COMMON INTERFACE TO CRYPTOGRAPHIC MODULES

HIGH ASSURANCE CRYPTOGRAPHIC MODULE INTERFACE SPECIFICATION

V0.6
OCTOBER 2009

ABSTRACT

Common Interface to Cryptographic Modules (CICM) specifies a programming interface to standardize the way software programs manage cryptographic modules and utilize cryptographic services offered by modules. Although a number of interfaces for commercial environments have been standardized and are in use, CICM is the first generic cryptographic interface to be developed that supports cryptographic modules separating two security domains and is thus ideal for the high assurance marketplace. The interface has been designed to also allow less demanding environments to take advantage of its features.

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

1.1 BACKGROUND

Sensitive data is increasingly under attack, whether in transit or at rest. The computer security community has responded to these threats by using cryptography to secure sensitive data. To counter the growing number and types of threats against systems processing sensitive data, module vendors have engineered a diverse set of cryptographic modules.

Systems that require cryptographic protection may utilize various cryptographic services including data encryption, signature generation, hashing, and keystream generation. Cryptographic modules providing these services and the key material they hold must be managed. All of these services have proprietary interfaces that differ significantly among module types, leading to the following problems:

- Replacement of one module type for another and reuse of module-dependent software are inhibited as applications require extensive modifications to adapt to new module types and their proprietary interfaces.
- Developers of systems that host cryptographic modules must accommodate different cryptographic module interfaces for different types of cryptographic modules.
- Test tools and procedures developed for one module usually will not work with other modules.
- Security evaluators must learn multiple module developers' interfaces, increasing evaluation time and expense.

To address these problems, the Common Interface to Cryptographic Modules (CICM) specification offers module developers a set of standard interfaces for the set of operations supported by high assurance cryptographic modules. Although many Application Programming Interfaces (APIs) intended for commercial cryptography are available, the CICM specification was designed for high assurance environments, but may be used in other environments as well.

Modules do not require changes to support the use of CICM. A module-specific abstraction layer between the library implementing CICM interfaces and the module performs the needed translations between the CICM model of a module and the model presented by a specific module. This abstraction component may be provided by the module developer, a module embedder/integrator, or another interested party. This arrangement is analogous to manufacturers of computer peripheral devices providing platform or operating system-specific drivers for their peripheral devices.

CICM is defined using Interface Definition Language (IDL), a specification language that describes a software interface in a language-neutral way. IDL compilers can generate a functionally equivalent CICM interface binding for common programming languages. The use of IDL in CICM is not intended to either prescribe or preclude a particular communications protocol such as General Inter-ORB Protocol (GIOP) between programs in different address spaces or on different devices.

Software developers who require the services of cryptographic modules to perform cryptographic operations use a CICM library in their desired language binding for the specific module from which they intend to access cryptographic services. The specification currently does not specify normative bindings for specific programming languages, although bindings for common languages can be generated from the IDL provided with the specification. However, normative bindings for one or more popular programming languages will be made available in a future release of the specification.

The benefits of using the CICM standard interfaces to access cryptographic services include:

- Provide a common, logical model of cryptographic modules that is straightforward to learn and easy to use.
- Enable the replacement of one cryptographic module for another without significant modifications to the client programs that interact and use the cryptographic module, assuming certain compatibility characteristics between the modules exist.
- Accommodate binding to multiple programming languages.
- Enforce the correct use of the API; in particular, interfaces must be used in the intended order, imposable at compile time or recognizable via static analysis.
- Support high assurance evaluation by enabling evaluators to quickly validate that a particular implementation uses only the required functions in the manner they were intended.

1.2 AUDIENCE

The CICM specification is written for computer programmers, software engineers, and technical architects with a background in data security and cryptography. Knowledge of object-oriented programming concepts is useful when reading IDL definitions. Software engineers may use the specification when developing software that integrates with cryptographic modules. Technical architects may use the specification when designing systems that incorporate cryptographic modules to secure data within the system or between systems.

Although the specification is targeted to software developers who will access module services using a compliant implementation, it also addresses module developers and others who implement library and other support software.

1.3 SCOPE OF THE SPECIFICATION

CICM interfaces provide a common way to access the following services offered by cryptographic modules:

- Cryptographic module management: Includes retrieving information about a specific module, managing access control, managing module events, and loading and managing software packages on modules.
- Key management: Includes the generation, storage, protection, and removal of key material, and support for message exchanges used in key agreement and key transfer protocols.
- Channel management: A channel defines a specific cryptographic transform and encapsulates all attributes associated with that transform. Channel management includes channel instantiation, channel control throughout its lifetime, providing data to a channel for transformation, and extracting transformed data from a channel.

Each of the above services is discussed in detail in Section 2 of the specification and is introduced normatively in Section 3.

The following elements are not addressed by the specification:

- Hardware interfaces, protocols, or design
- Details of specific protocols in the specification (The specification provides a means to move protocol messages into and out of a module, but does not dictate low level protocol.)
- Internal structure of certain types of data elements (e.g., software packages imported into a module, test results extracted from a module) that move through CICM interfaces
- Policy enforcement (The specification provides a means to convey policy elements to the module, but policy enforcement is considered a module responsibility.)
- Organization of the information stored and processed within a module
- Separation of commands/data for multiple security domains that access a single CICM library instance (e.g., Multiple Levels of Security).

1.4 USE CASES

A significant characteristic that differentiates CICM from other cryptographic interfaces is its ability to support cryptographic modules that separate two security domains. The use cases that follow capture this fundamental element of CICM interface design. These use cases can be divided into two basic types:

- Cryptographic transformation of data initiated in one security domain with the result made available in another security domain
- Cryptographic transformation of data within a single security domain: Cryptographic hash or digital signature operations may be initiated in the same security domain where the result is received. Other single domain use cases include data encryption/decryption for storage and keystream/random data generation.

The data-in-transit and data-at-rest use cases illustrated below incorporate multiple security domains, while the final use case depicts a transformation within a single domain.

1.4.1 DATA-IN-TRANSIT

The figure below shows a hardware device with an embedded cryptographic module providing encryption and decryption services between a secure and non-secure network. The secure side protocol logic subsystems access cryptographic services using CICM. In this use case, the High Assurance IP Encryptor (HAIPE) device utilizes CICM to enable the internal protocol logic of the device to access cryptographic services; the network to which the HAIPE device is connected does not interface to the protocol encryptor using CICM.

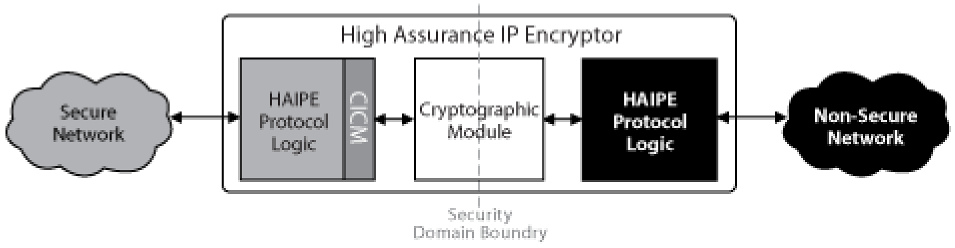


Figure 1. First Data-in-Transit Use Case, HAIPE

The following figure depicts the same use case in its end-to-end configuration.

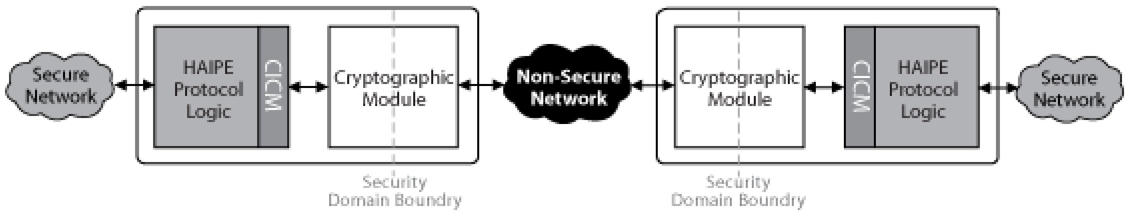


Figure 2. HAIPE Use Case in End-to-End Configuration

A second data-in-transit use case shows a tactical secure radio with an embedded cryptographic module providing encryption and decryption services between a local host and a radio frequency environment. The functional blocks that make up the tactical secure radio are logically identical to those in the first example.

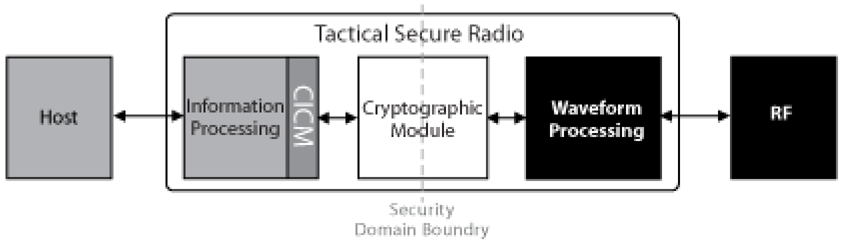


Figure 3. Second Data-in-Transit Use Case, Tactical Secure Radio

1.4.2 DATA-AT-REST

The figure below shows a cryptographic module providing encryption services for data stored on a disk and decryption services for data read from a disk. A file system driver accesses cryptographic services using CICM standard interfaces. This use case could apply to a laptop computer that contains encrypted data; it would prevent access to sensitive data from a lost or stolen laptop.

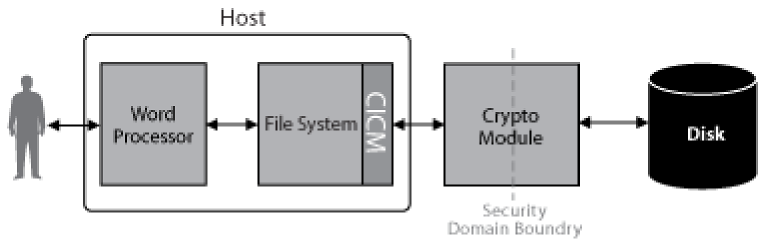


Figure 4. Data-at-Rest Use Case

1.4.3 SINGLE SECURITY DOMAIN

The following figure shows a cryptographic transform within a single security domain (it assumes that the transform does not change the classification of the data). The plaintext is conveyed to the module, transformed by an encryption algorithm, and results in ciphertext. This information is then returned to the same domain from which the plaintext originated. Other natural examples of a single domain use case include signing, which results in a digital signature; hashing, which results in a hash value; and keystream generation, which results in keystream data.

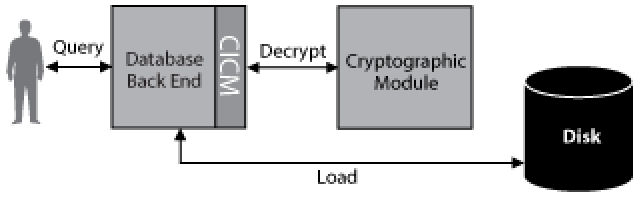


Figure 5. Single Security Domain Use Case

1.5 ASSUMPTIONS

The following assumptions were made in the development of CICM:

- Library implementers may implement part of the specification (refer to Section 4, Conformance and Extensions, for the normative rules).
- A *client program* initiates cryptographic transformations with a cryptographic module via the CICM API. Multiple client programs may concurrently access a single module from a single security domain, but CICM provides no support for controlling access to a module by two or more client programs.
- A module may be implemented as hardware, firmware, or software component, or any combination thereof.
- Although CICM is intended for use in high assurance environments, its use is not precluded in less demanding environments.
- One or more entities between the API library and the module translates CICM commands or interfaces to module-specific commands or interfaces.
- CICM makes no provision in the design of the API to guarantee the confidentiality, integrity, or authenticity of commands and data between a client program calling the API and a module. However, such protections can be applied in the library or runtime system software.
- Specialized hardware (e.g., hardware access tokens, key fill devices, trusted displays) independent of a module may require host (and thus API) interaction or may require no host interaction.

1.6 SPECIFICATION ORGANIZATION

This specification contains normative and informative (non-normative) material. The normative material is prescriptive and provides information that is necessary to claim conformance to the specification. The informative material is for informational purposes; it assists the reader in the understanding and use of the specification but does not contain provisions required for conformance.

Section 1 and Section 2 provide introductory material and are non-normative. Section 3 presents the namespaces, interfaces, methods, and attributes that comprise the specification, and is normative. Section 4, Conformance and Extensions, provides the specification conformance statement and is normative. Appendix A lists the status codes referred to within the document and is normative. Appendix B lists the terms used within the document and is non-normative.

In this document, the key words SHALL and SHOULD are to be interpreted as described in RFC 2119.

1.7 ACKNOWLEDGEMENTS

The MITRE Corporation has been tasked by the U.S. Air Force Cryptographic Modernization Program as part of their support to the Electronic System Center's Cryptologic System Group to pursue the development of an API for high assurance cryptographic modules.

Many individuals participated in the development and review of the CICM specification. The CICM development team consists of Ronald Albuquerque, Samuel Cardman, Greg Carrier, James Cottrell, Shirley Kawamoto, Daniel Lanz, Brent Midwood, Lev Novikov, Brian O'Hanlon, Rick Page, Adam Pennington, and Nguyen Thai. The document production team consists of Mark Dwyer, Amanda Lind, and Brian Parrish.

The CICM team wishes to thank the following individuals for participating in a review of the specification:

Bill Beckwith, Objective Interface Systems
Dennis Bourget, Viasat
Thom Brooke, Linquest Corp.
Randy Culver, RT Logic
John Davis, ITT Corporation
Eric Dube, The MITRE Corporation
Jan Duffy, Rockwell Collins
H.J. Eckles, General Dynamics
Mark Flinchbaugh, Harris Corp.
Carolyn Francisco, The MITRE Corporation
James Howard, L-3 Communications
Kent Kofstad, The MITRE Corporation
Hema Krishnamurthy, ITT Corporation
Prithvi Kumar, The MITRE Corporation
Chip McGrogan, L-3 Communications
Hank Morris, Concurrent Technologies Corp.
Jeff Picciotto, The MITRE Corporation
Tom Plachecki, General Dynamics
Ray Purvis, The MITRE Corporation
Mike Ridge, The MITRE Corporation
Harry Shaffer, The MITRE Corporation
Patrick Smith, Linquest Corp.
Wayne Staats, Rockwell Collins
James Steinwachs, Harris Corp.
Porter Taylor, The MITRE Corporation
Otaway Thomas, Arkham Technology
Bob Walcott, The MITRE Corporation
Blane Yamamoto, SafeNet Mykotronx

2 USING THE SPECIFICATION

2.1 SPECIFICATION CATEGORIES

CICM interfaces are divided into three major categories:

- Module management
- Key management
- Channel management

This section describes CICM support for capabilities made available by cryptographic modules to systems that depend upon high assurance cryptography.

2.2 MODULE MANAGEMENT

CICM's most fundamental element and its technique for abstracting modules is the CryptoModule interface. This interface provides the means to manage individual modules, and to access channel and key interfaces. Individual CryptoModule instances are accessible from the CICMRoot interface, enabling a single CICM library instance to provide access to multiple cryptographic modules available from a particular host.

The CryptoModule interface defines attributes that enable a caller to retrieve information about a module, including the module manufacturer, serial number, and version numbers. This interface also defines specialized attributes called managers that provide access to the services made available by the module. CryptoModule supports the managers described in the sections below.

2.2.1 MANAGING MODULE AUTHENTICATION

Modules may require a host or user to authenticate to the module before the module will enter an operational state, allowing it to accept commands and perform cryptographic transformations. In some cases, a specialized, removable hardware component will perform or participate in the authentication. This hardware component is termed a *hardware access token* in CICM nomenclature, although other communities may use different terminology. Most implementations utilizing hardware access tokens will transfer key material between the token and module, independent of the API. In cases where access tokens are not supported, a user may provide authentication credentials to the module via the API. In still other cases, support for multi-factor authentication will require a token and a user login. Note that the user and token holder may be different entities.

CICM provides interfaces that can be used separately or in combination with one another as appropriate for the system using them and for the authentication mechanisms offered by the module that is utilized by the system. Methods to manage module/token associations are available for systems where hardware access tokens are supported. Login methods and related user management methods are supported for systems that require user login.

The managers below support the functionality described above.

2.2.1.1 Managing Hardware Access Tokens

The token manager defines methods that support associating a token with a module, disassociating a token from a module, and disassociating a module from a token. The manager also supports retrieving a list of token associations on a module and module associations on a token.

2.2.1.2 Managing Users

The user manager defines methods that support adding users to and removing users from a module user database, and associating a user with a module-defined role. The manager also supports listing the user database, and the roles defined and supported by the module.

2.2.1.3 Logging in to a Module from a Host

The login manager defines methods that enable a user configured on a module to login to and logout from a module.

2.2.2 MANAGING SOFTWARE PACKAGES

The package manager defines methods that support importing and managing the executable images that reside on a cryptographic module. These methods enable module software/firmware packages to be imported and other software package management operations to be performed, including retrieving a list of packages, and activating or deleting a specific package.

The package manager enables packages to be imported into a module in segments rather as an atomic unit. This supports modules that must make special provisions to import executable images due to internal storage space limitations.

2.2.3 MANAGING LOGS

Modules generate log entries as they operate. The log manager defines methods that support retrieving individual log entries or extracting an entire log from a module. Additionally, clients may clear individual log entries or the entire module log.

2.2.4 MANAGING TESTS

Modules may incorporate built-in tests to validate that module functionality is operating as designed. Some tests may be externally initiated. The test manager defines methods that support host-initiated module tests.

2.2.5 MANAGING MODULE EVENTS

The event manager defines methods that support registering/unregistering module-generated event notifications received by a client program. Clients can register custom-developed callback procedures, called *listeners*, for specific module events. When the condition associated with a specific listener presents itself, the registered listener is called.

Examples of events for which listeners may be registered include:

- Hardware access token has been inserted or removed.
- Module is ready to receive traffic.
- Alarm condition is raised.

- Hardware zeroization condition raised.
- Continuous module/engine health test failed.
- Usable lifespan of key expired.
- Change in module power state.

2.2.6 MANAGING KEYS AND CHANNELS

Channel and key management interfaces are made available via managers in the CryptoModule interface. The asymmetric and symmetric key manager attributes allow for access to asymmetric keysets and symmetric keys, respectively, and the key database manager offers the ability to zeroize a module and reencrypt the module key database. The channel manager attribute allows channels to be created and used.

2.3 KEY MANAGEMENT

Cryptographic modules utilize key material under their protection as one input to perform a cryptographic transformation. Keys can be

- originate at a Key Infrastructure Component that has a trust relationship with the module
- may be agreed upon between the module and another entity
- may be generated on the module itself
- may be derived from information presented to the module by a client program. Once established on a module, they may be subject to client-initiated management operations or may be used as part of a cryptographic channel to effect cryptographic transformations.

CICM supports separate IDL interfaces for symmetric keys and asymmetric keysets. An asymmetric keyset may comprise an asymmetric key pair, the public and private key components of a keypair, the digital certificate corresponding to the keyset public key, one or more verification certificates in the certificate chain of trust, and related public domain parameters.

The asymmetric and symmetric key manager attributes allow for access to asymmetric keysets and symmetric keys, respectively.

2.3.1 CREATING AND ESTABLISHING KEYS

Keys may be moved into a module in one of several scenarios. Each scenario is described in detail below.

2.3.1.1 No Host Interaction Key Fill

Specialized hardware devices designed to transfer key from a key infrastructure component to a specific cryptographic module may fill key into a module without host involvement and thus no API interaction. In some cases, this process does not support transferring key metadata with a key. This requires host and API interaction to apply metadata to the key inside the module upon completion of the fill.

2.3.1.2 Client Program-Initiated

In some cases, key fill devices require host interaction to initiate a key fill. The API enables a key storage location to be specified or key tagging information to be associated with the filled key prior to the initiation of the fill.

Keys may be imported via the key import method or derived using a text-based secret provided by the user of the client program. Keys also may be generated directly on the module. Each case results in a persistent key.

A key also is implicitly established each time a channel is created using an asymmetric keyset and upon renegotiation. Keys resulting from channel-based key agreement are ephemeral; they are not generally managed outside of a channel. Ephemeral keys also may be destroyed when a channel is destroyed.

2.3.1.3 Module/Key Infrastructure Initiated

A facility to operate a key agreement protocol with an infrastructure component is supported. This facility also enables key material or key revocation information to be authenticated by one of the module's trust anchors, and then loaded into the module.

2.3.2 EXPORTING KEYS

Methods to export key material out of a module are supported. A module may require wrapping the key material prior to export or may disallow this operation.

2.3.3 LOCATING AND RETRIEVING INFORMATION ABOUT A KEY

A method to locate a specific key on a module based upon identification information associated with the key is supported. In addition, the entire key database may be listed.

2.3.4 APPLYING METADATA TO KEYS

Key metadata may be retrieved and set for individual keys. Metadata elements include the key identifier, alias, and classification. Keys, imported via a fill device, that are untagged may require certain metadata to be applied after the conclusion of the load.

2.3.5 PERFORMING OPERATIONS ON KEYS

A number of management operations on keys are supported. Keys may be wrapped (cryptographically protected) in preparation for export, or may be unwrapped after import. Keys may be zeroized, either individually or as the set of all key material on the module. Specialized operations to perform key conversions and updates also are available.

2.3.6 ENABLING REMOTE MANAGEMENT

The specification enables support for various key management-related protocol messages including remote key functions (e.g., remote zeroize or rekey), infrastructure-initiated key revocation, and trust anchor management.

2.4 CHANNEL MANAGEMENT

The CICM channel is the fundamental construct under which one or more related cryptographic transforms are performed, and within which all details and attributes associated with the transform are encapsulated, including the path through the module. Most channels accept data from a port in the local security domain, transform the data, and output the result on a port in another security domain. A channel also may perform transformations within a single security domain, or may accept data for transformation in one domain and output the result in another. The channel type determines which ports must be specified when a channel is created.

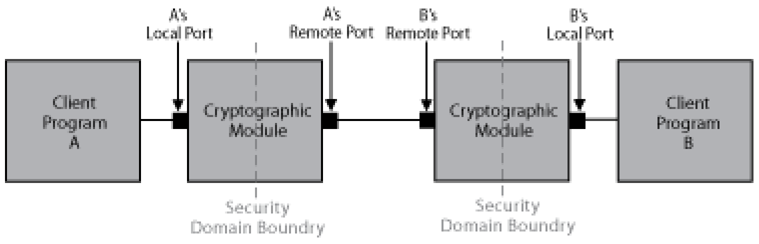


Figure 6. Local and Remote Port Nomenclature for Channels that Operate in Two Security Domains

Three classes of objects are fundamental to the creation and use of CICM channels. A *controller* is used to configure and control a channel. A *stream* enables data to be sent to a module to be transformed, and transformed data to be received using a controller as a foundation. A *conduit* is the sum of a controller and a stream. Thus, the term *channel* is only an abstraction representing the logical path through the module on which cryptographic transformations are performed.

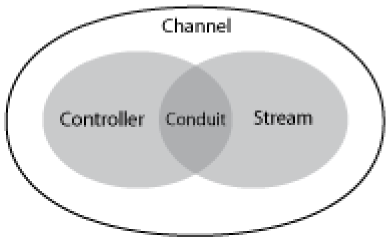


Figure 7. Relationship Between Channel, Conduit, Controller, and Stream

This division of responsibility makes channels very flexible. One client program can be responsible for creating and managing channels with a controller, and another can send data over this preconfigured channel for transformation using a stream. In some environments, data to be transformed never enters the host to pass through the API. Instead, it is clocked directly through the module. In this situation, a controller is configured, but no stream is configured since it would never be used. In other cases, a client program is required to configure the channel and pass data through the channel it configured. In this case, the client program configures a conduit, which incorporates a controller and a stream.

Both controllers and conduits accept symmetric keys, requiring that the client program configuring the channel and its remote peer share the same secret key. Alternatively, all peers may hold their own respective asymmetric keysets, requiring a key negotiation which, upon successful completion, results in each peer holding an ephemeral symmetric key. CICM supports a negotiator IDL interface for this purpose. A successful negotiation results in a negotiated controller or conduit.

CICM supports the following channel types:

- Encryption/decryption, including selective bypass
- Signature generation/verification
- Message Authentication Code (MAC) generation/verification
- Cryptographic hashing
- Keystream generation
- Random/pseudo-random data generation
- Key wrap

- Full bypass.

CICM also supports hybrid channel types. A channel that simultaneously supports encryption and signature, resulting in both ciphertext and a final signature value, is a hybrid channel.

Each of the types above differs in the way it is configured, its configuration options, and how it handles the cryptographic transformation of data. Consider the following examples portraying the diversity of the channel types:

- The encryption channel accepts plaintext to be transformed, and can return the resulting ciphertext directly to the caller or route it a different security domain
- The random data generation channel requires no data for transformation, but emits a random stream
- The signature channel accepts an indeterminate amount of data, and returns an algorithm-specific fixed-sized value
- The hashing channel does not accept a cryptographic key as a parameter, as most of the other channel types do (keyed hashes are supported by MAC channels)
- The decryption channel accepts a state vector input parameter, but does not allow a state vector to be generated.

This diversity results from the fundamental characteristics of the cryptographic primitives that are being abstracted. The channel manager defines the methods that support creating conduits, controllers, streams, and negotiators for each of the channel services listed above.

Interfaces for channel services are organized into 10 namespaces for modularity and as a mechanism to group similar channel types together. A *namespace* is an abstract container that holds related interfaces.

The CICM channel manager provides the ability to perform the functions described in the following sections.

2.4.1 CREATING CHANNELS

Creating a channel requires an awareness of the options available:

- The type of cryptographic operation desired (encryption, hashing, keystream generation, etc.)
- How the channel will be used (control-only, send/receive data only, or both control and send/receive data)
- The type of key that will be used for channels that require a symmetric key or an asymmetric keyset (hybrid channels accept two keys).

Selecting among these options enables the client program developer to determine what channel interface to use.

Consider the following example of a client program configuring a channel to perform encryption using an asymmetric keyset:

- The desired cryptographic operation is "encryption."

- The program only needs to control the channel, not send data over the channel to be encrypted.
- An asymmetric keyset is available, requiring a key agreement protocol negotiation before the channel will be usable.

Given the above information, creating the appropriate type of negotiator from the ChannelManager is straightforward:

```
// Assume references to cryptoModule, localPort, remotePort,
protocol,
//and key.

CICM::Status sCode;

// Retrieve a reference to the ChannelManager:
CICM::ChannelManager channelManager =
cryptoModule._get_channel_manager();

// Create the appropriate Negotiator:
CICM::Encrypt::ControllerNegotiator negotiator;
sCode =
channelManager.negotiate_encrypt_controller(localPort,
remotePort,
      protocol, key, &negotiator);
```

The above call results in the initiation of a key agreement protocol negotiation with its remote peer. To ensure that it is the expected peer, a human user at the client may validate information extracted from the peer's certificate. If the module uses a trusted display, it directly communicates the peer information to the display. Based upon user input at the display, host-independent negotiation is continued or aborted. If no trusted display is available, the client program requests information about the remote peer, displays it at the host for user confirmation, and provides positive confirmation via the API that the peer is valid, allowing the negotiation to continue.

The following example shows the display interactions required to retrieve a negotiated controller:

```
// Retrieve peer information:
CICM::PeerInfo peerInfo;
negotiator.get_remote_info(&peerInfo);

// Assume the user positively confirms that the peer is
valid.

// Complete the negotiation.
CICM::Encrypt::NegotiatedController negotiatedController;
sCode = negotiator.complete(&negotiatedController);
```

The resulting negotiated controller can be used to control and manage the channel.

The sections below describe the channel types that are supported.

2.4.1.1 Encryption and Decryption

CICM defines interfaces to support encryption and decryption between two security domains or within a single security domain. Additional variants are defined including hybrid channels that can concurrently compute integrity values. Another set of variants provides methods to perform encryption/decryption with selective bypass.

If an asymmetric keyset is used to create a channel, a negotiation process is initiated, which results in a negotiated channel. Negotiated versions of hybrid channels also are available. For those negotiator versions that combine encryption with integrity value generation, negotiation applies only to the encryption key specified when the channel is negotiated, not the signature or MAC key.

Channel-based multiple key wrap/unwrap support is provided via a special channels for that purpose.

CICM also supports encryption/decryption channels that operate in *coprocessor mode*. These channels accept their input and return their output as part of the same method call. Where relevant, the integrity value or verification status (verified/not verified) is returned when the final block of the input has been presented for transformation.

Duplex channel configurations that use the same key to perform encrypt and decrypt transformations also are supported. Negotiated versions of the duplex channel also are available.

2.4.1.2 Bypass

Bypass channels capable of defining a path through a module and then bypassing data from one security domain to a different domain are supported. Selective bypass also is supported on encryption and decryption channels.

2.4.1.3 Integrity

Interfaces to compute and validate integrity values using asymmetric key-derived digital signatures or symmetric key-derived MACs are available. A variant on the sign and verify interfaces accepts a previously generated hash value in place of a message.

2.4.1.4 Hashing

A channel to calculate a fixed-length cryptographic hash from an input message is available. Keyed hashes are supported by MAC channels.

2.4.1.5 Keystream Generation

Channels are supported to read keystream from a module.

2.4.1.6 Random Data

Separate interfaces are defined to retrieve random or pseudorandom data from a module.

2.4.2 MANAGING CHANNELS

Only conduits and controllers (not streams) can manage channels. Negotiators also can manage the negotiation aspects of a channel.

The management operations that can be performed on a channel are specific to each channel type, but the following general operations are supported:

- Generating, extracting, and setting state vectors
- Resynchronization
- Initiating a key rollover
- Initiating a key update.

Negotiators support the following general operations:

- Renegotiation
- Changing classification level/acknowledging change of classification level.

Managing state vectors is an important channel management capability. CICM provides a method to explicitly generate a state vector for those algorithms/modes that require a random *initialization vector* (IV), although modules may alternatively generate an IV as a byproduct of channel creation. CICM also provides a method to set the state vector on a channel. This may be used to:

- Set the decrypt channel to the IV generated/used on the encrypt side of a channel.
- Provide a vector on a block-by-block basis for appropriate algorithms/modes or at each time epoch (e.g., time-of-day encryption). In addition, a method is available to take a special state vector called a *synchronization vector* to assist in resynchronizing a channel.

2.4.3 USING CHANNELS

Only conduits and streams (not controllers) can send data for transformation and receive cryptographically transformed data on a channel.

The data operations that can be performed on a channel or stream are specific to each channel type, but the following general operations are supported:

- Sending data on a channel to initiate a cryptographic transformation:
    - Blocking send: Call does not return until data has been sent or the operation times out.
    - Non-blocking send: Call queues data for sending and returns immediately to the caller.
    - Poll: Determines status of non-blocking send operation.
- Receiving transformed data from a channel:
    - Blocking read: Blocks until data becomes available or the operation times out.
    - Non-blocking read: Call queues a buffer to receive data and returns immediately.
    - Poll: Determines status of non-blocking read operation.
    - Notification via callback that data has become available using a ChannelEventListener.

Although it is possible for multiple client programs to use the same stream, the specification provides no facilities to coordinate the parties participating in the communication.

Certain channel services support receiving an "answer" from a channel. For example, signature and hashing channels accept variable amounts of data for transformation before returning a final, constant-sized "answer" (a signature or a hash) to the caller. Composite channels require sending/receiving data and receiving a final "answer" after a discrete unit of data has been transformed.

The figure below depicts the use of a hybrid channel. Plaintext is sent through the CICM API for transformation. The module performs encrypt and sign transformations on the plaintext data. Ciphertext resulting from the encrypt transform emits from the module in a different security domain than the one in which it originated. When it is finished presenting data for transformation, the client program requests the signature that results from the transaction via the API.

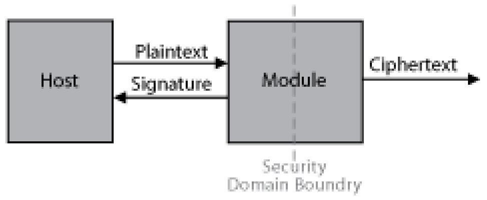


Figure 8. Hybrid SignEncrypt Channel Operations

Consider the following hybrid channel example where a client program configures a channel to simultaneously encrypt and sign data. The encryption operation utilizes a symmetric key, and the signature operation utilizes an asymmetric keyset. Cleartext is sent into the channel for transformation, and the resulting ciphertext emits in another security domain. When all data for transformation has been presented to the channel, the caller calls the associated "end" method to generate and retrieve the signature calculated over the plaintext.

```
// Assume references to cryptoModule, remotePort,
// signKey, encryptKey, signAlgorithm, encryptAlgorithm,
data_first,
// and data_second.

CICM::Status sCode;

// Get the ChannelManager:
CICM::ChannelManager channelManager =
cryptoModule._get_channel_manager();

// Create the appropriate conduit.
CICM::Encrypt::WithSignConduit conduit;
sCode =
channelManager.create_encrypt_with_sign_conduit(remotePort,
signKey, encryptKey, signAlgorithm, encryptAlgorithm,
&conduit);

// Encrypt and sign some data.
sCode = conduit.encrypt(data_first);
sCode = conduit.encrypt(data_second);

// Retrieve the signature.
CICM::SigBuffer signature;
sCode = conduit.end_get_signature(&signature);
```

Each type of channel supports a specific set of channel data operations. Channel types and the data operations they support are listed below:

- Encrypt, selective bypass with encryption, and full bypass write channels: Write data in the local security domain for transformation and output in another security domain.
- Decrypt, selective bypass with decryption, and full bypass read channels: Read transformed data from one security domain into the local security domain.
- Coprocessor channels: Data is presented for transformation and the result received within the same security domain.
- Duplex channels: Read/write exchange between two security domains.
- Keystream and random data generation: Transformation within module results in data stream that emits in the local domain.

For example, channels for encryption and bypass can send data. Channels for decryption, bypass, keystream generation, and random data generation can receive data. Duplex and coprocessor channels can send and receive data.

2.4.4 GROUPING CHANNELS

Controllers and conduits can be grouped to enable certain characteristics to be shared. One characteristic may be the state vector associated with the channels. This supports environments where two or more channels with related security rules supporting a single operation are used within a system. Whenever a shared characteristic is changed on a controller or conduit in a group, the effect of this change is applied to all controllers/conduits in the group.

2.4.5 RECEIVING NOTIFICATION OF CHANNEL EVENTS

The specification defines methods that support managing module event notifications. Similar support is available at the granularity of an individual conduit/controller. Conduits and controllers define methods that support registering/unregistering channel-specific module-generated event notifications captured by a client program. Clients can register custom-developed callback procedures called *listeners* for specific channel events. When the condition associated with a specific listener presents itself, the registered listener is called.

Examples of channel events for which listeners may be registered include:

- Data is available.
- Synchronization with peer has been lost.
- Remote peer no longer available.
- General channel error encountered.

2.4.6 DESTROYING CHANNELS

Conduits and controllers may be destroyed when their services are no longer needed. A channel is destroyed without regard for users who may have pending operations on the channel. Any ephemeral keys associated with the channel also may be destroyed. A stream ceases to function when its associated controller is destroyed. A destroyed channel is removed from any channel groups to which it belongs without effect upon other controllers/conduits in the group.

3 NORMATIVE SPECIFICATION

The namespaces, interfaces, datatypes, methods, and attributes that comprise the specification are presented in a prescriptive manner. The conventions used within the section are presented first. The subsections that follow introduce the API partitioned into its three major categories:

- Module management
- Key management
- Channel management.

For each category, each namespace is described followed by the interfaces contained within it. The datatype, method, and attribute definitions then follow each interface definition.

3.1 INTRODUCTION

3.1.1 CONVENTIONS

Understanding the design of the API and the IDL interfaces that compose CICM requires an understanding of the conventions used in this normative specification. The conventions used are introduced below.

3.1.1.1 Diagrams

This specification includes a number of diagrams showing the relationships among different components. What follows is a brief discussion of the diagram conventions illustrated through a hypothetical example.

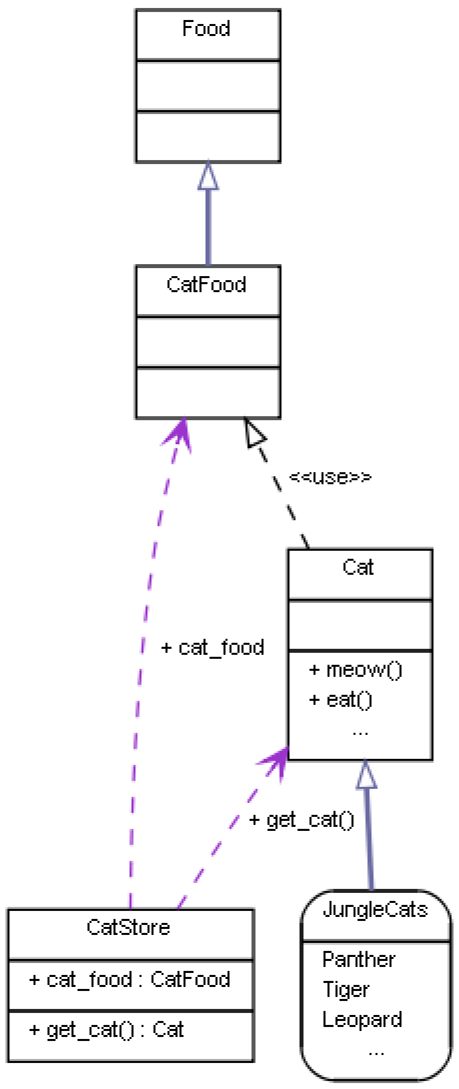


Figure 9. Specification Diagram Conventions

The figure above depicts four hypothetical IDL *interfaces:* `Food`, `CatFood`, `Cat`, and `CatStore`. Each interface is depicted as three stacked rectangles: a name, *attributes*, and *methods*. Interfaces that do not define attributes or methods have an empty rectangle (e.g., `Food`, `CatFood`, `Cat`).

A hypothetical IDL *namespace* called `JungleCats` also is shown. The namespace is depicted as a rounded rectangle with two parts: a name and a list of interfaces in that namespace.

An ellipsis (...) is used to denote an omission that is not shown for conciseness or relevance (e.g., methods of `Cat`, interfaces in `JungleCats`).

Attributes are formatted as:

```
+ name : type
```

Where:

- `name` is the name of the attribute.
- `type` is the attribute type.

For the example above, `CatStore` has an attribute called `cat_food`, which is of type `CatFood`. We reference this attribute as `CatStore::cat_food`. Moreover, since `CatFood` is depicted in this diagram, we draw an arrow indicating a relationship between `CatStore` and `CatFood`.

Methods are formatted as:

```
+ name() : return_type
```

Where:

- `name` is the name of the method.
- `return_type` is the type returned from the method.

For the example above, `CatStore` has a method called `get_cat()`, which has a return of type `Cat`. We reference this method as `CatStore::get_cat()`. As above, we draw an arrow indicating a relationship between `CatStore` and `Cat` since `Cat` appears in the diagram.

Note:

All CICM-defined attributes and methods have public scope. This is indicated by the + symbol in the above definitions.

The figure also depicts two other relationships using arrows: *inheritance* and *dependency*. An inheritance relationship is a type of generalization in which one interface (child) is defined in terms of one or more other interfaces (parents), and causes the child interface to have all of the attributes and methods, if any, defined in the parent interfaces. Inheritance is sometimes referred to as an "is-a" relationship because a child is considered a sub-type of the parent.

In the example above, an arrow is drawn from `CatFood` to `Food` indicating that `CatFood` inherits from (i.e., is a type of) `Food`. Similarly, all of the interfaces in the `JungleCats` namespace inherit from the `Cat` interface.

Dependency refers to the use of one interface by another interface. In the example above, `Cat` has a method `eat()` that has a parameter (not shown) of type `CatFood`. Since both interfaces are depicted in the diagram, an arrow is drawn from `Cat` to `CatFood` to indicate the dependency.

3.1.1.2 IDL Definitions

Throughout this specification, normative definitions are presented using the following format:

Method Cat::eat()

```
void eat( in CatFood food );
```

Provides the cat with food.

Parameters:

`[in]`*food*Food for the cat.

This same format is used to provide definitions for types, constants, attributes, interfaces, and namespaces.

3.1.2 NORMATIVE STATEMENTS

A number of concerns fundamental to the remainder of the normative specification are listed below.

3.1.2.1 Endianness

Endianness is the byte ordering used to represent data stored in a computer or transmitted between computers. CICM requires a big-endian ordering of bytes.

3.1.2.2 Blocking and Non-blocking Calls

All CICM methods block (wait for the operation defined by the method) to complete before returning, unless they are explicitly defined as non-blocking. For example, the CICM::Encrypt::Stream::encrypt method blocks when sending data on a stream to be encrypted, while its sibling CICM::Encrypt::Stream::encrypt_non_blocking is identified not only in its name as non-blocking, but also clearly within the documentation for the method.

3.1.2.3 IDL Language Mapping Conventions

Memory responsibilities and calling conventions shall follow the appropriate IDL language mapping conventions.

3.2 FUNDAMENTAL DEFINITIONS

3.2.1 NAMESPACE CICM

Namespace CICM

```
module CICM {
```

CICM is the top-level namespace for all CICM interfaces and sub-namespaces.

3.2.2 FUNDAMENTAL TYPES

3.2.2.1 General Types

Type CICM::UInt32

```
typedef unsigned long UInt32;
```

Unsigned 32-bit integer.

Type CICM::Bool

```
typedef boolean Bool;
```

Boolean value.

Type CICM::CharString

```
typedef string CharString;
```

Sequence of characters.

Type CICM::Buffer

```
typedef sequence<octet> Buffer;
```

Byte sequence, encapsulating the sequence of bytes, the length of the sequence, and the amount of allocated space.

3.2.2.2 Identifiers

Type CICM::ModuleId
```
typedef CICM::CharString ModuleId;
```
Unique cryptographic module identifier.

Type CICM::ModuleRecord
```
typedef CICM::CharString ModuleRecord;
```
Unique module association record.

Type CICM::TokenRecord
```
typedef CICM::CharString TokenRecord;
```
Unique token association record.

Type CICM::PackageId
```
typedef CICM::CharString PackageId;
```
Unique package identifier.

Type CICM::UserId
```
typedef CICM::CharString UserId;
```
Unique user identifier.

Type CICM::RoleId
```
typedef CICM::CharString RoleId;
```
Unique role identifier.

Type CICM::KeyId
```
typedef CICM::CharString KeyId;
```
Unique key identifier.

Type CICM::TransId
```
typedef CICM::UInt32 TransId;
```
Unique transaction identifier for read/write operations.

3.2.2.3 Status Codes

Type CICM::Status
```
typedef CICM::UInt32 Status;
```
Status of an executed method.
See also:

Appendix A for a full list of status codes.

3.2.2.4 Classifications

Type CICM::Classification

```
typedef CICM::UInt32 Classification;
```

Classification levels.

Constant CICM::C_LEVEL_UNCLASSIFIED

```
const CICM::Classification C_LEVEL_UNCLASSIFIED = 0x0000602F;
```

Value indicating unclassified classification level.

Constant CICM::C_LEVEL_CONFIDENTIAL

```
const CICM::Classification C_LEVEL_CONFIDENTIAL = 0x00006029;
```

Value indicating confidential classification level.

Constant CICM::C_LEVEL_SECRET

```
const CICM::Classification C_LEVEL_SECRET = 0x0000602A;
```

Value indicating secret classification level.

Constant CICM::C_LEVEL_TOP_SECRET

```
const CICM::Classification C_LEVEL_TOP_SECRET = 0x0000602C;
```

Value indicating top secret classification level.

3.2.2.5 Algorithms

Type CICM::HashAlgorithmId

```
typedef CICM::CharString HashAlgorithmId;
```

Unique hash algorithm identifier.

Type CICM::AsymEncrAlgorithmId

```
typedef CICM::CharString AsymEncrAlgorithmId;
```

Unique asymmetric encryption algorithm identifier.

Constant CICM::IMPLICIT_ASYM_ENCR_ALGO

```
const CICM::AsymEncrAlgorithmId IMPLICIT_ASYM_ENCR_ALGO = "IMPLICIT";
```

Value that indicates that the encryption algorithm is implicit in the key being provided to the module.

Type CICM::AsymSigAlgorithmId

```
typedef CICM::CharString AsymSigAlgorithmId;
```

Unique asymmetric signature algorithm identifier.

Constant CICM::IMPLICIT_ASYM_SIG_ALGO

```
const CICM::AsymSigAlgorithmId IMPLICIT_ASYM_SIG_ALGO = "IMPLICIT";
```

Value that indicates that the signature algorithm is implicit in the key being provided to the module.

Type CICM::KeyWrapAlgorithmId

```
typedef CICM::CharString KeyWrapAlgorithmId;
```

Unique key wrap algorithm identifier, incorporating both the algorithm and the mode.

Constant CICM::IMPLICIT_KEY_WRAP_ALGO

```
const CICM::KeyWrapAlgorithmId IMPLICIT_KEY_WRAP_ALGO = "IMPLICIT";
```

Value that indicates that the key wrap algorithm is implicit in the key being provided to the module.

Type CICM::SymEncrAlgorithmId

```
typedef CICM::CharString SymEncrAlgorithmId;
```

Unique symmetric encryption algorithm identifier, incorporating both the algorithm and the mode.

Constant CICM::IMPLICIT_SYM_ENCR_ALGO

```
const CICM::SymEncrAlgorithmId IMPLICIT_SYM_ENCR_ALGO = "IMPLICIT";
```

Value that indicates that the encryption algorithm is implicit in the key being provided to the module.

Type CICM::SymMacAlgorithmId

```
typedef CICM::CharString SymMacAlgorithmId;
```

Unique symmetric MAC algorithm identifier.

Constant CICM::IMPLICIT_SYM_MAC_ALGO

```
const CICM::SymMacAlgorithmId IMPLICIT_SYM_MAC_ALGO = "IMPLICIT";
```

Value that indicates that the MAC algorithm is implicit in the key being provided to the module.

Type CICM::ProtocolId

```
typedef CICM::CharString ProtocolId;
```

Unique key agreement protocol identifier.

Constant CICM::IMPLICIT_PROTOCOL_ID

```
const CICM::ProtocolId IMPLICIT_PROTOCOL_ID = "IMPLICIT";
```

Value that indicates that the key agreement protocol is implicit in the message being provided to the module.

3.2.2.6 Ports

Type CICM::RemotePort

```
typedef CICM::UInt32 RemotePort;
```

Remote module port.

Constant CICM::IMPLICIT_REMOTE_PORT

```
const CICM::RemotePort IMPLICIT_REMOTE_PORT = 0xFFFFFF99;
```

Value that indicates that the remote port value is implicit.

Type CICM::LocalPort

```
typedef CICM::UInt32 LocalPort;
```
Local module port.

Constant CICM::IMPLICIT_LOCAL_PORT
```
const CICM::LocalPort IMPLICIT_LOCAL_PORT = 0xFFFFFFBB;
```
Value that indicates that the local port value is implicit.

Constant CICM::FILL_INTERFACE_PORT
```
const CICM::LocalPort FILL_INTERFACE_PORT = 0xFFFFFFEE;
```
Value that represents the port on which keys are filled or exported.

3.2.2.7 State Vector

Type CICM::Vector
```
typedef CICM::Buffer Vector;
```
State vector, used to represent initialization vectors, synchronization vectors, counter values, and time-of-day values.

3.2.2.8 Integrity Buffers

Type CICM::HashBuffer
```
typedef CICM::Buffer HashBuffer;
```
Cryptographic hash.

Type CICM::MACBuffer
```
typedef CICM::Buffer MACBuffer;
```
Message authentication code (MAC).

Type CICM::SigBuffer
```
typedef CICM::Buffer SigBuffer;
```
Cryptographic signature.

3.2.3 FUNDAMENTAL INTERFACES

3.2.3.1 Interface CICM::CICMRoot

Interface CICM::CICMRoot
```
interface CICMRoot {
```
CICMRoot serves as the entry point to the CICM API and enables a specific cryptographic module of potentially many modules available to a host to be selected.

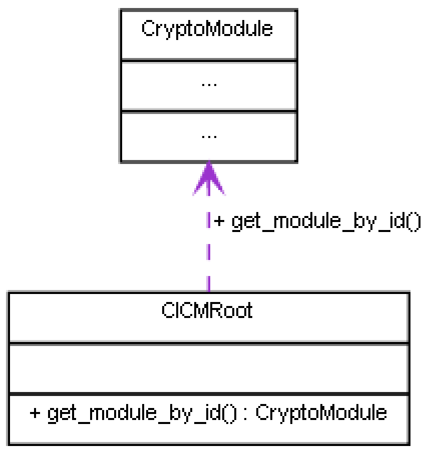


Figure 10. Interface Relationship Diagram for CICMRoot

3.2.3.1.1 CICM::CICMRoot Methods

Method CICM::CICMRoot::get_module_by_id()

```
        CICM::Status get_module_by_id(
                in  CICM::ModuleId id,
                out CICM::CryptoModule crypto_module_ref
        );
```

Returns a reference to the module with the given module unique identifier.

Parameters:

`[in]`	*id*	Unique identifier for the module.
`[out]`	*crypto_module_ref*	Module associated with the given identifier.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_MODULE_DOES_NOT_EXIST

Example (C++):

```
CICM::Status sCode;
CICM::CryptoModule device;

// Instantiate the root object.
CICM::CICMRoot cicm = new CICM::CICMRoot();

// Retrieve a reference to the module
// corresponding to the specified module identifier.
const string MODULE_ID = "CM10293495867";

// If found, [device] refers to the specified crypto module.
sCode = cicm.get_module_by_id(MODULE_ID, &device);
```

3.2.3.2 Interface CICM::CryptoModule

Interface CICM::CryptoModule

```
interface CryptoModule {
```

CICM::CryptoModule contains attributes that provide access to module-specific information and attributes that enable access to module *managers*, through which nearly all interface functionality is accessed.

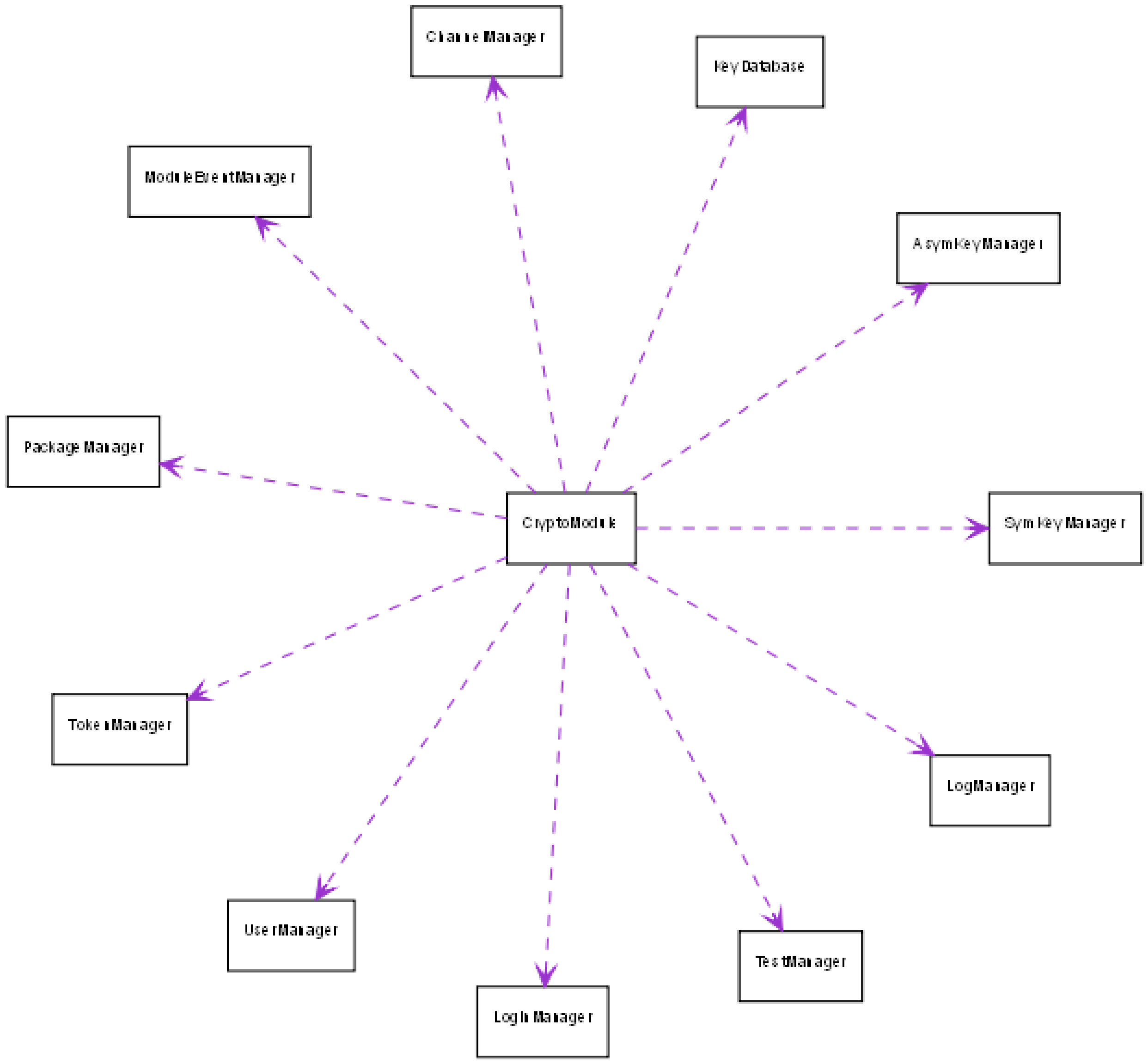


Figure 11. Interface Relationship Diagram for CryptoModule

3.2.3.2.1 CICM::CryptoModule Attributes

Attribute CICM::CryptoModule::module_id

```
        readonly attribute CICM::ModuleId module_id;
```

Unique identifier for this module.

Attribute CICM::CryptoModule::manufacturer
```
        readonly attribute CICM::CharString manufacturer;
```
Name of cryptographic module manufacturer.

Attribute CICM::CryptoModule::model
```
        readonly attribute CICM::CharString model;
```
Model of cryptographic module.

Attribute CICM::CryptoModule::serial_number
```
        readonly attribute CICM::CharString serial_number;
```
Serial number of cryptographic module.

Attribute CICM::CryptoModule::module_version
```
        readonly attribute CICM::CharString module_version;
```
Hardware version of cryptographic module.

Attribute CICM::CryptoModule::software_version
```
        readonly attribute CICM::CharString software_version;
```
Currently executing software/firmware version number.

Attribute CICM::CryptoModule::driver_version
```
        readonly attribute CICM::CharString driver_version;
```
CICM module-specific abstraction layer version number.

Attribute CICM::CryptoModule::library_version
```
        readonly attribute CICM::CharString library_version;
```
CICM library version number.

Attribute CICM::CryptoModule::role
```
        readonly attribute CICM::RoleId role;
```
Current security role in which module is operating.

Attribute CICM::CryptoModule::date_time
```
        attribute CICM::CharString date_time;
```
Current date/time. Intended for use only with module services that require coarse-grained time (e.g., timestamp on a log), not for time-of-day encryption.

Attribute CICM::CryptoModule::sym_key_manager
```
        readonly attribute CICM::SymKeyManager sym_key_manager;
```
Reference to CICM::SymKeyManager.

Attribute CICM::CryptoModule::asym_key_manager
```
        readonly attribute CICM::AsymKeyManager asym_key_manager;
```
Reference to CICM::AsymKeyManager.

Attribute CICM::CryptoModule::key_database

```
        readonly attribute CICM::KeyDatabase key_database;
```
Reference to CICM::KeyDatabase.

Attribute CICM::CryptoModule::channel_manager
```
        readonly attribute CICM::ChannelManager channel_manager;
```
Reference to CICM::ChannelManager.

Attribute CICM::CryptoModule::event_manager
```
        readonly attribute CICM::ModuleEventManager event_manager;
```
Reference to CICM::ModuleEventManager.

Attribute CICM::CryptoModule::package_manager
```
        readonly attribute CICM::PackageManager package_manager;
```
Reference to CICM::PackageManager.

Attribute CICM::CryptoModule::token_manager
```
        readonly attribute CICM::TokenManager token_manager;
```
Reference to CICM::TokenManager.

Attribute CICM::CryptoModule::user_manager
```
        readonly attribute CICM::UserManager user_manager;
```
Reference to CICM::UserManager.

Attribute CICM::CryptoModule::login_manager
```
        readonly attribute CICM::LoginManager login_manager;
```
Reference to CICM::LoginManager.

Attribute CICM::CryptoModule::test_manager
```
        readonly attribute CICM::TestManager test_manager;
```
Reference to CICM::TestManager.

Attribute CICM::CryptoModule::log_manager
```
        readonly attribute CICM::LogManager log_manager;
```
Reference to CICM::LogManager.

3.2.3.2.2 CICM::CryptoModule Methods

Method CICM::CryptoModule::configure_fill_interface()
```
        CICM::Status configure_fill_interface(
                in  CICM::Buffer interface_parameters,
                in  CICM::LocalPort fill_port
        );
```
Configure a module key fill interface.

Remarks:

This method accepts an opaque buffer containing a module-specific data structure specifying fill port configuration parameters.

The format of the interface parameters value is not defined by CICM. The Implementation Conformance Statement (see Section 4, Conformance and Extensions) must reference a standard format or define a module developer-specific format implemented by the module for this datatype.

Parameters:

`[in]`	*interface_parameters*	Opaque buffer containing the fill interface configuration parameters.
`[in]`	*fill_port*	Fill port to configure.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_INVALID_DATA_BUFFER, S_KEY_FILL_DEVICE_NOT_CONNECTED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::CryptoModule::reset()

```
CICM::Status reset();
```

Perform a software-initiated reset on the module.

Remarks:

This method attempts to restart a module in the event of a module failure or in the event a module has entered an alarm state. A CICM::S_OK status denotes that the command was accepted by the module or runtime system, not that any specific action has been initiated as a result of the reset request.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

3.2.3.3 Interface CICM::Iterator

Interface CICM::Iterator

```
interface Iterator {
```

Interface from which other iterators are inherited.

Remarks:

The specification does not define any specific order in which iterated elements are returned.

3.2.3.3.1 CICM::Iterator Types and Constants

Type CICM::Iterator::Status

```
typedef CICM::UInt32 Status;
```

Indicates whether or not there are more items over which to iterate.

Constant CICM::Iterator::C_ITERATOR_HAS_NEXT

```
const CICM::Iterator::Status C_ITERATOR_HAS_NEXT = 0x00006031;
```

There are more items in the list.

Constant CICM::Iterator::C_ITERATOR_NO_MORE

```
const CICM::Iterator::Status C_ITERATOR_NO_MORE = 0x00006032;
```

There are no more items in the list.

3.2.3.3.2 CICM::Iterator Methods

Method CICM::Iterator::has_next()

```
        CICM::Status has_next(
                out     CICM::Iterator::Status has_next
        );
```

Used with get_next() to determine if one or more additional elements are available to be retrieved.

Remarks:

For elements that have not already been processed, changes in the state of the list/database over which the iterator is being run during the lifetime of the iterator will be reflected in the results from calls to retrieve iterator elements.

Parameters:

`[out]` *has_next* Indicates whether more elements are available to be retrieved.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

3.3 MODULE MANAGEMENT

3.3.1 MANAGING HARDWARE ACCESS TOKENS

Cryptographic modules may rely upon hardware access tokens for two primary functions: to allow subjects (e.g., administrators or users in possession of a token) to be identified and authenticated so that privileges can be enforced and audit log entries can identify the subject; and to unlock all or some subset of cryptographic services. A hardware access token may be associated with a number of specific modules, and a module may likewise be associated with a number of specific tokens. The token management functions below enable tokens and modules to be associated with and disassociated from one another, and allow existing associations to be listed.

The removal of an association between a token and a module is straightforward if both the token and the module are available. However, if either the token or module are unavailable, or if a different module than the one originally associated with the token is used to remove an association from a token, the disassociation is less straightforward.

If a module requires that an administrative token be inserted prior to the token to which the association/disassociation will apply, the methods below may return an CICM::S_TOKEN_NOT_PRESENT or CICM::S_TOKEN_ADMIN_NOT_PRESENT status.

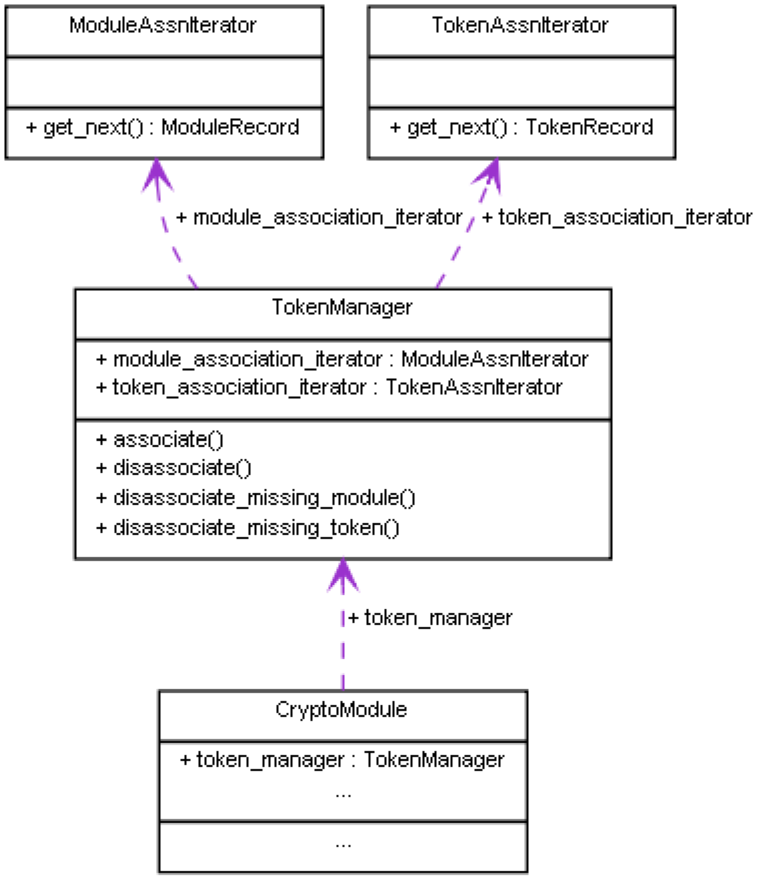
Modules that do not support hardware tokens may instead provide similar support via CICM::LoginManager. Modules may use CICM::LoginManager in tandem with tokens to support multi-factor authentication. See the Managing Module Authentication subsection in Section 2 for additional information.

3.3.1.1 Interface CICM::TokenManager

Interface CICM::TokenManager

```
interface TokenManager {
```

CICM::TokenManager supports associating and disassociating modules and tokens. It is accessed from CICM::CryptoModule via the CICM::CryptoModule::token_manager attribute. CICM::TokenManager constructs the CICM::ModuleAssnIterator and CICM::TokenAssnIterator interfaces.



ModuleAssnIterator
+ get_next() : ModuleRecord
TokenAssnIterator
+ get_next() : TokenRecord
+ module_association_iterator
+ token_association_iterator
TokenManager
+ module_association_iterator : ModuleAssnIterator
+ token_association_iterator : TokenAssnIterator
+ associate()
+ disassociate()
+ disassociate_missing_module()
+ disassociate_missing_token()
+ token_manager
CryptoModule
+ token_manager : TokenManager
...
...

Figure 12. Interface Relationship Diagram for TokenManager

Example (C++):

```
// See CICMRoot::get_module_by_id() to retrieve reference to CryptoModule.
CICM::CryptoModule device;

CICM::Status sCode;
CICM::tokenManager tokenManager;
```

```
// Retrieve reference to the token manager.
tokenManager = device._get_token_manager();

// Associate the current token with the module.
sCode = tokenManager.associate();

// Assume that some time later the token is lost or destroyed.

// Disassociate the token from the module.
CICM::TokenUniqueId tokenId = new CICM::TokenUniqueId("TOKEN1426864");
sCode = tokenManager.disassociate_missing_token(tokenId);
```

3.3.1.1.1 CICM::TokenManager Attributes

Attribute CICM::TokenManager::module_association_iterator

```
        readonly attribute CICM::ModuleAssnIterator
module_association_iterator;
```

Returns an iterator to enable each module identifier associated with the current token to be retrieved.

Remarks:

The returned iterator is set to the beginning of the iterated sequence.

Attribute CICM::TokenManager::token_association_iterator

```
        readonly attribute CICM::TokenAssnIterator
token_association_iterator;
```

Returns an iterator to enable each token identifier associated with the current module to be retrieved.

Remarks:

The returned iterator is set to the beginning of the iterated sequence.

3.3.1.1.2 CICM::TokenManager Methods

Method CICM::TokenManager::associate()

```
        CICM::Status associate(
                out CICM::ModuleRecord module_rec,
                out CICM::TokenRecord token_rec
        );
```

Associate the module and currently-inserted hardware access token.

Remarks:

The module and token record identifiers should be recorded for use in the disassociation process in the event that either the module or the token are no longer available or usable.

The formats of the module and token records are not defined by CICM. The Implementation Conformance Statement (see Section 4, Conformance and Extensions) must reference a standard format or define a module developer-specific format implemented by the module for these datatypes.

Parameters:

`[in]` *module_rec* Module record identifier of the newly associated module.
`[in]` *token_rec* Token record identifier of the newly associated token.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_TOKEN_ACCESS, S_TOKEN_RESOURCES, S_TOKEN_ASSOC_EXISTS, S_TOKEN_ASSOC_AT_MODULE, S_TOKEN_ASSOC_AT_TOKEN, S_TOKEN_ASSOC_GENERAL, S_TOKEN_TIMEOUT

Method CICM::TokenManager::disassociate()

```
        CICM::Status disassociate();
```

Disassociate the module and currently-inserted hardware access token when the associated module and token are both present and both recognize the association.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_TOKEN_ACCESS, S_TOKEN_ASSOC_NOT_EXIST, S_TOKEN_DISASSOC_GENERAL, S_TOKEN_TIMEOUT, S_TOKEN_LAST_ASSOCIATED

Method CICM::TokenManager::disassociate_missing_module()

```
        CICM::Status disassociate_missing_module(
                in  CICM::ModuleRecord module_rec
        );
```

Remove association information from the currently-inserted hardware access token when the associated module is not present.

Remarks:

The module on which this method is being executed is used as a surrogate to perform the disassociation (it is not the module that performed the initial association). The specific module to disassociate from the token is identified by a unique module identifier (e.g., a module serial number). Use CICM::ModuleAssnIterator to retrieve module record identifiers corresponding to modules associated with the inserted token.

The format of the module record is not defined by CICM. The Implementation Conformance Statement (see Section 4, Conformance and Extensions) must reference a standard format or define a module developer-specific format implemented by the module for this datatype.

Parameters:

`[in]` *module_rec* Identifies the module for which module identification information should be removed from the currently-inserted hardware access token.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_TOKEN_ACCESS, S_TOKEN_DISASSOC_GENERAL, S_TOKEN_REC_NOT_FOUND, S_TOKEN_TIMEOUT

Method CICM::TokenManager::disassociate_missing_token()

```
        CICM::Status disassociate_missing_token(
                in  CICM::TokenRecord token_rec
        );
```

Remove association information from the module on which this method is being executed when the associated token is not present.

Remarks:

The specific token to disassociate from the module is identified by a unique token identifier (e.g., a token serial number). Use CICM::TokenAssnIterator to retrieve token record identifiers corresponding to associated tokens from the module.

The format of the token record is not defined by CICM. The Implementation Conformance Statement (see Section 4, Conformance and Extensions) must reference a standard format or define a module developer-specific format implemented by the module for this datatype.

Parameters:

`[in]` *token_rec* Identifies the hardware access token for which token identification information should be removed from the module.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_ADMIN_NOT_PRESENT, S_TOKEN_ASSOC_NOT_EXIST, S_TOKEN_DISASSOC_GENERAL, S_TOKEN_REC_NOT_FOUND, S_TOKEN_TIMEOUT

3.3.1.2 Interface CICM::TokenAssnIterator

Interface CICM::TokenAssnIterator

```
interface TokenAssnIterator : CICM::Iterator {
```

CICM::TokenAssnIterator supports retrieving each token record from the token association list in the module.

3.3.1.2.1 CICM::TokenAssnIterator Inheritance

CICM::TokenAssnIterator inherits from: CICM::Iterator.

3.3.1.2.2 CICM::TokenAssnIterator Methods

Method CICM::TokenAssnIterator::get_next()

```
        CICM::Status get_next(
                out CICM::TokenRecord token_rec_ref
        );
```

Returns a reference to the next token.

Remarks:

Use CICM::Iterator::has_next to determine if additional elements exist.

Parameters:

`[out]` *token_rec_ref* Reference to next token.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

3.3.1.3 Interface CICM::ModuleAssnIterator

Interface CICM::ModuleAssnIterator

```
interface ModuleAssnIterator : CICM::Iterator {
```

CICM::ModuleAssnIterator supports retrieving each module record from the module association list in the token.

3.3.1.3.1 CICM::ModuleAssnIterator Inheritance

CICM::ModuleAssnIterator inherits from: CICM::Iterator.

3.3.1.3.2 CICM::ModuleAssnIterator Methods

Method CICM::ModuleAssnIterator::get_next()

```
        CICM::Status get_next(
                out CICM::ModuleRecord module_rec_ref
        );
```

Returns a reference to the next module record from the module association list in the token.

Remarks:

Use CICM::Iterator::has_next to determine if additional elements exist.

Parameters:

`[out]` *module_rec_ref* Reference to next module record.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

3.3.2 MANAGING USERS

These interfaces enable the management of users in support of password-based login. See the Managing Module Authentication subsection in Section 2 for additional information.

3.3.2.1 Interface CICM::UserManager

Interface CICM::UserManager

```
interface UserManager {
```

CICM::UserManager supports adding a user/password, modifying a user's password, and removing users; and associating and disassociating users from a role. It is accessed from CICM::CryptoModule via the CICM::CryptoModule::user_manager attribute. CICM::UserManager constructs the CICM::UserIdIterator and CICM::RoleIdIterator interfaces.

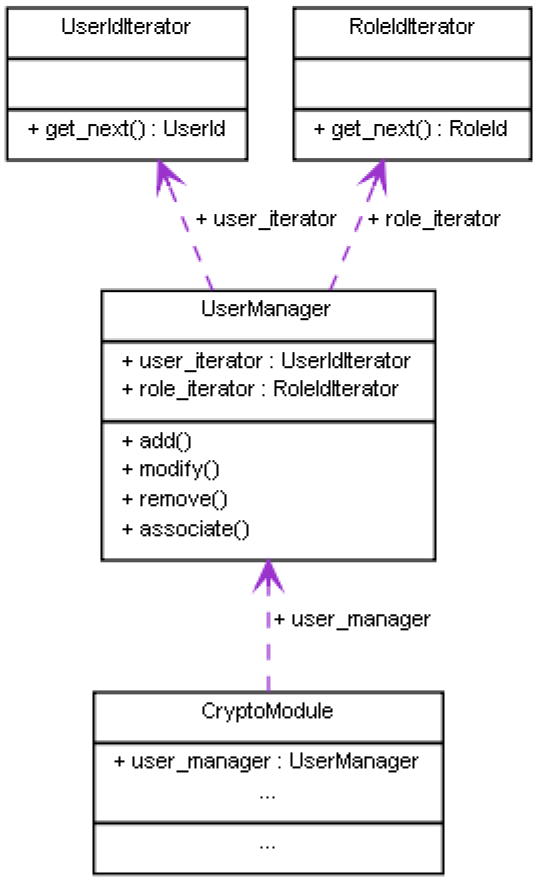


Figure 13. Interface Relationship Diagram for UserManager

Example (C++):

```
// See CICMRoot::get_module_by_id() to retrieve reference to CryptoModule.
CICM::CryptoModule device;

CICM::Status sCode;
CICM::UserManager userManager;

// Retrieve reference to the user manager.
userManager = device._get_user_manager();

// Create the user.
CICM::UserUniqueId userId = "bob_smith";
CICM::CharString password = "p@$$w0rd";
sCode = userManager.add(userId, password);

// Associate the user with a pre-defined module role.
CICM::RoleUniqueId roleId = "administrator";
sCode = userManager.associate(userId, roleId);
```

```
// Destroy the user.
sCode = userManager.remove(userId);
```

3.3.2.1.1 CICM::UserManager Attributes

Attribute CICM::UserManager::user_iterator
```
        readonly attribute CICM::UserIdIterator user_iterator;
```
Returns an iterator to enable an identifier for each user in the module user database to be retrieved.
Remarks:
The returned iterator is set to the beginning of the iterated sequence.

Attribute CICM::UserManager::role_iterator
```
        readonly attribute CICM::RoleIdIterator role_iterator;
```
Returns an iterator to enable an identifier for each role supported by the module to be retrieved.
Remarks:
The returned iterator is set to the beginning of the iterated sequence.

3.3.2.1.2 CICM::UserManager Methods

Method CICM::UserManager::add()
```
        CICM::Status add(
                in  CICM::UserId user,
                in  CICM::CharString password
        );
```
Add a user to the module user database.
Parameters:

`[in]` *user*	New user to add.
`[in]` *password*	New user's password.

Returns:
S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_USERNAME_INVALID, S_USER_EXISTS, S_PASSWORD_INVALID, S_PASSWORD_INVALID_CHAR, S_PASSWORD_INVALID_LEN

Method CICM::UserManager::modify()
```
        CICM::Status modify(
                in  CICM::UserId user,
                in  CICM::CharString password
        );
```
Change the password of a user in the module user database.
Parameters:

`[in]` *user*	User to modify.
`[in]` *password*	User's new password.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_USER_INVALID, S_PASSWORD_INVALID, S_PASSWORD_INVALID_CHAR, S_PASSWORD_INVALID_LEN

Method CICM::UserManager::remove()

```
CICM::Status remove(
        in  CICM::UserId user
);
```

Remove a user from the module user database.

Parameters:

`[in]` *user* User to remove.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_USER_INVALID

Method CICM::UserManager::associate()

```
CICM::Status associate(
        in  CICM::UserId user,
        in  CICM::RoleId role
);
```

Associate a role with the specified user.

Parameters:

`[in]` *user* User to associate.
`[in]` *role* Role to associate with the user.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_USER_INVALID, S_ROLE_INVALID, S_ROLE_ASSOCIATED, S_ROLE_MAX

Method CICM::UserManager::disassociate()

```
CICM::Status disassociate(
        in  CICM::UserId user,
        in  CICM::RoleId role
);
```

Disassociate a role from the specified user.

Parameters:

`[in]` *user* User to disassociate.
`[in]` *role* Role to disassociate from the user.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_USER_INVALID, S_ROLE_INVALID, S_ROLE_NOT_ASSOCIATED

3.3.2.2 Interface CICM::UserIdIterator

Interface CICM::UserIdIterator

```
interface UserIdIterator : CICM::Iterator {
```

CICM::UserIdIterator supports retrieving each user configured on a module.

3.3.2.2.1 CICM::UserIdIterator Inheritance

CICM::UserIdIterator inherits from: CICM::Iterator.

3.3.2.2.2 CICM::UserIdIterator Methods

Method CICM::UserIdIterator::get_next()

```
        CICM::Status get_next(
                out CICM::UserId user_id
        );
```

Returns the next user identifier.

Remarks:

Use CICM::Iterator::has_next to determine if additional elements exist.

Parameters:

`[out]` *user_id* Next user identifier.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

3.3.2.3 Interface CICM::RoleIdIterator

Interface CICM::RoleIdIterator

```
interface RoleIdIterator : CICM::Iterator {
```

CICM::RoleIdIterator supports retrieving each role available on a module.

3.3.2.3.1 CICM::RoleIdIterator Inheritance

CICM::RoleIdIterator inherits from: CICM::Iterator.

3.3.2.3.2 CICM::RoleIdIterator Methods

Method CICM::RoleIdIterator::get_next()

```
        CICM::Status get_next(
                out CICM::RoleId role_id
        );
```
Returns the next role identifier.

Remarks:

Use CICM::Iterator::has_next to determine if additional elements exist.

Parameters:

`[out]` *role_id* Reference to next role identifier.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

3.3.3 MANAGING LOGIN

These interfaces support a user configured on a module to login to a module using a password and, optionally, additional authentication data. See the Managing Module Authentication subsection in Section 2 for additional information.

Modules that support hardware tokens may use the login manager in tandem with the CICM::TokenManager to support multi-factor authentication.

3.3.3.1 Interface CICM::LoginManager

Interface CICM::LoginManager
```
interface LoginManager {
```
CICM::LoginManager supports user login to a module. It is accessed from CICM::CryptoModule via the CICM::CryptoModule::login_manager attribute. CICM::LoginManager constructs the CICM::Login interface. The LoginManager relies upon the CICM::UserManager to manage the users that are specified to the login methods.

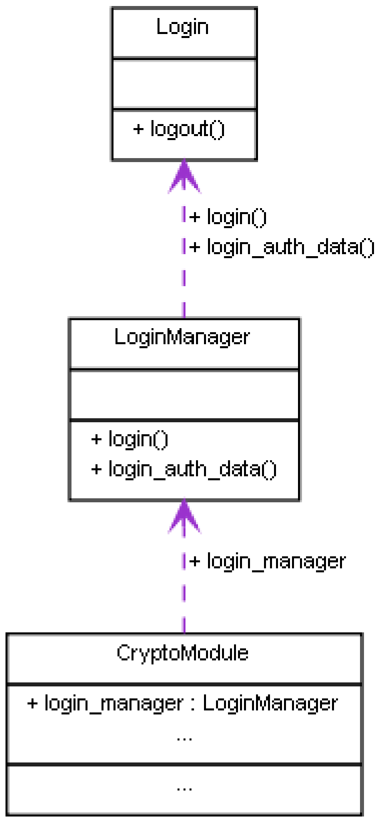


Figure 14. Interface Relationship Diagram for LoginManager

Example (C++):

```
// See CICMRoot::get_module_by_id() to retrieve reference to CryptoModule.
CICM::CryptoModule device;

CICM::Status sCode;
CICM::LoginManager loginManager;
CICM::Login loginRef;

// Retrieve reference to the login manager.
loginManager = device._get_login_manager();

// Login to the module.
CICM::UserUniqueId userId = "bob_smith";
CICM::CharString password = "p@$$w0rd";
sCode = loginManager.add(userId, password, &loginRef);

// Logout from the module.
sCode = loginRef.logout();
```

3.3.3.1.1 CICM::LoginManager Methods

Method CICM::LoginManager::login()

```
        CICM::Status login(
                in  CICM::UserId user,
                in  CICM::CharString password,
                out CICM::Login login_ref
        );
```

Login to the module with username/password.

Parameters:

`[in]`	*user*	User attempting to login.
`[in]`	*password*	User's password.
`[out]`	*login_ref*	Reference to state resulting from successful user login enabling the user to later logout.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_AUTHENTICATION_FAILED, S_USER_AUTHENTICATED

Method CICM::LoginManager::login_auth_data()

```
        CICM::Status login_auth_data(
                in  CICM::UserId user,
                in  CICM::CharString password,
                in  CICM::Buffer auth_data,
                out CICM::Login login_ref
        );
```

Login to the module with username/password, but provide additional (potentially host-stored) authentication data to the module for use in the authentication process.

Remarks:

This may be used in cases where the host supports a virtual token.

The format of the authentication data is not defined by CICM. The Implementation Conformance Statement (see Section 4, Conformance and Extensions) must reference a standard format or define a module developer-specific format implemented by the module for this datatype.

Parameters:

`[in]`	*user*	User attempting to login.
`[in]`	*password*	User's password.
`[in]`	*auth_data*	Additional host-stored authentication data.
`[out]`	*login_ref*	Reference to state resulting from successful user login enabling the user to later logout.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED,

S_INVALID_DATA_BUFFER, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_AUTHENTICATION_FAILED, S_USER_AUTHENTICATED

3.3.3.2 Interface CICM::Login

Interface CICM::Login

```
interface Login {
```

CICM::Login results from a successful user login to a module and enables the user to log out from the module.

3.3.3.2.1 CICM::Login Methods

Method CICM::Login::logout()

```
        CICM::Status logout();
```

Logout of the module.

Remarks:

This may be equivalent to disconnecting a hardware access token from a module in certain systems.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

3.3.4 MANAGING SOFTWARE PACKAGES

These interfaces support software, FPGA images, policy databases, configuration parameters, or other types of executable or interpretable code to be imported into and removed from a module.

3.3.4.1 Interface CICM::PackageManager

Interface CICM::PackageManager

```
interface PackageManager {
```

CICM::PackageManager supports the management of module software packages. It is accessed from CICM::CryptoModule via the CICM::CryptoModule::package_manager attribute. CICM::PackageManager constructs the CICM::PackageImporter, CICM::PackageIterator, and CICM::Package interfaces.

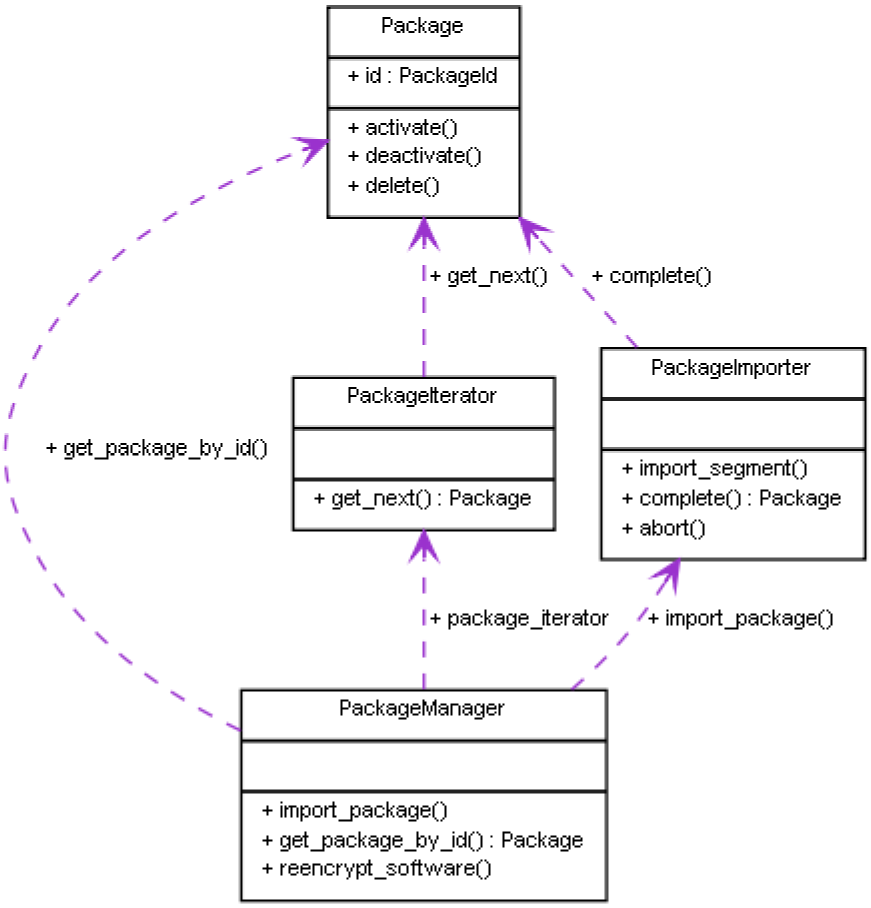


Figure 15. Interface Relationship Diagram for PackageManager

Example (C++):

```
// See CICMRoot::get_module_by_id() to retrieve reference to CryptoModule.
CICM::CryptoModule device;

CICM::Status sCode;
CICM::PackageManager packageManager;
CICM::PackageImporter packageImporter;

// Retrieve reference to the package manager.
packageManager = device._get_package_manager();

// Initialize the import process.
sCode = packageManager.import_package(CICM::Package::C_PACKAGE_FPGA_IMAGE,
        &packageImporter);

// Assume FPGA image data in [fpgaData].
CICM::Buffer fpgaData;
```

```
sCode = packageImporter.import_segment(fpgaData);

// Assume all segments are imported.

// Complete the import process.
CICM::Package fpgaPackage;
sCode = packageImporter.complete(&fpgaPackage);

// If successful, [fpgaPackage] is a reference to the imported package.

// Activate the package.
sCode = fpgaPackage.activate();
```

3.3.4.1.1 CICM::PackageManager Attributes

Attribute CICM::PackageManager::package_iterator

```
        readonly attribute CICM::PackageIterator package_iterator;
```

Returns an iterator to enable a reference to each package loaded on the module to be retrieved.

Remarks:

The returned iterator is set to the beginning of the iterated sequence.

3.3.4.1.2 CICM::PackageManager Methods

Method CICM::PackageManager::import_package()

```
        CICM::Status import_package(
                in  CICM::Package::PackageType package_type,
                out CICM::PackageImporter importer_ref
        );
```

Initiate the process of importing a package into the module.

Remarks:

The CICM::PackageImporter that results from this call is used to import package segments into the module. It is the responsibility of the caller to break a package into segments, import each individual segment, and then call CICM::PackageImporter::complete to receive a reference to the resulting package. Note that the key required to decrypt any encrypted package segments must be referenced within the package and must be available to the module; the key may be explicitly specified by using the CICM::PackageManager::import_package_with_key version of the call.

Parameters:

`[in]`	*package_type*	Type of the package being imported.
`[out]`	*importer_ref*	Reference to package importer interface which enables a package to be imported segment by segment.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED,

S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_PACKAGE_TYPE_INVALID, S_PACKAGE_KEY_NOT_AVAILABLE, S_PACKAGE_KEY_NOT_SPECIFIED

Method CICM::PackageManager::import_package_with_key()

```
CICM::Status import_package_with_key(
        in  CICM::Package::PackageType package_type,
        in  CICM::SymKey key_ref,
        out CICM::PackageImporter importer_ref
);
```

Initiate the process of importing a package into the module, specifying a reference to the key that will be used by CICM::PackageImporter to decrypt each package segment.

Remarks:

The CICM::PackageImporter that results from this call is used to import package segments into the module. It is the responsibility of the caller to break a package into segments, import each individual segment, and then call CICM::PackageImporter::complete to receive a reference to the resulting package.

Parameters:

`[in]`	*package_type*	Type of the package being imported.
`[in]`	*key_ref*	Reference to key to decrypt package segments.
`[out]`	*importer_ref*	Reference to package importer interface which enables a package to be imported segment by segment.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_PACKAGE_TYPE_INVALID

Method CICM::PackageManager::get_package_by_id()

```
CICM::Status get_package_by_id(
        in  CICM::PackageId package_id,
        out CICM::Package package_ref
);
```

Retrieve a reference to a package based upon a unique identifier associated with that package.

Parameters:

`[in]`	*package_id*	Package identifier.
`[out]`	*package_ref*	Reference to package corresponding to the specified identifier.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_NOT_AVAILABLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::PackageManager::reencrypt_software()

```
CICM::Status reencrypt_software();
```

Re-encrypt module software with a key managed by the module.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_INSUFFICIENT_ENTROPY

3.3.4.2 Interface CICM::PackageImporter

Interface CICM::PackageImporter

```
interface PackageImporter {
```

CICM::PackageImporter supports importing software packages, segment by segment. CICM::PackageImporter is constructed by the CICM::PackageManager::import_package and CICM::PackageManager::import_package_with_key methods and may not be instantiated independently. CICM::PackageImporter constructs the CICM::Package interface.

3.3.4.2.1 CICM::PackageImporter Methods

Method CICM::PackageImporter::import_segment()

```
        CICM::Status import_segment(
                in  CICM::Buffer package_data
        );
```

Import one segment of a package.

Remarks:

It is the responsibility of the caller to break a package into segments, import each individual segment, and then call CICM::PackageImporter::complete to receive a reference to the resulting package.

CICM does not specify the structure of the binary data that constitutes the package being imported. The Implementation Conformance Statement (see Section 4, Conformance and Extensions) must reference a standard format or define a module developer-specific format implemented by the module for this datatype.

Parameters:

`[in]` *package_data* Contents of the package.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_PACKAGE_INVALID

Method CICM::PackageImporter::complete()

```
        CICM::Status complete(
                out CICM::Package package_ref
        );
```

Declare the package import complete and retrieve a reference to the resulting package object.

Remarks:

If this method is called before the package is fully loaded, the CICM::S_PACKAGE_INVALID status results.

Parameters:

`[out]` *package_ref* Reference to resulting imported package.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_PACKAGE_INVALID

Method CICM::PackageImporter::abort()

```
    CICM::Status abort();
```

Abort a package import, resetting this CICM::PackageImporter instance, allowing a new package import session to begin.

Remarks:

Segments already imported are discarded.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

3.3.4.3 Interface CICM::Package

Interface CICM::Package

```
interface Package {
```

CICM::Package serves as a reference to a package previously loaded into a module, and supports activating, deactivating, and deleting the package. CICM::Package is constructed by the CICM::PackageManager::get_package_by_id and CICM::PackageImporter::complete methods and may not be instantiated independently.

3.3.4.3.1 CICM::Package Types and Constants

Type CICM::Package::PackageType

```
typedef CICM::UInt32 PackageType;
```

Supported package types.

Constant CICM::Package::C_PACKAGE_ALGORITHM

```
const CICM::Package::PackageType C_PACKAGE_ALGORITHM = 0x00006054;
```

Algorithm package type.

Constant CICM::Package::C_PACKAGE_CONFIG_PARAMS

```
const CICM::Package::PackageType C_PACKAGE_CONFIG_PARAMS = 0x00006057;
```

Configuration parameter package type.

Constant CICM::Package::C_PACKAGE_FPGA_IMAGE

```
const CICM::Package::PackageType C_PACKAGE_FPGA_IMAGE = 0x00006058;
```

FPGA image package type.

Constant CICM::Package::C_PACKAGE_POLICY_DB

```
const CICM::Package::PackageType C_PACKAGE_POLICY_DB = 0x0000605B;
```

Policy database package type.

Constant CICM::Package::C_PACKAGE_SOFTWARE

```
const CICM::Package::PackageType C_PACKAGE_SOFTWARE = 0x0000605D;
```

Software package type.

3.3.4.3.2 CICM::Package Attributes

Attribute CICM::Package::id

```
readonly attribute CICM::PackageId id;
```

Unique package identifier of this package.

3.3.4.3.3 CICM::Package Methods

Method CICM::Package::activate()

```
CICM::Status activate();
```

Activate a specific package on the module.

Remarks:

It may be necessary to reset the module before the specified package is activated in place of the currently activated package.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_PACKAGE_NOT_ACTIVATABLE, S_PACKAGE_ACTIVATED, S_PACKAGE_INVALID

Method CICM::Package::deactivate()

```
CICM::Status deactivate();
```

Deactivate a specific package on the module.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_PACKAGE_NOT_ACTIVE, S_PACKAGE_INVALID

Method CICM::Package::delete()

```
CICM::Status delete();
```

Delete a package from the module.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_PACKAGE_ACTIVATED, S_PACKAGE_INVALID

3.3.4.4 Interface CICM::PackageIterator

Interface CICM::PackageIterator

```
interface PackageIterator : CICM::Iterator {
```

CICM::PackageIterator supports retrieving a reference to each software package available on a module. CICM::PackageIterator constructs the CICM::Package interface.

3.3.4.4.1 CICM::PackageIterator Inheritance

CICM::PackageIterator inherits from: CICM::Iterator.

3.3.4.4.2 CICM::PackageIterator Methods

Method CICM::PackageIterator::get_next()

```
        CICM::Status get_next(
                out     CICM::Package package_ref
        );
```

Returns a reference to the next software package.

Remarks:

Use CICM::Iterator::has_next to determine if additional elements exist.

Parameters:

`[out]` *package_ref* Reference to next software package.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

3.3.5 MANAGING LOGS

These interfaces support the retrieval and removal of log entries.

3.3.5.1 Interface CICM::LogManager

Interface CICM::LogManager

```
interface LogManager {
```

CICM::LogManager supports retrieving or destroying an entire module log, or retrieving or deleting individual log entries. It is accessed from CICM::CryptoModule via the CICM::CryptoModule::log_manager attribute. CICM::LogManager constructs the CICM::LogEntryIterator interface.

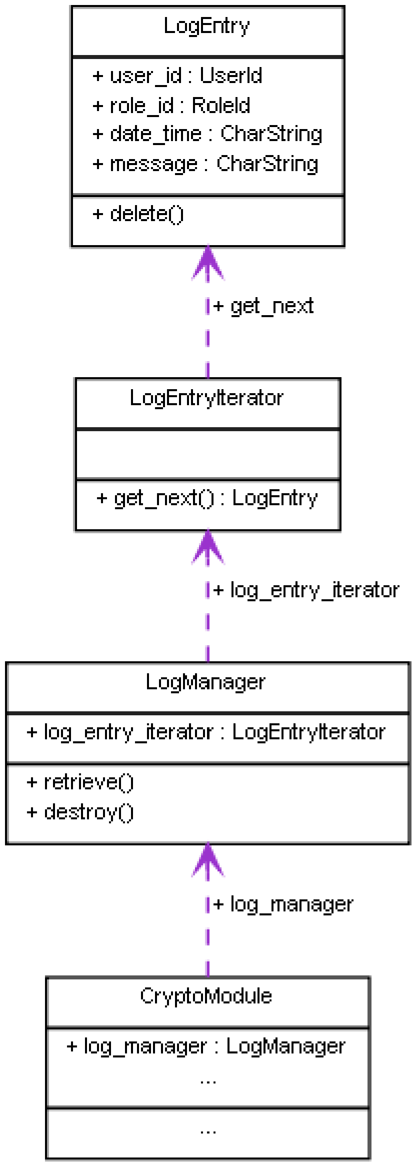


Figure 16. Interface Relationship Diagram for LogManager

Example (C++):

```
// See CICMRoot::get_module_by_id() to retrieve reference to CryptoModule.
CICM::CryptoModule device;

CICM::Status sCode;
CICM::LogManager logManager;

// Retrieve reference to the log manager.
logManager = device._get_log_manager();

// Retrieve reference to a log entry iterator.
```

```
CICM::LogEntryIterator iter;
iter = logManager._get_log_entry_iterator();

CICM::Iterator::Status status;
CICM::LogEntry entry;

// Confirm that there are log entries.
sCode = iter.hasNext(&status);

// Iterate over the log entries.
while( CICM::Iterator::C_ITERATOR_HAS_NEXT == status ) {
        sCode = iter.get_next(&entry);

        // Perform an operation on [entry].

        sCode = iter.hasNext(&status);
}

// Delete all of the log entries.
sCode = logManager.destroy();
```

3.3.5.1.1 CICM::LogManager Attributes

Attribute CICM::LogManager::log_entry_iterator

```
        readonly attribute CICM::LogEntryIterator log_entry_iterator;
```

Returns an iterator to enable a reference to each module CICM::LogEntry to be retrieved.

Remarks:

The returned iterator is set to the beginning of the iterated sequence.

3.3.5.1.2 CICM::LogManager Methods

Method CICM::LogManager::retrieve()

```
         CICM::Status retrieve(
                out CICM::Buffer log_ref
        );
```

Retrieve a reference to the entire module log.

Parameters:

`[out]` *log_ref* Reference to entire module log.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::LogManager::destroy()

```
        CICM::Status destroy();
```

Destroy all entries in the module log.
Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

3.3.5.2 Interface CICM::LogEntry

Interface CICM::LogEntry
```
interface LogEntry {
```
CICM::LogEntry serves as a reference to an individual log entry in the module log, and supports retrieving information about an individual log entry and deleting an individual log entry.

3.3.5.2.1 CICM::LogEntry Attributes

Attribute CICM::LogEntry::user_id
```
        readonly attribute CICM::UserId user_id;
```
User initiating the module action resulting in this log entry.

Attribute CICM::LogEntry::role_id
```
        readonly attribute CICM::RoleId role_id;
```
Role under which the module action resulting in this log entry was initiated.

Attribute CICM::LogEntry::message
```
        readonly attribute CICM::CharString message;
```
Log message associated with this log entry.

Attribute CICM::LogEntry::date_time
```
        readonly attribute CICM::CharString date_time;
```
Date/time of creation of this log entry.

3.3.5.2.2 CICM::LogEntry Methods

Method CICM::LogEntry::delete()
```
        CICM::Status delete();
```
Remove the current entry from the module log.
Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_LOG_ENTRY_INVALID

3.3.5.3 Interface CICM::LogEntryIterator

Interface CICM::LogEntryIterator
```
interface LogEntryIterator : CICM::Iterator {
```
CICM::LogEntryIterator supports retrieving a reference to each log entry in the module log. CICM::LogEntryIterator constructs the CICM::LogEntry interface.

3.3.5.3.1 CICM::LogEntryIterator Inheritance

CICM::LogEntryIterator inherits from: CICM::Iterator.

3.3.5.3.2 CICM::LogEntryIterator Methods

Method CICM::LogEntryIterator::get_next()
```
        CICM::Status get_next(
                out     CICM::LogEntry log_entry_ref
        );
```
Returns a reference to the next log entry.

Remarks:

Use CICM::Iterator::has_next to determine if additional elements exist.

Parameters:

`[out]` *log_entry_ref* Reference to next log entry.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

3.3.6 MANAGING TESTS

These interfaces support the initiation of module internal tests by client programs.

3.3.6.1 Interface CICM::TestManager

Interface CICM::TestManager
```
interface TestManager {
```
CICM::TestManager supports initiating client program-invoked module built-in tests. It is accessed from CICM::CryptoModule via the CICM::CryptoModule::test_manager attribute.

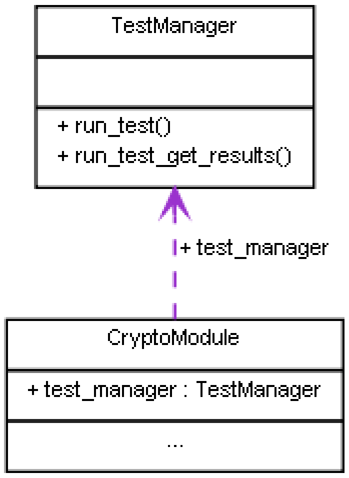


Figure 17. Interface Relationship Diagram for TestManager

3.3.6.1.1 CICM::TestManager Types and Constants

Type CICM::TestManager::Status

```
typedef CICM::UInt32 Status;
```

Test state at completion.

Constant CICM::TestManager::C_TEST_SUCCESS

```
const CICM::TestManager::Status C_TEST_SUCCESS = 0x00006062;
```

The test completed successfully.

Constant CICM::TestManager::C_TEST_FAILURE

```
const CICM::TestManager::Status C_TEST_FAILURE = 0x00006064;
```

The test failed.

3.3.6.1.2 CICM::TestManager Methods

Method CICM::TestManager::run_test()

```
    CICM::Status run_test(
            in  CICM::Buffer test_parameters,
            out CICM::TestManager::Status status
    );
```

Run module built-in tests specifying module-specific test parameters.

Remarks:

This method can only initiate tests that a client program can explicitly request (e.g., this method does not apply to a series of tests automatically initiated during a module's start-up sequence). Running built-in tests on some modules may result in an alarm if an error is encountered during the test run.

The format of the test parameters value is not defined by CICM. The Implementation Conformance Statement (see Section 4, Conformance and Extensions) must reference a standard format or define a module developer-specific format implemented by the module for this datatype.

Parameters:

`[in]`	*test_parameters*	Module-specific test parameters.
`[out]`	*status*	Status of test at completion.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_MODULE_IN_USE, S_INVALID_DATA_BUFFER, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

See also:

CICM::TestManager::run_test_get_results for the version of this method that returns test results.

Method CICM::TestManager::run_test_get_results()

```
    CICM::Status run_test_get_results(
            in  CICM::Buffer test_parameters,
            out CICM::Buffer test_results
    );
```

Run module built-in tests specifying module-specific test parameters and receiving module-specific results or data for later evaluation from the test run.

Remarks:

This method can only initiate tests that a client program can explicitly request (e.g., this method does not apply to a series of tests automatically initiated during a module's start-up sequence). Running built-in tests on some modules may result in an alarm if an error is encountered during the test run.

The formats of the test parameters and test results values are not defined by CICM. The Implementation Conformance Statement (see Section 4, Conformance and Extensions) must reference a standard format or define a module developer-specific format implemented by the module for these datatypes.

Parameters:

`[in]`	*test_parameters*	Module-specific test parameters.
`[out]`	*test_results*	Results of the test.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_MODULE_IN_USE, S_INVALID_DATA_BUFFER, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

See also:

CICM::TestManager::run_test for the version of this method that returns a simple test status value.

3.3.7 MANAGING MODULE EVENTS

In certain cases it may be necessary for a module to asynchronously notify a client program of an event. Client programs can register to receive module notifications using CICM::ModuleEventManager. This manager enables a client program to register a listener (callback) method designed to handle a specific condition. The event method prototype provided by the client program is defined in CICM::ModuleEventListener. CICM::ModuleEventListener also defines the conditions that may result in a notification, including: hardware requires attention, alarm, key expired, and health test failure.

In certain cases, a single event on a module may result in the generation of multiple notification messages. For example, CICM::ModuleEventListener::C_MODULE_ALARM may be followed by a CICM::ModuleEventListener::C_MODULE_NOT_READY_FOR_TRAFFIC.

3.3.7.1 Interface CICM::ModuleEventManager

Interface CICM::ModuleEventManager

```
interface ModuleEventManager {
```

CICM::ModuleEventManager supports registering and unregistering user-defined module event listeners (CICM::ModuleEventListener) for specific module events. It is accessed from CICM::CryptoModule via the CICM::CryptoModule::event_manager attribute.

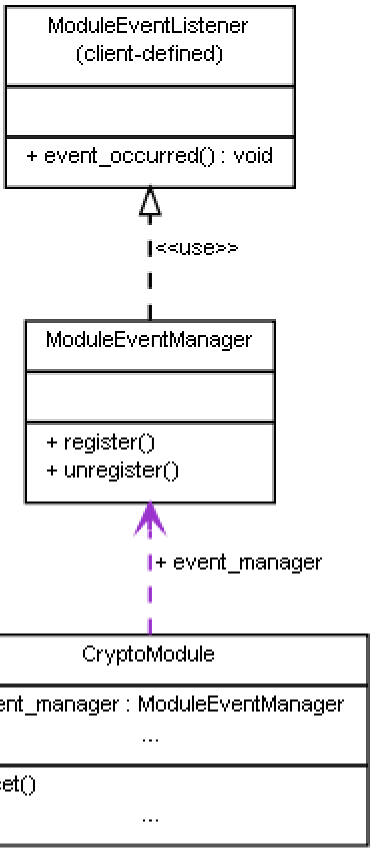


Figure 18. Interface Relationship Diagram for ModuleEventManagers

3.3.7.1.1 CICM::ModuleEventManager Methods

Method CICM::ModuleEventManager::register()

```
        CICM::Status register(
                in  CICM::ModuleEventListener::ModuleEvent event,
                in  CICM::ModuleEventListener listener
        );
```

Registers the listener for a specific module event.

Remarks:

The provided listener applies only to the client program from which the registration is initiated.

Parameters:

`[in]` *event* Event for which this listener is being registered.
`[in]` *listener* Listener that will receive a notification about the specified event.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_EVENT_REGISTERED, S_EVENT_NOT_SUPPORTED

Method CICM::ModuleEventManager::unregister()

```
        CICM::Status unregister(
                in  CICM::ModuleEventListener::ModuleEvent event
        );
```

Unregisters the listener associated with the specified event.

Remarks:

The listener associated with the specified event is only unregistered from the client program from which this method is called.

Parameters:

`[in]` *event* Event that will no longer have a listener associated with it.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_EVENT_NOT_REGISTERED

3.3.7.2 Interface CICM::ModuleEventListener

Interface CICM::ModuleEventListener

```
interface ModuleEventListener {
```

CICM::ModuleEventListener is unlike other CICM interfaces in that the interface is implemented by the developer of the client program to service a specific module event and is then registered via the CICM::ModuleEventManager.

3.3.7.2.1 CICM::ModuleEventListener Types and Constants

Type CICM::ModuleEventListener::ModuleEvent

```
typedef CICM::UInt32 ModuleEvent;
```

Events for which a ModuleEventListener can be notified.

Constant CICM::ModuleEventListener::C_MODULE_ACCESS_TOKEN_INSERTED

```
const CICM::ModuleEventListener::ModuleEvent
C_MODULE_ACCESS_TOKEN_INSERTED = 0x00002001;
```

Access token has been inserted.

Constant CICM::ModuleEventListener::C_MODULE_ACCESS_TOKEN_REMOVED

```
const CICM::ModuleEventListener::ModuleEvent
C_MODULE_ACCESS_TOKEN_REMOVED = 0x00002002;
```

Access token has been removed.

Constant CICM::ModuleEventListener::C_MODULE_ALARM

```
const CICM::ModuleEventListener::ModuleEvent C_MODULE_ALARM =
0x00002004;
```

Module has entered an alarm state.

Constant CICM::ModuleEventListener::C_MODULE_FAILURE

```
const CICM::ModuleEventListener::ModuleEvent C_MODULE_FAILURE =
0x00002007;
```

Non-critical module failure detected.

Constant CICM::ModuleEventListener::C_MODULE_INSUFFICIENT_ENTROPY

```
const CICM::ModuleEventListener::ModuleEvent
C_MODULE_INSUFFICIENT_ENTROPY = 0x00002008;
```

Insufficient entropy available to a cryptographic operation that requires it.

Constant CICM::ModuleEventListener::C_MODULE_KEY_EXPIRED_HARD

```
const CICM::ModuleEventListener::ModuleEvent C_MODULE_KEY_EXPIRED_HARD
= 0x0000200B;
```

Specific key has expired; the module can optionally include identifying information about the specific key that expired in the event_data buffer that is provided with the event itself.

Constant CICM::ModuleEventListener::C_MODULE_KEY_EXPIRED_SOFT

```
const CICM::ModuleEventListener::ModuleEvent C_MODULE_KEY_EXPIRED_SOFT
= 0x0000200D;
```

Specific key is within some system-defined delta of hard expiration; the module can optionally include identifying information about the specific key that is about to expire in the event_data buffer that is provided with the event itself.

Constant CICM::ModuleEventListener::C_MODULE_KEY_FILL_COMPLETE

```
const CICM::ModuleEventListener::ModuleEvent
C_MODULE_KEY_FILL_COMPLETE = 0x0000200E;
```

Key fill is complete.

Constant CICM::ModuleEventListener::C_MODULE_KEY_FILL_CONNECTED

```
const CICM::ModuleEventListener::ModuleEvent
C_MODULE_KEY_FILL_CONNECTED = 0x00002010;
```

Key fill device has been connected.

Constant CICM::ModuleEventListener::C_MODULE_KEY_FILL_INITIATED

```
const CICM::ModuleEventListener::ModuleEvent
C_MODULE_KEY_FILL_INITIATED = 0x00002013;
```

Key fill has been initiated.

Constant CICM::ModuleEventListener::C_MODULE_KEY_MEMORY

```
const CICM::ModuleEventListener::ModuleEvent C_MODULE_KEY_MEMORY =
0x00002015;
```

Out of internal key memory condition.

Constant CICM::ModuleEventListener::C_MODULE_KEY_PROTO_MESSAGE

```
const CICM::ModuleEventListener::ModuleEvent
C_MODULE_KEY_PROTO_MESSAGE = 0x00002016;
```

Key protocol message is available; see the Key Protocol Management documentation for additional information.

Constant CICM::ModuleEventListener::C_MODULE_LOG_FULL

```
const CICM::ModuleEventListener::ModuleEvent C_MODULE_LOG_FULL =
0x00002019;
```

Module log is full.

Constant CICM::ModuleEventListener::C_MODULE_LOG_NEAR_FULL

```
const CICM::ModuleEventListener::ModuleEvent C_MODULE_LOG_NEAR_FULL =
0x0000201A;
```

Module log is nearly full.

Constant CICM::ModuleEventListener::C_MODULE_LOGIN_FAILURE

```
const CICM::ModuleEventListener::ModuleEvent C_MODULE_LOGIN_FAILURE =
0x0000201C;
```

Attempted login failed.

Constant CICM::ModuleEventListener::C_MODULE_NOT_READY_FOR_TRAFFIC

```
const CICM::ModuleEventListener::ModuleEvent
C_MODULE_NOT_READY_FOR_TRAFFIC = 0x0000201F;
```

Module is not able to process traffic.

Constant CICM::ModuleEventListener::C_MODULE_POWER_MGMT_ENTER

```
const CICM::ModuleEventListener::ModuleEvent C_MODULE_POWER_MGMT_ENTER
= 0x00002020;
```

Module has entered power management state.

Constant CICM::ModuleEventListener::C_MODULE_POWER_MGMT_EXIT

```
const CICM::ModuleEventListener::ModuleEvent C_MODULE_POWER_MGMT_EXIT
= 0x00002023;
```

Module has exited power management state.

Constant CICM::ModuleEventListener::C_MODULE_POWER_OFF

```
const CICM::ModuleEventListener::ModuleEvent C_MODULE_POWER_OFF =
0x00002025;
```

Change in module power state to OFF detected.

Constant CICM::ModuleEventListener::C_MODULE_POWER_OFF_FAILURE

```
const CICM::ModuleEventListener::ModuleEvent
C_MODULE_POWER_OFF_FAILURE = 0x00002026;
```

Disorderly change in module power state to OFF detected.

Constant CICM::ModuleEventListener::C_MODULE_POWER_ON

```
const CICM::ModuleEventListener::ModuleEvent C_MODULE_POWER_ON =
0x00002029;
```

Change in module power state to ON detected.

Constant CICM::ModuleEventListener::C_MODULE_READY_FOR_TRAFFIC

```
const CICM::ModuleEventListener::ModuleEvent
C_MODULE_READY_FOR_TRAFFIC = 0x0000202A;
```

Module is ready to process traffic.

Constant CICM::ModuleEventListener::C_MODULE_REKEY_REQUEST

```
const CICM::ModuleEventListener::ModuleEvent C_MODULE_REKEY_REQUEST =
0x0000202C;
```

Rekey of a specific key is required; the module can optionally include identifying information about the specific key to be rekeyed in the event_data buffer that is provided with the event itself.

Constant CICM::ModuleEventListener::C_MODULE_TEST_FAILURE

```
const CICM::ModuleEventListener::ModuleEvent C_MODULE_TEST_FAILURE =
0x0000202F;
```

Module internal test has failed; the module can optionally include identifying information about the specific test that failed in the event_data buffer that is provided with the event itself.

Constant CICM::ModuleEventListener::C_MODULE_ZEROIZED

```
const CICM::ModuleEventListener::ModuleEvent C_MODULE_ZEROIZED =
0x00002031;
```

Module has been zeroized.

3.3.7.2.2 CICM::ModuleEventListener Methods

Method CICM::ModuleEventListener::event_occurred()

```
    void event_occurred(
                in  CICM::ModuleEventListener::ModuleEvent event,
                in  CICM::Buffer event_data
        );
```

Method implemented by client program that is called by the host runtime system to notify that a specific module event has occurred.

Remarks:

An opaque data field with additional information about the event in a module-specific format may optionally be provided with the event itself. This field may be of length zero.

The format of the event data value is not defined in this specification. The Implementation Conformance Statement (see Section 4, Conformance and Extensions) must reference a standard format or define a module-specific format for this datatype.

Note:

Because this method is called by the runtime system and not a client program, it does not return a status value.

Parameters:

`[in]` *event*	Event that occurred.
`[in]` *event_data*	Opaque data associated with the event (e.g., specific test that failed, key that will expire).

3.4 KEY MANAGEMENT

3.4.1 GENERAL KEY CONCEPTS

3.4.1.1 Interface CICM::Key

Interface CICM::Key

```
interface Key {
```

Interface from which symmetric and asymmetric key interfaces inherit.

3.4.1.1.1 Creating and Establishing Keys

This specification provides facilities to convey key material into a cryptographic module via one of several methods, including importing key into the module through the API or a key fill interface, or generating or deriving key on the module.

Key material may also be created through the use of a key establishment protocol between the module and a key infrastructure component or another module. Such a protocol is initiated between two or more parties to establish a secret key over a communications channel. The specification supports conveying generic protocol messages to and from a cryptographic module to effect the establishment of this secret key. See the CICM::KeyProtocolSender and CICM::KeyProtocolReceiver interfaces for additional information.

Key metadata may be retrieved and set for individual keys. Metadata elements include the key identifier, alias, and classification. Keys imported via a fill device that are untagged may require certain metadata to be applied after the conclusion of the load, for example.

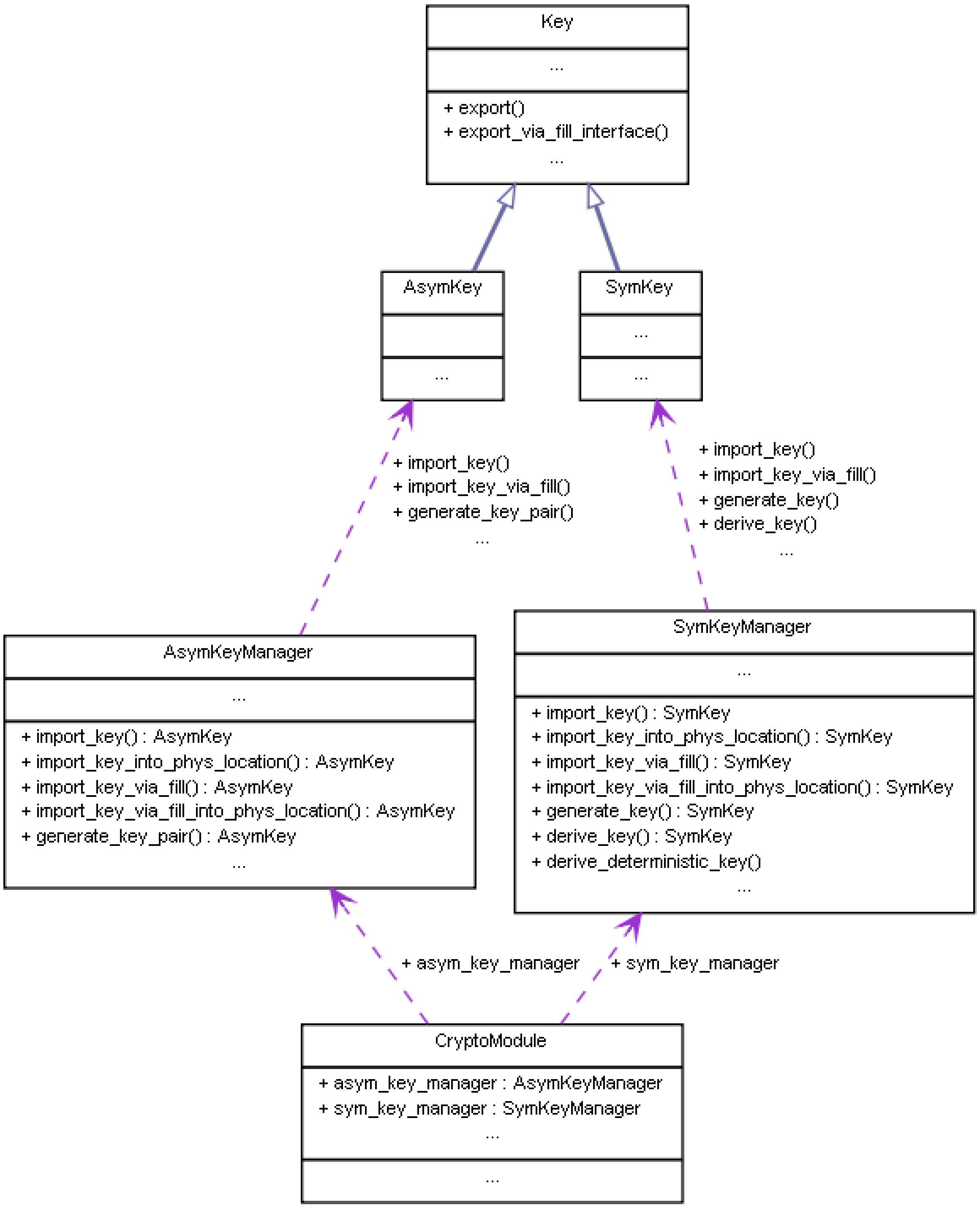

Key
...
+ export()
+ export_via_fill_interface()
...
AsymKey
...
SymKey
...
...
+ import_key()
+ import_key_via_fill()
+ generate_key_pair()
...
+ import_key()
+ import_key_via_fill()
+ generate_key()
+ derive_key()
...
AsymKeyManager
...
+ import_key() : AsymKey
+ import_key_into_phys_location() : AsymKey
+ import_key_via_fill() : AsymKey
+ import_key_via_fill_into_phys_location() : AsymKey
+ generate_key_pair() : AsymKey
...
SymKeyManager
...
+ import_key() : SymKey
+ import_key_into_phys_location() : SymKey
+ import_key_via_fill() : SymKey
+ import_key_via_fill_into_phys_location() : SymKey
+ generate_key() : SymKey
+ derive_key() : SymKey
+ derive_deterministic_key()
...
+ asym_key_manager
+ sym_key_manager
CryptoModule
+ asym_key_manager : AsymKeyManager
+ sym_key_manager : SymKeyManager
...
...

Figure 19. Interface Inheritance Diagram for AsymKey and SymKey, Depicting Key Creation Methods

3.4.1.1.2 Exporting Keys

Key material may also be exported out of a cryptographic module through the use of the key export functionality to enable transfer to another entity or for storage within the host system.

3.4.1.1.3 Destroying Keys

This specification provides the ability to permanently and irretrievably destroy key material. These capabilities apply to keys managed by the module, whether stored internal to the module or stored externally.

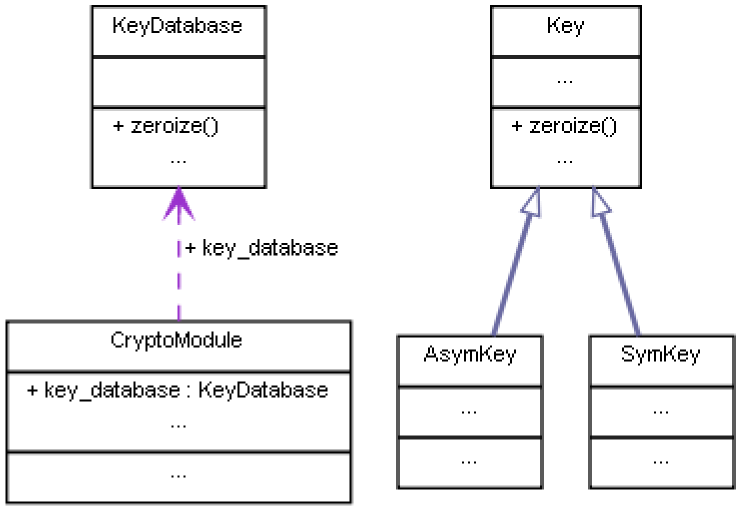


Figure 20. Interface Relationship and Inheritance Diagram Depicting Key Zeroization

3.4.1.1.4 Locating Keys

A key is typically designated by a global identifier defined by the external key management system from which the key originated. Alternatively, a key may be designated by a numeric value representing the physical storage location of the key within the module. Key location methods enable a key object representing the key specified by a supplied identifier to be retrieved by the caller.

Note that the format of the key identifier is not defined by CICM. The Implementation Conformance Statement (see Section 4, Conformance and Extensions) must reference a standard format or define a module developer-specific format implemented by the module for this datatype.

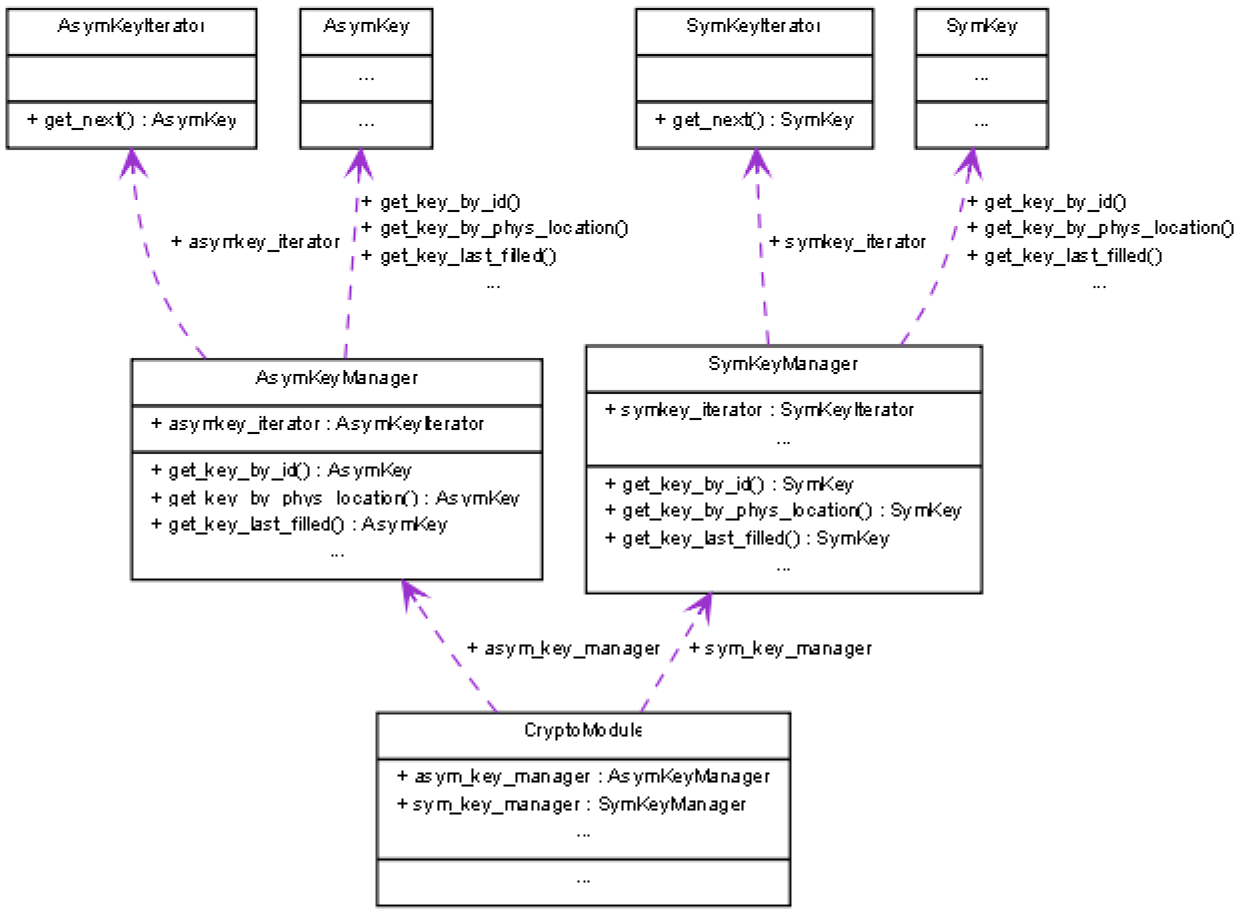

AsymKeyIterator
+ get_next() : AsymKey
AsymKey
...
...
SymKeyIterator
+ get_next() : SymKey
SymKey
...
...
+ asymkey_iterator
+ get_key_by_id()
+ get_key_by_phys_location()
+ get_key_last_filled()
...
+ symkey_iterator
+ get_key_by_id()
+ get_key_by_phys_location()
+ get_key_last_filled()
...
AsymKeyManager
+ asymkey_iterator : AsymKeyIterator
+ get_key_by_id() : AsymKey
+ get_key_by_phys_location() : AsymKey
+ get_key_last_filled() : AsymKey
...
SymKeyManager
+ symkey_iterator : SymKeyIterator
...
+ get_key_by_id() : SymKey
+ get_key_by_phys_location() : SymKey
+ get_key_last_filled() : SymKey
...
+ asym_key_manager
+ sym_key_manager
CryptoModule
+ asym_key_manager : AsymKeyManager
+ sym_key_manager : SymKeyManager
...
...

Figure 21. Interface Relationship Diagram for Key Managers

3.4.1.1.5 Protecting Keys

These methods enable the encryption and decryption of key material to support transferring keys between modules or other entities (including storage of key material external to the module).

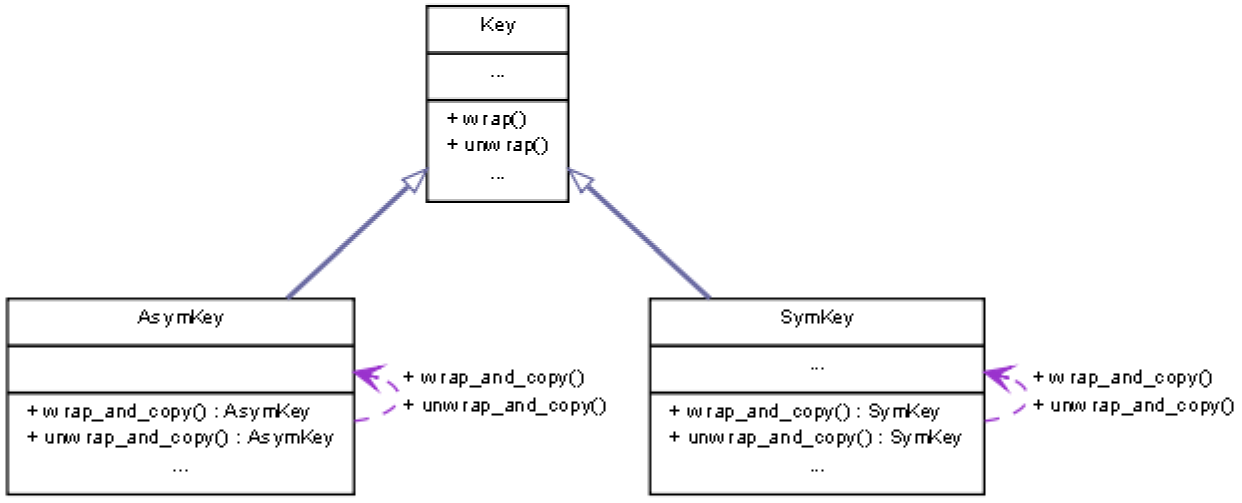


Figure 22. Interface Inheritance Diagram for AsymKey and SymKey, Depicting Key Protection Methods

3.4.1.1.6 CICM::Key Types and Constants

Type CICM::Key::State

```
typedef CICM::UInt32 State;
```

Indicates whether or not the key is valid.

Constant CICM::Key::C_KEY_INVALID

```
const CICM::Key::State C_KEY_INVALID = 0x00006010;
```

Key is invalid.

Constant CICM::Key::C_KEY_VALID_WRAPPED

```
const CICM::Key::State C_KEY_VALID_WRAPPED = 0x00006013;
```

Key is valid and in wrapped form.

Constant CICM::Key::C_KEY_VALID_UNWRAPPED

```
const CICM::Key::State C_KEY_VALID_UNWRAPPED = 0x00006015;
```

Key is valid and in unwrapped form.

Type CICM::Key::UsageStatus

```
typedef CICM::UInt32 UsageStatus;
```

Indicates whether a key usage is allowed or forbidden.

Constant CICM::Key::C_KEY_USAGE_ALLOWED

```
const CICM::Key::UsageStatus C_KEY_USAGE_ALLOWED = 0x00006016;
```

Key is valid for this usage.

Constant CICM::Key::C_KEY_USAGE_FORBIDDEN

```
const CICM::Key::UsageStatus C_KEY_USAGE_FORBIDDEN = 0x00006019;
```

Key is not valid for this usage.

3.4.1.1.7 CICM::Key Attributes

Attribute CICM::Key::identifier

```
    attribute CICM::CharString identifier;
```

Unique global identifier for this key.

Remarks:

This identifier is the string representation of a key management authority-specific global key identifier. The identifier may be composed of multiple components; the representation of these components within the CICM identifier string is not defined in this specification. The Implementation Conformance Statement (see Section 4, Conformance and Extensions) must reference a standard format or define a module developer-specific format implemented by the module for this identifier.

Warning:

This attribute SHALL be unimplemented if CICM::Key::location is implemented.

Attribute CICM::Key::location

```
        attribute CICM::UInt32 location;
```

Module-specific physical storage location for this key.

Warning:

This attribute should only be used when a module utilizes a key storage model that requires keys of particular types to be stored in specific locations or in situations where keys are tied to module capabilities based upon physical location of the key. Because of the architectural knowledge of a module required, its use is highly discouraged because it virtually guarantees that code using it will be incompatible with other dissimilar modules. This attribute SHALL be unimplemented if CICM::Key::identifier is implemented.

Attribute CICM::Key::alias

```
        attribute CICM::CharString alias;
```

Key alias, to assist in distinguishing one key from another.

Attribute CICM::Key::classification

```
        attribute CICM::Classification classification;
```

Key classification level.

Attribute CICM::Key::caveat

```
        attribute CICM::CharString caveat;
```

Key caveat, a protective marking or distribution/handling instruction that may augment classification level.

Attribute CICM::Key::authority

```
        attribute CICM::CharString authority;
```

Key management authority governing generation and use of key.

Attribute CICM::Key::state

```
        readonly attribute CICM::Key::State state;
```

State of key. A key may become invalid if zeroized, for example. Note that if an attempt is made to use an invalid key, the method accepting the key reference will return with an appropriate error status.

3.4.1.1.8 CICM::Key Methods

Method CICM::Key::wrap()

```
        CICM::Status wrap(
                in  CICM::Key kek,
                in  CICM::KeyWrapAlgorithmId algorithm
        );
```

Instruct module to wrap key, destroying the original unwrapped key and replacing it with the newly wrapped key.

Parameters:

`[in]` *kek*	Reference to key encryption key.
`[in]` *algorithm*	Key wrap algorithm/mode used to wrap key.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_CLASSIFICATION, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_INCOMPATIBLE, S_KEY_CLASSIFICATION, S_KEY_WRAPPED, S_KEY_NOT_WRAPPABLE, S_KEY_WRAPPED_EXISTS, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::Key::unwrap()

```
        CICM::Status unwrap(
                in  CICM::Key kek,
                in  CICM::KeyWrapAlgorithmId algorithm
        );
```

Instruct module to unwrap key, destroying the original wrapped key and replacing it with the newly unwrapped key.

Parameters:

`[in]` *kek*	Reference to key decryption key.
`[in]` *algorithm*	Key wrap algorithm/mode used to unwrap key.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_CLASSIFICATION, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_INCOMPATIBLE, S_KEY_CLASSIFICATION, S_KEY_UNWRAPPED_EXISTS, S_KEY_MALFORMED, S_KEY_METADATA_MALFORMED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::Key::export()

```
        CICM::Status export(
                out CICM::Buffer key_material
        );
```

Export key material from a cryptographic module as an opaque binary object.

Remarks:

The key must have been previously wrapped, if required. See CICM::Key::wrap.

Parameters:

`[out]` *key_material*	Binary version of the key.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_WRAPPED, S_KEY_NOT_WRAPPED, S_KEY_NOT_EXPORTABLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::Key::export_via_fill_interface()

```
        CICM::Status export_via_fill_interface(
                in  CICM::LocalPort fill_port
        );
```

Export key material from a cryptographic module via a port representing a key fill interface.

Remarks:

The key must have been previously wrapped, if required. See CICM::Key::wrap.

Parameters:

`[in]` *fill_port* The local port from which the key will emanate.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_WRAPPED, S_KEY_NOT_WRAPPED, S_KEY_NOT_EXPORTABLE, S_KEY_FILL_DEVICE_NOT_CONNECTED, S_KEY_FILL_NOT_INITIATED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::Key::zeroize()

```
        CICM::Status zeroize();
```

Zeroize the selected key.

Remarks:

Future cryptographic operations depending upon this key SHALL fail with a CICM::S_KEY_INVALID status.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

3.4.1.2 Interface CICM::KeyDatabase

Interface CICM::KeyDatabase

```
interface KeyDatabase {
```

CICM::KeyDatabase supports zeroizing keys and reencrypting a module key database. It is accessed from CICM::CryptoModule via the CICM::CryptoModule::key_database attribute.

3.4.1.2.1 CICM::KeyDatabase Methods

Method CICM::KeyDatabase::zeroize()

```
        CICM::Status zeroize();
```

Zeroize all key material on the module.

Remarks:

This method renders all instantiated key objects invalid. Future cryptographic operations depending upon zeroized keys may fail with a CICM::S_KEY_INVALID status.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::KeyDatabase::reencrypt()

```
      CICM::Status reencrypt();
```

Re-encrypt the module key database.

Remarks:

This method uses a module-managed key to protect the key database and only applies to keys managed by the module, whether stored internal to the module or stored externally. Keys stored external to the module and not directly managed by the module must utilize the key wrap methods to protect key material.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_INSUFFICIENT_ENTROPY

3.4.2 ASYMMETRIC KEYS

3.4.2.1 Interface CICM::AsymKeyManager

Interface CICM::AsymKeyManager

```
interface AsymKeyManager {
```

CICM::AsymKeyManager supports retrieving, importing, and generating asymmetric keysets. It is accessed from CICM::CryptoModule via the CICM::CryptoModule::asym_key_manager attribute. CICM::AsymKeyManager constructs the CICM::AsymKeyIterator and CICM::AsymKey interfaces.

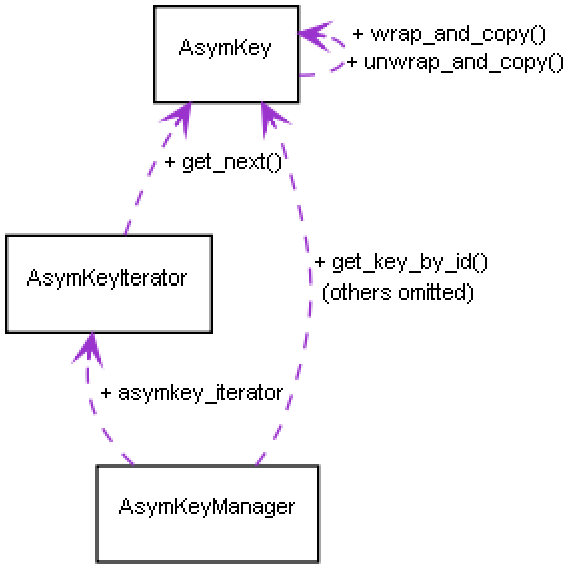


Figure 23. Interface Relationship Diagram for AsymKeyManager

3.4.2.1.1 CICM::AsymKeyManager Attributes

Attribute CICM::AsymKeyManager::asymkey_iterator

```
        readonly attribute CICM::AsymKeyIterator asymkey_iterator;
```

Returns an iterator to enable a reference to each asymmetric keyset in the module key database to be retrieved.

Remarks:

The returned iterator is set to the beginning of the iterated sequence.

3.4.2.1.2 CICM::AsymKeyManager Methods

Method CICM::AsymKeyManager::get_key_by_id()

```
        CICM::Status get_key_by_id(
                in  CICM::KeyId key_id,
                out CICM::AsymKey key_ref
        );
```

Retrieves a reference to the asymmetric keyset corresponding to the specified infrastructure-specific identifier.

Warning:

This method SHALL be unimplemented if CICM::AsymKeyManager::get_key_by_phys_location is implemented.

Parameters:

`[in]`	*key_id*	Unique identifier of the keyset to be retrieved.
`[out]`	*key_ref*	Reference to keyset corresponding to key identifier.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_NOT_AVAILABLE, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_INVALID_ID, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::AsymKeyManager::get_key_by_phys_location()

```
        CICM::Status get_key_by_phys_location(
                in  CICM::UInt32 phys_location,
                out CICM::AsymKey key_ref
        );
```

Retrieves a reference to the asymmetric keyset corresponding to the specified module physical storage location.

Warning:

This call should only be used when a module utilizes a key storage model that requires keys of particular types to be stored in specific locations or in situations where keys are tied to module capabilities based upon physical location of the key. Because of the architectural knowledge of a module required, its use is highly discouraged because it virtually guarantees that code using it will be incompatible with other dissimilar modules.

This method SHALL be unimplemented if CICM::AsymKeyManager::get_key_by_id is implemented.

Parameters:

`[in]` *phys_location* Physical location of the key to be retrieved.
`[out]` *key_ref* Reference to key corresponding to physical location.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_NOT_AVAILABLE, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_PHYSICAL_LOC, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::AsymKeyManager::get_key_last_filled()

```
    CICM::Status get_key_last_filled(
            out CICM::AsymKey key_ref
    );
```

Retrieves a reference to the asymmetric keyset corresponding to the keyset most recently filled via a key fill device.

Remarks:

A client program may need to initiate this action if key material imported into a module does not contain the corresponding key metadata. This method allows a reference to the last keyset filled over the key fill interface to be referenced to enable metadata to be applied directly to the resulting keyset. The error status CICM::S_NOT_AVAILABLE is returned if no key is filled.

Parameters:

`[out]` *key_ref* Object representing last filled key.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_NOT_AVAILABLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

See also:

SymKeyManager::get_key_last_filled for the symmetric version of this method.

Method CICM::AsymKeyManager::import_key()

```
    CICM::Status import_key(
            in  CICM::Buffer key_material,
            out CICM::AsymKey key_ref
    );
```

Import asymmetric keysets into a cryptographic module.

Remarks:

Keysets may be imported into a module as wrapped entities, necessitating the use of CICM::Key::unwrap to decrypt the keyset prior to use.

Keyset attributes may optionally be set to create or supplement key metadata.

The format of the key material value is not defined by CICM. The Implementation Conformance Statement (see Section 4, Conformance and Extensions) must reference a standard format or define a module developer-specific format implemented by the module for this datatype.

Warning:

This method SHALL be unimplemented if CICM::AsymKeyManager::import_key_into_phys_location is implemented.

Parameters:

`[in]`	*key_material*	Key material to be imported into the module.
`[out]`	*key_ref*	Reference to newly imported key.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_MALFORMED, S_KEY_METADATA_MALFORMED, S_KEY_TRUST_ANCHOR, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

See also:

CICM::SymKeyManager::import_key for the symmetric version of this method.

CICM::AsymKeyManager::import_key_via_fill for the key fill version of this method.

CICM::AsymKeyManager::import_key_into_phys_location for the version of this method that imports key into a specific module key location.

CICM::AsymKeyManager::import_key_via_fill_into_phys_location for the version of this method that fills key into a specific module key location.

Method CICM::AsymKeyManager::import_key_into_phys_location()

```
    CICM::Status import_key_into_phys_location(
            in  CICM::Buffer key_material,
            in  CICM::UInt32 phys_location,
            out CICM::SymKey key_ref
    );
```

Import key material into a specific physical key location in a cryptographic module.

Remarks:

Keysets may be imported into a module as wrapped entities, necessitating the use of CICM::Key::unwrap to decrypt the keyset prior to use.

Keyset object attributes may optionally be set to create or supplement keyset metadata.

The format of the key material value is not defined by CICM. The Implementation Conformance Statement (see Section 4, Conformance and Extensions) must reference a standard format or define a module developer-specific format implemented by the module for this datatype.

Warning:

This call should only be used when a module utilizes a key storage model that requires keys of particular types to be stored in specific locations or in situations where keys are tied to module capabilities based upon physical location of the key. Because of the architectural knowledge of a module required, its use is highly discouraged because it virtually guarantees that code using it will be incompatible with other dissimilar modules.

This method SHALL be unimplemented if CICM::AsymKeyManager::import_key is implemented.

Parameters:

`[in]`	*key_material*	Key material to be imported into the module.
`[in]`	*phys_location*	Physical location into which to import keyset.
`[out]`	*key_ref*	Reference to newly imported keyset.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_PHYSICAL_LOC, S_KEY_MALFORMED, S_KEY_METADATA_MALFORMED, S_KEY_TRUST_ANCHOR, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::AsymKeyManager::import_key_via_fill()

```
    CICM::Status import_key_via_fill(
            in  CICM::LocalPort fill_port,
            out CICM::AsymKey key_ref
    );
```

Initiate the import of key material via a key fill interface.

Remarks:

In some cases, a key fill device can initiate and effect the filling of key into a module completely independent of the host and thus any API control. In such cases, the host will utilize the CICM::AsymKeyManager::get_key_by_id, CICM::AsymKeyManager::get_key_by_phys_location, or CICM::AsymKeyManager::get_key_last_filled methods after the key fill has completed to enable future reference to the keyset.

Keysets may be imported into a module as wrapped entities, necessitating the use of CICM::Key::unwrap to decrypt the key prior to use.

Keyset attributes may optionally be set to create or supplement keyset metadata.

Warning:

This method SHALL be unimplemented if CICM::AsymKeyManager::import_key_via_fill_into_phys_location is implemented.

Parameters:

`[in]`	*fill_port*	Fill port on which to initiate import.
`[out]`	*key_ref*	Reference to newly imported keyset.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_MALFORMED, S_KEY_METADATA_MALFORMED, S_KEY_FILL_DEVICE_NOT_CONNECTED, S_KEY_FILL_NOT_INITIATED, S_KEY_TRUST_ANCHOR, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::AsymKeyManager::import_key_via_fill_into_phys_location()

```
        CICM::Status import_key_via_fill_into_phys_location(
                in  CICM::LocalPort fill_port,
                in  CICM::UInt32 phys_location,
                out CICM::AsymKey key_ref
        );
```

Initiate the import of key material into a specific key physical location via a key fill interface.

Remarks:

In some cases, a key fill device can initiate and effect the filling of key into a module completely independent of the host and thus any API control. In such cases, the host will utilize the CICM::AsymKeyManager::get_key_by_id, CICM::AsymKeyManager::get_key_by_phys_location, or CICM::AsymKeyManager::get_key_last_filled methods after the key fill has completed to enable future reference to the keyset.

Keysets may be imported into a module as wrapped entities, necessitating the use of CICM::Key::unwrap to decrypt the keyset prior to use.

Keyset attributes may optionally be set to create or supplement keyset metadata.

Warning:

This call should only be used when a module utilizes a key storage model that requires keys of particular types to be stored in specific locations or in situations where keys are tied to module capabilities based upon physical location of the key. Because of the architectural knowledge of a module required, its use is highly discouraged because it virtually guarantees that code using it will be incompatible with other dissimilar modules.

This method SHALL be unimplemented if CICM::AsymKeyManager::import_key_via_fill is implemented.

Parameters:

`[in]`	*fill_port*	Port of the key fill interface.
`[in]`	*phys_location*	Physical location into which to import keyset.
`[out]`	*key_ref*	Reference to newly imported keyset.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_PHYSICAL_LOC, S_KEY_MALFORMED, S_KEY_METADATA_MALFORMED, S_KEY_FILL_DEVICE_NOT_CONNECTED, S_KEY_FILL_NOT_INITIATED, S_KEY_TRUST_ANCHOR, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::AsymKeyManager::generate_key_pair()

```
        CICM::Status generate_key_pair(
                in  CICM::AsymEncrAlgorithmId algorithm,
                out CICM::AsymKey key_ref
        );
```

Generate an asymmetric key pair compatible with the characteristics of the specified algorithm.

Parameters:

`[in]` *algorithm* Desired algorithm of resulting asymmetric key pair.
`[out]` *key_ref* Reference to newly generated key pair.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_INSUFFICIENT_ENTROPY

3.4.2.2 Interface CICM::AsymKey

Interface CICM::AsymKey

```
interface AsymKey : CICM::Key {
```

CICM::AsymKey serves as an abstraction for an asymmetric keyset, which may comprise an asymmetric key pair, the public and private key components of a keypair, the digital certificate corresponding to the keyset public key, one or more verification certificates in the certificate chain of trust, and related public domain parameters; and supports operations on asymmetric keys, including wrapping and unwrapping.

3.4.2.2.1 CICM::AsymKey Inheritance

CICM::AsymKey inherits from: CICM::Key.

3.4.2.2.2 CICM::AsymKey Types and Constants

Type CICM::AsymKey::Usage

```
typedef CICM::UInt32 Usage;
```

Asymmetric key usage types.

Constant CICM::AsymKey::C_USAGE_ASYM_DATA_ENCIPHERMENT

```
const CICM::AsymKey::Usage C_USAGE_ASYM_DATA_ENCIPHERMENT =
0x00006001;
```

Key intended for enciphering data.

Constant CICM::AsymKey::C_USAGE_ASYM_KEY_ENCIPHERMENT

```
const CICM::AsymKey::Usage C_USAGE_ASYM_KEY_ENCIPHERMENT = 0x00006002;
```

Key intended for enciphering other keys.

Constant CICM::AsymKey::C_USAGE_CERT_SIGN

```
const CICM::AsymKey::Usage C_USAGE_CERT_SIGN = 0x00006004;
```

Key intended for signing/verifying digital certificates.

Constant CICM::AsymKey::C_USAGE_CRL_SIGN

```
const CICM::AsymKey::Usage C_USAGE_CRL_SIGN = 0x00006007;
```

Key intended for signing/verifying certificate revocation lists.

Constant CICM::AsymKey::C_USAGE_DIGITAL_SIGNATURE

```
        const CICM::AsymKey::Usage C_USAGE_DIGITAL_SIGNATURE = 0x00006008;
```
Key intended for producing digital signatures.

Constant CICM::AsymKey::C_USAGE_INFRA_KEY_AGREEMENT
```
        const CICM::AsymKey::Usage C_USAGE_INFRA_KEY_AGREEMENT = 0x0000600B;
```
Key intended for participating in an infrastructure key agreement protocol.

Constant CICM::AsymKey::C_USAGE_P2P_KEY_AGREEMENT
```
        const CICM::AsymKey::Usage C_USAGE_P2P_KEY_AGREEMENT = 0x0000600D;
```
Key intended for participating in a peer-to-peer key agreement protocol.

Constant CICM::AsymKey::C_USAGE_SEED
```
        const CICM::AsymKey::Usage C_USAGE_SEED = 0x0000600E;
```
Key intended to serve as seed material.

3.4.2.2.3 CICM::AsymKey Methods

Method CICM::AsymKey::wrap_and_copy()
```
        CICM::Status wrap_and_copy(
                in  CICM::Key kek,
                in  CICM::KeyWrapAlgorithmId algorithm,
                out CICM::AsymKey wrapped_key
        );
```
Instruct module to wrap keyset, resulting in two keysets, the original unwrapped keyset and the newly wrapped keyset.

Parameters:

`[in]`	*kek*	Reference to key encryption key.
`[in]`	*algorithm*	Key wrap algorithm/mode used to wrap keyset.
`[out]`	*wrapped_key*	Reference to resulting wrapped keyset.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_CLASSIFICATION, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_INCOMPATIBLE, S_KEY_CLASSIFICATION, S_KEY_WRAPPED, S_KEY_NOT_WRAPPABLE, S_KEY_WRAPPED_EXISTS, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

See also:

CICM::SymKey::wrap_and_copy for the symmetric version of this method.

Method CICM::AsymKey::unwrap_and_copy()
```
        CICM::Status unwrap_and_copy(
                in  CICM::Key kek,
                in  CICM::KeyWrapAlgorithmId algorithm,
                out CICM::AsymKey unwrapped_key
        );
```

Instruct module to unwrap key, resulting in two keys, the original wrapped key and the newly unwrapped key.

Parameters:

`[in]`	*kek*	Reference to key decryption key.
`[in]`	*algorithm*	Key wrap algorithm/mode used to unwrap keyset.
`[out]`	*unwrapped_key*	Reference to resulting unwrapped keyset.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_CLASSIFICATION, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_INCOMPATIBLE, S_KEY_CLASSIFICATION, S_KEY_UNWRAPPED_EXISTS, S_KEY_MALFORMED, S_KEY_METADATA_MALFORMED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

See also:

CICM::SymKey::unwrap_and_copy for the symmetric version of this method.

Method CICM::AsymKey::validate_key_usage()

```
        CICM::Status validate_key_usage(
                in  CICM::AsymKey::Usage usage_type,
                out CICM::Key::UsageStatus valid
        );
```

Validate that this keyset may be used for a specific purpose.

Remarks:

In some cases, a given keyset can be used for multiple purposes.

Parameters:

`[in]`	*usage_type*	Specific purpose to validate.
`[out]`	*valid*	Indiciates whether or not the key may be used for the specified purpose.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_WRAPPED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

See also:

CICM::SymKey::validate_key_usage for the symmetric version of this method.

3.4.2.3 Interface CICM::AsymKeyIterator

Interface CICM::AsymKeyIterator

```
interface AsymKeyIterator : CICM::Iterator {
```

CICM::AsymKeyIterator supports retrieving a reference to each usable asymmetric key on a module. CICM::AsymKeyIterator constructs the CICM::AsymKey interface.

3.4.2.3.1 CICM::AsymKeyIterator Inheritance

CICM::AsymKeyIterator inherits from: CICM::Iterator.

3.4.2.3.2 CICM::AsymKeyIterator Methods

Method CICM::AsymKeyIterator::get_next()

```
        CICM::Status get_next(
                out     CICM::AsymKey asym_key_ref
        );
```

Returns a reference to the next asymmetric key.

Remarks:

Use CICM::Iterator::has_next to determine if additional elements exist.

Parameters:

`[out]` *asym_key_ref* Reference to next asymmetric key.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

3.4.3 SYMMETRIC KEYS

3.4.3.1 Interface CICM::SymKeyManager

Interface CICM::SymKeyManager

```
interface SymKeyManager {
```

CICM::SymKeyManager supports retrieving, importing, generating and deriving symmetric keys; and operating key management protocols. It is accessed from CICM::CryptoModule via the CICM::CryptoModule::sym_key_manager attribute. CICM::SymKeyManager constructs the CICM::KeyProtocolSender, CICM::KeyProtocolReceiver, CICM::SymKeyIterator, and CICM::SymKey interfaces.

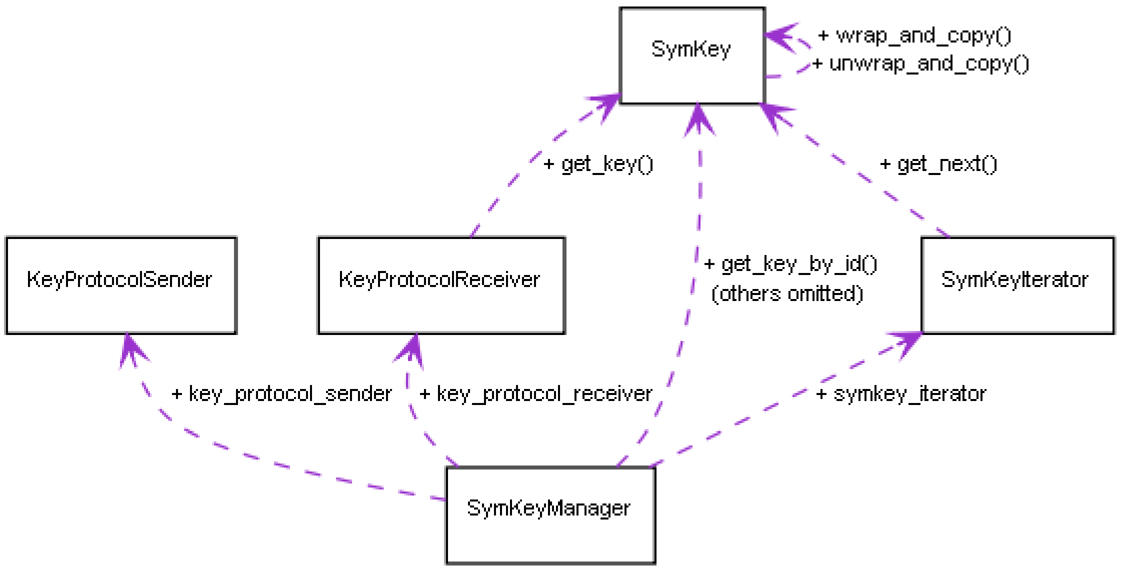


Figure 24. Interface Relationship Diagram for SymKeyManager

3.4.3.1.1 CICM::SymKeyManager Attributes

Attribute CICM::SymKeyManager::symkey_iterator

```
        readonly attribute CICM::SymKeyIterator symkey_iterator;
```

Returns an iterator to enable a reference to each symmetric key in the module key database to be retrieved.

Remarks:

The returned iterator is set to the beginning of the iterated sequence.

Attribute CICM::SymKeyManager::key_protocol_sender

```
        readonly attribute CICM::KeyProtocolSender key_protocol_sender;
```

CICM::KeyProtocolSender supports sending key management protocol-related messages into a module.

Attribute CICM::SymKeyManager::key_protocol_receiver

```
        readonly attribute CICM::KeyProtocolReceiver key_protocol_receiver;
```

CICM::KeyProtocolReceiver supports receiving key management protocol-related messages from a module. CICM::KeyProtocolReceiver constructs the CICM::SymKey interface.

3.4.3.1.2 CICM::SymKeyManager Methods

Method CICM::SymKeyManager::get_key_by_id()

```
        CICM::Status get_key_by_id(
                in  CICM::KeyId key_id,
                out CICM::SymKey key_ref
        );
```

Retrieves a reference to the symmetric key corresponding to the specified infrastructure-specific identifier.

Warning:

This method SHALL be unimplemented if CICM::SymKeyManager::get_key_by_phys_location is implemented.

Parameters:

`[in]` *key_id* Unique identifier of the key to be retrieved.
`[out]` *key_ref* Reference to key corresponding to key identifier.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_NOT_AVAILABLE, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_INVALID_ID, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::SymKeyManager::get_key_by_phys_location()

```
CICM::Status get_key_by_phys_location(
        in  CICM::UInt32 phys_location,
        out CICM::SymKey key_ref
);
```

Retrieves a reference to the symmetric key corresponding to the specified module physical storage location.

Warning:

This call should only be used when a module utilizes a key storage model that requires keys of particular types to be stored in specific locations or in situations where keys are tied to module capabilities based upon physical location of the key. Because of the architectural knowledge of a module required, its use is highly discouraged because it virtually guarantees that code using it will be incompatible with other dissimilar modules.

This method SHALL be unimplemented if CICM::SymKeyManager::get_key_by_id is implemented.

Parameters:

`[in]` *phys_location* Physical location of the key to be retrieved.
`[out]` *key_ref* Reference to key corresponding to physical location.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_NOT_AVAILABLE, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_PHYSICAL_LOC, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::SymKeyManager::get_key_last_filled()

```
CICM::Status get_key_last_filled(
        out CICM::SymKey key_ref
);
```

Retrieves a reference to the symmetric key corresponding to the key most recently filled via a key fill device.

Remarks:

A client program may need to initiate this action if key material imported into a module does not contain the corresponding key metadata. This method allows a reference to the last key filled over the key fill interface to be referenced to enable metadata to be applied directly to the resulting key. The error status CICM::S_NOT_AVAILABLE is returned if no key is filled.

Parameters:

`[out]` *key_ref* Reference to last filled key.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_NOT_AVAILABLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::SymKeyManager::import_key()

```
    CICM::Status import_key(
            in  CICM::Buffer key_material,
            out CICM::SymKey key_ref
    );
```

Import key material or seed key for pseudorandom data generation into a cryptographic module.

Remarks:

Keys may be imported into a module as wrapped entities, necessitating the use of CICM::Key::unwrap to decrypt the key prior to use.

Key attributes may optionally be set to create or supplement key metadata.

The format of the key material value is not defined by CICM. The Implementation Conformance Statement (see Section 4, Conformance and Extensions) must reference a standard format or define a module developer-specific format implemented by the module for this datatype.

Warning:

This method SHALL be unimplemented if CICM::SymKeyManager::import_key_into_phys_location is implemented.

Parameters:

`[in]`	*key_material*	Key material to be imported into the module.
`[out]`	*key_ref*	Reference to newly imported key.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_MALFORMED, S_KEY_METADATA_MALFORMED, S_KEY_TRUST_ANCHOR, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::SymKeyManager::import_key_into_phys_location()

```
    CICM::Status import_key_into_phys_location(
            in  CICM::Buffer key_material,
            in  CICM::UInt32 phys_location,
            out CICM::SymKey key_ref
    );
```

Import key material or seed key for pseudorandom data generation into a specific physical key location in a cryptographic module.

Remarks:

Keys may be imported into a module as wrapped entities, necessitating the use of CICM::Key::unwrap to decrypt the key prior to use.

Key attributes may optionally be set to create or supplement key metadata.

The format of the key material value is not defined by CICM. The Implementation Conformance Statement (see Section 4, Conformance and Extensions) must reference a standard format or define a module developer-specific format implemented by the module for this datatype.

Warning:

This call should only be used when a module utilizes a key storage model that requires keys of particular types to be stored in specific locations or in situations where keys are tied to module capabilities based upon physical location of the key. Because of the architectural knowledge of a module required, its use is highly discouraged because it virtually guarantees that code using it will be incompatible with other dissimilar modules.

This method SHALL be unimplemented if CICM::SymKeyManager::import_key is implemented.

Parameters:

`[in]`	*key_material*	Key material to be imported into the module.
`[in]`	*phys_location*	Physical location into which to import key.
`[out]`	*key_ref*	Reference to newly imported key.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_PHYSICAL_LOC, S_KEY_MALFORMED, S_KEY_METADATA_MALFORMED, S_KEY_TRUST_ANCHOR, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::SymKeyManager::import_key_via_fill()

```
    CICM::Status import_key_via_fill(
            in  CICM::LocalPort fill_port,
            out CICM::SymKey key_ref
    );
```

Initiate the import of key material via a key fill interface.

Remarks:

In some cases, a key fill device can initiate and effect the filling of key into a module completely independent of the host and thus any API control. In such cases, the host will utilize the CICM::SymKeyManager::get_key_by_id, CICM::SymKeyManager::get_key_by_phys_location, or CICM::SymKeyManager::get_key_last_filled methods after the key fill has completed to enable future reference to the key.

Keys may be imported into a module as wrapped entities, necessitating the use of CICM::Key::unwrap to decrypt the key prior to use.

Key attributes may optionally be set to create or supplement key metadata.

Warning:

This method SHALL be unimplemented if CICM::SymKeyManager::import_key_via_fill_into_phys_location is implemented.

Parameters:

`[in]`	*fill_port*	Port of the key fill interface.
`[out]`	*key_ref*	Reference to newly imported key.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_MALFORMED, S_KEY_METADATA_MALFORMED, S_KEY_FILL_DEVICE_NOT_CONNECTED, S_KEY_FILL_NOT_INITIATED, S_KEY_TRUST_ANCHOR, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::SymKeyManager::import_key_via_fill_into_phys_location()

```
CICM::Status import_key_via_fill_into_phys_location(
        in  CICM::LocalPort fill_port,
        in  CICM::UInt32 phys_location,
        out CICM::SymKey key_ref
);
```

Initiate the import of key material into a specific key physical location via a key fill interface.

Remarks:

In some cases, a key fill device can initiate and effect the filling of key into a module completely independent of the host and thus any API control. In such cases, the host will utilize the CICM::SymKeyManager::get_key_by_id, CICM::SymKeyManager::get_key_by_phys_location, or CICM::SymKeyManager::get_key_last_filled methods after the key fill has completed to enable future reference to the key.

Keys may be imported into a module as wrapped entities, necessitating the use of CICM::Key::unwrap to decrypt the key prior to use.

Key attributes may optionally be set to create or supplement key metadata.

Warning:

This call should only be used when a module utilizes a key storage model that requires keys of particular types to be stored in specific locations or in situations where keys are tied to module capabilities based upon physical location of the key. Because of the architectural knowledge of a module required, its use is highly discouraged because it virtually guarantees that code using it will be incompatible with other dissimilar modules.

This method SHALL be unimplemented if CICM::SymKeyManager::import_key_via_fill is implemented.

Parameters:

`[in]`	*fill_port*	Port of the key fill interface.
`[in]`	*phys_location*	Physical location into which to import key.
`[out]`	*key_ref*	Reference to newly imported key.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_PHYSICAL_LOC, S_KEY_MALFORMED, S_KEY_METADATA_MALFORMED, S_KEY_FILL_DEVICE_NOT_CONNECTED, S_KEY_FILL_NOT_INITIATED, S_KEY_TRUST_ANCHOR, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::SymKeyManager::generate_key()

```
CICM::Status generate_key(
        in  CICM::SymEncrAlgorithmId algorithm,
        out CICM::SymKey key_ref
)
```

Generate a symmetric key compatible with the characteristics of the specified algorithm.

Parameters:

[in]	*algorithm*	Desired algorithm of resulting symmetric key.
[out]	*key_ref*	Reference to newly generated key.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_INSUFFICIENT_ENTROPY

Method CICM::SymKeyManager::derive_key()

```
CICM::Status derive_key(
        in  CICM::CharString password,
        in  CICM::Buffer salt,
        in  CICM::UInt32 iteration_count,
        in  CICM::HashAlgorithmId hash_algorithm,
        in  CICM::SymEncrAlgorithmId algorithm,
        out CICM::SymKey key_ref
);
```

Derives a symmetric key from a password and other parameters using a password-based key derivation function (PBKDF).

Parameters:

[in]	*password*	Password for conversion into a cryptographic key.
[in]	*salt*	Binary salt value.
[in]	*iteration_count*	Positive integer representing number of iterations to apply to hashing algorithm.
[in]	*hash_algorithm*	Hash function applied to derive key.
[in]	*algorithm*	Desired algorithm/mode of resulting symmetric key.
[out]	*key_ref*	Reference to newly derived key.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE,

S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_PASSWORD_INVALID, S_PASSWORD_INVALID_CHAR, S_PASSWORD_INVALID_LEN, S_SALT_INVALID, S_ITERATION_COUNT_INVALID

Method CICM::SymKeyManager::derive_deterministic_key()

```
        CICM::Status derive_deterministic_key(
                in  CICM::SymKey key_prod_key,
                in  CICM::CharString shared_secret,
                in  CICM::SymEncrAlgorithmId algorithm,
                out CICM::SymKey key_ref
        );
```

Derives a symmetric key using a distributed deterministic key generation scheme.

Remarks:

This may be used by peers in an ad-hoc group who initially only share a key production key (KPK) but are subsequently able to share an additional secret.

Parameters:

`[in]`	*key_prod_key*	Key production key.
`[in]`	*shared_secret*	Text-based secret sharable amongst peers in a group.
`[in]`	*algorithm*	Desired algorithm/mode of resulting symmetric key.
`[out]`	*key_ref*	Reference to newly derived key.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_CLASSIFICATION, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_PASSWORD_INVALID, S_PASSWORD_INVALID_CHAR, S_PASSWORD_INVALID_LEN

3.4.3.2 Interface CICM::SymKey

Interface CICM::SymKey

```
interface SymKey : CICM::Key {
```

CICM::SymKey serves as a reference to a symmetric key contained within a module and supports operations on symmetric keys, including key conversion, updating, wrapping, and unwrapping.

3.4.3.2.1 CICM::SymKey Inheritance

CICM::SymKey inherits from: CICM::Key.

3.4.3.2.2 CICM::SymKey Types and Constants

Type CICM::SymKey::Usage

```
typedef CICM::UInt32 Usage;
```

Symmetric key usage types.

Constant CICM::SymKey::C_USAGE_GENERATE_KEYSTREAM

```
const CICM::SymKey::Usage C_USAGE_GENERATE_KEYSTREAM = 0x0000601A;
```

Key intended for generating keystream.

Constant CICM::SymKey::C_USAGE_KEY_PRODUCTION_KEY

```
const CICM::SymKey::Usage C_USAGE_KEY_PRODUCTION_KEY = 0x0000601C;
```

Key intended for producing other keys.

Constant CICM::SymKey::C_USAGE_MESSAGE_AUTHENTICATION_CODE

```
const CICM::SymKey::Usage C_USAGE_MESSAGE_AUTHENTICATION_CODE =
0x0000601F;
```

Key intended for computing a Message Authentication Code.

Constant CICM::SymKey::C_USAGE_SYM_DATA_ENCIPHERMENT

```
const CICM::SymKey::Usage C_USAGE_SYM_DATA_ENCIPHERMENT = 0x00006020;
```

Key intended for enciphering data.

Constant CICM::SymKey::C_USAGE_SYM_KEY_ENCIPHERMENT

```
const CICM::SymKey::Usage C_USAGE_SYM_KEY_ENCIPHERMENT = 0x00006023;
```

Key intended for enciphering other keys.

3.4.3.2.3 CICM::SymKey Attributes

Attribute CICM::SymKey::update_count

```
    readonly attribute CICM::UInt32 update_count;
```

Key update count.

3.4.3.2.4 CICM::SymKey Methods

Method CICM::SymKey::update()

```
    CICM::Status update();
```

Cryptographically update the key using the key's native algorithm. The update modifies the existing key.

Remarks:

The new update count resulting from a call to this method is available as an attribute of the key object.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_INCOMPATIBLE, S_KEY_WRAPPED, S_KEY_UPDATE_MAX, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

See also:

CICM::SymKey::update_with_algo for the version of this method that accepts an algorithm parameter.

Method CICM::SymKey::update_with_algo()

```
        CICM::Status update_with_algo(
                in  CICM::SymEncrAlgorithmId algorithm
        );
```

Cryptographically update the key using the specified key update algorithm. The update modifies the existing key.

Remarks:

The new update count resulting from a call to this method is available as an attribute of the key object.

Parameters:

`[in]` *algorithm* Cryptographic algorithm/mode to use to effect the key update.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_INCOMPATIBLE, S_KEY_WRAPPED, S_KEY_UPDATE_MAX, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

See also:

CICM::SymKey::update for the version of this method that does not require an algorithm to be specified.

Method CICM::SymKey::wrap_and_copy()

```
        CICM::Status wrap_and_copy(
                in  CICM::Key kek,
                in  CICM::KeyWrapAlgorithmId algorithm,
                out CICM::SymKey wrapped_key
        );
```

Instruct module to wrap key, resulting in two keys, the original unwrapped key and the newly wrapped key.

Parameters:

`[in]`	*kek*	Reference to key encryption key.
`[in]`	*algorithm*	Key wrap algorithm/mode used to wrap key.
`[out]`	*wrapped_key*	Reference to resulting wrapped key.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_CLASSIFICATION, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_INCOMPATIBLE, S_KEY_CLASSIFICATION, S_KEY_WRAPPED, S_KEY_NOT_WRAPPABLE, S_KEY_WRAPPED_EXISTS, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

See also:

CICM::AsymKey::wrap_and_copy for the asymmetric version of this method.

Method CICM::SymKey::unwrap_and_copy()

```
        CICM::Status unwrap_and_copy(
                in  CICM::Key kek,
                in  CICM::KeyWrapAlgorithmId algorithm,
                out CICM::SymKey unwrapped_key
        );
```

Instruct module to unwrap key, resulting in two keys, the original wrapped key and the newly unwrapped key.

Parameters:

`[in]`	*kek*	Reference to key decryption key.
`[in]`	*algorithm*	Key wrap algorithm/mode used to unwrap key.
`[out]`	*unwrapped_key*	Reference to resulting unwrapped key.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_CLASSIFICATION, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_INCOMPATIBLE, S_KEY_CLASSIFICATION, S_KEY_UNWRAPPED_EXISTS, S_KEY_MALFORMED, S_KEY_METADATA_MALFORMED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

See also:

CICM::AsymKey::unwrap_and_copy for the asymmetric version of this method.

Method CICM::SymKey::validate_key_usage()

```
        CICM::Status validate_key_usage(
                in  CICM::SymKey::Usage usage_type,
                out CICM::Key::UsageStatus valid
        );
```

Validate that this key may be used for a specific purpose.

Remarks:

In some cases, a given key can be used for multiple purposes.

Parameters:

`[in]`	*usage_type*	Specific purpose to validate.
`[out]`	*valid*	Indiciates if the key may be used for the specified purpose.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_WRAPPED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

See also:

CICM::AsymKey::validate_key_usage for the asymmetric version of this method.

3.4.3.3 Interface CICM::SymKeyIterator

Interface CICM::SymKeyIterator

```
interface SymKeyIterator : CICM::Iterator {
```

CICM::SymKeyIterator supports retrieving a reference to each usable symmetric key on a module. CICM::SymKeyIterator constructs the CICM::SymKey interface.

3.4.3.3.1 CICM::SymKeyIterator Inheritance

CICM::SymKeyIterator inherits from: CICM::Iterator.

3.4.3.3.2 CICM::SymKeyIterator Methods

Method CICM::SymKeyIterator::get_next()

```
        CICM::Status get_next(
                out     CICM::SymKey sym_key_ref
        );
```

Returns a reference to the next symmetric key.

Remarks:

Use CICM::Iterator::has_next to determine if additional elements exist.

Parameters:

`[out]` *sym_key_ref* Reference to next symmetric key.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

3.4.4 KEY PROTOCOL

The key management capabilities described here support certain key management protocols, including key establishment/distribution protocols such as Diffie-Hellman, EC-DH, and EC-MQV; trust anchor management protocols; the importation of key white lists (acceptable keys) or black lists (revoked keys or keys restricted administratively); and the execution of remote key functions. These key protocol IDL interfaces initiate a key agreement protocol between two or more entities to establish a secret key over an insecure communications channel. CICM channel negotiators (CICM::Negotiator) similarly initiate a key agreement protocol with a remote entity, but this typically results in an ephemeral key, in contrast to these key protocol interfaces which result in a persistent symmetric key that can be used by a variety of key management and channel management functions.

For the purposes of these key protocol-related interfaces, a client program using the CICM API is only an intermediary in a key management protocol being conducted between a cryptographic module and other participants in a protocol session. In this role, a client program determines which module(s) should be involved and conveys the protocol messages, but otherwise does not participate in the protocol session. The client may be responsible for determining which key agreement protocol to use as well as for the reliable transport of messages passed between the peer entities. The interactions that comprise a protocol session in its entirety may entail a number of exchanges among the participants in the protocol. Any results from calls to key protocol methods during the course of the protocol exchange must be communicated to the appropriate peer entity by the caller. The progress, success, or failure of the protocol session is determined by the modules and other active participants in the interaction.

CICM key protocol functionality is exported via two independent objects: CICM::KeyProtocolSender for protocol messages inbound to the module and CICM::KeyProtocolReceiver for messages outbound from the module. Access to each respective object is available via CICM::SymKeyManager::key_protocol_sender and CICM::SymKeyManager::key_protocol_receiver.

The key protocol methods support protocol sessions that may be long-term interactions potentially extending over several invocations of the controlling client program. To allow for this possibility, KeyProtocolReceiver::get_from_module may be used to "query" a module to determine if a response is ready and, if so, to retrieve response traffic (including current session status information) from the module. Thus, a single response from the module may take the form of a number of queries by the client program, with any productive response deferred until the module is ready:

query, response is C_PROTOCOL_RECEIVE_BUSY
query, response is C_PROTOCOL_RECEIVE_BUSY
...
query, response is C_PROTOCOL_RECEIVE_BUSY
query, results returned, response is C_PROTOCOL_RECEIVE_OKAY

In the example above, the client program expects either a "condition" update from the module as part of the protocol session, or the most recent results from the active session. Thus, the client program queries the module periodically until it is ready to produce a response.

The traffic relayed by these functions is part of a specific protocol session. The protocol governing this session is specified with the protocol parameter, either the value CICM::IMPLICIT_PROTOCOL_ID, denoting that the message itself indicates the protocol, or a unique protocol identifier designating the protocol directly. If the protocol parameter is CICM::IMPLICIT_PROTOCOL_ID but the message does not indicate the protocol, then the method fails, returning the CICM::S_PROTO_UNDETERMINED error status.

The initial message in a protocol exchange can be generated either by the cryptographic module or some other party, including some party in a key management infrastructure or the client program itself. If the initial message is generated by the cryptographic module, a two-step process allows the message to be retrieved from the module:

1. The module notifies the client program that a protocol message is available to be retrieved via the CICM module event listener facility (this requires a CICM::ModuleEventListener::C_MODULE_KEY_PROTO_MESSAGE to have been previously registered by the client program). The listener facility provides an opaque buffer as part of the notification that is passed to the module in the following step to identify the specific protocol message involved.
2. The CICM::KeyProtocolReceiver::get_from_module method is used to retrieve the message from the module.

If the initial message is generated by some other party, the CICM::KeyProtocolSender::put_into_module method is used to convey the message into the module.

Any key protocol-related key fill device interactions are outside the scope of the API.

Individual protocol events do not require a transaction identifier. Instead, each message itself indicates where it is to be forwarded (i.e. which module or other participant is to receive the message). This means that the client program must be capable of determining which module to associate with a given message, possibly by examining metadata conveyed with the message.

3.4.4.1 Participants in the Interaction

There are three types of participants in a protocol session using the key protocol functionality:

- The cryptographic modules themselves. Note that although these methods work with modules individually, there may in fact be several modules involved in a single protocol session, e.g., to deliver the same key material to several modules available to a host or in a net.
- The other participants (generally key management infrastructure components) that also take an active part in the session, i.e., those that generate and/or consume messages exchanged in the session (perhaps with status indications embedded in the messages). All active participants, to the extent they find necessary, maintain and update internal status as the session continues.
- The intermediary client program that conveys the messages among the active participants in the session (the cryptographic modules themselves and the other active participants in the session). Note that a module may generate a message for transmission to another module as part of the protocol session.

The CICM API is the interface between the intermediary client programs and the cryptographic modules. All exchanges for a specific protocol session between a given module and other modules or other active participants (Party 1, ..., Party N) are mediated by the intermediary client program using CICM. Although the client program does not directly participate in the protocol session, the client program may be asked during the negotiation process to display identifying information about the remote party in the protocol to a human user who must determine if the remote party is the expected remote party in the protocol and, if so, must positively acknowledge this assertion to allow the protocol to continue. Some protocol sessions will not require this peer validation interaction (e.g., validation of the peer using a trust anchor is deemed sufficient or an external trusted display handles the user interaction).

Except for recognizing the specific role of the cryptographic modules themselves, assignment of roles in the protocol to the active participants is out of scope for this document.

3.4.4.2 Return Status, Condition, and Session Status

The key protocol interfaces convey messages between modules and the intermediary client program but do not conduct the actual protocol session. However, the client program still needs to know something about the state of the session, so the key protocol methods impart three types of status information:

- Status return value indicates the status of the method call itself. In the event of a failure, it gives an indication of what the failure was. Note that the returned value does not indicate the state of the protocol session itself: It could report a faulty call (e.g., an invalid protocol identifier) even when the protocol session is still making progress, and it could report a successful exchange with the cryptographic module even when the module decides that the session has reached an error condition.
- The returned condition indication summarizes for the client program the state of the session, perhaps to be used along with other information to suggest what the client program should do next as part of the current protocol session.

- The current or resulting status of the protocol session is embedded in the message conveyed between the client program and the active participants. The client program typically will not interpret the contents of this message; instead, it will simply convey the message to the appropriate active participants, who may then interpret the status in the message and take the appropriate next step in the protocol.

3.4.4.3 Generic Scenario

The following diagram presents a generic scenario for a key protocol session. This diagram does not show message exchanges with any other active participants, since they are out of scope for the API. Note that, although responses from the module to the intermediary client program are represented as arrows in the diagram, in fact, the module actually conveys the response only when the client program explicitly asks or is prompted by an event to ask for it.

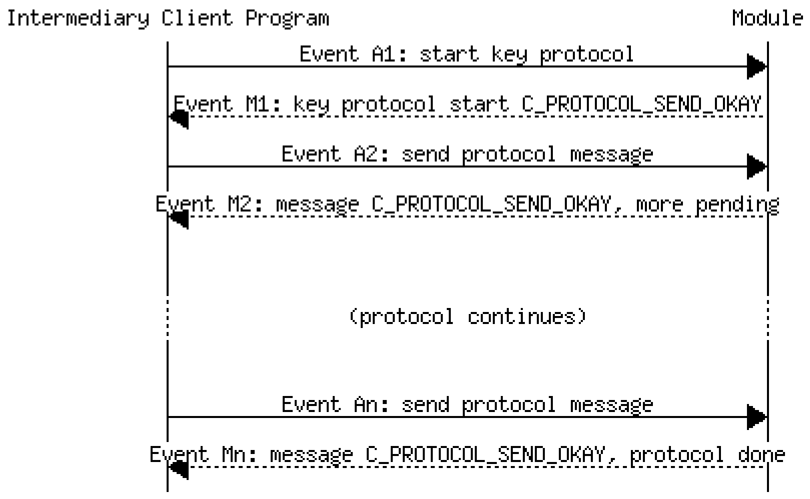

Intermediary Client Program
Module
Event A1: start key protocol
Event M1: key protocol start C_PROTOCOL_SEND_OKAY
Event A2: send protocol message
Event M2: message C_PROTOCOL_SEND_OKAY, more pending
(protocol continues)
Event An: send protocol message
Event Mn: message C_PROTOCOL_SEND_OKAY, protocol done

Figure 25. Generic Scenario for Key Protocol Session Initiated by a Key Infrastructure Component

The notional key infrastructure-initiated message exchanges are as follows:

- Event "A1": Start key protocol - The client program receives the first protocol message from a key infrastructure component and sends it to the module using KeyProtocolSender::put_into_module. The returned condition indicates that the protocol session is C_PROTOCOL_SEND_OKAY.
- Event "M1": Successful start to key protocol - The client program requests the module's response using KeyProtocolReceiver::get_from_module. This response is the module's first message in this exchange and includes the current status of the session. Multiple queries could respond with a BUSY condition before a C_PROTOCOL_SEND_OKAY condition is finally returned. The C_PROTOCOL_SEND_OKAY condition indicates that the session is in progress and a response message has been returned. The client program is responsible for forwarding the resulting response message to another active protocol participant.
- Event "A2": Send protocol message - The client program forwards the next message in the protocol sequence received from the infrastructure component to the module using the KeyProtocolSender::put_into_module method. The returned condition indicates that the protocol session is C_PROTOCOL_SEND_OKAY.
- Event "M2": Message C_PROTOCOL_SEND_OKAY, more pending - The module provides another message in this exchange using KeyProtocolReceiver::get_from_module. The returned condition

indicates that the protocol session is C_PROTOCOL_SEND_OKAY and a responding message has been returned.

- Event "An": Send protocol message - The client program forwards what is ultimately the last message in the protocol sequence to the module using KeyProtocolSender::put_into_module.
- Event "Mn": Message C_PROTOCOL_SEND_OKAY, protocol done - The module provides the corresponding last message in this exchange when the client program queries the module using KeyProtocolReceiver::get_from_module. The returned condition indicates that the protocol session is DONE and the last responding message in this protocol session has been returned. As shown here, the module determines (or at least reports to the client program) when the protocol session is done. Another active participant could plausibly make this determination.

3.4.4.4 Key Agreement Example Using Diffie-Hellman Protocol

The following example depicts a notional key infrastructure ("Entity A") initiating an authenticated Diffie-Hellman Discrete Logarithm (DH-DL) key agreement protocol with a cryptographic module ("Entity B"). A host running a client program using CICM for interactions with the module interposes itself between the key infrastructure and the module.

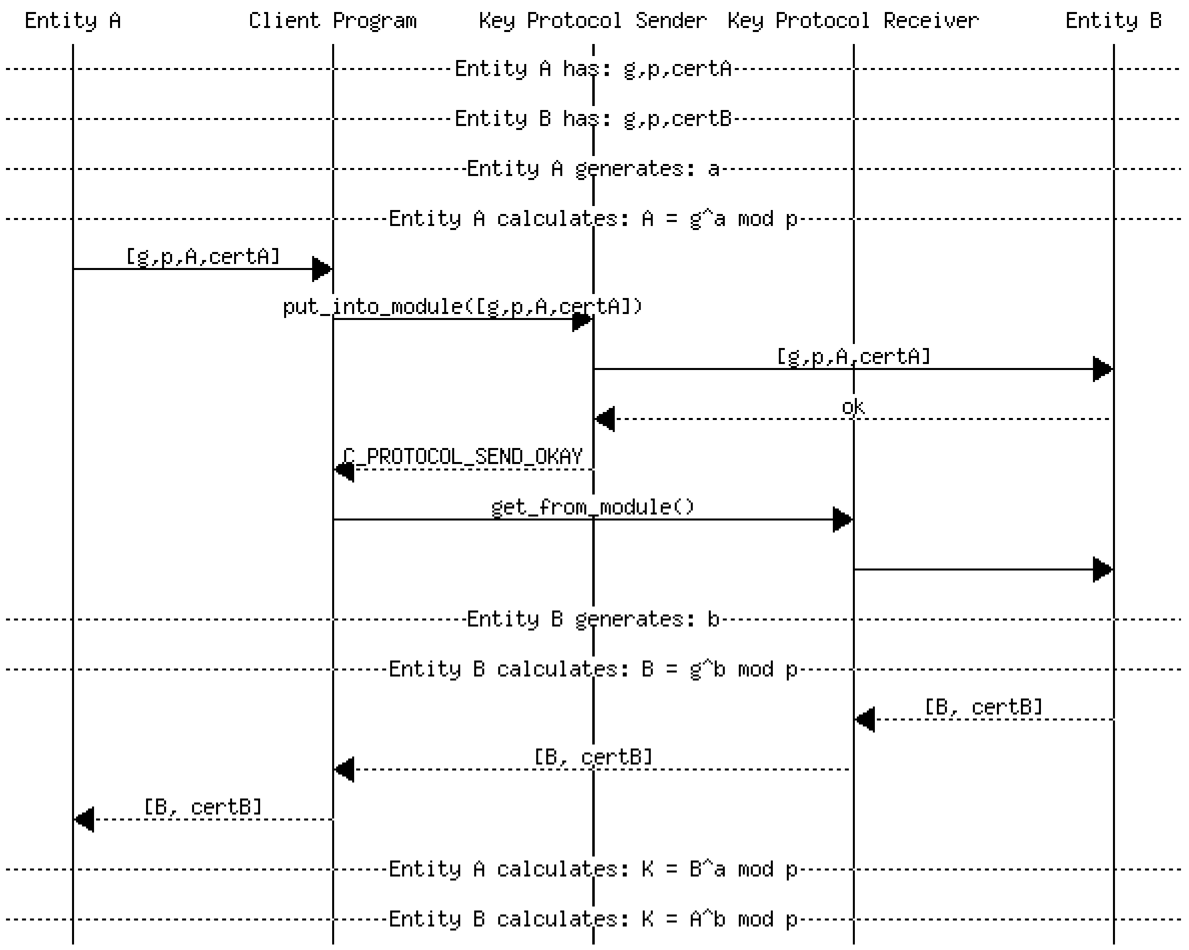


Figure 26. Example of a Two-key Diffie-Hellman Discrete Logarithm (DH-DL) Key Agreement Protocol Initiated by a Key Infrastructure

The following are the steps required to use CICM in the protocol example between a notional key infrastructure ("Entity A") and a cryptographic module ("Entity B"):

1. The following are prepositioned at both communicating elements:
    - Domain parameters g and p.
    - Digital certificate.
    - Signature verification key required to validate the certificate of the other entity.
2. "Entity A" generates random value a.
3. "Entity A" calculates A=g^a mod p.
4. "Entity A" signs the value A calculated above using its static private key whose corresponding static public key is contained in its certificate.
5. "Entity A" sends the signed value A and cert-A to the client program on the intermediary host as an opaque binary buffer.
6. The client program on the intermediary host calls KeyProtocolSender::put_into_module to send the opaque binary buffer received by the client program to the module; if a CICM::KeyProtocolSender::C_PROTOCOL_SEND_DISPLAY condition is returned from this call, the following steps SHOULD be performed (the steps do not appear in the diagram):
    a. "Entity A" calls CICM::Negotiator::get_remote_info to retrieve information about the remote peer; this information (including name/organization and classification level) are extracted from cert-B and returned in CICM::PeerInfo.
    b. The client program on the intermediary host displays the identifying information returned above in CICM::PeerInfo to a human user and asks for positive acknowledgement that the entity initiating the protocol is in fact a legal entity to initiate the protocol.
    c. If the human user does not recognize the remote entity and declines to give positive acknowledgement, the client program abandons the protocol. In this example, positive acknowledgement is given, and the client program calls CICM::KeyProtocolReceiver::get_from_module to request a protocol response.
7. "Entity B" generates random value b.
8. "Entity B" calculates B=g^b mod p, using parameters g and p that it previously agreed it would use when initiating a protocol exchange with "Entity A."
9. "Entity B" signs the value B calculated above using its static private key whose corresponding static public key is contained in its certificate.
10. "Entity B" returns an opaque binary buffer containing its signed value B and cert-B to the caller of KeyProtocolReceiver::get_from_module with a condition of DONE.
11. The client program on the intermediary host sends the opaque binary buffer to "Entity A".
12. Both entities verify their peer's certificate is valid.
13. "Entity A" calculates K=B^a mod p.
14. "Entity B" calculates K=A^b mod p.
15. Both entities now share a symmetric key K.

3.4.4.5 Protocol Support Examples

As previously stated, these methods support a wide range of key management protocols. The following is a notional list of such protocols with a description of their intended usage:

- Key agreement/distribution protocols - Key material can be distributed and keys can be agreed upon using the DH-DL, EC-DH, EC-MQV, or related protocols.
- Remote key functions - Messages containing key-related commands (i.e., zeroize, rekey) requiring authentication by the module prior to execution may be presented using these functions.

- Trust anchor management protocols - Trust anchor management commands and data may be sent as opaque data elements which are interpreted by the module in a protocol-specific manner.
- Key revocation messages - Opaque data elements identifying certificates associated with keys which have been revoked are accepted. The ability to handle a list of revoked keys allows a module to prevent further usage of these keys, including performing key agreement with an entity that is known to have been compromised or no longer possesses the roles or affiliations described in the certificate.

3.4.4.6 Interface CICM::KeyProtocolSender

Interface CICM::KeyProtocolSender

```
interface KeyProtocolSender : CICM::Negotiator {
```

CICM::KeyProtocolSender supports sending key management protocol-related messages into a module.

3.4.4.6.1 CICM::KeyProtocolSender Inheritance

CICM::KeyProtocolSender inherits from: CICM::Negotiator.

3.4.4.6.2 CICM::KeyProtocolSender Types and Constants

Type CICM::KeyProtocolSender::Condition

```
typedef CICM::UInt32 Condition;
```

Condition values summarize for the client program the state of the session. This information can be used along with other information to suggest what the client program should do next as part of the current protocol session.

Constant CICM::KeyProtocolSender::C_PROTOCOL_SEND_OKAY

```
const CICM::KeyProtocolSender::Condition C_PROTOCOL_SEND_OKAY =
0x00006045;
```

Denotes that the session is in progress and a response message is available.

Constant CICM::KeyProtocolSender::C_PROTOCOL_SEND_DONE

```
const CICM::KeyProtocolSender::Condition C_PROTOCOL_SEND_DONE =
0x00006046;
```

Denotes that the session terminated successfully.

Constant CICM::KeyProtocolSender::C_PROTOCOL_SEND_ERROR

```
const CICM::KeyProtocolSender::Condition C_PROTOCOL_SEND_ERROR =
0x00006049;
```

Denotes that the session terminated with an error condition.

Constant CICM::KeyProtocolSender::C_PROTOCOL_SEND_DISPLAY

```
const CICM::KeyProtocolSender::Condition C_PROTOCOL_SEND_DISPLAY =
0x0000604A;
```

As with the C_PROTOCOL_SEND_OKAY condition, denotes that the session is in progress and a response message is available, but additionally denotes that identification information extracted from the remote certificate is available via a call to the CICM::Negotiator::get_remote_info method; the information retrieved from a call to this method must be displayed to a human user on the host and validated before the protocol should be allowed to continue. Note that a trusted display may be employed by the module for the same purpose but, because no API interaction would be involved, the C_PROTOCOL_SEND_DISPLAY condition would not be returned.

Constant CICM::KeyProtocolSender::C_PROTOCOL_SEND_ABORTED

```
const CICM::KeyProtocolSender::Condition C_PROTOCOL_SEND_ABORTED =
0x0000604C;
```

Denotes that the human user reviewing the remote peer information chose to reject it and abort the protocol.

Constant CICM::KeyProtocolSender::C_PROTOCOL_SEND_MESSAGE_INVALID

```
const CICM::KeyProtocolSender::Condition
C_PROTOCOL_SEND_MESSAGE_INVALID = 0x0000604F;
```

Denotes that the conveyed message was found to be invalid for the protocol. This event does not terminate the protocol session.

Constant CICM::KeyProtocolSender::C_PROTOCOL_SEND_MESSAGE_INTEGRITY

```
const CICM::KeyProtocolSender::Condition
C_PROTOCOL_SEND_MESSAGE_INTEGRITY = 0x00006051;
```

Denotes that the conveyed message failed one or more integrity checks used in the protocol. This event does not terminate the protocol session.

Constant CICM::KeyProtocolSender::C_PROTOCOL_SEND_PROTOCOL_VIOLATION

```
const CICM::KeyProtocolSender::Condition
C_PROTOCOL_SEND_PROTOCOL_VIOLATION = 0x00006052;
```

Denotes that a message or attempted action unexpected at the current point in the protocol session was noted. This event does not terminate the protocol session.

3.4.4.6.3 CICM::KeyProtocolSender Methods

Method CICM::KeyProtocolSender::put_into_module()

```
        CICM::Status put_into_module(
                in  CICM::ProtocolId protocol,
                in  CICM::Buffer message,
                out CICM::KeyProtocolSender::Condition condition
        );
```

Initiate or recommence a key management protocol session, forwarding a message to the cryptographic module. If the C_PROTOCOL_SEND_DISPLAY condition results, the get_remote_info method should be called to retrieve identity information about the remote peer for display to and validation by the responsible user before the protocol negotiation is allowed to continue.

Remarks:

The format of the conveyed message is not defined by CICM. If the client program must be capable of generating the message, then the Implementation Conformance Statement (see

Section 4, Conformance and Extensions) must reference a standard format or define a module developer-specific format implemented by the module for this datatype.

Parameters:

`[in]`	*protocol*	Designate the protocol to be followed for this session; the same value must be used for this parameter for all calls to this method or to get_from_module, as part of the same protocol session.
`[in]`	*message*	Message conveyed to the module as part of the current protocol session.
`[out]`	*condition*	Condition of the current protocol session; the calling client program must interpret this value to determine what its next action must be.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_PROTO_INVALID, S_PROTO_INCOMPATIBLE, S_PROTO_UNDETERMINED

See also:

CICM::KeyProtocolSender::put_into_module_algo for the version of this method that accepts an algorithm.

Method CICM::KeyProtocolSender::put_into_module_algo()

```
    CICM::Status put_into_module_algo(
            in  CICM::ProtocolId protocol,
            in  CICM::Buffer message,
            in  CICM::SymEncrAlgorithmId algorithm,
            out CICM::KeyProtocolSender::Condition condition
    );
```

Initiate or recommence a key management protocol session, forwarding a message to the cryptographic module. If the C_PROTOCOL_SEND_DISPLAY condition results, the get_remote_info method should be called to retrieve identity information about the remote peer for display to and validation by the responsible user before the protocol negotiation is allowed to continue. This method differs from KeyProtocolSender::put_into_module in that it enables the caller to specify the desired algorithm of the resulting symmetric key.

Remarks:

The format of the conveyed message is not defined by CICM. If the client program must be capable of generating the message, then the Implementation Conformance Statement (see Section 4, Conformance and Extensions) must reference a standard format or define a module developer-specific format implemented by the module for this datatype.

Parameters:

`[in]`	*protocol*	Designate the protocol to be followed for this session; the same value must be used for this parameter for all calls to this method or to get_from_module, as part of the same protocol session.
`[in]`	*message*	Message conveyed to the module as part of the current protocol session.
`[in]`	*algorithm*	Algorithm/mode of resulting symmetric key.
`[out]`	*condition*	Condition of the current protocol session; the calling client program must interpret this value to determine what its next action must be.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_PROTO_INVALID, S_PROTO_INCOMPATIBLE, S_PROTO_UNDETERMINED

See also:

CICM::KeyProtocolSender::put_into_module for the version of this method that does not require an algorithm.

3.4.4.7 Interface CICM::KeyProtocolReceiver

Interface CICM::KeyProtocolReceiver

```
interface KeyProtocolReceiver {
```

CICM::KeyProtocolReceiver supports receiving key management protocol-related messages from a module. CICM::KeyProtocolReceiver constructs the CICM::SymKey interface.

3.4.4.7.1 CICM::KeyProtocolReceiver Types and Constants

Type CICM::KeyProtocolReceiver::Condition

```
typedef CICM::UInt32 Condition;
```

Condition values summarize for the client program the state of the session. This information can be used along with other information to suggest what the client program should do next as part of the current protocol session.

Constant CICM::KeyProtocolReceiver::C_PROTOCOL_RECEIVE_OKAY

```
const CICM::KeyProtocolReceiver::Condition C_PROTOCOL_RECEIVE_OKAY =
0x00006034;
```

Denotes that the session is in progress and a response message has been returned.

Constant CICM::KeyProtocolReceiver::C_PROTOCOL_RECEIVE_DONE

```
const CICM::KeyProtocolReceiver::Condition C_PROTOCOL_RECEIVE_DONE =
0x00006037;
```

Denotes that the session terminated successfully.

Constant CICM::KeyProtocolReceiver::C_PROTOCOL_RECEIVE_BUSY

```
const CICM::KeyProtocolReceiver::Condition C_PROTOCOL_RECEIVE_BUSY =
0x00006038;
```

Denotes that the session is in progress but no response message or error indication is available at the current time; in this case, the client program must make additional calls to CICM::KeyProtocolReceiver::get_from_module to determine when the response message has become available and retrieve the message.

Constant CICM::KeyProtocolReceiver::C_PROTOCOL_RECEIVE_ERROR

```
const CICM::KeyProtocolReceiver::Condition C_PROTOCOL_RECEIVE_ERROR =
0x0000603B;
```

Denotes that the session terminated with an error condition and a response message has been returned.

Constant CICM::KeyProtocolReceiver::C_PROTOCOL_RECEIVE_ABORTED

```
const CICM::KeyProtocolReceiver::Condition C_PROTOCOL_RECEIVE_ABORTED
= 0x0000603D;
```

Denotes that the human user reviewing the remote peer information chose to reject it and abort the protocol.

Constant CICM::KeyProtocolReceiver::C_PROTOCOL_RECEIVE_MESSAGE_INVALID

```
const CICM::KeyProtocolReceiver::Condition
C_PROTOCOL_RECEIVE_MESSAGE_INVALID = 0x0000603E;
```

Denotes that the conveyed message was found to be invalid for the protocol. This event does not terminate the protocol session.

Constant CICM::KeyProtocolReceiver::C_PROTOCOL_RECEIVE_MESSAGE_INTEGRITY

```
const CICM::KeyProtocolReceiver::Condition
C_PROTOCOL_RECEIVE_MESSAGE_INTEGRITY = 0x00006040;
```

Denotes that the conveyed message failed one or more integrity checks used in the protocol. This event does not terminate the protocol session.

Constant CICM::KeyProtocolReceiver::C_PROTOCOL_RECEIVE_VIOLATION

```
const CICM::KeyProtocolReceiver::Condition
C_PROTOCOL_RECEIVE_VIOLATION = 0x00006043;
```

Denotes that a message or attempted action unexpected at the current point in the protocol session was noted. This event does not terminate the protocol session.

3.4.4.7.2 CICM::KeyProtocolReceiver Methods

Method CICM::KeyProtocolReceiver::abort()

```
CICM::Status abort();
```

Abort negotiation.

Remarks:

This method may be called at any point in the negotiation process for any reason. However, it must be called in the event the identification information for the remote peer does not correspond to the expected peer.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_NOT_IN_PROGRESS

Method CICM::KeyProtocolReceiver::get_from_module()

```
CICM::Status get_from_module(
            in  CICM::ProtocolId protocol,
            out CICM::Buffer message,
            out CICM::KeyProtocolReceiver::Condition condition
    );
```

Initiate or recommence a key management protocol session, soliciting a response from the cryptographic module.

Remarks:

The format of the conveyed message is not defined by CICM. If the client program must be capable of interpreting the contents of the message, then the Implementation Conformance Statement (see Section 4, Conformance and Extensions) must reference a standard format or define a module developer-specific format implemented by the module for this datatype.

Parameters:

`[in]`	*protocol*	Designate the protocol to be followed for this session; the same value must be used for this parameter for all calls to this method or to CICM::KeyProtocolSender::put_into_module, as part of the same protocol session.
`[out]`	*message*	Message returned from the module as part of the current protocol session; message may be of length zero.
`[out]`	*condition*	Condition of the current protocol session; the calling client program must interpret this value to determine what its next action must be.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_PROTO_INVALID, S_PROTO_INCOMPATIBLE, S_PROTO_UNDETERMINED

Method CICM::KeyProtocolReceiver::get_key()

```
    CICM::Status get_key(
            out CICM::SymKey key_ref
    );
```

At successful conclusion of a key agreement/distribution protocol session (when the returned condition is C_PROTOCOL_RECEIVE_DONE), this method is called to retrieve a reference to the key resulting from the session.

Remarks:

Note that, in those cases where the protocol session does not result in a key (e.g., a key revocation message, key white list or black list is presented to the module via this interface), calling this method will result in an CICM::S_INVALID_STATE error.

Parameters:

`[out]` *key_ref* Reference to key resulting from a successful protocol session.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED,

S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_IN_PROGRESS, S_NEGOTIATION_NOT_IN_PROGRESS

3.5 CHANNEL MANAGEMENT

3.5.1 CHANNEL ABSTRACTIONS

3.5.1.1 Interface CICM::ChannelManager

Interface CICM::ChannelManager

```
interface ChannelManager {
```

CICM::ChannelManager supports the creation and negotiation of cryptographic channels. It is accessed from CICM::CryptoModule via the CICM::CryptoModule::channel_manager attribute. CICM::ChannelManager enables a variety of different channel types to be constructed.

Note:

Conforming implementations need only implement one or more of the ChannelManager's dependencies thereby limiting which negotiators, controllers, streams, and channels are available.

3.5.1.1.1 CICM::ChannelManager Inheritance

CICM::ChannelManager inherits from: CICM::Answer::ChannelManager, CICM::BypassRead::ChannelManager, CICM::BypassWrite::ChannelManager, CICM::Coprocessor::ChannelManager, CICM::Decrypt::ChannelManager, CICM::DecryptBypass::ChannelManager, CICM::Duplex::ChannelManager, CICM::Emit::ChannelManager, CICM::Encrypt::ChannelManager and CICM::EncryptBypass::ChannelManager.

3.5.1.1.2 CICM::ChannelManager Methods

Method CICM::ChannelManager::create_controller_group()

```
        CICM::Status create_controller_group(
                out CICM::ControllerGroup controller_group_ref
        );
```

Creates a CICM::ControllerGroup to group controllers and conduits together.

Parameters:

`[out]` *controller_group_ref* Reference to the created controller group.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

3.5.1.2 Interface CICM::Channel

Interface CICM::Channel

```
interface ChannelEventListener {
```

Defines the logical path through the module. Interface from which all conduit, streams and controllers inherit.

Channels are created via the CICM::ChannelManager interface.

3.5.1.2.1 CICM::Channel Attributes

Attribute CICM::Channel::event_manager
```
        readonly attribute CICM::ChannelEventManager event_manager;
```
Provides access to the event manager.

3.5.1.3 Interface CICM::Conduit

Interface CICM::Conduit
```
interface Conduit :
        CICM::Controller,
        CICM::Stream {
```
Interface from which all other conduits are inherited. A conduit is a combination of a stream and controller.

3.5.1.3.1 CICM::Conduit Inheritance

CICM::Conduit inherits from: CICM::Controller and CICM::Stream.

3.5.1.4 Interface CICM::Controller

Interface CICM::Controller
```
interface Controller : CICM::Channel {
```
Interface from which all other controllers are inherited. Controls general characteristics of a cryptographic transformation, but does not provide data to be transformed.
Remarks:

> There may be cases in which a client program wishes to delegate responsibility for sending or receiving data from the module to another process while retaining the authority to manage the channel. To support this task, both processes must share a known local port. The client-program responsible for controlling the channel creates a CICM::Controller of the appropriate type after which the corresponding CICM::Stream may be obtained by the second process. A stream is tied to the specific controller that configured the channel by a common port value.

3.5.1.4.1 CICM::Controller Inheritance

CICM::Controller inherits from: CICM::Channel.

3.5.1.4.2 CICM::Controller Methods

Method CICM::Controller::destroy()
```
        CICM::Status destroy();
```
Destroys the controller.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

3.5.1.5 Interface CICM::Stream

Interface CICM::Stream

```
interface Stream : CICM::Channel {};
```

Interface from which all streams inherit. Streams manage the flow of data on a channel, but not its attributes.

3.5.1.5.1 CICM::Stream Inheritance

CICM::Stream inherits from: CICM::Channel.

3.5.2 CONDUIT ABSTRACTIONS

3.5.2.1 Interface CICM::AbstractMACConduit

Interface CICM::AbstractMACConduit

```
interface AbstractMACConduit : CICM::Conduit {
```

Interface from which other MAC conduits are inherited.

3.5.2.1.1 CICM::AbstractMACConduit Inheritance

CICM::AbstractMACConduit inherits from: CICM::Conduit.

3.5.2.1.2 CICM::AbstractMACConduit Attributes

Attribute CICM::AbstractMACConduit::mac_key

```
        readonly attribute CICM::SymKey mac_key;
```

The key used for computing the MAC.

Attribute CICM::AbstractMACConduit::mac_algorithm

```
        readonly attribute CICM::SymMacAlgorithmId mac_algorithm;
```

The algorithm used to MAC the data.

3.5.2.1.3 CICM::AbstractMACConduit Methods

Method CICM::AbstractMACConduit::end_get_mac()

```
        CICM::Status end_get_mac(
                out CICM::MACBuffer mac
        );
```

Direct the module to compute and output the MAC value, and reset the channel to accept additional data.
Parameters:

`[out]` *mac* Computed MAC value.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

3.5.2.2 Interface CICM::AbstractSignConduit

Interface CICM::AbstractSignConduit
```
interface AbstractSignConduit : CICM::Conduit {
```
Interface from which other sign conduits are inherited.

3.5.2.2.1 CICM::AbstractSignConduit Inheritance

CICM::AbstractSignConduit inherits from: CICM::Conduit.

3.5.2.2.2 CICM::AbstractSignConduit Attributes

Attribute CICM::AbstractSignConduit::sign_key
```
        readonly attribute CICM::AsymKey sign_key;
```
Key used for signing the data.

Attribute CICM::AbstractSignConduit::sign_algorithm
```
        readonly attribute CICM::AsymSigAlgorithmId sign_algorithm;
```
Algorithm used to sign the data.

3.5.2.2.3 CICM::AbstractSignConduit Methods

Method CICM::AbstractSignConduit::end_get_signature()
```
        CICM::Status end_get_signature(
                out CICM::SigBuffer signature
        );
```
Direct the module to compute and output the signature, and reset the conduit to accept additional data.
Parameters:

`[out]` *signature* The computed signature.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

3.5.2.3 Interface CICM::AbstractVerifyConduit

Interface CICM::AbstractVerifyConduit
```
interface AbstractVerifyConduit : CICM::Conduit {
```
Interface from which other verification conduits are inherited.

3.5.2.3.1 CICM::AbstractVerifyConduit Inheritance

CICM::AbstractVerifyConduit inherits from: CICM::Conduit.

3.5.2.3.2 CICM::AbstractVerifyConduit Types and Constants

Type CICM::AbstractVerifyConduit::VerifyStatus
```
typedef CICM::UInt32 VerifyStatus;
```
Verification status (data verifies/does not verify).

Constant CICM::AbstractVerifyConduit::C_DATA_VERIFIED
```
const CICM::AbstractVerifyConduit::VerifyStatus C_DATA_VERIFIED =
0x00006025;
```
Data verifies.

Constant CICM::AbstractVerifyConduit::C_DATA_NOT_VERIFIED
```
const CICM::AbstractVerifyConduit::VerifyStatus C_DATA_NOT_VERIFIED =
0x00006026;
```
Data does not verify.

3.5.2.4 Interface CICM::AbstractMACVerifyConduit

Interface CICM::AbstractMACVerifyConduit
```
interface AbstractMACVerifyConduit : CICM::AbstractVerifyConduit {
```
Interface from which other MAC verify conduits are inherited.

3.5.2.4.1 CICM::AbstractMACVerifyConduit Inheritance

CICM::AbstractMACVerifyConduit inherits from: CICM::AbstractVerifyConduit.

3.5.2.4.2 CICM::AbstractMACVerifyConduit Attributes

Attribute CICM::AbstractMACVerifyConduit::verify_key
```
        readonly attribute CICM::SymKey verify_key;
```
Key used to verify the MAC.

Attribute CICM::AbstractMACVerifyConduit::verify_algorithm
```
        readonly attribute CICM::SymMacAlgorithmId verify_algorithm;
```
Algorithm used to verify the data.

3.5.2.4.3 CICM::AbstractMACVerifyConduit Methods

Method CICM::AbstractMACVerifyConduit::end_get_verified()

```
        CICM::Status end_get_verified(
                in  CICM::MACBuffer mac,
                out CICM::AbstractVerifyConduit::VerifyStatus status
        );
```

Direct the module to compute and output the MAC verification status, and reset the channel to accept additional data for verification.

Parameters:

`[in]` *mac* Message authentication code.

`[out]` *status* Status indicating whether or not the data verifies.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_DATA_INVALID, S_CHANNEL_DATA_INVALID_LEN

3.5.2.5 Interface CICM::AbstractSigVerifyConduit

Interface CICM::AbstractSigVerifyConduit

```
interface AbstractSigVerifyConduit : CICM::AbstractVerifyConduit {
```

Interface from which other signature verification conduits are inherited.

3.5.2.5.1 CICM::AbstractSigVerifyConduit Inheritance

CICM::AbstractSigVerifyConduit inherits from: CICM::AbstractVerifyConduit.

3.5.2.5.2 CICM::AbstractSigVerifyConduit Attributes

Attribute CICM::AbstractSigVerifyConduit::verify_key

```
        readonly attribute CICM::AsymKey verify_key;
```

Key used to verify the signature.

Attribute CICM::AbstractSigVerifyConduit::verify_algorithm

```
        readonly attribute CICM::AsymSigAlgorithmId verify_algorithm;
```

Algorithm used to verify the data.

3.5.2.5.3 CICM::AbstractSigVerifyConduit Methods

Method CICM::AbstractSigVerifyConduit::end_get_verified()

```
        CICM::Status end_get_verified(
                in  CICM::SigBuffer signature,
                out CICM::AbstractVerifyConduit::VerifyStatus status
        );
```

Direct the module to compute and output the verification status, and reset the channel to accept additional data for verification.

Parameters:

`[in]`	*signature*	Signature.
`[out]`	*status*	Status indicating whether or not the data verifies.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_DATA_INVALID, S_CHANNEL_DATA_INVALID_LEN

3.5.3 STREAM ABSTRACTIONS

3.5.3.1 Interface CICM::WriteStream

Interface CICM::WriteStream

```
interface WriteStream : CICM::Stream {
```

Interface from which other streams that write data to the module inherit.

3.5.3.1.1 CICM::WriteStream Inheritance

CICM::WriteStream inherits from: CICM::Stream.

3.5.3.1.2 CICM::WriteStream Types and Constants

Type CICM::WriteStream::WriteStatus

```
typedef CICM::UInt32 WriteStatus;
```

Status of an non-blocking write.

Constant CICM::WriteStream::C_WRITE_NOT_READY

```
const CICM::WriteStream::WriteStatus C_WRITE_NOT_READY = 0x00006067;
```

Module is not ready to receive data for writing.

Constant CICM::WriteStream::C_WRITE_READY

```
const CICM::WriteStream::WriteStatus C_WRITE_READY = 0x00006068;
```

Module is ready to receive data for writing.

3.5.3.2 Interface CICM::ReadStream

Interface CICM::ReadStream

```
interface ReadStream : CICM::Stream {
```

Interface from which all other streams that read data from the module inherit.

3.5.3.2.1 CICM::ReadStream Inheritance

CICM::ReadStream inherits from: CICM::Stream.

3.5.3.2.2 CICM::ReadStream Types and Constants

Type CICM::ReadStream::ReadStatus
```
typedef CICM::UInt32 ReadStatus;
```
Status of a non-blocking read.

Constant CICM::ReadStream::C_READ_NOT_READY
```
const CICM::ReadStream::ReadStatus C_READ_NOT_READY = 0x0000605E;
```
Module does not have data ready for reading.

Constant CICM::ReadStream::C_READ_READY
```
const CICM::ReadStream::ReadStatus C_READ_READY = 0x00006061;
```
Module has data ready for reading.

3.5.4 CONTROLLER ABSTRACTIONS

3.5.4.1 Interface CICM::MultiDomainController

Interface CICM::MultiDomainController
```
interface MultiDomainController : CICM::Controller {
```
Interface from which any other multi-domain-related controller or conduit inherits.

Remarks:

Provides read access to the local and remote port values associated with the controller.

3.5.4.1.1 CICM::MultiDomainController Inheritance

CICM::MultiDomainController inherits from: CICM::Controller.

3.5.4.1.2 CICM::MultiDomainController Attributes

Attribute CICM::MultiDomainController::local_port
```
        readonly attribute CICM::LocalPort local_port;
```
Local port associated with this controller.

Attribute CICM::MultiDomainController::remote_port
```
        readonly attribute CICM::RemotePort remote_port;
```
Remote port associated with this controller.

3.5.4.2 Interface CICM::SymKeyController

Interface CICM::SymKeyController
```
interface SymKeyController : CICM::Controller {
```

Interface from which all controllers using a symmetric key inherit.

3.5.4.2.1 CICM::SymKeyController Inheritance

CICM::SymKeyController inherits from: CICM::Controller.

3.5.4.2.2 CICM::SymKeyController Attributes

Attribute CICM::SymKeyController::key

```
        readonly attribute CICM::SymKey key;
```

3.5.4.2.3 CICM::SymKeyController Methods

Method CICM::SymKeyController::update_key()

```
        CICM::Status update_key();
```

Cryptographically update the key associated with this controller using the key's native algorithm.

Remarks:

To update an arbitrary key, use CICM::SymKey::update.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

See also:

CICM::SymKey::update_key_with_algo for the version of this method that accepts an algorithm.

Method CICM::SymKeyController::update_key_with_algo()

```
        CICM::Status update_key_with_algo(
                in  CICM::SymEncrAlgorithmId algorithm
        );
```

Cryptographically update the key associated with this controller using the specified algorithm.

Remarks:

To update an arbitrary key, use CICM::SymKey::update.

Parameters:

`[in]` *algorithm* Cryptographic algorithm/mode to use to effect the key update.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

See also:

CICM::SymKey::update_key for the version of this method that does not require an algorithm to be specified.

Method CICM::SymKeyController::rollover_key()

```
        CICM::Status rollover_key();
```

Specify that the next pre-placed key be used with this controller.

Remarks:

A call to this method may be required when a key reaches the end of its specified or usable lifespan or for other reasons that require that a controller move to a new key.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_NO_NEXT, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

See also:

CICM::SymKey::rollover_key_with_key for the version of this method that accepts a key.

Method CICM::SymKeyController::rollover_key_with_key()

```
        CICM::Status rollover_key_with_key(
                in  CICM::SymKey next_key
        );
```

Specify the next pre-placed key to be used with this controller.

Remarks:

A call to this method may be required when a key reaches the end of its specified or usable lifespan or for other reasons that require that a controller move to a new key.

Parameters:

`[in]` *next_key* Reference to pre-placed key to use with this controller.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_CLASSIFICATION, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

See also:

CICM::SymKey::rollover_key for the version of this method for which the key is implicit.

3.5.4.3 Interface CICM::AsymKeyController

Interface CICM::AsymKeyController

```
interface AsymKeyController : CICM::Controller {
```

Interface from which all controllers using an asymmetric key inherit.

3.5.4.3.1 CICM::AsymKeyController Inheritance

CICM::AsymKeyController inherits from: CICM::Controller.

3.5.4.3.2 CICM::AsymKeyController Attributes

Attribute CICM::AsymKeyController::key

```
        readonly attribute CICM::AsymKey key;
```
Provides read-only access to the key associated with a controller.

3.5.4.4 Interface CICM::NegotiatedController

Interface CICM::NegotiatedController
```
interface NegotiatedController :
        CICM::MultiDomainController,
        CICM::AsymKeyController,
        CICM::Negotiator {
```
Interface from which all other negotiated controllers inherit.

A controller that uses a negotiated key.

3.5.4.4.1 CICM::NegotiatedController Inheritance

CICM::NegotiatedController inherits from: CICM::MultiDomainController, CICM::AsymKeyController and CICM::Negotiator.

3.5.4.4.2 CICM::NegotiatedController Attributes

Attribute CICM::NegotiatedController::negotiated_grade
```
        readonly attribute CICM::Classification negotiated_grade;
```
Returns the grade (classification level) of the channel.

3.5.4.4.3 CICM::NegotiatedController Methods

Method CICM::NegotiatedController::renegotiate()
```
        CICM::Status renegotiate();
```
Renegotiates the traffic encryption key with the associated peer.

Remarks:

In cases where the key is no longer usable, e.g. it expired or was zeroized, the session must be torn down and rebuilt. Note that renegotiation may not need to be explicitly initiated in all cases because some modules will automatically initiate a renegotiation when sensing a certain condition (e.g., key expiration, cumulative traffic volume threshold passed, etc.).

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_CHANNEL_ERROR

Method CICM::NegotiatedController::initiate_grade_change()
```
        CICM::Status initiate_grade_change(
                 in  CICM::Classification new_grade
        );
```
Requests a change of grade (classification level) for the current traffic.

Parameters:

`[in]` *new_grade* New grade for the traffic.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

Method CICM::NegotiatedController::acknowledge_grade_change()
```
        CICM::Status acknowledge_grade_change();
```
Positively acknowledges the requested change of grade.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

3.5.4.5 Interface CICM::SetVectorController

Interface CICM::SetVectorController
```
interface SetVectorController : CICM::Controller {
```
Provides access to the state vector associated with a controller.

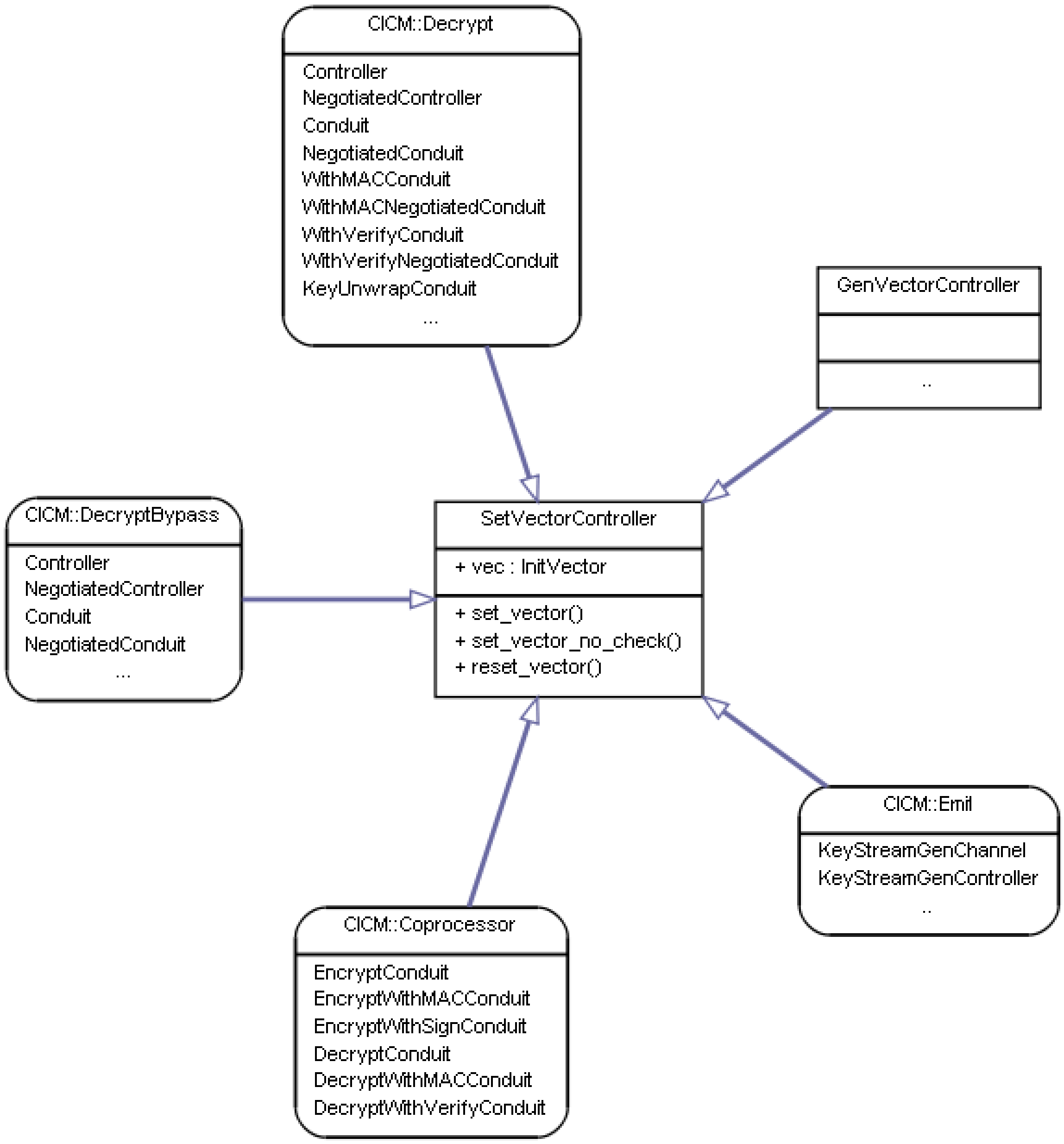


Figure 27. Interface Inheritance Diagram for SetVectorController

3.5.4.5.1 CICM::SetVectorController Inheritance

CICM::SetVectorController inherits from: CICM::Controller.

3.5.4.5.2 CICM::SetVectorController Attributes

Attribute CICM::SetVectorController::vec

```
        readonly attribute CICM::Vector vec;
```

State vector associated with this controller.

3.5.4.5.3 CICM::SetVectorController Methods

Method CICM::SetVectorController::set_vector()

```
        CICM::Status set_vector(
                in  CICM::Vector vec
        );
```

Set channel state vector.

Remarks:

This state value may be set when a controller is initially created (initialization vector). It may also be set on a block-by-block basis for appropriate algorithms/modes or for each time epoch (e.g., for time-of-day encryption).

Parameters:

`[in]` *vec* State vector.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_INVALID_VECTOR, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::SetVectorController::set_vector_no_check()

```
        CICM::Status set_vector_no_check(
                in  CICM::Vector vec
        );
```

Set channel state vector without a policy check.

Remarks:

This state value may be set when a controller is initially created (initialization vector) or on a per-message basis, depending upon how the cryptographic algorithm uses the vector. The length and format of the vector are specific to the algorithm associated with the channel and system in which the channel is being used.

This version of the set_vector() method may be used on the decrypt side, for example, to specify no TOD check in cases where TOD rules are not enforced for decryption.

Parameters:

`[in]` *vec* State vector.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_INVALID_VECTOR, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

Method CICM::SetVectorController::reset_vector()

```
        CICM::Status reset_vector();
```

Reset channel state vector to system-dependent value.

Remarks:

This can be used by the client program to manage time-of-day or counter rollover.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

3.5.4.6 Interface CICM::GenVectorController

Interface CICM::GenVectorController

```
interface GenVectorController : CICM::SetVectorController {
```

Enables an state vector to be generated.

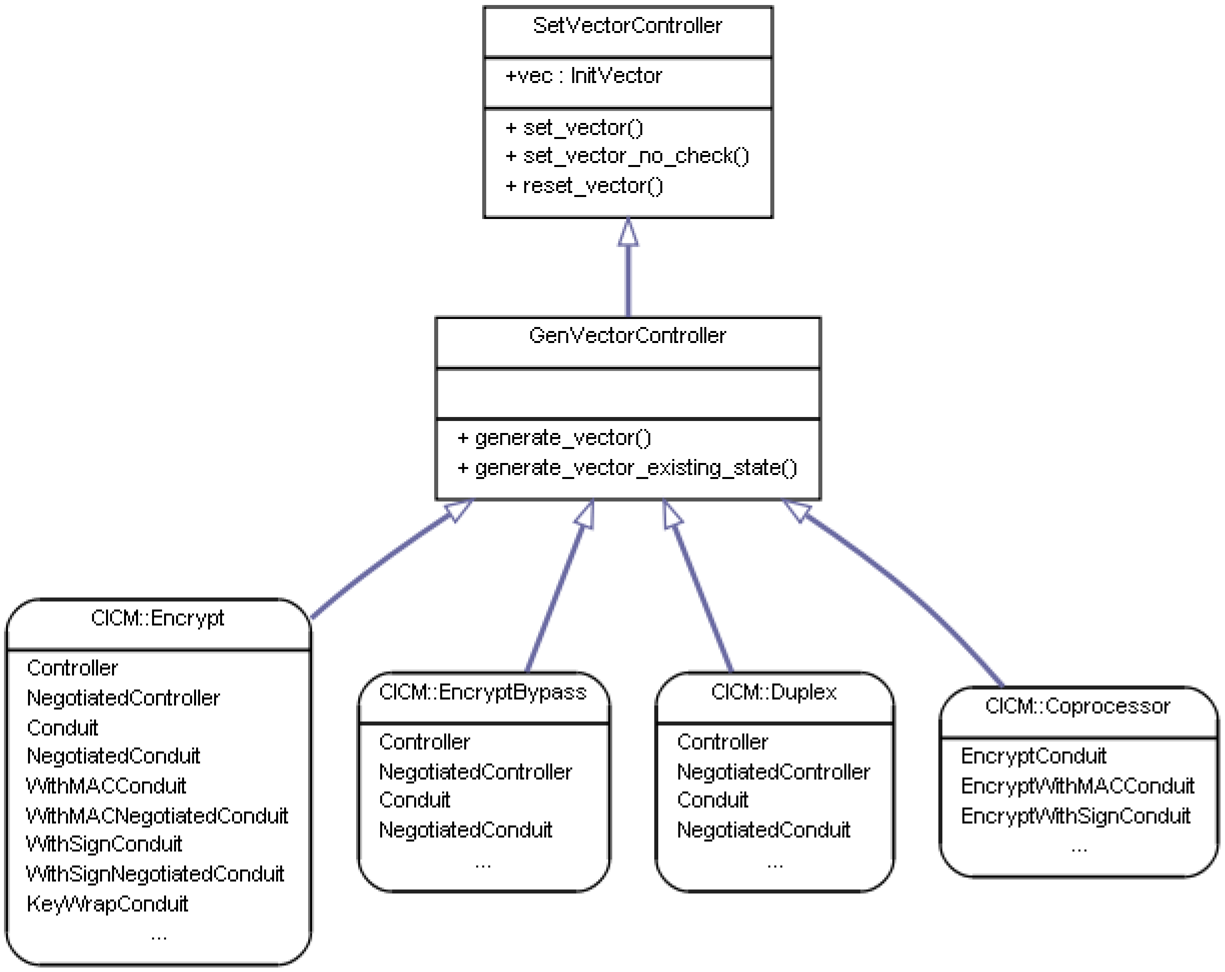


Figure 28. Interface Inheritance Diagram for GenVectorController

3.5.4.6.1 CICM::GenVectorController Inheritance

CICM::GenVectorController inherits from: CICM::SetVectorController.

3.5.4.6.2 CICM::GenVectorController Methods

Method CICM::GenVectorController::generate_vector()

```
        CICM::Status generate_vector();
```

Generate a vector for this controller utilizing new random state.

Remarks:

It may not be necessary for a client program to explicitly generate a vector. Some cryptographic modules will implicitly generate an IV, for example, as a byproduct of channel creation. In other cases, an existing vector will be associated with a controller by calling the CICM::SetVectorController::set_vector method.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_INSUFFICIENT_ENTROPY

Method CICM::GenVectorController::generate_vector_existing_state()

```
        CICM::Status generate_vector_existing_state();
```

Generate a vector for this controller utilizing the latest unused state.

Remarks:

It may not be necessary for a client program to explicitly generate a vector. Some cryptographic modules will implicitly generate an IV, for example, as a byproduct of channel creation. In other cases, an existing vector will be associated with a controller by calling the CICM::SetVectorController::set_vector method.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_INSUFFICIENT_ENTROPY

3.5.4.7 Interface CICM::ResyncController

Interface CICM::ResyncController

```
interface ResyncController : CICM::Controller {
```

Provides methods to resynchronize a controller or conduit.

3.5.4.7.1 CICM::ResyncController Inheritance

CICM::ResyncController inherits from: CICM::Controller.

3.5.4.7.2 CICM::ResyncController Methods

Method CICM::ResyncController::resync()

```
        CICM::Status resync();
```

Resynchronize the channel.

Remarks:

A client program-initiated resync is required when the host is responsible for ensuring cryptographic synchronization is maintained because the operating mode used does not enable the module to determine that it is out of sync. The action taken by the module as the result of a call to this method will differ based upon characteristics of the cryptographic algorithm, communications path framing, and details of the protocol used to achieve cryptographic synchronization between two modules.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

See also:

CICM::ResyncController::resync_with_sync_vector for the version of this method that accepts a synchronization vector.

Method CICM::ResyncController::resync_with_sync_vector()

```
        CICM::Status resync_with_sync_vector(
                in  CICM::Vector vec
        );
```

Resynchronize the channel, using the specified synchronization vector (required by certain operating modes to initiate a resync).

Remarks:

A client program-initiated resync is required when the host is responsible for ensuring cryptographic synchronization is maintained because the operating mode used does not enable the module to determine that it is out of sync. The action taken by the module as the result of a call to this method will differ based upon characteristics of the cryptographic algorithm, communications path framing, and details of the protocol used to achieve cryptographic synchronization among modules.

Parameters:

`[in]` *vec* Synchronization vector to use to resynchronize the channel.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_INVALID_VECTOR, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

See also:

CICM::ResyncController::resync for the version of this method that does not require a synchronization vector.

3.5.5 CHANNEL NEGOTIATION

3.5.5.1 Negotiating Channels and Controllers

When creating an encryption or decryption channel using an asymmetric keyset, a negotiation process must be initiated between the two communicating entities, resulting in an ephemeral symmetric key held by each entity. The following details the steps in the negotiation process:

1. Retrieve a negotiator. The client program uses the CICM::ChannelManager to create the appropriate CICM::Negotiator, thus initiating the negotiation.
2. Validate remote peer. Most systems will use a trust anchor to validate that the remote peer is legitimate and will further verify the peer appears on the appropriate access control list(s). In some cases, the above validation will be sufficient. In others, it will be necessary to display information about the remote peer to a human user to receive a positive response from the user that the displayed peer is the expected peer. This validation procedure differs depending upon the display configuration:
    - If the system is configured with a trusted display, information about the remote peer is displayed to the trusted display and the user is solicited for a response which is returned to the module. The trusted display interactions take place independent of the API.
    - If the system is not configured with a trusted display, the client program uses CICM::Negotiator::get_remote_info to retrieve information about the remote peer and then displays this information to the user (independent of CICM) to allow the user to determine if this is the expected remote peer; the client program calls CICM::Negotiator::abort_negotiation to abort the negotiation if the user rejects the remote peer.
3. Complete negotiation. The client program explicitly completes the negotiation using the negotiator's `complete()` method.

A successful negotation results in a negotiated controller.

3.5.5.2 Interface CICM::Negotiator

Interface CICM::Negotiator
```
interface Negotiator {
```

CICM::Negotiator is an abstraction inherited by controllers and CICM::KeyProtocolSender to assist in the management of the negotiation process.

3.5.5.2.1 CICM::Negotiator Methods

Method CICM::Negotiator::get_remote_info()
```
        CICM::Status get_remote_info(
                out     CICM::PeerInfo peer_info
        );
```

Method CICM::Negotiator::abort_negotiation()
```
        CICM::Status abort_negotiation();
```

3.5.5.3 Interface CICM::PeerInfo

Interface CICM::PeerInfo

```
interface PeerInfo {
```

Information about a peer entity participating in a key negotiation.

3.5.5.3.1 CICM::PeerInfo Attributes

Attribute CICM::PeerInfo::peer_name

```
        readonly attribute CICM::CharString peer_name;
```

Name/organization of remote entity participating in key agreement prototcol.

Attribute CICM::PeerInfo::classification

```
        readonly attribute CICM::Classification classification;
```

Highest security classification level at which the remote entity participating in the key agreement protocol is capable of communicating.

Attribute CICM::PeerInfo::compartment

```
        readonly attribute CICM::CharString compartment;
```

Compartment of remote entity participating in key agreement protocol.

Attribute CICM::PeerInfo::message

```
        readonly attribute CICM::CharString message;
```

Message to be displayed regarding the remote entities' participation in key agreement protocol.

3.5.6 ENCRYPTION CHANNEL MANAGEMENT

Namespace CICM::Encrypt

```
module Encrypt {
```

The CICM::Encrypt namespace contains interfaces that support encryption operations between two independent security domains.

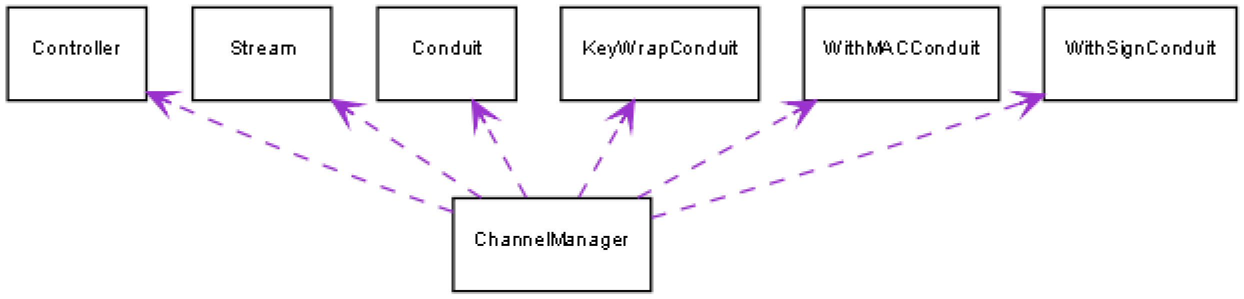


Figure 29. Interface Relationship Diagram for Encryption Channels

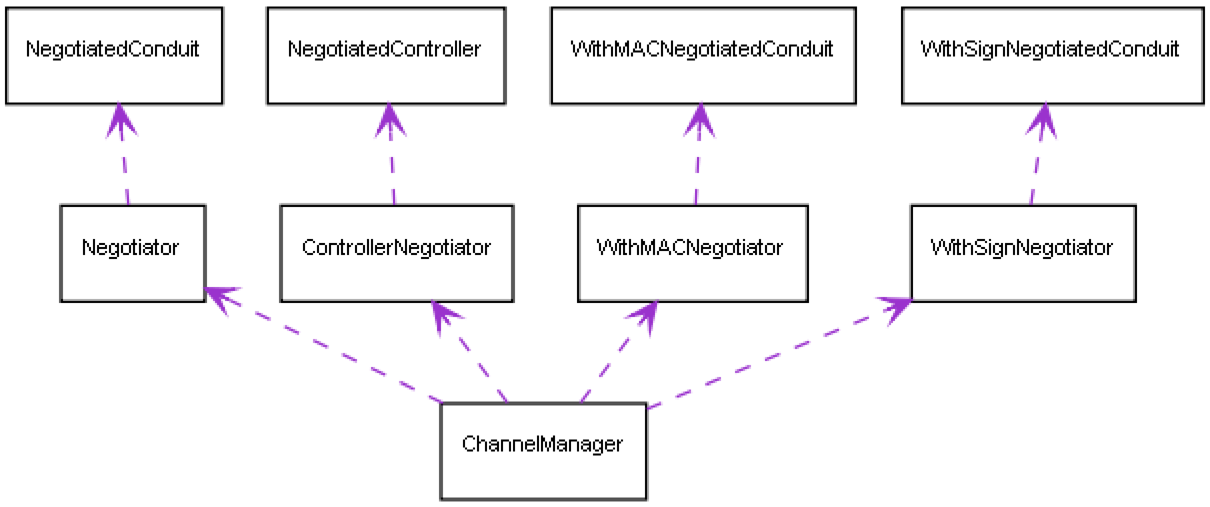


Figure 30. Interface Relationship Diagram for Negotiated Encryption Channels

3.5.6.1 Interface CICM::Encrypt::ChannelManager

Interface CICM::Encrypt::ChannelManager

```
interface ChannelManager {
```

CICM::Encrypt::ChannelManager is an abstraction inherited by CICM::ChannelManager that supports the creation of encryption negotiators, conduits, controllers, and streams. See CICM::ChannelManager for additional information.

3.5.6.1.1 CICM::Encrypt::ChannelManager Methods

Method CICM::Encrypt::ChannelManager::negotiate_encrypt_conduit()

```
        CICM::Status negotiate_encrypt_conduit(
                in  CICM::RemotePort remote_port,
                in  CICM::ProtocolId protocol,
                in  CICM::AsymKey key_ref,
                out CICM::Encrypt::Negotiator negotiator_ref
        );
```

Initiate a negotiation to establish a shared key with a peer. The channel that results will encrypt a stream of data.

Parameters:

[in]	*remote_port*	Remote port.
[in]	*protocol*	Protocol identifier.
[in]	*key_ref*	Reference to negotiation key.
[out]	*negotiator_ref*	Reference to resulting negotiator.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_PROTO_INVALID, S_PROTO_INCOMPATIBLE, S_PROTO_UNDETERMINED, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Encrypt::ChannelManager::negotiate_encrypt_with_mac_conduit()

```
CICM::Status negotiate_encrypt_with_mac_conduit(
        in  CICM::RemotePort remote_port,
        in  CICM::ProtocolId protocol,
        in  CICM::SymKey mac_key_ref,
        in  CICM::AsymKey nego_key_ref,
        in  CICM::SymMacAlgorithmId mac_algorithm,
        out CICM::Encrypt::WithMACNegotiator negotiator_ref
);
```

Initiate a negotiation to establish a shared key with a peer. The channel that results will MAC and encrypt a stream of data.

Remarks:

Negotiation applies only to the negotiation key, not the MAC key.

Parameters:

[in]	*remote_port*	Remote port.
[in]	*protocol*	Protocol identifier.
[in]	*mac_key_ref*	Reference to MAC key.
[in]	*nego_key_ref*	Reference to negotiation key.
[in]	*mac_algorithm*	MAC algorithm.
[out]	*negotiator_ref*	Reference to resulting negotiator.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_PROTO_INVALID, S_PROTO_INCOMPATIBLE, S_PROTO_UNDETERMINED, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Encrypt::ChannelManager::negotiate_encrypt_with_sign_conduit()

```
CICM::Status negotiate_encrypt_with_sign_conduit(
        in  CICM::RemotePort remote_port,
        in  CICM::ProtocolId protocol,
        in  CICM::AsymKey sign_key_ref,
        in  CICM::AsymKey nego_key_ref,
        in  CICM::AsymSigAlgorithmId sign_algorithm,
        out CICM::Encrypt::WithSignNegotiator negotiator_ref
);
```

Initiate a negotiation to establish a shared key with a peer. The channel that results will sign and encrypt a stream of data.

Remarks:

Negotiation applies only to the negotiation key, not the signature key.

Parameters:

`[in]`	*remote_port*	Remote port.
`[in]`	*protocol*	Protocol identifier.
`[in]`	*sign_key_ref*	Reference to signature key.
`[in]`	*nego_key_ref*	Reference to negotiation key.
`[in]`	*sign_algorithm*	Signature algorithm.
`[out]`	*negotiator_ref*	Reference to resulting negotiator.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_PROTO_INVALID, S_PROTO_INCOMPATIBLE, S_PROTO_UNDETERMINED, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Encrypt::ChannelManager::negotiate_encrypt_controller()

```
        CICM::Status negotiate_encrypt_controller(
                in  CICM::LocalPort local_port,
                in  CICM::RemotePort remote_port,
                in  CICM::ProtocolId protocol,
                in  CICM::AsymKey key_ref,
                out CICM::Encrypt::ControllerNegotiator negotiator_ref
        );
```

Initiate a negotiation to establish a shared key with a peer, resulting in a controller to manage an encryption channel.

Parameters:

`[in]`	*local_port*	Local port.
`[in]`	*remote_port*	Remote port.
`[in]`	*protocol*	Protocol identifier.
`[in]`	*key_ref*	Reference to encryption key.
`[out]`	*negotiator_ref*	Reference to resulting negotiator.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_PROTO_INVALID, S_PROTO_INCOMPATIBLE, S_PROTO_UNDETERMINED, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Encrypt::ChannelManager::create_encrypt_conduit()

```
CICM::Status create_encrypt_conduit(
        in  CICM::RemotePort remote_port,
        in  CICM::SymKey key_ref,
        in  CICM::SymEncrAlgorithmId algorithm,
        out CICM::Encrypt::Conduit conduit_ref
);
```

Create conduit to encrypt a stream of data.

Parameters:

`[in]`	*remote_port*	Remote port.
`[in]`	*key_ref*	Reference to encryption key.
`[in]`	*algorithm*	Encryption algorithm/mode.
`[out]`	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Encrypt::ChannelManager::create_encrypt_with_mac_conduit()

```
CICM::Status create_encrypt_with_mac_conduit(
        in  CICM::RemotePort remote_port,
        in  CICM::SymKey mac_key_ref,
        in  CICM::SymKey encrypt_key_ref,
        in  CICM::SymMacAlgorithmId mac_algorithm,
        in  CICM::SymEncrAlgorithmId encr_algorithm,
        out CICM::Encrypt::WithMACConduit conduit_ref
);
```

Create conduit to MAC and encrypt a stream of data.

Parameters:

[in]	*remote_port*	Remote port.
[in]	*mac_key_ref*	Reference to MAC key.
[in]	*encrypt_key_ref*	Reference to encryption key.
[in]	*mac_algorithm*	MAC algorithm.
[in]	*encr_algorithm*	Encryption algorithm/mode.
[out]	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Encrypt::ChannelManager::create_encrypt_with_sign_conduit()

```
CICM::Status create_encrypt_with_sign_conduit(
        in  CICM::RemotePort remote_port,
        in  CICM::AsymKey sign_key_ref,
        in  CICM::SymKey encrypt_key_ref,
        in  CICM::AsymSigAlgorithmId sign_algorithm,
        in  CICM::SymEncrAlgorithmId encr_algorithm,
        out CICM::Encrypt::WithSignConduit conduit_ref
);
```

Create conduit to sign and encrypt a stream of data.

Parameters:

[in]	*remote_port*	Remote port.
[in]	*sign_key_ref*	Reference to signature key.
[in]	*encrypt_key_ref*	Reference to encryption key.
[in]	*sign_algorithm*	Signature algorithm.
[in]	*encr_algorithm*	Encryption algorithm/mode.
[out]	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Encrypt::ChannelManager::create_key_wrap_conduit()

```
CICM::Status create_key_wrap_conduit(
        in  CICM::RemotePort remote_port,
        in  CICM::SymKey kek_ref,
        in  CICM::KeyWrapAlgorithmId algorithm,
        out CICM::Encrypt::KeyWrapConduit conduit_ref
);
```

Create conduit to wrap a key.

Remarks:

This type of conduit may be used to wrap key material bound for a peer cryptographic module. To wrap individual keys already in the module, use CICM::Symkey::wrap or CICM::Asymkey::wrap.

Parameters:

`[in]`	*remote_port*	Remote port.
`[in]`	*kek_ref*	Reference to key encryption key.
`[in]`	*algorithm*	Key wrapping algorithm/mode.
`[out]`	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Encrypt::ChannelManager::create_encrypt_controller()

```
        CICM::Status create_encrypt_controller(
                in  CICM::LocalPort local_port,
                in  CICM::RemotePort remote_port,
                in  CICM::SymKey key_ref,
                in  CICM::SymEncrAlgorithmId algorithm,
                out CICM::Encrypt::Controller controller_ref
        );
```

Create controller to configure and control an encryption channel.

Remarks:

In some cases, hosts may depend upon separate processes to control and use a channel. This method returns the channel controller and must be called before the corresponding stream is retrieved.

Parameters:

`[in]`	*local_port*	Local port.
`[in]`	*remote_port*	Remote port.
`[in]`	*key_ref*	Reference to encryption key.
`[in]`	*algorithm*	Encryption algorithm/mode.
`[out]`	*controller_ref*	Reference to resulting controller.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Encrypt::ChannelManager::get_encrypt_stream()

```
        CICM::Status get_encrypt_stream(
                in  CICM::LocalPort local_port,
                out CICM::Encrypt::Stream stream_ref
        );
```

Create stream associated with previously created controller to accept data for transformation.

Parameters:

`[in]` *local_port* Local port.

`[out]` *stream_ref* Reference to resulting stream.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_NOT_FOUND

3.5.6.2 Interface CICM::Encrypt::Stream

Interface CICM::Encrypt::Stream

```
interface Stream : CICM::WriteStream {
```

CICM::Encrypt::Stream supports encryption operations between two independent security domains. The resulting stream is capable of accepting data for transformation, but not managing the channel. It is created by calling CICM::ChannelManager::get_encrypt_stream.

3.5.6.2.1 CICM::Encrypt::Stream Inheritance

CICM::Encrypt::Stream inherits from: CICM::WriteStream.

3.5.6.2.2 CICM::Encrypt::Stream Methods

Method CICM::Encrypt::Stream::encrypt()

```
        CICM::Status encrypt(
                in  CICM::Buffer buffer
        );
```

Sends data to the module to be encrypted. The method blocks until data is sent.

Parameters:

`[in]` *buffer* Plaintext to encrypt.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_WRAPPED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_IO_ERROR, S_CHANNEL_DATA_INVALID, S_CHANNEL_DATA_INVALID_LEN

Method CICM::Encrypt::Stream::encrypt_non_blocking()

```
        CICM::Status encrypt_non_blocking(
                in  CICM::Buffer buffer,
                in  CICM::TransId transaction_id
        );
```

Registers a buffer of data to be sent to the module for encryption and then immediately returns control to the caller. The length of the data is encapsulated in the buffer parameter. The caller may use the CICM::Encrypt::Stream::encrypt_poll method to proactively poll the channel to determine the status of the operation. The caller is responsible for maintaining any necessary metadata associated with the transaction_id parameter. Memory responsibilities and calling conventions shall follow the appropriate IDL language mapping conventions.

Parameters:

`[in]`	*buffer*	Plaintext to encrypt.
`[in]`	*transaction_id*	Unique transaction id that will be used by the CICM::Encrypt::Stream::encrypt_poll method to determine to which buffer the poll status applies.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_WRAPPED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_IO_ERROR, S_CHANNEL_DATA_INVALID, S_CHANNEL_DATA_INVALID_LEN

Method CICM::Encrypt::Stream::encrypt_poll()

```
        CICM::Status encrypt_poll(
                in  CICM::TransId transaction_id,
                out CICM::WriteStream::WriteStatus status
        );
```

Returns the status of the non-blocking encryption operation specified by the transaction_id parameter. Memory responsibilities and calling conventions shall follow the appropriate IDL language mapping conventions.

Parameters:

`[in]`	*transaction_id*	Unique transaction id previously specified to the CICM::Encrypt::Stream::encrypt_non_blocking method that allows the poll operation to determine to which buffer the poll status applies.
`[out]`	*status*	Status of the non-blocking operation corresponding to the transaction_id parameter.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_WRAPPED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_IO_ERROR

3.5.6.3 Interface CICM::Encrypt::KeyWrapStream

Interface CICM::Encrypt::KeyWrapStream

```
interface KeyWrapStream : CICM::Stream {
```

CICM::Encrypt::KeyWrapStream is an abstraction that allows key material to be presented to a stream for wrapping prior to passing into a different security domain.

3.5.6.3.1 CICM::Encrypt::KeyWrapStream Inheritance

CICM::Encrypt::KeyWrapStream inherits from: CICM::Stream.

3.5.6.3.2 CICM::Encrypt::KeyWrapStream Methods

Method CICM::Encrypt::KeyWrapStream::wrap_key()

```
        CICM::Status wrap_key(
                in CICM::Key key_ref
        );
```

Write the key to be wrapped to the channel stream.

Remarks:

The method blocks until the key has been written.

Parameters:

`[in]` *key_ref* Reference to key to be wrapped.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_CLASSIFICATION, S_KEY_WRAPPED, S_KEY_NOT_WRAPPABLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_IO_ERROR

3.5.6.4 Interface CICM::Encrypt::Controller

Interface CICM::Encrypt::Controller

```
interface Controller :
        CICM::MultiDomainController,
        CICM::SymKeyController,
        CICM::GenVectorController,
        CICM::ResyncController {
```

CICM::Encrypt::Controller supports encryption operations between two independent security domains. The resulting controller is capable of managing the channel, but not accepting data for transformation. It is created by calling CICM::ChannelManager::create_encrypt_controller.

3.5.6.4.1 CICM::Encrypt::Controller Inheritance

CICM::Encrypt::Controller inherits from: CICM::MultiDomainController, CICM::SymKeyController, CICM::GenVectorController and CICM::ResyncController.

3.5.6.5 Interface CICM::Encrypt::NegotiatedController

Interface CICM::Encrypt::NegotiatedController

```
interface NegotiatedController :
        CICM::NegotiatedController,
        CICM::GenVectorController,
        CICM::ResyncController {
```

CICM::Encrypt::NegotiatedController is the negotiated version of CICM::Encrypt::Controller. It is the result of a successful negotiation by CICM::Encrypt::ControllerNegotiator.

3.5.6.5.1 CICM::Encrypt::NegotiatedController Inheritance

CICM::Encrypt::NegotiatedController inherits from: CICM::NegotiatedController, CICM::GenVectorController and CICM::ResyncController.

3.5.6.6 Interface CICM::Encrypt::Conduit

Interface CICM::Encrypt::Conduit

```
interface Conduit :
        CICM::Conduit,
        CICM::Encrypt::Controller,
        CICM::Encrypt::Stream {
```

CICM::Encrypt::Conduit supports encryption operations between two independent security domains. The resulting conduit is capable of both managing the channel and accepting data for transformation. It is created by calling CICM::ChannelManager::create_encrypt_conduit.

3.5.6.6.1 CICM::Encrypt::Conduit Inheritance

CICM::Encrypt::Conduit inherits from: CICM::Conduit, CICM::Encrypt::Controller and CICM::Encrypt::Stream.

3.5.6.7 Interface CICM::Encrypt::NegotiatedConduit

Interface CICM::Encrypt::NegotiatedConduit

```
interface NegotiatedConduit :
        CICM::Conduit,
        CICM::Encrypt::NegotiatedController,
        CICM::Encrypt::Stream {
```

CICM::Encrypt::NegotiatedConduit is the negotiated version of CICM::Encrypt::Conduit. It is the result of a successful negotiation by CICM::Encrypt::Negotiator.

3.5.6.7.1 CICM::Encrypt::NegotiatedConduit Inheritance

CICM::Encrypt::NegotiatedConduit inherits from: CICM::Conduit, CICM::Encrypt::NegotiatedController and CICM::Encrypt::Stream.

3.5.6.8 Interface CICM::Encrypt::WithMACConduit

Interface CICM::Encrypt::WithMACConduit

```
interface WithMACConduit :
        CICM::AbstractMACConduit,
        CICM::Encrypt::Conduit {
```

CICM::Encrypt::WithMACConduit supports encryption operations between two independent security domains with the receipt of a MAC value in the initiating domain. The resulting conduit is capable of both managing the channel and accepting data for transformation. It is created by calling CICM::ChannelManager::create_encrypt_with_mac_conduit.

3.5.6.8.1 CICM::Encrypt::WithMACConduit Inheritance

CICM::Encrypt::WithMACConduit inherits from: CICM::AbstractMACConduit and CICM::Encrypt::Conduit.

3.5.6.9 Interface CICM::Encrypt::WithMACNegotiatedConduit

Interface CICM::Encrypt::WithMACNegotiatedConduit

```
interface WithMACNegotiatedConduit :
        CICM::AbstractMACConduit,
        CICM::Encrypt::NegotiatedConduit {
```

CICM::Encrypt::WithMACNegotiatedConduit is the negotiated version of CICM::Encrypt::WithMACConduit. It is the result of a successful negotiation by CICM::Encrypt::WithMACNegotiator.

3.5.6.9.1 CICM::Encrypt::WithMACNegotiatedConduit Inheritance

CICM::Encrypt::WithMACNegotiatedConduit inherits from: CICM::AbstractMACConduit and CICM::Encrypt::NegotiatedConduit.

3.5.6.10 Interface CICM::Encrypt::WithSignConduit

Interface CICM::Encrypt::WithSignConduit

```
interface WithSignConduit :
        CICM::AbstractSignConduit,
        CICM::Encrypt::Conduit {
```

CICM::Encrypt::WithSignConduit supports encryption operations between two independent security domains with the receipt of a signature value in the initiating domain. The resulting conduit is capable of both managing the channel and accepting data for transformation. It is created by calling CICM::ChannelManager::create_encrypt_with_sign_conduit.

3.5.6.10.1 CICM::Encrypt::WithSignConduit Inheritance

CICM::Encrypt::WithSignConduit inherits from: CICM::AbstractSignConduit and CICM::Encrypt::Conduit.

3.5.6.11 Interface CICM::Encrypt::WithSignNegotiatedConduit

Interface CICM::Encrypt::WithSignNegotiatedConduit

```
interface WithSignNegotiatedConduit :
        CICM::AbstractSignConduit,
        CICM::Encrypt::NegotiatedConduit {
```

CICM::Encrypt::WithSignNegotiatedConduit is the negotiated version of CICM::Encrypt::WithSignConduit. It is the result of a successful negotiation by CICM::Encrypt::WithSignNegotiator.

3.5.6.11.1 CICM::Encrypt::WithSignNegotiatedConduit Inheritance

CICM::Encrypt::WithSignNegotiatedConduit inherits from: CICM::AbstractSignConduit and CICM::Encrypt::NegotiatedConduit.

3.5.6.12 Interface CICM::Encrypt::KeyWrapConduit

Interface CICM::Encrypt::KeyWrapConduit

```
interface KeyWrapConduit :
        CICM::Encrypt::Controller,
        CICM::Encrypt::KeyWrapStream {
```

CICM::Encrypt::KeyWrapConduit supports key wrapping operations between two independent security domains. The resulting conduit is capable of both managing the channel and accepting keys for transformation. It is created by calling CICM::ChannelManager::create_key_wrap_conduit.

3.5.6.12.1 CICM::Encrypt::KeyWrapConduit Inheritance

CICM::Encrypt::KeyWrapConduit inherits from: CICM::Encrypt::Controller and CICM::Encrypt::KeyWrapStream.

3.5.6.13 Interface CICM::Encrypt::ControllerNegotiator

Interface CICM::Encrypt::ControllerNegotiator

```
interface ControllerNegotiator : CICM::Negotiator {
```

CICM::Encrypt::ControllerNegotiator initiates a negotiation to establish a shared key with a remote entity that is used to support encryption operations between two independent security domains. The result of a successful negotiation is a CICM::Encrypt::NegotiatedController which is capable of managing the channel, but not accepting data for transformation. CICM::Encrypt::ControllerNegotiator is created by calling CICM::ChannelManager::negotiate_encrypt_controller.

3.5.6.13.1 CICM::Encrypt::ControllerNegotiator Inheritance

CICM::Encrypt::ControllerNegotiator inherits from: CICM::Negotiator.

3.5.6.13.2 CICM::Encrypt::ControllerNegotiator Methods

Method CICM::Encrypt::ControllerNegotiator::complete()

```
        CICM::Status complete(
                out     CICM::Encrypt::NegotiatedController controller_ref
        );
```

Complete negotiation and retrieve a negotiated negotiated encrypt controller.

Parameters:

`[out]` *controller_ref* Reference to resulting controller.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_NOT_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

3.5.6.14 Interface CICM::Encrypt::Negotiator

Interface CICM::Encrypt::Negotiator

```
interface Negotiator : CICM::Negotiator {
```

CICM::Encrypt::Negotiator initiates a negotiation to establish a shared key with a remote entity that is used to support encryption operations between two independent security domains. The result of a successful negotiation is a CICM::Encrypt::NegotiatedConduit which is capable of both managing the channel and accepting data for transformation. CICM::Negotiator is created by calling CICM::Encrypt::ChannelManager::negotiate_encrypt_conduit.

3.5.6.14.1 CICM::Encrypt::Negotiator Inheritance

CICM::Encrypt::Negotiator inherits from: CICM::Negotiator.

3.5.6.14.2 CICM::Encrypt::Negotiator Methods

Method CICM::Encrypt::Negotiator::complete()

```
        CICM::Status complete(
                out CICM::Encrypt::NegotiatedConduit conduit_ref
        );
```

Complete negotiation and retrieve a negotiated encrypt conduit.

Parameters:

`[out]` *conduit_ref* Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_NOT_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

3.5.6.15 Interface CICM::Encrypt::WithMACNegotiator

Interface CICM::Encrypt::WithMACNegotiator
```
interface WithMACNegotiator : CICM::Negotiator {
```
CICM::Encrypt::WithMACNegotiator initiates a negotiation to establish a shared key with a remote entity that is used to support encryption operations between two independent security domains. Additionally, a message authentication code is received in the initiating domain. The result of a successful negotiation is a CICM::Encrypt::WithMACNegotiatedConduit which is capable of both managing the channel and accepting data for transformation. CICM::Encrypt::WithMACNegotiator is created by calling CICM::ChannelManager::negotiate_encrypt_with_mac_conduit.

3.5.6.15.1 CICM::Encrypt::WithMACNegotiator Inheritance

CICM::Encrypt::WithMACNegotiator inherits from: CICM::Negotiator.

3.5.6.15.2 CICM::Encrypt::WithMACNegotiator Methods

Method CICM::Encrypt::WithMACNegotiator::complete()
```
        CICM::Status complete(
                out CICM::Encrypt::WithMACNegotiatedConduit conduit_ref
        );
```
Complete negotiation and retrieve a negotiated MAC encrypt conduit.
Parameters:
`[out]` *conduit_ref* Reference to resulting conduit.
Returns:
S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_NOT_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

3.5.6.16 Interface CICM::Encrypt::WithSignNegotiator

Interface CICM::Encrypt::WithSignNegotiator
```
interface WithSignNegotiator : CICM::Negotiator {
```
CICM::Encrypt::WithSignNegotiator initiates a negotiation to establish a shared key with a remote entity that is used to support encryption operations between two independent security domains. Additionally, a signature value is received in the initiating domain. The result of a successful negotiation is a CICM::Encrypt::WithSignNegotiatedConduit which is capable of both managing the channel and

accepting data for transformation. CICM::Encrypt::WithSignNegotiator is created by calling CICM::ChannelManager::negotiate_encrypt_with_sign_conduit.

3.5.6.16.1 CICM::Encrypt::WithSignNegotiator Inheritance

CICM::Encrypt::WithSignNegotiator inherits from: CICM::Negotiator.

3.5.6.16.2 CICM::Encrypt::WithSignNegotiator Methods

Method CICM::Encrypt::WithSignNegotiator::complete()

```
        CICM::Status complete(
                out CICM::Encrypt::WithSignNegotiatedConduit conduit_ref
        );
```

Complete negotiation and retrieve a negotiated sign encrypt conduit.

Parameters:

`[out]` *conduit_ref* Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_NOT_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

3.5.7 DECRYPTION CHANNEL MANAGEMENT

Namespace CICM::Decrypt

```
module Decrypt {
```

The CICM::Decrypt namespace contains interfaces that support decryption operations between two independent security domains.

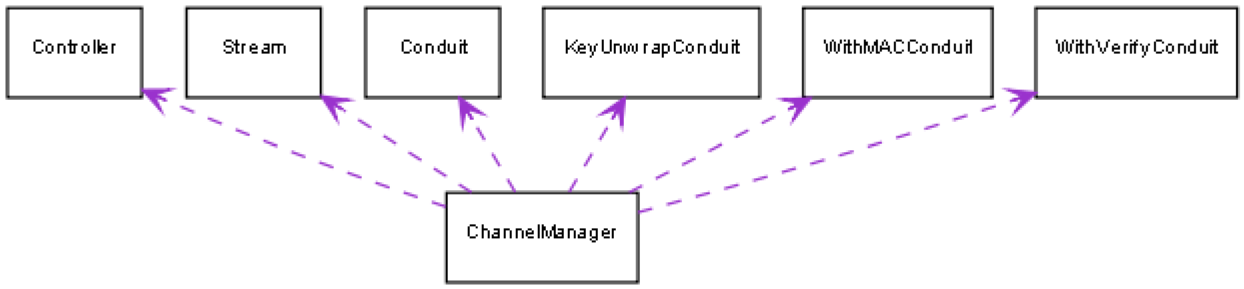


Figure 31. Interface Relationship Diagram for Decryption Channels

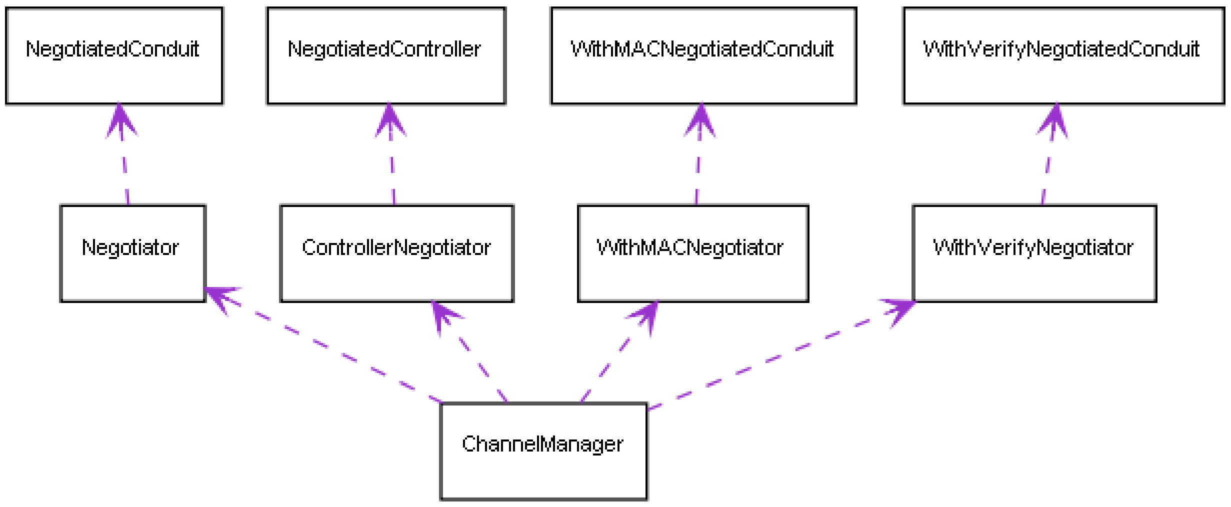


Figure 32. Interface Relationship Diagram for Negotiated Decryption Channels

3.5.7.1 Interface CICM::Decrypt::ChannelManager

Interface CICM::Decrypt::ChannelManager

```
interface ChannelManager {
```

CICM::Decrypt::ChannelManager is an abstraction inherited by CICM::ChannelManager that supports the creation of decryption negotiators, conduits, controllers, and streams. See CICM::ChannelManager for additional information.

3.5.7.1.1 CICM::Decrypt::ChannelManager Methods

Method CICM::Decrypt::ChannelManager::negotiate_decrypt_conduit()

```
        CICM::Status negotiate_decrypt_conduit(
                in  CICM::RemotePort remote_port,
                in  CICM::ProtocolId protocol,
                in  CICM::AsymKey key_ref,
                out CICM::Decrypt::Negotiator negotiator_ref
        );
```

Creates a negotiator that, upon successful negotiation, results in a CICM::Decrypt::NegotiatedConduit.

Parameters:

[in]	*remote_port*	Remote port.
[in]	*protocol*	Protocol identifier.
[in]	*key_ref*	Reference to negotiation key.
[out]	*negotiator_ref*	Reference to resulting negotiator.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_PROTO_INVALID, S_PROTO_INCOMPATIBLE, S_PROTO_UNDETERMINED, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Decrypt::ChannelManager::negotiate_decrypt_with_mac_conduit()

```
        CICM::Status negotiate_decrypt_with_mac_conduit(
                in  CICM::RemotePort remote_port,
                in  CICM::ProtocolId protocol,
                in  CICM::SymKey verify_key_ref,
                in  CICM::AsymKey nego_key_ref,
                in  CICM::SymMacAlgorithmId verify_algorithm,
                out CICM::Decrypt::WithMACNegotiator negotiator_ref
        );
```

Creates a negotiator that, upon successful negotiation, results in a CICM::Decrypt::MACNegotiatedConduit.

Remarks:

Negotiation applies only to the negotiation key, not the MAC key.

Parameters:

`[in]`	*remote_port*	Remote port.
`[in]`	*protocol*	Protocol identifier.
`[in]`	*verify_key_ref*	Reference to verification key.
`[in]`	*nego_key_ref*	Reference to negotiation key.
`[in]`	*verify_algorithm*	Verification algorithm.
`[out]`	*negotiator_ref*	Reference to resulting negotiator.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_PROTO_INVALID, S_PROTO_INCOMPATIBLE, S_PROTO_UNDETERMINED, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Decrypt::ChannelManager::negotiate_decrypt_with_verify_conduit()

```
        CICM::Status negotiate_decrypt_with_verify_conduit(
                in  CICM::RemotePort remote_port,
                in  CICM::ProtocolId protocol,
                in  CICM::AsymKey verify_key_ref,
                in  CICM::AsymKey nego_key_ref,
                in  CICM::AsymSigAlgorithmId verify_algorithm,
                out CICM::Decrypt::WithVerifyNegotiator negotiator_ref
        );
```

Initiate a negotiation to establish a shared key with a peer. The channel that results will decrypt and verify a stream of data.

Remarks:

Negotiation applies only to the negotiation key, not the signature verification key.

Parameters:

`[in]`	*remote_port*	Remote port.
`[in]`	*protocol*	Protocol identifier.
`[in]`	*verify_key_ref*	Reference to verification key.
`[in]`	*nego_key_ref*	Reference to negotiation key.
`[in]`	*verify_algorithm*	Verification algorithm.
`[out]`	*negotiator_ref*	Reference to resulting negotiator.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_PROTO_INVALID, S_PROTO_INCOMPATIBLE, S_PROTO_UNDETERMINED, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Decrypt::ChannelManager::negotiate_decrypt_controller()

```
        CICM::Status negotiate_decrypt_controller(
                in  CICM::LocalPort local_port,
                in  CICM::RemotePort remote_port,
                in  CICM::ProtocolId protocol,
                in  CICM::AsymKey key_ref,
                out CICM::Decrypt::ControllerNegotiator negotiator_ref
        );
```

Initiate a negotiation to establish a shared key with a peer, resulting in a controller to manage a decryption channel.

Parameters:

`[in]`	*local_port*	Local port.
`[in]`	*remote_port*	Remote port.
`[in]`	*protocol*	Protocol identifier.
`[in]`	*key_ref*	Reference to negotiation key.
`[out]`	*negotiator_ref*	Reference to resulting negotiator.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE,

S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_PROTO_INVALID, S_PROTO_INCOMPATIBLE, S_PROTO_UNDETERMINED, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Decrypt::ChannelManager::create_decrypt_conduit()

```
CICM::Status create_decrypt_conduit(
        in  CICM::RemotePort remote_port,
        in  CICM::SymKey key_ref,
        in  CICM::SymEncrAlgorithmId algorithm,
        out CICM::Decrypt::Conduit conduit_ref
);
```

Create channel to decrypt a stream of data.

Parameters:

`[in]`	*remote_port*	Remote port.
`[in]`	*key_ref*	Reference to decryption key.
`[in]`	*algorithm*	Decryption algorithm/mode.
`[out]`	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Decrypt::ChannelManager::create_decrypt_with_mac_conduit()

```
CICM::Status create_decrypt_with_mac_conduit(
        in  CICM::RemotePort remote_port,
        in  CICM::SymKey verify_key_ref,
        in  CICM::SymKey decrypt_key_ref,
        in  CICM::SymMacAlgorithmId verify_algorithm,
        in  CICM::SymEncrAlgorithmId decrypt_algorithm,
        out CICM::Decrypt::WithMACConduit conduit_ref
);
```

Create channel to MAC verify and decrypt a stream of data.

Parameters:

`[in]`	*remote_port*	Remote port.
`[in]`	*verify_key_ref*	Reference to verification key.
`[in]`	*decrypt_key_ref*	Reference to decryption key.

`[in]`	*verify_algorithm*	Verification algorithm.
`[in]`	*decrypt_algorithm*	Decryption algorithm/mode.
`[out]`	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Decrypt::ChannelManager::create_decrypt_with_verify_conduit()

```
CICM::Status create_decrypt_with_verify_conduit(
        in  CICM::RemotePort remote_port,
        in  CICM::AsymKey verify_key_ref,
        in  CICM::SymKey decrypt_key_ref,
        in  CICM::AsymSigAlgorithmId verify_algorithm,
        in  CICM::SymEncrAlgorithmId decrypt_algorithm,
        out CICM::Decrypt::WithVerifyConduit conduit_ref
);
```

Create channel to verify and decrypt a stream of data.

Parameters:

`[in]`	*remote_port*	Remote port.
`[in]`	*verify_key_ref*	Reference to verification key.
`[in]`	*decrypt_key_ref*	Reference to decryption key.
`[in]`	*verify_algorithm*	Verification algorithm.
`[in]`	*decrypt_algorithm*	Decryption algorithm/mode.
`[out]`	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Decrypt::ChannelManager::create_key_unwrap_conduit()

```
CICM::Status create_key_unwrap_conduit(
        in  CICM::RemotePort remote_port,
        in  CICM::SymKey kek_ref,
        in  CICM::KeyWrapAlgorithmId algorithm,
        out CICM::Decrypt::KeyUnwrapConduit conduit_ref
);
```

Create channel to unwrap a key. This type of channel may be used to bulk unwrap key material originating at a key infrastructure component or from a peer cryptographic module. Note that, to unwrap individual keys already in the module, use CICM::Symkey::unwrap or CICM::Asymkey::unwrap.

Parameters:

`[in]`	*remote_port*	Remote port.
`[in]`	*kek_ref*	Reference to key encryption key.
`[in]`	*algorithm*	Key unwrapping algorithm/mode.
`[out]`	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Decrypt::ChannelManager::create_decrypt_controller()

```
    CICM::Status create_decrypt_controller(
            in  CICM::LocalPort local_port,
            in  CICM::RemotePort remote_port,
            in  CICM::SymKey key_ref,
            in  CICM::SymEncrAlgorithmId algorithm,
            out CICM::Decrypt::Controller controller_ref
    );
```

Create controller to configure and control an decryption channel.

Remarks:

In some cases, hosts may depend upon separate processes to control and use a channel. This method returns the channel controller and must be called before the corresponding stream is retrieved.

Parameters:

`[in]`	*local_port*	Local port.
`[in]`	*remote_port*	Remote port.
`[in]`	*key_ref*	Reference to decryption key.
`[in]`	*algorithm*	Decryption algorithm/mode.
`[out]`	*controller_ref*	Reference to resulting controller.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_INVALID, S_KEY_EXPIRED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Decrypt::ChannelManager::get_decrypt_stream()

```
        CICM::Status get_decrypt_stream(
                in  CICM::LocalPort local_port,
                out CICM::Decrypt::Stream stream_ref
        );
```

Create stream associated with previously created controller to receive transformed data.

Parameters:

`[in]` *local_port* Local port.

`[out]` *stream_ref* Reference to resulting stream.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_NOT_FOUND

3.5.7.2 Interface CICM::Decrypt::Stream

Interface CICM::Decrypt::Stream

```
interface Stream : CICM::ReadStream {
```

CICM::Decrypt::Stream supports decryption operations between two independent security domains. The resulting stream is capable of accepting transformed data, but not managing the channel. It is created by calling CICM::ChannelManager::get_decrypt_stream.

3.5.7.2.1 CICM::Decrypt::Stream Inheritance

CICM::Decrypt::Stream inherits from: CICM::ReadStream.

3.5.7.2.2 CICM::Decrypt::Stream Methods

Method CICM::Decrypt::Stream::decrypt()

```
        CICM::Status decrypt(
                out CICM::Buffer buffer
        );
```

Read plaintext data off of decrypt channel stream. The method blocks until data becomes available.

Parameters:

`[out]` *buffer* Plaintext resulting from decryption operation.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_IO_ERROR, S_CHANNEL_BUFFER_LEN, S_INTEGRITY

Method CICM::Decrypt::Stream::decrypt_non_blocking()

```
        CICM::Status decrypt_non_blocking(
                 out CICM::Buffer buffer,
                 in  CICM::TransId transaction_id
         );
```

Registers a buffer into which plaintext resulting from the decryption operation will be copied, and then immediately returns control to the caller. The size of the allocated buffer and length of the resulting plaintext is encapsulated in the buffer parameter. The caller may use the CICM::Decrypt::Stream::decrypt_poll method to proactively poll the channel to determine the status of the operation. The caller is responsible for maintaining any necessary metadata associated with the transaction_id parameter. Memory responsibilities and calling conventions shall follow the appropriate IDL language mapping conventions.

Parameters:

`[out]`	*buffer*	Plaintext resulting from decryption operation.
`[in]`	*transaction_id*	Unique transaction id that will be used by the CICM::Decrypt::Stream::decrypt_poll method to determine to which buffer the poll status applies.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_IO_ERROR, S_CHANNEL_BUFFER_LEN, S_INTEGRITY

Method CICM::Decrypt::Stream::decrypt_poll()

```
         CICM::Status decrypt_poll(
                 in  CICM::TransId transaction_id,
                 out CICM::ReadStream::ReadStatus status
         );
```

Returns the status of the non-blocking decryption operation specified by the transaction_id parameter. Upon completion of the operation, the caller must use the metadata associated with the transaction_id parameter to determine which buffer has been populated. Memory responsibilities and calling conventions shall follow the appropriate IDL language mapping conventions.

Parameters:

`[in]`	*transaction_id*	Unique transaction id previously specified to the CICM::Decrypt::Stream::decrypt_non_blocking method that allows the poll operation to determine to which buffer the poll status applies.
`[out]`	*status*	Status of the non-blocking operation corresponding to the transaction_id parameter.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_WRAPPED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_IO_ERROR

3.5.7.3 Interface CICM::Decrypt::KeyUnwrapStream

Interface CICM::Decrypt::KeyUnwrapStream

```
interface KeyUnwrapStream : CICM::Stream {
```

CICM::Decrypt::KeyUnwrapStream is an abstraction that allows unwrapped key material received from another domain to be retrieved.

3.5.7.3.1 CICM::Decrypt::KeyUnwrapStream Inheritance

CICM::Decrypt::KeyUnwrapStream inherits from: CICM::Stream.

3.5.7.3.2 CICM::Decrypt::KeyUnwrapStream Methods

Method CICM::Decrypt::KeyUnwrapStream::unwrap_sym_key()

```
        CICM::Status unwrap_sym_key(
                out CICM::SymKey key_ref
        );
```

Read one unwrapped symmetric key off of channel stream and return a reference to the key. The method blocks until a key becomes available.

Parameters:

`[in]` *key_ref* Reference to key encryption key.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_IO_ERROR

Method CICM::Decrypt::KeyUnwrapStream::unwrap_asym_key()

```
        CICM::Status unwrap_asym_key(
                out CICM::AsymKey key_ref
        );
```

Read one unwrapped asymmetric key off of channel stream and return a reference to the key. The method blocks until a key becomes available.

Parameters:

`[in]` *key_ref* Reference to key encryption key.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_IO_ERROR

3.5.7.4 Interface CICM::Decrypt::Controller

Interface CICM::Decrypt::Controller

```
interface Controller :
        CICM::MultiDomainController,
        CICM::SymKeyController,
        CICM::SetVectorController,
        CICM::ResyncController {
```

CICM::Decrypt::Controller supports decryption operations between two independent security domains. The resulting controller is capable of managing the channel, but not accepting transformed data. It is created by calling CICM::ChannelManager::create_decrypt_controller.

3.5.7.4.1 CICM::Decrypt::Controller Inheritance

CICM::Decrypt::Controller inherits from: CICM::MultiDomainController, CICM::SymKeyController, CICM::SetVectorController and CICM::ResyncController.

3.5.7.5 Interface CICM::Decrypt::NegotiatedController

Interface CICM::Decrypt::NegotiatedController

```
interface NegotiatedController :
        CICM::NegotiatedController,
        CICM::SetVectorController,
        CICM::ResyncController {
```

CICM::Decrypt::NegotiatedController is the negotiated version of CICM::Decrypt::Controller. It is the result of a successful negotiation by CICM::Decrypt::ControllerNegotiator.

3.5.7.5.1 CICM::Decrypt::NegotiatedController Inheritance

CICM::Decrypt::NegotiatedController inherits from: CICM::NegotiatedController, CICM::SetVectorController and CICM::ResyncController.

3.5.7.6 Interface CICM::Decrypt::Conduit

Interface CICM::Decrypt::Conduit

```
interface Conduit :
        CICM::Conduit,
        CICM::Decrypt::Controller,
        CICM::Decrypt::Stream {
```

CICM::Decrypt::Conduit supports decryption operations between two independent security domains. The resulting conduit is capable of both managing the channel and accepting transformed data. It is created by calling CICM::ChannelManager::create_decrypt_conduit.

3.5.7.6.1 CICM::Decrypt::Conduit Inheritance

CICM::Decrypt::Conduit inherits from: CICM::Conduit, CICM::Decrypt::Controller and CICM::Decrypt::Stream.

3.5.7.7 Interface CICM::Decrypt::NegotiatedConduit

Interface CICM::Decrypt::NegotiatedConduit

```
interface NegotiatedConduit :
        CICM::Conduit,
        CICM::Decrypt::NegotiatedController,
        CICM::Decrypt::Stream {
```

CICM::Decrypt::NegotiatedConduit is the negotiated version of CICM::Decrypt::Conduit. It is the result of a successful negotiation by CICM::Decrypt::Negotiator.

3.5.7.7.1 CICM::Decrypt::NegotiatedConduit Inheritance

CICM::Decrypt::NegotiatedConduit inherits from: CICM::Conduit, CICM::Decrypt::NegotiatedController and CICM::Decrypt::Stream.

3.5.7.8 Interface CICM::Decrypt::WithMACConduit

Interface CICM::Decrypt::WithMACConduit

```
interface WithMACConduit :
        CICM::AbstractMACVerifyConduit,
        CICM::Decrypt::Conduit {
```

CICM::Decrypt::WithMACConduit supports decryption operations between two independent security domains with the receipt of an indication as to whether MAC verification succeeded or failed in the initiating domain. The resulting conduit is capable of both managing the channel and accepting data for transformation. It is created by calling CICM::ChannelManager::create_decrypt_with_mac_conduit.

3.5.7.8.1 CICM::Decrypt::WithMACConduit Inheritance

CICM::Decrypt::WithMACConduit inherits from: CICM::AbstractMACVerifyConduit and CICM::Decrypt::Conduit.

3.5.7.9 Interface CICM::Decrypt::WithMACNegotiatedConduit

Interface CICM::Decrypt::WithMACNegotiatedConduit

```
interface WithMACNegotiatedConduit :
        CICM::AbstractMACVerifyConduit,
        CICM::Decrypt::NegotiatedConduit {
```

CICM::Decrypt::WithMACNegotiatedConduit is the negotiated version of CICM::Decrypt::WithMACConduit. It is the result of a successful negotiation by CICM::Decrypt::WithMACNegotiator.

3.5.7.9.1 CICM::Decrypt::WithMACNegotiatedConduit Inheritance

CICM::Decrypt::WithMACNegotiatedConduit inherits from: CICM::AbstractMACVerifyConduit and CICM::Decrypt::NegotiatedConduit.

3.5.7.10 Interface CICM::Decrypt::WithVerifyConduit

Interface CICM::Decrypt::WithVerifyConduit

```
interface WithVerifyConduit :
        CICM::AbstractSigVerifyConduit,
        CICM::Decrypt::Conduit {
```

CICM::Decrypt::WithVerifyConduit supports decryption operations between two independent security domains with the receipt of an indication as to whether signature verification succeeded or failed in the initiating domain. The resulting conduit is capable of both managing the channel and accepting transformed data. It is created by calling CICM::ChannelManager::create_decrypt_with_verify_conduit.

3.5.7.10.1 CICM::Decrypt::WithVerifyConduit Inheritance

CICM::Decrypt::WithVerifyConduit inherits from: CICM::AbstractSigVerifyConduit and CICM::Decrypt::Conduit.

3.5.7.11 Interface CICM::Decrypt::WithVerifyNegotiatedConduit

Interface CICM::Decrypt::WithVerifyNegotiatedConduit
```
interface WithVerifyNegotiatedConduit :
        CICM::AbstractSigVerifyConduit,
        CICM::Decrypt::NegotiatedConduit {
```
CICM::Decrypt::WithVerifyNegotiatedConduit is the negotiated version of CICM::Decrypt::WithVerifyConduit. It is the result of a successful negotiation by CICM::Decrypt::WithVerifyNegotiator.

3.5.7.11.1 CICM::Decrypt::WithVerifyNegotiatedConduit Inheritance

CICM::Decrypt::WithVerifyNegotiatedConduit inherits from: CICM::AbstractSigVerifyConduit and CICM::Decrypt::NegotiatedConduit.

3.5.7.12 Interface CICM::Decrypt::KeyUnwrapConduit

Interface CICM::Decrypt::KeyUnwrapConduit
```
interface KeyUnwrapConduit :
        CICM::Decrypt::Controller,
        CICM::Decrypt::KeyUnwrapStream {
```
CICM::Decrypt::KeyUnwrapConduit supports key unwrapping operations between two independent security domains. The resulting conduit is capable of both managing the channel and accepting transformed keys. It is created by calling CICM::ChannelManager::create_key_unwrap_conduit.

3.5.7.12.1 CICM::Decrypt::KeyUnwrapConduit Inheritance

CICM::Decrypt::KeyUnwrapConduit inherits from: CICM::Decrypt::Controller and CICM::Decrypt::KeyUnwrapStream.

3.5.7.13 Interface CICM::Decrypt::Negotiator

Interface CICM::Decrypt::Negotiator
```
interface Negotiator : CICM::Negotiator {
```
CICM::Decrypt::Negotiator initiates a negotiation to establish a shared key with a remote entity that is used to support encryption operations between two independent security domains. The result of a successful negotiation is a CICM::Decrypt::NegotiatedConduit which is capable of both managing the

channel and accepting data for transformation. CICM::Decrypt::Negotiator is created by calling CICM::ChannelManager::negotiate_decrypt_conduit.

3.5.7.13.1 CICM::Decrypt::Negotiator Inheritance

CICM::Decrypt::Negotiator inherits from: CICM::Negotiator.

3.5.7.13.2 CICM::Decrypt::Negotiator Methods

Method CICM::Decrypt::Negotiator::complete()

```
        CICM::Status complete(
                out CICM::Decrypt::NegotiatedConduit conduit_ref
        );
```

Complete negotiation and retrieve a negotiated decrypt conduit.

Parameters:

`[out]` *conduit_ref* Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_NOT_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

3.5.7.14 Interface CICM::Decrypt::ControllerNegotiator

Interface CICM::Decrypt::ControllerNegotiator

```
interface ControllerNegotiator : CICM::Negotiator {
```

CICM::Decrypt::ControllerNegotiator initiates a negotiation to establish a shared key with a remote entity that is used to support decryption operations between two independent security domains. The result of a successful negotiation is a CICM::Decrypt::NegotiatedController which is capable of managing the channel, but not accepting data for transformation. CICM::Decrypt::ControllerNegotiator is created by calling CICM::ChannelManager::negotiate_decrypt_controller.

3.5.7.14.1 CICM::Decrypt::ControllerNegotiator Inheritance

CICM::Decrypt::ControllerNegotiator inherits from: CICM::Negotiator.

3.5.7.14.2 CICM::Decrypt::ControllerNegotiator Methods

Method CICM::Decrypt::ControllerNegotiator::complete()

```
        CICM::Status complete(
                out    CICM::Decrypt::NegotiatedController controller_ref
        );
```

Complete negotiation and retrieve a negotiated control-only decrypt channel.

Parameters:

`[out]` *controller_ref* Reference to resulting controller.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_NOT_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

3.5.7.15 Interface CICM::Decrypt::WithMACNegotiator

Interface CICM::Decrypt::WithMACNegotiator

```
interface WithMACNegotiator : CICM::Negotiator {
```

CICM::Decrypt::WithMACNegotiator initiates a negotiation to establish a shared key with a remote entity that is used to support decryption operations between two independent security domains. Additionally, a message authentication code is received in the initiating domain. The result of a successful negotiation is a CICM::Decrypt::WithMACNegotiatedConduit which is capable of both managing the channel and accepting data for transformation. CICM::Decrypt::WithMACNegotiator is created by calling CICM::ChannelManager::negotiate_decrypt_with_mac_conduit.

3.5.7.15.1 CICM::Decrypt::WithMACNegotiator Inheritance

CICM::Decrypt::WithMACNegotiator inherits from: CICM::Negotiator.

3.5.7.15.2 CICM::Decrypt::WithMACNegotiator Methods

Method CICM::Decrypt::WithMACNegotiator::complete()

```
        CICM::Status complete(
                out CICM::Decrypt::WithMACNegotiatedConduit conduit_ref
        );
```

Complete negotiation and retrieve a negotiated MAC verify decrypt conduit.

Parameters:

`[out]` *conduit_ref* Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_NOT_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

3.5.7.16 Interface CICM::Decrypt::WithVerifyNegotiator

Interface CICM::Decrypt::WithVerifyNegotiator

```
interface WithVerifyNegotiator : CICM::Negotiator {
```

CICM::Decrypt::WithVerifyNegotiator initiates a negotiation to establish a shared key with a remote entity that is used to support decryption operations between two independent security domains. Additionally, an indication as to whether verification succeeded or failed is received in the initiating domain. The result of a successful negotiation is a CICM::Decrypt::WithVerifyNegotiatedConduit which is capable of both managing the channel and accepting data for transformation.
CICM::Decrypt::WithVerifyNegotiator is created by calling CICM::ChannelManager::negotiate_decrypt_with_verify_conduit.

3.5.7.16.1 CICM::Decrypt::WithVerifyNegotiator Inheritance

CICM::Decrypt::WithVerifyNegotiator inherits from: CICM::Negotiator.

3.5.7.16.2 CICM::Decrypt::WithVerifyNegotiator Methods

Method CICM::Decrypt::WithVerifyNegotiator::complete()

```
        CICM::Status complete(
                out CICM::Decrypt::WithVerifyNegotiatedConduit conduit_ref
        );
```

Complete negotiation and retrieve a negotiated verify and decrypt conduit.
Parameters:
`[out]` *conduit_ref* Reference to resulting conduit.
Returns:
S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_NOT_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

3.5.8 DUPLEX CHANNEL MANAGEMENT

Namespace CICM::Duplex

```
module Duplex {
```

The CICM::Duplex namespace contains interfaces that support encryption/decryption operations between two independent security domains.

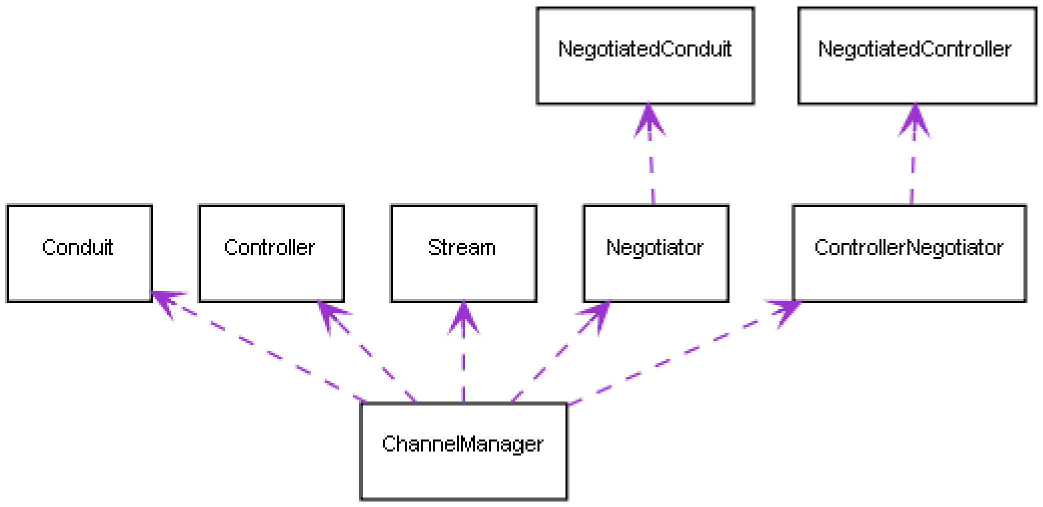


Figure 33. Interface Relationship Diagram for Duplex Channels

3.5.8.1 Interface CICM::Duplex::ChannelManager

Interface CICM::Duplex::ChannelManager

```
interface ChannelManager {
```

CICM::Duplex::ChannelManager is an abstraction inherited by CICM::ChannelManager that supports the creation of encryption/decryption negotiators, conduits, controllers, and streams. See CICM::ChannelManager for additional information.

3.5.8.1.1 CICM::Duplex::ChannelManager Methods

Method CICM::Duplex::ChannelManager::negotiate_duplex_conduit()

```
        CICM::Status negotiate_duplex_conduit(
                in  CICM::RemotePort remote_port,
                in  CICM::ProtocolId protocol,
                in  CICM::AsymKey key_ref,
                out CICM::Duplex::Negotiator negotiator_ref
        );
```

Initiate a negotiation to establish a shared key with a peer, resulting in a conduit that results will encrypt/decrypt data.

Parameters:

`[in]`	*remote_port*	Remote port.
`[in]`	*protocol*	Protocol identifier.
`[in]`	*key_ref*	Reference to negotiation key.
`[out]`	*negotiator_ref*	Reference to resulting negotiator.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_PROTO_INVALID, S_PROTO_INCOMPATIBLE, S_PROTO_UNDETERMINED, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Duplex::ChannelManager::negotiate_duplex_controller()

```
        CICM::Status negotiate_duplex_controller(
                in  CICM::LocalPort local_port,
                in  CICM::RemotePort remote_port,
                in  CICM::ProtocolId protocol,
                in  CICM::AsymKey key_ref,
                out CICM::Duplex::ControllerNegotiator negotiator_ref
        );
```

Initiate a negotiation to establish a shared key with a peer, resulting in a controller to manage a duplex channel.

Parameters:

`[in]`	*local_port*	Local port.
`[in]`	*remote_port*	Remote port.
`[in]`	*protocol*	Protocol identifier.
`[in]`	*key_ref*	Reference to negotiation key.
`[out]`	*negotiator_ref*	Reference to resulting negotiator.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_PROTO_INVALID, S_PROTO_INCOMPATIBLE, S_PROTO_UNDETERMINED, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Duplex::ChannelManager::create_duplex_conduit()

```
        CICM::Status create_duplex_conduit(
                in  CICM::RemotePort remote_port,
                in  CICM::SymKey key_ref,
                in  CICM::SymEncrAlgorithmId algorithm,
                out CICM::Duplex::Conduit conduit_ref
        );
```

Create duplex channel to encrypt/decrypt a stream of data.

Parameters:

`[in]`	*remote_port*	Remote port.
`[in]`	*key_ref*	Reference to decryption key.
`[in]`	*algorithm*	Encryption/decryption algorithm/mode.
`[out]`	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Duplex::ChannelManager::create_duplex_controller()

```
CICM::Status create_duplex_controller(
        in  CICM::LocalPort local_port,
        in  CICM::RemotePort remote_port,
        in  CICM::SymKey key_ref,
        in  CICM::SymEncrAlgorithmId algorithm,
        out CICM::Duplex::Controller controller_ref
);
```

Create controller to configure and control a duplex channel.

Remarks:

In some cases, hosts may depend upon separate processes to control and use a channel. This method returns the channel controller and must be called before the corresponding stream is retrieved.

Parameters:

`[in]`	*local_port*	Local port.
`[in]`	*remote_port*	Remote port.
`[in]`	*key_ref*	Reference to negotiation key.
`[in]`	*algorithm*	Encryption/decryption algorithm/mode.
`[out]`	*controller_ref*	Reference to resulting controller.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_INVALID, S_KEY_EXPIRED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Duplex::ChannelManager::get_duplex_stream()

```
CICM::Status get_duplex_stream(
        in  CICM::LocalPort local_port,
        out CICM::Duplex::Stream stream_ref
        );
```

Create stream associated with previously created controller to accept data for transformation.

Parameters:

`[in]` *local_port* Local port.
`[out]` *stream_ref* Reference to resulting stream.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_NOT_FOUND

3.5.8.2 Interface CICM::Duplex::Stream

Interface CICM::Duplex::Stream

```
interface Stream :
        CICM::Encrypt::Stream,
        CICM::Decrypt::Stream {
```

CICM::Duplex::Stream supports encryption/decryption operations between two independent security domains. The resulting stream is capable of accepting data for transformation and receiving transformed data, but not managing the channel. It is created by calling CICM::ChannelManager::get_duplex_stream.

3.5.8.2.1 CICM::Duplex::Stream Inheritance

CICM::Duplex::Stream inherits from: CICM::Encrypt::Stream and CICM::Decrypt::Stream.

3.5.8.3 Interface CICM::Duplex::Controller

Interface CICM::Duplex::Controller

```
interface Controller :
        CICM::Encrypt::Controller,
        CICM::Decrypt::Controller {
```

CICM::Duplex::Controller supports encryption/decryption operations between two independent security domains. The resulting controller is capable of managing the channel, but not accepting data for transformation and receiving transformed data. It is created by calling CICM::ChannelManager::create_duplex_controller.

3.5.8.3.1 CICM::Duplex::Controller Inheritance

CICM::Duplex::Controller inherits from: CICM::Encrypt::Controller and CICM::Decrypt::Controller.

3.5.8.4 Interface CICM::Duplex::NegotiatedController

Interface CICM::Duplex::NegotiatedController

```
interface NegotiatedController :
        CICM::Encrypt::NegotiatedController,
        CICM::Decrypt::NegotiatedController {
```

CICM::Duplex::NegotiatedController is the negotiated version of CICM::Duplex::Controller. It is the result of a successful negotiation by CICM::Duplex::ControllerNegotiator.

3.5.8.4.1 CICM::Duplex::NegotiatedController Inheritance

CICM::Duplex::NegotiatedController inherits from: CICM::Encrypt::NegotiatedController and CICM::Decrypt::NegotiatedController.

3.5.8.5 Interface CICM::Duplex::Conduit

Interface CICM::Duplex::Conduit

```
interface Conduit :
        CICM::Conduit,
        CICM::Duplex::Controller,
        CICM::Duplex::Stream {
```

CICM::Duplex::Conduit supports encryption/decryption operations between two independent security domains. The resulting conduit is capable of both managing the channel and accepting data for transformation and receiving transformed data. It is created by calling CICM::ChannelManager::create_duplex_conduit.

3.5.8.5.1 CICM::Duplex::Conduit Inheritance

CICM::Duplex::Conduit inherits from: CICM::Conduit, CICM::Duplex::Controller and CICM::Duplex::Stream.

3.5.8.6 Interface CICM::Duplex::NegotiatedConduit

Interface CICM::Duplex::NegotiatedConduit

```
interface NegotiatedConduit :
        CICM::Duplex::NegotiatedController,
        CICM::Duplex::Stream {
```

CICM::Duplex::NegotiatedConduit is the negotiated version of CICM::Duplex::Conduit. It is the result of a successful negotiation by CICM::Duplex::Negotiator.

3.5.8.6.1 CICM::Duplex::NegotiatedConduit Inheritance

CICM::Duplex::NegotiatedConduit inherits from: CICM::Duplex::NegotiatedController and CICM::Duplex::Stream.

3.5.8.7 Interface CICM::Duplex::ControllerNegotiator

Interface CICM::Duplex::ControllerNegotiator

```
interface ControllerNegotiator : CICM::Negotiator {
```

CICM::Duplex::ControllerNegotiator initiates a negotiation to establish a shared key with a remote entity that is used to support encryption/decryption operations between two independent security domains. The result of a successful negotiation is a CICM::Duplex::NegotiatedController which is capable of managing the channel, but not accepting data for transformation. CICM::Duplex::ControllerNegotiator is created by calling CICM::ChannelManager::negotiate_duplex_controller.

3.5.8.7.1 CICM::Duplex::ControllerNegotiator Inheritance

CICM::Duplex::ControllerNegotiator inherits from: CICM::Negotiator.

3.5.8.7.2 CICM::Duplex::ControllerNegotiator Methods

Method CICM::Duplex::ControllerNegotiator::complete()

```
        CICM::Status complete(
                out     CICM::Duplex::NegotiatedController controller_ref
        );
```

Complete negotiation and retrieve a negotiated control-only duplex conduit.

Parameters:

`[out]` *controller_ref* Reference to resulting controller.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_NOT_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

3.5.8.8 Interface CICM::Duplex::Negotiator

Interface CICM::Duplex::Negotiator

```
interface Negotiator : CICM::Negotiator {
```

CICM::Duplex::Negotiator initiates a negotiation to establish a shared key with a remote entity that is used to support encryption/decryption operations between two independent security domains. The result of a successful negotiation is a CICM::Duplex::NegotiatedConduit which is capable of both managing the channel and accepting data for transformation. CICM::Duplex::Negotiator is created by calling CICM::ChannelManager::negotiate_duplex_conduit.

3.5.8.8.1 CICM::Duplex::Negotiator Inheritance

CICM::Duplex::Negotiator inherits from: CICM::Negotiator.

3.5.8.8.2 CICM::Duplex::Negotiator Methods

Method CICM::Duplex::Negotiator::complete()

```
         CICM::Status complete(
                out     CICM::Duplex::NegotiatedConduit conduit_ref
        );
```

Complete negotiation and retrieve a negotiated duplex channel.

Parameters:

`[out]` *conduit_ref* Reference to resulting channel.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_NOT_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

3.5.9 BYPASS (SEND) CHANNEL MANAGMENT

Namespace CICM::BypassWrite

```
module BypassWrite {
```

The CICM::BypassWrite namespace contains channels that support full bypass write operations between two independent security domains.

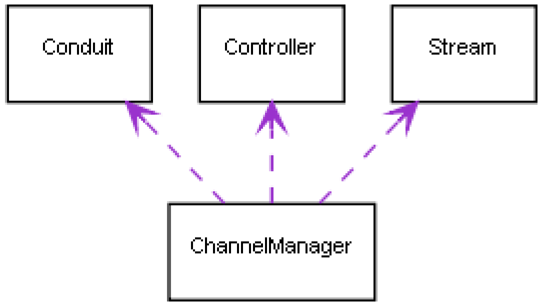


Figure 34. Interface Relationship Diagram for Sending Bypass Data

3.5.9.1 Interface CICM::BypassWrite::ChannelManager

Interface CICM::BypassWrite::ChannelManager

```
interface ChannelManager {
```

CICM::BypassWrite::ChannelManager is an abstraction inherited by CICM::ChannelManager that supports the creation of full bypass conduits, controllers, and streams for writing. See CICM::ChannelManager for additional information.

3.5.9.1.1 CICM::BypassWrite::ChannelManager Methods

Method CICM::BypassWrite::ChannelManager::create_bypass_write_conduit()

```
        CICM::Status create_bypass_write_conduit(
                in  CICM::RemotePort remote_port,
                out CICM::BypassWrite::Conduit conduit_ref
        );
```

Creates a conduit to write bypass data.

Parameters:

`[in]` *remote_port* Remote port.

`[out]` *conduit_ref* Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::BypassWrite::ChannelManager::create_bypass_write_controller()

```
CICM::Status create_bypass_write_controller(
        in  CICM::LocalPort local_port,
        in  CICM::RemotePort remote_port,
        out CICM::BypassWrite::Controller controller_ref
);
```

Returns the controller of a channel that writes bypass data.

Remarks:

In some cases, hosts may depend upon separate processes to control and use a channel. This method returns the channel controller and must be called before the corresponding stream is retrieved.

See also:

CICM::BypassWrite::ChannelManager::get_bypass_write_stream for the method that returns the channel stream.

Parameters:

`[in]` *local_port* Local port.

`[in]` *remote_port* Remote port.

`[out]` *controller_ref* Reference to resulting controller.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::BypassWrite::ChannelManager::get_bypass_write_stream()

```
CICM::Status get_bypass_write_stream(
        in  CICM::LocalPort local_port,
        out CICM::BypassWrite::Stream stream_ref
);
```

Returns the stream corresponding to a pre-existing controller on the given local port.

See also:

CICM::BypassWrite::ChannelManager::create_bypass_write_controller for the method that returns the channel controller.

Parameters:

`[in]` *local_port* Local port.
`[out]` *stream_ref* Reference to resulting stream.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_NOT_FOUND

3.5.9.2 Interface CICM::BypassWrite::Stream

Interface CICM::BypassWrite::Stream

```
interface Stream : CICM::WriteStream {
```

CICM::BypassWrite::Stream supports full bypass between two independent security domains. The resulting stream is capable of accepting data for bypass, but not managing the channel. It is created by calling CICM::ChannelManager::get_bypass_write_stream.

3.5.9.2.1 CICM::BypassWrite::Stream Inheritance

CICM::BypassWrite::Stream inherits from: CICM::WriteStream.

3.5.9.2.2 CICM::BypassWrite::Stream Methods

Method CICM::BypassWrite::Stream::write_bypass()

```
        CICM::Status write_bypass(
                in  CICM::Buffer buffer
        );
```

Write bypass data to a channel stream. The method blocks until the data has been sent.

Parameters:

`[in]` *buffer* Data to bypass.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_IO_ERROR, S_CHANNEL_DATA_INVALID, S_CHANNEL_DATA_INVALID_LEN, S_BYPASS_DATARATE_EXCEEDED, S_BYPASS_DATALIMIT_EXCEEDED

Method CICM::BypassWrite::Stream::write_bypass_non_blocking()

```
        CICM::Status write_bypass_non_blocking(
                in  CICM::Buffer buffer,
                in  CICM::TransId transaction_id
        );
```

Registers a buffer of data to be sent to the module for bypass and then immediately returns control to the caller. The length of the data is encapsulated in the buffer parameter. The caller may use the CICM::BypassWrite::Stream::write_bypass_poll method to proactively poll the channel to determine the status of the operation. The caller is responsible for maintaining any necessary metadata associated with the transaction_id parameter. Memory responsibilities and calling conventions shall follow the appropriate IDL language mapping conventions.

Parameters:

`[in]`	*buffer*	Data to bypass.
`[in]`	*transaction_id*	Unique transaction id that will be used by the CICM::BypassWrite::Stream::write_bypass_poll method to determine to which buffer the poll status applies.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_IO_ERROR, S_CHANNEL_DATA_INVALID, S_CHANNEL_DATA_INVALID_LEN, S_BYPASS_DATARATE_EXCEEDED, S_BYPASS_DATALIMIT_EXCEEDED

Method CICM::BypassWrite::Stream::write_bypass_poll()

```
        CICM::Status write_bypass_poll(
                in  CICM::TransId transaction_id,
                out CICM::WriteStream::WriteStatus status
        );
```

Returns the status of the non-blocking bypass operation specified by the transaction_id parameter. Memory responsibilities and calling conventions shall follow the appropriate IDL language mapping conventions.

Parameters:

`[in]`	*transaction_id*	Unique transaction id previously specified to the CICM::BypassWrite::Stream::write_bypass_non_blocking method that allows the poll operation to determine to which buffer the poll status applies.
`[out]`	*status*	Status of the non-blocking operation corresponding to the transaction_id parameter.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_IO_ERROR

3.5.9.3 Interface CICM::BypassWrite::Controller

Interface CICM::BypassWrite::Controller

```
interface Controller : CICM::MultiDomainController {};
```

CICM::BypassWrite::Controller supports full bypass between two independent security domains. The resulting controller is capable of managing the channel, but not accepting data for bypass. It is created by calling CICM::ChannelManager::create_bypass_write_controller.

3.5.9.3.1 CICM::BypassWrite::Controller Inheritance

CICM::BypassWrite::Controller inherits from: CICM::MultiDomainController.

3.5.9.4 Interface CICM::BypassWrite::Conduit

Interface CICM::BypassWrite::Conduit

```
interface Conduit :
        CICM::Conduit,
        CICM::BypassWrite::Controller,
        CICM::BypassWrite::Stream {
```

CICM::BypassWrite::Conduit supports full bypass between two security domains. The resulting conduit is capable of both managing the channel and accepting data for bypass. It is created by calling CICM::ChannelManager::create_bypass_write_conduit.

3.5.9.4.1 CICM::BypassWrite::Conduit Inheritance

CICM::BypassWrite::Conduit inherits from: CICM::Conduit, CICM::BypassWrite::Controller and CICM::BypassWrite::Stream.

3.5.10 BYPASS (READ) CHANNEL MANAGEMENT

Namespace CICM::BypassRead

```
module BypassRead {
```

The CICM::BypassRead namespace contains channels that support full bypass read operations between two independent security domains.

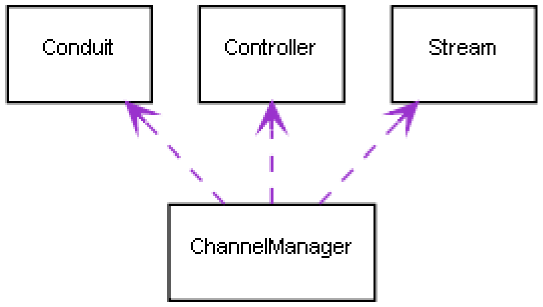


Figure 35. Interface Relationship Diagram for Receiving Bypass Data

3.5.10.1 Interface CICM::BypassRead::ChannelManager

Interface CICM::BypassRead::ChannelManager

```
interface ChannelManager {
```

CICM::BypassRead::ChannelManager is an abstraction inherited by CICM::ChannelManager that supports the creation of full bypass conduits, controllers, and streams for reading. See CICM::ChannelManager for additional information.

3.5.10.1.1 CICM::BypassRead::ChannelManager Methods

Method CICM::BypassRead::ChannelManager::create_bypass_read_conduit()

```
        CICM::Status create_bypass_read_conduit(
                in  CICM::RemotePort remote_port,
                out CICM::BypassRead::Conduit conduit_ref
        );
```

Creates a conduit to read bypass data.

Parameters:

`[in]` *remote_port* Remote port.

`[out]` *conduit_ref* Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::BypassRead::ChannelManager::create_bypass_read_controller()

```
        CICM::Status create_bypass_read_controller(
                in  CICM::LocalPort local_port,
                in  CICM::RemotePort remote_port,
                out CICM::BypassRead::Controller controller_ref
        );
```

Returns the controller of a channel that reads bypass data.

Remarks:

In some cases, hosts may depend upon separate processes to control and use a channel. This method returns the channel controller and must be called before the corresponding stream is retrieved.

See also:

CICM::BypassRead::ChannelManager::get_bypass_read_stream for the method that returns the channel stream.

Parameters:

`[in]` *local_port* Local port.

`[in]` *remote_port* Remote port.

`[out]` *controller_ref* Reference to resulting controller.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::BypassRead::ChannelManager::get_bypass_read_stream()

```
        CICM::Status get_bypass_read_stream(
                in  CICM::LocalPort local_port,
                out CICM::BypassRead::Stream stream_ref
        );
```

Returns the stream corresponding to a pre-existing controller on the given local port.

See also:

CICM::BypassRead::ChannelManager::get_bypass_read_controller for the method that returns the channel controller.

Parameters:

`[in]` *local_port* Local port.
`[out]` *stream_ref* Reference to resulting stream.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_NOT_FOUND

3.5.10.2 Interface CICM::BypassRead::Stream

Interface CICM::BypassRead::Stream

```
interface Stream : CICM::ReadStream {
```

CICM::BypassRead::Stream supports full bypass between two independent security domains. The resulting stream is capable of accepting bypassed data, but not managing the channel. It is created by calling CICM::ChannelManager::get_bypass_read_stream.

3.5.10.2.1 CICM::BypassRead::Stream Inheritance

CICM::BypassRead::Stream inherits from: CICM::ReadStream.

3.5.10.2.2 CICM::BypassRead::Stream Methods

Method CICM::BypassRead::Stream::read_bypass()

```
        CICM::Status read_bypass(
                out CICM::Buffer buffer
        );
```

Read bypass data off of channel stream. The method blocks until data becomes available.

Parameters:

`[out]` *buffer* Bypassed data read from module.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE,

S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_WRAPPED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_IO_ERROR, S_CHANNEL_BUFFER_LEN

Method CICM::BypassRead::Stream::read_bypass_non_blocking()

```
        CICM::Status read_bypass_non_blocking(
                out  CICM::Buffer buffer,
                in   CICM::TransId transaction_id
        );
```

Registers a buffer into which bypass data will be copied, and then immediately returns control to the caller. The size of the allocated buffer and length of the resulting bypassed data is encapsulated in the buffer parameter. The caller may use the CICM::BypassRead::Stream::read_bypass_poll method to proactively poll the channel to determine the status of the operation. The caller is responsible for maintaining any necessary metadata associated with the transaction_id parameter. Memory responsibilities and calling conventions shall follow the appropriate IDL language mapping conventions.

Parameters:

`[out]`	*buffer*	Bypassed data to read.
`[in]`	*transaction_id*	Unique transaction id that will be used by the CICM::BypassRead::Stream::read_bypass_poll method to determine to which buffer the poll status applies.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_WRAPPED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_IO_ERROR, S_CHANNEL_BUFFER_LEN

Method CICM::BypassRead::Stream::read_bypass_poll()

```
        CICM::Status read_bypass_poll(
                in  CICM::TransId transaction_id,
                out CICM::ReadStream::ReadStatus status
        );
```

Returns the status of the non-blocking bypass operation specified by the transaction_id parameter. Upon completion of the operation, the caller must use the metadata associated with the transaction_id parameter to determine which buffer has been populated. Memory responsibilities and calling conventions shall follow the appropriate IDL language mapping conventions.

Parameters:

`[in]`	*transaction_id*	Unique transaction id previously specified to the CICM::BypassRead::Stream::read_bypass_non_blocking method that allows the poll operation to determine to which buffer the poll status applies.
`[out]`	*status*	Status of the non-blocking operation corresponding to the transaction_id parameter.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_IO_ERROR

3.5.10.3 Interface CICM::BypassRead::Controller

Interface CICM::BypassRead::Controller

```
interface Controller : CICM::MultiDomainController {};
```

CICM::BypassRead::Controller supports full bypass between two independent security domains. The resulting controller is capable of managing the channel, but not accepting bypassed data. It is created by calling CICM::ChannelManager::create_bypass_read_controller.

3.5.10.3.1 CICM::BypassRead::Controller Inheritance

CICM::BypassRead::Controller inherits from: CICM::MultiDomainController.

3.5.10.4 Interface CICM::BypassRead::Conduit

Interface CICM::BypassRead::Conduit

```
interface Conduit :
        CICM::Conduit,
        CICM::BypassRead::Controller,
        CICM::BypassRead::Stream {
```

CICM::BypassRead::Conduit supports full bypass between two independent security domains. The resulting conduit is capable of both managing the channel and accepting bypassed data. It is created by calling CICM::ChannelManager::create_bypass_read_conduit.

3.5.10.4.1 CICM::BypassRead::Conduit Inheritance

CICM::BypassRead::Conduit inherits from: CICM::Conduit, CICM::BypassRead::Controller and CICM::BypassRead::Stream.

3.5.11 ENCRYPTION WITH SELECTIVE BYPASS CHANNEL MANAGEMENT

Namespace CICM::EncryptBypass

```
module EncryptBypass {
```

The CICM::EncryptBypass namespace contains interfaces that support encryption with selective bypass operations between two indepenent security domains.

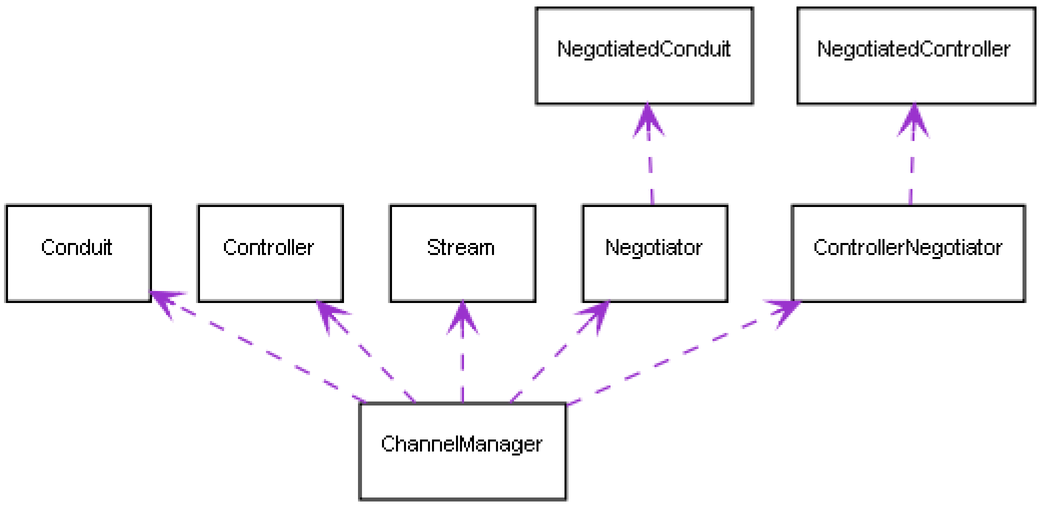


Figure 36. Interface Relationship Diagram for Encryption Channels with Selective Bypass

3.5.11.1 Interface CICM::EncryptBypass::ChannelManager

Interface CICM::EncryptBypass::ChannelManager

```
interface ChannelManager {
```

CICM::EncryptBypass::ChannelManager is an abstraction inherited by CICM::ChannelManager that supports the creation of encryption with selective bypass negotiators, conduits, controllers, and streams. See CICM::ChannelManager for additional information.

Remarks:

Note that if a system is using selective bypass channels to perform header bypass, policy may govern that a system performs the bypass function before the encryption function and performs the bypass function only once.

3.5.11.1.1 CICM::EncryptBypass::ChannelManager Methods

Method CICM::EncryptBypass::ChannelManager::negotiate_encrypt_bypass_conduit()

```
        CICM::Status negotiate_encrypt_bypass_conduit(
                in  CICM::RemotePort remote_port,
                in  CICM::ProtocolId protocol,
                in  CICM::AsymKey key_ref,
                out CICM::EncryptBypass::Negotiator negotiator_ref
        );
```

Initiate a negotiation to establish a shared key with a peer. The channel that results will selectively encrypt or bypass a stream of data.

Parameters:

`[in]`	*remote_port*	Remote port.
`[in]`	*protocol*	Protocol identifier.
`[in]`	*key_ref*	Reference to negotiation key.
`[out]`	*negotiator_ref*	Reference to resulting negotiator.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_PROTO_INVALID, S_PROTO_INCOMPATIBLE, S_PROTO_UNDETERMINED, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::EncryptBypass::ChannelManager::negotiate_encrypt_bypass_controller()

```
CICM::Status negotiate_encrypt_bypass_controller(
        in  CICM::LocalPort local_port,
        in  CICM::RemotePort remote_port,
        in  CICM::ProtocolId protocol,
        in  CICM::AsymKey key_ref,
        out CICM::EncryptBypass::ControllerNegotiator negotiator_ref
);
```

Initiate a negotiation to establish a shared key with a peer, resulting in a controller to manage an encrypt with bypass channel.

Parameters:

`[in]`	*local_port*	Remote port.
`[in]`	*remote_port*	Remote port.
`[in]`	*protocol*	Protocol identifier.
`[in]`	*key_ref*	Reference to negotiation key.
`[out]`	*negotiator_ref*	Reference to resulting negotiator.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_PROTO_INVALID, S_PROTO_INCOMPATIBLE, S_PROTO_UNDETERMINED, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::EncryptBypass::ChannelManager::create_encrypt_bypass_conduit()

```
CICM::Status create_encrypt_bypass_conduit(
        in  CICM::RemotePort remote_port,
        in  CICM::SymKey key_ref,
        in  CICM::SymEncrAlgorithmId algorithm,
        out CICM::EncryptBypass::Conduit conduit_ref
);
```

Create conduit to selectively encrypt or bypass a stream of data.

Parameters:

`[in]`	*remote_port*	Remote port.
`[in]`	*key_ref*	Reference to encryption key.
`[in]`	*algorithm*	Encryption algorithm/mode.
`[out]`	*conduit_ref*	Reference to resulting negotiator.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::EncryptBypass::ChannelManager::create_encrypt_bypass_controller()

```
        CICM::Status create_encrypt_bypass_controller(
                in  CICM::LocalPort local_port,
                in  CICM::RemotePort remote_port,
                in  CICM::SymKey key_ref,
                in  CICM::SymEncrAlgorithmId algorithm,
                out CICM::EncryptBypass::Controller controller_ref
        );
```

Create controller to configure and control an encrypt with bypass channel.

Remarks:

In some cases, hosts may depend upon separate processes to control and use a channel. This method returns the channel controller and must be called before the corresponding stream is retrieved.

Parameters:

`[in]`	*local_port*	Local port.
`[in]`	*remote_port*	Remote port.
`[in]`	*key_ref*	Reference to encryption key.
`[in]`	*algorithm*	Encryption algorithm/mode.
`[out]`	*controller_ref*	Reference to resulting controller.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::EncryptBypass::ChannelManager::get_encrypt_bypass_stream()

```
        CICM::Status get_encrypt_bypass_stream(
                in  CICM::LocalPort local_port,
                out CICM::EncryptBypass::Stream stream_ref
        );
```

Create stream associated with previously created controller to accept data for transformation.

Parameters:

`[in]` *local_port* Local port.

`[out]` *stream_ref* Reference to resulting stream.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_NOT_FOUND

3.5.11.2 Interface CICM::EncryptBypass::Stream

Interface CICM::EncryptBypass::Stream

```
interface Stream :
        CICM::Encrypt::Stream,
        CICM::BypassWrite::Stream {
```

CICM::EncryptBypass::Stream supports encryption and selective bypass operations between two independent security domains. The resulting stream is capable of accepting data for transformation, but not managing the channel. It is created by calling CICM::ChannelManager::get_encrypt_bypass_stream.

3.5.11.2.1 CICM::EncryptBypass::Stream Inheritance

CICM::EncryptBypass::Stream inherits from: CICM::Encrypt::Stream and CICM::BypassWrite::Stream.

3.5.11.3 Interface CICM::EncryptBypass::NegotiatedController

Interface CICM::EncryptBypass::NegotiatedController

```
interface NegotiatedController : CICM::Encrypt::NegotiatedController {};
```

CICM::EncryptBypass::NegotiatedController is the negotiated version of CICM::EncryptBypass::Controller. It is the result of a successful negotiation by CICM::EncryptBypass::ControllerNegotiator.

3.5.11.3.1 CICM::EncryptBypass::NegotiatedController Inheritance

CICM::EncryptBypass::NegotiatedController inherits from: CICM::Encrypt::NegotiatedController.

3.5.11.4 Interface CICM::EncryptBypass::Controller

Interface CICM::EncryptBypass::Controller

```
interface Controller : CICM::Encrypt::Controller {};
```

CICM::EncryptBypass::Controller supports encryption and selective bypass operations between two independent security domains. The resulting controller is capable of managing the channel, but not accepting data for transformation/bypass. It is created by calling CICM::ChannelManager::create_encrypt_bypass_controller.

3.5.11.4.1 CICM::EncryptBypass::Controller Inheritance

CICM::EncryptBypass::Controller inherits from: CICM::Encrypt::Controller.

3.5.11.5 Interface CICM::EncryptBypass::Conduit

Interface CICM::EncryptBypass::Conduit

```
interface Conduit :
        CICM::Encrypt::Conduit,
        CICM::EncryptBypass::Stream {
```

CICM::EncryptBypass::Conduit supports encryption and selective bypass operations between two independent security domains. The resulting conduit is capable of both managing the channel and accepting data for transformation/bypass. It is created by calling CICM::ChannelManager::create_encrypt_bypass_conduit.

3.5.11.5.1 CICM::EncryptBypass::Conduit Inheritance

CICM::EncryptBypass::Conduit inherits from: CICM::Encrypt::Conduit and CICM::EncryptBypass::Stream.

3.5.11.6 Interface CICM::EncryptBypass::NegotiatedConduit

Interface CICM::EncryptBypass::NegotiatedConduit

```
interface NegotiatedConduit :
        CICM::Encrypt::NegotiatedController,
        CICM::EncryptBypass::Stream {
```

CICM::EncryptBypass::NegotiatedConduit is the negotiated version of CICM::EncryptBypass::Conduit. It is the result of a successful negotiation by CICM::EncryptBypass::Negotiator.

3.5.11.6.1 CICM::EncryptBypass::NegotiatedConduit Inheritance

CICM::EncryptBypass::NegotiatedConduit inherits from: CICM::Encrypt::NegotiatedController and CICM::EncryptBypass::Stream.

3.5.11.7 Interface CICM::EncryptBypass::ControllerNegotiator

Interface CICM::EncryptBypass::ControllerNegotiator

```
interface ControllerNegotiator : CICM::Negotiator {
```

CICM::EncryptBypass::ControllerNegotiator initiates a negotiation to establish a shared key with a remote entity that is used to support encryption and selective bypass operations between two independent security domains. The result of a successful negotiation is a CICM::EncryptBypass::NegotiatedController which is capable of managing the channel, but not accepting data for transformation/bypass. CICM::EncryptBypass::ControllerNegotiator is created by calling CICM::ChannelManager::negotiate_encrypt_bypass_controller.

3.5.11.7.1 CICM::EncryptBypass::ControllerNegotiator Inheritance

CICM::EncryptBypass::ControllerNegotiator inherits from: CICM::Negotiator.

3.5.11.7.2 CICM::EncryptBypass::ControllerNegotiator Methods

Method CICM::EncryptBypass::ControllerNegotiator::complete()

```
        CICM::Status complete(
                out CICM::EncryptBypass::NegotiatedController controller_ref
        );
```

Complete negotiation and retrieve a negotiated encrypt bypass control-only channel.

Parameters:

`[out]` *controller_ref* Reference to resulting controller.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_NOT_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

3.5.11.8 Interface CICM::EncryptBypass::Negotiator

Interface CICM::EncryptBypass::Negotiator

```
interface Negotiator : CICM::Negotiator {
```

CICM::EncryptBypass::Negotiator initiates a negotiation to establish a shared key with a remote entity that is used to support encryption and bypass operations between two independent security domains. Additionally, selective bypass is supported on the same conduit. The result of a successful negotiation is a CICM::EncryptBypass::NegotiatedConduit which is capable of both managing the channel and accepting data for transformation/bypass. CICM::EncryptBypass::Negotiator is created by calling CICM::ChannelManager::negotiate_encrypt_bypass_conduit.

3.5.11.8.1 CICM::EncryptBypass::Negotiator Inheritance

CICM::EncryptBypass::Negotiator inherits from: CICM::Negotiator.

3.5.11.8.2 CICM::EncryptBypass::Negotiator Methods

Method CICM::EncryptBypass::Negotiator::complete()

```
        CICM::Status complete(
                out CICM::EncryptBypass::NegotiatedConduit conduit_ref
        );
```

Complete negotiation and retrieve a negotiated encrypt bypass conduit.

Parameters:

`[out]` *conduit_ref* Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_NOT_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

3.5.12 DECRYPTION WITH SELECTIVE BYPASS CHANNEL MANAGEMENT

Namespace CICM::DecryptBypass

```
module DecryptBypass {
```

The CICM::DecryptBypass namespace contains interfaces that support decryption with selective bypass operations between two independent security domains.

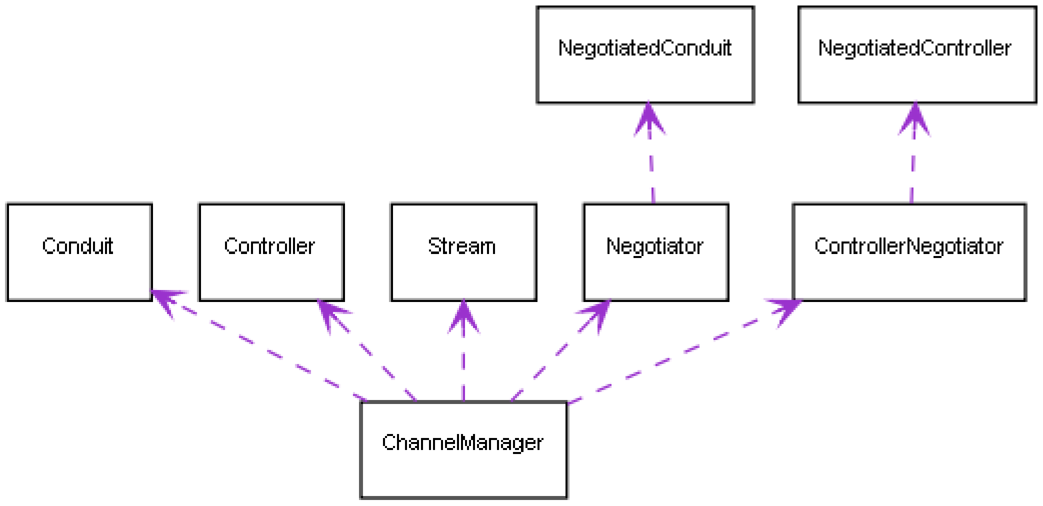


Figure 37. Interface Relationship Diagram for Decryption Channels with Selective Bypass

3.5.12.1 Interface CICM::DecryptBypass::ChannelManager

Interface CICM::DecryptBypass::ChannelManager

```
interface ChannelManager {
```

CICM::DecryptBypass::ChannelManager is an abstraction inherited by CICM::ChannelManager that supports the creation of decryption with selective bypass negotiators, conduits, controllers, and streams. See CICM::ChannelManager for additional information.

Remarks:

CICM does not specify the structure of the data that is read from a selective bypass channel which may contain special formatting to indicate which subset of the data was bypassed. The Implementation Conformance Statement (see Section 4, Conformance and Extensions) must reference a standard format or define a module developer-specific format implemented by the module for data read from selective bypass channels.

3.5.12.1.1 CICM::DecryptBypass::ChannelManager Methods

Method CICM::DecryptBypass::ChannelManager::negotiate_decrypt_bypass_conduit()

```
        CICM::Status negotiate_decrypt_bypass_conduit(
                in  CICM::RemotePort remote_port,
                in  CICM::ProtocolId protocol,
                in  CICM::AsymKey key_ref,
                out CICM::DecryptBypass::Negotiator negotiator_ref
        );
```

Initiate a negotiation to establish a shared key with a peer. The channel that results will selectively decrypt or bypass a stream of data.

Parameters:

`[in]`	*remote_port*	Remote port.
`[in]`	*protocol*	Protocol identifier.
`[in]`	*key_ref*	Reference to negotiation key.
`[out]`	*negotiator_ref*	Reference to resulting negotiator.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_PROTO_INVALID, S_PROTO_INCOMPATIBLE, S_PROTO_UNDETERMINED, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::DecryptBypass::ChannelManager::negotiate_decrypt_bypass_controller()

```
        CICM::Status negotiate_decrypt_bypass_controller(
                in  CICM::LocalPort local_port,
                in  CICM::RemotePort remote_port,
                in  CICM::ProtocolId protocol,
                in  CICM::AsymKey key_ref,
                out CICM::DecryptBypass::ControllerNegotiator negotiator_ref
        );
```

Initiate a negotiation to establish a shared key with a peer, resulting in a controller to manage a decrypt with bypass channel.

Parameters:

`[in]`	*local_port*	Local port.
`[in]`	*remote_port*	Remote port.
`[in]`	*protocol*	Protocol identifier.
`[in]`	*key_ref*	Reference to negotiation key.
`[out]`	*negotiator_ref*	Reference to resulting negotiator.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_PROTO_INVALID, S_PROTO_INCOMPATIBLE, S_PROTO_UNDETERMINED, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::DecryptBypass::ChannelManager::create_decrypt_bypass_conduit()

```
        CICM::Status create_decrypt_bypass_conduit(
                in  CICM::RemotePort remote_port,
                in  CICM::SymKey key_ref,
                in  CICM::SymEncrAlgorithmId algorithm,
                out CICM::DecryptBypass::Conduit conduit_ref
        );
```

Create conduit to selectively decrypt or bypass a stream of data.

Remarks:

Both decrypted and bypassed data are read from the same stream using the appropriate decrypt() call; the entity reading from this stream must distinguish between decrypted and bypassed data, if necessary.

Parameters:

[in]	*remote_port*	Remote port.
[in]	*key_ref*	Reference to decryption key.
[in]	*algorithm*	Decryption algorithm/mode.
[out]	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::DecryptBypass::ChannelManager::create_decrypt_bypass_controller()

```
        CICM::Status create_decrypt_bypass_controller(
                in  CICM::LocalPort local_port,
                in  CICM::RemotePort remote_port,
                in  CICM::SymKey key_ref,
                in  CICM::SymEncrAlgorithmId algorithm,
                out CICM::DecryptBypass::Controller controller_ref
        );
```

Create controller to configure and control a decrypt with bypass channel.

Remarks:

In some cases, hosts may depend upon separate processes to control and use a channel. This method returns the channel controller and must be called before the corresponding stream is retrieved.

Parameters:

`[in]`	*local_port*	Local port.
`[in]`	*remote_port*	Remote port.
`[in]`	*key_ref*	Reference to decryption key.
`[in]`	*algorithm*	Decryption algorithm/mode.
`[out]`	*controller_ref*	Reference to resulting controller.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::DecryptBypass::ChannelManager::get_decrypt_bypass_stream()

```
        CICM::Status get_decrypt_bypass_stream(
                in  CICM::LocalPort local_port,
                out CICM::DecryptBypass::Stream stream_ref
        );
```

Create stream associated with previously created controller to receive transformed data.

Remarks:

Both decrypted and bypassed data are read from the same stream using the appropriate decrypt() call; the entity reading from this stream must distinguish between decrypted and bypassed data, if necessary.

Parameters:

`[in]`	*local_port*	Local port.
`[out]`	*stream_ref*	Reference to resulting stream.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_LOCAL_PORT_INVALID, S_LOCAL_PORT_INCOMPATIBLE, S_LOCAL_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_NOT_FOUND

3.5.12.2 Interface CICM::DecryptBypass::Stream

Interface CICM::DecryptBypass::Stream

```
interface Stream : CICM::Decrypt::Stream {};
```

CICM::DecryptBypass::Stream supports decryption and selective bypass operations between two independent security domains. The resulting stream is capable of accepting transformed/bypassed data,

but not managing the channel. It is created by calling
CICM::ChannelManager::get_decrypt_bypass_stream.

Remarks:

> CICM does not specify the structure of the data that is read from a selective bypass channel which may contain special formatting to indicate which subset of the data was bypassed. The Implementation Conformance Statement (see Section 4, Conformance and Extensions) must reference a standard format or define a module developer-specific format implemented by the module for data read from selective bypass channels.

3.5.12.2.1 CICM::DecryptBypass::Stream Inheritance

CICM::DecryptBypass::Stream inherits from: CICM::Decrypt::Stream.

3.5.12.3 Interface CICM::DecryptBypass::Controller

Interface CICM::DecryptBypass::Controller

```
interface Controller : CICM::Decrypt::Controller {
```

CICM::DecryptBypass::Controller supports decryption and selective bypass operations between two independent security domains. The resulting controller is capable of managing the channel, but not accepting transformed/bypassed data. It is created by calling
CICM::ChannelManager::create_decrypt_bypass_controller.

3.5.12.3.1 CICM::DecryptBypass::Controller Inheritance

CICM::DecryptBypass::Controller inherits from: CICM::Decrypt::Controller.

3.5.12.4 Interface CICM::DecryptBypass::NegotiatedController

Interface CICM::DecryptBypass::NegotiatedController

```
interface NegotiatedController : CICM::Decrypt::NegotiatedController {
```

CICM::DecryptBypass::NegotiatedController is the negotiated version of
CICM::DecryptBypass::Controller. It is the result of a successful negotiation by
CICM::DecryptBypass::ControllerNegotiator.

3.5.12.4.1 CICM::DecryptBypass::NegotiatedController Inheritance

CICM::DecryptBypass::NegotiatedController inherits from: CICM::Decrypt::NegotiatedController.

3.5.12.5 Interface CICM::DecryptBypass::Conduit

Interface CICM::DecryptBypass::Conduit

```
interface Conduit :
        CICM::Decrypt::Conduit,
        CICM::DecryptBypass::Controller,
        CICM::DecryptBypass::Stream {
```

CICM::DecryptBypass::Conduit supports decryption and selective bypass operations between two independent security domains. The resulting conduit is capable of both managing the channel and accepting transformed/bypassed data. It is created by calling CICM::ChannelManager::create_decrypt_bypass_conduit.

3.5.12.5.1 CICM::DecryptBypass::Conduit Inheritance

CICM::DecryptBypass::Conduit inherits from: CICM::Decrypt::Conduit, CICM::DecryptBypass::Controller and CICM::DecryptBypass::Stream.

3.5.12.6 Interface CICM::DecryptBypass::NegotiatedConduit

Interface CICM::DecryptBypass::NegotiatedConduit

```
interface NegotiatedConduit :
        CICM::Decrypt::NegotiatedConduit,
        CICM::DecryptBypass::NegotiatedController,
        CICM::DecryptBypass::Stream {
```

CICM::DecryptBypass::NegotiatedConduit is the negotiated version of CICM::DecryptBypass::Conduit. It is the result of a successful negotiation by CICM::DecryptBypass::Negotiator.

3.5.12.6.1 CICM::DecryptBypass::NegotiatedConduit Inheritance

CICM::DecryptBypass::NegotiatedConduit inherits from: CICM::Decrypt::NegotiatedConduit, CICM::DecryptBypass::NegotiatedController and CICM::DecryptBypass::Stream.

3.5.12.7 Interface CICM::DecryptBypass::ControllerNegotiator

Interface CICM::DecryptBypass::ControllerNegotiator

```
interface ControllerNegotiator : CICM::Negotiator {
```

CICM::DecryptBypass::ControllerNegotiator initiates a negotiation to establish a shared key with a remote entity that is used to support encryption and selective bypass operations between two independent security domains. The result of a successful negotiation is a CICM::DecryptBypass::NegotiatedController which is capable of managing the channel, but not accepting data for transformation. CICM::DecryptBypass::ControllerNegotiator is created by calling CICM::ChannelManager::negotiate_decrypt_bypass_controller.

3.5.12.7.1 CICM::DecryptBypass::ControllerNegotiator Inheritance

CICM::DecryptBypass::ControllerNegotiator inherits from: CICM::Negotiator.

3.5.12.7.2 CICM::DecryptBypass::ControllerNegotiator Methods

Method CICM::DecryptBypass::ControllerNegotiator::complete()

```
        CICM::Status complete(
                out CICM::DecryptBypass::NegotiatedController controller_ref
        );
```

Complete negotiation and retrieve a negotiated control-only decrypt bypass channel.
Parameters:

`[out]` *controller_ref* Reference to resulting controller.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_NOT_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

3.5.12.8 Interface CICM::DecryptBypass::Negotiator

Interface CICM::DecryptBypass::Negotiator

```
interface Negotiator : CICM::Negotiator {
```

CICM::DecryptBypass::Negotiator initiates a negotiation to establish a shared key with a remote entity that is used to support encryption operations between two independent security domains. Additionally, selective bypass is supported on the same conduit. The result of a successful negotiation is a CICM::DecryptBypass::NegotiatedConduit which is capable of both managing the channel and accepting data for transformation. CICM::DecryptBypass::Negotiator is created by calling CICM::ChannelManager::negotiate_decrypt_bypass_conduit.

3.5.12.8.1 CICM::DecryptBypass::Negotiator Inheritance

CICM::DecryptBypass::Negotiator inherits from: CICM::Negotiator.

3.5.12.8.2 CICM::DecryptBypass::Negotiator Methods

Method CICM::DecryptBypass::Negotiator::complete()

```
        CICM::Status complete(
                out     CICM::DecryptBypass::NegotiatedConduit conduit_ref
        );
```

Complete negotiation and retrieve a negotiated decrypt bypass conduit.

Parameters:

`[out]` *conduit_ref* Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_NEGOTIATION_ABORTED, S_NEGOTIATION_FAILURE, S_NEGOTIATION_NOT_IN_PROGRESS, S_NEGOTIATION_TIMEOUT, S_CERT_LOCAL_INVALID, S_CERT_LOCAL_EXPIRED, S_CERT_REMOTE_INVALID, S_CERT_REMOTE_EXPIRED, S_CERT_REMOTE_PATH, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET

3.5.13 RANDOM, PSEUDORANDOM AND KEYSTREAM CHANNEL MANAGEMENT

Namespace CICM::Emit

```
module Emit {
```

The CICM::Emit namespace contains interfaces that generate data originating in a cryptographic module such as random, pseudorandom, and keystream data.

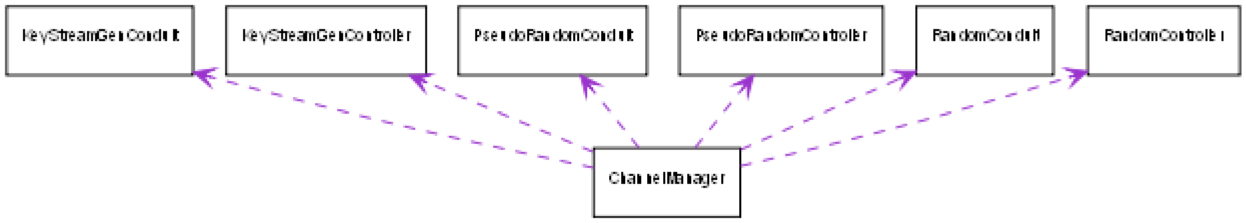


Figure 38. Interface Relationship Diagram for Keystream Generation and Random Channels

3.5.13.1 Interface CICM::Emit::ChannelManager

Interface CICM::Emit::ChannelManager

```
interface ChannelManager {
```

CICM::Emit::ChannelManager is an abstraction inherited by CICM::ChannelManager that supports the creation of conduits and controllers to generate keystream, pseudorandom, and random data. See CICM::ChannelManager for additional information.

3.5.13.1.1 CICM::Emit::ChannelManager Methods

Method CICM::Emit::ChannelManager::create_key_stream_gen_controller()

```
        CICM::Status create_key_stream_gen_controller(
                in  CICM::RemotePort remote_port,
                in  CICM::SymKey key_ref,
                in  CICM::SymEncrAlgorithmId algorithm,
                out     CICM::Emit::KeyStreamGenController controller_ref
        );
```

Create controller to generate keystream.

Parameters:

[in]	*remote_port*	Remote port.
[in]	*key_ref*	Reference to keystream generation key.
[in]	*algorithm*	Keystream generation algorithm/mode.
[out]	*controller_ref*	Reference to resulting controller.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Emit::ChannelManager::create_pseudorandom_controller()

```
        CICM::Status create_pseudorandom_controller(
             in  CICM::RemotePort remote_port,
             in  CICM::SymKey seed,
             out CICM::Emit::PseudoRandomController controller_ref
     );
```

Create controller to generate pseudorandom data.

Parameters:

`[in]`	*remote_port*	Remote port.
`[in]`	*seed*	Seed material for pseudorandom generator.
`[out]`	*controller_ref*	Reference to resulting controller.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX, S_INSUFFICIENT_ENTROPY

Method CICM::Emit::ChannelManager::create_random_controller()

```
     CICM::Status create_random_controller(
             in  CICM::RemotePort remote_port,
             out CICM::Emit::RandomController controller_ref
     );
```

Create controller to generate random data.

Parameters:

`[in]`	*remote_port*	Remote port.
`[out]`	*controller_ref*	Reference to resulting controller.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_REMOTE_PORT_INVALID, S_REMOTE_PORT_IN_USE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX, S_INSUFFICIENT_ENTROPY

Method CICM::Emit::ChannelManager::create_key_stream_gen_conduit()

```
     CICM::Status create_key_stream_gen_conduit(
             in  CICM::SymKey key_ref,
             in  CICM::SymEncrAlgorithmId algorithm,
             out     CICM::Emit::KeyStreamGenConduit conduit_ref
     );
```

Create conduit to generate keystream.

Parameters:

`[in]`	*key_ref*	Reference to keystream generation key.
`[in]`	*algorithm*	Keystream generation algorithm/mode.
`[out]`	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_INVALID, S_KEY_EXPIRED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Emit::ChannelManager::create_pseudorandom_conduit()

```
        CICM::Status create_pseudorandom_conduit(
                in  CICM::SymKey seed,
                out CICM::Emit::PseudoRandomConduit conduit_ref
        );
```

Create conduit to generate pseudorandom data.

Parameters:

`[in]`	*seed*	Seed material for pseudorandom generator.
`[out]`	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_INVALID, S_KEY_EXPIRED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX, S_INSUFFICIENT_ENTROPY

Method CICM::Emit::ChannelManager::create_random_conduit()

```
        CICM::Status create_random_conduit(
                out CICM::Emit::RandomConduit conduit_ref
        );
```

Create conduit to generate random data.

Parameters:

`[out]`	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX, S_INSUFFICIENT_ENTROPY

3.5.13.2 Interface CICM::Emit::GetStream

Interface CICM::Emit::GetStream

```
interface GetStream : CICM::ReadStream {
```

CICM::Emit::GetStream is an abstraction inherited by conduits in the CICM::Emit namespace that allows data to be read from the stream.

3.5.13.2.1 CICM::Emit::GetStream Inheritance

CICM::Emit::GetStream inherits from: CICM::ReadStream.

3.5.13.2.2 CICM::Emit::GetStream Methods

Method CICM::Emit::GetStream::get()

```
        CICM::Status get(
                in  CICM::UInt32 length,
                out CICM::Buffer buffer
        );
```

Reads a buffer of data from the module. The method blocks until data becomes available.

Parameters:

`[in]`	*length*	Number of bytes to retrieve.
`[out]`	*buffer*	Buffer of data read from stream.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_WRAPPED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_IO_ERROR, S_CHANNEL_BUFFER_LEN, S_INSUFFICIENT_ENTROPY

Method CICM::Emit::GetStream::get_non_blocking()

```
        CICM::Status get_non_blocking(
                in  CICM::UInt32 length,
                out CICM::Buffer buffer,
                in  CICM::TransId transaction_id
        );
```

Registers a buffer into which transformed data will be copied, and then control immediately returns to the caller. The size of the allocated buffer and length of the resulting transformed data is encapsulated in the buffer parameter. The caller may use the CICM::Emit::GetStream::get_poll method to proactively poll the channel to determine the status of the operation. The caller is responsible for maintaining any necessary metadata associated with the transaction_id parameter. Memory responsibilities and calling conventions shall follow the appropriate IDL language mapping conventions.

Parameters:

`[in]`	*length*	Number of bytes to retrieve.
`[out]`	*buffer*	Transformed data.
`[in]`	*transaction_id*	Unique transaction id that will be used by the CICM::Emit::GetStream::get_poll method to determine to which buffer the poll status applies.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_WRAPPED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_IO_ERROR, S_CHANNEL_BUFFER_LEN, S_INSUFFICIENT_ENTROPY

Method CICM::Emit::GetStream::get_poll()

```
        CICM::Status get_poll(
                in  CICM::TransId transaction_id,
                out CICM::ReadStream::ReadStatus status
        );
```

Returns the status of the non-blocking get operation specified by the transaction_id parameter. Upon completion of the operation, the caller must use the metadata associated with the transaction_id parameter to determine which buffer has been populated. Memory responsibilities and calling conventions shall follow the appropriate IDL language mapping conventions.

Parameters:

`[in]`	*transaction_id*	Unique transaction id previously specified to the CICM::Emit::GetStream::get_non_blocking method that allows the poll operation to determine to which buffer the poll status applies.
`[out]`	*status*	Status of the non-blocking operation corresponding to the transaction_id parameter.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_KEY_WRAPPED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_PEER_RESET, S_CHANNEL_IO_ERROR

3.5.13.3 Interface CICM::Emit::Controller

Interface CICM::Emit::Controller

```
interface Controller : CICM::Controller {
```

CICM::Emit::Controller is an abstraction from which all other controllers in the CICM::Emit namespace inherit.

3.5.13.3.1 CICM::Emit::Controller Inheritance

CICM::Emit::Controller inherits from: CICM::Controller.

3.5.13.3.2 CICM::Emit::Controller Attributes

Attribute CICM::Emit::Controller::remote_port

```
        readonly attribute CICM::RemotePort remote_port;
```

The remote port associated with this controller.

3.5.13.4 Interface CICM::Emit::RandomController

Interface CICM::Emit::RandomController

```
interface RandomController : CICM::Emit::Controller {};
```

CICM::Emit::RandomController supports creating a channel to read random data from a module. The resulting controller is capable of managing the channel, but not reading random data. It is created by calling CICM::ChannelManager::create_random_controller.

3.5.13.4.1 CICM::Emit::RandomController Inheritance

CICM::Emit::RandomController inherits from: CICM::Emit::Controller.

3.5.13.5 Interface CICM::Emit::RandomConduit

Interface CICM::Emit::RandomConduit

```
interface RandomConduit :
        CICM::Conduit,
        CICM::Emit::GetStream {
```

CICM::Emit::RandomConduit supports reading random data from a module. The resulting conduit is capable of both managing the channel and reading random data. It is created by calling CICM::ChannelManager::create_random_conduit.

3.5.13.5.1 CICM::Emit::RandomConduit Inheritance

CICM::Emit::RandomConduit inherits from: CICM::Conduit and CICM::Emit::GetStream.

3.5.13.6 Interface CICM::Emit::PseudoRandomController

Interface CICM::Emit::PseudoRandomController

```
interface PseudoRandomController :
        CICM::SymKeyController,
        CICM::Emit::Controller {
```

CICM::Emit::PseudoRandomController supports creating a channel to read pseudorandom data from a module. The resulting controller is capable of managing the channel, but not reading pseudorandom data. It is created by calling CICM::ChannelManager::create_pseudorandom_controller.

3.5.13.6.1 CICM::Emit::PseudoRandomController Inheritance

CICM::Emit::PseudoRandomController inherits from: CICM::SymKeyController and CICM::Emit::Controller.

3.5.13.7 Interface CICM::Emit::PseudoRandomConduit

Interface CICM::Emit::PseudoRandomConduit

```
interface PseudoRandomConduit :
        CICM::Conduit,
        CICM::SymKeyController,
        CICM::Emit::GetStream {
```

CICM::Emit::PseudoRandomConduit supports reading pseudorandom data from a module. The resulting conduit is capable of both managing the channel and reading pseudorandom data. It is created by calling CICM::ChannelManager::create_pseudorandom_conduit.

3.5.13.7.1 CICM::Emit::PseudoRandomConduit Inheritance

CICM::Emit::PseudoRandomConduit inherits from: CICM::Conduit, CICM::SymKeyController and CICM::Emit::GetStream.

3.5.13.8 Interface CICM::Emit::KeyStreamGenController

Interface CICM::Emit::KeyStreamGenController

```
interface KeyStreamGenController :
        CICM::SymKeyController,
        CICM::GenVectorController,
        CICM::Emit::Controller {
```

CICM::Emit::KeyStreamGenController supports creating a channel to read keystream from a module. The resulting controller is capable of managing the channel, but not reading keystream. It is created by calling CICM::ChannelManager::create_key_stream_gen_controller.

3.5.13.8.1 CICM::Emit::KeyStreamGenController Inheritance

CICM::Emit::KeyStreamGenController inherits from: CICM::SymKeyController, CICM::GenVectorController and CICM::Emit::Controller.

3.5.13.9 Interface CICM::Emit::KeyStreamGenConduit

Interface CICM::Emit::KeyStreamGenConduit

```
interface KeyStreamGenConduit :
        CICM::Conduit,
        CICM::SymKeyController,
        CICM::GenVectorController,
        CICM::Emit::GetStream {
```

CICM::Emit::KeyStreamGenConduit supports reading keystream from a module. The resulting conduit is capable of both managing the channel and reading keystream. It is created by calling CICM::ChannelManager::create_key_stream_gen_conduit.

3.5.13.9.1 CICM::Emit::KeyStreamGenConduit Inheritance

CICM::Emit::KeyStreamGenConduit inherits from: CICM::Conduit, CICM::SymKeyController, CICM::GenVectorController and CICM::Emit::GetStream.

3.5.14 INTEGRITY CHANNEL MANAGEMENT

Namespace CICM::Answer

```
module Answer {
```

The CICM::Answer namespace contains interfaces that support cryptographic operations that return an "answer" such a hash or a signature within a single security domain.

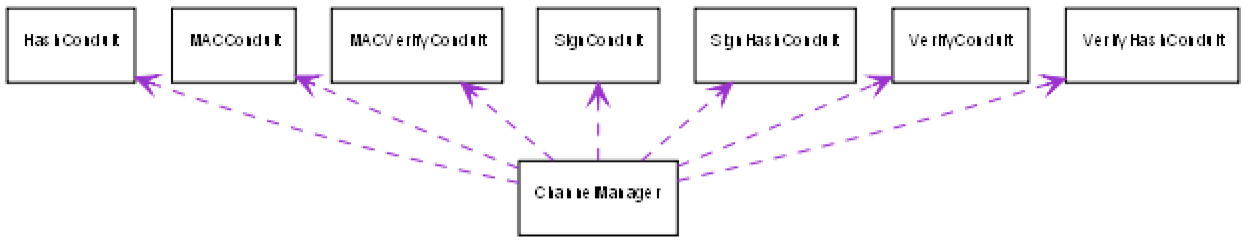


Figure 39. Interface Relationship Diagram for Channels that Return an "Answer"

3.5.14.1 Interface CICM::Answer::ChannelManager

Interface CICM::Answer::ChannelManager

```
interface ChannelManager {
```

CICM::Answer::ChannelManager is an abstraction inherited by CICM::ChannelManager that supports the creation of conduits to sign, MAC, and hash data. See CICM::ChannelManager for additional information.

3.5.14.1.1 CICM::Answer::ChannelManager Methods

Method CICM::Answer::ChannelManager::create_hash_conduit()

```
        CICM::Status create_hash_conduit(
                in  CICM::HashAlgorithmId algorithm,
                out CICM::Answer::HashConduit conduit_ref
        );
```

Create conduit to calculate and generate a hash value.

Parameters:

`[in]` *algorithm* Hash algorithm.
`[out]` *conduit_ref* Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Answer::ChannelManager::create_mac_conduit()

```
        CICM::Status create_mac_conduit(
                in  CICM::SymKey key_ref,
                in  CICM::SymMacAlgorithmId algorithm,
                out CICM::Answer::MACConduit conduit_ref
        );
```

Create conduit to calculate and generate a MAC.

Parameters:

`[in]` *key_ref* Reference to MAC key.
`[in]` *algorithm* MAC algorithm.
`[out]` *conduit_ref* Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_INVALID, S_KEY_EXPIRED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Answer::ChannelManager::create_mac_verify_conduit()

```
CICM::Status create_mac_verify_conduit(
        in  CICM::SymKey key_ref,
        in  CICM::SymMacAlgorithmId algorithm,
        out CICM::Answer::MACVerifyConduit conduit_ref
);
```

Create conduit to verify a MAC.

Parameters:

`[in]`	*key_ref*	Reference to verification key.
`[in]`	*algorithm*	MAC algorithm.
`[out]`	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_INVALID, S_KEY_EXPIRED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Answer::ChannelManager::create_sign_conduit()

```
CICM::Status create_sign_conduit(
        in  CICM::AsymKey key_ref,
        in  CICM::AsymSigAlgorithmId algorithm,
        out CICM::Answer::SignConduit conduit_ref
);
```

Create conduit to calculate and generate a signature.

Parameters:

`[in]`	*key_ref*	Reference to signature key.
`[in]`	*algorithm*	Signature algorithm.
`[out]`	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Answer::ChannelManager::create_sign_hash_conduit()

```
        CICM::Status create_sign_hash_conduit(
                in  CICM::AsymKey key_ref,
                in  CICM::AsymSigAlgorithmId algorithm,
                out CICM::Answer::SignHashConduit conduit_ref
        );
```

Create conduit to calculate and generate a signature accepting a previously generated hash value as input.

Parameters:

[in]	*key_ref*	Reference to signature key.
[in]	*algorithm*	Signature algorithm.
[out]	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Answer::ChannelManager::create_verify_conduit()

```
        CICM::Status create_verify_conduit(
                in  CICM::AsymKey key_ref,
                in  CICM::AsymSigAlgorithmId algorithm,
                out CICM::Answer::VerifyConduit conduit_ref
        );
```

Create conduit to verify a signature.

Parameters:

[in]	*key_ref*	Reference to verification key.
[in]	*algorithm*	Verification algorithm.
[out]	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Answer::ChannelManager::create_verify_hash_conduit()

```
        CICM::Status create_verify_hash_conduit(
                in  CICM::AsymKey key_ref,
                in  CICM::AsymSigAlgorithmId algorithm,
                out CICM::Answer::VerifyHashConduit conduit_ref
        );
```

Create conduit to verify a signature accepting a previously generated hash value as input.

Parameters:

`[in]`	*key_ref*	Reference to verification key.
`[in]`	*algorithm*	Verification algorithm.
`[out]`	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

3.5.14.2 Interface CICM::Answer::PutStream

Interface CICM::Answer::PutStream

```
interface PutStream : CICM::Stream {
```

A stream that can write data to a module.

3.5.14.2.1 CICM::Answer::PutStream Inheritance

CICM::Answer::PutStream inherits from: CICM::Stream.

3.5.14.2.2 CICM::Answer::PutStream Methods

Method CICM::Answer::PutStream::put()

```
        CICM::Status put(
                in  CICM::Buffer buffer
        );
```

Sends data to the module for transformation.

Remarks:

This method blocks until the data is sent to the module.

Parameters:

`[in]` *buffer* Buffer to write to the module.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_IO_ERROR, S_CHANNEL_DATA_INVALID, S_CHANNEL_DATA_INVALID_LEN

3.5.14.3 Interface CICM::Answer::HashConduit

Interface CICM::Answer::HashConduit

```
interface HashConduit :
        CICM::Conduit,
        CICM::Answer::PutStream {
```

CICM::Answer::HashConduit supports hashing operations within a single security domain. It is created by calling CICM::ChannelManager::create_hash_conduit.

Remarks:

Keyed hashes are supported by MAC channels.

3.5.14.3.1 CICM::Answer::HashConduit Inheritance

CICM::Answer::HashConduit inherits from: CICM::Conduit and CICM::Answer::PutStream.

3.5.14.3.2 CICM::Answer::HashConduit Attributes

Attribute CICM::Answer::HashConduit::algorithm

```
        readonly attribute CICM::HashAlgorithmId algorithm;
```

Algorithm used to compute the hash.

3.5.14.3.3 CICM::Answer::HashConduit Methods

Method CICM::Answer::HashConduit::end_get_hash()

```
        CICM::Status end_get_hash(
                out     HashBuffer hash
        );
```

Direct the module to compute and output the message digest value, and reset the conduit to accept additional data.

Parameters:

`[out]` *hash* Resulting hash.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR

3.5.14.4 Interface CICM::Answer::MACConduit

Interface CICM::Answer::MACConduit

```
interface MACConduit :
        CICM::AbstractMACConduit,
        CICM::Answer::PutStream {
```

CICM::Answer::MACConduit supports message authentication code operations within a single security domain. It is created by calling CICM::ChannelManager::create_mac_conduit.

3.5.14.4.1 CICM::Answer::MACConduit Inheritance

CICM::Answer::MACConduit inherits from: CICM::AbstractMACConduit and CICM::Answer::PutStream.

3.5.14.5 Interface CICM::Answer::MACVerifyConduit

Interface CICM::Answer::MACVerifyConduit
```
interface MACVerifyConduit :
        CICM::AbstractMACVerifyConduit,
        CICM::Answer::PutStream {
```
CICM::Answer::MACVerifyConduit supports message authentication code verification operations within a single security domain. It is created by calling CICM::ChannelManager::create_mac_verify_conduit.

3.5.14.5.1 CICM::Answer::MACVerifyConduit Inheritance

CICM::Answer::MACVerifyConduit inherits from: CICM::AbstractMACVerifyConduit and CICM::Answer::PutStream.

3.5.14.6 Interface CICM::Answer::SignConduit

Interface CICM::Answer::SignConduit
```
interface SignConduit :
        CICM::AbstractSignConduit,
        CICM::Answer::PutStream {
```
CICM::Answer::SignConduit supports signature operations within a single security domain. It is created by calling CICM::ChannelManager::create_sign_conduit.

3.5.14.6.1 CICM::Answer::SignConduit Inheritance

CICM::Answer::SignConduit inherits from: CICM::AbstractSignConduit and CICM::Answer::PutStream.

3.5.14.7 Interface CICM::Answer::SignHashConduit

Interface CICM::Answer::SignHashConduit
```
interface SignHashConduit : CICM::Answer::SignConduit {};
```
CICM::Answer::SignHashConduit supports signature operations accepting a pre-generated hash value within a single security domain. It is created by calling CICM::ChannelManager::create_sign_hash_conduit.

3.5.14.7.1 CICM::Answer::SignHashConduit Inheritance

CICM::Answer::SignHashConduit inherits from: CICM::Answer::SignConduit.

3.5.14.8 Interface CICM::Answer::VerifyConduit

Interface CICM::Answer::VerifyConduit
```
interface VerifyConduit :
        CICM::AbstractSigVerifyConduit,
        CICM::Answer::PutStream {
```

CICM::Answer::VerifyConduit supports verification operations within a single security domain. It is created by calling CICM::ChannelManager::create_verify_conduit.

3.5.14.8.1 CICM::Answer::VerifyConduit Inheritance

CICM::Answer::VerifyConduit inherits from: CICM::AbstractSigVerifyConduit and CICM::Answer::PutStream.

3.5.14.9 Interface CICM::Answer::VerifyHashConduit

Interface CICM::Answer::VerifyHashConduit

```
interface VerifyHashConduit : CICM::Answer::VerifyConduit {};
```

CICM::Answer::VerifyHashConduit supports verification operations accepting a pre-generated hash value within a single security domain. It is created by calling CICM::ChannelManager::create_verify_hash_conduit.

3.5.14.9.1 CICM::Answer::VerifyHashConduit Inheritance

CICM::Answer::VerifyHashConduit inherits from: CICM::Answer::VerifyConduit.

3.5.15 SINGLE-DOMAIN CHANNEL MANAGEMENT

Namespace CICM::Coprocessor

```
module Coprocessor {
```

The CICM::Coprocessor namespace contains interfaces that support encryption/decryption operations within a single security domain.

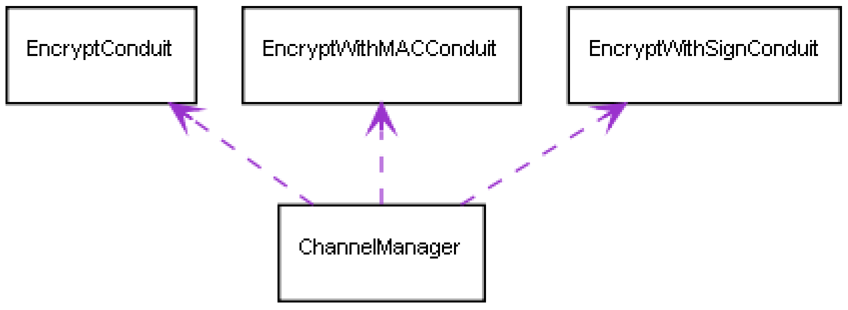


Figure 40. Interface Relationship Diagram for Single-Domain Encryption Channels

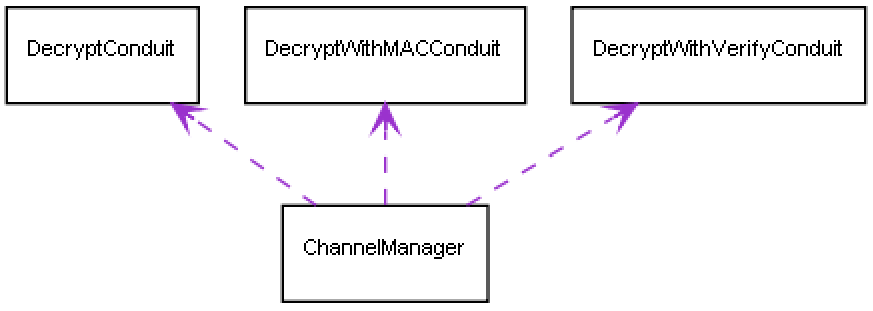


Figure 41. Interface Relationship Diagram for Single-Domain Decryption Channels

3.5.15.1 Interface CICM::Coprocessor::ChannelManager

Interface CICM::Coprocessor::ChannelManager

```
interface ChannelManager {
```

CICM::Coprocessor::ChannelManager is an abstraction inherited by CICM::ChannelManager that supports the creation of conduits to encrypt and decrypt data within a single security domain. See CICM::ChannelManager for additional information.

3.5.15.1.1 CICM::Coprocessor::ChannelManager Methods

Method CICM::Coprocessor::ChannelManager::create_coprocessor_encrypt_conduit()

```
        CICM::Status create_coprocessor_encrypt_conduit(
                in  CICM::SymKey key_ref,
                in  CICM::SymEncrAlgorithmId algorithm,
                out CICM::Coprocessor::EncryptConduit conduit_ref
        );
```

Create conduit to encrypt a stream of data within a single security domain.

Parameters:

`[in]`	*key_ref*	Reference to encryption key.
`[in]`	*algorithm*	Encryption algorithm/mode.
`[out]`	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_INVALID, S_KEY_EXPIRED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Coprocessor::ChannelManager::create_coprocessor_encrypt_with_mac_conduit()

```
        CICM::Status create_coprocessor_encrypt_with_mac_conduit(
                in  CICM::SymKey mac_key_ref,
                in  CICM::SymKey encrypt_key_ref,
                in  CICM::SymMacAlgorithmId mac_algorithm,
                in  CICM::SymEncrAlgorithmId encrypt_algorithm,
                out CICM::Coprocessor::EncryptWithMACConduit conduit_ref
        );
```

Create conduit to MAC and encrypt a stream of data within a single security domain.

Parameters:

`[in]`	*mac_key_ref*	Reference to MAC key.
`[in]`	*encrypt_key_ref*	Reference to encryption key.
`[in]`	*mac_algorithm*	MAC algorithm.
`[in]`	*encrypt_algorithm*	Encryption algorithm/mode.

[out] *conduit_ref* Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_INVALID, S_KEY_EXPIRED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Coprocessor::ChannelManager::create_coprocessor_encrypt_with_sign_conduit()

```
CICM::Status create_coprocessor_encrypt_with_sign_conduit(
        in  CICM::AsymKey sign_key_ref,
        in  CICM::SymKey encrypt_key_ref,
        in  CICM::AsymSigAlgorithmId sign_algorithm,
        in  CICM::SymEncrAlgorithmId encrypt_algorithm,
        out CICM::Coprocessor::EncryptWithSignConduit conduit_ref
);
```

Create conduit to sign and encrypt a stream of data within a single security domain.

Parameters:

[in]	*sign_key_ref*	Reference to signature key.
[in]	*encrypt_key_ref*	Reference to encryption key.
[in]	*sign_algorithm*	Signature algorithm.
[in]	*encrypt_algorithm*	Encryption algorithm/mode.
[out]	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Coprocessor::ChannelManager::create_coprocessor_decrypt_conduit()

```
CICM::Status create_coprocessor_decrypt_conduit(
        in  CICM::SymKey key_ref,
        in  CICM::SymEncrAlgorithmId algorithm,
        out CICM::Coprocessor::DecryptConduit conduit_ref
);
```

Create conduit to decrypt a stream of data within a single security domain.

Parameters:

[in]	*key_ref*	Reference to decryption key.
[in]	*algorithm*	Decryption algorithm/mode.
[out]	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_INVALID, S_KEY_EXPIRED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Coprocessor::ChannelManager::create_coprocessor_decrypt_with_mac_conduit()

```
        CICM::Status create_coprocessor_decrypt_with_mac_conduit(
                in  CICM::SymKey mac_key_ref,
                in  CICM::SymKey decrypt_key_ref,
                in  CICM::SymMacAlgorithmId mac_algorithm,
                in  CICM::SymEncrAlgorithmId encrypt_algorithm,
                out CICM::Coprocessor::DecryptWithMACConduit conduit_ref
        );
```

Create conduit to MAC verify and decrypt a stream of data within a single security domain.

Parameters:

`[in]`	*mac_key_ref*	Reference to MAC key.
`[in]`	*decrypt_key_ref*	Reference to decryption key.
`[in]`	*mac_algorithm*	MAC algorithm.
`[in]`	*encrypt_algorithm*	Encryption algorithm/mode.
`[out]`	*conduit_ref*	Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_INVALID, S_KEY_EXPIRED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

Method CICM::Coprocessor::ChannelManager::create_coprocessor_decrypt_with_verify_conduit()

```
        CICM::Status create_coprocessor_decrypt_with_verify_conduit(
                in  CICM::AsymKey verify_key_ref,
                in  CICM::SymKey decrypt_key_ref,
                in  CICM::AsymSigAlgorithmId verify_algorithm,
                in  CICM::SymEncrAlgorithmId decrypt_algorithm,
                out CICM::Coprocessor::DecryptWithVerifyConduit conduit_ref
        );
```

Create conduit to verify and decrypt a stream of data within a single security domain.

Parameters:

`[in]`	*verify_key_ref*	Reference to verification key.
`[in]`	*decrypt_key_ref*	Reference to decryption key.
`[in]`	*verify_algorithm*	Verification algorithm.
`[in]`	*decrypt_algorithm*	Decryption algorithm/mode.

`[out]` *conduit_ref*   Reference to resulting conduit.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_USED_INVALID, S_KEY_USED_EXPIRED, S_KEY_USED_WRAPPED, S_KEY_USED_CONTEXT, S_KEY_USED_COMPONENT_NOT_AVAIL, S_KEY_INVALID, S_KEY_EXPIRED, S_ALGO_INVALID, S_ALGO_INCOMPATIBLE, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_MAX

3.5.15.2 Interface CICM::Coprocessor::Stream

Interface CICM::Coprocessor::Stream

```
interface Stream : CICM::Stream {
```

CICM::Coprocessor::Stream is an abstraction inherited by all conduits in the CICM::Coprocessor namespace.

3.5.15.2.1 CICM::Coprocessor::Stream Inheritance

CICM::Coprocessor::Stream inherits from: CICM::Stream.

3.5.15.2.2 CICM::Coprocessor::Stream Methods

Method CICM::Coprocessor::Stream::get_final_buffer()

```
        CICM::Status get_final_buffer(
                out CICM::Buffer buffer
        );
```

Returns the final block of transformed data, if available. The method blocks until data becomes available.

Remarks:

In certain cases, it may be necessary to retrieve the last transformed block of data, if, for example, a partial block remains after the last full block was retrieved. This call **must** be called after all data is supplied to the module and **must** precede any end_get_* calls.

Parameters:

`[out]` *buffer* Buffer of data read from stream.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_KEY_INVALID, S_KEY_EXPIRED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT

3.5.15.3 Interface CICM::Coprocessor::EncryptConduit

Interface CICM::Coprocessor::EncryptConduit

```
interface EncryptConduit :
        CICM::Conduit,
        CICM::SymKeyController,
        CICM::GenVectorController,
        CICM::ResyncController,
        CICM::Coprocessor::Stream {
```

CICM::Coprocessor::EncryptConduit supports encryption operations within a single security domain. The resulting conduit is capable of managing the channel, accepting data for transformation, and receiving the result. It is created by calling CICM::ChannelManager::create_coprocessor_encrypt_conduit.

3.5.15.3.1 CICM::Coprocessor::EncryptConduit Inheritance

CICM::Coprocessor::EncryptConduit inherits from: CICM::Conduit, CICM::SymKeyController, CICM::GenVectorController, CICM::ResyncController and CICM::Coprocessor::Stream.

3.5.15.3.2 CICM::Coprocessor::EncryptConduit Methods

Method CICM::Coprocessor::EncryptConduit::encrypt()

```
        CICM::Status encrypt(
                in  CICM::Buffer plaintext,
                out CICM::Buffer ciphertext
        );
```

Send plaintext to the module to be encrypted, receiving the ciphertext resulting from the transformation as the result.

Parameters:

`[in]` *plaintext* Plaintext to encrypt.
`[out]` *ciphertext* Ciphertext resulting from encryption operation.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_IO_ERROR, S_CHANNEL_DATA_INVALID, S_CHANNEL_DATA_INVALID_LEN, S_CHANNEL_BUFFER_LEN

3.5.15.4 Interface CICM::Coprocessor::EncryptWithMACConduit

Interface CICM::Coprocessor::EncryptWithMACConduit

```
interface EncryptWithMACConduit :
        CICM::AbstractMACConduit,
        CICM::Coprocessor::EncryptConduit {
```

CICM::Coprocessor::EncryptWithMACConduit supports encryption with MAC operations within a single security domain. The resulting conduit is capable of managing the channel, accepting data for transformation, and receiving the result (both ciphertext and a MAC value). It is created by calling CICM::ChannelManager::create_coprocessor_encrypt_with_mac_conduit.

3.5.15.4.1 CICM::Coprocessor::EncryptWithMACConduit Inheritance

CICM::Coprocessor::EncryptWithMACConduit inherits from: CICM::AbstractMACConduit and CICM::Coprocessor::EncryptConduit.

3.5.15.5 Interface CICM::Coprocessor::EncryptWithSignConduit

Interface CICM::Coprocessor::EncryptWithSignConduit

```
interface EncryptWithSignConduit :
        CICM::AbstractSignConduit,
        CICM::Coprocessor::EncryptConduit {
```

CICM::Coprocessor::EncryptWithSignConduit supports encryption with signature operations within a single security domain. The resulting conduit is capable of managing the channel, accepting data for transformation, and receiving the result (both ciphertext and a signature). It is created by calling CICM::ChannelManager::create_coprocessor_encrypt_with_sign_conduit.

3.5.15.5.1 CICM::Coprocessor::EncryptWithSignConduit Inheritance

CICM::Coprocessor::EncryptWithSignConduit inherits from: CICM::AbstractSignConduit and CICM::Coprocessor::EncryptConduit.

3.5.15.6 Interface CICM::Coprocessor::DecryptConduit

Interface CICM::Coprocessor::DecryptConduit

```
interface DecryptConduit :
        CICM::Conduit,
        CICM::SymKeyController,
        CICM::SetVectorController,
        CICM::ResyncController,
        CICM::Coprocessor::Stream {
```

CICM::Coprocessor::DecryptConduit supports decryption operations within a single security domain. The resulting conduit is capable of managing the channel, accepting data for transformation, and receiving the result. It is created by calling CICM::ChannelManager::create_coprocessor_decrypt_conduit.

3.5.15.6.1 CICM::Coprocessor::DecryptConduit Inheritance

CICM::Coprocessor::DecryptConduit inherits from: CICM::Conduit, CICM::SymKeyController, CICM::SetVectorController, CICM::ResyncController and CICM::Coprocessor::Stream.

3.5.15.6.2 CICM::Coprocessor::DecryptConduit Methods

Method CICM::Coprocessor::DecryptConduit::decrypt()

```
        CICM::Status decrypt(
                in  CICM::Buffer ciphertext,
                out CICM::Buffer plaintext
        );
```

Send ciphertext to the module to be decrypted, receiving the plaintext resulting from the transformation as the result.

Parameters:

`[in]` *ciphertext* Ciphertext to decrypt.
`[out]` *plaintext* Plaintext resulting from decryption operation.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_ERROR, S_CHANNEL_IO_ERROR, S_CHANNEL_DATA_INVALID, S_CHANNEL_DATA_INVALID_LEN, S_CHANNEL_BUFFER_LEN

3.5.15.7 Interface CICM::Coprocessor::DecryptWithMACConduit

Interface CICM::Coprocessor::DecryptWithMACConduit

```
interface DecryptWithMACConduit :
        CICM::AbstractMACVerifyConduit,
        CICM::Coprocessor::DecryptConduit {
```

CICM::Coprocessor::DecryptWithMACConduit supports encryption with MAC verification operations within a single security domain. The resulting conduit is capable of managing the channel, accepting data for transformation, and receiving the result (both plaintext and an indication as to whether verification succeeded or failed). It is created by calling CICM::ChannelManager::create_coprocessor_decrypt_with_mac_conduit.

3.5.15.7.1 CICM::Coprocessor::DecryptWithMACConduit Inheritance

CICM::Coprocessor::DecryptWithMACConduit inherits from: CICM::AbstractMACVerifyConduit and CICM::Coprocessor::DecryptConduit.

3.5.15.8 Interface CICM::Coprocessor::DecryptWithVerifyConduit

Interface CICM::Coprocessor::DecryptWithVerifyConduit

```
interface DecryptWithVerifyConduit :
        CICM::AbstractSigVerifyConduit,
        CICM::Coprocessor::DecryptConduit {
```

CICM::Coprocessor::DecryptWithVerifyConduit supports encryption with signature verification operations within a single security domain. The resulting conduit is capable of managing the channel, accepting data for transformation, and receiving the result (both plaintext and an indication as to whether verification succeeded or failed). It is created by calling CICM::ChannelManager::create_coprocessor_decrypt_with_verify_conduit.

3.5.15.8.1 CICM::Coprocessor::DecryptWithVerifyConduit Inheritance

CICM::Coprocessor::DecryptWithVerifyConduit inherits from: CICM::AbstractSigVerifyConduit and CICM::Coprocessor::DecryptConduit.

3.5.16 CHANNEL EVENT MANAGEMENT

3.5.16.1 Interface CICM::ChannelEventManager

Interface CICM::ChannelEventManager

```
interface ChannelEventManager {
```

CICM::ChannelEventManager supports registering and unregistering user-defined channel event listeners (CICM::ChannelEventListener) for specific channel events. It is accessed from any channel via its CICM::Channel::event_manager attribute.

Remarks:

In certain cases it may be necessary for a channel to asynchronously notify a client program of an event. Client programs can register to receive channel notifications using CICM::ChannelEventManager. This manager enables a client program to register a listener (callback) method designed to handle a specific condition. The event method prototype provided by the client program is defined in CICM::ChannelEventListener. CICM::ChannelEventListener also defines the conditions that may result in a notification, including: channel failure or channel destruction.

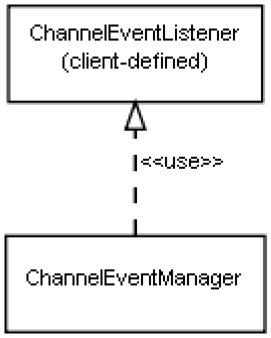


Figure 42. Interface Relationship Diagram for ChannelEventManager

3.5.16.1.1 CICM::ChannelEventManager Methods

Method CICM::ChannelEventManager::register()

```
        CICM::Status register(
                in  CICM::ChannelEventListener::ChannelEvent event,
                in  CICM::ChannelEventListener listener
        );
```

Registers the listener for a specific channel event.

Remarks:

The provided listener applies only to the client program from which the registration is initiated.

Parameters:

`[in]` *event* Event for which this listener is being registered.

`[in]` *listener* Listener that will receive a notification about the specified event.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_EVENT_REGISTERED, S_EVENT_NOT_SUPPORTED

Method CICM::ChannelEventManager::unregister()

```
        CICM::Status unregister(
                in  CICM::ChannelEventListener::ChannelEvent event
        );
```

Unregisters the listener associated with the specified event.

Remarks:

The listener associated with the specified event is only unregistered from the client program from which this method is called.

Parameters:

`[in]` *event* Event that will no longer have a listener associated with it.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_EVENT_NOT_REGISTERED

3.5.16.2 Interface CICM::ChannelEventListener

Interface CICM::ChannelEventListener

```
interface ChannelEventListener {
```

CICM::ChannelEventListener is unlike other CICM interfaces in that the interface is implemented by the developer of the client program to service a specific channel event and is then registered via the CICM::ChannelEventManager.

3.5.16.2.1 CICM::ChannelEventListener Types and Constants

Type CICM::ChannelEventListener::ChannelEvent

```
typedef CICM::UInt32 ChannelEvent;
```

Events for which a ChannelEventListener can be notified.

Constant CICM::ChannelEventListener::C_CHANNEL_DATA_AVAILABLE

```
const CICM::ChannelEventListener::ChannelEvent
C_CHANNEL_DATA_AVAILABLE = 0x00004001;
```

Data from remote peer is available.

Constant CICM::ChannelEventListener::C_CHANNEL_ERROR

```
const CICM::ChannelEventListener::ChannelEvent C_CHANNEL_ERROR =
0x00004002;
```

General error has occurred on the channel.

Constant CICM::ChannelEventListener::C_CHANNEL_INSUFFICIENT_ENTROPY

```
const CICM::ChannelEventListener::ChannelEvent
C_CHANNEL_INSUFFICIENT_ENTROPY = 0x00004004;
```

Insufficient entropy available on the channel.

Constant CICM::ChannelEventListener::C_CHANNEL_LOST_SYNC

```
const CICM::ChannelEventListener::ChannelEvent C_CHANNEL_LOST_SYNC =
0x00004007;
```

Cryptographic synchronization with remote peer has been lost; this may not be detectable by the cryptographic module.

Constant CICM::ChannelEventListener::C_CHANNEL_PEER_RESET

```
const CICM::ChannelEventListener::ChannelEvent C_CHANNEL_PEER_RESET =
0x00004008;
```

Remote peer is no longer available; this may not be detectable by the cryptographic module.

3.5.16.2.2 CICM::ChannelEventListener Methods

Method CICM::ChannelEventListener::event_occurred()

```
        void event_occurred(
                in  CICM::ChannelEventListener::ChannelEvent event,
                in  CICM::Buffer event_data
        );
```

Method implemented by client program that receives a message about a channel event that occurred. An opaque data field with additional information about the event in a module-specific format may optionally be provided with the event itself.

Remarks:

The format of the event data value is not defined by CICM. The Implementation Conformance Statement (see Section 4, Conformance and Extensions) must reference a standard format or define a module-specific format for this datatype.

Note:

Because this method is called by the runtime system and not a client program, it does not return a status value.

Parameters:

`[in]` *event* Event that occurred.
`[in]` *event_data* Opaque data associated with the event.

3.5.17 CHANNEL GROUPS

3.5.17.1 Interface CICM::ControllerGroup

Interface CICM::ControllerGroup

```
interface ControllerGroup {
```

Group of controllers and/or conduits.

Remarks:

All of the controllers in a controller group share certain characteristics, such as a state vector. Whenever a shared characteristic is changed on a member of the group, the change is applied to all members of the group. Thus, if a client program has grouped controllers/conduits to follow advancing TOD rules, for example, when one controller/conduit performs an operation at time X as expressed in the TOD value, all other grouped controllers/conduits are prohibited from using a TOD value earlier than time X.

3.5.17.1.1 CICM::ControllerGroup Methods

Method CICM::ControllerGroup::add()

```
        CICM::Status add(
                in CICM::Controller controller_ref
```

```
          );
```

Add a controller or conduit to this controller group.

Parameters:

`[in]` *controller_ref* Controller to add to the group.

Returns:

S_OK, S_GENERAL_ERROR, S_NON_FUNCTIONAL, S_OPERATION_FAILED, S_POLICY_VIOLATION, S_MODULE_RESOURCES, S_HOST_RESOURCES, S_INVALID_STATE, S_ALARM_STATE, S_MODULE_NOT_AVAILABLE, S_TIMEOUT, S_NOT_AUTHENTICATED, S_NOT_AUTHORIZED, S_TOKEN_NOT_PRESENT, S_TOKEN_ADMIN_NOT_PRESENT, S_CHANNEL_IN_GROUP, S_CHANNEL_CLASSIFICATION

4 CONFORMANCE AND EXTENSIONS

4.1 CONFORMANCE

Many modules will not require the implementation of the full specification to support a module's capabilities. Thus, the CICM conformance model was developed to be flexible. This model does not normatively prescribe the implementation of specific functional subsets of the specification. Instead, CICM outlines a normative Implementation Conformance Statement (ICS) and associated documentation that SHALL be supplied with any conformant implementation.

The ICS guides the developer of a library for a specific module to record the implementation state and presence of extensions for each section of the specification. The gradations of the implementation state are relatively coarse: "implemented," "partially implemented," or "not implemented." Extensions are identified as interface extensions or status code extensions, and are recorded as "existing" or "not-existing." An analysis of the resulting matrix enables a software developer using the API or an architect designing a system integrating with a specific cryptographic module to quickly determine if a developer's library will meet user requirements. Those specification sections marked "partially implemented" or for which extensions are indicated may require additional analysis to determine what elements have been extended or are not implemented, and the resulting repercussions on the system utilizing the library.

CICM interfaces are organized into three major sections: module management, channel management, and key management. Each section is partitioned differently into logical subsections in the ICS. The module management section is partitioned into subsections by individual module managers. The channel management section is partitioned into subsections by channel type. And the key management section is partitioned into subsections by the type of key and class of operation performed on the key.

An Implementation Data Specification (IDS) based on the ICS also is required. For each implemented interface containing an *opaque* data parameter (module-specific or infrastructure-specific parameter not described in detail in the specification), the IDS requires a detailed specification of the data structure for each parameter.

An implementation conforms to the specification if it meets the following conditions:

- A CICM library implementation SHALL include only the subset of interfaces corresponding to the functionality supported by the module for which it was designed. The implementation SHOULD implement the full subset of interfaces implemented by the module. A library SHALL implement a non-zero set of interfaces corresponding to functionality implemented by the module that reasonably maps back to the CICM interface and is appropriate for the system in use.
- A CICM library SHALL minimally implement the CICMRoot and CryptoModule interfaces, both fundamental parts of the specification without which no other interfaces can be implemented. The CryptoModule interfaces SHALL implement minimally one manager, which must be at least "partially" implemented (e.g., simply implementing non-functional inherited or dependent interfaces is non-conformant).
- A CICM library SHALL be made available with a corresponding ICS.
- A CICM library SHALL be made available with an IDS corresponding to its ICS. The format of any module-specific data structures defined as opaque data elements in the specification with which a client program using CICM must have knowledge SHALL be documented by the module

developer and SHALL be made available as the IDS. If the implementation implements no interfaces with opaque data parameters and includes no extensions, the IDS SHALL state that the implementation requires no IDS entries.

- A CICM interface is only conformant if it also implements any inherited and all dependent interfaces (e.g., an Encrypt::WithSignConduit requires that symmetric keys and asymmetric keysets also be implemented). The exception is the CICM::ChannelManager, which only requires the implementation of one or more of its inherited interfaces.
- Any interfaces that are not implemented precisely as specified in the normative portion of the specification SHALL be identified as extensions to the specification.
- Extensions to the CICM specification SHALL NOT contradict nor cause the non-conformance of functionality defined in the normative specification, SHALL follow the requirements and guidelines of the normative specification, and SHALL be clearly described in supporting documentation.
- Memory responsibilities and calling conventions SHALL follow the appropriate IDL language mapping conventions.

4.1.1 IMPLEMENTATION CONFORMANCE STATEMENT CONTENTS

A library implementation conforming to the CICM specification SHALL be accompanied by an ICS. The ICS is generated by the module developer or implementer of a CICM-conformant library for a specific cryptographic module configuration (including any associated hardware/firmware/software) and SHALL contain the following information:

- Details regarding the product and version of the specification to which it conforms, including:
    - CICM version number
    - Product manufacturer/name, version number (hardware, firmware, and software)
    - Configuration details, including patch state
    - Date of claim
- Capability Support Matrix, listing the major sections of the specification and their implementation state ("I"=implemented, "P"=partially implemented, and "N"=not implemented), and the presence of any extensions
- List of developer-defined extensions to specification. Extensions SHALL be divided into four classes: interface extensions, status code extensions, event listener extensions, and constant extensions. Extensions SHALL be documented as specified in the IDS.
- List of unique identifiers for all supported cryptographic algorithms, organized by class of algorithm, and all supported key agreement protocols; each algorithm/protocol unique identifier SHALL be in CICM-specified format (refer to the section "Generating Unique Identifiers").

A sample CICM ICS is presented on the following pages.

CICM V0.6 Implementation Conformance Statement

Product Claiming Conformance

CryptoCorp KGV-XYZ, Version 1.2.3

Date of Claim

1 October 2009

Capability Support Matrix

Section/Subsection	Implementation State	Interface Extensions	Status Code Extensions
Module Management			
Event Manager	I	N	N
Token Manager	P	N	N
Login Manager	I	N	N
User Manager	P	I	I
Test Manager	N	N	N
Log Manager	N	N	N
Package Manager	P	N	N
Key Management			
AsymKeyManager	N	N	N
SymKeyManager	P	N	N
GenericKeyManager	N	N	N
KeyProtocol	N	N	N
Channel Management			
Event Manager	N	N	N
Groups	N	N	N
Encrypt			
Encrypt::Stream	N	N	N
Encrypt::Controller	N	N	N
Encrypt::NegotiatedController	N	N	N
Encrypt::Conduit	N	N	N
Encrypt::NegotiatedConduit	N	N	N
Encrypt::WithMACConduit	N	N	N
Encrypt::WithMACNegotiatedConduit	N	N	N
Encrypt::WithSignConduit	N	N	N
Encrypt::WithSignNegotiatedConduit	N	N	N
Encrypt::KeyWrapConduit	N	N	N
Encrypt with Selective Bypass			
EncryptBypass::Stream	N	N	N
EncryptBypass::NegotiatedController	N	N	N
EncryptBypass::Controller	N	N	N
EncryptBypass::Conduit	N	N	N
EncryptBypass::NegotiatedConduit	N	N	N
Decrypt			
Decrypt::Stream	N	N	N
Decrypt::Controller	N	N	N

Decrypt::NegotiatedController	N	N	N
Decrypt::Conduit	N	N	N
Decrypt::NegotiatedConduit	N	N	N
Decrypt::WithMACConduit	N	N	N
Decrypt::WithMACNegotiatedConduit	N	N	N
Decrypt::WithVerifyConduit	N	N	N
Decrypt::WithVerifyNegotiatedConduit	N	N	N
Decrypt::KeyUnwrapConduit	N	N	N
Decrypt with Selective Bypass			
DecryptBypass::Stream	N	N	N
DecryptBypass::Controller	N	N	N
DecryptBypass::NegotiatedController	N	N	N
DecryptBypass::Conduit	N	N	N
DecryptBypass::NegotiatedConduit	N	N	N
Duplex			
Duplex::Stream	N	N	N
Duplex::Controller	N	N	N
Duplex::NegotiatedController	N	N	N
Duplex::Conduit	N	N	N
Duplex::NegotiatedConduit	N	N	N
Full Bypass (Write)			
BypassWrite::Stream	N	N	N
BypassWrite::Controller	N	N	N
BypassWrite::Conduit	N	N	N
Full Bypass (Read)			
BypassRead::Stream	N	N	N
BypassRead::Controller	N	N	N
BypassRead::Conduit	N	N	N
Emit			
Emit::RandomController	N	N	N
Emit::RandomConduit	N	N	N
Emit::PseudoRandomController	N	N	N
Emit::PseudoRandomConduit	N	N	N
Emit::KeyStreamGenController	N	N	N
Emit::KeyStreamGenConduit	N	N	N
Integrity			
Answer::HashConduit	N	N	N
Answer::MACConduit	N	N	N
Answer::MACVerifyConduit	N	N	N
Answer::SignConduit	N	N	N
Answer::SignHashConduit	N	N	N
Answer::VerifyConduit	N	N	N
Answer::VerifyHashConduit	N	N	N
Single Domain			
Coprocessor::EncryptConduit	N	N	N
Coprocessor::EncryptWithMACConduit	N	N	N
Coprocessor::EncryptWithSignConduit	N	N	N
Coprocessor::DecryptConduit	N	N	N

Coprocessor::DecryptWithMACConduit	N	N	N
Coprocessor::DecryptWithVerifyConduit	N	N	N

Interface Extensions

CICM::UserManager::enable()
CICM::UserManager::disable()

Status Code Extensions

CICM::S_USER_ALREADY_ENABLED
CICM::S_USER_ALREADY_DISABLED

Module/Channel Event Listener Extensions

None

Constant Extensions

None

Supported Algorithms

AES128-CBC
3DES-OFB

4.1.2 IMPLEMENTATION DATA SPECIFICATION CONTENTS

The IDS serves as the detailed supporting documentation for the ICS. Conformance with the CICM specification requires that:

- Each implemented interface that accepts an opaque data object SHALL reference an existing standard or document the data structure associated with that object in sufficient detail to allow an implementer to create new objects and manipulate existing objects. The exception to this requirement is those cases where a client program will NOT be allowed to manipulate the opaque data object (e.g., CICM::KeyProtocolReceiver::get_from_module or CICM::PackageImporter::import_segment).
- Each interface extension listed in the ICS SHALL be clearly described in the IDS and SHALL be documented in a manner similar to the normative CICM documentation.
- Each status code extension listed in the ICS SHALL be referenced in the IDS with a corresponding description, numeric code, and a list of CICM interfaces to which the extension applies.
- Each module or channel event listener extension listed in the ICS SHALL be referenced in the IDS with corresponding description, numeric code, and data structure definition associated with the event_data parameter, if applicable.
- Each extended constant value listed in the ICS shall be referenced in the IDS with corresponding description and numeric code.

Examples of interfaces requiring an IDS entry to be conformant include:

- CICM::SymKeyManager::get_key_by_id, where the key identifier is specific to the key management system in use.

- CICM::LogManager::retrieve, where the log returned from the method call will vary from module-to-module.
- CICM::ModuleEventListener::event_occurred, where the event_data parameter passed to a client program as part of an event notification is system specific.

Note that the event listener callbacks (CICM::ModuleEventListener::event_occurred and CICM::ChannelEventListener::event_occurred) require that the event_data parameter be described for each event type implemented.

4.1.3 GENERATING UNIQUE IDENTIFIERS

CICM does not provide a list of algorithms with their corresponding normative unique identifiers. Instead, normative guidance is provided for generating the identifiers for the different classes of algorithms defined in the specification and for key agreement protocols. These identifiers are used by software developers when specifying algorithms or protocols as parameters to CICM methods. This identifier generation guidance is intended to promote interoperability, and encourage the use of the same identifier for algorithms among vendors.

Three major components may be combined to form a unique algorithm identifier: an algorithm (ALGO), that may be precisely specified as an encryption algorithm (ENCRALGO), signature algorithm (SIGALGO), MAC algorithm (MACALGO), or hash algorithm (HASHALGO); a mode (MODE); and an encoding scheme (SCHEME), that may be precisely specified as an encryption scheme (ENCRSCHEME) or a signature scheme (SIGSCHEME). Note that some components above may not apply to certain algorithms. In addition, applicable modes and components need not always be specified. For encryption and signature algorithms, if a length is required, the length SHALL be appended to the algorithm without a dash ("-") delimiter. Otherwise, components are concatenated with a dash ("-").

Alternatively, an identifier can consist of a simple personality designation (PERSONALITY). The personality consists of a combination of parameters that comprise a logically complete crypto, and specifies a specific equipment type or configuration for which algorithm, mode, and any other parameters are implicit. The designation may contain dashes.

Certain algorithms may be appropriate for and thus listed under more than one algorithm class. Below are the classes of algorithms and format of the identifiers for each class:
Asymmetric encryption algorithm identifiers (AsymEncrAlgorithmId)
  Format: ENCRALGO ["-" ENCRSCHEME] | PERSONALITY
  Examples: "RSA1024-OAEP"
Asymmetric signature algorithm identifiers (AsymSigAlgorithmId)
  Format: SIGALGO ["-" HASHALGO ["-" SIGSCHEME]] | PERSONALITY
  Examples: "DSA-SHA1" or "RSA1024-SHA256-PKCS1V1_5"
Symmetric encryption algorithm identifiers (SymEncrAlgorithmId)
  Format: ENCRALGO | PERSONALITY
  Examples: "AES128" or "3DES"
Symmetric MAC algorithm identifiers (SymMacAlgorithmId)
  Format: MACALGO [- HASHALGO] | PERSONALITY
  Examples: "HMAC-SHA1" or "UMAC"
Hash algorithm identifiers (HashAlgorithmId)
  Format: HASHALGO | PERSONALITY

  Examples: "MD5" or "SHA1"
Key wrap algorithm identifiers (KeyWrapAlgorithmId)
  Format: ENCRALGO | PERSONALITY
  Examples: AESKW

Two major components may be combined to form a key agreement protocol identifier: the key agreement protocol including its version number (KEYAGREEPROTO) and the protocol's associated algorithm suite including its version number (ALGOSUITE). The following is the format for key agreement protocol identifiers.
Key agreement protocol identifier (ProtocolId)
  Format: KEYAGREEPROTO "-" ALGOSUITE
  Examples: "IKE2.0-FIREFLY"

Note that the resulting identifiers may not be compatible with those identifiers defined for other module developers' implementations. A client program utilizing an identifier corresponding to one algorithm for a specific module may be required to modify the identifier for the same algorithm for a different type of module. Discrepancies may be discovered through a brief review of the ICS "Supported Algorithms" section.

4.1.4 CONFORMANCE VERIFICATION

In the future, test assertions may be made available to allow results from different organizations to be compared, and to provide proof of conformance to the specification.

4.2 EXTENSIONS

An extension is a mechanism to define functionality beyond what is defined in the official specification. In the interest of promoting interoperability, extensions to the specification are discouraged except where necessary. Extensions to the specification enable module developers to add functionality unanticipated by the specification developers and to support proprietary features.

4.2.1 EXTENDING AN INTERFACE

Developers may augment CICM interfaces by extending CICM IDL by adding new methods/attributes to existing interfaces or by deriving off existing CICM interfaces. Extensions SHALL be documented in the ICS.

4.2.2 EXTENDING CODES

CICM codes are constants that share a single 32-bit space. A number of datatypes for different purposes correspond to ranges in this space. The "CICM" codes are normatively defined in the specification; the "extended" codes are module developer-defined extensions. The codes, with their corresponding ranges and uses, are as follows:

`0x00000000` - `0x00001000` CICM status codes
`0x00001001` - `0x00002000` Extended status codes
`0x00002001` - `0x00003000` CICM module event codes
`0x00003001` - `0x00004000` Extended module event codes
`0x00004001` - `0x00005000` CICM channel event codes
`0x00005001` - `0x00006000` Extended channel event codes

`0x00006001` - `0x00007000` CICM generic constants
`0x00007001` - `0x00008000` Extended generic constants
`0x00008001` - `0x7FFFFFFF` RESERVED

Normatively-defined CICM codes SHOULD be used whenever possible. If any of the extended codes above are defined, they SHALL be documented as specified below.

4.2.2.1 Extending Status Codes

The return value from CICM methods informs the caller of the status of the call. CICM does not utilize the IDL exception mechanism to report errors.

The specification normatively defines a set of error codes in the range of `0x00000000` - `0x00001000`, which may not be modified or extended. A block of codes in the range of `0x00001001` - `0x00002000` are reserved for module developer-defined status codes. Any codes defined in this range SHALL be documented in the ICS.

4.2.2.2 Extending Module/Channel Event Codes

The specification supports registering and unregistering user-defined channel event listeners for specific module and channel events. Module events in the range of `0x00003001` - `0x00004000` and channel events in the range of `0x00004001` - `0x00005000` are normatively defined and may not be modified or extended. A block of module events in the range `0x00003001` - `0x00004000` and channel events in the range of `0x00005001` - `0x00006000` are reserved for module developer-defined events. Any codes defined in this range SHALL be documented in the ICS.

4.2.2.3 Extending Constants

A number of constants are normatively defined for specification use in the range of `0x00006001` - `0x00007000`. Module developer-defined constants may be specified in the range of `0x00007001` - `0x00008000`. Any constants defined in this range SHALL be documented in the ICS.

APPENDIX A: STATUS CODES

Each method defined in CICM returns a status value to inform the caller as to the outcome of the call. The documentation for each individual method lists the status codes that may be returned in the event a call to the method results in failure.

The status value CICM::S_OK is returned if a method completes successfully. The output parameters of any methods that return a status other than CICM::S_OK are invalid and SHALL not be referenced or used.

CICM methods can fail for a variety of reasons, including:

- Invalid, illegal, out-of-range, or poorly formed parameters
- Resources insufficient or unavailable
- Unsupported capabilities
- Policy violation
- Hardware failure.

For additional information concerning extending status codes, see Section 4, Conformance and Extensions.

CICM status codes are defined below.

S_OK = 0x00000000
No error.

S_GENERAL_ERROR = 0x00000003
Unrecoverable error occurred, potentially leaving module in an inconsistent state.

S_NON_FUNCTIONAL = 0x00000005
Tamper event or other condition has rendered module non-functional.

S_OPERATION_FAILED = 0x00000006
Method encountered a general failure, but detailed information about the failure is not available.

S_POLICY_VIOLATION = 0x00000009
Module policy does not permit the requested action.

S_MODULE_RESOURCES = 0x0000000A
Module resources necessary to perform the requested operation are not available.

S_HOST_RESOURCES = 0x0000000C
Host resources necessary to perform the requested operation are not available.

S_INVALID_STATE = 0x0000000F
Module is in a state that does not allow this operation to be performed.

S_ALARM_STATE = 0x00000011
Module has entered an alarm state.

S_MODULE_NOT_AVAILABLE = 0x00000012
Module has been powered down, disconnected, or is otherwise unavailable..

S_TIMEOUT = 0x00000014
Time to receive response from call exceeded threshold.

S_NOT_AUTHENTICATED = 0x00000017
User has not authenticated to module.

S_NOT_AUTHORIZED = 0x00000018
User is not authorized to call method.

S_MODULE_DOES_NOT_EXIST = 0x0000001B
No module with the specified unique identifier exists.

S_MODULE_IN_USE = 0x0000001D
Module test initiated when channels or other module resources are in use.

S_NOT_AVAILABLE = 0x0000001E
Information is not available or cannot be found.

S_INVALID_VECTOR = 0x00000021
Invalid vector provided; this may be because the length or format of the vector is inappropriate for the algorithm or system with which the vector is being used.

S_INVALID_DATA_BUFFER = 0x00000022
Data in user-specified buffer parameter is invalid.

S_KEY_USED_INVALID = 0x00000024
Key specified as parameter to method is invalid; this could denote that the key has been zeroized, a failed parity check, or other conditions that prevent the use of the key.

S_KEY_USED_EXPIRED = 0x00000027
Key specified as parameter to method has expired and may not be used.

S_KEY_USED_CLASSIFICATION = 0x00000028
Key specified as parameter to method at wrong classification level.

S_KEY_USED_WRAPPED = 0x0000002B
Key specified as parameter to method may not be used in the context until it has been unwrapped.

S_KEY_USED_CONTEXT = 0x0000002D
Attempt to use key in an illegal context as defined by the module; e.g., a key is specified for use on a channel but, due to module architecture, the key is unavailable to that channel.

S_KEY_USED_COMPONENT_NOT_AVAIL = 0x0000002E
Asymmetric key specified as parameter to method contains only a public key (possibly in a certificate) or only a private key, when the other component is needed by the called method.

S_KEY_INVALID = 0x00000030
Key is invalid; this could denote that the key has been zeroized, a failed parity check, or other conditions that prevent the use of the key.

S_KEY_EXPIRED = 0x00000033
Key has expired and may not be used.

S_KEY_INCOMPATIBLE = 0x00000035
Key type (e.g., TEK, KEK) incompatible with intended usage.

S_KEY_CLASSIFICATION = 0x00000036
Key at wrong classification level.

S_KEY_WRAPPED = 0x00000039
Key may not be used in this context until it has been unwrapped.

S_KEY_NOT_WRAPPED = 0x0000003A
Key is not wrapped.

S_KEY_NOT_WRAPPABLE = 0x0000003C
Module is not able to wrap key.

S_KEY_NOT_EXPORTABLE = 0x0000003F
Key is not exportable, potentially because it has not been wrapped or other policy disallows it.

S_KEY_WRAPPED_EXISTS = 0x00000041

Wrapped key already exists.

S_KEY_UNWRAPPED_EXISTS = 0x00000042

Unwrapped key already exists.

S_KEY_UPDATE_MAX = 0x00000044

Maximum number of updates for this key has been exceeded.

S_KEY_INVALID_ID = 0x00000047

Invalid key identifier specified.

S_KEY_PHYSICAL_LOC = 0x00000048

Invalid key physical location specified.

S_KEY_ILLEGAL_CONVERSION = 0x0000004B

Target algorithm is incompatible with algorithm associated with specified key.

S_KEY_MALFORMED = 0x0000004D

Key material supplied is malformed.

S_KEY_METADATA_MALFORMED = 0x0000004E

Key metadata supplied is malformed.

S_KEY_NO_NEXT = 0x00000050

No next key available for rollover.

S_KEY_WRONG_TYPE = 0x00000053

Illegal attempt to process a symmetric key with an asymmetric method or an asymmetric key with a symmetric method.

S_KEY_FILL_DEVICE_NOT_CONNECTED = 0x00000055

Key fill device not connected.

S_KEY_FILL_NOT_INITIATED = 0x00000056

Manual key fill device interaction not initiated within system-defined time limit.

S_KEY_TRUST_ANCHOR = 0x00000059

Trust anchor required but is unavailable.

S_LOCAL_PORT_INVALID = 0x0000005A

Local port specified is invalid.

S_LOCAL_PORT_INCOMPATIBLE = 0x0000005C

Local port specified cannot be used in intended manner.

S_LOCAL_PORT_IN_USE = 0x0000005F

Local port specified is currently in use.

S_REMOTE_PORT_INVALID = 0x00000060

Remote port specified is invalid.

S_REMOTE_PORT_IN_USE = 0x00000063

Remote port specified is currently in use.

S_ALGO_INVALID = 0x00000065

Malformed string or unsupported/invalid algorithm specified.

S_ALGO_INCOMPATIBLE = 0x00000066

Algorithm incompatible with intended usage (e.g., encryption, signature, hashing).

S_TOKEN_NOT_PRESENT = 0x00000069

Token must be inserted to perform the requested operation and no token is available to the module.

S_TOKEN_ADMIN_NOT_PRESENT = 0x0000006A

Administrator token must be inserted to perform the requested operation and either no token is present or the inserted token is not an administrator token.

S_TOKEN_ACCESS = 0x0000006C
Token I/O error.
S_TOKEN_RESOURCES = 0x0000006F
Token resources necessary to perform the requested operation are not available.
S_TOKEN_ASSOC_EXISTS = 0x00000071
Association between module and token already exists.
S_TOKEN_ASSOC_AT_MODULE = 0x00000072
Association failed because module will allow no new associations.
S_TOKEN_ASSOC_AT_TOKEN = 0x00000074
Association failed because token will allow no new associations.
S_TOKEN_ASSOC_NOT_EXIST = 0x00000077
Association between module and token does not exist at the module, at the token, or both.
S_TOKEN_ASSOC_GENERAL = 0x00000078
Unspecified token association error occurred.
S_TOKEN_DISASSOC_GENERAL = 0x0000007B
Unspecified token disassociation error occurred.
S_TOKEN_REC_NOT_FOUND = 0x0000007D
Specified record not found.
S_TOKEN_TIMEOUT = 0x0000007E
Timeout for insertion of token has been exceeded.
S_TOKEN_LAST_ASSOCIATED = 0x00000081
Cannot disassociate the last associated token from this module.
S_PACKAGE_NOT_ACTIVATABLE = 0x00000082
Specified package is not executable.
S_PACKAGE_ACTIVATED = 0x00000084
Specified package is currently running.
S_PACKAGE_NOT_ACTIVE = 0x00000087
Specified package is not currently running.
S_PACKAGE_INVALID = 0x00000088
Specified package is invalid.
S_PACKAGE_TYPE_INVALID = 0x0000008B
Specified package type is invalid.
S_PACKAGE_KEY_NOT_AVAILABLE = 0x0000008D
Package is encrypted and the key specified for use to decrypt package is not available on the module.
S_PACKAGE_KEY_NOT_SPECIFIED = 0x0000008E
Package is encrypted but no key is specified to decrypt it.
S_LOG_ENTRY_INVALID = 0x00000090
Log entry is invalid.
S_EVENT_REGISTERED = 0x00000093
An event has already been registered by this process for this event type.
S_EVENT_NOT_REGISTERED = 0x00000095
An event has not been registered by this process for this event type.
S_EVENT_NOT_SUPPORTED = 0x00000096
Event is not supported in this implementation.
S_TRUSTED_DISPLAY = 0x00000099
Peer information is available at trusted display.

S_NEGOTIATION_ABORTED = 0x0000009A
Negotiation was aborted.
S_NEGOTIATION_FAILURE = 0x0000009C
Negotiation failed.
S_NEGOTIATION_IN_PROGRESS = 0x0000009F
Negotiation is already in progress.
S_NEGOTIATION_NOT_IN_PROGRESS = 0x000000A0
No negotiation has been initiated.
S_NEGOTIATION_TIMEOUT = 0x000000A3
Negotiation timed out.
S_CERT_LOCAL_INVALID = 0x000000A5
Local certificate used in a key negotiation is invalid; the certificate may be corrupted or does not verify.
S_CERT_LOCAL_EXPIRED = 0x000000A6
Local certificate used in a key negotiation has expired.
S_CERT_REMOTE_INVALID = 0x000000A9
Remote certificate used in a key negotiation is invalid; the certificate may be corrupted or does not verify.
S_CERT_REMOTE_EXPIRED = 0x000000AA
Remote certificate used in a key negotiation has expired.
S_CERT_REMOTE_PATH = 0x000000AC
Certificates to enable verification of remote certificate's certification path are not available.
S_PROTO_INVALID = 0x000000AF
Malformed string or unsupported/invalid protocol specified.
S_PROTO_INCOMPATIBLE = 0x000000B1
Protocol specified is incompatible with intended usage.
S_PROTO_UNDETERMINED = 0x000000B2
An "implicit" protocol has been specified, but the protocol message does not indicate the protocol.
S_CHANNEL_ERROR = 0x000000B4
Generic conduit/controller error encountered.
S_CHANNEL_PEER_RESET = 0x000000B7
Peer crypto reset conduit/controller or conduit/controller ceased operation.
S_CHANNEL_MAX = 0x000000B8
Limit on total number of conduits/controllers has been reached.
S_CHANNEL_NOT_FOUND = 0x000000BB
Conduit/controller not found.
S_CHANNEL_IO_ERROR = 0x000000BD
Conduit/controller I/O error.
S_CHANNEL_DATA_INVALID = 0x000000BE
Input data to cryptographic operation is invalid (e.g., plaintext for encryption or ciphertext for decryption).
S_CHANNEL_DATA_INVALID_LEN = 0x000000C0
Plaintext (for encryption) or ciphertext (for decryption) input data to cryptographic operation has an inappropriate length; this could denote that the data is too short, too long, or is not a multiple of some particular block size.
S_CHANNEL_BUFFER_LEN = 0x000000C3
Output of function is too large for supplied buffer.

S_CHANNEL_IN_GROUP = 0x000000C5
Conduit/controller already exists as part of group.

S_CHANNEL_CLASSIFICATION = 0x000000C6
Conduits/controllers are not of the same classification.

S_BYPASS_DATARATE_EXCEEDED = 0x000000C9
Bypass data rate exceeded.

S_BYPASS_DATALIMIT_EXCEEDED = 0x000000CA
Bypass data limit exceeded.

S_INTEGRITY = 0x000000CC
In those cases where an encryption algorithm supplies both confidentiality and integrity (an integrity value is transmitted with the ciphertext), the final decrypt may fail with this integrity error if the integrity check fails.

S_AUTHENTICATION_FAILED = 0x000000CF
Authentication to the module failed; this could denote that a password is incorrect or that additional authentication data supplied is invalid.

S_USER_AUTHENTICATED = 0x000000D1
Specified user has already authenticated to module.

S_USERNAME_INVALID = 0x000000D2
Username is invalid.

S_USER_EXISTS = 0x000000D4
User already exists.

S_USER_INVALID = 0x000000D7
User does not exist.

S_ROLE_INVALID = 0x000000D8
Role does not exist.

S_ROLE_ASSOCIATED = 0x000000DB
User already associated with this role.

S_ROLE_NOT_ASSOCIATED = 0x000000DD
User not associated with this role.

S_ROLE_MAX = 0x000000DE
Maximum number of roles already associated with this user.

S_PASSWORD_INVALID = 0x000000E1
Specified password does not meet module policy.

S_PASSWORD_INVALID_CHAR = 0x000000E2
Specified password has invalid characters in it.

S_PASSWORD_INVALID_LEN = 0x000000E4
Length of specified password is either too long or too short.

S_SALT_INVALID = 0x000000E7
Invalid salt specified.

S_ITERATION_COUNT_INVALID = 0x000000E8
Invalid iteration count specified.

S_INSUFFICIENT_ENTROPY = 0x000000EB
Insufficient entropy available.

APPENDIX B: TERMS

alarm

Output signal that denotes that the module has entered an alarm state. An alarm condition may prohibit a module from performing cryptographic operations.

asymmetric key

Pair of related keys, a public key known to everyone and a private key known only to the owning entity. See *symmetric key* and *asymmetric keyset*.

asymmetric keyset

May comprise one more of the following components: an asymmetric key pair, the public and private key components of a keypair, the digital certificate corresponding to the keyset public key, one or more verification certificates in the certificate chain of trust, and related public domain parameters. See also *asymmetric key*.

asynchronous notification

Delivery of an indication of a condition or event where, from the point of view of the recipient (the client program), the delivery occurs asynchronously via a *callback*. See also *event* and *event notification*.

attribute

State associated with an instance of an interface.

authentication

Security measure designed to establish the validity of a transmission, message, or originator, or a means of verifying an individual's authorization to receive specific categories of information (CNSSI 4009).

authorization

Access privileges granted to a user, program, or process (CNSSI 4009).

blocking

A call to a method is *blocking* if the method does not return program control to the caller until either the operation has completed or an error is recognized. See also *non-blocking*.

buffer

Collection of binary data.

bypass

In cryptography, this is an operation whereby all of the data is passed from one security domain through the cryptographic module to the other security domain without having a cryptographic transformation applied to it. See also *selective bypass*.

callback

Procedure provided by the *client program* that is to be invoked when an appropriate condition or event is recognized. See also *asynchronous notification*.

certificate

Digitally signed document that binds a public key with an identity. The certificate contains, at a minimum, the identity of the issuing Certification Authority, the user identification information, and the user's public key (CNSSI 4009).

certificate revocation list (CRL)

List of certificate serial numbers corresponding to certificates that have been revoked or are no longer valid.

channel
Abstraction under which one or more cryptographic transforms are performed and within which all details associated with the transform are encapsulated, including the path through the module. See also conduit, controller, and stream.

channel type
Cryptographic transform to be applied on a channel.

client program
Program linked to a CICM library running as a single process on a *host* computer system that accesses cryptographic services and/or to manages a cryptographic module.

conduit
Abstraction that encapsulates *channel* control and data flow. See also *channel*, *controller*, and *stream*.

controller
Abstraction used to configure and control a channel. See also *channel*, *conduit*, and *stream*.

coprocessor mode
Mode of operation in which cryptographic transformations are performed within a single security domain. For example, in coprocessor mode, a client program provides plaintext to a module, the plaintext is transformed, and the resulting ciphertext is returned to the same client program.

cryptographic module
Abstraction of hardware, firmware, or software components that makes cryptographic services available to client programs via one or more *channels*.

cryptographic synchronization
Process by which a receiving decrypting cryptographic logic attains the same internal state as the transmitting encrypting logic.

cryptographic transform
The specific cryptographic process that is to be applied to a stream of data or is used to generate keystream or random data, often (but not always) based upon a cryptographic key. Transforms include encryption, decryption, signing, keystream generation, hashing, and random data generation.

driver
Conceptual component residing on a host that enables the exchange of commands and data between the client program and a module. Module-specific abstraction layer that serves as a translation mechanism between the individual functions defined in the CICM library and the commands specific to a given cryptographic module. This component also provides a conduit for data between a host and a module.

ephemeral symmetric key
Symmetric cryptographic key generated as part of a key negotiation process. Ephemeral keys may be destroyed when the channel or session utilizing the ephemeral key completes. Ephemeral keys are not visible if a client program lists the keys on a module. See also *static key*.

event
Situation occurring on a module or a channel for which a client program may be notified.

event notification
Call from the host runtime system to a client program announcing that a specific situation has occurred. See *callback* and *asynchronous notification*.

grade
Negotiated classification level of a channel.

hardware access token

Removable device used to provide locking and unlocking features for the cryptographic capabilities of a cryptographic module.

host

Computer system upon which a *client program* linked to a CICM library executes.

hybrid channel

Channel that simultaneously supports two fundamental cryptographic services; for example, an encryption with signature channel transforms data, resulting in both ciphertext and a final signature value.

iterator

Software construct that enables a software program to walk through a list of related items.

key

See *symmetric key* and *asymmetric key*.

key agreement protocol

Protocol that allows two or more participants to negotiate an ephemeral symmetric key without disclosing the resulting key material to non-participants. The protocol is conducted in such a way that all participants influence the outcome.

key encryption key

Key that encrypts or decrypts another key for transmission or storage.

key fill device

Devices that read-in, transfer, and store key material.

key fill interface

Set of protocols, electrical connections, and physical characteristics that comprise the connecting link between a *key fill device* and a cryptographic module. CICM enables a key fill interface to be configured and actions to be initiated on a *key fill device* via the key fill interface.

key infrastructure

Set of hardware, software, people, policies, and procedures needed to create, manage, store, distribute, and revoke key material.

key rollover

Process of moving from one key to another in a pre-defined sequence of keys; may also be referred to as "key supersession."

key tag

Identification information associated with certain types of electronic key (CNSSI 4009).

key unwrap

Process whereby an encrypted cryptographic key is decrypted using a cryptographic module and a different key.

key update

Deterministic one-way transformation of a symmetric key (and its current update count) to a new key.

key wrap

Process whereby a cryptographic key is encrypted by a cryptographic module using a separate key in a manner sufficient to protect the key at the level of its classification.

keystream

Sequence of symbols produced by a cryptographic module using a cryptographic key to combine with plain text to produce cipher text, control transmission security processes, or produce key (CNSSI 4009).

listener
Method registered by the client program that will be called upon the occurrence of a specific module event.

local port
Port on module in same security domain in which client program is located to which commands are presented and through which data is sent/received. See also *remote port*.

manager
Specialized attributes that encapsulate different classes of module, key and channel management functionality.

message authentication code (MAC)
Data associated with an authenticated message allowing a receiver to verify the integrity of the message (CNSSI 4009).

method
Single CICM library function that performs a specific task.

namespace
An abstract container that holds related interfaces.

non-blocking
A method is *non-blocking* if it initiates an operation and then returns control to the caller, usually before the outcome of the operation has been determined. See also *blocking*.

opaque data object
Binary object accepted by or returned from a method call whose structure is imposed by some entity unrelated to the CICM specification.

package
Software, FPGA image, policy database, configuration parameters, or other types of executable or interpretable code that may be imported into and removed from a module.

persistent key
See *static key*.

policy
Precise specification of the security rules under which a cryptographic module will operate.

port
Identifier that designates a logical interface through which data moves into and out of a cryptographic module. See also *local port* and *remote port*.

remote port
Port in non-local security domain from which transformed data is received. See also *local port*.

role
A designation to which users are assigned that identifies a job type defined in terms of the privileges of that user.

security domain
System or group of systems operating under a common security policy. Communication between domains is controlled in a well-defined manner.

selective bypass
Portion of the traffic through a channel that is not to be cryptographically transformed. Also commonly referred to as "header bypass."

static key
Cryptographic key imported into or established on a module that will remain on the module until it is explicitly removed. See also *ephemeral key*.

stream

An abstraction representing an entity utilizing an existing controller to enable data to be sent to a module to be transformed and transformed data to be received using a controller as a foundation.

symmetric key

Usually a sequence of random or pseudorandom bits used initially to set up and periodically change the operations performed in crypto-equipment for the purpose of encrypting or decrypting electronic signals (CNSSI 4009). See *asymmetric key*.

system

Hardware and software components, including the cryptographic module, that meet a specific set of security-related requirements.

tamper

Output signal from module that denotes it has detected a tamper event.

token

See *hardware access token*.

trusted display

Hardware component independent of a host to enter or display information to be directly sent to/received from a cryptographic module.

zeroize

Input signal instructing the module to clear its memory of any sensitive cryptographic material. CICM supports both a module zeroize (destroying all key material on module) and zeroizing an individual key.