parameter SHALL be the Bytes option, or the method SHALL fail and return an error status code.

Since the Values parameter is an optional parameter, excluding the parameter from a `Set` method invocation SHALL NOT cause the method to fail. An otherwise correct invocation of the `Set` method that does not contain the Values parameter SHALL succeed but have no effect.

5.3.3.7.2.1 Bytes

When this method is invoked on a byte table, this parameter is used. It is a bytes value, used when attempting to modify the values in a byte table, The byte table is modified beginning at the byte address indicated in the Where parameter, or at the beginning of the table if the Where parameter is omitted.

If the Where parameter is a Row, the Values parameter SHALL be Bytes or the method invocation SHALL fail and return an error status code.

5.3.3.7.2.2 RowValues

This value is used when attempting to modify an object in an object table. When this method is invoked on an object table, as either an object method or a table method, this parameter is a list of column

numbers and values, where the columns are those to be changed and the associated values are the values to be set to those columns.

If the Where parameter is a UID, the Values parameter shall be RowVals or the method invocation shall fail and return an error status code.

5.3.3.7.3 Set Result

The `Set` method returns an empty list. Success or failure of the requested modifications is determinable based on the status code returned in response to the method invocation.

5.3.3.7.4 Fails

a. If the table/object doesn't exist.
b. If an attempt is made to change the value of an UID or other system cell.
c. If an attempt is made to set a cell to a value larger than that cell's type allows.
d. If the method is invoked on a byte table and the Values parameter contains column values
e. `Set` is restricted by an access control limitation on any of the rows and columns requested.

5.3.3.8 Basic Table Method Group - Next (Table Method)

The `Next` method is used to iterate through an object table, returning that the `UID` column value for zero or more rows in the table based on the current ordering of the rows in the table; the requested starting point for the method's operation; and the number of uids requested. For information on the operation of this method, see 5.3.4.2.7.

```
TableUID.Next [
       Where = uidref,
       Count = uinteger ]
=>
[ Result : list [ uidref ... ] ]
```

5.3.3.8.1 Where

This parameter identifies the row from which iteration begins. If the Where parameter is specified, the `Next` method returns a list of zero or more `UID` column values following the specified row.

If Where is not specified, the first row in the TPer's current ordering of the table SHALL be the first `UID` column value returned.

5.3.3.8.2 Count

This parameter identifies the number of rows through which the method is to iterate, beginning at the row specified in the Where parameter or, if Where is omitted, beginning at the first row of the table.

If the Count parameter is omitted, the method iterates through to the last row in the table's ordering.

5.3.3.8.3 Next Result

5.3.3.8.3.1 Result

The result of this method is a list of `UID` column values.

5.3.3.8.4 Fails

a. If the table/object doesn't exist.

5.3.3.9 Basic Table Method Group - GetFreeSpace (SP Method)

The `GetFreeSpace` method is an SP method that enables the host to retrieve the number of rows that MAY be additionally created in each table.

```
ThisSP.GetFreeSpace [ ]
=>
[ FreeSpace : uinteger, TableRows : table_sizes ]
```

5.3.3.9.1 GetFreeSpace Result

5.3.3.9.1.1 FreeSpace

The FreeSpace result value is the approximate amount of free space (in bytes) available in the SP.

5.3.3.9.1.2 TableRows

The second is a list containing the `UID` column value of each table descriptor object and the number of rows that MAY be additionally created for each table (separately) under current conditions of the SP and the TPer. This number MAY change in subsequent invocations of this method, based on modifications subsequent to the method invocation.

The number of rows returned for a table(s) is not directly related to the free space remaining on the SP. The number of rows is only indicative of how many rows the system is able to generate per table.

5.3.3.10 Basic Table Method Group - GetFreeRows (Object Method)

The `GetFreeRows` method is a table method that enables the host to retrieve the number of rows that MAY be additionally created in a table.

```
TableObjectUID.GetFreeRows [ ]
=>
[ FreeRows : uinteger ]
```

5.3.3.10.1 GetFreeRows Result

5.3.3.10.1.1 FreeRows

The result of this method is the number of rows that MAY be additionally created for that table.

The number of rows returned for a table(s) is not directly related to the free space remaining on the SP. The number of rows is only indicative of how many rows the system is able to generate per table.

5.3.3.10.2 Fails

a. When the table TableObjectUID does not exist in the SP.

5.3.3.11 Method Manipulation Group - DeleteMethod (Meta-Method)

Successful invocation of the `DeleteMethod` method removes the indicated SP/method, table/method, or object/method access control association from the `AccessControl` table.

The `DeleteMethod` method allows the host to prevent the usage of certain methods on certain tables, objects, or the SP by removing the access control association that permits the method to be invoked.

This does not remove the capability of invoking the indicated method from the SP entirely. It only removes the indicated access control association that allows the method to be invoked in that particular fashion.

The association that is deleted from the `AccessControl` table is the row where the `InvokingID` column value is the **InvokingID** parameter of the method, and the value of the `MethodID` column is the uid referenced in the **MethodID** parameter of the `DeleteMethod` invocation. There is no mechanism that enables a deleted access control association to be re-added.

```
AccessControlTableUID.DeleteMethod [
       InvokingID : uidref { SP/table/object },
       MethodID : uidref { MethodID } ]
=>
[   ]
```

5.3.3.11.1 InvokingID

This parameter is the uidref to the SP, table, or object identified in the InvokingID portion of the access control association to be deleted from the `AccessControl` table.

5.3.3.11.2 MethodID

This parameter is the uidref to the method portion of the access control association to be deleted from the `AccessControl` table. This is the uidref of the method object in the `MethodID` table.

5.3.3.11.3 DeleteMethod Result

5.3.3.11.3.1 Result

The `DeleteMethod` method returns an empty list. Success or failure of the method invocation is determinable based on the status code returned in response to the method invocation.

5.3.3.11.4 Fails

a. If the InvokingID/MethodID combination does not exist.
b. If the DeleteMethodACL is not authenticated.

5.3.3.12 Access Control Method Group - Authenticate (SP Method)

Authorities invoked during session startup are implicitly authenticated. The `Authenticate` method is used to explicitly authenticate an authority within a session, i.e., after a session has already successfully begun. See 5.3.4.1.14.

```
ThisSP.Authenticate [
       Authority : uidref { AuthorityObjectUID },
       Proof = bytes ]
=>
[ Result : typeOr { Success : boolean, Challenge : bytes } ]
```

5.3.3.12.1 Authority

This parameter is the `UID` column value of the authority object the host is attempting to authenticate.

5.3.3.12.2 Proof

This is the proof the host is submitting as part of the authentication process.

5.3.3.12.3 Authenticate Result

The `Authenticate` method's result is dependant on the parameterized authority's `Operation` column value and the step in the authentication protocol.

5.3.3.12.3.1 Success

This parameter is used to indicate whether the authority was successfully authenticated. The value returned is True if the authority was successfully authenticated, and False otherwise.

5.3.3.12.3.2 Challenge

This result is returned in response to the method invocation when the method is invoked as the first step of authenticating an authority that requires a challenge response protocol.

5.3.3.12.4 Fails

a. If the authority called out in the method invocation does not exist.

5.3.3.13 Access Control Method Group - GetACL (Meta-Method)

This method is used to retrieve the contents of an access control association's ACL, which are stored in the `AccessControl` table.

```
AccessControlTableUID.GetACL [
       InvokingID : uidref { SP/table/object },
       MethodID : uidref { MethodID } ]
=>
[ Result : access_control_list ]
```

5.3.3.13.1 InvokingID

This parameter is the uidref to the SP, table, or object identified in the InvokingID portion of the access control association to be retrieved from the `AccessControl` table.

5.3.3.13.2 MethodID

This parameter is the uidref to the method portion of the access control association to be retrieved form the `AccessControl` table. This is the uidref of the method object in the `MethodID` table.

5.3.3.13.3 GetACL Result

5.3.3.13.3.1 Result

This method returns a list of uidrefs to `ACE` objects.

5.3.3.13.4 Fails

a. If the InvokingID/MethodID combination does not exist.
b. If the GetACLACL is not authenticated.

5.3.3.14 Access Control Method Group - AddACE (Meta-Method)

This method is used to add an `ACE` object uidref to the `ACL` column of an existing SP/method, table/method, or object/method access control association, which is a row in the `AccessControl` table.

```
AccessControlTableUID.AddACE [
       InvokingID : uidref { SP/table/object },
       MethodID : uidref { MethodID },
ACE : uidref { ACEObjectUID} ]
=>
[   ]
```

5.3.3.14.1 InvokingID

The InvokingID parameter is the uidref to ThisSP (always `0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01`), the table, or the object of the access control association.

5.3.3.14.2 MethodID

The MethodID parameter is the uidref to the method of the access control association. This is the uidref of the method object in the `MethodID` table.

5.3.3.14.3 ACE

The ACE parameter is a uidref to the `ACE` to be added to the `ACL` column of the appropriate `AccessControl` table row, as identified by the InvokingID and MethodID parameters.

5.3.3.14.4 AddACE Result

5.3.3.14.4.1 Result

The AddACE method returns an empty list. Success or failure of the method invocation is determinable based on the status code returned in response to the method invocation.

5.3.3.14.5 Fails

a. If the InvokingID/MethodID combination does not exist.
b. If the ACE does not exist in the ACE table.
c. If the ACE already exists in the ACL of the invoked access control association.
d. If the ACL of the invoked access control association is full.
e. If the AddACEACL is not authenticated.

5.3.3.15 Access Control Method Group - RemoveACE (Meta-Method)

This method is used to remove an ACE object uidref from the ACL column of an existing SP/method, table/method, or object/method access control association, which are rows in the AccessControl table.

```
AccessControlTableUID.RemoveACE [
       InvokingID : uidref { SP/table/object },
       MethodID : uidref { MethodID },
ACE : uidref { ACEObjectUID} ]
=>
[   ]
```

5.3.3.15.1 InvokingID

The InvokingID parameter is the uidref to ThisSP (always 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01), the table, or the object of the access control association.

5.3.3.15.2 MethodID

The MethodID parameter is the uidref to the method of the access control association. This is the uidref of the method object in the MethodID table.

5.3.3.15.3 ACE

The ACE parameter is a uidref to the ACE to be removed from the ACL column of the appropriate AccessControl table row.

5.3.3.15.4 RemoveACE Result

5.3.3.15.4.1 Result

The RemoveACE method returns an empty list. Success or failure of the method invocation is determinable based on the status code returned in response to the method invocation.

5.3.3.15.5 Fails

a. If the InvokingID/MethodID combination does not exist.
b. If the ACE does not exist in the ACE table.
c. If the RemoveACEACL is not authenticated.

5.3.3.16 Key Related Method Group - GenKey (Object Method)

This section describes the method used for key creation.

An existing object in any of the C_AES_*, K_AES_*, C_EC_*, C_PIN, C_RSA_* or C_HMAC_* tables is filled in with new key material This method fills in the new key as appropriate for the type of the credential on which the method was invoked.

```
CredentialObjectUID.GenKey [
       PublicExponent = uinteger,
       PinLength = uinteger ]
=>
[   ]
```

5.3.3.16.1 PublicExponent

This parameter identifies the public exponent to be used when the method is invoked on a C_RSA_1024 or C_RSA_2048 object. If a value is not specified for this parameter, then the keys SHALL be calculated using the public exponent 2^{16}+1 (65537).

5.3.3.16.2 PinLength

If this method is invoked on a C_PIN object, then a new value with PinLength characters is generated and stored in that C_PIN object's Password column. The character set used to generate the C_PIN value is referenced in the C_PIN table's CharacterSet column, or the default character set if the C_PIN table's CharacterSet column is the NULL UID (see C_PIN table description in section 5.3.2.12).

For other behavior of GenKey on C_PIN objects see 5.3.4.1.1.2.

If PinLength is not specified in the method invocation, the default value is 32. The maximum permitted value for the PinLength parameter is 32.

5.3.3.16.3 GenKey Result

5.3.3.16.3.1 Result

The GenKey method returns an empty list. Success or failure of the method invocation is determinable based on the status code returned in response to the method invocation.

5.3.3.16.4 Fails

a. If the credential object does not exist.
b. If a bad exponent is included.
c. If PublicExponent is provided when GenKey is invoked on a non-C_RSA_* credential object.
d. If PinLength is provided when GenKey is invoked on a non-C_PIN credential object
e. If PinLength is greater than the size of the Password column in the C_PIN table.

5.3.3.17 Key Related Method Group - GetPackage Method (Object Method)

The purpose of this method is to retrieve key material from a credential table in a secure manner. In addition to fulfilling the access control restrictions for invoking GetPackage, this method should only succeed if access control is fulfilled for WrappingKey.Encrypt and SigningKey.Sign.

```
CredentialObjectUID.GetPackage [
       Purpose : package_purpose,
       WrappingKey = uidref { CredentialObjectUID },
       SigningKey = uidref { CredentialObjectUID },
       Date = date,
       Log = bytes
=>
[ Result : package ]
```

5.3.3.17.1 Purpose

This parameter defines the host-supplied usage of the package.

5.3.3.17.2 WrappingKey

This is a uidref to the credential used to encrypt the key material in the invoking object. If no wrapping key is supplied, the key material is included in the package in plaintext.

5.3.3.17.3 SigningKey

This is a uidref to the credential used to sign the hash of the package contents. If no signing key is supplied, no hash is included in the package.

5.3.3.17.4 Date

This is an optional parameter with which the host MAY supply a date to be included in the result package

5.3.3.17.5 Log

This is an optional parameter with which the host MAY supply a host-defined value to be included in the result package.

5.3.3.17.6 GetPackage Result

5.3.3.17.6.1 Result

The return result is a “package” (see 5.1.4.2.12).

5.3.3.17.7 Fails

a. If the object does not exist
b. If the WrappingKey is not a valid credential.
c. If the SigningKey is not a valid credential.

5.3.3.18 Key Related Method Group - SetPackage Method (Object Method)

The `SetPackage` method is used to set the key material columns of a credential with key material that is sent securely to the TPer (see 5.3.4.5). For other behavior of `SetPackage` on C_PIN objects see 5.3.4.1.1.2.

```
CredentialObjectUID.SetPackage [
Value : package,
       WrappingKey = uidref { CredentialObjectUID },
       SigningKey = uidref { CredentialObjectUID },
       ]
=>
[ ]
```

5.3.3.18.1 Value

This is a package that is generated as the result of a `GetPackage` method (see 5.3.3.17)

5.3.3.18.2 WrappingKey

This is a uidref to the credential to be used to decrypt the encrypted key material in the parameterized package. This parameter MAY be omitted, if the key material in the parameterized package is being sent in plaintext.

5.3.3.18.3 SigningKey

This is a uidref to the credential to be used to verify the signed hash in the parameterized package. This parameter MAY be omitted, if the package contents have not been hashed and signed.

5.3.3.18.4 SetPackage Result

This method returns an empty results list. Success or failure of the method invocation is determined by the method's accompanying status code.

5.3.3.18.5 Fails

a. If the object does not exist

b. If the WrappingKey is not a valid credential.
c. If the SigningKey is not a valid credential.
d. If the signed hash is not verifiable

5.3.4 Description

5.3.4.1 Authentication

5.3.4.1.1 Credential Tables

Credential tables represent an extensible basis for providing the public and private parts of authentication mechanisms and key stores. Each credential table represents a different mechanism or key store type and each row a different authentication or key using the mechanism or key store represented by the table. The credential tables supported SHALL be listed in the `CryptoSuite` table in the Admin SP.

In the credential table definitions, marked columns (those shaded gray) MAY be hidden. The mechanism(s), if any, used to hide the values stored in those columns are discoverable via the SecretProtect table (see 5.3.2.8).

5.3.4.1.1.1 GenKey on C_PIN Objects

If the value of the `CharSet` column is a NULL UID reference, then the default character set used when creating a new `PIN` column value with the `GenKey` method SHALL be made up of the set of valid ASCII printable characters (see [7]). The default value of the `CharSet` column is zeroes.

If the `CharSet` column value is not zeroes, it SHALL be a uid to a byte table in which SHALL be defined a character set to be used when creating a new `PIN` column value with the `GenKey` method to generate a new `PIN` column value.

A `CharSet` column value of `0x00 0x00 0x00 0x0B 0x00 0x00 0x00 0x01` SHALL indicate that character set is not restricted, and all byte values are legal.

The TPer SHALL NOT enforce the character set when the host sets the PIN column via the Set method.

5.3.4.1.2 Authentication Attempt Limits with C_PIN Objects

A mechanism to limit the number of authentication attempts with a `C_PIN` credential, either during session startup or using the `Authenticate` method, is provided for in the `C_PIN` table through the `Tries` and `TryLimit` columns.

The default value of the `Tries` column when a new `C_PIN` object is created is 0. If the value of the `TryLimit` column is not 0, then the value of the `Tries` column is incremented by the TPer on every failed authentication attempt, including the implicit authentication if the authority is a Signing Authority invoked during session startup.

When the value of the `Tries` column is equal to the value of the `TryLimit` column, and the `TryLimit` column is not equal to 0, further attempts to authenticate using this credential SHALL always fail (until the value of the `Tries` column is reset), but `Tries` SHALL NOT increment beyond `TryLimit`.

The value of the `Tries` column is set to 0 by the TPer upon successful invocation of the `Authenticate` method or implicit session startup authentication of the authority referencing this `C_PIN` object.

The value of the `Tries` column MAY be reset from the host by successful invocation of the `Set` method on that cell to set the value to 0 (access control SHALL be properly fulfilled).

Additionally, the value of the `Tries` column SHALL be reset to 0 after a power cycle if the value of the `Persistence` column is False. Otherwise, the value of the `Tries` column SHALL persist across power cycles. Other TCG resets (i.e. Hardware Reset) SHALL NOT cause the value of the `Tries` column to be reset, even if the value of the `Persistence` column is False.

Successful invocation of methods on a C_PIN object that modify the value of the PIN column also set the value of that object's Tries column to 0. These methods are GenKey, Set, and SetPackage. If TryLimit is 0, there is no limit to the number of Tries, and Tries SHALL remain 0.

The value of the Tries column is not subject to transactional rollback when changed by the TPer. The TPer SHALL be able to set the Tries column value during a Read-Only session, but the host SHALL only be able to set this column during a Read-Write session.

5.3.4.1.2 Authorities

Authorities are made up of two types – Class and Individual.

Class authorities MAY be members of another class authority, but this SHALL NOT be permitted to expand beyond a single level. The TPer SHALL enforce that class authorities are not permitted to be members of a class authority that is already a member of another class authority. Class authorities SHALL NOT reference credentials or secure messaging requirements. Class authorities SHALL NOT be directly authenticated – authentication attempts that reference class authorities SHALL always fail.

Individual authorities MAY be members of class authorities. When an individual authority is authenticated, either from session startup or explicitly via the Authenticate method, the class authority that the individual authority references is considered to be authenticated also. If that class authority also references a class, then the class authority referenced by the initial class authority is also considered to be authenticated.

The default authorities that MAY be supplied by the Base Template are enumerated in Table 209.

Table 209 Default Base Template Authorities

Name	UID	Common Name	IsClass	Class	Operation
Anybody	00 00 00 09 00 00 00 01	Anybody	False		Sign
Admins	00 00 00 09 00 00 00 02	Admin	True		
Makers	00 00 00 09 00 00 00 03	Maker	True		
MakerSymK	00 00 00 09 00 00 00 04	Maker	False	Makers	SymK
MakerPuK	00 00 00 09 00 00 00 05	Maker	False	Makers	Sign
SID	00 00 00 09 00 00 00 06	TPerOwner	False		Password
TPerSign	00 00 00 09 00 00 00 07	TPerSign	False		TPerSign
TPerExch	00 00 00 09 00 00 00 08	TPerExch	False		TPerExchange
AdminExch	00 00 00 09 00 00 00 09	Admin	False	Admins	Exchange

5.3.4.1.2.1 Anybody

The Anybody authority MAY used as the Host Signing Authority during session startup, and when used in this way allows session startup without providing a proof or secret. If a value is included in the StartSession method's HostChallenge parameter where the Anybody authority is called out as the HostSigningAuthority, the HostChallenge parameter SHALL be ignored by the TPer. Assuming all StartSession method syntax is correct, the method SHALL complete successfully.

The Anybody authority is always considered "authenticated" within a session, even if the Anybody authority was not specifically called out during session startup. Invocations of the Authenticate method that use the Anybody authority SHALL always succeed, assuming all other Authenticate

method syntax is correct. Values in the Challenge parameter of that Authenticate method SHALL be ignored.

The Anybody authority counts against the maximum number of authenticated authorities permitted per session (as reported in the Properties method response MaxAuthentications field, if such a limit exists). Thus, if the maximum number of authorities within a session is 9, the Anybody authority counts as one of these and the host MAY authenticate up to 8 additional authorities (during session startup or through the use of the Authenticate method).

5.3.4.1.2.2 Makers

The members of the Makers authority class permit the manufacturer of the TPer to open an authenticated session to the TPer. The MakerPuK (i.e., Manufacturer) authority only has the Manufacturer Public Key (not the private) and a Certificate attesting to this, which is signed by the Manufacturer.

5.3.4.1.2.3 SID

The SID authority is used by the TPer owner to authenticate to the Admin SP and perform operations such as freezing or deleting SPs.

A copy of the SID is also present in each SP. This SID authority and credential provides the personalizing host with a default password authority that MAY be used to open sessions or verify physical presence. When an SP is issued or created, the value of the Password column of the C_PIN credential object referenced by the SID authority is the same as the value of the Password column of the C_PIN credential object referenced by the SID authority in the Admin SP. Modifications to the SID authority's referenced C_PIN credential object in some SP (even the Admin SP) do not affect any other SP.

By default, the SID credential object (the C_PIN credential object referenced by the SID authority) has a Password column value length of 25 characters. The default character set for this value is made up of the capital letters A-Z inclusive and the numbers 0-9 inclusive, excluding the letter "I" and the letter "O". By default, the CharSet column of the SID credential is the uid to a byte table that stores this character set. Subsequent invocations of GenKey on SID with this CharSet column value utilize this character set to generate the new SID value.

The host application SHOULD manage translation of I's/1's and O's/0's.

5.3.4.1.2.4 TPerSign and TPerExch

The authorities TPerSign and TPerExch are references to the TPer's signing and exchange keys, and allow a host with knowledge of the TPer's credentials to open a secure session with an authenticated TPer. In the Admin SP Authority table, the Credential column contains the reference to locate the appropriate credential for use with this authority.

These TPerSign and TPerExch authorities are present in the Authority table of each SP. The credentials to which these authorities contain references are represented by objects in the appropriate credential tables. The actual key values MAY be stored in only a single location, but the implementation SHALL maintain appropriate references to these credentials so that they are usable in each SP on the TPer.

In all SPs, the values TPerSign and TPerExchange in the Authority table's Operation column indicate that the signing or exchange operation is to be performed with the TPer credentials as referenced in the Admin SP's Authority table.

5.3.4.1.2.5 AdminExch

The AdminExch authority represents the initial credential value submitted during issuance. This is the authority that enables the host to open a secure, implicitly authenticated session to the host's SP and personalize that SP. In the case of the Authority table in the Admin SP, the Base Template Authority

AdminExch SHALL be disabled. At issuance, prior to personalization, the AdminExch authority has a RespExch column value set to the AdminExch authority's UID.

5.3.4.1.3 Authority Operations

The Operation column of the Authority table identifies the authentication method for which an authority object SHALL be used.

The value of an Operation for a given authority SHALL match the purpose for which that authority is being used during session startup. For instance, an authority with an Operation value of Signing or None SHALL only be able to be successfully invoked during session startup as either a HostSigningAuthority or SPSigningAuthority.

The operation types and their requirements are as follows:

a. **None –** This describes an authority that MAY be used to reference a response signing or exchange authority. If used during session startup, an authority with this Operation column value SHALL be referenced as the HostSigningAuthority or SPSigningAuthority. Referencing an authority with this Operation column value as an exchange authority SHALL result in an error. In the event that an attempt is made to invoke Authenticate with an authority that has an Operation column value of None, that authentication SHALL succeed if all other necessary requirements, as defined by the Authority object, are met.

b. **Password –** This describes an authority that MAY be used and authenticated with its referenced C_PIN credential, either during session startup or using the Authenticate method. If used during session startup, this authority SHALL be referenced as the HostSigningAuthority or the SPSigningAuthority. Referencing an authority with this Operation column value in another authority parameter of the session startup methods SHALL result in an error.

c. **Sign –** This describes an authority that MAY be used and authenticated using a challenge and response with its referenced public key (RSA/EC) credential, either during session startup or using the Authenticate method. If used during session startup, an authority with this Operation column value SHALL be referenced as the HostSigningAuthority or as the SPSigningAuthority. Referencing an authority with this Operation column value in another authority parameter of the session startup methods SHALL result in an error. The Sign operation encompasses both Signing and Verification activities – the TPer SHALL perform a signature operation for the SPSigningAuthority, and SHALL perform a signature verification for the HostSigningAuthority.

d. **Exchange –** This describes an authority that MAY be used during session startup, and SHALL be referenced as the HostExchangeAuthority or the SPExchangeAuthority. Referencing an authority with this Operation column value in another authority parameter of the session startup methods SHALL result in an error. The credential referenced by this authority SHALL be used to encrypt session keys for transmission to the other party involved in the session. This authority SHALL NOT be able to be authenticated explicitly using the Authenticate method.

e. **SymK –** This describes an authority that MAY be used and authenticated using a challenge and response with its referenced symmetric key credential, either during session startup or using the Authenticate method. If invoked during session startup, an authority with this Operation column value SHALL be referenced as the HostSigningAuthority or the SPSigningAuthority. Referencing an authority with this Operation column value in another authority parameter of the session startup methods SHALL result in an error.

f. **HMAC –** This describes an authority that MAY be used and authenticated using a challenge and response with its referenced HMAC key credential and the referenced HMAC algorithm, either during session startup or using the Authenticate method. The HMAC credential referenced by the authority using this operation identifies the hash algorithm used to generate the HMAC. If invoked during session startup, an authority with

this `Operation` column value SHALL be referenced as the HostSigningAuthority or the SPSigningAuthority. Referencing an authority with this `Operation` column value in another authority parameter of the session startup methods SHALL result in an error.

g. **TPerSign –** This describes the signing authority that represents the TPer, which enables the host to verify the TPer credentials. This authority MAY be used and authenticated using a challenge and response with its referenced public key (RSA/EC) credential during session startup. If used during session startup, an authority with this `Operation` column value SHALL be referenced as the SPSigningAuthority. Referencing an authority with this `Operation` column value in another authority parameter of the session startup methods SHALL result in an error. The TPer signing credential contains certificate chains that establish the validity of this authority.

h. **TPerExchange –** This describes the exchange authority that represents the TPer. This authority enables the host to establish a secure session with an SP using the TPer's exchange authority. Referencing an authority with this `Operation` column value in another authority parameter of the session startup methods SHALL result in an error. The credential referenced by this authority SHALL be used to encrypt session keys for transmission to the other party involved in the session. This authority SHALL NOT be able to be authenticated explicitly using the `Authenticate` method. The TPer exchange credential contains certificate chains that establish the validity of this authority.

5.3.4.1.4 Disabled Authorities

If the value of the `Enabled` column of an authority is False (see 5.3.2.10.6), then the authority is disabled and SHALL NOT be authenticatable.

Attempts to authenticate a disabled authority during session startup (see 5.3.4.1.5) SHALL result in session startup failure, and a SyncSession status code of NOT_AUTHORIZED (see 5.1.5.2).

Attempts to authenticate a disabled authority using the `Authenticate` method SHALL return a result of False and a method status of SUCCESS (see 5.3.4.1.14.1).

Once an authority is authenticated within a session, setting its `Enabled` column to False SHALL NOT affect the authenticated state of the authority.

5.3.4.1.5 Session Startup

Session startup involves the exchange of either two or four methods between the host and the SP with which the host is attempting to start the session.

The properties of the session – i.e., whether secure messaging is required, the secure messaging type, and the type of message authentication – are controlled by values in the columns of authority objects referenced as parameters to the methods, and are determined independently for each communicator.

When the `StartSession` method is invoked, the authorities to be used for that session are referenced as parameters. The following identifies the order of authority precedence in the `StartSession` invocation. For the invoked host authorities, the following list defines the "Host Control Authority" that identifies the Host-to-SP session property requirements, including the secure messaging properties for communications from the host to the SP:

a. HostSigningAuthority

b. If no HostSigningAuthority is invoked, then the HostExchangeAuthority is the "Host Control Authority".

c. If neither the HostSigningAuthority nor the HostExchangeAuthority invoked, then there is no "Host Control Authority".

For SP response authorities referenced from the "Host Control Authority", the following list defines the "SP Control Authority" that identifies the SP-to-Host session property requirements, including the secure messaging properties for communications from the SP to the host:

a. SPSigningAuthority

b. If no SPSigningAuthority is referenced, then the SPExchangeAuthority is the “SP Control Authority”.

c. If neither the SPSigningAuthority nor the SPExchangeAuthority invoked, then there is no “SP Control Authority”.

If the `StartSession` method fails, the return result is formatted as a `SyncSession` method invocation from the TPer, using only the HostSessionID and SPSessionID parameters, with a non-Success status code.

If the `StartTrustedSession` method fails, the return result is formatted as a `SyncTrustedSession` method invocation from the TPer, using only the HostSessionID and SPSessionID parameters, with a non-Success status code.

5.3.4.1.6 Secure Messaging Control

As indicated in section 5.3.4.1.4, control of secure messaging for a session is determined independently for each communicator. The authorities invoked in the `StartSession` method determine the secure messaging types and algorithms required, and, based on the authorities included in the session startup, the encrypting credential used to exchange session key(s) for secure messaging.

If the Host Signing Authority is invoked in `StartSession`, this authority determines if secure messaging is required on messages from the Host to the TPer, and the type of secure messaging. In this circumstance, the Host Signing Authority is the “Host Control Authority” for messaging from the Host to the TPer.

If the Host Signing Authority is not present, and the Host Exchange Authority is present, the Host Exchange Authority determines if secure messaging is required on messages from the Host to the TPer, and the type of secure messaging. In this circumstance, the Host Exchange Authority is the “Host Control Authority” for messaging from the Host to the TPer.

If the Host Signing Authority is the “Host Control Authority”, its Response (SP) Signing and Response (SP) Exchange Authorities determine the authorities used to represent the SP. If the Host Signing Authority is not invoked in `StartSession`, and the Host Exchange Authority is, it is the Host Exchange Authority that is the “Host Control Authority” whose Response Signing and Response Exchange Authorities determine the authorities used to represent the SP.

If the Response (SP) Signing Authority is linked from the “Host Control Authority”, it is considered the “SP Control Authority”, and determines if secure messaging is required on messages from the TPer to the Host, and the type of secure messaging. If the Response (SP) Signing Authority is not linked from the “SP Control Authority” and the Response (SP) Exchange Authority is, then it is the Response (SP) Exchange Authority that serves as the “SP Control Authority” and determines if secure messaging is required on messages from the TPer to the Host, and the type of secure messaging.

If the value of a “Control Authority’s” `Secure` column is 0, then secure messaging SHALL NOT be permitted for messaging from that communicator for the session to start successfully – all messaging exchanges for sessions controlled by that authority SHALL be in plaintext.

5.3.4.1.7 Hashing and Signing Method Parameters

If the Host Signing Authority is the “Host Control Authority”, then its `Authority.HashAndSign` value identifies whether or not it is required that the parameters of the Host-to-TPer session startup methods (`StartSession/StartTrustedSession`) be hashed. If `Authority.HashAndSign=True`, then the parameters of these methods are hashed and signed using `Authority.Credential` and the hash protocol identified in that credential.

If the Host Exchange Authority is the “Host Control Authority”, then its `Authority.HashAndSign` value identifies whether or not it is required that the parameters of the Host-to-TPer session startup methods (`StartSession/StartTrustedSession`) be hashed and signed. If `Authority.HashAndSign=T`, then

the parameters of these methods are hashed and signed using `Authority.Credential` and the hash protocol identified in that credential.

If the Response (SP) Signing Authority is the "SP Control Authority", then its `Authority.HashAndSign` value identifies whether or not it is required that the parameters of the TPer-to-Host session startup methods (`SyncSession/SyncTrustedSession`) be hashed and signed. If `Authority.HashAndSign=T`, then the parameters of these methods are hashed and signed using `Authority.Credential` and the hash protocol identified in that credential.

If the Response (SP) Exchange Authority is the "SP Control Authority", then its `Authority.HashAndSign` value identifies whether or not it is required that the parameters of the TPer-to-Host session startup methods (`StartSession/StartTrustedSession`) be hashed and signed. If `Authority.HashAndSign=T`, then the parameters of these methods are hashed and signed using `Authority.Credential` and the hash protocol identified in that credential.

5.3.4.1.8 Signed Hashing During Session Startup

If a Signing Authority is invoked in a session startup method (for either the Host or the SP), and the authority's `HashAndSign` column indicates that hashing is required, the signing `Credential` referenced in that Signing Authority's row of the `Authority` table and the hash protocol identified in the `Hash` column of the associated credential are used for the hash/sign operation on the session startup methods.

Session startup SHALL fail if an authority indicates that hashing and signing the session startup methods are required and does not include the signed hash as a parameter of the method invocation.

If a Signing Authority requires the hash/sign operation to be performed, that Authority's row of the `Authority` table SHALL indicate an Operation/Credential pair of Signing/Private Key, SymK/Symmetric Key, or HMAC/Symmetric Key.

The signed hash is sent as the last parameter of the method call and hashes the entire method call (except the hash).

Note that the HostSigningAuthority and SPSigningAuthority provide separate controls for hashing/signing method invocations from the host to the SP and from the SP to the host, respectively. This means that hashing and signing MAY be performed in one of the two communications directions, in both directions, or in neither direction, depending on the `HashAndSign` column values of the HostSigningAuthority and SPSigningAuthority.

5.3.4.1.9 Session Key Exchange

If the Host Signing Authority requires secure messaging, session keys are encrypted with the SP Exchange Authority's symmetric key or public key. If there is no SP Exchange Authority, or if there is an SP Exchange Authority but it does not reference a credential, then session keys are encrypted with the Host Exchange Authority's symmetric key. If there is also no Host Exchange Authority, or if there is a Host Exchange Authority but it does not reference an appropriate credential, an error SHALL be returned.

If the SP Signing Authority requires secure messaging, session keys are encrypted with the Host Exchange Authority's symmetric key or public key. If there is no Host Exchange Authority, or if there is a Host Exchange Authority but it does not reference a credential, then session keys are encrypted with the SP Exchange Authority's symmetric key. If there is also no SP Exchange Authority, or if there is an SP Exchange Authority but it does not reference an appropriate credential, an error SHALL be returned.

If there is no Host Signing Authority, the Host Exchange Authority MAY require secure messaging. If so, session keys are encrypted with the SP Exchange Authority's symmetric or public key. If there is no SP Exchange Authority, or if there is an SP Exchange Authority but it does not reference a credential, then session keys are encrypted with the Host Exchange Authority's symmetric key. If the Host Exchange Authority does not reference an appropriate credential, an error SHALL be returned.

If there is no SP Signing Authority, the SP Exchange Authority MAY require secure messaging. If so, session keys are encrypted with the Host Exchange Authority's symmetric key or public key. If there is no Host Exchange Authority, or if there is a Host Exchange Authority but it does not reference a credential, then session keys are encrypted with the SP Exchange Authority's symmetric key. If the SP Exchange Authority does not reference an appropriate credential, an error SHALL be returned.

When a key is required for integrity checking, that key SHALL always be exchanged as the HostIntegritySessionKey or SPIntegritySessionKey parameters in the `StartTrustedSession` or `SyncTrustedSession` method invocations. These keys are not used in the CCM and GCM authenticated encryption modes. When a key is used for message encryption, that key is exchanged as the HostEncryptSessionKey or SPEncryptSessionKey, even for authenticated encryption modes.

For secure messaging modes that use HMAC for message integrity, the following list identifies the size of the key that SHALL be exchanged as the integrity session key for the appropriate HMAC usage.

a. For HMAC using SHA 256: The integrity session key SHALL be 256 bits.
b. For HMAC using SHA 384: The integrity session key SHALL be 384 bits.
c. For HMAC using SHA 512: The integrity session key SHALL be 512 bits.

For integrity algorithms that utilize public key cryptography, the Host uses the private key corresponding to the chained down certificate supplied in the `StartSession` algorithm to sign the hash of the message. The hash protocol for this operation is identified in the host control authority's credential. The SP uses the chained down certificate supplied in the `SyncSession` algorithm to sign the hash of the message. The hash protocol for this operation is identified in the SP control authority's credential.

5.3.4.1.10 Session Startup Authorities

If the HostSigningAuthority is specified in the `StartSession` invocation, and that authority has an `Operation` column value of Signing, SymK, or HMAC, then the SP SHALL respond to the `StartSession` invocation with a `SyncSession` invocation that contains the SPChallenge parameter, which holds a 32-byte nonce. The host SHALL sign the SPChallenge nonce using the HostSigningAuthority's credential as appropriate, and then submit the signed challenge back to the SP in a `StartTrustedSession` method invocation.

For HostSigningAuthorities with an `Operation` column value of "SymK", the host SHALL encrypt the challenge using the appropriate symmetric key. This encryption SHALL be in ECB mode.

If the HostSigningAuthority is specified, and that authority has an `Operation` column value of Password, then the credential referenced by the authority SHALL be a `C_PIN` object for session startup to resolve properly. The host SHALL send the value of the PIN via the HostChallenge parameter in the `StartSession` method invocation.

If the SPSigningAuthority is referenced, and that authority has an `Operation` column value of Signing, SymK, or HMAC, then the host SHALL include in its `StartSession` method invocation the HostChallenge parameter, which holds a 32-byte nonce. The SP SHALL sign the HostChallenge nonce using the SPSigningAuthority's credential as appropriate, and submit the signed challenge back to the host in its `SyncTrustedSession` method.

For SPSigningAuthorities with an `Operation` column value of "SymK", the device SHALL encrypt the challenge supplied by the host using the appropriate symmetric key. This encryption SHALL be in ECB mode.

If the SPSigningAuthority is referenced, and that authority has an `Operation` column value of Password, then the credential referenced by that authority SHALL be a `C_PIN` object for session startup to resolve properly. The SP SHALL send the value of the PIN via the SPChallenge parameter of the `SyncSession` method.

When session startup successfully completes, all authorities invoked during the session startup process SHALL be considered authenticated.

5.3.4.1.11 EC-MQV Session Startup

It is possible to use the session startup method exchanges to start a session using EC-MQV. In order to do so, it is necessary to follow the following protocol):

1. The host ECMQV ephemeral public key is conveyed in HostChallenge.
2. The host ECMQV static public key is conveyed in HostExchangeCert
3. The SP ECMQV ephemeral public key is conveyed in SPChallenge
4. The SP ECMQV static public key is conveyed in SPExchangeCert
5. The Full ECMQV, C(2,2,ECC MQV) scheme of NIST SP 800-56A, Section 6.1.1.4 is used.
6. The key derivation function (KDF) is Concatenation KDF as defined in Section 5.8.1 of NIST SP 800-56A. The AlgorithmID is the ASCII encoding of the string “TCG Storage ECMQV”. The PartyUInfo is the uinteger Host value and the PartyVInfo is the uinteger SP value. Supplementary fields are not used (that is, they are empty). The hash function used in the KDF is SHA-256 if the elliptic curve is defined over a 163-bit, 192-bit, 224-bit, 256-bit, 233-bit or 283-bit field, and is SHA-384 if the elliptic curve is defined over a 384-bit field.
7. The C(2,2) “Bilateral Key Confirmation” as defined in Section 8.4.1 of NIST SP 800-56A is used (see below).
8. The value MacTagU is conveyed in HostResponse
9. The value MacTagV is conveyed in SPResponse

5.3.4.1.12 EC-DH Session Startup

It is possible to use the session startup method exchanges to start a session using EC-DH. In order to do so, it is necessary to follow the following protocol:

1. The host ECDH static public key is conveyed in HostExchangeCert
2. The host nonce (the value $Nonce_U$ used in key derivation and key confirmation) is conveyed in HostChallenge
3. The SP ECDH static public key is conveyed in SPExchangeCert
4. The SP nonce (the value $Nonce_V$ used in just key confirmation) is conveyed in the SPChallenge
5. The Cofactor Static Unified Model C(0,2,ECC CDH) scheme of NIST SP 800-56A, Section 6.3.2 is used.
6. The key derivation function (KDF) is Concatenation KDF as defined in Section 5.8.1 of NIST SP 800-56A. The AlgorithmID is the ASCII encoding of the string “TCG Storage ECDH”. The PartyUInfo is the uinteger Host value and the PartyVInfo is the uinteger SP value. Supplementary fields are not used (that is, they are empty). The hash function used in the KDF is SHA-256 if the elliptic curve is defined over a 163-bit, 192-bit, 224-bit, 256-bit, 233-bit or 283-bit field, and is SHA-384 if the elliptic curve is defined over a 384-bit field.
7. The C(0,2) “Bilateral Key Confirmation” as defined in Section 8.4.8 of NIST SP 800-56A is used (see below).
8. The value MacTagU is conveyed in HostResponse
9. The value MacTagV is conveyed in SPResponse

5.3.4.1.13 Certificate Presentation

If an authority has a value of True in its `PresentCertificate` column; the authority references a public key credential; and that credential references a certificate, then a certificate chain SHALL be presented when that authority is referenced as an SP authority.

For details on certificate contents/formatting, see the TCG Storage Certificates Specification.

5.3.4.1.14 Explicit Authentication with the Authenticate Method

In addition to authentication of authorities that participate in a successful startup of a session, authentication of an authority MAY also be achieved through successful invocation of the `Authenticate` method.

The `Authenticate` method is an SP method, and as such its Invoking ID parameter SHALL be "ThisSP".

If the invoked authority requires password authentication (the value of the `Operation` column of the invoked authority is “Password”), one call to `Authenticate` is made and the Proof parameter is the password. The response is either True or False – True if authentication was successful and False if authentication was unsuccessful.

If the authority requires challenge and response (the value of the `Operation` column of the authority is “Sign”, “SymK”, or “HMAC”), the host invokes the `Authenticate` method twice. In the first invocation the method parameter list SHALL consist of the Authority parameter, which references the valid Authority object UID of the authority the host is attempting to authenticate. In the second invocation, the Authority parameter SHALL be the same Authority object UID as referenced in the initial `Authenticate` method invocation and the Proof parameter is the response to the SP’s challenge. The Success response is returned to the second invocation– this SHALL be either True if authentication was successful or False if authentication was unsuccessful.

For authorities that require symmetric key challenge/response (have an Operation column value of "SymK"), the Mode column value of that authority’s associated symmetric key credential SHALL be "ECB". If such an authority is invoked as the Authority parameter of the `Authenticate` method, the result of the intial method invocation SHALL be a 32-byte nonce that is the challenge from the SP.

After receipt of the 32-byte challenge, the host SHALL encrypt that nonce using the appropriate symmetric key, in ECB mode. The host SHALL then transmit the encrypted challenge to the TPer as the Challenge parameter of the second `Authenticate` method invocation. The TPer SHALL then validate the response.

If an attempt is made to invoke the `Authenticate` method on an authority that requires secure messaging, and the required secure messaging parameters as defined in the `Secure` column of the `Authority` table are not currently fulfilled, then the `Authenticate` method invocation SHALL fail.

When the `Authenticate` method invocation protocol requires the host to invoke the `Authenticate` method twice, the second `Authenticate` method MAY be sent anytime during the session. If the TPer receives any `Authenticate` method after the first `Authenticate` method has been invoked, the TPer SHALL attempt to resolve that authentication attempt, which SHALL fail if the second `Authenticate` does not contain the appropriate response parameters.

The implementation MAY limit the number of authorities that are able to be authenticated at any one time within a single session (as recorded in the MaxAuthentications value of the `Properties` method). If the authentication attempt would cause the MaxAuthentications property value to be exceeded for the session, a properly invoked `Authenticate` method SHALL return a status of SUCCESS and a result of False.

5.3.4.1.14.1 Authenticate Method Failures

`Authenticate` returns different status codes and return results dependant on the success or failure of the method.

The nature of the `Authenticate` method prescribes two states:

a. Awaiting Challenge

b. Awaiting Challenge Response

Behavior of the `Authenticate` method when the SP is in the Awaiting Challenge state:

1. The method returns INVALID_PARAMETER with an empty result list if the following conditions apply, and remains in the Awaiting Challenge state:
    a. There is no authority supplied to the method, or an invalid authority (as in, an authority that does not exist in the `Authority` table) is supplied to the method, or
    b. A class authority is supplied to the method, or
    c. An incorrect optional parameter identifier or extra parameters are supplied, or a proof is supplied to a non-Password/non-Anybody authority.
2. If the conditions defined in 5.1.5.15 are met, the method SHALL either return AUTHORITY_LOCKED_OUT with an empty result list and remain in the Awaiting Challenge state, or follow the result and status behavior defined in 5 below.
3. If the `Authenticate` invocation does not violate any of the conditions in 1 and 2, above, and if the following conditions are met, then the method returns SUCCESS with a result of True (authentication succeeded), and remains in the Awaiting Challenge state:
    a. The authority invoked in the method is a valid individual authority object in the `Authority` table.
    b. All of the authority's attributes are appropriate for the existing session and authentication attempt – the secure messaging properties are appropriate, the authority is enabled, has an `Operation` column value of Password, etc.
    c. The authority references a valid `C_PIN` credential or the authority is the Anybody authority.
    d. For authorities other than the Anybody authority, the correct password was submitted to the `Authenticate` method invocation.
4. If the `Authenticate` invocation does not violate any of the conditions in 1 and 2 above, and if the following conditions are met, then the method returns SUCCESS with a result of a 32-byte challenge, and transitions to the Awaiting Challenge Response state:
    a. The authority invoked in the method is a valid individual authority object in the `Authority` table.
    b. All of the authority's attributes are appropriate for the existing session and authentication attempt – the secure messaging properties are appropriate, the authority is enabled, references a valid credential, etc.
5. If the `Authenticate` invocation does not violate any of the conditions in 1 and 2 above, and if the following conditions are met, then the method returns SUCCESS with a result of False (authentication failed), and remains in the Awaiting Challenge state:
    a. The authority is a valid authority but one or more of its attributes are not appropriate or valid – the secure messaging requirements have not been met, the authority is disabled, the authority references an invalid credential, the authority has an `Operation` column value of Exchange or TPerSign, etc.
    b. The authority is a valid authority but an incorrect password is submitted to the `Authenticate` method invocation.

Behavior of the `Authenticate` method when the SP is in the Awaiting Challenge Response state:

1. The method returns INVALID_PARAMETER with an empty result list if the following conditions apply, and transitions to the Awaiting Challenge state:
    a. There is no authority supplied to the method, or an invalid authority (as in, an authority that does not exist in the `Authority` table) is supplied to the method
    b. An incorrect optional parameter identifier or extra parameters are supplied.

2. If the Authenticate invocation does not violate any of the conditions in 1 above, and if the following conditions are met, then the method returns SUCCESS with a result of True (authentication succeeded), and transitions to the Awaiting Challenge state:
    a. The authority invoked in the method is a valid individual authority object in the Authority table.
    b. The authority invoked in the method is the same authority invoked in the initial Authenticate invocation.
    c. All of the authority's attributes are appropriate for the existing session and authentication attempt – the secure messaging properties are appropriate, the authority is enabled, references a valid credential, etc.
    d. The correct response was submitted to the Authenticate method invocation.
3. If the Authenticate invocation does not violate any of the conditions in 1 above, and if the following conditions are met, then the method returns SUCCESS with a result of False (authentication failed), and transitions to the Awaiting Challenge state:
    a. The authority is a class authority.
    b. The authority invoked in the method is not the same authority invoked in the initial Authenticate invocation.
    c. The authority is a valid authority but one or more of its attributes are not appropriate or valid – the secure messaging requirements have not been met, the authority is disabled, the authority references an invalid credential, the authority has exceed its uses, etc.
    d. The authority is a valid authority but an incorrect challenge response is submitted to the Authenticate method invocation.

5.3.4.2 Table Management

5.3.4.2.1 Creating Tables

New tables are created via successful invocation of the CreateTable method. The Name-CommonName combination of the table created SHALL be unique in the Table table.

When a new table is created using the CreateTable method, the columns for the table are specified in the Columns parameter. The type of this parameter is a typeOr (for more information on types, see the format specification in section 5.1.1).

a. The first option of this typeOr represents a list of column names and the uid of the type (from the Type table) to be associated with that column. This is to be used if the table does not have unique column.
b. The second option of this typeOr represents a struct made up of two lists. The first list is made up of a list of column names and the uid of the type (from the Type table) to be associated with that column. The first list represents the set of columns in the table the combination of which is required to be unique (these are the unique columns). The second list represents the set of columns in the table that are not part of the unique columns of that table, and is a list of the column names and the uid of the type to be associated with that column.

For a byte table, all rows SHALL exist at table creation. For object tables, no rows SHALL exist, but MAY be inserted using the CreateRow method.

The mechanism by which allocation of rows to a table is accomplished is implementation specific. A manufacturer MAY choose to allocate rows statically (create all rows at table creation) or dynamically (at each CreateRow method invocation).

The total number of rows that are able to be created for a table based on existing conditions SHALL be obtainable using the GetFreeRows method.

The CreateTable method uses the MinSize parameter to define the initial number of rows that SHALL be allocated for the new table. The created table SHALL always be able to have CreateRow invoked on it that many times. If the MinSize is too large (requests more rows than MAY be allocated for that table), the CreateTable method invocation SHALL result in an error.

MinSize is recorded in the MinSize column of the Table table. The MinSize column in the Table table MAY be changed using a Set method invocation. Access control requirements SHALL be fulfilled as normal. The TPer SHALL return an error if an attempt is made to set a lower value than is recorded in the MinSize column. The TPer MAY reject the request and return an error.

The actual number of rows that have been created for a table are reflected in the value of the Table table's Rows column.

The optional MaxSize parameter defines the maximum number of rows that MAY be created for the table. Note that this is a host-supplied number, and that the TPer is not required to guarantee that the table can grow to MaxSize rows. However, the TPer SHALL guarantee that the table never has more than MaxSize rows.

The MaxSize parameter value is recorded in the MaxSize column of the Table table. Access control requirements SHALL be fulfilled as normal to permit this value to be changed, but the TPer MAY prevent the value from being changed – in such a case the TPer SHALL return an INVALID_PARAMETER. Attempts to set MaxSize to a value lower than MinSize or the current size of the table SHALL result in failure with the INVALID_PARAMETER status code.

The optional HintSize parameter represents a number of rows larger than MinSize that is requested for the created table. It is a host-specified number of rows that the host suggests should be supplied by the TPer for that table, if sufficient amount of row space is available. This allows the TPer to optimize the allocation of rows for the table. The TPer is not required to allocate the number of rows requested in HintSize.

When a new table is created, in addition to space being allocated for that table, other side effects occur as well:

a. A row in the Table table SHALL be created. This row is a table descriptor object that stores metadata about the newly created table, including the name of the table, the number of rows allocated for the table, and the number of free rows in the table. The table descriptor object also stores the type of the table (bytes or object), as well as a reference to the Column table row of the table's first column. Tables and their Table table entries are differentiated from each other by different UID composition. A table is referred to by a UID derived from that of the associated Table table entry (see section 3.2.5.3).

b. Rows in the Column table SHALL be created for each of the columns in the new table. Each row in the Column table stores metadata about a column in the newly created table, including the column's name and data type, whether the column value is part of the unique columns for the table (requires uniqueness across the table), and whether the column is subject to transactional rollback, as well as the uidref to the row in the Column table that stores metadata about the next column in the created table.

c. Rows in the AccessControl table SHALL be created for each of the methods available for the newly created table, its associated Table table row, and the newly created Column table rows (see 5.3.4.2.3). The access control associations that are created are as follows:

    a. TableUID.Next

    b. TableUID.Get

    c. TableUID.Set

    d. TableObjectUID.Get

e. ColumnObjectUID*.Get

f. ACEObjectUID.Get/Set (if a new ACE is created)

d. Rows in the ACE table MAY be created to limit access control on certain portions of the table or its associated Table table row.

If the new table is unable to be created due to insufficient additional space in the Table table, the method SHALL fail (INSUFFICIENT_SPACE). If the associated rows in all associated tables are unable to be created, then the invocation of the CreateTable method SHALL fail (INSUFFICIENT_ROWS).

5.3.4.2.2 Retrieving Table Data

Rows in the Table table are table descriptor objects. Since each table descriptor is an object, each row in that table, like any other object, has its own methods that MAY be used to retrieve the data stored in that object. So, upon creation of a table, the control authority for a session is set in the access control associations for the Get method that enable the host to retrieve that table descriptor object's data after authenticating that authority.

Table data, the data stored in table cells, MAY be retrievable through successful invocation of the Get method on that table. When the Get method is invoked on a table or object, only the values that are readable based on currently authenticated authorities and their associated ACE restrictions for the method SHALL be returned.

Cell values that have been requested but are not permitted to be read by the currently authenticated authorities are not returned. Since the return value of the method for non-byte tables is a list of name-value pairs, cells to which the host invoking the Get method does not have access are omitted from the return result. If a column is known to exist but not returned with a value, then the host is able to discern that it did not have permission to invoke Get on that cell. It is not an error to request columns that are not permitted to be retrieved.

If the currently authenticated authorities do not satisfy the access control restrictions for invoking Get on a byte table, the method SHALL return an empty results list.

5.3.4.2.3 Creating Table Rows

Tables MAY be modified in the following ways:

a. New rows MAY be created

b. Existing rows MAY be deleted

c. Cell values MAY be changed

In most cases, new rows SHALL be added to a table through successful invocation of the CreateRow method on that table. Exceptions to this, where rows are created in tables by methods other than CreateRow, are defined as follows:

a. Table – the TPer automatically creates rows in the Table table upon successful invocation of the CreateTable method.

b. Column – the TPer automatically creates rows in the Column table upon successful invocation of the CreateTable method.

c. MethodID – the TPer SHALL NOT permit rows to be created in the MethodID table via the CreateRow method.

d. AccessControl the TPer automatically creates rows in the AccessControl table when a table or object is created, except in the case where those rows are created in the AccessControl table.

e. Log – the TPer SHALL NOT permit rows to be created in the Log table via the CreateRow method.

f. LogList – the TPer automatically creates rows in the LogList table upon successful invocation of the CreateLog method.

Successful invocation of the CreateRow method creates a row in the invoking table where the column values for that row are the values passed as parameters to the method invocation. Successful invocation of the CreateRow method requires that the host supply values for each column in the row.

When invoking the CreateRow method on a table that requires certain column values to be unique, an attempt to create a row with parameterized values equivalent to the values in the unique columns of some row of that table SHALL fail.

When a row in an object table is created, a number of AccessControl table rows are created that correspond to the default methods permitted for the created object. ACLs are set on those methods, and in the meta-ACL columns associated with those methods, as follows:

a. Using the HostSigningAuthority from the StartSession method, if provided.

b. Otherwise, using the HostExchangeAuthority from the StartSession method, if provided.

c. Otherwise, using the Anybody Authority.

A new ACE SHALL be created in the ACE table for the new rows of the AccessControl table to reference in its ACLs. All of the new AccessControl table rows reference the same ACE in their ACLs.

For the new ACE created in the ACE table, additional new rows are created in the AccessControl table for the new ACE's methods. Those new AccessControl table rows SHALL also reference the newly created ACE in their ACLS.

Note that CreateRow MAY be limited in some instances based on required default values for some table columns, and MAY be required to make certain validity checks when creating rows for some tables.

If a new object is unable to be created due to insufficient additional space in the containing table, the CreateRow method SHALL fail (INSUFFICIENT_SPACE). If the associated rows in all associated tables are unable to be created, then the invocation of the CreateRow method SHALL fail (INSUFFICIENT_ROWS).

5.3.4.2.4 Deleting Table Rows

Rows MAY be deleted from a table in two ways:

a. Successful invocation of the DeleteRow method on the table

b. Successful invocation of the Delete object method on an object

When an object is deleted by a successful invocation of the DeleteRow method, the side effects of the method are the same as if the object had been deleted via invocation of the Delete method.

Deleting objects MAY have side effects. For instance, invoking the Delete method on a Table table row has the side effect of deleting the table with which that table descriptor object is associated. Side effects that occur upon deletion of objects are documented where appropriate.

Deleting objects SHALL also cause all AccessControl table rows where this object's UID appears in the InvokingID column to be deleted.

Deletion of table rows via invocation of DeleteRow or Delete SHALL NOT be permitted for the following tables:

a. Column – the TPer automatically deletes rows from the Column table when a table is deleted.

b. MethodID – the TPer SHALL NOT permit rows to be deleted from the MethodID table.

c. AccessControl – the TPer automatically deletes rows from the AccessControl table whan a table or object is deleted.

d. LogList - the TPer automatically deletes rows from the LogList table when a Log table is deleted.

5.3.4.2.5 Deleting Tables

As indicated in Section 5.3.4.2.4, a table SHALL be deleted by successful invocation of the Delete method or the DeleteRow method on the table descriptor object (Table table row) associated with the table to be deleted.

When the method resolves, the following occurs:

a. The table descriptor object associated with the table is deleted.

b. The table itself is deleted.

c. All associated AccessControl table rows are deleted. This includes methods associated with both the table itself as well as the table descriptor object.

d. All associated Column table rows SHALL be deleted.

Due to their reusability, ACEs that were created when a table was created SHALL NOT be deleted when that table is deleted, as the ACE MAY still be in use by another table. ACEs created as a side effect of creating a table or object SHALL be deleted only by direct host action.

5.3.4.2.6 Modifying Tables

In most cases, modifications to tables are accomplished using the Set method (access control permitting). Other cases that allow modification of tables without use of the Set method are noted in the appropriate section of this specification.

Unlike with the Get method, when the Set method is invoked on a table but access control does not permit some cell to be changed that the Set method invocation is attempting to change, the entire method fails and returns NOT_AUTHORIZED. All changes parameterized in the Set method SHALL be made for the method to resolve successfully.

Columns submitted in the RowValues parameter of the Set method are permitted to be sent in any order. Including the same column multiple times in a single Set method invocation SHALL result in the method failing with the status INVALID_PARAMETER.

5.3.4.2.7 Iterating Through Tables

The Next method is used to discover an ordering of rows in an object table. Since the ordering of object tables is unspecified, the ordering that is discovered by successful invocation(s) of this method on an object table is some undefined ordering, the "current" ordering.

When successfully invoked on an object table, the Next method returns a list of zero or more uidrefs "following" the specified Where row in the current ordering. If a value for the Where parameter is not specified in the method invocation, the first element, if any, of the list of uidrefs, SHALL denote the "beginning" of the ordering, i.e. the row that has no predecessor in the current ordering.

The implementation is required to discover a consistent ordering of all rows of an object table only if the object table is not modified between invocations of Next. Actions that cause modifications to the object table that could result in a new ordering SHALL be specified in each SSC, and SHALL include at least the method calls that add or delete rows if those are permitted by the SSC.

If both the Where parameter and the Count parameter are omitted, the scope of the Next method's return value is the entire table.

If the Where parameter is included in the invocation and the Count parameter is omitted, the scope of the Next method's return value begins at the starting point in the table's ordering indicated by the row following that identified by the Where parameter, and ends at the end of the table's row ordering.

5.3.4.3 Access Control

Access control describes the system used to prevent modification of an SP's contents by a host that does not have proper authorization to make those modifications.

The AccessControl table stores access control associations between methods and the tables, objects, or SP upon which those methods MAY operate. Each row is an access control association made up of a reference to the method and a reference to the object/table/SP; an Access Control List (ACL); meta-ACLs; and columns that store the logging settings for the access control association and its associated meta-ACL methods.

Each InvokingID/MethodID combination in the AccessControl table SHALL be unique within the table.

Each ACL column SHALL hold a limited number of ACEs. The actual number that each ACL column MAY store is SSC-dependent.

If the ACL column for an access control association is empty (ie contains no ACEs), then that InvokingID/MethodID combination SHALL NOT be invocable. Attempts to invoke that InvokingID/MethodID combination SHALL fail and result in status code NOT_AUTHORIZED.

5.3.4.3.1 Meta-ACLs

The ability to add ACEs to or delete ACEs from ACLs provides granularity of access control, as the authorization required to add or remove ACEs from ACLs MAY be different from the authorization required to invoke the method described in the access control association.

The methods that perform the operations that add or remove ACEs from ACLs, or retrieve the list of ACEs from an ACL, are AddACE, RemoveACE, and GetACL. Additionally, access control associations MAY be removed from the AccessControl table through successful invocation of the DeleteMethod method. These four methods together are the meta-ACL methods.

Each access control association stores ACLs that govern the authorization requirements for these methods on that access control association. A separate column in the AccessControl table exists for each of the meta-ACL methods to store the ACL for that meta-ACL method. A separate column in the AccessControl table also exists for each of the meta-ACL methods to store the logging settings for that meta-ACL method. The AccessControl table does not have rows for access control associations representing the meta-ACL methods.

In order to add an ACE to an ACL, the ACL for the AddACE method associated with that access control association (stored in the AddACEACL column) SHALL be satisfied. RemoveACE, GetACL, and DeleteMethod function in a similar way, and would have to fulfill the ACL stored in the RemoveACEACL, GetACLACL, and DeleteMethodACL column respectively.

5.3.4.3.2 BooleanExpr Column Format

The BooleanExpr column of the ACE table has a type of AC_element. Table 210 contains an encoding example of the AC_element list representing the infix ACE expression:

((00 00 00 09 00 00 00 32 AND 00 00 00 09 00 00 00 24) OR 00 00 00 09 00 00 82 73) AND 00 00 00 09 00 00 77 28

Table 210 ACE_expression Encoding Example

Token	Meaning
F0	Start List
F2	Start Name
A4 00 00 0C 05	Half-UID – Authority_object_ref

Token	Meaning
A8 00 00 00 09 00 00 00 32	Authority_object_ref
F3	End Name
F2	Start Name
A4 00 00 0C 05	Half-UID – Authority_object_ref
A8 00 00 00 09 00 00 00 24	Authority_object_ref
F3	End Name
F2	Start Name
A4 00 00 04 0E	Half-UID – boolean_ACE
00	boolean_ACE - AND
F3	End Name
F2	Start Name
A4 00 00 0C 05	Half-UID – Authority_object_ref
A8 00 00 00 09 00 00 82 73	Authority_object_ref
F3	End Name
F2	Start Name
A4 00 00 04 0E	Half-UID – boolean_ACE
01	boolean_ACE - OR
F3	End Name
F2	Start Name
A4 00 00 0C 05	Half-UID – Authority_object_ref
A8 00 00 00 09 00 00 77 28	Authority_object_ref
F3	End Name
F2	Start Name
A4 00 00 04 0E	Half-UID – boolean_ACE
00	boolean_ACE - AND
F3	End Name
F1	End List

5.3.4.3.3 Modifying the BooleanExpr Column

The `BooleanExpr` column of the `ACE` table is modifiable using the `Set` method. Some implementations MAY allow the `BooleanExpr` column to be set to an empty list. If the `BooleanExpr` column value is an empty list, that ACE cannot be satisfied and as such always resolves to False.

If an implementation does not permit the `BooleanExpr` column to be set to an empty list, attempts to do so SHALL result in the `Set` method failing with a status of INVALID_PARAMETER.

5.3.4.4 Deleting the SP

The TPer owner is able to delete an SP by opening a session to the Admin SP and invoking the `Delete` method on the SP object in the Admin SP's `SP` table, as defined in 5.4.4.2. However, the SP owner is probably unable to delete the SP in this way, and instead uses the `DeleteSP` method.

The SP SHALL NOT be deleted until the session is successfully closed. When the method takes effect, it produces the same results as defined in 5.4.4.2.

5.3.4.5 SetPackage Method Operation

The `SetPackage` method takes a value that is the result of a successful `GetPackage` method invocation, a credential object uidref to the credential that is used to decrypt the encrypted key contents of the package, and a credential object uidref to the credential that is used to verify the signed hash of the contents of the package.

The TPer decrypts the key material using the credential referenced by the WrappingKey and verifies the signed hash using credential referenced by the SigningKey and its associated hash algorithm. The TPer then sets the columns of the invoking credential object with the decrypted key material from the package.

The Log portion of the unwrapped package causes that log entry to be made to the default `Log` table, along with the Date portion, if these values exist in the package. If the Log Template has not been issued into the SP, then the Log and Date data are disregarded.

5.3.4.6 Default Logging Settings

The default logging settings associated with the Template methods assume that the Log Template has been issued with the SP. Otherwise, these values should be disregarded, as values of log control columns SHALL be ignored if the Log Template has not been issued with an SP.

a. Session startup logging (controlled in the `Authority` table) and logging invocation of the `Authenticate` method SHALL have default settings of LogAlways.
b. The following method invocations SHALL by default log as LogAlways:
    a. AddACE
    b. RemoveACE
    c. DeleteMethod
    d. Delete
    e. CreateTable
    f. CreateRow
    g. DeleteRow
c. Invocation of the following method invocations SHALL by default log as LogFailure:
    a. DeleteSP
    b. Set
    c. GenKey
d. All other methods described in the Base Template SHALL be default log as LogNever.

5.3.5 Life Cycle

5.3.5.1 Base Template-Specific Life Cycle State Descriptions/Exceptions

An SP issued with the Base Template has the following characteristics based on the current life cycle state of that SP:

a. **Disabled –** A Base Template-enabled SP that is in the Disabled state SHALL NOT be able to perform any user-invoked SP operations enabled, with the exceptions noted in section 4.5.2. These exceptions include invocation of the `Authenticate` method, the `DeleteSP` method, and the `Set` method used to re-enable the SP. Session Manager protocol layer methods invoked to the disabled SP SHALL operate as normal.

b. **Frozen –** Attempts to open sessions to an SP in the Issued-Frozen state SHALL fail.

c. **Issued-Disabled-Frozen** – Attempts to open sessions to an SP in the Issued-Disabled-Frozen state SHALL fail.

5.4 Admin Template

5.4.1 Overview

Begin Informative Content

The purpose of the Admin Template is to provide to the Admin SP the capability to optionally issue additional SPs and to maintain information about the TPer.

End Informative Content

5.4.2 Data Structures

5.4.2.1 TPer Metadata Group - TPerInfo (Object Table)

The table in this section describes the metadata that the Admin SP stores about the TPer. The `TPerInfo` table SHALL contain exactly one row that is always readable by the Anybody authority.

Table 211 TPerInfo Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Bytes		uinteger_8
0x02	GUDID		bytes_12
0x03	Generation		uinteger_4
0x04	FirmwareVersion		uinteger_4
0x05	ProtocolVersion		uinteger_4
0x06	SpaceForIssuance		uinteger_8
0x07	SSC		SSC

5.4.2.1.1 UID

This is the unique identifier of this row in the table.

This column SHALL NOT be modifiable by the host.

5.4.2.1.2 Bytes

This value is the size in bytes of the TPer's entire protected storage area.

This column SHALL NOT be modifiable by the host.

5.4.2.1.3 GUDID

This column value is the TPer's globally unique serial number - See Table 212 and accompanying text for GUDID content description

This column SHALL NOT be modifiable by the host.

5.4.2.1.4 Generation

This is the generation number of the volume.

This column SHALL NOT be modifiable by the host.

5.4.2.1.5 FirmwareVersion

This is the manufacturer-defined revision number of the TPer firmware.

This column SHALL NOT be modifiable by the host.

5.4.2.1.6 ProtocolVersion

This is the revision number of the interface messaging protocol, defined by this specification or an SSC.

This column SHALL NOT be modifiable by the host.

5.4.2.1.7 SpaceForIssuance

This is the total amount of available bytes remaining for issuance.

This column SHALL NOT be modifiable by the host.

5.4.2.1.8 SSC

This is a list of the names of the SSCs supported by the TPer. An SSC's name is defined in the specification that defines it.

This column SHALL NOT be modifiable by the host.

5.4.2.2 TPer Metadata Group - Serial Number Contents

Table 212 GUDID Column Contents Description

Byte\Bit	7	6	5	4	3	2	1	0
0	0x02							
1	0x23							
2	0x00							
3	0x08							
4	NAA (0x05)				(MSB) IEEE COMPANY ID			
5	IEEE COMPANY ID							
6								
7	IEEE COMPANY ID (LSB)				(MSB) VENDOR-SPECIFIC IDENTIFIER			
8	VENDOR-SPECIFIC IDENTIFIER							
9								
10								
11	VENDOR-SPECIFIC IDENTIFIER (LSB)							

This structure meets the requirements of an identification descriptor as in SPC-3, and specifically conforms to the NAA IEEE Registered format defined in that document.

5.4.2.3 TPer Metadata Group - CryptoSuite (Object Table)

The table in this section describes the metadata the TPer keeps about its cryptographic capabilities. The rows in this table SHALL represent all of the crypto functionality on the TPer available to SPs.

The times recorded in the `CryptoSuite` table SHALL be average time based on 100 independent samples using randomly generated keys. The times MAY be taken when the TPer is otherwise idle and represent relative performance of the operations, not a guarantee of actual performance in the field.

Every type of crypto functionality present on the TPer SHALL have one row where the value of the `Special` column is False, and MAY have one or more rows where the value of the `Special` column is

True. Rows where `Special`=True represent functionality that MAY be defined by special properties of the device such as hardware accelerators or pre-computed cache values (in the case, for example, of some key generation or random number provisioning).

Table 213 CryptoSuite Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	CryptoCall		name
0x02	CryptoLen		uinteger_2
0x03	CryptoOp		name
0x04	Special		boolean
0x05	Time		uinteger_4
0x06	Variance		uinteger_4

5.4.2.3.1 UID

This is the unique identifier of this row in the table.

This column SHALL NOT be modifiable by the host.

5.4.2.3.2 CryptoCall

This is the name of the crypto type whose characteristics are presented in this row.

This column SHALL NOT be modifiable by the host.

5.4.2.3.3 CryptoLen

This is the key length of the crypto type whose characteristics are presented in this row.

This column SHALL NOT be modifiable by the host.

5.4.2.3.4 CryptoOp

This is the name of the crypto operation being timed for this CryptoType/CryptoLen combination (i.e., KeyGen, Encrypt, Decrypt, Sign, Verify, Hash)

This column SHALL NOT be modifiable by the host.

5.4.2.3.5 Special

This column value defines if special operating properties exist for this CryptoType/CryptoLen combination.

This column SHALL NOT be modifiable by the host.

5.4.2.3.6 Time

This is the nominal operation time, in milliseconds, associated with this crypto type.

This column SHALL NOT be modifiable by the host.

5.4.2.3.7 Variance

Nominal operation time or variance in milliseconds, as applicable.

This column SHALL NOT be modifiable by the host.

SPs on the TPer Group - Template (Object Table)

The table in this section describes the data that the Admin SP keeps about all of its templates. The `Template` table SHALL have one row for each template that MAY be issued by the TPer.

Table 214 Template Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	RevisionNumber		uinteger_4
0x03	Instances		uinteger_2
0x04	MaxInstances		uinteger_2

5.4.2.3.8 UID

This is the unique identifier of this row in the table.

This column SHALL NOT be modifiable by the host.

5.4.2.3.9 Name

This is the unique name of this template.

This column SHALL NOT be modifiable by the host.

5.4.2.3.10 RevisionNumber

For Templates defined by the TCG Core Specification, this value is the TCG Core Specification revision number of the Template.

This column SHALL NOT be modifiable by the host.

5.4.2.3.11 Instances

This is the number of SPs on the TPer that are currently instantiated from this Template. Only deleting an SP decrements this number.

If the value of the `Instances` column is equal to the value of the `MaxInstances` column for a given Template, then attempts to issue additional SPs incorporating that Template SHALL result in an error.

This column SHALL NOT be modifiable by the host.

5.4.2.3.12 MaxInstances

This is the maximum number of SPs that MAY be instantiated from this Template at any one time. If this value is 0, then there is no limit on number of instances.

This column SHALL NOT be modifiable by the host.

5.4.2.4 SPs on the TPer Group - SP (Object Table)

The table in this section describes the data that the Admin SP keeps about all of the SPs on the TPer. The Admin SP SHALL be `UID=0x00 0x00 0x02 0x05 0x00 0x00 0x00 0x01` in the `SP` Table, and the values in this table SHALL be readable by the Anybody authority.

Table 215 SP Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name

Column Number	Column Name	IsUnique	Column Type
0x02	ORG		Authority_object_ref
0x03	EffectiveAuth		max_bytes_32
0x04	DateofIssue		date
0x05	Bytes		uinteger_8
0x06	LifeCycleState		life_cycle_state
0x07	Frozen		boolean_def_false

5.4.2.4.1 UID

This is the unique identifier of this row in the table.

This column SHALL NOT be modifiable by the host.

5.4.2.4.2 Name

This is the unique name of the SP represented by this row.

This column SHALL NOT be modifiable by the host.

5.4.2.4.3 ORG

This is the uidref to the Authority object that authorized this SP.

This column SHALL NOT be modifiable by the host.

5.4.2.4.4 EffectiveAuth

This is the OID of the chained-down certificate that actually authorized issuance of this SP.

This column SHALL NOT be modifiable by the host.

5.4.2.4.5 DateofIssue

This value is the date of Issuance (enabled by Clock Template). If the Clock Template is not issued into the Admin SP, this value is set to an empty struct.

This column SHALL NOT be modifiable by the host.

5.4.2.4.6 Bytes

This is the size in bytes of the SP.

This column SHALL NOT be modifiable by the host.

5.4.2.4.7 LifeCycleState

This column value represents the life cycle state of this SP. The `LifeCycleState` column SHALL NOT be written directly - the TPer changes it as appropriate.

This column SHALL NOT be modifiable by the host.

5.4.2.4.8 Frozen

This column provides TPer Owner control over whether sessions MAY be opened on this SP. A value of True in this column indicates that attempts to open sessions to this SP SHALL fail. The default value of this column at issuance is False.

5.4.3 Methods

5.4.3.1 IssueSP (SP Method)

This method is used to issue SPs on those TPers where the SPs are not fixed by the manufacturer.

ORGs that are permitted to only issue certain Templates into new SPs are controlled by attributes in the ORG's certificate.

```
ThisSP.IssueSP [
       SPName : name,
       Size : uinteger,
       Templates : list [ TemplateObjectUID ... ],
       AdminExch : key_size,
       Enabled : boolean ]
=>
[ UID : uid, Size : uinteger ]
```

5.4.3.1.1 SPName

This is the name for the newly created SP. This SHALL be the value of the `Name` column of the `SP` table.

5.4.3.1.2 Size

This value is the size in 512 byte blocks that is requested for this SP. If the TPer is unable to allocate the requested Size, the `IssueSP` method invocation SHALL result in an error.

5.4.3.1.3 Templates

This is the list of templates from which the SP is to be created. It is a list of UIDs of templates to be included in the issued SP. The UIDs are those of the templates as recorded for each template in the Admin SP's `Template` table.

Issuance always assumes the Base template, but it is not an error to list it. The list of templates MAY include any template except the template named "Admin," though the templates available for use are restricted by the `MaxInstances` allowed for each template.

The methods and tables from the templates requested become part of the issued SP.

5.4.3.1.4 AdminExch

A C_AES_* object of the size indicated by the parameter used is created upon SP issuance that uses the key submitted in this parameter. The AdminExch authority in the new SP SHALL have a `Credential` column value that references the newly created credential.

5.4.3.1.5 Enabled

This value identifies whether the SP is issued in an Enabled or Disabled state. The value submitted to this parameter is set to the new SP's `SPInfo` table.

5.4.3.1.6 IssueSP Result

5.4.3.1.6.1 UID

This return value is the UID of the SP as assigned to the SP object in the Admin SP's `SP` table.

5.4.3.1.6.2 Size

The returned Size is the size actually allocated. The Size returned SHALL be equal to or greater than the `Size` requested.

5.4.3.1.7 Fails

a. If there is already an SP of the same name.
b. If the maximum number of SPs permitted for this template already exist.
c. If there's not enough free space for the new SP in the TPer.

5.4.4 Descriptions

There SHALL be exactly one Admin SP on every TPer that has SPs. The Admin SP SHALL NOT be able to be disabled or deleted.

For TPers that have SPs when a TPer is shipped from the manufacturer there SHALL be two or more predefined Templates and at least one SP (the Admin SP). There MAY also be additional SPs issued on the TPer during the manufacturing process.

5.4.4.1 Templates and the Admin SP

Template metadata is stored in the Admin SP's `Table`, `Column`, and `MethodID` tables.

In the Admin SP, the value of the `TemplateID` column in the `Table` table and `MethodID` table MAY be zeroes. When the value of the `TemplateID` column in the `Table` or `MethodID` tables is zeroes (null uid reference) it indicates that the row is a normal, active `Table` or MethodID table row of the Admin SP. Otherwise, when it is not zeroes, the value of the `TemplateID` column in a row of the Admin SP's `Table` table or `MethodID` table SHALL be the uid of the row of the `Templates` table to which that table belongs. In addition, this indicates a table or method to be created when an SP is issued using that template.

Rows with a non-zero `TemplateID` are readable by Anybody.

The `Rows` column of the `Table` table on the Admin SP MAY be `Set` by the `Issuers` authority. The `Rows` column indicates how many free rows are available in the given table after issuance is complete. Since the process of issuing an SP MAY create rows in various tables, it is simpler to have this indicate "room left for the host application to use" rather than "total space to allocate."

In issued SPs (i.e. SPs other than the Admin SP), the `TemplateID` column value SHALL always be a NULL UID.

5.4.4.2 Deleting SPs via the Admin SP

The TPer owner, or a host with equivalent permissions, is able to delete an SP by opening a session to the Admin SP and invoking the `Delete` method on the SP object in the Admin SP's `SP` table.

This method SHALL only operate within a Read-Write session to the Admin SP. The SP SHALL NOT be deleted until the session is successfully closed. Upon successful deletion of the SP, the following changes are made:

1. The row in the Admin SP's `SP` table that represents this SP is deleted.
2. The value of the `Instances` column of the Admin SP's `Template` table is reduced by 1 for each of the templates that had been issued into the SP being deleted.
3. The SP itself is deleted. The means of deletion is implementation-specific. Once the SP has been deleted, the Host SHALL no longer have the capability to open sessions to the SP.
4. Any TPer functionality affected by the existence of the SP based on the templates incorporated into it is modified as defined in the appropriate Template reference section of this specification.

5.4.4.3 Admin SP Sessions

An open Read-Write session to the Admin SP SHALL NOT be able to be combined with sessions of any type open to any other SPs on the TPer (including sessions that are already open when the attempt to open a Read-Write session to the Admin SP is made).

5.4.4.3.1 Issuance Sessions

Issuance requires a session to the Admin SP that incorporates HostSigning, HostExchange, SPExchange, and optionally SPSigning, all based on Manufacturer controlled Certificates. The Admin SP SHALL require that the HostSigningAuthority and the HostExchangeAuthority (which MAY be different) are present in the `ORG` section of the `Authority` Table. Certificate chain down is possible, to the `Chain Limit`.

The critical method in issuance is `IssueSP`. The ACL on this method contains exactly one ACE that SHALL NOT be changed, and it requires a Boolean combination of Authorities: (HostSigning AND HostExchange AND SPExchange). The SPSigningAuthority is optional but recommended. Issuance SHALL NOT be deemed completed until the method has completed successfully and the session has successfully closed.

During issuance, the host is responsible for providing logging, and fetching information it requires to confirm the issuance.

5.4.4.4 Authorities

The authorities that SHALL be required by the Admin Template are enumerated in Table 216.

Table 216 Default Admin Template Authorities

Name	UID	Common Name	IsClass	Class
Issuers	00 00 00 09 00 00 02 01	SPControl	True	
Editors	00 00 00 09 00 00 02 02	SPControl	True	
Deleters	00 00 00 09 00 00 02 03	SPControl	True	
Servers	00 00 00 09 00 00 02 04	SPControl	True	
Reserve0	00 00 00 09 00 00 02 05	SPControl	True	
Reserve1	00 00 00 09 00 00 02 06	SPControl	True	
Reserve2	00 00 00 09 00 00 02 07	SPControl	True	
Reserve3	00 00 00 09 00 00 02 08	SPControl	True	

The TPerExch authority allows a secure session to be established immediately with the Admin SP. Note the corresponding credentials contain certificate chains that establish the validity of TPerSign and TPerExch signed by the manufacturer.

In addition to the authorities defined in Table 216, if a TPer supports Issuance, then it SHALL be required that the Admin SP have up to an additional 2^{16} entries in this table, in blocks of 16, starting immediately after the default Base and Admin Template authorities. These are called ORG authority blocks.

ORG0 is the ORG (anization) of the manufacturer or SP licensing authority. Other ORGs MAY include other SP licensing authorities. The classes include SP Issuance authorities (Issuers), SP ORG Editors that are able to edit values within an ORG block, SP Deleters that are restricted to deleting ORG authorities within a 16 block, and SP Servers that are used to set up confidential messaging between Issuance participants. Members of the Servers class are Sign and Exchange authorizations in order to permit secure messaging.

5.4.4.5 Default Logging Settings

The default logging settings associated with the Admin Template methods are:

a. The default logging for Admin SP method, `IssueSP`, is LogAlways.

b. All other methods that apply to the Admin SP SHALL be as described in the Base Template reference section (See Section 5.3.4.4).

5.4.5 Life Cycle

5.4.5.1 Admin Template-Specific Life Cycle State Descriptions/Exceptions

The Admin SP has the following characteristics based on the current life cycle state of that SP:

a. **Disabled –** Access control SHALL prevent the Admin SP from entering the Disabled state.

b. **Frozen –** Access control SHALL prevent the Admin SP from entering the Frozen state.

c. **Issued-Disabled-Frozen** – Access control SHALL prevent the Admin SP from entering the Issued-Disabled-Frozen state.

5.5 Clock Template

5.5.1 Overview

Begin Informative Content
The Clock Template enables an SP to manage information about time.
End Informative Content

A TPer MAY support any number of SPs that incorporate the Clock Template, up to the limit imposed by the SSC or implementation and reported in the Admin SP's `Template` table.

5.5.2 Terminology

Table 217 Clock Template Terminology

Term	Definition
ExactTime	ExactTime is a time value represented by the clock_time type. ExactTime is a return value in the GetClock method and a parameter of the SetClockHigh and SetClockLow methods.
HighTime	HighTime represents the actual current High Trust time value, which is the value of the HighSetTime column plus the time elapsed on the IncrementalClock since the HighSetTime value was set. This is the value returned as ExactTime when GetClock is invoked and High Trust time is returned.
High Trust	A time value retrieved from a remote but strongly protected source of time
IncrementalClock	Each Clock Template-enabled SP SHALL have an incremental clock that is accessible from the TPer and is used to measure time intervals.
LagTime	The time period recorded by the Host Application between when it read time from its time source and when it received the OK result from the SP upon successful receipt and processing of the ExactTime parameter of the SetClockHigh or SetClockLow method.
LowTime	LowTime represents the actual current Low Trust time value, which is the value of the LowSetTime column plus the time elapsed on the IncrementalClock since the LowSetTime value was set. This is the value returned as ExactTime when GetClock is invoked and Low Trust time is returned.
Low Trust	An immediate but not strongly protected source of time, such as the local PC clock
MonotonicTime	A 64-bit persistent counter needed for clock requests that require a counter. MonotonicTime operations increment the counter independent of transactions and of the Read/Write state of the session.

Term	Definition
MonotonicIncrement	This is a counter kept in main memory that is used to reduce the number of writes to media that are needed to support the MonotonicTime counter.
Timer Mode	The Clock Template-enabled SP operates in this mode after a power cycle/hardware reset, or if time values have never been set. Retrieving the time in Timer mode returns the value of IncrementalClock and MonotonicTime.

5.5.3 Data Structures

5.5.3.1 ClockTime (Object Table)

The ClockTime table SHALL contain exactly one row, with `UID=0x00 0x00 0x04 0x01 0x00 0x00 0x00 0x01`.

Table 218 ClockTime Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	HaveHigh		boolean
0x02	HighByWhom		Authority_object_ref
0x03	HighSetTime		clock_time
0x04	HighInitialTimer		clock_time
0x05	HighLag		lag
0x06	HaveLow		boolean
0x07	LowByWhom		Authority_object_ref
0x08	LowSetTime		clock_time
0x09	LowInitialTimer		clock_time
0x0A	LowLag		lag
0x0B	MonotonicBase		uinteger_8
0x0C	MonotonicReserve		uinteger_8
0x0D	TrustMode		clock_kind

5.5.3.1.1 UID

This is the unique identifier of this row in the table.

This column SHALL NOT be modifiable by the host.

5.5.3.1.2 HaveHigh

If the value of this column is True, then the values in the High Trust time columns (`HighByWhom`, `HighSetTime`, `HighInitialTimer`, and `HighLag`) are meaningful. If the value of the `TrustMode` column is Low or Timer then the value of this column SHALL be False.

This column SHALL NOT be modifiable by the host.

5.5.3.1.3 HighByWhom

This is the uidref to the Authority that is in the control authority in the session in which the High Trust time was set. This value is valid only if `HaveHigh` is set to True; otherwise it should be zeroes.

This column SHALL NOT be modifiable by the host.

5.5.3.1.4 HighSetTime

The value of this column is the time set to the value of the ExactTime parameter of the `SetClockHigh` method when that method is successfully invoked. This value is valid only if `HaveHigh` is set to True; otherwise it should be zeroes.

5.5.3.1.5 HighInitialTimer

The value of this column is set to the value of the IncrementalClock when the ExactTime parameter of the `SetClockHigh` method was received. This value is valid only if `HaveHigh` is set to True; otherwise it should be zeroes.

This column SHALL NOT be modifiable by the host.

5.5.3.1.6 HighLag

The value of this column is set by the `SetClockHigh` method. This value is valid only if `HaveHigh` is set to True; otherwise it should be zeroes. This represents seconds and fractions of a second.

This column SHALL NOT be modifiable by the host.

5.5.3.1.7 HaveLow

If the value of this column is True, then the values in the Low Trust time columns (`LowByWhom`, `LowSetTime`, `LowInitialTimer`, and `LowLag`) are meaningful. If `TrustMode` is High or Timer, then this value SHALL be False.

This column SHALL NOT be modifiable by the host.

5.5.3.1.8 LowByWhom

Authority that set the Low Trust time. This value is valid only if `HaveLow` is set to True; otherwise it should be zeroes. This is the uidref to the Authority that is the control authority of the session.

This column SHALL NOT be modifiable by the host.

5.5.3.1.9 LowSetTime

The value of this column is the time set to the value of the ExactTime parameter of the `SetClockLow` method when that method is successfully invoked. This value is valid only if `HaveLow` is set to True; otherwise it SHOULD be zeroes.

This column SHALL NOT be modifiable by the host.

5.5.3.1.10 LowInitialTimer

The value of this column is set to the value of the IncrementalClock when the ExactTime parameter of the `SetClockLow` method was received and processed. This value is valid only if `HaveLow` is set to True; otherwise it SHOULD be zeroes.

This column SHALL NOT be modifiable by the host.

5.5.3.1.11 LowLag

The value of this column is set by the `SetClockLow` method. This value is valid only if `HaveLow` is set to True; otherwise it SHOULD be zeroes. This represents seconds and fractions of a second.

This column SHALL NOT be modifiable by the host.

5.5.3.1.12 MonotonicBase

The monotonic time counter value is periodically saved here.

This column SHALL NOT be modifiable by the host.

5.5.3.1.13 MonotonicReserve

The value of this column indicates the frequency that the value of the `MonotonicBase` column is updated. The value of MonotonicIncrement is added to MonotonicBase whenever MonotonicIncrement == MonotonicReserve.

This column SHALL NOT be modifiable by the host.

5.5.3.1.14 TrustMode

This column value identifies whether `HaveHigh`, `HaveLow`, both, or neither are currently in effect.

5.5.4 Methods

The following section identifies methods that operate on the Clock.

5.5.4.1 GetClock (Table Method)

This method is used to fetch information about the current time. See 5.5.5.7.

Successful invocation of this method increments the MonotonicTime.

```
ClockTimeTableUID.GetClock [ ]
=>
[ Kind : clock_kind, ExactTime : clock_time, LagTime : lag, MonotonicTime : uinteger
]
```

5.5.4.1.1 GetClock Result

5.5.4.1.1.1 Kind

This value returns the type of time currently active.

5.5.4.1.1.2 ExactTime

This value is the current time stored in the `ClockTime` table. The value returned is dependent on the clock Kind that is currently active.

5.5.4.1.1.3 LagTime

This value returns the lag time associated with the current time stored in the `ClockTime` table. The value returned is dependent on the clock Kind that is currently active.

5.5.4.1.1.4 MonotonicTime

This value is the current value of the monotonic counter. See 5.5.5.2

5.5.4.1.2 Fails

a. If ClockTimeTableUID is not the uid of the `ClockTime` table

5.5.4.2 ResetClock (Table Method)

Successful invocation of this method resets the Clock Template-enabled SP's clock values and puts the SP into Timer mode. This method is invoked automatically when a TPer undergoes a hardware reset/power cycle.

```
ClockTimeTableUID.ResetClock [ ]
=>
[ ]
```

5.5.4.2.1 ResetClock Result

5.5.4.2.1.1 Result

The `ResetClock` method returns an empty list. Success or failure of the method invocation is determinable based on the status code returned in response to the method invocation.

5.5.4.2.2 Fails

a. If ClockTimeTableUID is not the uid of the `ClockTime` table

5.5.4.3 SetClockHigh (Table Method)

This is the first method in the method pair used to set the time from a High Trust source. For more information on setting High Trust time, see 5.5.5.1.2.

```
ClockTimeTableUID.SetClockHigh [
       ExactTime : clock_time ]
=>
[   ]
```

5.5.4.3.1 ExactTime

This is the time value sent by the host to be stored in the `ClockTime` table.

5.5.4.3.2 SetClockHigh Result

5.5.4.3.2.1 Result

The `SetClockHigh` method returns an empty list. Success or failure of the method invocation is determinable based on the status code returned in response to the method invocation.

5.5.4.3.3 Fails

a. If `ClockTimeTableUID` is not the uid of the `ClockTime` table.
b. If the `SetLagHigh` method is not received immediately after the `SetClockHigh` method.
c. If the value of `TrustMode` is not Low.

5.5.4.4 SetLagHigh (Table Method)

This is the second method in the method pair used to set the time from a High Trust source. For more information on setting High Trust time, see 5.5.5.1.2.

```
ClockTimeTableUID.SetLagHigh [
       LagTime : lag ]
=>
[ LowPreserved : boolean ]
```

5.5.4.4.1 LagTime

This value is the differential, in seconds and fractions of a second, recorded by the host application between the time when it read the time source and the time it received confirmation from the TPer that the time it sent was received.

5.5.4.4.2 SetLagHigh Result

5.5.4.4.2.1 LowPreserved

This value is a Boolean that returns True if Low Trust time values are preserved after setting of the High Trust time source, and returns False otherwise.

5.5.4.5 SetClockLow (Table Method)

This is the first method in the method pair used to set the time from a Low Trust source. This invocation is accepted only when the value of the `TrustMode` column is not “High”. For more information see 5.5.5.1.3.

```
ClockTimeTableUID.SetClockLow [
       ExactTime : clock_time ]
=>
[   ]
```

5.5.4.5.1 ExactTime

This is the time value sent by the host to be stored in the `ClockTime` table.

5.5.4.5.2 SetClockLow Result

5.5.4.5.2.1 Result

The `SetClockLow` method returns an empty list. Success or failure of the method invocation is determinable based on the status code returned in response to the method invocation.

5.5.4.5.3 Fails

a. If `ClockTimeTableUID` is not the uid of the `ClockTime` table.
b. If the `SetLagLow` method is not received immediately after the `SetClockLow` method.
c. If the values of `HighSetTime` and `HighLag` do not bracket the `SetClockLow` method's ExactTime combined with the `SetLagLow` method's LagTime.
d. If the value of the `TrustMode` column is not "High".

5.5.4.6 SetLagLow (Table Method)

This is the second method in the method pair used to set the time from a Low Trust source. For more information on setting High Trust time, see 5.5.5.1.3.

```
ClockTimeTableUID.SetLagLow [
       LagTime : lag ]
=>
[   ]
```

5.5.4.6.1 LagTime

This value is the differential, in seconds and fractions of a second, recorded by the host application between the time when it read the time source and the time it received confirmation from the TPer that the time it sent was received.

5.5.4.6.2 SetLagLow Result

5.5.4.6.2.1 Result

The `SetLagLow` method returns an empty list. Success or failure of the method invocation is determinable based on the status code returned in response to the method invocation.

5.5.4.7 IncrementCounter (Table Method)

This method increments and then returns the value of the monotonic counter. For more information see 5.5.5.2.

For two calls to `IncrementCounter`, the later call SHALL always return a value that is greater than that returned by the earlier call.

This method is permitted in a Read-Only session. The incrementing of the counter's value is not subject to transactional rollback.

```
ClockTimeUID.IncrementCounter [ ]
=>
[ MonotonicTime : uinteger ]
```

5.5.4.7.1 IncrementCounter Result

5.5.4.7.1.1 MonotonicTime

This method returns a uinteger that is the current value of the monotonic counter.

5.5.4.7.2 Fails

a. If `ClockTimeTableUID` is not the uid of the `ClockTime` table.

5.5.5 Descriptions

Begin Informative Content
The Clock Template enables an SP to keep track of date-time utilizing two time markers:

a. A time value called ExactTime that records a time stamp that is interpretable either in Generalized Time or UTC Time format.
b. An error value called LagTime

End Informative Content

5.5.5.1 Setting the Time

The SP that incorporates the Clock Template receives the time from a Host Application. It is expected that the Host Application or some process communicating through the Host Application monitors and records the time lag between the point when the Host Application reads the clock time from the source it is using to get the time, and the point when the Host Application receives confirmation from the SP that the value has been received.

The Host Application then sends to the SP the lag that it has recorded and, on receipt of this value, the SP records both the time and the lag in the `ClockTime` table. In this way, the SP has a value for the time and is able to bracket the error.

5.5.5.1.1 High Trust vs. Low Trust

A distinction is made between time from a High Trust source and time from a Low Trust source. A High Trust source MAY be a remote but strongly protected source of time. A Low Trust source MAY be an immediate but not strongly protected source of time, such as the local PC clock.

The High Trust source is expected to be able to provide a more authoritative time, but with a larger lag, so the High Trust source is used to bracket the Low Trust source. In this way, a Low Trust but accurate time MAY also be detected and used.

When the TrustMode is LowAndHigh and both High Trust and Low Trust values are present, then the Low Trust time is rejected if it isn't confirmed by the High Trust time. Specifically, the following should be true (See 5.5.5.3 for descriptions of LowTime and HighTime):

a. LowTime $\geq$ HighTime.
b. LowTime + LowLag $\leq$ HighTime + HighLag.

If either of these conditions is not true, the Low Trust value is discarded because it is probably wrong. This means that if Low Trust values exist in the `ClockTime` table, a `SetClockHigh` method invocation is received, and either of the above conditions is false, then the Low Trust values are set to 0; or, if High Trust values exist in the `ClockTime` table and a `SetClockLow` method invocation is received, the method invocation fails if either of the above conditions is not true.

The SP that incorporates the Clock Template MAY accept a Low Trust time with or without an existing High Trust bracket, or just a High Trust time.

5.5.5.1.2 Setting High Trust Time

High Trust Time is set using the `SetClockHigh`/`SetLagHigh` method pair.

The invocation of these methods operate as follows:

1. The host invokes the SetClockHigh method on the ClockTime table. The **ExactTime** input is received and processed. At this time the value that is eventually used to set HighInitialTimer should be computed by the SP, by reading the IncrementalClock value.
2. The SP returns a Result of "True", indicating that the ExactTime value was received and processed.
3. The host invokes the SetLagHigh method on the ClockTime table. This method SHALL be the next method invoked by the host after invocation of the SetClockHigh method invocation. If it is not, the SetClockHigh method's ExactTime values to be stored in the ClockTime table SHALL NOT be saved to the table. When the SetLagHigh method is received under this condition, the LagTime input is received and processed. If this invocation is accepted, then
    1. The value of the HaveHigh column is set to True.
    2. The value of the HighByWhom column is set to the authority for this session.
    3. The value of the HighSetTime column is set to ExactTime.
    4. The value of the HighInitialTimer column is set to the previously read IncrementalClock value.
    5. The value of the HighLag column is set to LagTime.
4. If the new HighSetTime and HighLag values do not bracket existing LowSetTime and LowLag values, then the value of the HaveLow column is set to False and the values of the LowByWhom, LowSetTime, LowInitialTimer, and LowLag columns are set to zeroes. For additional information, see 5.5.5.1.1.

Once all of the above steps have been processed, the SP SHALL return the method result. If step 3 was completed successfully, the Result is returned as True. If any of the updates in step 3 were not successfully completed, Result is returned as False. If the HaveLow column is not set to False from True and the LowByWhom, LowSetTime, LowInitialTimer and LowLag columns are not set to zeroes due to the described bracketing, then LowPreserved is returned as True. Otherwise, Low is returned as False.

5.5.5.1.3 Setting Low Trust TIme

Low Trust Time is set using the SetClockLow/SetLagLow method pair.

The invocation of these methods operate as follows:

1. The host invokes the SetClockLow method on the ClockTime table. The ExactTime input is received and processed. At this time the value that is eventually used to set LowInitialTimer should be computed by the SP by reading the IncrementalClock value.
2. The SP returns a Result of "True", indicating that the ExactTime value was received and processed.
3. The host invokes the SetLagLow method on the ClockTime table. This method SHALL be the next method invoked by the host after invocation of the SetClockLow method invocation. If it is not, the SetClockLow method's ExactTime values to be stored in the ClockTime table SHALL NOT be saved to the table. When the SetLagLow method is received under this condition, the LagTime input is received and processed. If this invocation is accepted, then
    1. The value of the HaveLow column is set to True.
    2. The value of the LowByWhom column is set to the authority for this session.
    3. The value of the LowSetTime column is set to ExactTime.
    4. The value of the LowInitialTimer column is set to the previously read IncrementalClock value.

5. The value of the LowLag column is set to LagTime

4. If the value of TrustMode is LowAndHigh and HaveHigh is True, then this call SHALL be accepted only when the existing HighSetTime and HighLag values bracket the new LowSetTime and LowLag values. For additional information, see 5.5.5.1.1.

Once all of the above steps have been processed, the SP SHALL return the method result. If step 3 was completed successfully and the condition noted in step 4 was met, the updates are made to the ClockTime table and the Result is returned as True. If any of the updates in step 3 were not successfully completed or the condition noted in Step 4 was not met, Result is returned as False.

5.5.5.2 Monotonic Counter

An SP that incorporates the Clock Template also SHALL independently maintain a counter that increments every time a clock time is read – this is a 64-bit persistent counter called MonotonicTime. The counter is incremented independent of transactions and of the Read/Write state of the session.

This counter is needed for clock requests that require a counter, since it is possible to have the SP time set back in time, and to enable differentiation between multiple requests received at the same clock time.

For each SP that incorporates the Clock Template, there SHALL also be a counter kept in main memory called MonotonicIncrement. This counter is used to reduce the number of writes to media that are needed to support the MonotonicTime.

The value of the virtual variable MonotonicTime that the user sees (via the IncrementCounter or GetClock methods) SHALL be:

MonotonicTime = MonotonicBase + MonotonicIncrement

The following is always true:

0 ≤ MonotonicIncrement ≤ MonotonicReserve

Note that in this case, MonotonicBase and MonotonicReserve are not necessarily the values stored in the ClockTime table. Rather, these values of MonotonicBase and MonotonicReserve are written to media only as needed to guarantee that the IncrementCounter method always returns a unique value after a power cycle, etc.

In order to reduce writes to media, the MonotonicBase value stored in the ClockTime table is only occasionally updated. This is controlled by the value in the MonotonicReserve column of the ClockTime table.

The host MAY increment the Monotonic Counter value directly via invocation of the IncrementCounter method. This is done as follows:

```
if ++MonotonicIncrement == MonotonicReserve
        MonotonicBase += MonotonicReserve
        MonotonicIncrement = 0
        ClockTime.MonotonicBase = MonotonicBase
return MonotonicTime = MonotonicBase + MonotonicIncrement
```

5.5.5.3 Incremental Clock

Each TPer SHALL have a quickly accessible incremental clock. This is referred to as IncrementalClock. Although this clock does not have the correct absolute time, it is accurate in measuring time intervals.

To support Host interaction with the Clock Template-enabled SP, two virtual variables – HighTime and LowTime – are used. HighTime and LowTime internally represent actual current time values.

Calculation of the values of HighTime and LowTime uses the original time set (the value of the HighSetTime or LowSetTime columns), the value of IncrementalClock when those columns were set (the value of the HighInitialTimer or LowInitialTimer columns), and the current value of IncrementalClock:

HighTime = HighSetTime + (IncrementalClock – HighInitialTimer)

LowTime = LowSetTime + (IncrementalClock – LowInitialTimer)

The HighTime value is changed to new value *v* as follows (as when a SetClockHigh or SetClockLow method invocation is received):

HighSetTime = v

HighInitialTimer = IncrementalClock

The LowTime virtual variable is changed similarly.

This approach avoids the need to update the media as the value of IncrementalClock changes.

Some TPers MAY also include other special hardware that is used to implement the Clock Template. These include a real-time clock (with battery backup) and non-volatile memory that is used to store monotonic counter values.

5.5.5.4 Timer Mode

The Clock Template provides an additional time mode, Timer Mode, to identify when the time has been un-set after a disk controller reset or if the SP has never had a time set.

After a TPer reset or upon issuance, the SP is in Timer mode. In Timer mode, the time is incremented, but a successful invocation of the GetClock method SHALL return a clock_kind of “Timer”, the values of the IncrementalClock and MonotonicTime, and a LagTime of 0. This indicates that the time value is not able to be trusted as an absolute because of the reset.

The ResetClock method is invoked at power up or after a TPer reset, before the SP that incorporates the Clock Template is accessible. This places the TPer into Timer Mode.

If the TPer has a real-time clock, the TPer SHALL use that value at power up or after a TPer reset while the real-time clock has power. Otherwise, the TPer reverts to the behavior previously described.

5.5.5.5 Storing Time

The clock_time data type is used to represent time in the ClockTime and other tables. This type MAY be used to represent either UTC or Generalized time.

5.5.5.6 Storing LagTime

LagTime is stored in the ClockTime table and represented as a method parameter or return result by the lag_time abstract type.

5.5.5.7 Reading the Time

The current time values stored by the Clock Template-enabled SP are retrieved using the GetClock method. The values returned are dependant on the trust values of the time stored.

If the value of the HaveLow column is True and the value of the HaveHigh column is False, then the result of the GetClock method invocation, in pseudo code, are [“Low”, LowTime, LowLag, MonotonicTime].

If the value of the HaveHigh column is True, then the result of the GetClock method invocation, in pseudo code, are [“High”, HighTime, HighLag, MonotonicTime].

If the value of both the HaveLow and HaveHigh columns is False then the result, in pseudo code, is [“Timer”, IncrementalClock, 0, MonotonicTime].

5.5.5.8 Resetting the Clock

The time values in a Clock Template-enabled SP are reset using the ResetClock method.

The following properties are set when this method is invoked:

a. The `HaveHigh` column of the `ClockTime` table is set to False, and the values of the `HighByWhom, HighSetTime`, `HighInitialTimer`, and `HighLag` columns are set to zeroes.

b. The `HaveLow` column of the `ClockTime` table is set to False, and the values of the `LowByWhom, LowSetTime`, `LowInitialTimer`, and `LowLag` columns are set to zeroes.

c. MonotonicIncrement = 0

d. `ClockTime.TrustMode` = Timer

e. If `ClockTime.MonotonicReserve` == `0x00`

   a. `ClockTime.MonotonicReserve` = <some small value, e.g. 100>

   b. `ClockTime.MonotonicBase` = `ClockTime.MonotonicBase` + `ClockTime.MonotonicReserve`

   c. MonotonicBase = `ClockTime.MonotonicBase`

   d. MonotonicReserve = `ClockTime.MonotonicReserve`

Note that this guarantees that the MonotonicTime value always increases (although it MAY, in a `ResetClock`, skip up to the value of MonotonicReserve).

5.5.5.9 Default Logging Settings

The default logging settings associated with the Clock Template methods are:

a. The default logging for all Clock Template-enabled methods (`ResetClock`, `SetClockHigh`, `SetClockLow`, `IncrementCounter`) is LogAlways.

b. All other methods that apply to the `ClockTime` table are as described in the Base Template reference section (See Section 5.3.4.4).

5.5.6 Life Cycle

5.5.6.1 Clock Template-Specific Life Cycle State Descriptions/Exceptions

An SP issued with the Clock Template has the following characteristics based on the current life cycle state of that SP:

a. **Disabled** – If the Clock Template-enabled SP has entered the Disabled state, the SP SHALL log authentication, session startup, and method invocation attempts, if the Log Template has been issued into the SP. These log entries SHALL have timestamps of the kind appropriate to that log entry. TPer resets SHALL cause the SP's `ClockTime` table to revert to Timer mode. Log entries added while the SP is in the Disabled state SHALL NOT be retrievable – see other behaviors of SPs in the Disabled state, as described in section 4.5.2.

b. **Frozen** – Attempts to open sessions to an SP in the Issued-Frozen state SHALL fail.

c. **Issued-Disabled-Frozen** – Attempts to open sessions to an SP in the Issued-Disabled-Frozen state SHALL fail.

5.6 Crypto Template

5.6.1 Overview

Begin Informative Content

The Crypto Template provides a set of cryptographic methods that operate on public and symmetric key store tables, collectively called Credential tables, provided by the Base and other Templates. The Crypto Template also provides a set of tables that supports these methods.

The set of cryptographic methods that the Crypto Template provides support functionality that includes Encryption, Decryption, Signing, Verifying, Hashing, HMAC, and XOR. Other Templates MAY provide Credential tables to an SP. Credential tables in an SP that does not incorporate the Crypto Template MAY be key stores or contain credentials for use in media encryption, secure messaging, and authentication. Incorporating the Crypto Template into an SP enables the host to perform encryption, decryption, signing, and verification on the TPer, using keys and data stored on the TPer.

End Informative Content

5.6.2 Terminology

Table 219 Crypto Template Terminology

Term	Definition
"stream"	The term "stream", used in quotation marks in this section, is not related to session or messaging streams. Rather, this term is used to identify a single operational context related to a particular cryptographic operation. A "stream" is created using an initialization method, is operated on by one or more calculation methods of the type appropriate to the initialized "stream", and is closed by a finalization method. A particular context SHALL deal only with the operation associated with it (encrypt, decrypt, HMAC, or hash).

5.6.3 Data Structures

Begin Informative Content

The Crypto Template provides tables similar to the Credential tables described by the Base and other Templates. However, unlike those Credential tables, which represent key stores for authentication associations, the Crypto Template's tables are Credential support tables optimized for incremental on-TPer operations.

End Informative Content

5.6.3.1 Cryptographic Support Group - H_SHA_1 (Object Table)

This section describes the support table for use with SHA-1 hashing operations. Objects in this table are used with the `HashInit`, `Hash`, `HashFinalize`, `HMACInit`, `HMAC`, `HMACFinalize`, `Sign`, and `Verify` methods.

Table 220 H_SHA_1 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	Proof		bytes_20
0x04	Accumulator		bytes_20
0x05	Signer		cred_object_uidref

5.6.3.1.1 UID

This is the unique identifier of this row in the table.

This column SHALL NOT be modifiable by the host.

5.6.3.1.2 Name

This is the name of the object.

For H_SHA_1 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.6.3.1.3 CommonName

This is a name that MAY be shared by multiple H_SHA_1 objects.

For H_SHA_1 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.6.3.1.4 Proof

This is the proof to be checked when the Verify method is invoked on this credential object; or this is the proof to be created when the Sign method is invoked on this Credential object.

5.6.3.1.5 Accumulator

This is the accumulator where a new hash is incrementally created upon invocation of the Hash or HMAC methods on this credential, or where an initial condition is set.

5.6.3.1.6 Signer

This is a uidref to the signing/verification credential object. This is the signing credential whose public key or symmetric key decrypts the proof to reveal the proof's underlying hash when the Verify method is invoked; or whose private or symmetric key encrypts the proof when the Sign method is invoked on this credential; or whose HMAC key is used when the HMAC methods are invoked on this credential.

5.6.3.2 Cryptographic Support Group - H_SHA_256 (Object Table)

This section describes the support table for use with SHA-256 hashing operations. Objects in this table are used with the HashInit, Hash, HashFinalize, HMACInit, HMAC, HMACFinalize, Sign, and Verify methods.

Table 221 H_SHA_256 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	Proof		bytes_32
0x04	Accumulator		bytes_32
0x05	Signer		cred_object_uidref

5.6.3.2.1 UID

This is the unique identifier of this row in the table.

This column SHALL NOT be modifiable by the host.

5.6.3.2.2 Name

This is the name of the object.

For H_SHA_256 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.6.3.2.3 CommonName

This is a name that MAY be shared by multiple H_SHA_256 objects.

For H_SHA_256 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.6.3.2.4 Proof

This is the proof to be checked when the Verify method is invoked on this credential object; or this is the proof to be created when the Sign method is invoked on this Credential object.

5.6.3.2.5 Accumulator

This is the accumulator where a new hash is incrementally created upon invocation of the Hash or HMAC methods on this credential, or where an initial condition is set.

5.6.3.2.6 Signer

This is a uidref to the signing/verification credential object. This is the signing credential whose public key or symmetric key decrypts the proof to reveal the proof's underlying hash when the Verify method is invoked; or whose private or symmetric key encrypts the proof when the Sign method is invoked on this credential; or whose HMAC key is used when the HMAC methods are invoked on this credential.

5.6.3.3 Cryptographic Support Group - H_SHA_384 (Object Table)

This section describes the support table for use with SHA-384 hashing operations. Objects in this table are used with the HashInit, Hash, HashFinalize, HMACInit, HMAC, HMACFinalize, Sign, and Verify methods.

Table 222 H_SHA_384 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	Proof		bytes_48
0x04	Accumulator		bytes_48
0x05	Signer		cred_object_uidref

5.6.3.3.1 UID

This is the unique identifier of this row in the table.

This column SHALL NOT be modifiable by the host.

5.6.3.3.2 Name

This is the name of the object.

For H_SHA_384 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.6.3.3.3 CommonName

This is a name that MAY be shared by multiple H_SHA_384 objects.

For H_SHA_384 objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.6.3.3.4 Proof

This is the proof to be checked when the Verify method is invoked on this credential object; or this is the proof to be created when the Sign method is invoked on this Credential object.

5.6.3.3.5 Accumulator

This is the accumulator where a new hash is incrementally created upon invocation of the Hash or HMAC methods on this credential, or where an initial condition is set.

5.6.3.3.6 Signer

This is a uidref to the signing/verification credential object. This is the signing credential whose public key or symmetric key decrypts the proof to reveal the proof's underlying hash when the `Verify` method is invoked; or whose private or symmetric key encrypts the proof when the `Sign` method is invoked on this credential; or whose HMAC key is used when the HMAC methods are invoked on this credential.

5.6.3.4 Cryptographic Support Group - H_SHA_512 (Object Table)

This section describes the support table for use with SHA-512 hashing operations. Objects in this table are used with the `HashInit`, `Hash`, `HashFinalize`, `HMACInit`, `HMAC`, `HMACFinalize`, `Sign`, and `Verify` methods.

Table 223 H_SHA_512 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	Proof		bytes_64
0x04	Accumulator		bytes_64
0x05	Signer		cred_object_uidref

5.6.3.4.1 UID

This is the unique identifier of this row in the table.

This column SHALL NOT be modifiable by the host.

5.6.3.4.2 Name

This is the name of the object.

For `H_SHA_512` objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.6.3.4.3 CommonName

This is a name that MAY be shared by multiple `H_SHA_512` objects.

For `H_SHA_512` objects that exist at issuance, this column SHALL NOT be modifiable by the host.

5.6.3.4.4 Proof

This is the proof to be checked when the `Verify` method is invoked on this credential object; or this is the proof to be created when the `Sign` method is invoked on this Credential object. T

5.6.3.4.5 Accumulator

This is the accumulator where a new hash is incrementally created upon invocation of the Hash or HMAC methods on this credential, or where an initial condition is set.

5.6.3.4.6 Signer

This is a uidref to the signing/verification credential object. This is the signing credential whose public key or symmetric key decrypts the proof to reveal the proof's underlying hash when the `Verify` method is invoked; or whose private or symmetric key encrypts the proof when the `Sign` method is invoked on this credential; or whose HMAC key is used when the HMAC methods are invoked on this credential.

5.6.4 Methods

5.6.4.1 Random Number Related Method Group - Random (SP Method)

This section describes the method used to generate random numbers. This method returns a sequence of random bytes of a specified size. The quality of random numbers generated is under the purview of the conformance profile.

```
ThisSP.Random[
Count : uinteger,
BufferOut = cell_block ]
=>
[ Result : bytes ]
```

5.6.4.1.1 Count

This parameter specifies the size, bytes, of the sequence of random bytes to be generated.

5.6.4.1.2 BufferOut

This value identifies a specific cell or range of cells to which the generated bytes are stored.

All cells identified by the BufferOut parameter and to be written with the method result SHALL be of type bytes. Cells are filled from the lowest column number to the highest column number.

5.6.4.1.3 Random Result

5.6.4.1.3.1 Result

This is the value randomly generated by invocation of this method. If the BufferOut parameter is specified, Result SHALL be empty.

5.6.4.2 Random Number Related Method Group – Stir (SP Method)

The purpose of this method is to add additional information for use by the `Random` method for subsequent invocations of that method.

```
ThisSP.Stir[
       Value : typeOr { Input : bytes, Internal : boolean } ]
=>
[   ]
```

5.6.4.2.1 Value

5.6.4.2.1.1 Input

Invocation of the `Stir` method with the bytes Input parameter allows the host to pass a string of bytes of its choice as the information to be added for use by the `Random` method.

5.6.4.2.1.2 Internal

Invocation of the `Stir` method with the Boolean Internal parameter indicates that the TPer should generate the information to be used by the `Random` method.

Invocation of the `Stir` method with a Value parameter of False SHALL result in the method invocation failing and returning a non-success status code.

5.6.4.2.2 Fails

a. BufferOut is not big enough to hold "Count" bytes
b. `Set` access control is not satisfied for BufferOut
c. BufferOut does not reference a byte table or references a cell that is not of type bytes

5.6.4.2.3 Stir Result

5.6.4.2.3.1 Result

The `Stir` method returns an empty list. Success or failure of the method invocation is determinable based on the status code returned in response to the method invocation.

5.6.4.3 Decryption Method Group – DecryptInit (Object Method)

This method is used to initiate a decryption "stream" using the credential object that invoked the method. Only one decryption "stream" SHALL be able to be open at any one time for any individual credential object.

```
CredentialObjectUID.DecryptInit [
IV = bytes ]
=>
[   ]
```

5.6.4.3.1 IV

If the **IV** parameter is included, the parameterized IV is used in place of that which is stored in the credential object itself.

5.6.4.3.2 DecriptInit Result

5.6.4.3.2.1 Result

The `DecryptInit` method returns an empty list. Success or failure of the method invocation is determinable based on the status code returned in response to the method invocation.

5.6.4.3.3 Fails

a. If the object does not exist
b. If the object does not contain a valid credential
c. If the object currently already has a decryption "stream" open

5.6.4.4 Decryption Method Group - Decrypt (Object Method)

The `Decrypt` method causes the TPer to perform decryption on the data supplied using the credential object that invoked the method. The value passed upon invocation of the `Decrypt` method is decrypted using the key in the specified credential.

This method SHALL require that the `DecryptInit` method has been invoked previously during the session to start a decryption "stream" for the invoking credential object, and that the `DecryptFinalize` method has not yet been invoked to close that "stream". Invoking the `Decrypt` method when there is no open decryption "stream" (i.e. before `DecryptInit` or after `DecryptFinalize`) SHALL result in an error.

```
CredentialObjectUID.Decrypt [
Input : typeOr { Data : bytes, Buffer : cell_block },
BufferOut = cell_block ]
=>
[ Result : bytes ]
```

5.6.4.4.1 Input

The value of the **Input** parameter MAY be either a bytes value, wherein bytes to be decrypted are passed as a parameter of the method; or a cellblock that addresses a subset of a table on the SP that holds the data to be decrypted.

5.6.4.4.1.1 Data

This input parameter is used to supply bytes directly from the host for decryption. The required length of the bytes is dependant upon the mode of operation selected for the credential. Should padding be required, the host SHALL perform it.

5.6.4.4.1.2 Buffer

This input parameter is used to identify a block of table cells where the data to be decrypted is stored. If the host invokes the `Decrypt` method using the data addressed via the cellblock as data input, then in addition to fulfilling the access control on the `Decrypt` method, the host SHALL also fulfill the access control required to invoke the `Get` method on the entirety of that cellblock.

5.6.4.4.2 BufferOut

This parameter identifies the cells to which the decrypted data should be set. If this value is specified, the Input byte length SHALL be equal in size to or smaller than the cellblock specified for the Result.

If the host invokes the `Decrypt` method using the BufferOut cellblock as the target for the result bytes, then in addition to fulfilling the access control on the `Decrypt` method, the host SHALL also fulfill the access control required to invoke the `Set` method on the entirety of that cellblock.

5.6.4.4.3 Decrypt Result

5.6.4.4.3.1 Result

The result of the invocation of this method is the plaintext value of the bytes submitted for decryption. If the BufferOut parameter is specified, the method Result SHALL be empty.

5.6.4.4.4 Fails

a. If the object does not exist
b. If the object does not contain a valid credential
c. If the DataInput cellblock reference is not a to valid cellblock
d. If the DataInput is a cellblock reference and Get access control on that cellblock has not been fulfilled
e. If the BufferOut is not a valid cellblock
f. If BufferOut has been specified and Set access control on that cellblock has not been fulfilled
g. If the DataInput byte size is not the same size as or smaller than the BufferOut cell size (if specified)
h. If Decrypt has been invoked when no decryption “stream” is open

5.6.4.5 Decryption Method Group – DecryptFinalize (Object Method)

Invocation of this method closes the decryption “stream” associated with this object.

```
CredentialObjectUID.DecryptFinalize [ ]
=>
[ Result : bytes ]
```

5.6.4.5.1 DecryptFinalize Result

5.6.4.5.1.1 Result

The result of this method is any remaining decrypted data from the previous `Decrypt` method invocation that has not yet been returned to the host.

5.6.4.5.2 Fails

a. If the object does not exist
b. If the object does not contain a valid credential
c. If there is no decryption “stream” open for this credential object

5.6.4.6 Encryption Method Group – EncryptInit (Object Method)

This method is used to initiate an encryption "stream" using the credential object that invoked the method. Only one encryption "stream" SHALL be able to be open at any one time for any individual credential object.

```
CredentialObjectUID.EncryptInit [
IV = bytes ]
=>
[  ]
```

5.6.4.6.1 IV

If the IV parameter is included, the parameterized IV is used in place of that which MAY be stored in the credential object itself.

5.6.4.6.2 EncryptInit Result

5.6.4.6.2.1 Result

The `EncryptInit` method returns an empty list. Success or failure of the method invocation is determinable based on the status code returned in response to the method invocation.

5.6.4.6.3 Fails

a. If the object does not exist
b. If the object does not contain a valid credential
c. If the object currently already has an encryption "stream" open

5.6.4.7 Encrytion Method Group - Encrypt (Object Method)

The `Encrypt` method causes the TPer to perform encryption on the data supplied using the credential object that invoked the method. The value passed upon invocation of the `Encrypt` method is encrypted using the key in the specified credential.

This method SHALL require that the `EncryptInit` method has been invoked previously during the session to start an encryption "stream" for the invoking credential object, and that the `EncryptFinalize` method has not yet been invoked to close that "stream". Invoking the `Encrypt` method when there is no open encryption "stream" (i.e. before `EncryptInit` or after `EncryptFinalize`) SHALL result in an error.

```
CredentialObjectUID.Encrypt [
Input : typeOr { Data : bytes, Buffer : cell_block },
BufferOut = cell_block ]
=>
[ Result : bytes ]
```

5.6.4.7.1 Input

The value of the Input parameter MAY be either a bytes value, wherein bytes to be encrypted are passed as a parameter of the method; or a cellblock that addresses a subset of a table on the SP that holds the data to be decrypted.

5.6.4.7.1.1 Data

This input parameter is used to supply bytes directly from the host for encryption. The required length of the bytes is dependant upon the mode of operation selected for the credential. Should padding be required, the host SHALL perform it.

5.6.4.7.1.2 Buffer

This input parameter is used to identify a block of table cells where the data to be encrypted is stored. If the host invokes the Encrypt method using the data addressed via the cellblock as data input, then in addition to fulfilling the access control on the Encrypt method, the host SHALL also fulfill the access control required to invoke the Get method on the entirety of that cellblock.

5.6.4.7.2 BufferOut

This parameter identifies the cells to which the encrypted data is to be set. If this value is specified, the Input byte length SHALL be equal in size to or smaller than the cellblock specified for the Result.

If the host invokes the Encrypt method using the BufferOut cellblock as the target for the result bytes, then in addition to fulfilling the access control on the Encrypt method, the host SHALL also fulfill the access control required to invoke the Set method on the entirety of that cellblock.

5.6.4.7.3 Encrypt Result

5.6.4.7.3.1 Result

The result of the invocation of this method is the ciphertext value of the bytes submitted for encryption. If the BufferOut parameter is specified, the method Result SHALL be empty.

5.6.4.7.4 Fails

a. If the object does not exist
b. If the object does not contain a valid credential
c. If the DataInput cellblock reference is not a to valid cellblock
d. If the DataInput is a cellblock reference and Get access control on that cellblock has not been fulfilled
e. If the BufferOut is not a valid cellblock
f. If BufferOut has been specified and Set access control on that cellblock has not been fulfilled
g. If the DataInput byte size is not the same size as or smaller than the Result cell size (if specified).
h. If Encrypt has been invoked when no encryption “stream” is open

5.6.4.8 Encryption Method Group – EncryptFinalize (Object Method)

Invocation of this method closes the encryption “stream” associated with this object.

```
CredentialObjectUID.EncryptFinalize [ ]
=>
[ Result : bytes]
```

5.6.4.8.1 EncryptFinalize Result

5.6.4.8.1.1 Result

The result of this method is any remaining encrypted data from the previous Encrypt method invocation that has not yet been returned to the host.

5.6.4.8.2 Fails

a. If the object does not exist
b. If the object does not contain a valid credential
c. If there is no encryption “stream” open for this credential object

5.6.4.9 Sign (Object Method)

This method is used to sign a data input using the private part of a public-private key pair.

```
CredentialObjectUID.Sign
HashObjectUID.Sign[
Input = typeOr { Data : bytes, Buffer : cell_block },
BufferOut = cell_block ]
=>
[ Result : bytes ]
```

5.6.4.9.1 Input

The value of the Input parameter MAY be either a bytes value, wherein bytes to be signed are passed as a parameter of the method; or a cellblock that addresses a subset of a table on the SP that holds the data to be signed.

For the `Sign` method invoked on an asymmetric credential object, the Input value is signed using the private part of the key pair of the specified public key credential.

For the `Sign` method invoked on a hash object, the data in the hash object's `Accumulator` column is signed using the private part of the key pair of the public key credential referenced in the hash object's `Signer` column. It is an error for the `Sign` method to be invoked on a hash object and have the Input parameter specified.

When invoking `Sign` on a hash object, access control on the `Sign` method for the credential referenced in the `Signer` column of the hash object SHALL be fulfilled.

5.6.4.9.1.1 Data

This input parameter is used to supply bytes directly from the host for signing. Should padding be required, the host SHALL perform it.

5.6.4.9.1.2 Buffer

This input parameter is used to identify a block of table cells where the data to be signed is stored. If the host invokes the `Sign` method using the data addressed via the cellblock as data input, then in addition to fulfilling the access control on the `Sign` method, the host SHALL also fulfill the access control required to invoke the `Get` method on the entirety of that cellblock.

5.6.4.9.2 BufferOut

This parameter identifies the cells to which the signed data should be set. If the this value is specified, the Input byte length SHALL be equal in size to or smaller than the cellblock specified for the Result.

If this parameter is specified as the target of the method result, in addition to fulfilling the access control requirements to invoke the `Sign` method, the host SHALL also fulfill the access control requirements necessary to invoke the `Set` method on the entirety of the specified cellblock.

5.6.4.9.3 Sign Result

5.6.4.9.3.1 Result

The result of the invocation of this method is the signed bytes that were submitted to the method. If the BufferOut parameter is specified, the method Result SHALL be empty.

5.6.4.9.4 Fails

a. If the invoking object does not exist
b. If the `Sign` method is invoked on a hash object and that object does not reference a valid public key credential (RSA, EC)
c. If the invoking credential, or the credential referenced from the hash object, does not contain a valid private key
d. If the DataInput cellblock reference is not a to valid cellblock
e. If the host application has not fulfilled the access control requirements necessary to invoke the `Get` method on the DataInput cellblock

f. If the BufferOut is not a valid cellblock
g. If the host application has not fulfilled the access control requirements necessary to invoke the `Set` method on the BufferOut cellblock

5.6.4.10 Verify (Object Method)

This method MAY be invoked on a hash object or a public key credential. It is used to verify a signed hash against a proof.

```
CredentialObjectUID.Verify [
HashObjectUID.Verify [
Input : typeOr { Data : bytes, Buffer : cell_block},
Data : typeOr { Proof : bytes, ProofBuffer : cell_block } ]
=>
[ Result : boolean ]
```

5.6.4.10.1 Input

The value of the **Input** parameter MAY be either a bytes value, wherein bytes to be verified are passed as a parameter of the method; or a cellblock that addresses a subset of a table on the SP that holds the data to be verified.

5.6.4.10.1.1 Data

This input parameter is used to supply bytes directly from the host for verification.

5.6.4.10.1.2 Buffer

This input parameter is used to identify a block of table cells where the data to be verified is stored. If the host invokes the `Verify` method using the data addressed via the cellblock as data input, then in addition to fulfilling the access control on the `Verify` method, the host SHALL also fulfill the access control required to invoke the `Get` method on the entirety of that cellblock.

5.6.4.10.2 Data

The value of the Data parameter identifies the data that the Input is to be compared against. The value of this parameter MAY be either a bytes value, wherein the proof data bytes are passed as a parameter of the method; or a cellblock that addresses a subset of a table on the SP that holds the proof data.

5.6.4.10.2.1 Proof

This input parameter is used to supply bytes directly from the host against which the data input is to be verified.

5.6.4.10.2.2 ProofBuffer

This input parameter is used to identify a block of table cells where the data to be verified against stored. If the host invokes the `Verify` method using the data addressed via the cellblock as data input, then in addition to fulfilling the access control on the `Verify` method, the host SHALL also fulfill the access control required to invoke the `Get` method on the entirety of that cellblock.

5.6.4.10.3 Verify Result

5.6.4.10.3.1 Result

The result of this method invocation is a Boolean value that is True if the verification of the data against the proof is successful, and False otherwise.

5.6.4.10.4 Fails

a. If the invoking object does not exist.
b. If the invoking credential object is not a valid public key credential (RSA, EC).
c. If the invoking hash object does not reference a public key credential.

d. If the host application has not fulfilled the access control requirements necessary to invoke the Get method on the DataInput or Proof cellblock.
e. If the DataInput or Proof are not valid cellblocks.

5.6.4.11 Hash Method Group – HashInit (Object Method)

This method is used to initiate a hash "stream" using the hash object that invoked the method. Only one hash "stream" SHALL be able to be open at any one time for any individual hash object.

Invocation of this method is required before the Hash method MAY be successfully invoked. In preparation for beginning the hash "stream", upon successful invocation of the Hashinit method, the invoking hash object's Accumulator column is set to zero.

```
HashObjectUID.HashInit [
       BufferOut = cell_block ]
=>
[   ]
```

5.6.4.11.1 BufferOut

This parameter is used to identify a block of table cells to which the hashed data is to be set. If this value is specified, the Input byte length SHALL be equal in size to or smaller than the cellblock specified for the Result.

If the host invokes this method using the BufferOut cellblock as the target for the result, then in addition to fulfilling the access control on the Hash method, the host SHALL also fulfill the access control required to invoke the Set method on the entirety of that cellblock.

5.6.4.11.2 HashInit Result

5.6.4.11.2.1 Result

The HashInit method returns an empty list. Success or failure of the method invocation is determinable based on the status code returned in response to the method invocation.

5.6.4.11.3 Fails

a. If the hash object does not exist
b. If BufferOut has been specified and is not a valid cellblock
c. If BufferOut has been specified and is not larger than or equal to the size of the hash calculation result
d. If BufferOut has been specified and Set access control on that cellblock has not been fulfilled
e. If the hash object currently already has a hash "stream" open

5.6.4.12 Hash Method Group – Hash (Object Method)

Invocation of the Hash method causes the data supplied in the Input parameter to be hashed. The TPer hashes the data on block boundaries as they are reached.

This method SHALL require that the HashInit method has been invoked previously during the session to start a hash "stream" for the invoking hash object, and that the HashFinalize method has not yet been invoked to close that "stream". Invoking the Hash method when there is not an open hash "stream" (i.e. before HashInit or after HashFinalize) SHALL result in an error.

```
HashObjectUID.Hash [
Input : typeOr { Data : bytes, BufferIn : cell_block } ]
=>
[ Result : bytes ]
```

5.6.4.12.1 Input

The value of the Input parameter MAY be either a bytes value, wherein bytes to be hashed are passed as a parameter of the method; or a cellblock that addresses a subset of a table on the SP that holds the data to be hashed.

5.6.4.12.1.1 Data

This input parameter is used to supply bytes directly from the host for hashing.

5.6.4.12.1.2 BufferIn

This input parameter is used to identify a block of table cells where the data to be hashed is stored. If the host invokes the `Hash` method using the data addressed via the cellblock as data input, then in addition to fulfilling the access control on the `Hash` method, the host SHALL also fulfill the access control required to invoke the `Get` method on the entirety of that cellblock.

5.6.4.12.2 Hash Result

5.6.4.12.2.1 Result

The method returns the data input that has been consumed in the hash as the method result.

Results of the hashing operation are stored in the BufferOut cellblock of the `HashInit` method, if that parameter is included. Otherwise the results are stored in the invoking hash object's `Accumulator` column. If the BufferOut parameter is specified in the `HashInit` method, the `Hash` method Result SHALL be empty.

5.6.4.12.3 Fails

a. If the object does not exist
b. If the DataInput cellblock reference is not to a valid cellblock
c. If the DataInput parameter references a cellblock and `Get` access control on that cellblock has not been fulfilled
d. If `Hash` has been invoked when no hash "stream" is open
e. If `Hash` is unable to write to BufferOut cellblock

5.6.4.13 Hash Method Group – HashFinalize (Object Method)

Invocation of the `HashFinalize` method causes the TPer to flush the remaining, non-blocked data through the hash and sets the BufferOut cellblock specified in the `HashInit` method. If the BufferOut cellblock was not supplied to the `HashInit` method, the hash result is set to the `Accumulator` column of the invoking hash object.

If there is no open hash "stream" for the invoking hash object, the method invocation SHALL fail.

```
HashObjectUID.HashFinalize [ ]
=>
[ Result : bytes ]
```

5.6.4.13.1 HashFinalize Result

5.6.4.13.1.1 Result

The method returns the input data not consumed by the hash prior to this method invocation. If the BufferOut parameter is specified in the `HashInit` method, the `Hash` method Result SHALL be empty.

5.6.4.13.2 Fails

a. If the object does not exist
b. If the object does not reference a valid symmetric credential object that contains a valid key
c. If BufferOut is not a valid cellblock

d. If BufferOut is specified and Set access control on that cellblock has not been fulfilled
e. If HMACFinalize has been invoked when no HMAC "stream" is open

5.6.4.14 HMAC Method Group – HMACInit (Object Method)

This method is used to initiate an HMAC "stream" using the hash object that invoked the method. Only one HMAC "stream" SHALL be able to be open at any one time for any individual hash object.

Invocation of this method is required before the HMAC method MAY be successfully invoked. In preparation for beginning the HMAC "stream", upon successful invocation of the HMACInit method, the invoking hash object's Accumulator column is set to zero.

```
HashObjectUID.HMACInit [
BufferOut = cell_block ]
=>
[   ]
```

5.6.4.14.1 BufferOut

This parameter is used to identify a block of table cells to which the HMACed data is to be set. If the this value is specified, the Input byte length SHALL be equal in size to or smaller than the cellblock specified for the Result.

If the host invokes this method using the BufferOut cellblock as the target for the result, then in addition to fulfilling the access control on the HMAC method, the host SHALL also fulfill the access control required to invoke the Set method on the entirety of that cellblock.

5.6.4.14.2 HMACInit Result

5.6.4.14.2.1 Result

The HMACInit method returns an empty list. Success or failure of the method invocation is determinable based on the status code returned in response to the method invocation.

5.6.4.14.3 Fails

a. If the hash object does not exist
b. If the hash object does not reference a valid symmetric credential object that contains a valid key
c. If the hash object currently already has an HMAC "stream" open

5.6.4.15 HMAC Method Group – HMAC (Object Method)

Invocation of the HMAC method causes the data input to be hashed using the HMAC algorithm with the symmetric key credential referenced from the invoking hash object.

This method SHALL require that the HMACInit method has been invoked previously during the session to start an HMAC "stream" for the invoking hash object, and that the HMACFinalize method has not yet been invoked to close that "stream". Invoking the HMAC method when there is not an open HMAC "stream" (i.e. before HMACInit or after HMACFinalize) SHALL result in an error.

```
HashObjectUID.HMAC [
Input : typeOr { Data : bytes, Buffer : cell_block } ]
=>
[ Result : bytes ]
```

5.6.4.15.1 Input

The value of the Input parameter MAY be either a bytes value, wherein bytes to have the HMAC operation performed are passed as a parameter of the method; or a cellblock that addresses a subset of a table on the SP that holds the data to have the HMAC operation performed.

5.6.4.15.1.1 Data

This input parameter is used to supply bytes directly from the host for the HMAC operation.

5.6.4.15.1.2 Buffer

This input parameter is used to identify a block of table cells where the data upon which the operation is to occur is stored. If the host invokes the `HMAC` method using the data addressed via the cellblock as data input, then in addition to fulfilling the access control on the `HMAC` method, the host SHALL also fulfill the access control required to invoke the `Get` method on the entirety of that cellblock.

5.6.4.15.2 HMAC Result

5.6.4.15.2.1 Result

The method returns the DataInput that has been consumed in the HMAC operation as the method result.

Results of the operation are stored in the BufferOut cellblock of the `HMACInit` method, if that parameter is included. Otherwise the results are stored in the invoking hash object's `Accumulator` column. If the BufferOut parameter is specified in the `HMACInit` method, the `HMAC` method Result SHALL be empty.

5.6.4.15.3 Fails

a. If the object does not exist
b. If the object does not reference a valid symmetric credential object that contains a valid key
c. If the DataInput cellblock reference is not a to valid cellblock
d. If the DataInput is a cellblock reference and Get access control on that cellblock has not been fulfilled
e. If `HMAC` has been invoked when no HMAC "stream" is open

5.6.4.16 HMAC Method Group – HMACFinalize (Object Method)

Invocation of the `HMACFinalize` method causes the TPer to flush the remaining, non-blocked data through the hash, computes the HMAC, and sets the result to the **BufferOut** cellblock. If the BufferOut cellblock has not been supplied, the HMAC result is set to the `Accumulator` column of the invoking Hash object.

If there is no open HMAC "stream" for the invoking Hash object, the method invocation SHALL fail.

```
HashObjectUID.HMACFinalize [  ]
=>
[ Result : bytes ]
```

5.6.4.16.1 HMACFinalize Result

5.6.4.16.1.1 Result

The method returns the input data not consumed by the HMAC prior to this method invocation. If the BufferOut parameter is specified in the `HashInit` method, the `HMAC` method Result SHALL be empty.

5.6.4.16.2 Fails

a. If the object does not exist
b. If the object does not reference a valid symmetric credential object that contains a valid key
c. If BufferOut is not a valid cellblock

d. If BufferOut is specified and Set access control on that cellblock has not been fulfilled
e. If the BufferOut cellblock is specified and it is not larger than or equal to the size of the HMAC result
f. If HMACFinalize has been invoked when no HMAC "stream" is open

5.6.4.17 XOR (SP Method)

This method invocation causes the input data is XORed using the pattern specified in the PatternInput parameter.

```
ThisSP.XOR[
PatternInput : uidref {ByteTable},
DeletePattern : boolean,
Input : typeOr { Data : bytes, BufferIn : cell_block },
BufferOut = cell_block ]
=>
[ Result : bytes ]
```

5.6.4.17.1 PatternInput

This parameter is a reference to the byte table that stores the pattern. The host SHALL be required to fulfill access control requirements necessary to invoke the Get method on the PatternInput byte table, or the XOR method invocation SHALL fail.

5.6.4.17.2 DeletePattern

This parameter identifies whether the table identified in the PatternInput parameter is zeroed. If the DeletePattern parameter is True, the cell values in the referenced table SHALL be set to 00s after the pattern is applied to the XOR operation. If the parameter value is False, no change is made to the value stored in the cells of the PatternInput byte table.

5.6.4.17.3 Input

That Input parameter MAY be either a bytes value, wherein bytes to be XORed are passed as a parameter of the method; or a cellblock that addresses a subset of a table on the SP that holds the data to be XORed.

5.6.4.17.3.1 Data

This input parameter is used to supply bytes directly from the host for the XOR operation.

5.6.4.17.3.2 BufferIn

This input parameter is used to identify a block of table cells where the data upon which the operation is to occur is stored. If the host invokes the XOR method using the data addressed via the cellblock as data input, then in addition to fulfilling the access control on the XOR method, the host SHALL also fulfill the access control required to invoke the Get method on the entirety of that cellblock.

5.6.4.17.4 BufferOut

This parameter is used to identify a block of table cells to which the XORed data is to be set. If this value is specified, the Input byte length SHALL be equal in size to or smaller than the cellblock specified to receive the result of the operation.

If the host invokes this method using the BufferOut cellblock as the target for the result, then in addition to fulfilling the access control on the XOR method, the host SHALL also fulfill the access control required to invoke the Set method on the entirety of that cellblock.

5.6.4.17.5 XOR Result

5.6.4.17.5.1 Result

The method returns the result of the XOR operation.

If the BufferOut parameter is specified, the result of the operation is stored in that location. Otherwise the result of the operation is returned in the method result.

If the BufferOut parameter is specified, the Result value SHALL be empty.

5.6.4.17.6 Fails

a. If the PatternInput is not a byte table
b. If the Input cellblock reference is not to a valid cellblock
c. If BufferOut is not a valid cellblock
d. If PatternInput is smaller than the input data
e. If BufferOut is smaller than the input data
f. If associated access control conditions, as described in 5.6.5.4 are not met

5.6.5 Descriptions

5.6.5.1 Cellblocks

Methods of the Crypto Template utilize cellblocks for parameters. Cellblocks are the abstract type cell_block, and define a set of rows and columns that make up a contiguous area of a table. Use of cellblocks as method parameters necessitate special access control condition requirements.

Any cellblock used as input data to a method requires that the host invoking the method satisfy the access control requirement necessary to invoke the `Get` method on the entirety of the parameterized cellblock.

Any cellblock used as an output buffer for a method requires that the host invoking the method satisfy the access control requirement necessary to invoke the `Set` method on the entirety of the parameterized cellblock.

Exceptions or additions to this, such as are required for the `XOR` method's PatternInput parameter, are noted in the method's description.

5.6.5.2 Hashing

Invocation of the `HashInit` method, followed by one or more `Hash` method invocations and the `HashFinalize` method on a `H_SHA_*` object, causes the data parameterized in or referenced from the `Hash` method invocations to be hashed in the manner described in [10].

A hash "stream" is initiated using the `HashInit` method invoked upon a hash object. Only one hash "stream" SHALL be open at any one time for any individual hash object. During a session, invoking the `HashInit` method on a hash object after invoking `HashInit` on that object but before invoking the `HashFinalize` method SHALL cause the second `HashInit` method invocation to fail.

The `HashInit` method SHALL be invoked prior to invocation of the `Hash` method. In preparation for beginning the hash "stream", upon successful invocation of the `HashInit` method, the invoking hash object's `Accumulator` column is set to zeroes. After invocation of the `HashInit` method, the `Set` method MAY be used to set an initial condition in the `Accumulator` column. Invoking the `Set` method on the `Accumulator` column after the `HashInit` invocation and after one or more successful `Hash` invocations MAY cause the final hash result to be an unexpected or inconsistent value.

The `HashInit` method has a parameter that allows the host to specify a particular cellblock as the target of the hash's final result. This cellblock SHALL be set to the final hash result upon successful invocation of the `HashFinalize` method. Access control requirements necessary to permit invocation of the `Set` method on the entirety of this cellblock SHALL be fulfilled, or the `HashInit` method invocation SHALL fail. If the BufferOut parameter of the `HashInit` method was used, the cellblock SHALL be larger than or equal to the size of the expected final hash calculation result, or the `HashInit` method invocation SHALL fail.

Successful invocation of the `Hash` method causes the data input to the `Hash` method to be hashed with the value currently stored in the `Accumulator` column of the invoking hash object. The `Hash` method

returns as its result the input data that has been consumed in the hash. Hashing is done at block boundaries appropriate for the hash object type.

The Hash method SHALL accept either bytes passed across the interface as a parameter of the method invocation; or SHALL identify a cellblock that holds the data to be hashed. If the data is addressed via cellblock, the host SHALL fulfill access control requirements necessary to invoke the Get method on the entirety of that cellblock, or the Hash method invocation SHALL fail.

Upon invocation of the HashFinalize method, the TPer flushes the remaining, non-blocked data through the hash function represented by the invoking hash object. Upon completion of hashing, the final hash result is set to the cellblock specified in the HashInit method. If this parameter was not included in the HashInit method, then the Accumulator column of the invoking hash object is set to the final hash result. The HashFinalize method returns any data that had previously been input but had not yet been hashed and returned in the Hash method. The HashFinalize method closes the hash "stream" on the invoking hash object.

If the BufferOut parameter of the HashInit method was used, the referenced cellblock SHALL be set to the final hash value. Access control requirements necessary to permit invocation of the Set method on the entirety of this cellblock SHALL be fulfilled, or the HashFinalize method SHALL fail. If the BufferOut parameter of the HashInit method was used, the cellblock SHALL be larger than or equal to the size of the expected final hash calculation result, or the HashFinalize method invocation SHALL fail.

Invoking Hash or HashFinalize on a hash object that does not have an open "stream" SHALL cause that method invocation to fail.

5.6.5.3 HMAC

Invocation of the HMACInit method, followed be one or more HMAC method invocations and the HMACFinalize method on a H_SHA_* object, causes the data parameterized in or referenced from the HMAC method invocation to have a message authentication code computed on that input data using the H_SHA_* object upon which the method was invoked, the HMAC key referenced from that H_SHA_* object, and the HMAC algorithm described in [13].

An HMAC "stream" is initiated using the HMACInit method invoked upon a hash object. Only one HMAC "stream" SHALL be open at any one time for any individual hash object.

During a session, invoking the HMACInit method on a hash object after invoking HMACInit on that object but before invoking the HMACFinalize method SHALL cause the second HMACInit method invocation to fail. The HMACInit method SHALL be invoked prior to invocation of the HMAC method.

Successful invocation of the HMAC method causes the data input to be hashed on block boundaries as they are reached. Intermediate results are stored as internal state and are not accessible from the host.

The HMAC method SHALL accept either bytes passed across the interface as a parameter of the method invocation; or SHALL identify a cellblock that holds the data to be hashed. If the data is addressed via cellblock, the host SHALL fulfill access control requirements necessary to invoke the Get method on the entirety of that cellblock, or the HMAC method invocation SHALL fail.

Successful invocation of the HMAC method returns the input data that was consumed in the hash method.

Upon invocation of the HMACFinalize method, the TPer flushes the remaining, non-blocked data through the hash function represented by the invoking hash object, computes the HMAC, and sets the result to the BufferOut cellblock if specified. If the BufferOut cellblock is specified, the host SHALL be required to fulfill access control requirements necessary to successfully invoke the Set method on the entirety of that cellblock.

If the BufferOut cellblock has been specified in the HMACFinalize method invocation, that cellblock SHALL be larger than or equal to the size of the HMAC calculation result or the HMACFinalize method invocation SHALL fail.

If the BufferOut cellblock has not been supplied, the HMAC result is set to the Accumulator column of the invoking hash object.

The HMACFinalize method invocation SHALL return the data that had been supplied as input to the HMAC method that had not yet been consumed.

Invoking HMAC or HMACFinalize on a hash object that does not have an open "stream" SHALL cause that method invocation to fail.

5.6.5.4 XOR

Invocation of the XOR method causes the XOR method's input data to be XORed with the pattern specified in the PatternInput parameter of the method invocation.

The XOR method SHALL accept either bytes passed across the interface as a parameter of the method invocation; or SHALL identify a cellblock that holds the data to be XORed. If the data is addressed via cellblock, the host SHALL fulfill access control requirements necessary to invoke the Get method on the entirety of that cellblock, or the XOR method invocation SHALL fail.

The PatternInput parameter of the method SHALL be the uid of a byte table that holds the pattern with which the input data SHALL be XORed. The PatternInput SHALL be the same size or larger than the input data or the method SHALL fail. The host SHALL be required to fulfill access control requirements necessary to invoke the Get method on the PatternInput byte table, or the XOR method invocation SHALL fail.

The DeletePattern parameter identifies the behavior of the PatternInput table after the XOR operation is complete. If the DeletePattern parameter is True, then at completion of the XOR operation the contents of the byte table referenced as the PatternInput SHALL be set to all 00's. If the DeletePattern parameter is False, the method SHALL NOT alter the contents of the referenced byte table. If DeletePattern is True, , the host SHALL fulfill access control requirements that permit invocation of the Set method on the PatternInput table or the XOR method invocation SHALL fail.

If the host's intention is to use the XOR method as a one-time pad, the host SHOULD invoke the XOR method with a DeletePattern value of True.

The XOR method returns data in one of two ways. If the BufferOut parameter is specified in the method invocation, then the XOR method result is set to that cellblock. The BufferOut parameter is specified, the cellblock SHALL be the same size or larger than the XOR result. The host SHALL be required to fulfill access control requirements necessary to invoke the Set method on the entirety of that cellblock, or the XOR method invocation SHALL fail. If the BufferOut parameter is specified, the method result SHALL be empty.

If the BufferOut parameter is not specified, the method result SHALL be the result of the XOR operation.

5.6.5.5 Signing

Signing of a selected input is accomplished as described in the following subsections. The exact algorithms used in the signing and verification of digital signatures are defined in [11], and are dependent on the public scheme (RSA, EC, etc.) used.

5.6.5.5.1 Invocation of Sign on a Public Key Credential

Invocation of the Sign method is be done by invoking the method on a key pair credential object that has a private key (for example, C_RSA_1024 or C_EC_256 objects) and either passing in data or referencing data to be used as the input for signing. The TPer utilizes the private key stored in the credential object referenced in the invocation and performs a private key signing operation on the input data.

For this usage, if a cellblock is referenced to hold the output of the Sign method invocation, then no data is returned, and the result of the Sign is stored in the referenced cell_block. The host SHALL fulfill access control requirements necessary to invoke the Set method on the entirety of that target cellblock.

If a cellblock is not referenced to hold the Sign method output, the data returned is the result of the signing operation performed on the input data with the private key of the referenced credential object.

5.6.5.5.2 Invocation of Sign on a Hash Object

A second way to accomplish signing is to invoke the Sign method on a H_SHA_* object. If the invocation is done in this manner, then the H_SHA_* object upon which the method was invoked SHALL reference a key pair credential object that has a private key. The signing operation in this case is done using the private key of that referenced credential object.

When invocation of the Sign method is done on a H_SHA_* object, the signing operation MAY be performed on either:

a. Data parameterized in or referenced from the Sign method invocation. If this is the case, the signed data SHALL either be stored in a cell_block referenced in the invocation or returned as the result of the method invocation. If a cellblock is referenced as the target of the signed data, the host SHALL be required to fulfill access control requirements necessary to invoke the Set method on the entirety of that cellblock, or the Sign method invocation SHALL fail.

b. Or, if no input data or reference is included in the method invocation, the signing operation is performed on the value of the H_SHA_* object's Accumulator column. If this is the case, the signed data SHALL be stored to the Proof column of the H_SHA_* object, and MAY be retrieved with a successful invocation of the Get method on that column.

5.6.5.6 Verifying

Verification of a signed hash is accomplished as described in the following subsections. The exact algorithms used in the signing and verification of digital signatures are defined in [11], and are dependent on the public scheme (RSA, EC, etc.) used.

5.6.5.6.1 Invocation of Verify on a Public Key Credential

Verification MAY be performed by invoking the Verify method on a public key credential.

The Verify method is invoked on a public key credential. The proof to be verified against MAY be supplied in one of two ways.

a. The proof MAY be parameterized in bytes as the Proof parameter of the Verify method invocation.

b. The proof MAY be stored in a cellblock, which is addressed by the Proof parameter of the Verify method.

The value to be verified MAY be supplied by the DataInput parameter in one of two ways:

a. The value for verification MAY be supplied in bytes as the DataInput parameter of the Verify method invocation.

b. The value for verification MAY be stored in a cellblock, which is addressed by the DataInput parameter of the Verify method.

If either the DataInput or Proof parameters are supplied and address a cellblock, the host SHALL be required to fulfill the access control requirements necessary to invoke the Get method on the entirety of that cellblock or the Verify method invocation SHALL fail.

Verification of the input against the proof is performed using the public key of the invoking public key credential.

Invocation of the Verify method SHALL return True if the verified value matches the proof. Otherwise, the method invocation SHALL return False.

5.6.5.6.2 Invocation of Verify on a Hash Object

To perform signed hash verification in this way, the Verify method is invoked on a hash object.

The proof to be verified against MAY be supplied in one of three ways.

a. The proof MAY be parameterized in bytes as the Proof parameter of the Verify method invocation.

b. The proof MAY be stored in a cellblock, which is addressed by the Proof parameter of the Verify method.

c. If Proof parameter is not supplied to the Verify method, the proof used SHALL be the value of the invoking hash object's Proof column.

The value to be verified MAY also be supplied in one of three ways.

a. The value for verification MAY be supplied in bytes as the DataInput parameter of the Verify method invocation.

b. The value for verification MAY be stored in a cellblock, which is addressed by the DataInput parameter of the Verify method.

c. If the DataInput parameter is not supplied to the Verify method, the proof used SHALL be the value of the invoking hash object's Accumulator column.

If the Proof parameter is supplied and is a cellblock, the host SHALL be required to fulfill the access control requirements necessary to invoke the Get method on the entirety of that cellblock or the Verify method invocation SHALL fail.

Verification of the input against the proof is performed using the public key of the public key credential that SHALL be referenced from the invoking hash object.

Invocation of the Verify method SHALL return True if the verified value matches the proof. Otherwise, the method invocation SHALL return False.

5.6.5.7 Encrypting

Invocation of the EncryptInit method, followed by one or more Encrypt method invocations and the EncryptFinalize method, encrypts data that has either been sent to the Crypto template-enabled SP from the host in the method invocation, or that is currently stored in the SP.

Successful invocation of the EncryptInit method is used to initiate an encryption "stream" using the credential object that invoked the method. Only one encryption "stream" SHALL be open in a session at any one time for a particular credential object. During a session, invoking the EncryptInit method on a credential object after invoking EncryptInit on that object but before invoking the EncryptFinalize method SHALL cause the second EncryptInit method invocation to fail.

If the optional IV parameter is used in the EncryptInit method, the parameterized IV is used in place of that which is stored in the ResidualData column of the invoking credential. Otherwise, the value of the ResidualData column of the invoking credential is used as the IV, as required by the value of the credential object's Mode column.

As indicated, the host MAY generate an initialization vector externally and either pass it as a parameter to the EncryptInit method, or Set the ResidualData column of the symmetric credential object that is referenced for use with the encryption. Alternatively, the host MAY invoke the Random method and set its output to the ResidualData column of the symmetric credential object that is referenced for use with the encryption. For details and guidelines on generation and handling of initialization vectors, see [14].

If the EncryptInit method is invoked with an IV and the credential object or credential object's Mode column value do permit use of an IV, then the EncryptInit method SHALL fail.

The EncryptInit method, upon successful invocation, sets the ResidualData column of the invoking credential to zero.

Successful invocation of the Encrypt method causes the TPer to use the key stored in the invoked credential object to encrypt the input data.

The input data MAY either be parameterized in the Encrypt method invocation, or stored in a table that is referenced as a cellblock from the Encrypt method invocation. If the Encrypt method's DataInput parameter references a cellblock, the host SHALL fulfill access control requirements necessary to invoke the Get method on the entirety of that cellblock or the method invocation SHALL fail.

If a cellblock is referenced to hold the output of the Encrypt method invocation, then no data is returned, and the result of the Encrypt is stored in the referenced cellblock. If the output cellblock is used, the host SHALL be required to fulfill the access control requirements necessary to permit invocation of the Set method on the entirety of the referenced cellblock. If the cellblock is not specified, the data returned is the result of the encryption operation performed on the input data.

The length of the data input to the Encrypt method SHALL be as required by the block size of the particular key type and mode used. Should padding be required, the host SHALL perform it.

Successful invocation of the Encrypt method causes the invoking symmetric credential object's ResidualData column to have the value specified in Table 185.

Upon invocation of the EncryptFinalize method, the encryption "stream" for the invoking credential SHALL be closed. Invoking Encrypt or EncryptFinalize on a hash object that does not have an open "stream" SHALL cause that method invocation to fail.

Note that only one write session is open at any given point in time. After closing a write session and opening another write session to the same SP, the host MAY find that the ResidualData value MAY have been modified by another write session.

5.6.5.8 Decrypting

Invocation of the DecryptInit method, followed by one or more Decrypt method invocations and the DecryptFinalize method invocation decrypts data that has either been sent to the Crypto template-enabled SP from the host in the method invocation, or that is currently stored in the SP.

Successful invocation of the DecryptInit method is used to initiate a decryption "stream" using the credential object that invoked the method. Only one decryption "stream" SHALL be open in a session at any one time for a particular credential object. During a session, invoking the DecryptInit method on a credential object after invoking DecryptInit on that object but before invoking the DecryptFinalize method SHALL cause the second DecryptInit method invocation to fail.

If the optional IV parameter is used in the DecryptInit method, the parameterized IV is used in place of that which is stored in the ResidualData column of the invoking credential. Otherwise, the value of the ResidualData column of the invoking credential is used as the IV, as required by the value of the credential object's Mode column.

If the DecryptInit method is invoked with an IV and the credential object or credential object's Mode column value do permit use of an IV, then the DecryptInit method SHALL fail.

The DecryptInit method, upon successful invocation, sets the ResidualData column of the invoking credential to zero.

Successful invocation of the Decrypt method causes the TPer to use the key stored in the invoked credential object to decrypt the input data. Decryption is performed using the key stored in invoking the credential object.

The input data MAY either be parameterized in the Decrypt method invocation, or stored in a table that is referenced as a cell_block from the Decrypt method invocation. If the Decrypt method's DataInput parameter references a cellblock as the data input, the host SHALL fulfill access control requirements

necessary to invoke the Get method on the entirety of that cellblock or the method invocation SHALL fail.

If a cellblock is referenced to hold the output of the Decrypt method invocation, then no data is returned, and the result of the Decrypt is stored in the referenced cellblock. If the output cellblock is used, the host SHALL be required to fulfill the access control requirements necessary to permit invocation of the Set method on the entirety of the referenced cellblock. If the cellblock is not specified, the data returned is the result of the decryption operation performed on the input data.

The length of the data input to the Decrypt method SHALL be as required by the block size of the particular key type and mode used. Should padding be required, the host SHALL perform it.

Successful invocation of the Decrypt method causes the invoking symmetric credential object's ResidualData column to have the value specified in Table 185.

Upon invocation of the DecryptFinalize method, the decryption "stream" for the invoking credential SHALL be closed. Invoking Decrypt or DecryptFinalize on a hash object that does not have an open "stream" SHALL cause that method invocation to fail.

Note that only one write session is open at any given point in time. After closing a write session and opening another write session to the same SP, the host MAY find that the ResidualData value MAY have been modified by another write session.

5.6.5.9 Default Logging Settings

The default logging settings associated with the Crypto Template methods are:

a. The default logging setting for the Delete object method on objects in the H_SHA_* tables, and for invocation of the Verify method, is LogAlways.

b. The default setting for all instances of the Crypto Template methods (Sign, HashInit, Hash, HashFinalize, HMACInit, HMAC, HMACFinalize, XOR, EncryptInit, Encrypt, EncryptFinalize, DecryptInit, Decrypt, and DecryptFinalize) is LogSuccess.

c. All other methods that apply to the H_SHA_* tables SHALL be as described in the Base Template reference section on Default Logging Settings (See 5.3.4.4).

5.6.6 Life Cycle

5.6.6.1 Crypto Template-Specific Life Cycle State Descriptions/Exceptions

An SP issued with the Crypto Template has the following characteristics based on the current life cycle state of that SP:

a. **Disabled –** A Crypto Template-enabled SP that is in the Disabled state SHALL NOT be able to perform any user-invoked SP operations, with the exceptions noted in section 4.5.2.

b. **Frozen –** Attempts to open sessions to an SP in the Issued-Frozen state SHALL fail.

c. **Issued-Disabled-Frozen** – Attempts to open sessions to an SP in the Issued-Disabled-Frozen state SHALL fail.

5.7 Locking Template

5.7.1 Overview

Begin Informative Content

The Locking Template defines mechanisms for access control to user data, including controlling media encryption, user data encryption key management, and Read/Write lock state.

Side effects occur when writing cells of the tables of the SP that incorporates the Locking Template. These side effects include enabling Read/Write locking, enabling encryption with a certain encryption key, and initiating the re-encryption process.

End Informative Content

The Locking Template also enables an SP to manage re-encryption of data. For the purpose of this specification, re-encryption is defined as the process by which user data is transformed from 1) encrypted data using the active encryption key to encrypted data with a new encryption key (re-encryption), 2) clear text data to encrypted data with a new key (encryption), or 3) encrypted data to clear text data (decryption).

Re-encryption has the following basic attributes:

1. This process operates as a background TPer operation. Re-encryption MAY operate concurrently with normal User Data Interface Commands.
2. Re-encryption processes are linked to a specific LBA range. Multiple concurrent re-encryption operations are permitted up to the available TPer re-encryption resources.

5.7.1.1 Terminology

Table 224 Locking Template Terminology

Term	Definition
Global range	The entire User-Addressable LBA Range
Key Changing	The changing of a Credential reference
KeysAvailable	Condition: Host has provided enough information to enable access to Locking LBA range keys. See the KeysAvailableCfg column in the LockingInfo Table for more information.
LBA Range	A sub-section of the User-Addressable LBA Range
MBR Shadowing	This allows loading of preboot code that MAY be necessary to unlock an LBA range that starts at LBA 0 for reading and writing
Media Encryption	Inline encryption of data to media
Re-encryption	Encryption of the original cleartext media data, which MAY have been previously encrypted, to the media with a different key.
TPer_Error_Detect	Condition: A TPer re-encryption error has been detected.
TPer_Job_Done	Condition: TPer has completed re-encryption without errors
TPer_Key_Error	Not all keys are valid
TPer_Ready	Condition: ALL the following resources & conditions required to begin or continue re-encryption are true: - All TPer resources are available, such as buffer space, re-encryption H/W & S/W resources. - Re-encryption keys are valid - TPer_Error_Detect condition is False - TPer_Reset_Stop condition is False - PAUSE_req is False - KeysAvailable is True
TPer_Reset_Detect	Condition: A reset condition has been detected.
TPer_Reset_Stop	Condition: A reset condition is detected that does not permit the Re-encryption process to continue.
User-Addressable LBA Range	The user-accessible storage space on a Storage Device, where user data is stored

5.7.2 Data Structures

Begin Informative Content

The Locking Template contains the following tables:

a. **LockingInfo**: Information about the TPer's configuration

b. **Locking**: The storage encryption and read/write locking controls covering different contiguous ranges of storage space on the TPer.

c. **MBRControl** and **MBR**: For MBR shadowing, needed to handle pre-boot authentication environments.

d. **K_AES_128 and K_AES_256:** Tables used to store media encryption keys

End Informative Content

5.7.2.1 LockingInfo (Object Table)

The `LockingInfo` table is an object table that SHALL contain exactly one row, which contains information about the TPer's configuration.

Table 225 LockingInfo Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name		name
0x02	Version		uinteger_4
0x03	EncryptSupport		enc_supported
0x04	MaxRanges		uinteger_4
0x05	MaxReEncryptions		uinteger_4
0x06	KeysAvailableCfg		keys_avail_conds

5.7.2.1.1 UID

This is the unique identifier of this row of the `LockingInfo` table.

This column SHALL NOT be modifiable by the host.

5.7.2.1.2 Name

This is the manufacturer-defined name for this feature.

This column SHALL NOT be modifiable by the host.

5.7.2.1.3 Version

This is a manufacturer-defined version number for this feature.

This column SHALL NOT be modifiable by the host.

5.7.2.1.4 EncryptSupport

This defines the types of encryption supported by this template. The value of this column is "None" if the drive does not support media encryption.

This column SHALL NOT be modifiable by the host.

5.7.2.1.5 MaxRanges

This value defines the maximum number of supportable LBA ranges in addition to the Global Range. If this value is 0, then the only range available is the entire Global Range of the Storage Device.

This column SHALL NOT be modifiable by the host.

5.7.2.1.6 MaxReEncryptions

This value defines the maximum number of simultaneous re-encryption operations supported. Simultaneous re-encryptions refer to the number of different LBA ranges that MAY be concurrently re-encrypted. A value of 0 indicates Re-encryption is not supported.

This column SHALL NOT be modifiable by the host.

5.7.2.1.7 KeysAvailableCfg

This column defines which conditions are required for re-encryption to proceed.

This column SHALL NOT be modifiable by the host.

5.7.2.2 Locking (Object Table)

`Locking` table rows define encryption, re-encryption, read locking, and write locking configuratoin for the Storage Device's LBA ranges. An LBA range is defined as an ordered sequence of `RangeLength` logical blocks (as appropriate to the device, typically LBAs), numbered consecutively starting at LBA `RangeStart`.

The `Locking` table SHALL always have at least one row. This required row of the `Locking` table SHALL represent the entire User-Addressable LBA Range, called the Global Range. This row SHALL have a `UID` column value of `0x00 0x00 0x08 0x02 0x00 0x00 0x00 0x01` and a `Name` column value of "Global Range". This row SHALL NOT be deletable.

Table 226 Locking Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name		name
0x02	CommonName		name
0x03	RangeStart		uinteger_8
0x04	RangeLength		uinteger_8
0x05	ReadLockEnabled		boolean
0x06	WriteLockEnabled		boolean
0x07	ReadLocked		boolean
0x08	WriteLocked		boolean
0x09	LockOnReset		reset_types
0x0A	ActiveKey		mediakey_object_uidref
0x0B	NextKey		mediakey_object_uidref
0x0C	ReEncryptState		reencrypt_state
0x0D	ReEncryptRequest		reencrypt_request
0x0E	AdvKeyMode		adv_key_mode
0x0F	VerifyMode		verify_mode
0x10	ContOnReset		reset_types
0x11	LastReEncryptLBA		uinteger_8
0x12	LastReEncStat		last_reenc_stat
0x13	GeneralStatus		gen_status

5.7.2.2.1 UID

This is the unique identifier for this `Locking` object.

5.7.2.2.2 Name

This is the name of the `Locking` object.

5.7.2.2.3 CommonName

This is a name that MAY be shared by multiple `Locking` objects.

5.7.2.2.4 RangeStart

This column value defines the starting LBA value for this range. In non-Global Range rows, this column MAY be modifiable based on access control settings. Changes to this column are subject to the same constraints and checks defined for this column when rows of the `Locking` table are created (see 5.7.3.3).

For the Locking object that represents the Global Range, this column value SHALL be zeroes.

5.7.2.2.5 RangeLength

This column value defines the quantity of contiguous LBAs for this LBA range (starting with the value defined in the `RangeStart` column). In non-Global Range rows, this column MAY be modifiable based on access control settings. Changes to this column are subject to the same constraints and checks defined for this column when rows of the `Locking` table are created (see 5.7.3.3).

For the Locking object that represents the Global Range, this column value SHALL be zeroes.

5.7.2.2.6 ReadLockEnabled

The value of this column determines whether or not the read-locking feature is enabled for this LBA range and indicates whether or not the `ReadLocked` column value is meaningful for this range. If the value of the `ReadLockEnabled` column is False, the read-locking feature is disabled, and the value of the `ReadLocked` column is disregarded. If the value of the `ReadLockEnabled` column is True, the read-locking feature is enabled, and the value of the `ReadLocked` column identifies the current read locking state.

5.7.2.2.7 WriteLockEnabled

The value of this column determines whether or not the write-locking feature is enabled for this LBA range and indicates whether or not the `WriteLocked` column value is meaningful for this range. If the value of the `WriteLockEnabled` column is False, the write-locking feature is disabled, and the value of the `WriteLocked` column is ignored. If the value of the `WriteLockEnabled` column is True, the write-locking feature is enabled, and the value of the `WriteLocked` column identifies the current write locking state.

5.7.2.2.8 ReadLocked

The value of this column identifies the current read lock state for the associated LBA Range if the range's `ReadLockEnabled` column is True. This column is ignored if the range's `ReadLockEnabled` column is False. If this value is True, then the TPer SHALL NOT allow requests to read user data. If this value is False, then the TPer SHALL allow requests to read user data (see 5.7.3.2).

The `Set` method MAY be invoked by the host to change the value of this column and alter the read-lock state. Setting the column value to True read locks the range. Setting the column value to False read unlocks the range.

5.7.2.2.9 WriteLocked

The value of this column identifies the current write lock state for the associated LBA Range if the range's `WriteLockEnabled` column is True. This column is ignored if the range's `WriteLockEnabled` column is False. If this value is True, then the TPer SHALL NOT allow requests to write user data. If this value is False, then the TPer SHALL allow requests to write user data (see 5.7.3.2).

The Set method MAY be invoked by the host to change the value of this column and alter the write lock state. Setting the column value to True write locks the range. Setting the column value to False write unlocks the range.

5.7.2.2.10 LockOnReset

This value defines the locking behavior of this LBA range at reset, dependent on reset type. The values enumerated in this column identify the reset types that cause the values of the ReadLocked and WriteLocked columns of the Locking table to be set to True.

The Global Range's LockOnReset value defines global TPer behavior. All other rows override the Global Range's behavior, unless otherwise specified in an SSC.

An empty set indicates that ReadLocked and WriteLocked do not change on any reset.

5.7.2.2.11 ActiveKey

This column points to this LBA range's media encryption key. If the value of this column is a NULL UID then data in this range is stored in plaintext.

5.7.2.2.12 NextKey

This column identifies the LBA range's next media encryption key. This value and the referenced media encryption key object SHALL be writable when the value of the ReEncryptState column is IDLE only. Otherwise, attempts to invoke any of the Set, Delete, DeleteRow, or GenKey methods on the associated credential object SHALL return an error.

User Data SHALL be returned to clear text when the value stored in NextKey is a NULL UID and re-encryption has been requested.

5.7.2.2.13 ReEncryptState

The value of this column identifies the currently applicable Re-encryption state (see 5.7.3.3). The value in the column affects the TPer's response to the host's requests in the ReencryptRequest column.

Reset configuration (ContOnReset) and a detected reset condition define the reported ReEncryptState and PauseStatus values.

This column SHALL NOT be modifiable by the host.

5.7.2.2.14 ReEncryptRequest

A host application requests a re-encryption operation by invoking the Set method on this column. Successful invocation of the Get method on this column SHALL return no value.

Only state transitions described in 5.7.3.7.1 SHALL be valid.

5.7.2.2.15 AdvKeyMode

This value defines when the value of the NextKey column moves to the ActiveKey column and whether ReEncryptState transitions to COMPLETED or IDLE when the re-encryption process completes.

5.7.2.2.16 VerifyMode

This column value defines the verification requirement during re-encryption. When True, a Read Verify SHALL be performed on the re-encrypted LBA before the LBA is considered good.

5.7.2.2.17 ContOnReset

This column value is a set of reset conditions. This value defines how a re-encryption process reacts to reset conditions.
An empty set means the TPer_Reset_Stop condition is set for any reset condition. The ReEncryptState value is set to PAUSED.

For each listed reset entry, the re-encryption process MAY continue after the associated reset is detected.

5.7.2.2.18 LastReEncryptLBA

This column value defines the last good re-encrypted LBA for this region. This field is only valid when the ReEncryptState is ACTIVE, COMPLETED, PENDING, or PAUSED. Typically, when the ReencryptState is ACTIVE, this value is updated periodically. In COMPLETED, PENDING, or PAUSED this value SHALL be valid. When no LBA has been successfully been re-encrypted, the value SHALL be 0xFFFFFFFF_FFFFFFFF.

This column SHALL NOT be modifiable by the host.

5.7.2.2.19 LastReEncStat

This column value defines the last good re-encryption read-modify-write-verify sequence. This column value is only valid when the ReencryptState is COMPLETED, PENDING or PAUSED. When the LastReEncStat value is anything other than SUCCESS, the value of LastReEncryptLBA+1 is the LBA in error or the LBA located at RangeStart if LastReEncryptLBA contains 0xFFFFFFFF_FFFFFFFF. Valid values are defined in table Table 101.

This column SHALL NOT be modifiable by the host.

5.7.2.2.20 GeneralStatus

This field defines why the re-encryption operation arrived at the PAUSED or PENDING state. The value in this column is only valid when in the PAUSED or PENDING state. The values are defined in table Table 87.

This column SHALL NOT be modifiable by the host.

5.7.2.3 Media Encryption Key Table Group - K_AES_128 (Object Table)

This table is used to store media encryption keys, cipher mode of operation, and associated metadata used with the Advanced Encryption Standard (see [12]).

Table 227 K_AES_128 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	Key		key_128
0x04	Mode		symmetric_mode_media

5.7.2.3.1 UID

This is the unique identifier for this object.

This column SHALL NOT be modifiable by the host.

5.7.2.3.2 Name

This is the name assigned to this object.

For objects in this table that exist at issuance, this column SHALL NOT be modifiable by the host.

5.7.2.3.3 CommonName

This is a name that MAY be shared by multiple K_AES_128 objects.

For objects in this table that exist at issuance, this column SHALL NOT be modifiable by the host.

5.7.2.3.4 Key

This column stores the key associated with this K_AES_128 object. Non-tweakable cipher modes such as ECB, CBC, CFB, OFB, GCM, CCM, and CTR SHALL use the 16-byte option for the key size. Tweakable cipher modes such as XTS or LRW SHALL use the 32-byte option

For MediaEncryption mode, the content of the Key column MAY be vendor-specific.

5.7.2.3.5 Mode

This column defines the encryption mode with which this object SHALL be used. Valid modes are defined in 5.1.3.73. MediaEncryption mode permits a vendor-specific encryption mode.

5.7.2.4 Media Encryption Key Table Group - K_AES_256 (Object Table)

This table is used to store media encryption keys, cipher mode of operation, and associated metadata used with the Advanced Encryption Standard (see [12]).

Table 228 K_AES_256 Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	Key		key_256
0x04	Mode		symmetric_mode_media

5.7.2.4.1 UID

This is the unique identifier for this object.

This column SHALL NOT be modifiable by the host.

5.7.2.4.2 Name

This is the name assigned to this object.

For objects in this table that exist at issuance, this column SHALL NOT be modifiable by the host.

5.7.2.4.3 CommonName

This is a name that MAY be shared by multiple K_AES_256 objects.

For objects in this table that exist at issuance, this column SHALL NOT be modifiable by the host.

5.7.2.4.4 Key

This column stores the key associated with this K_AES_256 object. Non-tweakable cipher modes such as ECB, CBC, CFB, OFB, GCM, CCM, and CTR SHALL use the 32-byte option for the key size. Tweakable cipher modes such as XTS or LRW SHALL use the 64-byte option

For MediaEncryption mode, the content of the Key column MAY be vendor-specific.

5.7.2.4.5 Mode

This column defines the encryption mode with which this object SHALL be used. Valid modes are defined in 5.1.3.73. MediaEncryption mode permits a vendor-specific encryption mode.

5.7.2.5 MBRControl (Object Table)

The MBRControl table contains information that controls the use of the MBR table. This table SHALL have only one row, with UID=0x00 0x00 0x08 0x03 0x00 0x00 0x00 0x01.

Table 229 MBRControl Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Enable		boolean
0x02	Done		boolean
0x03	MBRDoneOnReset		reset_types

5.7.2.5.1 UID

This is the unique identifier of this table row.

This column SHALL NOT be modifiable by the host.

5.7.2.5.2 Enable

This column value identifies if this feature is enabled or disabled.

MBR shadowing is active when `Enable` is True and `Done` is False. When MBR shadowing is active, the TPer responds to LBA requests for LBA 0 to the LBA that maps to the end of the `MBR` table with values from the `MBR` table.

5.7.2.5.3 Done

This value indicates whether the TPer has completed processing the contents of the `MBR` table. After the occurrence of a reset event that is listed in `MBRDoneOnReset`, until the host sets the `MBRControl` table's `Done` column to True, LBA requests made by the host, for LBA 0 up to the LBA that maps to the end of the `MBR` table, SHALL only be fulfillable by values from the `MBR` table.

The `Done` column is set to False on every occurrence of a reset event that is listed in `MBRDoneOnReset`.

The state of `Done` SHALL NOT affect the capacity of the Storage Device.

5.7.2.5.4 MBRDoneOnReset

This column value identifies the reset types that set the `Done` column to False.

The `MBRDoneOnReset` column identifies the types of resets that cause `Done` to be automatically set to False. If the set is empty, the device SHALL NOT change the value of the `Done` column on any reset. At issuance, the default value SHALL be Power Cycle.

5.7.2.6 MBR (Byte Table)

See 5.7.3.6.

5.7.3 Description

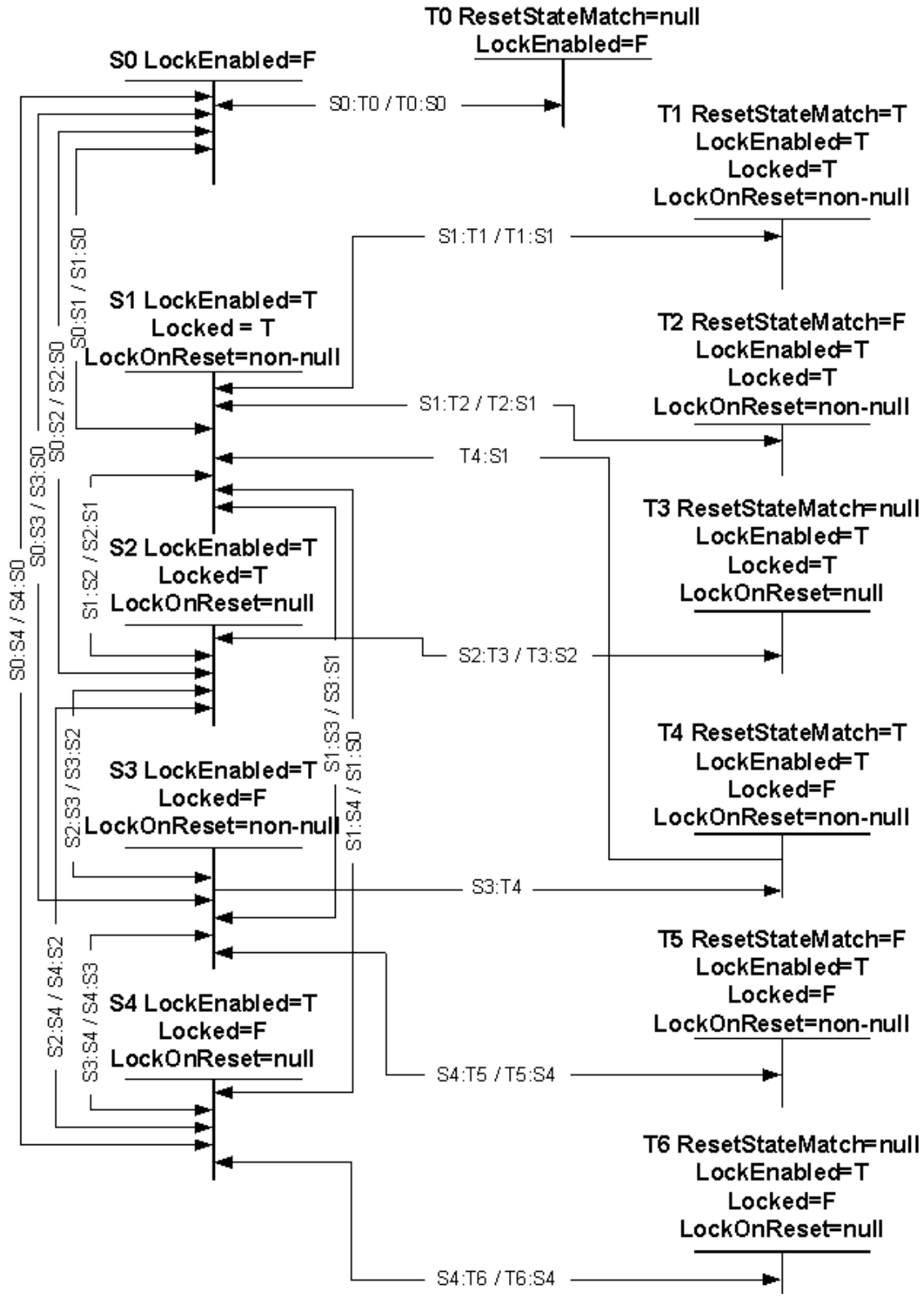
5.7.3.1 Locking State Descriptions

Figure 11 displays the states and state transitions for read lock and write lock. For simplicity, the diagram and the accompanying textual information describe the operation of locking in general, rather than both read lock and/or write lock in particular.

Note that the reset behavior of both read and write locking for each locking object is controlled at the same point, by a single column in the `Locking` table, called `LockOnReset`.

When a reset is described in these state transitions, “reset” is used generically to refer to qualifying resets, as determined by the value of the `LockOnReset` column and the reset behavior associated with particular resets as determined by the appropriate interface-specific description of that reset. Interface-specific reset definitions are defined in [2].

Figure 11 Locking State Diagram



5.7.3.1.1 State Descriptions

This section describes the states that are used in Figure 11 , and the column values that each state represents.

S0 LockEnabled=F

This describes the state where the TPer's Locking feature is turned off. Locking is not possible. The `Locked` column and `LockOnReset` column values are disregarded.

S1 LockEnabled=T/Locked=T/LockOnReset=non-null

This describes the state where the TPer's Locking feature is turned on. Locking is possible. The `Locked` state is currently True, indicating that the range is locked. `LockOnReset` is non-null, indicating that, upon any of the listed reset events, the range SHALL lock.

S2 LockEnabled=T/Locked=T/LockOnReset=null

This describes the state where the TPer's Locking feature is turned on. Locking is possible. The Locked state is currently True, indicating that the range is locked. `LockOnReset` is "False" (null set), indicating that reset events do not cause the range to lock. The range SHALL maintain current locking state (the value of the `Locked` column remains the same, True) through all resets.

S3 LockEnabled=T/Locked=F/ LockOnReset=non-null

This describes the state where the TPer's Locking feature is turned on. Locking is possible. The `Locked` state is currently False, indicating that the range is not locked. `LockOnReset` is "True" (non-null set), indicating that the listed reset events cause the range to lock.

S4 LockEnabled=T/Locked=F/ LockOnReset=null

This describes the state where the TPer's Locking feature is turned on. Locking is possible. The current Locked state is False, indicating that the range is not locked. `LockOnReset` is "False" (null set), indicating that reset events do not cause the range to lock. The range SHALL maintain current locking state (False in this case) through all reset events.

T0 ResetStateMatch=null/LockEnabled=F

This is the transition state where a reset is occurring and the Locking feature is disabled.

T1 ResetStateMatch=T/LockEnabled=T/Locked=T/ LockOnReset=non-null

This describes a transition state where a reset is occurring, and the range had the accompanying attributes - the locking feature is turned on, the range is locked, and the `LockOnReset` value applies to the currently occurring reset state.

T2 ResetStateMatch=F/LockEnabled=T/Locked=T/ LockOnReset=non-null

This describes a transition state where a reset is occurring, and the range had the accompanying attributes - the locking feature is turned on, the range is locked, and the `LockOnReset` value does not apply to the currently occurring reset state. This state is functionally equivalent to T3.

T3 ResetStateMatch=null /LockEnabled=T/Locked=T/ LockOnReset=null

This describes a transition state where a reset is occurring, and the range had the accompanying attributes - the locking feature is turned on, the range is locked, and the `LockOnReset` value is null. This state is functionally equivalent to T2.

T4 ResetStateMatch=T/LockEnabled=T/Locked=F/ LockOnReset=non-null

This describes a transition state where a reset is occurring, and the range had the accompanying attributes - the locking feature is turned on, the range is not locked, and the `LockOnReset` value applies to the currently occurring reset state.

T5 ResetStateMatch=F/LockEnabled=T/Locked=F/ LockOnReset=non-null

This describes a transition state where a reset is occurring, and the range had the accompanying attributes - the locking feature is turned on, the range is not locked, and the `LockOnReset` value value does not apply to the currently occurring reset state. This state is functionally equivalent to T6.

T6 ResetStateMatch=null /LockEnabled=T/Locked=F/ LockOnReset=null

This describes a transition state where a reset is occurring and the range had the accompanying attributes - the locking feature is turned on, the range is not locked, and the `LockOnReset` value is null. This state is functionally equivalent to T5.

5.7.3.1.2 State Change Descriptions

This section describes the state changes depicted in the picture. In parentheses next to each state transition identifier are the values that change to cause that transition. "Reset" indicates that a reset occurs to cause the state change. "ResetStateMatch" is used to indicate if a reset event type that occurred is applicable or matches the `LockOnReset` column value.

S0:T0 (Reset)

This state change occurs as the result of some device reset event. The locking range with `LockingEnabled`=F exits the reset state into its previous state.

S0:S1 (LockEnabled=T, Locked=T, LockOnReset=non-null)

This state change occurs when the host invokes the `Set` method to change the range's `LockEnabled` column value to True, the `Locked` column value to True, and the `LockOnReset` column value to non-null.

S0:S2 (LockEnabled=T, Locked=T, LockOnReset=null)

This state change occurs when the host invokes the `Set` method to change the range's `LockEnabled` column value to True, the `Locked` column value to True, and the `LockOnReset` column value to null.

S0:S3 (LockEnabled=T, Locked=F, LockOnReset=non-null)

This state change occurs when the host invokes the `Set` method to change the range's `LockEnabled` column value to True, the `Locked` column value to False, and the `LockOnReset` column value to non-null.

S0:S4 (LockEnabled=T, Locked=F, LockOnReset=null)

This state change occurs when the host invokes the `Set` method to change the range's `LockEnabled` column value to True, the `Locked` column value to False, and the `LockOnReset` column value to null.

S1:S0 (LockEnabled=F)

This state change occurs when the host invokes the `Set` method to change the range's `LockEnabled` column value to False from True.

S1:S2 (LockOnReset=null)

This state change occurs when the host invokes the `Set` method to change the range's `LockOnReset` column value to null from non-null. The value of the `LockEnabled` column is still True, and the value of the corresponding `Locked` column is still True.

S1:S3 (Locked=F)

This state change occurs when the host invokes the `Set` method to change the range's `Locked` column to False from True. The value of the corresponding `LockEnabled` column is still True, and the value of the `LockOnReset` column is still non-null.

S1:S4 (Locked=F, LockOnReset=null)

This state change occurs when the host invokes the Set method to change the range's Locked column to False from True, and the value of the LockOnReset column from null to non-null. The value of the corresponding LockEnabled column is still True.

S1:T1 (Reset, ResetStateMatch=T)

This state change occurs as the result of some device reset event, where the reset type matches the value defined in the LockOnReset column.

S1:T2 (Reset, ResetStatematch=F)

This state change occurs as the result of some device reset event, where the reset type does not match the value defined in the LockOnReset column.

S2:S0 (LockEnabled=F)

This state change occurs when the host invokes the Set method to change the range's LockEnabled column value to False from True.

S2:S1 (LockOnReset=non-null)

This state change occurs when the host invokes the Set method to change the range's LockOnReset column value to non-null from null. The value of the LockEnabled column remains True, and the value of the corresponding Locked column remains True.

S2:S3 (Locked=F, LockOnReset=non-null)

This state change occurs when the host invokes the Set method to change the range's Locked column value to False from True, and to change the range's LockOnReset column value to non-null from null. The value of the LockEnabled column remains True.

S2:S4 (Locked=F)

This state change occurs when the host invokes the Set method to change the range's Locked column value to False from True. The value of the corresponding LockEnabled column remains True, and the value of the LockOnReset column remains null.

S2:T3 (Reset)

This state change occurs as the result of some device reset event. The range with a LockOnReset column value of null and other column values of the S2 state exits the T3 state back into the S2 state.

S3:S0 (LockEnabled=F)

This state change occurs when the host invokes the Set method to change the range's LockEnabled column value to False from True.

S3:S1 (Locked=F)

This state change occurs when the host invokes the Set method to change the range's Locked column value to False from True. The value of the corresponding LockEnabled column remains True, and the value of the LockOnReset column remains non-null.

S3:S2 (Locked=T, LockOnReset=null)

This state change occurs when the host invokes the Set method to change the range's Locked column value to True from False, and to change the range's LockOnReset column value to null from non-null. The value of the LockEnabled column remains True.

S3:S4 (LockOnReset=null)

This state change occurs when the host invokes the Set method to change the range's LockOnReset column value to null from non-null. The value of the LockEnabled column remains True, and the value of the corresponding Locked column remains False.

S3:T4 (Reset, ResetStateMatch=T)

This state change occurs as the result of some device reset event, where the reset type matches the value defined in the LockOnReset column.

S3:T5 (Reset, ResetStateMatch=F)

This state change occurs as the result of some device reset event, where the reset type does not match the value defined in the LockOnReset column.

S4:S0 (LockEnabled=F)

This state change occurs when the host invokes the Set method to change the range's LockEnabled column value to False from True.

S4:S1 (Locked=T, LockOnReset=non-null)

This state change occurs when the host invokes the Set method to change the range's Locked column value to False from True, and the value of the LockOnReset column from null to non-null. The value of the corresponding LockEnabled column remains True.

S4:S2 (Locked=T)

This state change occurs when the host invokes the Set method to change the range's Locked column value to False from True. The value of the LockEnabled column remains True, and the value of the LockOnReset column remains null.

S4:S3 (LockOnReset=non-null)

This state change occurs when the host invokes the Set method to change the range's LockOnReset column from null to non-null. The value of the LockEnabled column remains True, and the value of the corresponding Locked column remains False.

S4:T6 (Reset)

This state change occurs as the result of some device reset event. The range with a LockOnReset column value of null and other column values of the S4 state exits the T6 state back into the S4 state.

T0:S0 (Reset recover)

This state change occurs as the result of a recovery from some device reset event. This state change represents behavior of a range for which LockEnabled is False.

T1:S1 (Reset recover)

This state change occurs as the result of a recovery from some device reset event. In this case, the reset event matched that specified in the range's LockOnReset column. This causes the device to enter the S1 state upon reset recovery, with a LockEnabled column value of True, a corresponding Locked column value of True, and the same LockOnReset column value as existed immediately preceding entry to T1.

T2:S1 (Reset recover)

This state change occurs as the result of a recovery from some device reset event. In this case, the reset event did not match that specified in the range's LockOnReset column. This causes the device to recover from the reset with the same LockEnabled, Locked, and LockOnReset column values that existed previous to entry to T2.

T3:S2 (Reset recover)

This state change occurs as the result of a recovery from some device reset event. In this case, the LockOnReset column value of null causes the range to recover from the reset with the same LockEnabled, Locked, and LockOnReset column values that existed previous to entry to T3.

T4:S1 (Reset recover)

This state change occurs as the result of a recovery from some device reset event. In this case, the reset event matched that specified in the range's LockOnReset column. This causes the device to enter the S1 state upon reset recovery, with a LockEnabled column value of True, a corresponding Locked column value of True, and the same LockOnReset column value as existed immediately preceding entry to T4.

T5:S3 (Reset recover)

This state change occurs as the result of a recovery from some device reset event. In this case, the reset event did not match that specified in the range's LockOnReset column. This causes the device to recover from the reset with the same LockEnabled, Locked, and LockOnReset column values that existed previous to entry to T5.

T6:S4 (Reset recover)

This state change occurs as the result of a recovery from some device reset event. In this case, the LockOnReset column value of null causes the range to recover from the reset with the same LockEnabled, Locked, and LockOnReset column values that existed previous to entry to T6.

5.7.3.2 Reading/Writing User Data

This section identifies the device response for attempts by the host to read or write user data.

Table 230 specifies the device response for all cases when the host attempts to read user data.

Table 231 specifies the device response for all cases when the host attempts to write user data.

Table 230 Interface Read Command Access

MBRControl Enable	MBRControl Done	Starting LBA Within MBR	Ending LBA within MBR	ReadLockEnabled for Requested LBA range	ReadLocked for Requested LBA Range	Required Behavior
True	False	True	True	N/A	N/A	Return Data from MBR table
True	False	True	False	N/A	N/A	Transfer no data to the host and terminate the command with a "Data Protection Error" (see [2])
True	False	False	False	False	N/A	Return user data
True	False	False	False	True	False	Return user data
True	False	False	False	True	True	Return all zeroes
True	False	False	False	True	Mixed (when crossing range boundaries)	Transfer no data to the host and terminate the command with a "Data Protection Error" (see [2])
True	True	N/A	N/A	False	N/A	Return user data
True	True	N/A	N/A	True	False	Return user data
True	True	N/A	N/A	True	True	Transfer no data to the host and terminate the command with a "Data Protection Error" (see [2])
True	True	N/A	N/A	True	Mixed (when crossing range boundaries)	Transfer no data to the host and terminate the command with a "Data Protection Error" (see [2])

MBRControl Enable	MBRControl Done	Starting LBA Within MBR	Ending LBA within MBR	ReadLockEnabled for Requested LBA range	ReadLocked for Requested LBA Range	Required Behavior
False	N/A	N/A	N/A	False	N/A	Return user data
False	N/A	N/A	N/A	True	False	Return user data
False	N/A	N/A	N/A	True	True	Transfer no data to the host and terminate the command with a "Data Protection Error" (see [2])
False	N/A	N/A	N/A	True	Mixed (when crossing range boundaries)	Transfer no data to the host and terminate the command with a "Data Protection Error" (see [2])

Table 231 Interface Write Command Access

MBRControl Enable	MBRControl Done	Starting LBA Within MBR	Ending LBA within MBR	WriteLockEnabled for Requested LBA range	WriteLocked for Requested LBA Range	Required Behavior
True	False	True	N/A	N/A	N/A	Transfer no data from the host and terminate the command with a "Data Protection Error" (see [2])
True	False	False	False	False	N/A	Write user data
True	False	False	False	True	False	Write user data
True	False	False	False	True	True	Transfer no data from the host and terminate the command with a "Data Protection Error" (see [2])
True	False	False	False	True	Mixed (when crossing range boundaries)	Transfer no data from the host and terminate the command with a "Data Protection Error" (see [2])
True	True	N/A	N/A	False	N/A	Write user data
True	True	N/A	N/A	True	False	Write user data
True	True	N/A	N/A	True	True	Transfer no data from the host and terminate the command with a "Data Protection Error" (see [2])
True	True	N/A	N/A	True	Mixed (when crossing range boundaries)	Transfer no data from the host and terminate the command with a "Data Protection Error" (see [2])
False	N/A	N/A	N/A	False	N/A	Write user data
False	N/A	N/A	N/A	True	False	Write user data

MBRControl Enable	MBRControl Done	Starting LBA Within MBR	Ending LBA within MBR	WriteLockEnabled for Requested LBA range	WriteLocked for Requested LBA Range	Required Behavior
False	N/A	N/A	N/A	True	True	Transfer no data from the host and terminate the command with a "Data Protection Error" (see [2])
False	N/A	N/A	N/A	True	Mixed (when crossing range boundaries)	Transfer no data and terminate the command with a "Data Protection Error" (see [2])

5.7.3.3 Creating Locking Ranges

The TPer SHALL enforce that the creation of additional `Locking` objects complies with the following rules:

a. Each additional row in this table represents a contiguous “subdivision” of the entire User-Addressable LBA Range.

~~b.~~ The number of rows in this table SHALL NOT exceed the value of the `Locking_Info` table’s `MaxRanges` column + 1. If `MaxRanges` = 0, this is and SHALL only be a one row table.

c. New rows of the `Locking` table are created using the `CreateRow` method. A valid `CreateRow` method SHALL contain the following attributes:

   a. The specified `RangeStart` and `RangeLength` values SHALL NOT overlap the LBA range defined by any other row but the Global Range. The TPer SHALL validate the parameterized LBA range before creating a row. An overlapping request SHALL result in the `CreateRow` method failing and returning an error.

5.7.3.4 Zero Length Locking Ranges

`Locking` objects other than the Global Range that have a `RangeLength` column value of 0 do not have any LBAs under their control, and thus do not overlap any other row even if their `RangeStart` column values match.

Any `Set` method invocation that results in a non-Global Range's `Locking` table row's `RangeLength` column value becoming non-0, or that does not change a non-0 column `RangeLength` column value but does change a `RangeStart` column value, is subject to restrictions defined for overlapping column ranges.

5.7.3.5 Changing RangeStart and RangeLength Values

Begin Informative Content

Modifying an LBA range's `RangeStart` and/or `RangeLength` column values causes the locking and encryption control of a subset of the logical blocks on media to transition from one LBA range to another. Since each LBA range is encrypted with a different media encryption key, the logical blocks whose control transitions from one LBA range to another, when read, will not return any meaningful data (i.e., the data on those logical blocks will be lost).

End Informative Content

Modifications to the `RangeStart`/`RangeLength` column values of a Locking object SHALL NOT cause loss of data for the logical blocks whose control does not transition from one LBA range to another. If

the RangeStart/RangeLength column values are returned to their previous values, it is not guaranteed that the original data is preserved on the logical blocks whose control transitioned from one LBA range to another.

5.7.3.6 MBR Table

Begin Informative Content

The intended purpose of the MBR table is to store code to be processed by the host after a power cycle of the host and TPer, for example as part of the system bootstrapping process where the host's BIOS reads the contents of LBA 0 and executes the instructions stored there. One use of this feature is to load a pre-boot authentication program for authenticating the system's user and unlocking the Storage Device.

The size of the MBR table is retrievable from the Table table of the SP that incorporates the Locking template.

End Informative Content

The values in the MBR table SHALL only be modifiable by properly authenticated Set method invocations, even during the boot process. The size of the table is Security Subsystem Class (SSC) specific, and is to be specified in the description of each SSC that supports the use of this functionality.

Media encryption does not apply to the code stored in the MBR table, since this code is part of the secure area and media encryption applies to the user LBA ranges defined in the Locking table.

MBR table data is byte addressable, so the host is required to map byte addresses to LBA addresses when performing Get/Set operations on the MBR table.

5.7.3.7 Re-encryption

The host has the following re-encryption responsibilities:

a. Configures re-encryption options

b. Initiates re-encryption operations

c. Manages error recovery strategy when TPer detects errors. The host defines the next re-encryption state

d. Optionally, acknowledge re-encryption completion

The TPer has the following re-encryption responsibilities:

a. Maintain persistent re-encryption state and status information across power cycles.

b. Quiesce and report re-encryption state and status

c. Detect re-encryption errors, completion and reset conditions

Re-encryption and read commands and write commands are concurrent activities. Re-encryption is a TPer background task. As such, synchronization between read command and write command processing and background re-encryption processing is required. The means by which this synchronization is accomplished is implementation dependent. However, the normal Storage Device firmware requires a way to view the re-encryption process so that the proper encryption keys are selected for user data read commands and write commands.

Attempts to modify the RangeStart and RangeLength columns of a Locking object that is undergoing re-encryption (the Locking object's ReEncryptState column value is not IDLE) SHALL fail and return a non-success status (FAIL) for the invoked method.

Attempts to delete a Locking object that has a ReEncryptState column of ACTIVE SHALL fail and return a non-success status (FAIL) for the invoked method.

When the Global Locking Range is undergoing re-encryption (the Global Range's ReEncryptState column value is not IDLE):

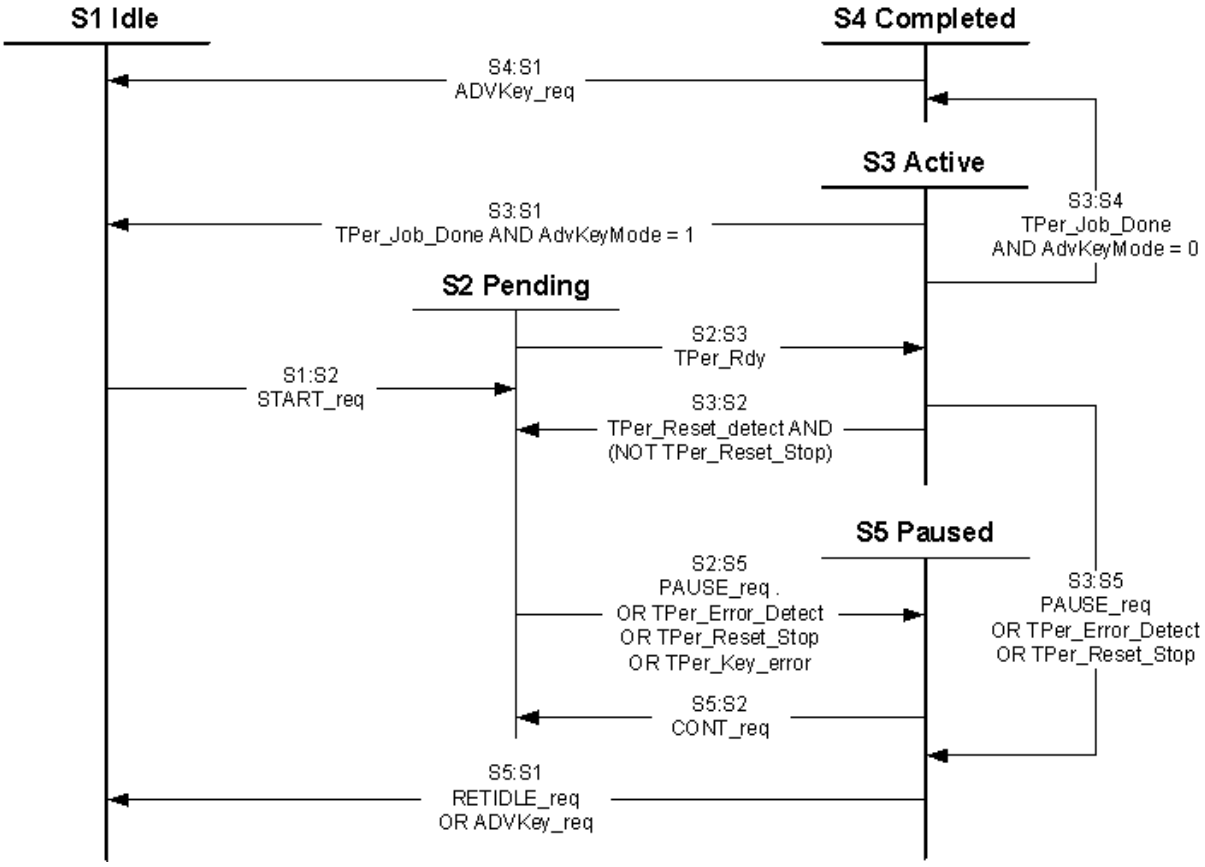
a. Attempts to modify the `RangeStart` and `RangeLength` columns of any `Locking` object SHALL fail and return a non-success status (FAIL) for the invoked method.

b. Attempts to delete any `Locking` object SHALL fail and return a non-success status (FAIL) for the invoked method.

c. Attempts to create a new `Locking` object SHALL fail and return a non-success status (FAIL) for the invoked method.

5.7.3.7.1 Re-encryption State Descriptions

Figure 12 LBA Range Re-encryption State Diagram



S1: IDLE: Re-encryption is idle in this state. No re-encryption is executing for this LBA range.

Transition S1:S2 `START_req` has been detected.

S2: PENDING: This LBA Range's re-encryption process is waiting to start or continue re-encryption

Transition S2:S3 `TPer_Ready` condition is met.

Transition S2:S5 `PAUSE_req` has been detected OR TPer detected error condition

S3: ACTIVE: This LBA Range's re-encryption process is executing
Transition S3:S4 `TPer_Job_Done` condition is met AND `AdvKeyMode = 0`.

Transition S3:S2 `TPer_Reset_detect` condition is met AND `TPer_Reset_Stop` is not met.

Transition S3:S5 A TPer_Error_Detect condition is met OR TPer_Reset_Stop condition is met OR PAUSE_req has been detected

Transition S3:S1 Tper_Job_Done condition is met AND ADVKeyMode = 1 AND re-encryption has completed

S4: COMPLETED: This LBA Range's re-encryption process has completed without errors.

Transition S4:S1 ADVKey_req has been detected.

S5: PAUSED: This LBA Range's re-encryption has temporarily halted. The re-encryption process has been quiesced and awaiting host intervention.

Transition S5:S2 CONT_req has been detected

Transition S5:S1 ADVKey_req OR RETIDLE_req has been detected

5.7.3.7.2 ActiveKey Column Modifications

The following rules define how and when the ActiveKey column value is modified:

a. Host Application directly writes ActiveKey column value
b. When the ReEncryptState column value is COMPLETED or PAUSED and the Host sets the ReEncryptRequest column value to ADVKey_req, the TPer moves the NextKey column value to the ActiveKey column (setting the NextKey column to a NULL UID).
c. When the ReEncryptState column value is ACTIVE AND AdvKeyMode = 1 AND TPer_Job_Done condition is detected, the TPer moves the NextKey column value to the ActiveKey column (setting the NextKey column to a null uid reference).

When ReEncryptState value is PAUSED AND the Host sets ReEncryptRequest to ADVKey_req, the TPer moves the NextKey column value to the ActiveKey column (setting the NextKey column to a NULL UID).

5.7.3.7.3 ReEncryptState Column Values

When the ReEncryptState column value is:

a. 1 = IDLE: re-encryption is not active for this LBA range.
b. 2 = PENDING: This LBA Range's re-encryption process is waiting to start or continue re-encryption.
c. 3 = ACTIVE: This LBA Range's re-encryption process is executing
d. 4 = COMPLETED: This LBA Range's re-encryption process has completed without errors
e. 5 = PAUSED: This LBA Range's re-encryption has temporarily halted.

5.7.3.7.4 ReEncryption Request Attempts

If a Set method invocation attempts to set a value to the ReEncryptRequest column that is not valid for the current ReEncryptState column value, then this Set method invocation SHALL return an error.

a. 1 = START_req: Host requests a new re-encryption process. Only accepted when the ReEncryptState column value is IDLE. The TPer changes the value of the ReEncryptState column to PENDING.
b. 2 = ADVKEY_req: Host requests TPer to change the ReEncryptState column value to IDLE AND move the value of the NextKey column to the ActiveKey column (this move also sets the value of the NextKey column to a null uid reference). This request is only valid when the ReEncryptState column is COMPLETED or PAUSED.

c. 3 = RETIDLE_req: Host requests a return to the IDLE state WITHOUT moving keys. This request provides the host application control over re-encryption resources and background activity. This request is only valid when the ReEncryptState column is PAUSED.
d. 4 = CONT_req: Host requests the re-encryption process transition from ReEncryptState=PAUSED to ReEncryptState=PENDING. Re-encryption begins at (LastReEncryptLBA + 1). Invocation of the Set method to perform this operation SHALL only succeed if the current ReEncryptState column value is PAUSED.
e. 5 = PAUSE_req: Host requests the quiescing of the re-encryption process. Invocation of the Set method to perform this operation SHALL only succeed if the current ReEncryptState column value is ACTIVE or PENDING.

5.7.3.7.5 AdvKeyMode Column Values

a. When AdvKeyMode = 0 AND TPer_Job_Done condition is detected AND ReencryptState is ACTIVE, TPer changes the ReEncryptState value to COMPLETED.

b. When AdvKeyMode = 1 AND TPer_Job_Done condition is detected AND the value of the ReencryptState column is ACTIVE, the TPer changes the ReEncryptState column value to IDLE. In addition, the TPer changes the value of the ActiveKey column to be the value of the NextKey column, and then sets the value of the NextKey column to a null uid reference.

c. When AdvKeyMode = 0 AND Reencryptstate is COMPLETED AND AdvKey_req is True, the TPer changes the ReEncryptState column value to IDLE AND NextKey becomes ActiveKey.

5.7.3.8 Default Logging Settings

The default logging settings associated with the Locking Template methods are:

a. The default logging setting for the Delete object method, the DeleteRow table method, and the Set method on objects in the Locking table SHALL be LogAlways.

b. The default logging setting for the Set method on the Locking_Info table, the MBR_Control table, and the MBR table SHALL be LogAlways.

c. All other methods that apply to the Locking Template-related tables SHALL be as described in the Base Template reference section on Default Logging Settings (See 5.3.4.4).

5.7.4 Life Cycle

5.7.4.1 Locking Template-Specific Life Cycle State Descriptions/Exceptions

An SP issued with the Locking Template has the following characteristics based on the current life cycle state of that SP:

a. **Disabled –** A Locking Template-enabled SP that is in the Disabled state SHALL NOT be able to perform any user-invoked SP operations, with the exceptions noted in section 4.5.2. This MAY result in boot-up failure (if MBR Shadowing is enabled), or the inability to lock or unlock certain LBA ranges for reading and/or writing.

b. **Frozen** – Attempts to open sessions to an SP in the Issued-Frozen state SHALL fail. This MAY result in boot-up failure (if MBR Shadowing is enabled), or the inability to lock or unlock certain LBA ranges for reading and/or writing.

c. **Issued-Disabled-Frozen** – Attempts to open sessions to an SP in the Issued-Disabled-Frozen state SHALL fail. This MAY result in boot-up failure (if MBR Shadowing is enabled), or the inability to lock or unlock certain LBA ranges for reading and/or writing.

5.8 Log Template

5.8.1 Overview

Begin Informative Content

The Log Template is designed to maintain a log of the activities on the SP into which it was issued. The purpose of providing this service is to aid in audits, forensic analysis, and general monitoring of the operation of the SP.

End Informative Content

An issued SP that incorporates the Log Template SHOULD incorporate the Clock Template to exploit the full capabilities of logging. See Section 5.5 for details on the Clock Template.

5.8.1.1 Terminology

Table 232 Log Template Terminology

Term	Definition
Default log	This is the initial log table created for an SP that incorporates the Log Template. By default, all authority operations, access control associations, transaction events, and session startup events log to this table.

5.8.2 Data Structures

5.8.2.1 Log (Object Table)

`Log` tables are object tables that store log entries. Each row in a `Log` table is an entry.

There MAY be more than one `Log` table in an SP. Each of these `Log` tables SHALL have a unique name. Only the default `Log` table, which has the name “Log”, stores System log entries. The Log table described in this section acts as the template for all log tables. Log tables are created using the `CreateLog` method. User-created log tables each have an associated row in the `LogList` table.

Log tables SHALL only be accessible via table level methods. Individual log table rows SHALL NOT have associated rows in the `AccessControl` table.

Table 233 Log Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Prev		log_row_ref
0x02	Next		log_row_ref
0x03	Session		uinteger_4
0x04	SigningAuthority		Authority_object_ref
0x05	SigningAuthName		name
0x06	ExchangeAuthority		Authority_object_ref
0x07	ExchangeAuthName		name
0x08	MonotonicTime		uinteger_8
0x09	ExactTime		clock_time
0x0A	TimeKind		clock_kind
0x0B	LogKind		log_kind
0x0C	Name		name

0x0D	Data		max_bytes_64

5.8.2.1.1 UID

This is the unique identifier for this log entry.

This column SHALL NOT be modifiable by the host.

5.8.2.1.2 Prev

The value of this column is the UID of the log entry (Log table row) added in this table immediately preceding the addition of this log entry. If this row represents the first log entry added to the table, then this value SHALL be the UID of the Log table row that is the final row used prior to reuse of the first row. The value of this column is assigned upon creation of the table, and SHALL be updated only in the event that the number of rows present in the table is modified.

This column SHALL NOT be modifiable by the host.

5.8.2.1.3 Next

The value of this column is the UID of the next log entry (Log table row) added in this table subsequent to this log entry. This value in this column MAY be to a UID of a row that has not yet been used (i.e. has a `LogKind` value of 0). The value of this column is assigned upon creation of the table, and SHALL be updated only in the event that the number of rows present in the table is modified. If this row represents the last log entry added to the table, then this value SHALL be the UID of the first row that is reused.

This column SHALL NOT be modifiable by the host.

5.8.2.1.4 Session

The value of this column is the session number assigned by the TPer. All log entries for a single session SHALL share the same unique session number.

This column SHALL NOT be modifiable by the host.

5.8.2.1.5 SigningAuthority

This value is the UID column value of the Host Signing Authority, if any, that opened the session.

This column SHALL NOT be modifiable by the host.

5.8.2.1.6 SigningAuthName

This is the Name column value of the Host Signing Authority, if any, that opened the session.

This column SHALL NOT be modifiable by the host.

5.8.2.1.7 ExchangeAuthority

This value is the UID column value of the Host Exchange Authority, if any, that opened the session.

This column SHALL NOT be modifiable by the host.

5.8.2.1.8 ExchangeAuthName

This is the Name column value of the Host Exchange Authority, if any, that opened the session

This column SHALL NOT be modifiable by the host.

5.8.2.1.9 MonotonicTime

This is the value of the monotonic counter when the log entry was created, as defined in section 5.5. Note that if the Clock Template was not issued into this SP then this value SHALL be 0.

This column SHALL NOT be modifiable by the host.

5.8.2.1.10 ExactTime

This is the time value (if any) when this log entry was added, as defined in section 5.5. Note that if the Clock Template was not issued into this SP then this value SHALL be zeroes.

This column SHALL NOT be modifiable by the host.

5.8.2.1.11 TimeKind

This is the kind of time used (if any), as defined in section 5.5. Note that if the Clock Template was not issued into this SP then this value SHALL be zero.

This column SHALL NOT be modifiable by the host.

5.8.2.1.12 LogKind

This is the user-provided type of this log entry. If the log is system generated, the value of this column SHALL be "System".

This column SHALL NOT be modifiable by the host.

5.8.2.1.13 Name

This is the name, assigned by the user, upon creation of the log entry. For system generated log entries, the `Name` column value SHALL be "System".

This column SHALL NOT be modifiable by the host.

5.8.2.1.14 Data

This is the actual log information associated with this log entry.

This column SHALL NOT be modifiable by the host.

5.8.2.2 LogList (Object Table)

The `LogList` table is an object table that contains exactly one row for each `Log` table, and contains information about that log.

The `LogList` row with `UID=0x00 0x00 0x0A 0x02 0x00 0x00 0x00 0x01` is automatically created on SP issuance with the name `Log`. A corresponding `Log` table is also created at Issuance. The uid of the created `Log` table is referenced in the `LogList's Log` column.

At creation, the initial `LogList` table row SHALL have default values of `HighSecurity=false` and a `Serial` column value that is a uidref to the `UID` column value of the first row in the Log table to be used.

Table 234 LogList Table Description

Column Number	Column Name	IsUnique	Column Type
0x00	UID		uid
0x01	Name	Yes	name
0x02	CommonName	Yes	name
0x03	Log		Table_object_ref
0x04	Serial		log_row_ref
0x05	HighSecurity		boolean

5.8.2.2.1 UID

This is the unique identifier of this LogList object.

This column SHALL NOT be modifiable by the host.

5.8.2.2.2 Name

This is the name of the associated Log table.

This column SHALL NOT be modifiable by the host.

5.8.2.2.3 CommonName

This is a name that MAY be shared by multiple Log tables.

This column SHALL NOT be modifiable by the host.

5.8.2.2.4 Log

This is the unique identifier of the associated Log table. This is the same as the unique identifier recorded in the `Table` table entry for that log table.

This column SHALL NOT be modifiable by the host.

5.8.2.2.5 Serial

This is the cursor for the associated Log table. The log is circular. The `Serial` column value is the `UID` column value of the most recently added entry in the Log table, and is updated for each log entry, in the order defined by the Next column values in the Log table. Any row of the Log table that has `LogKind` = 0 marks that row as free and unused. At Log table creation, before any log entries have been added, the `Serial` column value SHALL be the null uid.

This column SHALL NOT be modifiable by the host.

5.8.2.2.6 HighSecurity

This column value identifies the relative frequency with which log entries are committed to persistent storage.

When `HighSecurity` is True, every log message is committed to persistent storage when received. When False, messages MAY be queued for later writing (some messages could potentially be lost when a TPer reset occurs).

5.8.3 Methods

5.8.3.1 AddLog (Table Method)

`AddLog` adds a log entry to the `Log` table on which the method was invoked.

Successful invocation of this method automatically sets the value of the `LogKind` column value to 9.

This method is not subject to transactional abort/rollback, and as such if successfully invoked within a Read-Only session its effect is persistent. If multiple Read-Only sessions are open to the same SP, the TPer is required to update the shared log without corruption. Log entries from each session are guaranteed to be in their proper relative order, but no guarantee is made about the relative ordering of entries between separate sessions.

```
LogTableUID.AddLog[
LogEntryName : name,
Data : bytes ]
=>
[ ]
```

5.8.3.1.1 LogEntryName

This is the value that is used for the `Name` column in the added Log object.

5.8.3.1.2 Data

This is the value that is used in the `Data` column in the added Log object.

5.8.3.1.3 AddLog Result

5.8.3.1.3.1 Result

The AddLog method returns an empty list. Success or failure of the method invocation is determinable based on the status code returned in response to the method invocation.

5.8.3.1.4 Fails

a. If the referenced log table does not exist

5.8.3.2 CreateLog (Table Method)

Successful invocation of this method creates a row in the LogList table with the given name and security level, and creates a corresponding Log table with the name given in the method invocation. The Log table described in 5.8.2.1 is used as the template for the new table. ACLs are set for the new row in the LogList table as if CreateRow had been used to create it, as described in 5.8.5. A row in the Table table is created as normal for the new log table.

```
LogListUID.CreateLog[
NewLogTableName : name,
HighSecurity : boolean,
MinSize : uinteger,
MaxSize = uinteger,
Hintsize = uinteger,
CommonName = name ]
=>
[ LogListUID : uid, LogTableUID : uid, Rows : uinteger ]
```

The result of a successful CreateLog method invocation is the uid of the new LogList object, the uid of the new Log table, and the number of rows created in the new Log table.

5.8.3.2.1 NewLogTableName

This is the name assigned to the new Log table, and will be used in the Name columns of both the LogList table and the Table table.

5.8.3.2.2 HighSecurity

This is the initial value to be used in the HighSecurity column of the newly created LogList object.

5.8.3.2.3 MinSize

The MinSize parameter is used to define the initial number of rows allocated for the new table.

5.8.3.2.4 MaxSize

The optional MaxSize parameter defines the host-requested maximum number of rows that MAY be created for the table.

5.8.3.2.5 HintSize

The optional HintSize parameter is used to suggest a number of rows to be created for the table.

5.8.3.2.6 CommonName

The CommonName parameter is the CommonName column value for this table in the Table table. The NewLogTableName-CommonName combination SHALL be unique within the Table table.

5.8.3.2.7 CreateLog Result

5.8.3.2.7.1 LogListUID

This is the UID column value that is assigned to the newly created LogList object in the LogList table.

5.8.3.2.7.2 LogTableUID

This is the `UID` column value that is assigned to the newly created Log table in the `Table` table.

5.8.3.2.7.3 Rows

This value is the number of rows allocated for usage for the table.

5.8.3.2.8 Fails

a. If a log table with the specified name already exists.
b. If there isn't space in the SP for the new table.
c. If metadata/support tables (i.e. `Table`, `Column`, `Method`, or `ACE`) are not all able to create all required rows to support this table.
d. If TPer determines MinSize is too large.

5.8.3.3 ClearLog (Table Method)

All entries in the indicated `Log` table are removed.

```
LogTableUID.ClearLog [ ]
=>
[ ]
```

5.8.3.3.1 ClearLog Result

5.8.3.3.1.1 Result

The `ClearLog` method returns an empty list. Success or failure of the method invocation is determinable based on the status code returned in response to the method invocation.

5.8.3.3.2 Fails

a. If the referenced log table does not exist.

5.8.3.4 FlushLog (Table Method)

Upon successful invocation of this method, all entries that exist only in the main memory and have not yet been committed to media are committed to the indicated `Log` table on media.

When `HighSecurity` is true, `FlushLog` is implicitly invoked after any `AddLog` method invocation.

```
LogTableUID.FlushLog [ ]
=>
[ ]
```

5.8.3.4.1 FlushLog Result

5.8.3.4.1.1 Result

The `FlushLog` method returns an empty list. Success or failure of the method invocation is determinable based on the status code returned in response to the method invocation.

The result is not generated until the persistent storage commit is complete.

5.8.3.4.2 Fails

a. If the referenced log table does not exist.

5.8.4 Descriptions

Logs are cyclical. Implementation SHALL prevent uncontrolled logging recursion.

5.8.4.1 Types of Logging

There are two types of logging:

a. **User** – User logging is the result of invocation of the `AddLog` method on a `Log` table.

b. **System** – System log entries SHALL be stored in the default `Log` table, or the log table designated by the `LogTo` column of the `Authority` or `AccessControl` table. System logging occurs automatically as the result of four classes of events:

   a. Authentication attempts against an authority (success/failure). This includes authentications made as a part of session startup, as well as due to invocations of the `Authenticate` method. Logging for these events are controlled in the `Authority` table.

   b. Method invocations (success/failure). Logging for these events are controlled in the `AccessControl` table.

   c. Transaction events (TransactionStart, TransactionEnd, TransactionAbort). Logging for transaction events is always to the default log table.

   d. Each template reference section includes a description of the default logging values for methods and authorities provided to an SP by that template.

5.8.4.2 Log Entries

Each log entry is a row in a `Log` table. Each of these rows includes columns for the Session ID, uidrefs to Session authorities, the names of those authorities, a timestamp, a monotonic counter value, a `LogKind`, a name for the log entry, and a data field that holds up to 64 bytes of data for a log message.

The value of the data field for system entries is dependent on the value of the `LogKind` column.

The `LogKind` column SHALL have one of the values defined in Table 235.

Table 235 LogKind Column Values

LogKind Column Value	Meaning
0	Available
1	Method failed
2	Method succeeded
3	Authentication attempt failed
4	Authentication attempt succeeded
5	Transaction opened successfully
6	Transaction committed successfully
7	Transaction aborted
8	Session ended successfully
9	User generated
10 to 23	reserved

The structure of the system entry is defined in Table 236.

Table 236 System Log Entry Structure

Bytes	Values
0 to 7	The invoking uid of the method (either a table or object UID, or this SP)

Bytes	Values
9 to 15	The uid of the method involved in the operation. If there is no method, these bytes SHALL be zeroes.
16 to 17	Status code for the operation.

5.8.4.3 Log Table Operation

Logs are maintained in a cyclical manner. All rows in a Log table SHALL be pre-allocated (that is, none of them are free). UIDs for these rows SHALL be assigned at the time of table creation. The value 0 in the `LogKind` column of a row indicates that that row has not yet been used. As log entries are added, the value in the `LogKind` column for each of those used rows changes to reflect the type of log entry added.

If dynamic row allocation is supported, the log table MAY have additional rows created. New rows SHALL be added at the point in the table immediately following the value of the `Serial` column in the LogList object that represents that table. The linking values in the `Prev` and `Next` columns of the affected Log table SHALL be updated accordingly. These newly added rows, like the rows present at table creation, are considered allocated and have the value 0 in the `LogKind` column.

Logs are maintained in a cyclical manner. For efficiency, all rows in a `Log` table should be pre-allocated (that is, none of them is free). The value 0 in the `LogKind` column of a row indicates that that row has not yet been used. As log entries are added, the value in the `LogKind` column for each of those used rows changes to reflect the type of log entry added.

If dynamic row allocation is supported, the log table MAY have additional rows created. New rows are added at the end of the table. These rows, like the rows present at table creation, are considered allocated and have the value 0 in the `LogKind` column. The number of rows in a table MAY be changed by invoking the `Set` method on the `Rows` column of the `Table` table. For information on adding rows to a table, see section 5.3.4.2.1.

A `Log` table row contains a timestamp and uidrefs to and names of the authorities used to start the session in which the activity is being logged. Actual log data (the value stored to the `Data` column) depends on the `LogKind` field (see section 5.8.4.2).

If the Clock Template has not been issued into the SP with the Log Template, when a new entry is created in a `Log` table the values for the `MonotonicTime`, `ExactTime`, and `TimeKind` columns SHALL be 0.

The TPer SHALL atomically add log entries to a log table if multiple read sessions are open to the SP and are affecting that log table.

5.8.4.4 Deleting a Log Table

Because a `Log` table and the `LogList` table is kept in sync, there SHALL be no ACL to allow the `Log` table to be deleted via the `Table` table. The LogList object's `Delete` method SHALL be used to delete a `Log` table. Successful invocation of that method deletes the `Log` table and its associated entries in both the `LogList` table and the `Table` table. For more information, see the Log Template Life Cycle section 5.8.5.

5.8.4.5 Specifying a Log Table

If supported, a host MAY define zero or more Log tables additional to that supplied by the Log Template by default. If this capability is supported, the host MAY specify that specific actions be logged in the host-designated Log tables.

The `LogTo` column of the `AccessControl` table allows the host to associate an access control association to a particular Log table. All access control associations enumerated in the `AccessControl` table at issuance SHALL be logged to the default log.

The LogTo column of the Authority table allows the host to associate an authority's operations to a particular Log table. All authorities at issuance SHALL be logged to the default log.

5.8.4.6 Default Logging Settings

The default logging settings associated with the Log Template methods are:

a. The default logging setting for the Delete object method on objects in the LogList table is LogAlways.

b. The default logging setting for the ClearLog method on all Log tables is LogAlways.

c. All other methods that apply to the Log and LogList tables are as described in the Base Template reference section (See Section 5.3.4.4).

5.8.5 Life Cycle

5.8.5.1 Log Template-Specific Life Cycle State Descriptions/Exceptions

Method invocations that occur to the Admin SP during the Issuance process are logged to the Admin SP default Log table (if the Log Template is part of the Admin SP and if logging of those method invocations is enabled).

An SP issued with the Log Template has the following characteristics based on the current life cycle state of that SP:

a. **Disabled** – A Log Template-enabled SP in the Disabled state SHALL log authority authentication attempts, session startup attempts, and all method invocation attempts (dependent on log settings in the Authority and AccessControl tables). A Log Template-enabled SP that is in the Disabled state SHALL NOT be able to perform any user-invoked SP operations, with the exceptions noted in section 4.5.2.

b. **Frozen –** Attempts to open sessions to an SP in the Issued-Frozen state SHALL fail.

c. **Issued-Disabled-Frozen** – Attempts to open sessions to an SP in the Issued-Disabled-Frozen state SHALL fail.

6 Appendix 1 – Required UID Assignments

6.1 Required UID Assignments Overview

The tables in this section define the required UID assignments for objects, methods, and tables, and table rows as required by this specification.

6.2 Reserved UIDs

The first 2^{16} + 1 uids in each table are reserved for use by the TCG. This represents reservation of the lower 4 bytes of each uid, `0x00 0x00 0x00 0x00` to `0x00 0x01 0x00 0x00` inclusive. Reservation allows categorization of table rows using the lower order bytes of each UID.

These default UIDs enable grouping for the following purposes:

a. To categorize rows of the `Table` table by Template

b. To categorize rows of the `MethodID` table by Template

c. To categorize rows of the `Type` table by type format

Each of these categories in each of the respective tables is grouped into 128 categories with 512 UIDs reserved for each category. The following tables identify the values reserved, and the associated category for which they are reserved. For additional information on UID assignment, see section 3.2.5.3.

Table 237 MethodID Table and Table Table LSB Value Ranges Assignment

Template Name	Range Start	Range End
Unassigned	00 00 00 00	00 00 00 00
Base	00 00 00 01	00 00 02 00
Admin	00 00 02 01	00 00 04 00
Clock	00 00 04 01	00 00 06 00
Crypto	00 00 06 01	00 00 08 00
Locking	00 00 08 01	00 00 0A 00
Log	00 00 0A 01	00 00 0C 00
Unassigned	00 00 0C 01	00 01 00 00
Non-reserved non-assigned	00 01 00 01	FF FE FF FF
Vendor unique (upper 64K)	FF FF 00 00	FF FF FF FF

Table 238 Type Table Reserved LSB Value Ranges

Type Format Name	Reserved Start	Reserved End
Base_Type	00 00 00 01	00 00 02 00
Simple_Type	00 00 02 01	00 00 04 00
Enumeration_Type	00 00 04 01	00 00 06 00
Alternative_Type	00 00 06 01	00 00 08 00
List_Type	00 00 08 01	00 00 0A 00
Restricted_Reference_Type {5}	00 00 0A 01	00 00 0C 00
Restricted_Reference_Type {6}	00 00 0C 01	00 00 0F 00
General_Reference_Type	00 00 0F 01	00 00 10 00
General_Reference_Table_Type	00 00 10 01	00 00 12 00
Name_Value_Name_Type	00 00 12 01	00 00 14 00

Type Format Name	Reserved Start	Reserved End
Name_Value_Uinteger_Type	00 00 14 01	00 00 16 00
Name_Value_Integer_Type	00 00 16 01	00 00 18 00
Struct Type	00 00 18 01	00 00 1A 00
Set Type	00 00 1A 01	00 00 1C 00
Unassigned	00 00 1C 01	00 01 00 00
Unassigned	00 00 00 00	

6.3 Assigned UIDs

The tables in this section display the assigned UIDs required to be used with the associated tables, methods, objects, and table rows. For additional information on UID assignment, see section 3.2.5.3.

The tables in this section describe:

a. Special Purpose UIDs (Table 239) – this descriptive table contains UIDs assigned special meanings/functions in the Core Specification, and a brief description of their functions.

b. Table UIDs (Table 240) – this descriptive table contains UIDs assigned to all table descriptor objects (objects in the `Table` table), as well as the UIDs assigned to the tables themselves.

c. Session Manager Method UIDs (Table 241) – this descriptive table contains the UIDs assigned to Session Manager layer methods.

d. MethodID UIDs (Table 242) – this descriptive table contains the UIDs assigned in the `MethodID` table to all preinstalled methods.

e. Authority UIDs (Table 243) – this descriptive table contains the UIDs assigned in the `Authority` table to each of the default authorities described in the Core Spec.

f. Single Row Table UIDs (Table 244) – this descriptive table contains the UIDs assigned to rows in the tables described in the TCG Core Specification as having only one row.

g. Table Default Rows (Table 245) – In some instances, the TCG Core Specification also defines the UIDs of certain objects within some tables. This descriptive table contains the UIDs assigned to these objects.

h. Template Table UIDs (Table 246) – this descriptive table contains the UIDs assigned to all of the Templates defined in this specification that would appear in the Admin SP's `Template` table.

i. SPTemplates Table UIDs (Table 247) – this descriptive table contains the UIDs assigned to all of the Templates defined in this specification that would appear in an SP's `SPTemplates` table.

j. SecretProtect Table UIDs (Table 248) – this descriptive table contains the UIDs assigned to all of the key storage table's "hidden" columns that could appear in an SP's `SecretProtect` table.

Table 239 Special Purpose UIDs

UID	Purpose
00 00 00 00 00 00 00 00	Used to represent null uid
00 00 00 00 00 00 00 01	Used as the SPUID, the UID that identifies "This SP" – used as the InvokingID for invocation of SP methods
00 00 00 00 00 00 00 FF	Used as the SMUID, the UID that identifies "the Session manager" – used as InvokingID for invocation of Session Manager layer methods
00 00 00 00 00 00 FF xx	Identifies UIDs assigned to Session Manager layer methods, where xx is the UID assigned to a particular method (see Table 241)
00 00 00 0B 00 00 00 01	Used in the C_PIN table's `CharSet` column to indicate that the `GenKey` character set is not restricted (all byte values are legal).

Table 240 Table UIDs

UID of Table Descriptor Object	UID of Table	Table Name	Template
00 00 00 01 00 00 00 01	00 00 00 01 00 00 00 00	Table	Base
00 00 00 01 00 00 00 02	00 00 00 02 00 00 00 00	SPInfo	Base
00 00 00 01 00 00 00 03	00 00 00 03 00 00 00 00	SPTemplates	Base
00 00 00 01 00 00 00 04	00 00 00 04 00 00 00 00	Column	Base
00 00 00 01 00 00 00 05	00 00 00 05 00 00 00 00	Type	Base
00 00 00 01 00 00 00 06	00 00 00 06 00 00 00 00	MethodID	Base
00 00 00 01 00 00 00 07	00 00 00 07 00 00 00 00	AccessControl	Base
00 00 00 01 00 00 00 08	00 00 00 08 00 00 00 00	ACE	Base
00 00 00 01 00 00 00 09	00 00 00 09 00 00 00 00	Authority	Base
00 00 00 01 00 00 00 0A	00 00 00 0A 00 00 00 00	Certificates	Base
00 00 00 01 00 00 00 0B	00 00 00 0B 00 00 00 00	C_PIN	Base
00 00 00 01 00 00 00 0C	00 00 00 0C 00 00 00 00	C_RSA_1024	Base
00 00 00 01 00 00 00 0D	00 00 00 0D 00 00 00 00	C_RSA_2048	Base
00 00 00 01 00 00 00 0E	00 00 00 0E 00 00 00 00	C_AES_128	Base
00 00 00 01 00 00 00 0F	00 00 00 0F 00 00 00 00	C_AES_256	Base
00 00 00 01 00 00 00 10	00 00 00 10 00 00 00 00	C_EC_160	Base
00 00 00 01 00 00 00 11	00 00 00 11 00 00 00 00	C_EC_192	Base
00 00 00 01 00 00 00 12	00 00 00 12 00 00 00 00	C_EC_224	Base
00 00 00 01 00 00 00 13	00 00 00 13 00 00 00 00	C_EC_256	Base
00 00 00 01 00 00 00 14	00 00 00 14 00 00 00 00	C_EC_384	Base
00 00 00 01 00 00 00 15	00 00 00 15 00 00 00 00	C_EC_521	Base
00 00 00 01 00 00 00 16	00 00 00 16 00 00 00 00	C_EC_163	Base
00 00 00 01 00 00 00 17	00 00 00 17 00 00 00 00	C_EC_233	Base
00 00 00 01 00 00 00 18	00 00 00 18 00 00 00 00	C_EC_283	Base
00 00 00 01 00 00 00 19	00 00 00 19 00 00 00 00	C_HMAC_160	Base
00 00 00 01 00 00 00 1A	00 00 00 1A 00 00 00 00	C_HMAC_256	Base
00 00 00 01 00 00 00 1B	00 00 00 1B 00 00 00 00	C_HMAC_384	Base
00 00 00 01 00 00 00 1C	00 00 00 1C 00 00 00 00	C_HMAC_512	Base
00 00 00 01 00 00 00 1D	00 00 00 1D 00 00 00 00	SecretProtect	Base
00 00 00 01 00 00 02 01	00 00 02 01 00 00 00 00	TPerInfo	Admin
00 00 00 01 00 00 02 03	00 00 02 03 00 00 00 00	CryptoSuite	Admin

UID of Table Descriptor Object	UID of Table	Table Name	Template
00 00 00 01 00 00 02 04	00 00 02 04 00 00 00 00	Template	Admin
00 00 00 01 00 00 02 05	00 00 02 05 00 00 00 00	SP	Admin
00 00 00 01 00 00 04 01	00 00 04 01 00 00 00 00	ClockTime	Clock
00 00 00 01 00 00 06 01	00 00 06 01 00 00 00 00	H_SHA_1	Crypto
00 00 00 01 00 00 06 02	00 00 06 02 00 00 00 00	H_SHA_256	Crypto
00 00 00 01 00 00 06 03	00 00 06 03 00 00 00 00	H_SHA_384	Crypto
00 00 00 01 00 00 06 04	00 00 06 04 00 00 00 00	H_SHA_512	Crypto
00 00 00 01 00 00 0A 01	00 00 0A 01 00 00 00 00	Log	Log
00 00 00 01 00 00 0A 02	00 00 0A 02 00 00 00 00	LogList	Log
00 00 00 01 00 00 08 01	00 00 08 01 00 00 00 00	LockingInfo	Locking
00 00 00 01 00 00 08 02	00 00 08 02 00 00 00 00	Locking	Locking
00 00 00 01 00 00 08 03	00 00 08 03 00 00 00 00	MBRControl	Locking
00 00 00 01 00 00 08 04	00 00 08 04 00 00 00 00	MBR	Locking
00 00 00 01 00 00 08 05	00 00 08 05 00 00 00 00	K_AES_128	Locking
00 00 00 01 00 00 08 06	00 00 08 06 00 00 00 00	K_AES_256	Locking

Table 241 Session Manager Method UIDs

Method UID	Method Name
00 00 00 00 00 00 FF 01	Properties
00 00 00 00 00 00 FF 02	StartSession
00 00 00 00 00 00 FF 03	SyncSession
00 00 00 00 00 00 FF 04	StartTrustedSession
00 00 00 00 00 00 FF 05	SyncTrustedSession
00 00 00 00 00 00 FF 06	CloseSession

Table 242 MethodID UIDs

UID in MethodID Table	Method Name	Template
00 00 00 06 00 00 00 01	DeleteSP	Base
00 00 00 06 00 00 00 02	CreateTable	Base
00 00 00 06 00 00 00 03	Delete	Base
00 00 00 06 00 00 00 04	CreateRow	Base
00 00 00 06 00 00 00 05	DeleteRow	Base
00 00 00 06 00 00 00 06	OBSOLETE *	
00 00 00 06 00 00 00 07	OBSOLETE *	
00 00 00 06 00 00 00 08	Next	Base
00 00 00 06 00 00 00 09	GetFreeSpace	Base
00 00 00 06 00 00 00 0A	GetFreeRows	Base
00 00 00 06 00 00 00 0B	DeleteMethod	Base
00 00 00 06 00 00 00 0C	OBSOLETE *	
00 00 00 06 00 00 00 0D	GetACL	Base
00 00 00 06 00 00 00 0E	AddACE	Base
00 00 00 06 00 00 00 0F	RemoveACE	Base
00 00 00 06 00 00 00 10	GenKey	Base

UID in MethodID Table	Method Name	Template
00 00 00 06 00 00 00 11	Reserved for SSC Usage	
00 00 00 06 00 00 00 12	GetPackage	Base
00 00 00 06 00 00 00 13	SetPackage	Base
00 00 00 06 00 00 00 16	Get	Base
00 00 00 06 00 00 00 17	Set	Base
00 00 00 06 00 00 00 1C	Authenticate	Base
00 00 00 06 00 00 02 01	IssueSP	Admin
00 00 00 06 00 00 02 02	Reserved for SSC Usage	
00 00 00 06 00 00 02 03	Reserved for SSC Usage	
00 00 00 06 00 00 04 01	GetClock	Clock
00 00 00 06 00 00 04 02	ResetClock	Clock
00 00 00 06 00 00 04 03	SetClockHigh	Clock
00 00 00 06 00 00 04 04	SetLagHigh	Clock
00 00 00 06 00 00 04 05	SetClockLow	Clock
00 00 00 06 00 00 04 06	SetLagLow	Clock
00 00 00 06 00 00 04 07	IncrementCounter	Clock
00 00 00 06 00 00 06 01	Random	Crypto
00 00 00 06 00 00 06 02	Salt	Crypto
00 00 00 06 00 00 06 03	DecryptInit	Crypto
00 00 00 06 00 00 06 04	Decrypt	Crypto
00 00 00 06 00 00 06 05	DecryptFinalize	Crypto
00 00 00 06 00 00 06 06	EncryptInit	Crypto
00 00 00 06 00 00 06 07	Encrypt	Crypto
00 00 00 06 00 00 06 08	EncryptFinalize	Crypto
00 00 00 06 00 00 06 09	HMACInit	Crypto
00 00 00 06 00 00 06 0A	HMAC	Crypto
00 00 00 06 00 00 06 0B	HMACFinalize	Crypto
00 00 00 06 00 00 06 0C	HashInit	Crypto
00 00 00 06 00 00 06 0D	Hash	Crypto
00 00 00 06 00 00 06 0E	HashFinalize	Crypto
00 00 00 06 00 00 06 0F	Sign	Crypto
00 00 00 06 00 00 06 10	Verify	Crypto
00 00 00 06 00 00 06 11	XOR	Crypto
00 00 00 06 00 00 0A 01	AddLog	Log
00 00 00 06 00 00 0A 02	CreateLog	Log
00 00 00 06 00 00 0A 03	ClearLog	Log
00 00 00 06 00 00 0A 04	FlushLog	Log
00 00 00 06 00 00 08 03	Reserved for SSC Usage	

*Note: See TCG Storage Architecture Core Specification version 0.9.

Table 243 Authority UIDs

UID in Authority Table	Authority Name	Template
00 00 00 09 00 00 00 01	Anybody	Base
00 00 00 09 00 00 00 02	Admins	Base
00 00 00 09 00 00 00 03	Makers	Base

00 00 00 09 00 00 00 04	MakerSymK	Base
00 00 00 09 00 00 00 05	MakerPuK	Base
00 00 00 09 00 00 00 06	SID	Base
00 00 00 09 00 00 00 07	TPerSign	Base
00 00 00 09 00 00 00 08	TPerExch	Base
00 00 00 09 00 00 00 09	AdminExch	Base
00 00 00 09 00 00 02 01	Issuers	Admin
00 00 00 09 00 00 02 02	Editors	Admin
00 00 00 09 00 00 02 03	Deleters	Admin
00 00 00 09 00 00 02 04	Servers	Admin
00 00 00 09 00 00 02 05	Reserve0	Admin
00 00 00 09 00 00 02 06	Reserve1	Admin
00 00 00 09 00 00 02 07	Reserve2	Admin
00 00 00 09 00 00 02 08	Reserve3	Admin

Table 244 Single Row Table Row UIDs

UID of Row	Single Row Table Name
00 00 00 02 00 00 00 01	SPInfo
00 00 02 01 00 00 00 01	TPerInfo
00 00 08 01 00 00 00 01	LockingInfo
00 00 08 03 00 00 00 01	MBRControl

Table 245 Table Default Rows

UID of Row	Table Name	Row Name
00 00 00 0B 00 00 00 01	C_PIN	SID
00 00 02 05 00 00 00 01	SP	Admin
00 00 04 01 00 00 00 01	ClockTime	Clock
00 00 0A 02 00 00 00 01	LogList	Log
00 00 08 02 00 00 00 01	Locking	Global Range

Table 246 Template Table UIDs

UID of Row	Template Name
00 00 02 04 00 00 00 01	Base
00 00 02 04 00 00 00 02	Admin
00 00 02 04 00 00 00 03	Clock
00 00 02 04 00 00 00 04	Crypto
00 00 02 04 00 00 00 05	Log
00 00 02 04 00 00 00 06	Locking
00 00 02 04 00 00 00 07	Reserved for SSC usage

Table 247 SPTemplates Table UIDs

UID of Row	SPTemplates Name
00 00 00 03 00 00 00 01	Base

UID of Row	SPTemplates Name
00 00 00 03 00 00 00 02	Admin
00 00 00 03 00 00 00 03	Clock
00 00 00 03 00 00 00 04	Crypto
00 00 00 03 00 00 00 05	Log
00 00 00 03 00 00 00 06	Locking
00 00 00 03 00 00 00 07	Reserved for SSC usage

Table 248 SecretProtect Table UIDs

UID of Row	Table	ColumnNumber
00 00 00 1D 00 00 00 01	C_PIN	0x03
00 00 00 1D 00 00 00 02	C_RSA_1024	0x06
00 00 00 1D 00 00 00 03	C_RSA_1024	0x07
00 00 00 1D 00 00 00 04	C_RSA_1024	0x08
00 00 00 1D 00 00 00 05	C_RSA_1024	0x09
00 00 00 1D 00 00 00 06	C_RSA_1024	0x0A
00 00 00 1D 00 00 00 07	C_RSA_1024	0x0B
00 00 00 1D 00 00 00 08	C_RSA_2048	0x06
00 00 00 1D 00 00 00 09	C_RSA_2048	0x07
00 00 00 1D 00 00 00 0A	C_RSA_2048	0x08
00 00 00 1D 00 00 00 0B	C_RSA_2048	0x09
00 00 00 1D 00 00 00 0C	C_RSA_2048	0x0A
00 00 00 1D 00 00 00 0D	C_RSA_2048	0x0B
00 00 00 1D 00 00 00 0E	C_AES_128	0x03
00 00 00 1D 00 00 00 0F	C_AES_256	0x03
00 00 00 1D 00 00 00 10	C_EC_160	0x08
00 00 00 1D 00 00 00 11	C_EC_192	0x08
00 00 00 1D 00 00 00 12	C_EC_224	0x08
00 00 00 1D 00 00 00 13	C_EC_256	0x08
00 00 00 1D 00 00 00 14	C_EC_384	0x08
00 00 00 1D 00 00 00 15	C_EC_521	0x08
00 00 00 1D 00 00 00 16	C_EC_163	0x0B
00 00 00 1D 00 00 00 17	C_EC_233	0x09
00 00 00 1D 00 00 00 18	C_EC_283	0x0B
00 00 00 1D 00 00 00 19	C_HMAC_160	0x03
00 00 00 1D 00 00 00 1A	C_HMAC_256	0x03
00 00 00 1D 00 00 00 1B	C_HMAC_384	0x03
00 00 00 1D 00 00 00 1C	C_HMAC_512	0x03
00 00 00 1D 00 00 00 1D	K_AES_128	0x03
00 00 00 1D 00 00 00 1E	K_AES_256	0x03