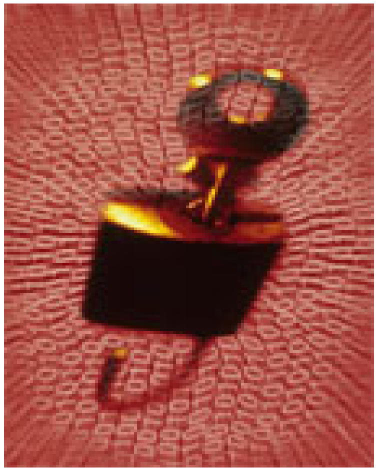
International Tactical Radio Security Services (IRSS) API

Technical Overview



2 Dec 2011

Reminder on Restricted and Controlled Information Policy

- Participants in this meeting are reminded that the Wireless Innovation Forum is an international organization, and they are prohibited from disclosing export restricted or controlled information during the course of this meeting.
- In addition, participants are reminded that they are prohibited from making input contributions containing export restricted or controlled information to the Wireless Innovation Forum. Members wishing additional information on these prohibitions are referred to the Wireless Innovation/SDR Forum’s Policy on Restricted and Controlled Information (SDRF Policy 009) available on the web.

Meeting Agenda

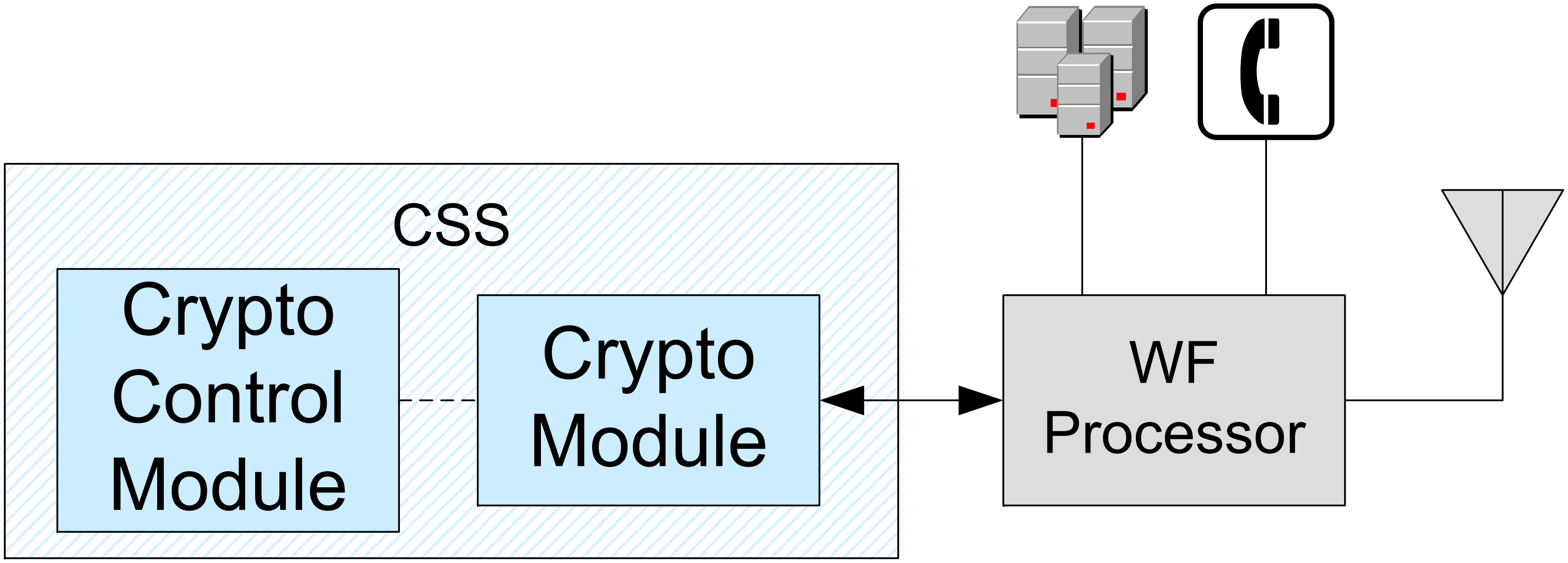
- **Radio Topologies**
- **Control Module**
  - Channel Management Interface
  - Certificate Management Interface
  - Key Management Interface
- **Infosec Module**
  - Cryptographic Channel Interfaces
  - TRANSEC Channel Interface
- **Bypass Module**
  - Bypass Channel Interfaces
- **IandA Module**
  - IandA Channel Interfaces
  - Random Interface
- **Protocol Module**
  - Protocol Channel Interfaces

Radio Topologies

Radio Topologies

Simple Commercial Topology

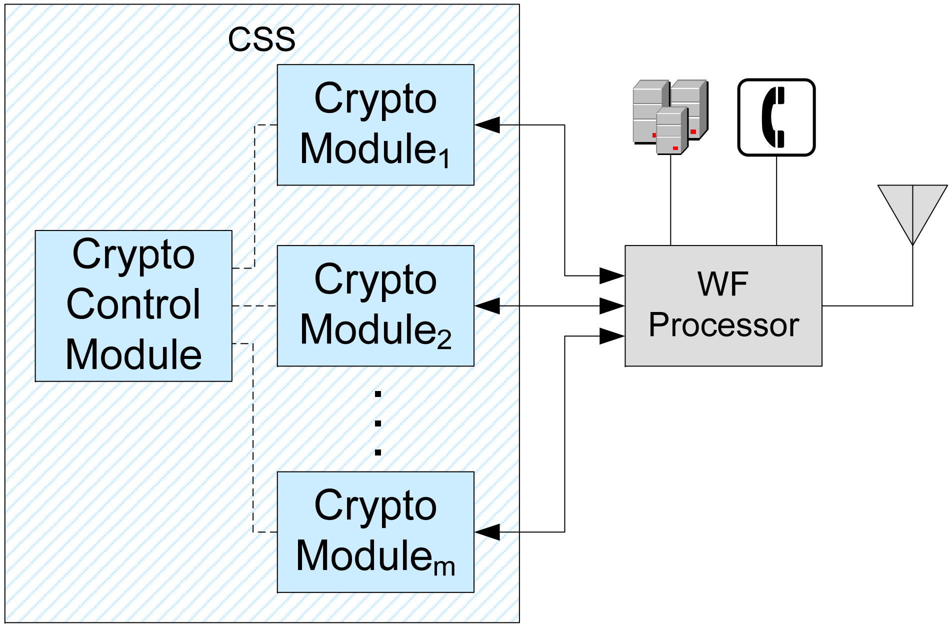
- Single WF processor
- Single Security Domain
- Single Crypto Module
- Single Crypto Control Module (separate or not from crypto module)



Radio Topologies

Constrained Crypto Topology (uncommon)

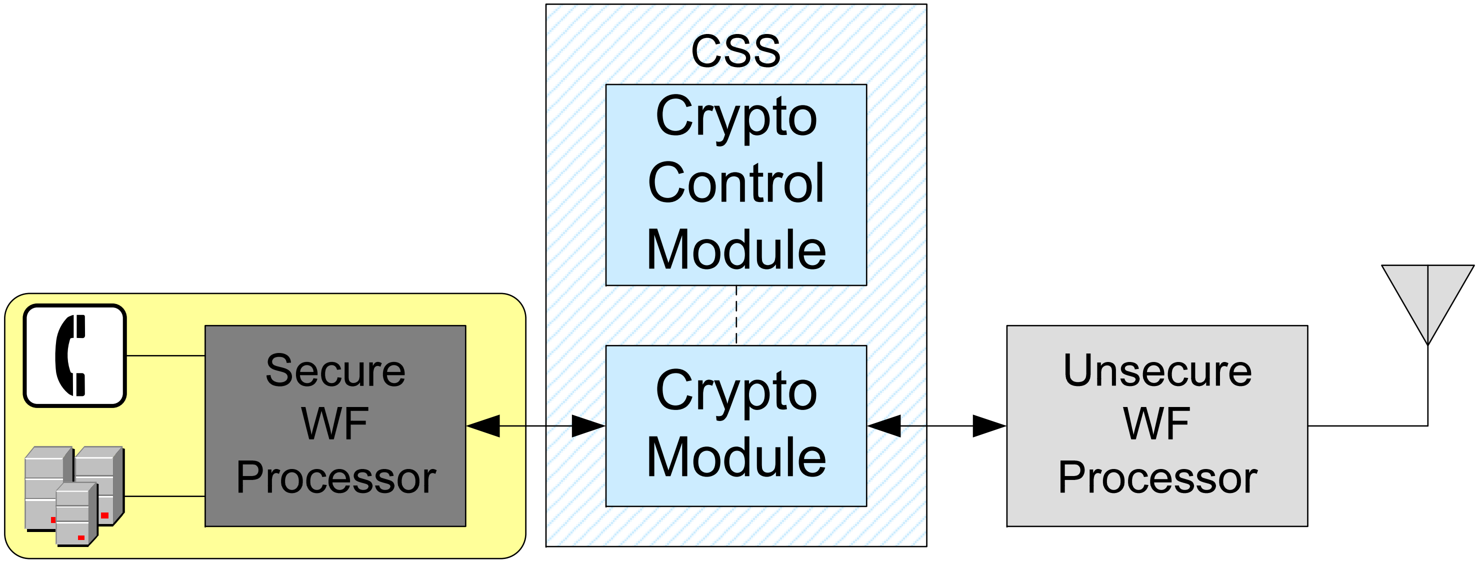
- Single WF processor, possibly multiple waveforms / channels
- Single Security Domain
- Multiple Crypto Modules
- Single Crypto Control Module



Radio Topologies

Standard Military Topology

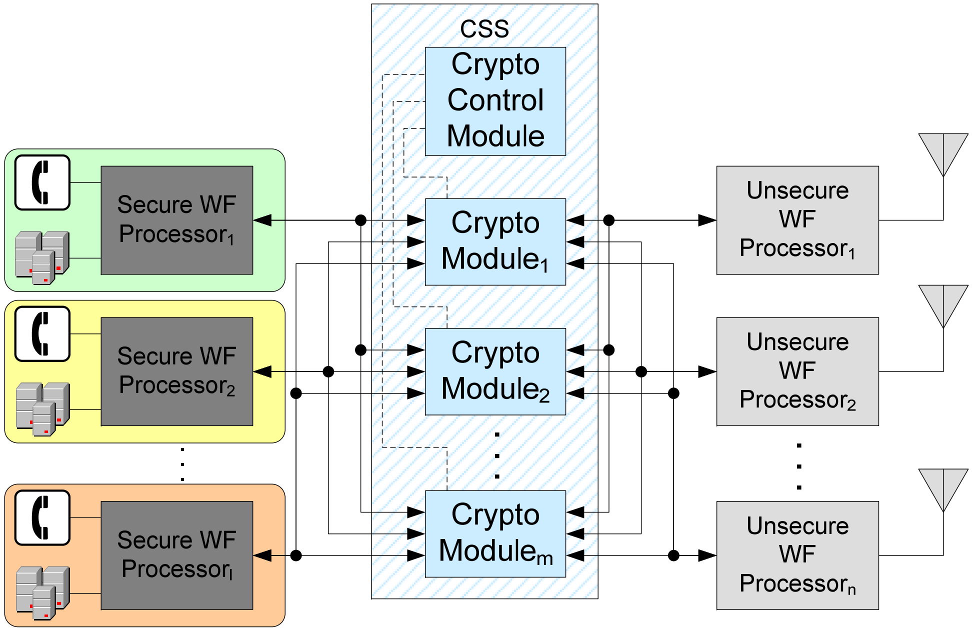
- Separate Secure and Unsecure WF Processors
- Two Security Domains – plaintext, ciphertext
- Single Crypto Module
- Single Crypto Control Module (separate or not from crypto module)



Radio Topologies

Multichannel Military Topology

- Separate Secure and Unsecure WF Processors
- Multiple Security Domains on plaintext side
- Multiple Crypto Modules
- Single Crypto Control Module (shared key management, etc)
- Some form of routing topologies between WF processors and Crypto modules

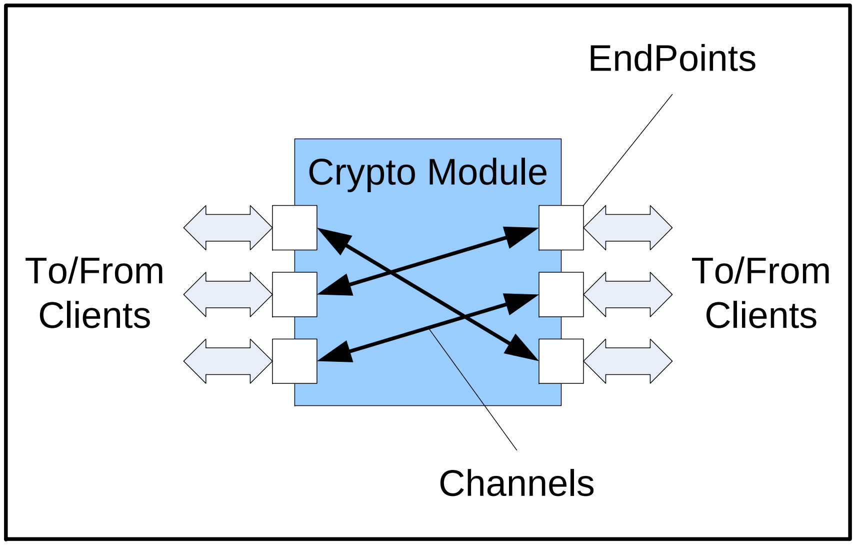


Control Module:

Channel Management

Channel Management

- **What is a channel?**
  - A communication path to/from the security subsystem defined by:
    - the crypto module (CM) providing the service,
    - the access points (called endpoints) used to interface with the CM,
    - service specific configuration information
- **Types of Channels Created and Managed by the API:**
  - Cryptographic Channels
  - TRANSEC Channels
  - Bypass Channels
  - Integrity and Authentication Channels
    - Hash Channels
    - MAC Channels
    - Signature Channels
    - Signature Verification Channels
  - Protocol Channels

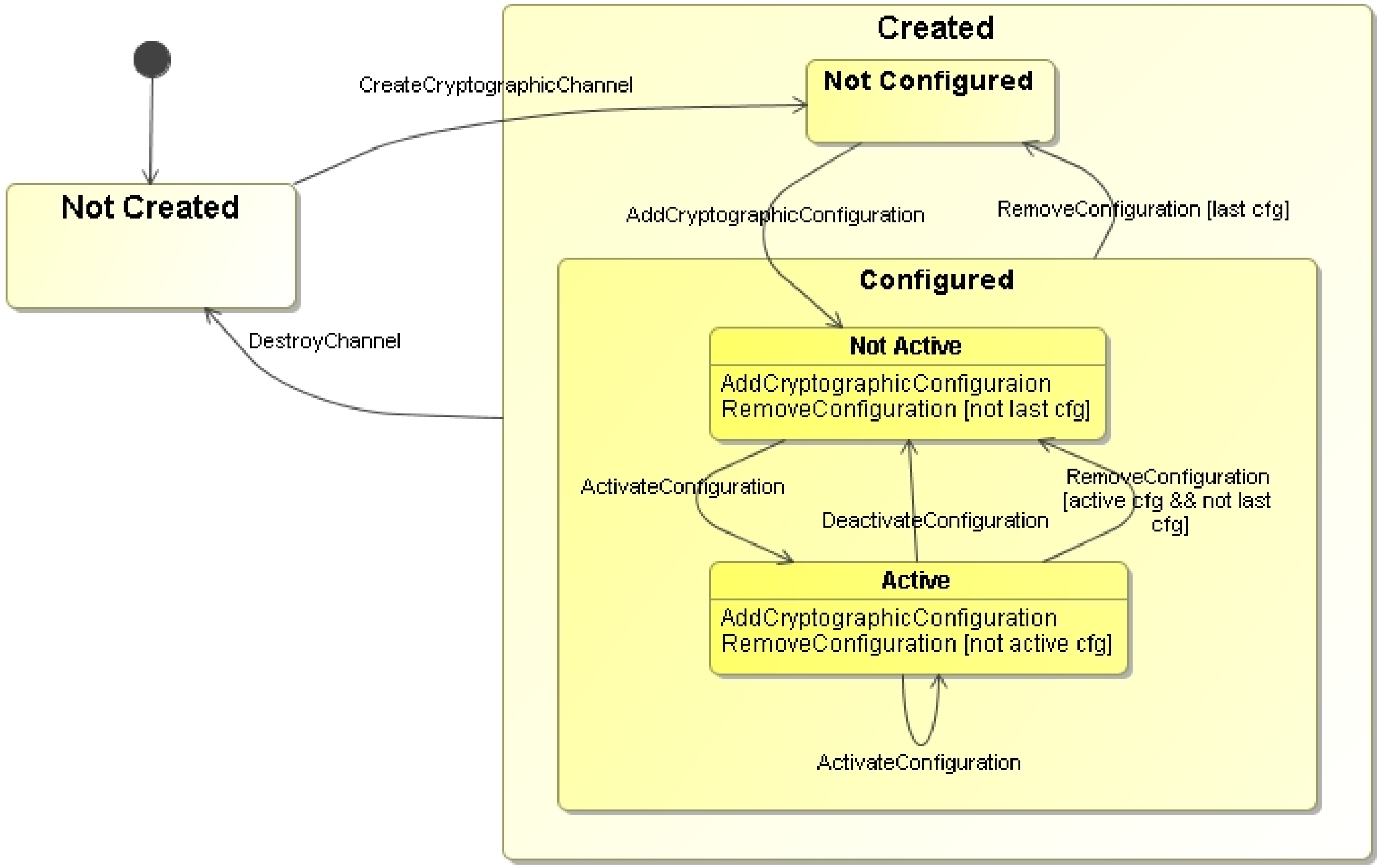


Channel Management

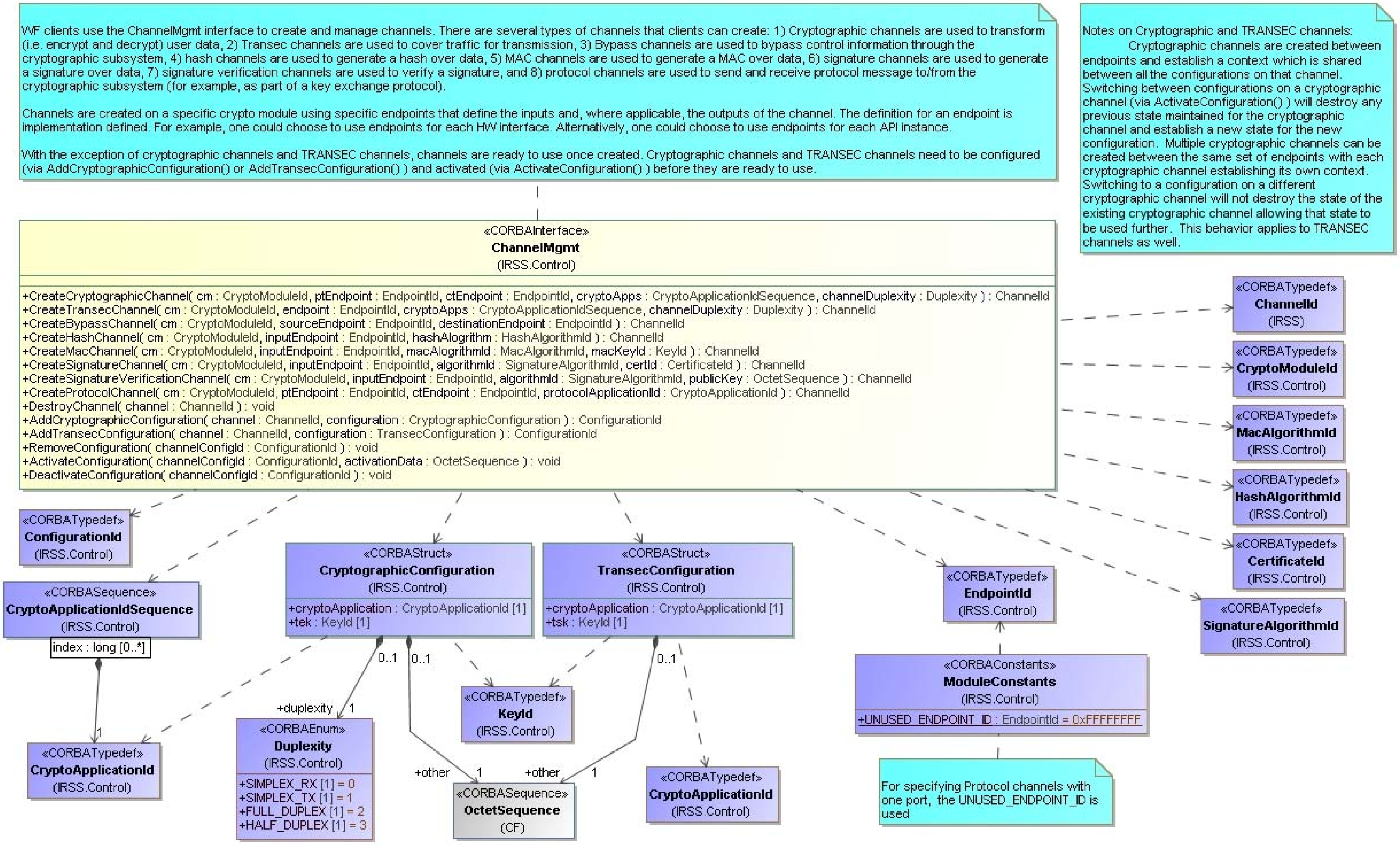
- **Channel Creation:**
  - Done between endpoints on a CM. Definition of endpoints is implementation defined. Examples include:
    - IRSS IDL API instance (e.g. “Port”)
    - Physical HW interfaces into a CM
    - IP address
  - Allocates cryptographic resources for the channel
  - Establishes a cryptographic context for state management
- **Channel Destruction:**
  - Releases cryptographic resources for reuse by another client.
- **Special considerations for Cryptographic and TRANSEC channels**
  - Supports multiple configurations per channel
    - A client must activate a configuration to use it.
    - Only one configuration can be active at a time
    - Activation of a new configuration loses the context of the previous configuration
  - If you need to maintain multiple simultaneous contexts, you should create multiple channels (could use same endpoints)
    - e.g. TDMA stream-based waveforms

Channel Management

Cryptographic Channel Lifecycle


Created
Not Configured
CreateCryptographicChannel
Not Created
AddCryptographicConfiguration
RemoveConfiguration [last cfg]
DestroyChannel
Configured
Not Active
AddCryptographicConfiguraion
RemoveConfiguration [not last cfg]
ActivateConfiguration
DeactivateConfiguration
RemoveConfiguration [active cfg && not last cfg]
Active
AddCryptographicConfiguration
RemoveConfiguration [not active cfg]
ActivateConfiguration

Channel Management



WF clients use the ChannelMgmt interface to create and manage channels. There are several types of channels that clients can create: 1) Cryptographic channels are used to transform (i.e. encrypt and decrypt) user data, 2) Transec channels are used to cover traffic for transmission, 3) Bypass channels are used to bypass control information through the cryptographic subsystem, 4) hash channels are used to generate a hash over data, 5) MAC channels are used to generate a MAC over data, 6) signature channels are used to generate a signature over data, 7) signature verification channels are used to verify a signature, and 8) protocol channels are used to send and receive protocol message to/from the cryptographic subsystem (for example, as part of a key exchange protocol).

Channels are created on a specific crypto module using specific endpoints that define the inputs and, where applicable, the outputs of the channel. The definition for an endpoint is implementation defined. For example, one could choose to use endpoints for each HW interface. Alternatively, one could choose to use endpoints for each API instance.

With the exception of cryptographic channels and TRANSEC channels, channels are ready to use once created. Cryptographic channels and TRANSEC channels need to be configured (via AddCryptographicConfiguration() or AddTransecConfiguration()) and activated (via ActivateConfiguration()) before they are ready to use.

Notes on Cryptographic and TRANSEC channels:

Cryptographic channels are created between endpoints and establish a context which is shared between all the configurations on that channel. Switching between configurations on a cryptographic channel (via ActivateConfiguration()) will destroy any previous state maintained for the cryptographic channel and establish a new state for the new configuration. Multiple cryptographic channels can be created between the same set of endpoints with each cryptographic channel establishing its own context. Switching to a configuration on a different cryptographic channel will not destroy the state of the existing cryptographic channel allowing that state to be used further. This behavior applies to TRANSEC channels as well.

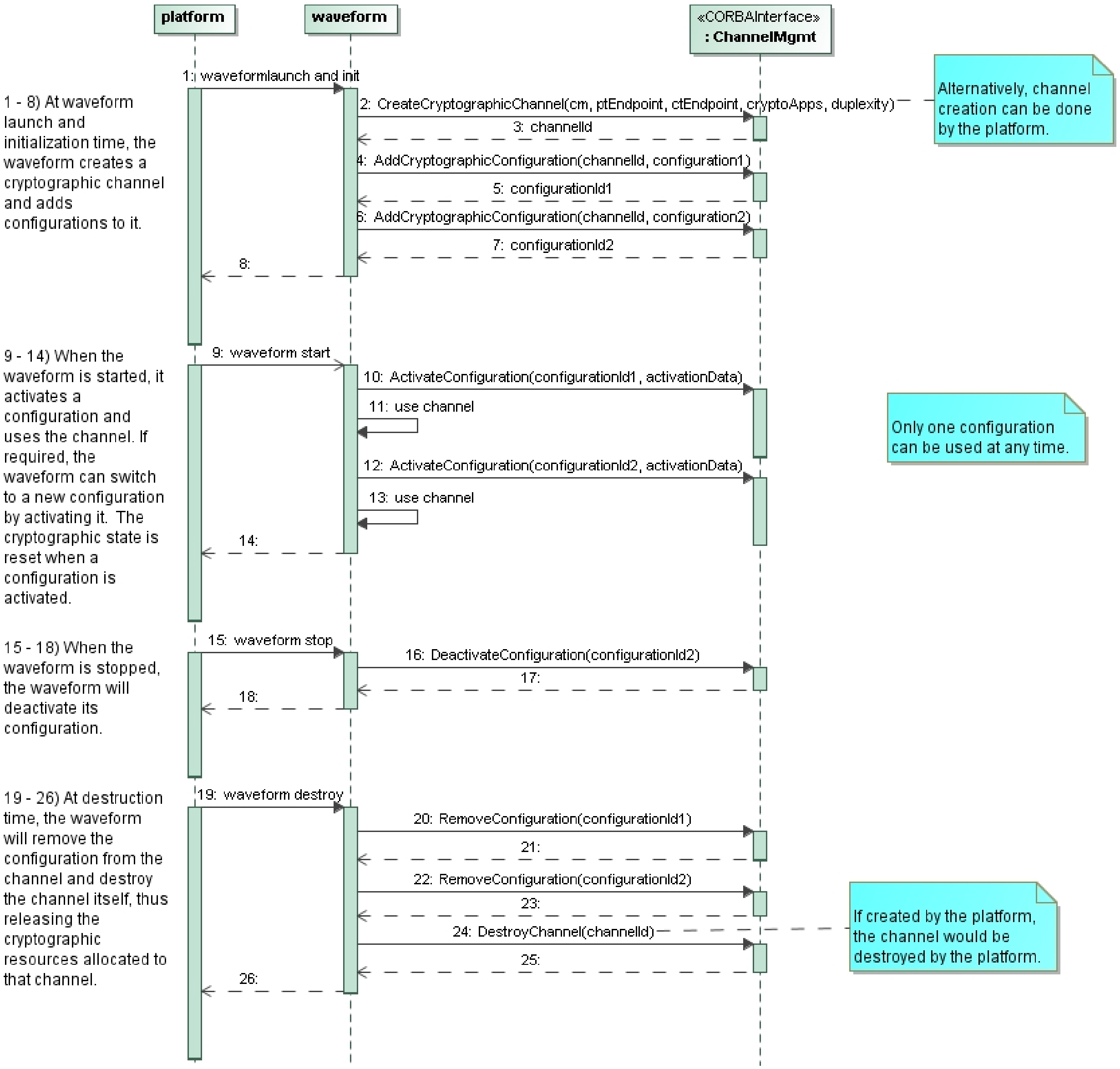
«CORBAInterface»
ChannelMgmt
(IRSS.Control)

```
+CreateCryptographicChannel( cm : CryptoModuleId, ptEndpoint : EndpointId, ctEndpoint : EndpointId, cryptoApps : CryptoApplicationIdSequence, channelDuplexity : Duplexity ) : ChannelId
+CreateTransecChannel( cm : CryptoModuleId, endpoint : EndpointId, cryptoApps : CryptoApplicationIdSequence, channelDuplexity : Duplexity ) : ChannelId
+CreateBypassChannel( cm : CryptoModuleId, sourceEndpoint : EndpointId, destinationEndpoint : EndpointId ) : ChannelId
+CreateHashChannel( cm : CryptoModuleId, inputEndpoint : EndpointId, hashAlogrithm : HashAlgorithmId ) : ChannelId
+CreateMacChannel( cm : CryptoModuleId, inputEndpoint : EndpointId, macAlogrithmId : MacAlgorithmId, macKeyId : KeyId ) : ChannelId
+CreateSignatureChannel( cm : CryptoModuleId, inputEndpoint : EndpointId, algorithmId : SignatureAlgorithmId, certId : CertificateId ) : ChannelId
+CreateSignatureVerificationChannel( cm : CryptoModuleId, inputEndpoint : EndpointId, algorithmId : SignatureAlgorithmId, publicKey : OctetSequence ) : ChannelId
+CreateProtocolChannel( cm : CryptoModuleId, ptEndpoint : EndpointId, ctEndpoint : EndpointId, protocolApplicationId : CryptoApplicationId ) : ChannelId
+DestroyChannel( channel : ChannelId ) : void
+AddCryptographicConfiguration( channel : ChannelId, configuration : CryptographicConfiguration ) : ConfigurationId
+AddTransecConfiguration( channel : ChannelId, configuration : TransecConfiguration ) : ConfigurationId
+RemoveConfiguration( channelConfigId : ConfigurationId ) : void
+ActivateConfiguration( channelConfigId : ConfigurationId, activationData : OctetSequence ) : void
+DeactivateConfiguration( channelConfigId : ConfigurationId ) : void
```

«CORBATypedef» ChannelId (IRSS)
«CORBATypedef» CryptoModuleId (IRSS.Control)
«CORBATypedef» MacAlgorithmId (IRSS.Control)
«CORBATypedef» HashAlgorithmId (IRSS.Control)
«CORBATypedef» CertificateId (IRSS.Control)
«CORBATypedef» SignatureAlgorithmId (IRSS.Control)
«CORBATypedef» ConfigurationId (IRSS.Control)
«CORBASequence» CryptoApplicationIdSequence (IRSS.Control)
index : long [0..*]
1
«CORBATypedef» CryptoApplicationId (IRSS.Control)
«CORBAStruct» CryptographicConfiguration (IRSS.Control)
+cryptoApplication : CryptoApplicationId [1]
+tek : KeyId [1]
«CORBAStruct» TransecConfiguration (IRSS.Control)
+cryptoApplication : CryptoApplicationId [1]
+tsk : KeyId [1]
0..1
0..1
0..1
+duplexity
1
«CORBAEnum» Duplexity (IRSS.Control)
+SIMPLEX_RX [1] = 0
+SIMPLEX_TX [1] = 1
+FULL_DUPLEX [1] = 2
+HALF_DUPLEX [1] = 3
«CORBATypedef» KeyId (IRSS.Control)
+other
1
+other
1
«CORBASequence» OctetSequence (CF)
«CORBATypedef» CryptoApplicationId (IRSS.Control)
«CORBATypedef» EndpointId (IRSS.Control)
«CORBAConstants» ModuleConstants (IRSS.Control)
+UNUSED_ENDPOINT_ID : EndpointId = 0xFFFFFFFF

For specifying Protocol channels with one port, the UNUSED_ENDPOINT_ID is used

Channel Management – Single Channel



1 - 8) At waveform launch and initialization time, the waveform creates a cryptographic channel and adds configurations to it.

1: waveformlaunch and init

2: CreateCryptographicChannel(cm, ptEndpoint, ctEndpoint, cryptoApps, duplexity)

3: channelId

4: AddCryptographicConfiguration(channelId, configuration1)

5: configurationId1

6: AddCryptographicConfiguration(channelId, configuration2)

7: configurationId2

8:

Alternatively, channel creation can be done by the platform.

9 - 14) When the waveform is started, it activates a configuration and uses the channel. If required, the waveform can switch to a new configuration by activating it. The cryptographic state is reset when a configuration is activated.

9: waveform start

10: ActivateConfiguration(configurationId1, activationData)

11: use channel

12: ActivateConfiguration(configurationId2, activationData)

13: use channel

14:

Only one configuration can be used at any time.

15 - 18) When the waveform is stopped, the waveform will deactivate its configuration.

15: waveform stop

16: DeactivateConfiguration(configurationId2)

17:

18:

19 - 26) At destruction time, the waveform will remove the configuration from the channel and destroy the channel itself, thus releasing the cryptographic resources allocated to that channel.

19: waveform destroy

20: RemoveConfiguration(configurationId1)

21:

22: RemoveConfiguration(configurationId2)

23:

24: DestroyChannel(channelId)

25:

26:

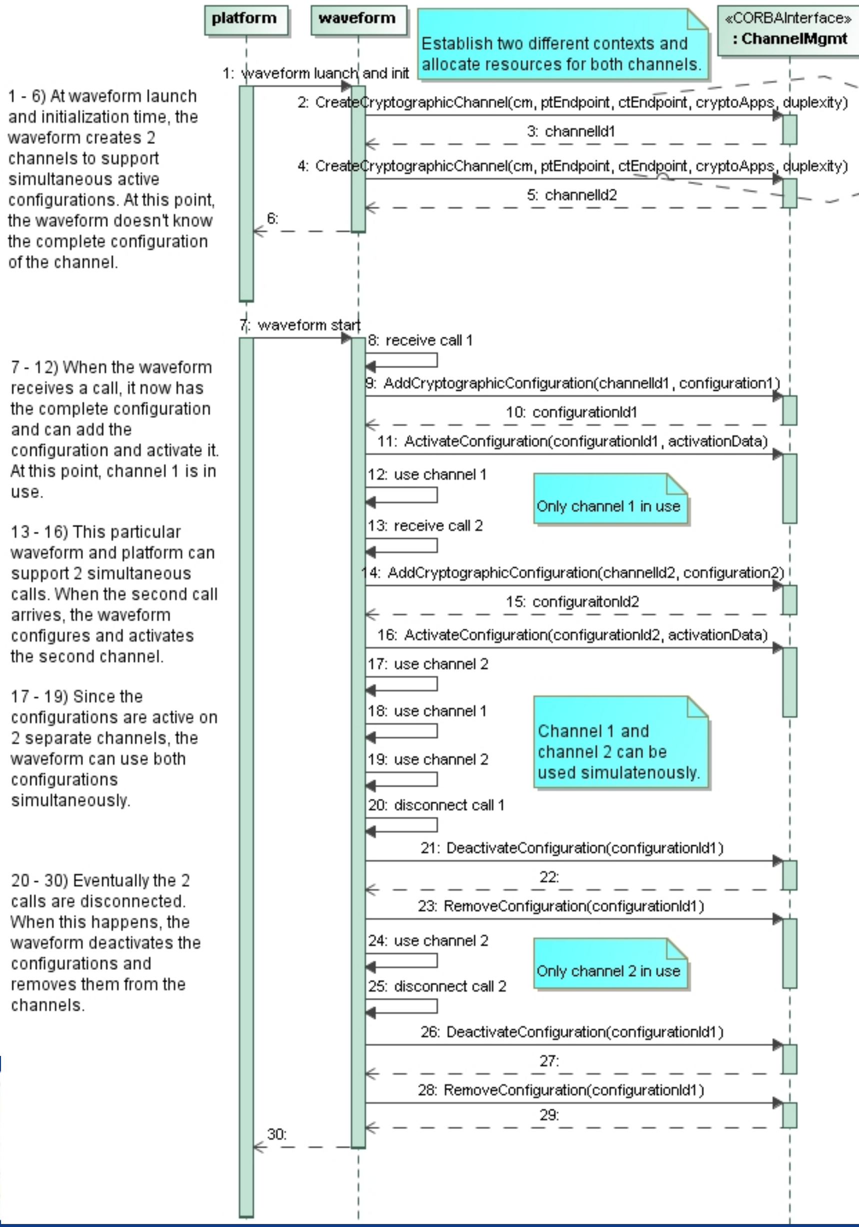
If created by the platform, the channel would be destroyed by the platform.

WIRELESS INNOVATION FORUM

Driving the future of radio communications and systems worldwide

SDR forum version 2.0

Channel Management – Multichannel



platform

waveform

«CORBAInterface» : ChannelMgmt

Establish two different contexts and allocate resources for both channels.

These channels may use the same sets of endpoints if the crypto module supports it. Otherwise, two different sets of endpoints would be used to create the two channels.

1 - 6) At waveform launch and initialization time, the waveform creates 2 channels to support simultaneous active configurations. At this point, the waveform doesn't know the complete configuration of the channel.

1: waveform luanch and init

2: CreateCryptographicChannel(cm, ptEndpoint, ctEndpoint, cryptoApps, duplexity)

3: channelId1

4: CreateCryptographicChannel(cm, ptEndpoint, ctEndpoint, cryptoApps, duplexity)

5: channelId2

6:

7 - 12) When the waveform receives a call, it now has the complete configuration and can add the configuration and activate it. At this point, channel 1 is in use.

7: waveform start

8: receive call 1

9: AddCryptographicConfiguration(channelId1, configuration1)

10: configurationId1

11: ActivateConfiguration(configurationId1, activationData)

12: use channel 1

Only channel 1 in use

13 - 16) This particular waveform and platform can support 2 simultaneous calls. When the second call arrives, the waveform configures and activates the second channel.

13: receive call 2

14: AddCryptographicConfiguration(channelId2, configuration2)

15: configuraitonId2

16: ActivateConfiguration(configurationId2, activationData)

17 - 19) Since the configurations are active on 2 separate channels, the waveform can use both configurations simultaneously.

17: use channel 2

18: use channel 1

19: use channel 2

Channel 1 and channel 2 can be used simultaneously.

20 - 30) Eventually the 2 calls are disconnected. When this happens, the waveform deactivates the configurations and removes them from the channels.

20: disconnect call 1

21: DeactivateConfiguration(configurationId1)

22:

23: RemoveConfiguration(configurationId1)

24: use channel 2

Only channel 2 in use

25: disconnect call 2

26: DeactivateConfiguration(configurationId1)

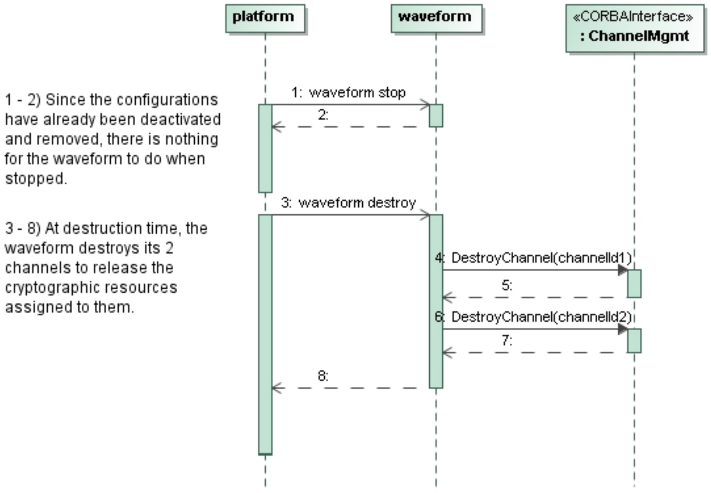
27:

28: RemoveConfiguration(configurationId1)

29:

30:

Multi-Channel Management Usage (cont.)



platform

waveform

«CORBAInterface» : ChannelMgmt

1 - 2) Since the configurations have already been deactivated and removed, there is nothing for the waveform to do when stopped.

1: waveform stop

2:

3 - 8) At destruction time, the waveform destroys its 2 channels to release the cryptographic resources assigned to them.

3: waveform destroy

4: DestroyChannel(channelId1)

5:

6: DestroyChannel(channelId2)

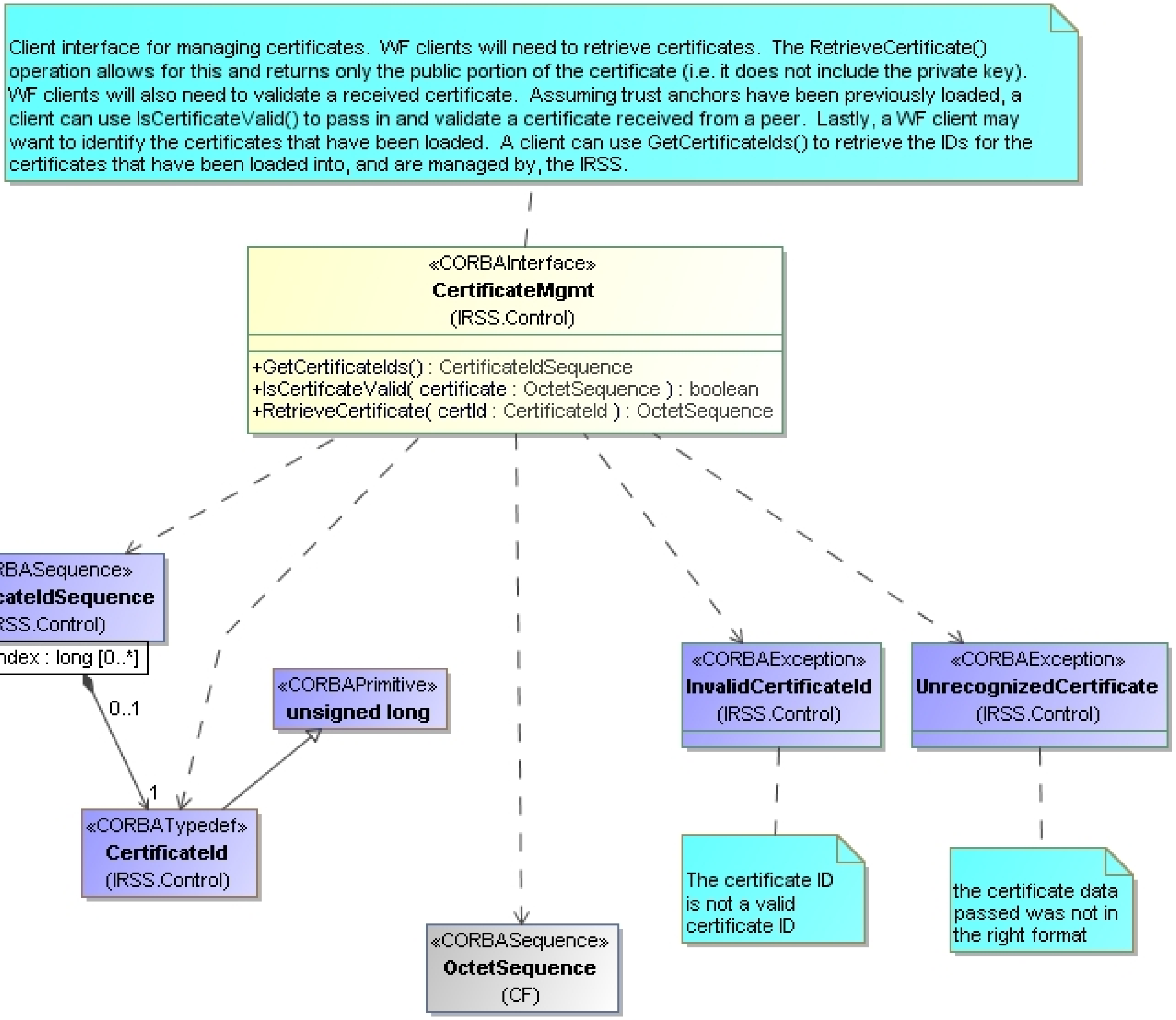
7:

8:

Control Module:

Certificate Management

Certificate Management



Client interface for managing certificates. WF clients will need to retrieve certificates. The RetrieveCertificate() operation allows for this and returns only the public portion of the certificate (i.e. it does not include the private key). WF clients will also need to validate a received certificate. Assuming trust anchors have been previously loaded, a client can use IsCertificateValid() to pass in and validate a certificate received from a peer. Lastly, a WF client may want to identify the certificates that have been loaded. A client can use GetCertificateIds() to retrieve the IDs for the certificates that have been loaded into, and are managed by, the IRSS.

«CORBAInterface»
CertificateMgmt
(IRSS.Control)

+GetCertificateIds() : CertificateIdSequence
+IsCertifcateValid(certificate : OctetSequence) : boolean
+RetrieveCertificate(certId : CertificateId) : OctetSequence

«CORBASequence»
CertificateIdSequence
(IRSS.Control)

index : long [0..*]

0..1

1

«CORBAPrimitive»
unsigned long

«CORBATypedef»
CertificateId
(IRSS.Control)

«CORBASequence»
OctetSequence
(CF)

«CORBAException»
InvalidCertificateId
(IRSS.Control)

«CORBAException»
UnrecognizedCertificate
(IRSS.Control)

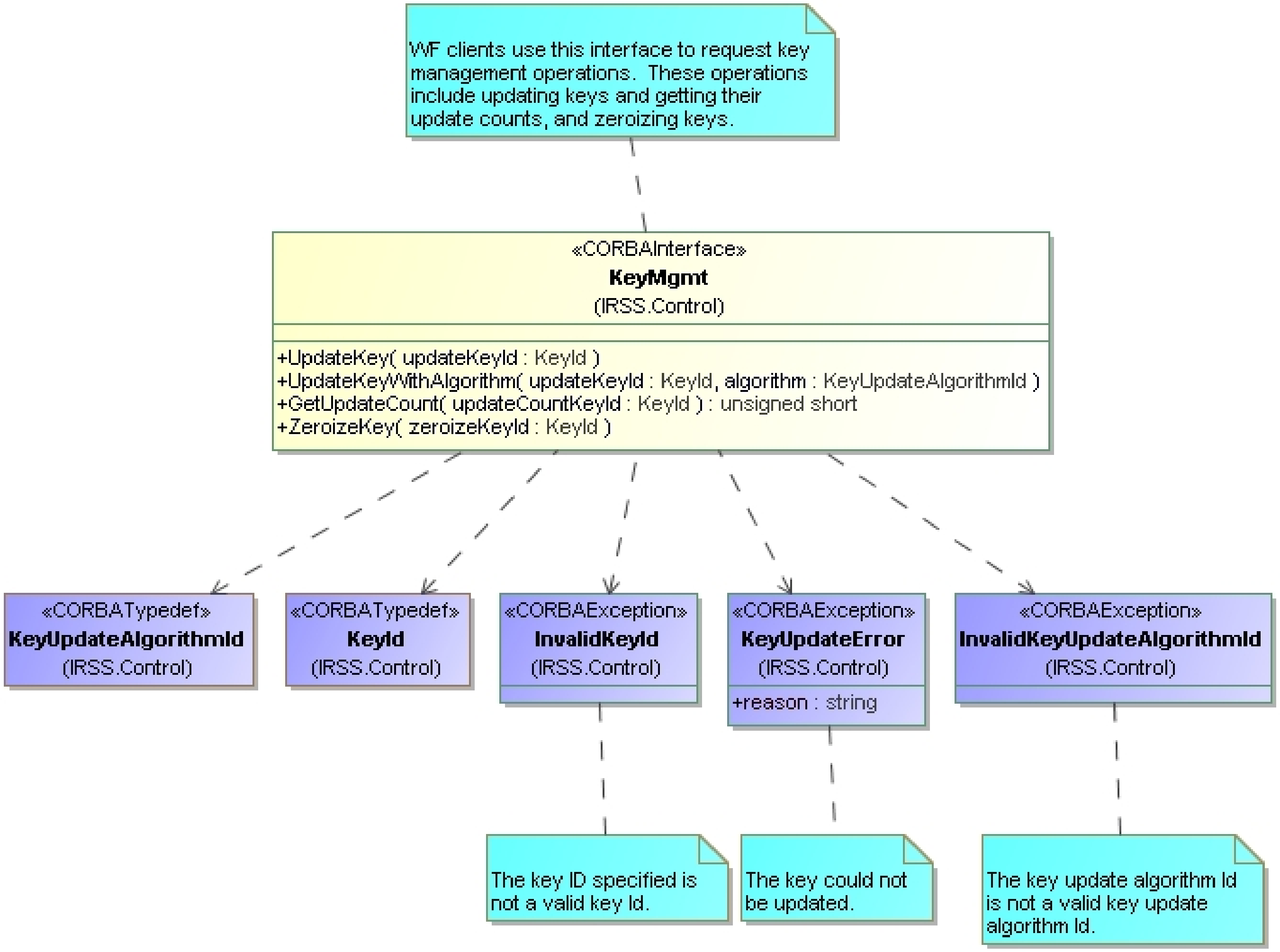
The certificate ID is not a valid certificate ID

the certificate data passed was not in the right format

Control Module:

Key Management

Key Management



WF clients use this interface to request key management operations. These operations include updating keys and getting their update counts, and zeroizing keys.

«CORBAInterface»
KeyMgmt
(IRSS.Control)

+UpdateKey(updateKeyId : KeyId)
+UpdateKeyWithAlgorithm(updateKeyId : KeyId, algorithm : KeyUpdateAlgorithmId)
+GetUpdateCount(updateCountKeyId : KeyId) : unsigned short
+ZeroizeKey(zeroizeKeyId : KeyId)

«CORBATypedef»
KeyUpdateAlgorithmId
(IRSS.Control)

«CORBATypedef»
KeyId
(IRSS.Control)

«CORBAException»
InvalidKeyId
(IRSS.Control)

«CORBAException»
KeyUpdateError
(IRSS.Control)
+reason : string

«CORBAException»
InvalidKeyUpdateAlgorithmId
(IRSS.Control)

The key ID specified is not a valid key Id.

The key could not be updated.

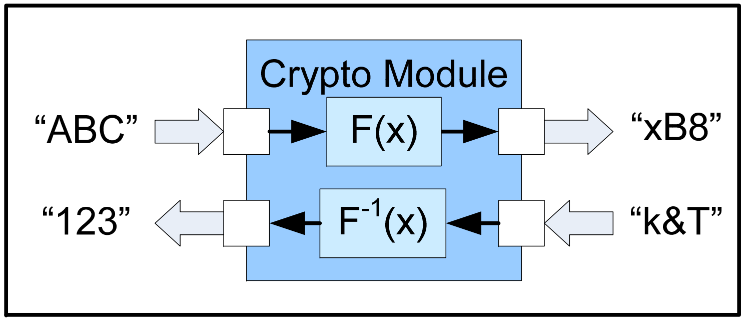
The key update algorithm Id is not a valid key update algorithm Id.

INFOSEC Module:

Cryptographic Channels

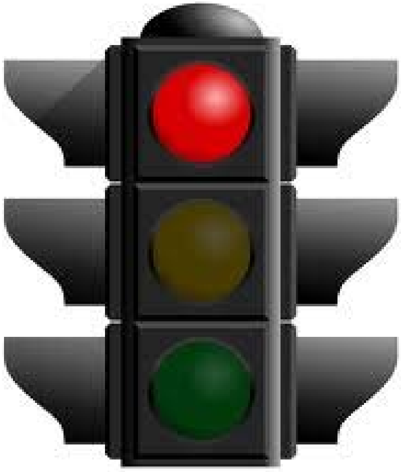
Cryptographic Channels

- **Used to encrypt/decrypt user data via transform requests**
  - Port connections distinguish encrypt requests from decrypt requests
    - Operation is the same: e.g. TransformStream(…)
- **API defines *Channel* and *Consumer* interfaces for both the IRSS and waveform clients, respectively**
- **Two types of transformations: streaming and packet-based**
  - Streams are generally long “messages” processed across multiple calls to the security subsystem
    - Tagged with SOM and EOM to delimit start and end of messages
    - Cryptographic state is maintained across calls
    - Use cases: legacy circuit-switched waveforms, file encryption/decryption
  - Packets are short, self-contained data bundles processed as a unit
    - Multiple packets processed via a single transform request
    - Key selection could vary from packet to packet
    - Use cases: networking waveforms

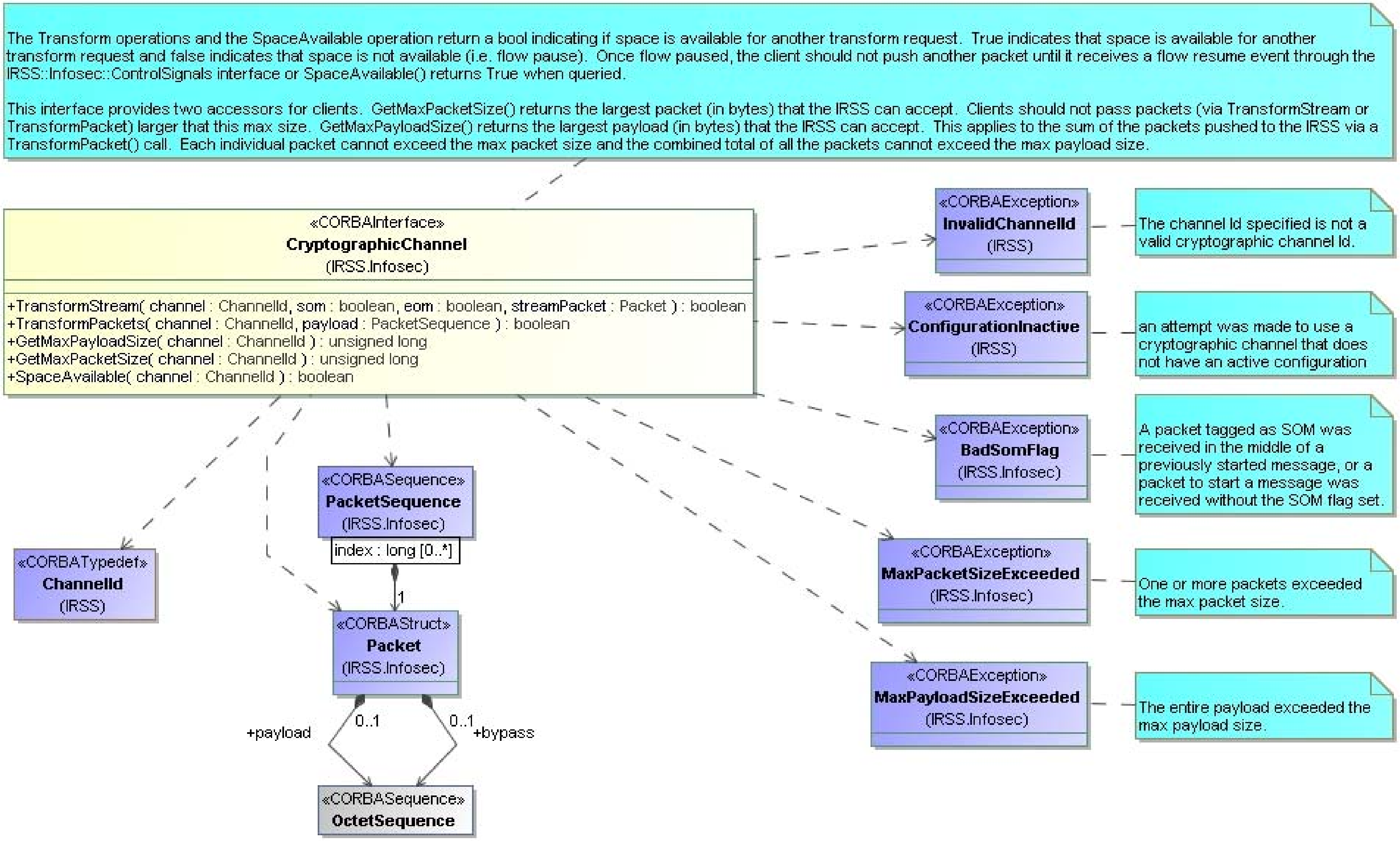


Cryptographic Channels

- **Flow control to the crypto module is defined**
  - Patterned after public JTRS APIs
  - "Space Available" boolean enables flow control via return value
  - Control signal back to client indicates resume
- **Flow control to the waveform client is not defined**
  - RSS can flow pause the waveform, but not vice versa
  - In most cases, waveform is designed at a system level to unconditionally accept the crypto's output
  - Waveform ⇔ Waveform control flows can be instituted if full flow control is required



Cryptographic Channels - Provider



The Transform operations and the SpaceAvailable operation return a bool indicating if space is available for another transform request. True indicates that space is available for another transform request and false indicates that space is not available (i.e. flow pause). Once flow paused, the client should not push another packet until it receives a flow resume event through the IRSS::Infosec::ControlSignals interface or SpaceAvailable() returns True when queried.

This interface provides two accessors for clients. GetMaxPacketSize() returns the largest packet (in bytes) that the IRSS can accept. Clients should not pass packets (via TransformStream or TransformPacket) larger that this max size. GetMaxPayloadSize() returns the largest payload (in bytes) that the IRSS can accept. This applies to the sum of the packets pushed to the IRSS via a TransformPacket() call. Each individual packet cannot exceed the max packet size and the combined total of all the packets cannot exceed the max payload size.

«CORBAInterface»
CryptographicChannel
(IRSS.Infosec)

+TransformStream(channel : ChannelId, som : boolean, eom : boolean, streamPacket : Packet) : boolean
+TransformPackets(channel : ChannelId, payload : PacketSequence) : boolean
+GetMaxPayloadSize(channel : ChannelId) : unsigned long
+GetMaxPacketSize(channel : ChannelId) : unsigned long
+SpaceAvailable(channel : ChannelId) : boolean

«CORBAException»
InvalidChannelId
(IRSS)

The channel Id specified is not a valid cryptographic channel Id.

«CORBAException»
ConfigurationInactive
(IRSS)

an attempt was made to use a cryptographic channel that does not have an active configuration

«CORBAException»
BadSomFlag
(IRSS.Infosec)

A packet tagged as SOM was received in the middle of a previously started message, or a packet to start a message was received without the SOM flag set.

«CORBAException»
MaxPacketSizeExceeded
(IRSS.Infosec)

One or more packets exceeded the max packet size.

«CORBAException»
MaxPayloadSizeExceeded
(IRSS.Infosec)

The entire payload exceeded the max payload size.

«CORBATypedef»
ChannelId
(IRSS)

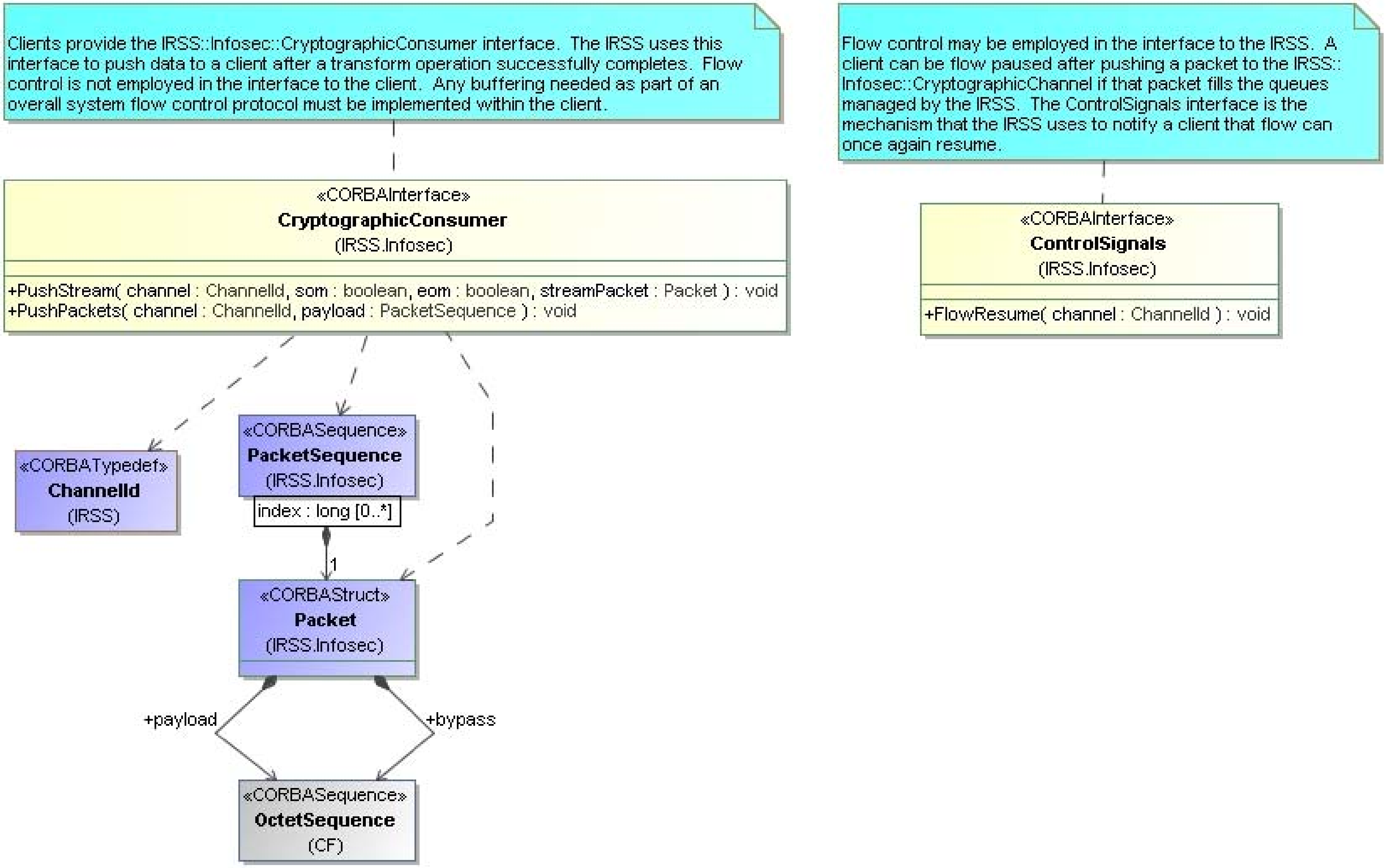
«CORBASequence»
PacketSequence
(IRSS.Infosec)

index : long [0..*]

«CORBAStruct»
Packet
(IRSS.Infosec)

«CORBASequence»
OctetSequence

Cryptographic Channels - Client



Clients provide the IRSS::Infosec::CryptographicConsumer interface. The IRSS uses this interface to push data to a client after a transform operation successfully completes. Flow control is not employed in the interface to the client. Any buffering needed as part of an overall system flow control protocol must be implemented within the client.

Flow control may be employed in the interface to the IRSS. A client can be flow paused after pushing a packet to the IRSS::Infosec::CryptographicChannel if that packet fills the queues managed by the IRSS. The ControlSignals interface is the mechanism that the IRSS uses to notify a client that flow can once again resume.

«CORBAInterface»
CryptographicConsumer
(IRSS.Infosec)

```
+PushStream( channel : ChannelId, som : boolean, eom : boolean, streamPacket : Packet ) : void
+PushPackets( channel : ChannelId, payload : PacketSequence ) : void
```

«CORBAInterface»
ControlSignals
(IRSS.Infosec)

```
+FlowResume( channel : ChannelId ) : void
```

«CORBATypedef»
ChannelId
(IRSS)

«CORBASequence»
PacketSequence
(IRSS.Infosec)

index : long [0..*]

«CORBAStruct»
Packet
(IRSS.Infosec)

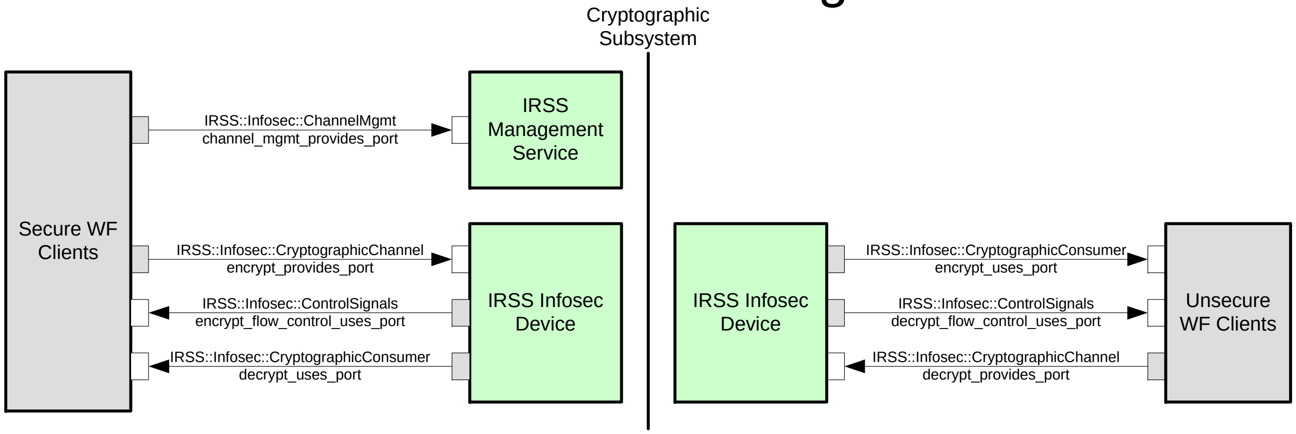
«CORBASequence»
OctetSequence
(CF)

Driving the future of radio communications and systems worldwide

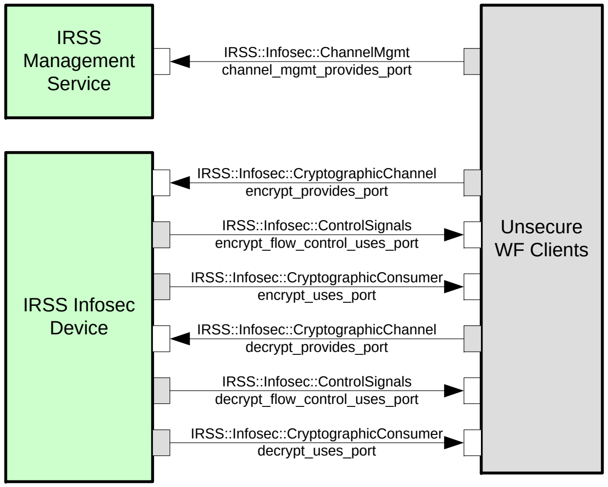
version 2.0

Cryptographic Channels

Two-Sided Port Diagram


Cryptographic Subsystem
Secure WF Clients
IRSS Management Service
IRSS Infosec Device
IRSS Infosec Device
Unsecure WF Clients
IRSS::Infosec::ChannelMgmt
channel_mgmt_provides_port
IRSS::Infosec::CryptographicChannel
encrypt_provides_port
IRSS::Infosec::ControlSignals
encrypt_flow_control_uses_port
IRSS::Infosec::CryptographicConsumer
decrypt_uses_port
IRSS::Infosec::CryptographicConsumer
encrypt_uses_port
IRSS::Infosec::ControlSignals
decrypt_flow_control_uses_port
IRSS::Infosec::CryptographicChannel
decrypt_provides_port

One-Sided Port Diagram


IRSS Management Service
IRSS Infosec Device
Unsecure WF Clients
IRSS::Infosec::ChannelMgmt
channel_mgmt_provides_port
IRSS::Infosec::CryptographicChannel
encrypt_provides_port
IRSS::Infosec::ControlSignals
encrypt_flow_control_uses_port
IRSS::Infosec::CryptographicConsumer
encrypt_uses_port
IRSS::Infosec::CryptographicChannel
decrypt_provides_port
IRSS::Infosec::ControlSignals
decrypt_flow_control_uses_port
IRSS::Infosec::CryptographicConsumer
decrypt_uses_port

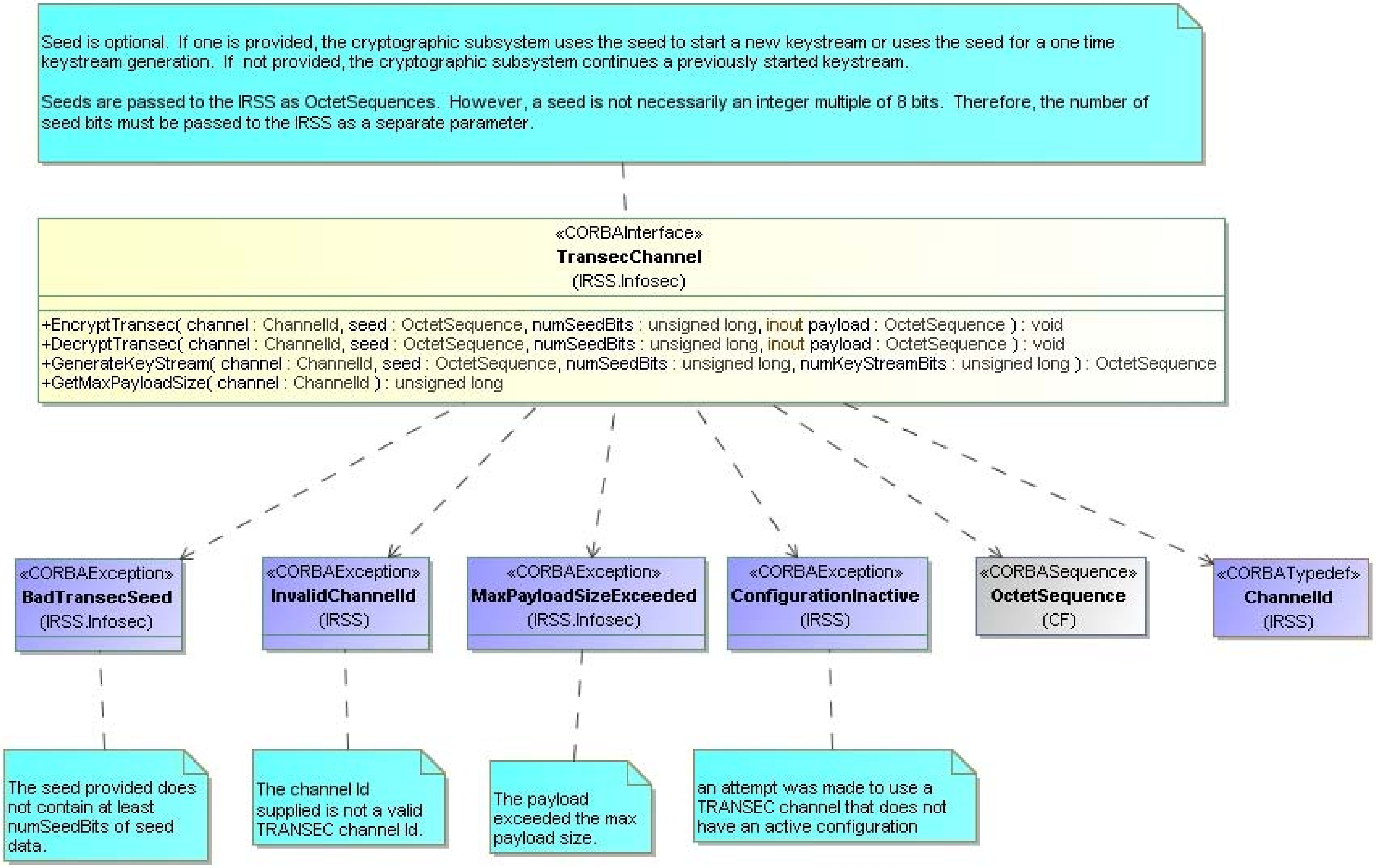
INFOSEC Module:

TRANSEC Channels

TRANSEC Channels

- **Used to cover the *means* of information transfer, not the information itself**
- **Includes support for:**
  - keystream generation – returns a sequence of bits, based on a seed, used by the waveform to manipulate a transmission
  - TRANSEC encryption/decryption – interface functions similarly to encryption, in that information is provided, manipulated by the cryptographic application, and returned.
- **Seed is optional**
  - If provided (typical case), used to start a TRANSEC request (e.g. keystream generation) or for a one time TRANSEC request
  - If not provided, used to continue a TRANSEC request

TRANSEC Channels



Seed is optional. If one is provided, the cryptographic subsystem uses the seed to start a new keystream or uses the seed for a one time keystream generation. If not provided, the cryptographic subsystem continues a previously started keystream.

Seeds are passed to the IRSS as OctetSequences. However, a seed is not necessarily an integer multiple of 8 bits. Therefore, the number of seed bits must be passed to the IRSS as a separate parameter.

«CORBAInterface»
TransecChannel
(IRSS.Infosec)

+EncryptTransec(channel : ChannelId, seed : OctetSequence, numSeedBits : unsigned long, inout payload : OctetSequence) : void
+DecryptTransec(channel : ChannelId, seed : OctetSequence, numSeedBits : unsigned long, inout payload : OctetSequence) : void
+GenerateKeyStream(channel : ChannelId, seed : OctetSequence, numSeedBits : unsigned long, numKeyStreamBits : unsigned long) : OctetSequence
+GetMaxPayloadSize(channel : ChannelId) : unsigned long

«CORBAException»
BadTransecSeed
(IRSS.Infosec)

«CORBAException»
InvalidChannelId
(IRSS)

«CORBAException»
MaxPayloadSizeExceeded
(IRSS.Infosec)

«CORBAException»
ConfigurationInactive
(IRSS)

«CORBASequence»
OctetSequence
(CF)

«CORBATypedef»
ChannelId
(IRSS)

The seed provided does not contain at least numSeedBits of seed data.

The channel Id supplied is not a valid TRANSEC channel Id.

The payload exceeded the max payload size.

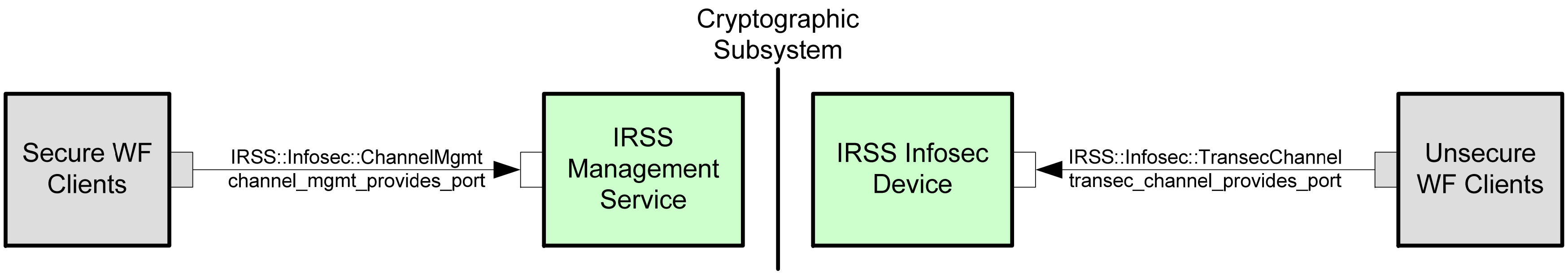
an attempt was made to use a TRANSEC channel that does not have an active configuration

Driving the future of radio communications and systems worldwide

version 2.0

TRANSEC Channels

Two-Sided Port Diagram



Cryptographic Subsystem

Secure WF Clients

IRSS::Infosec::ChannelMgmt
channel_mgmt_provides_port

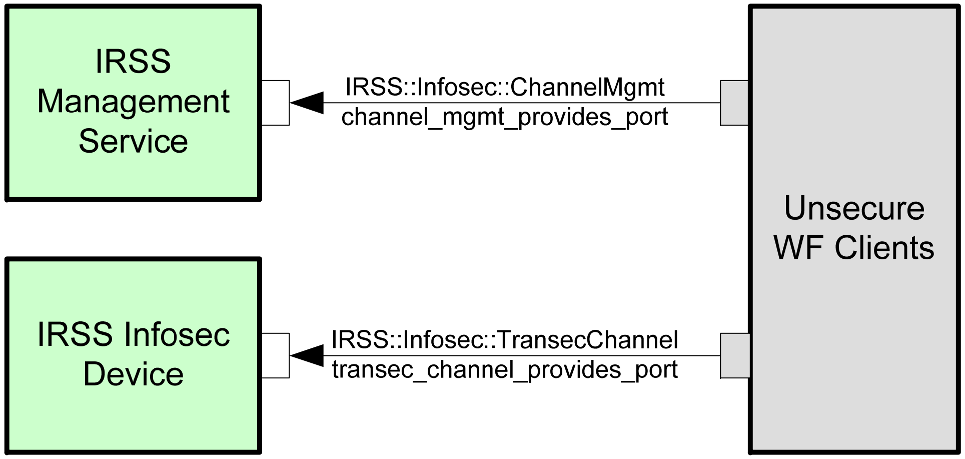
IRSS Management Service

IRSS Infosec Device

IRSS::Infosec::TransecChannel
transec_channel_provides_port

Unsecure WF Clients

One-Sided Port Diagram



IRSS Management Service

IRSS::Infosec::ChannelMgmt
channel_mgmt_provides_port

IRSS Infosec Device

IRSS::Infosec::TransecChannel
transec_channel_provides_port

Unsecure WF Clients

TRANSEC Channels - Usage

Via ChannelMgmt inteface:
1) Create a Transec channel. This allocates the cryptographic resources and returns the channel Id to use.

3 - 6) Add and activate the Transec configuration

Via BypassChannel inteface:
7 - 8) PT side bypasses the transecChId to CT side using a (previously created) bypass channel.

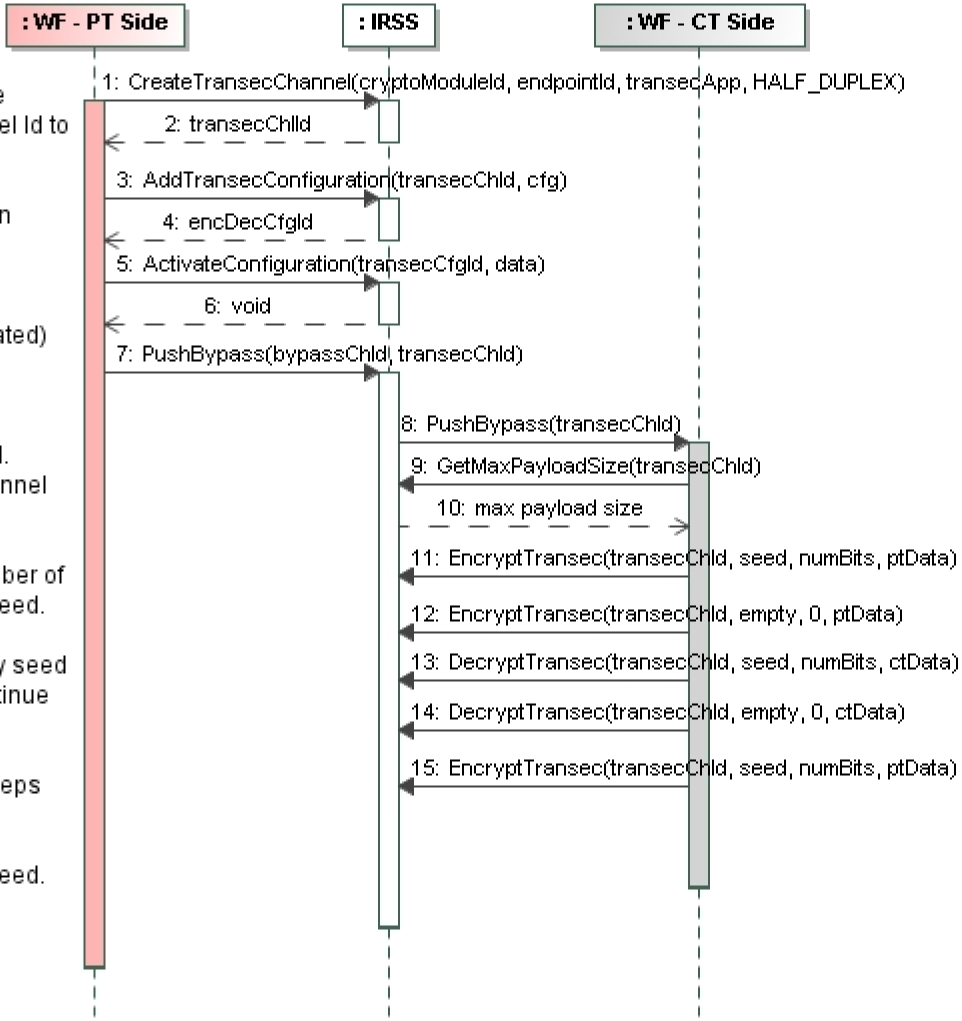
Via TransecChannel inteface:
9 - 10) Get the max payload size for the channel. Packets to be encrypted / decrypted via this channel cannot exceed this max size.

11) Encrypt a packet, passing in seed and number of seed bits. The algorithm is initialized with the seed.

12) Encrypt another packet passing in an empty seed sequence and numSeedBits of 0. This will continue processing without re-initializing the algorithm.

13 - 14) Decrypt instead of encrypt. Similar to steps 11 and 12.

15) Encrypt another packet while specifying a seed. This will re-initialize the algorithm again.



: WF - PT Side

: IRSS

: WF - CT Side

1: CreateTransecChannel(cryptoModuleId, endpointId, transecApp, HALF_DUPLEX)

2: transecChId

3: AddTransecConfiguration(transecChId, cfg)

4: encDecCfgId

5: ActivateConfiguration(transecCfgId, data)

6: void

7: PushBypass(bypassChId, transecChId)

8: PushBypass(transecChId)

9: GetMaxPayloadSize(transecChId)

10: max payload size

11: EncryptTransec(transecChId, seed, numBits, ptData)

12: EncryptTransec(transecChId, empty, 0, ptData)

13: DecryptTransec(transecChId, seed, numBits, ctData)

14: DecryptTransec(transecChId, empty, 0, ctData)

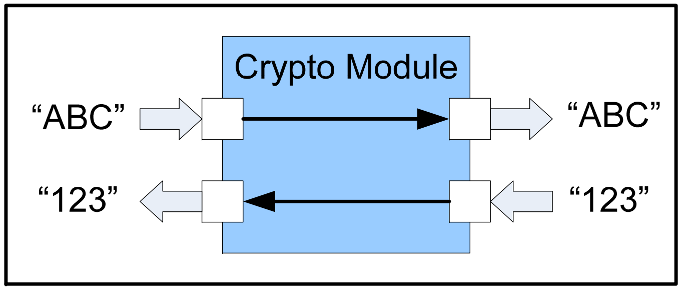
15: EncryptTransec(transecChId, seed, numBits, ptData)

Bypass Module:

Bypass Channels

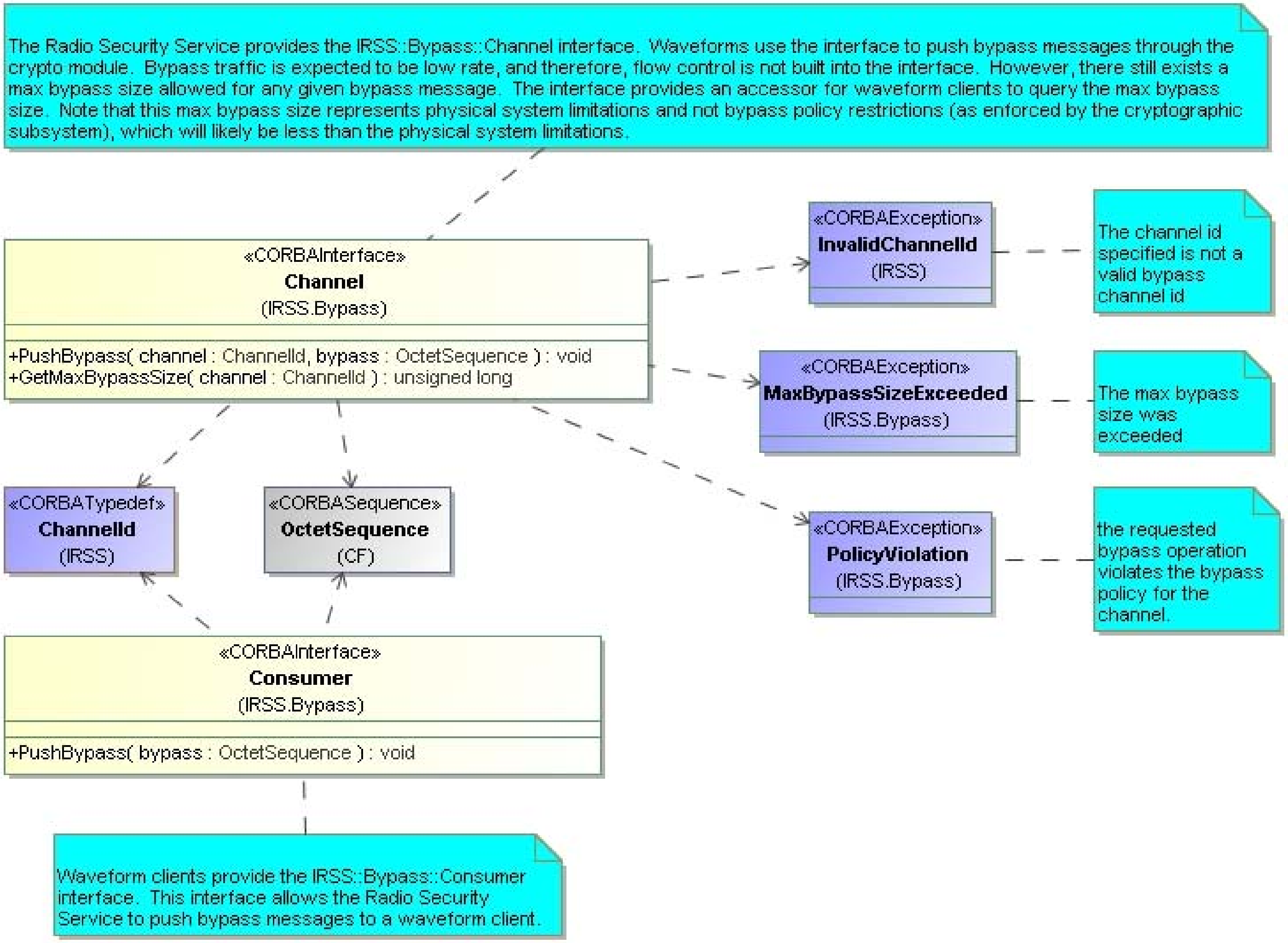
Bypass Channels

- **Used to bypass control traffic through the security subsystem in platforms with multiple security domains**
- **Bypass channel are unidirectional**
  - Allows for direction dependent policy enforcement
  - Create two channel for bypass in both directions
- **API defines *Channel* and *Consumer* interfaces for both the IRSS and waveform clients**
  - Clients (on one side) invoke push operations on the IRSS to initiate a bypass request
  - IRSS (on the alternate side) invokes push requests on the clients to complete the request



- **Flow control is not defined in either interface**
  - Bypass traffic is expected to be low data rate.

Bypass Channels



The Radio Security Service provides the IRSS::Bypass::Channel interface. Waveforms use the interface to push bypass messages through the crypto module. Bypass traffic is expected to be low rate, and therefore, flow control is not built into the interface. However, there still exists a max bypass size allowed for any given bypass message. The interface provides an accessor for waveform clients to query the max bypass size. Note that this max bypass size represents physical system limitations and not bypass policy restrictions (as enforced by the cryptographic subsystem), which will likely be less than the physical system limitations.

«CORBAInterface»
Channel
(IRSS.Bypass)

+PushBypass(channel : ChannelId, bypass : OctetSequence) : void
+GetMaxBypassSize(channel : ChannelId) : unsigned long

«CORBAException»
InvalidChannelId
(IRSS)

The channel id specified is not a valid bypass channel id

«CORBAException»
MaxBypassSizeExceeded
(IRSS.Bypass)

The max bypass size was exceeded

«CORBATypedef»
ChannelId
(IRSS)

«CORBASequence»
OctetSequence
(CF)

«CORBAException»
PolicyViolation
(IRSS.Bypass)

the requested bypass operation violates the bypass policy for the channel.

«CORBAInterface»
Consumer
(IRSS.Bypass)

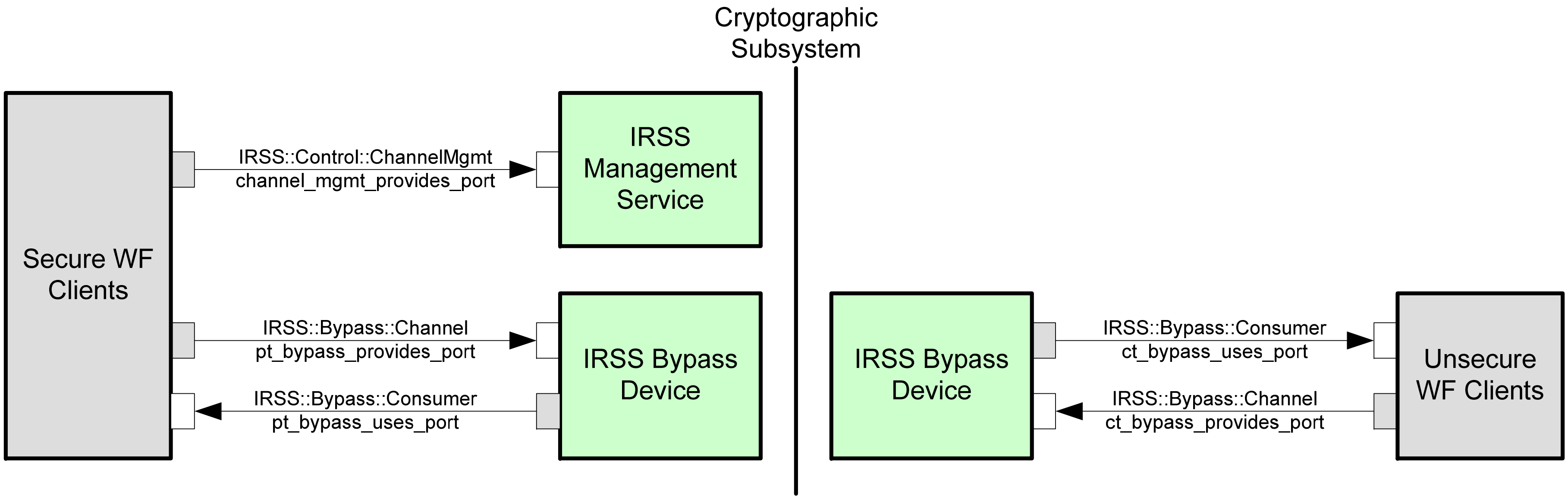
+PushBypass(bypass : OctetSequence) : void

Waveform clients provide the IRSS::Bypass::Consumer interface. This interface allows the Radio Security Service to push bypass messages to a waveform client.

WIRELESS INNOVATION FORUM™ *Driving the future of radio communications and systems worldwide*

SDR forum version 2.0

Bypass Channels



Bypass Channels - Usage

Via ChannelMgmt interface:
1 - 2) The PT side WF component creates a bypass channel for PT to CT bypass.
5 - 6) The PT side WF component creates a bypass channel for CT to PT bypass.

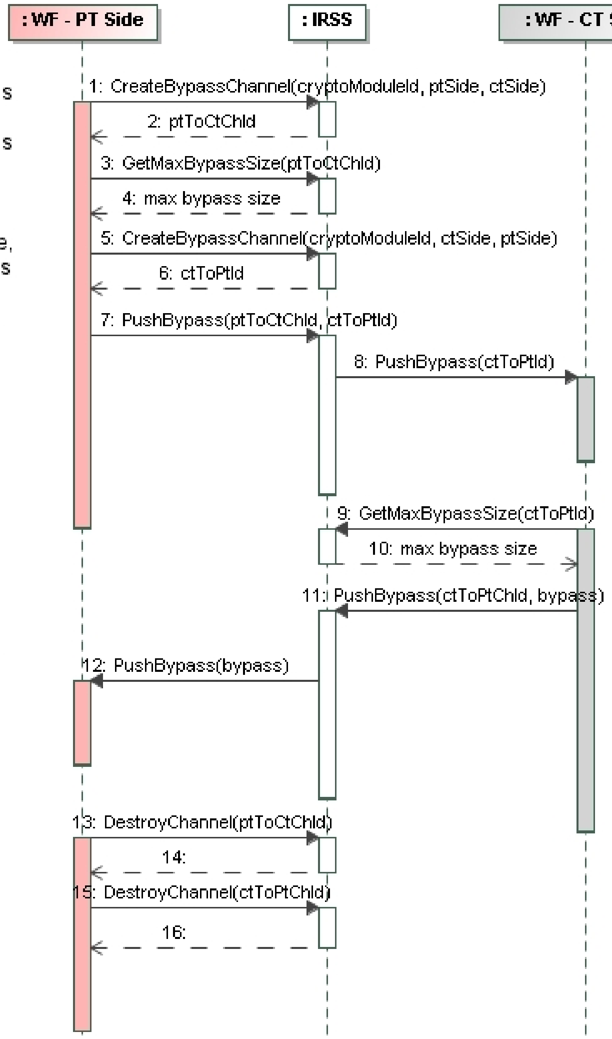
Via Bypass::Channel interface:
3 - 4) Before pushing a bypass message, the waveform must query the max bypass size from the IRSS. This size cannot be exceeded by any one bypass message.
7) The PT side WF pushes a bypass message to the PT side IRSS instance with the ctToPtId.

Via Bypass::Consumer interface
8) The IRSS pushes the bypass message to the CT side WF.

Via Bypass::Channel interface:
9 - 10) The CT Side WF component queries the max bypass size
11) The CT side WF pushes a bypass message to the CT side IRSS instance.

Via Bypass::Consumer interface
12) The IRSS pushes the bypass message to the PT side WF.

Via ChannelMgmt interface:
13 - 16) The bypass channels are destroyed


: WF - PT Side
: IRSS
: WF - CT Side
1: CreateBypassChannel(cryptoModuleId, ptSide, ctSide)
2: ptToCtChId
3: GetMaxBypassSize(ptToCtChId)
4: max bypass size
5: CreateBypassChannel(cryptoModuleId, ctSide, ptSide)
6: ctToPtId
7: PushBypass(ptToCtChId, ctToPtId)
8: PushBypass(ctToPtId)
9: GetMaxBypassSize(ctToPtId)
10: max bypass size
11: PushBypass(ctToPtChId, bypass)
12: PushBypass(bypass)
13: DestroyChannel(ptToCtChId)
14:
15: DestroyChannel(ctToPtChId)
16:

Wireless Innovation Forum

Driving the future of radio communications and systems worldwide

SDR forum version 2.0

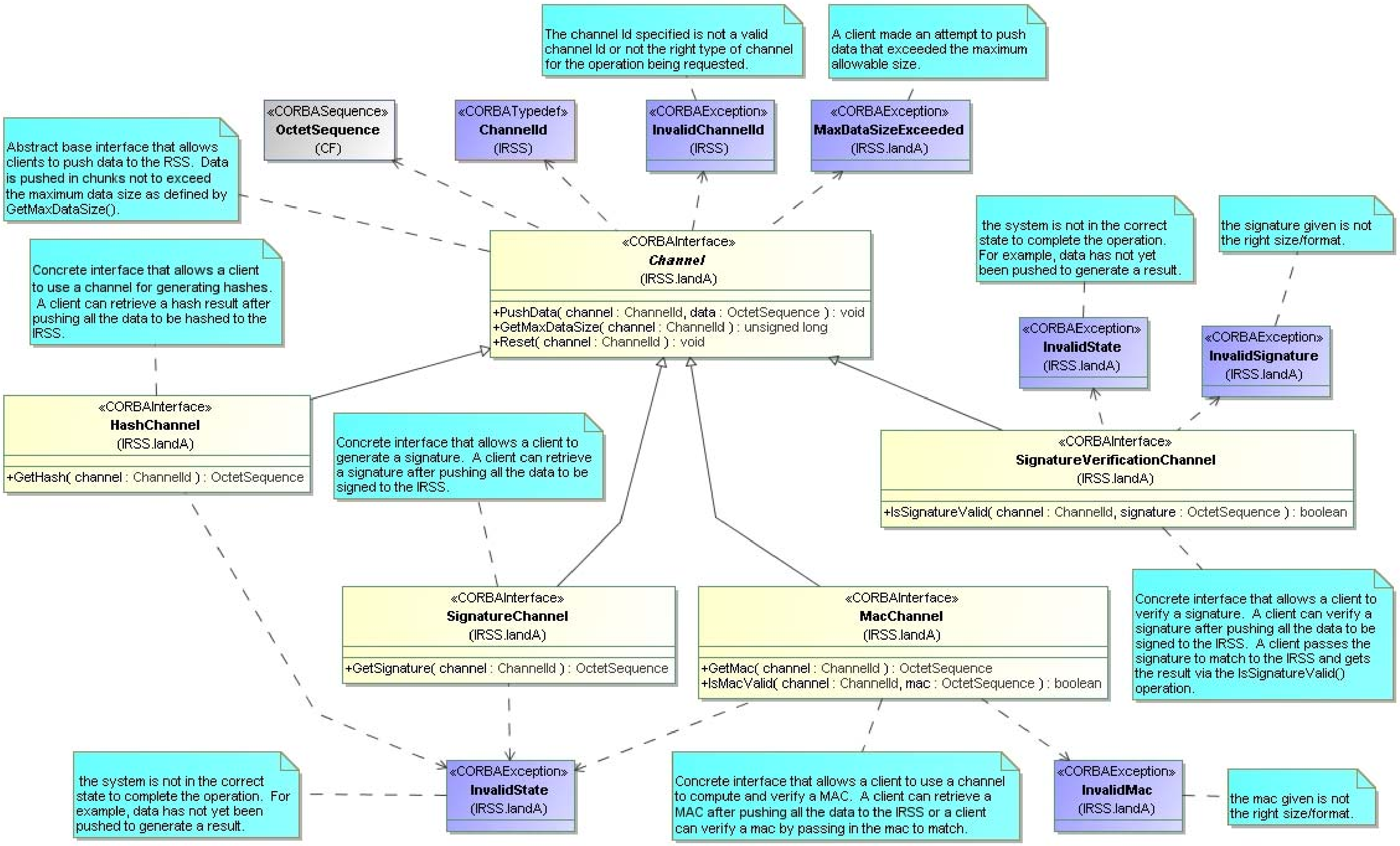
IandA Module:

IandA Channels/Random Interface

IandA Channels/Random Interface

- **Used to provide security services to clients**
  - Hash generation
  - MAC generation/verification
  - Signature generation/verification
  - Random number generation
- **IandA Channels**
  - Common base interface for pushing data to the security subsystem
  - Unique derived interfaces for querying results
- **Random Interface**
  - Used to generate random numbers
  - Supports two modes:
    - True random: uses unpredictable (e.g. noise) conditions to generate random number sequences
    - Pseudorandom: seed based algorithm for generating repeatable random number sequences

IandA Channels



The channel Id specified is not a valid channel Id or not the right type of channel for the operation being requested.

A client made an attempt to push data that exceeded the maximum allowable size.

Abstract base interface that allows clients to push data to the RSS. Data is pushed in chunks not to exceed the maximum data size as defined by GetMaxDataSize().

«CORBASequence»
OctetSequence
(CF)

«CORBATypedef»
ChannelId
(IRSS)

«CORBAException»
InvalidChannelId
(IRSS)

«CORBAException»
MaxDataSizeExceeded
(IRSS.IandA)

«CORBAInterface»
Channel
(IRSS.IandA)

+PushData(channel : ChannelId, data : OctetSequence) : void
+GetMaxDataSize(channel : ChannelId) : unsigned long
+Reset(channel : ChannelId) : void

the system is not in the correct state to complete the operation. For example, data has not yet been pushed to generate a result.

the signature given is not the right size/format.

Concrete interface that allows a client to use a channel for generating hashes. A client can retrieve a hash result after pushing all the data to be hashed to the IRSS.

«CORBAException»
InvalidState
(IRSS.IandA)

«CORBAException»
InvalidSignature
(IRSS.IandA)

«CORBAInterface»
HashChannel
(IRSS.IandA)

+GetHash(channel : ChannelId) : OctetSequence

Concrete interface that allows a client to generate a signature. A client can retrieve a signature after pushing all the data to be signed to the IRSS.

«CORBAInterface»
SignatureVerificationChannel
(IRSS.IandA)

+IsSignatureValid(channel : ChannelId, signature : OctetSequence) : boolean

«CORBAInterface»
SignatureChannel
(IRSS.IandA)

+GetSignature(channel : ChannelId) : OctetSequence

«CORBAInterface»
MacChannel
(IRSS.IandA)

+GetMac(channel : ChannelId) : OctetSequence
+IsMacValid(channel : ChannelId, mac : OctetSequence) : boolean

Concrete interface that allows a client to verify a signature. A client can verify a signature after pushing all the data to be signed to the IRSS. A client passes the signature to match to the IRSS and gets the result via the IsSignatureValid() operation.

the system is not in the correct state to complete the operation. For example, data has not yet been pushed to generate a result.

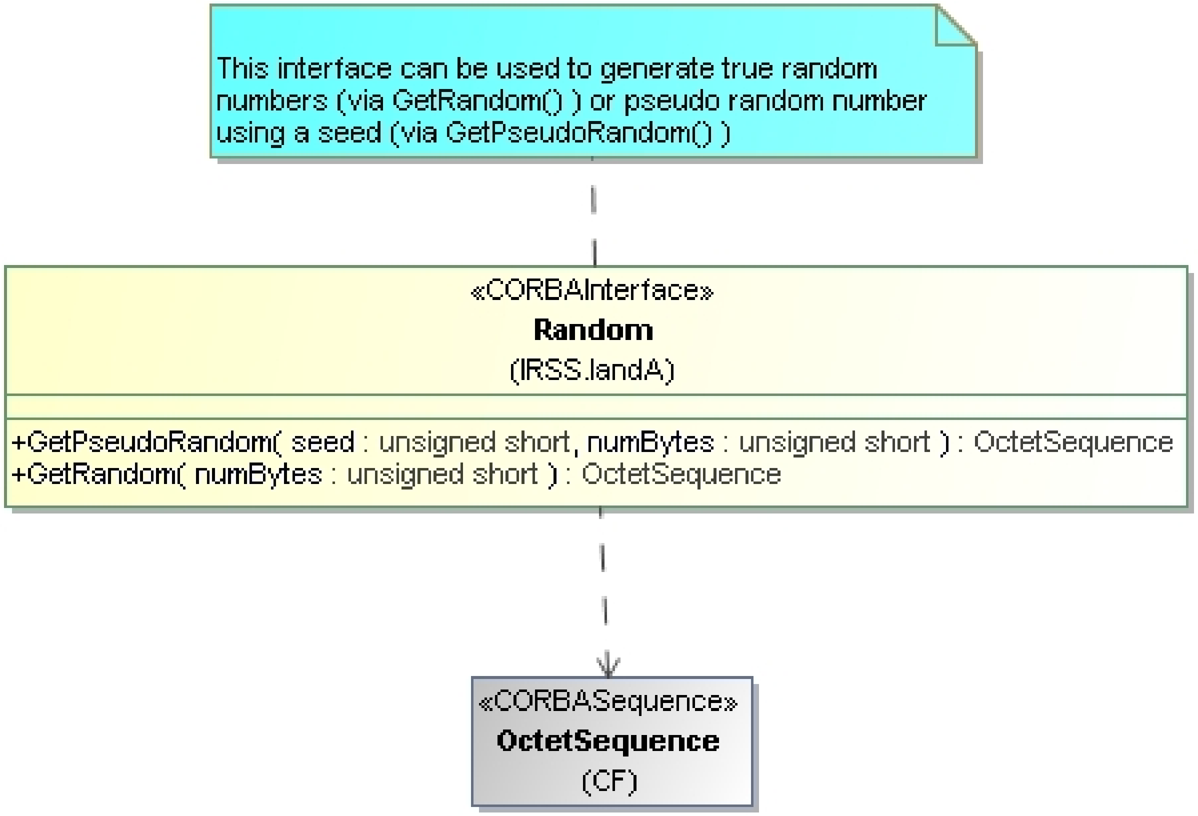
«CORBAException»
InvalidState
(IRSS.IandA)

Concrete interface that allows a client to use a channel to compute and verify a MAC. A client can retrieve a MAC after pushing all the data to the IRSS or a client can verify a mac by passing in the mac to match.

«CORBAException»
InvalidMac
(IRSS.IandA)

the mac given is not the right size/format.

Random Interface



This interface can be used to generate true random numbers (via GetRandom()) or pseudo random number using a seed (via GetPseudoRandom())

«CORBAInterface»
Random
(IRSS.IandA)

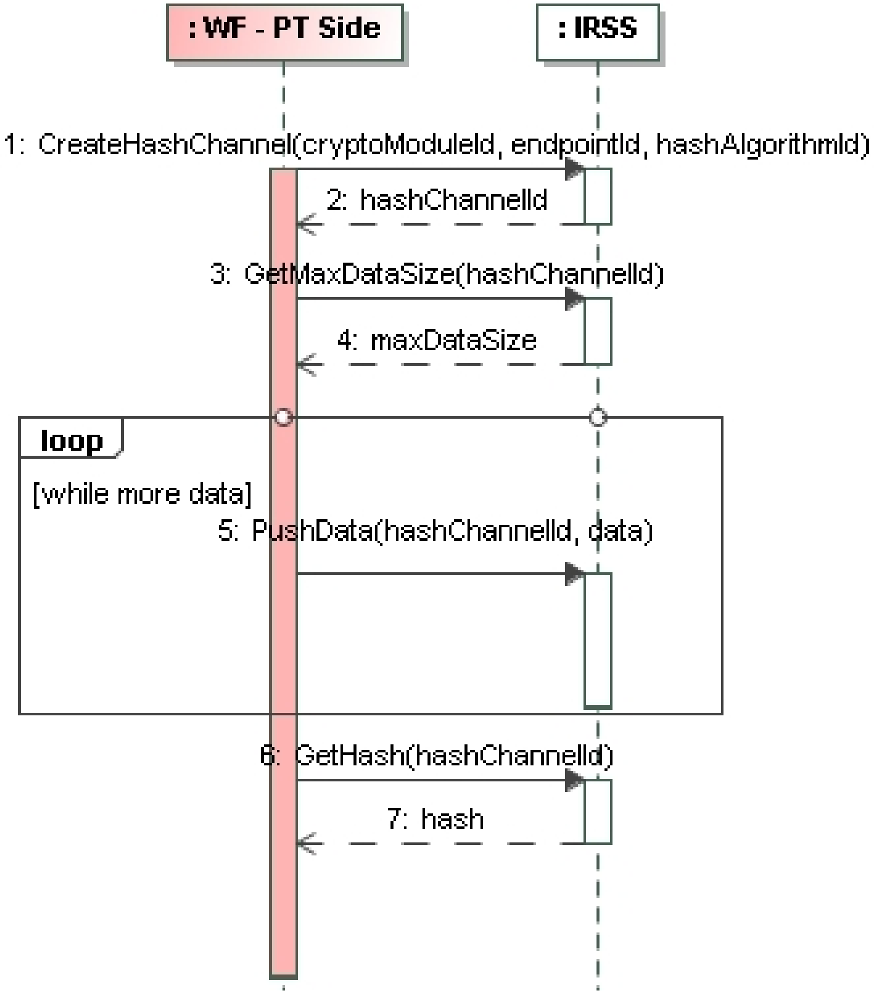
+GetPseudoRandom(seed : unsigned short, numBytes : unsigned short) : OctetSequence
+GetRandom(numBytes : unsigned short) : OctetSequence

«CORBASequence»
OctetSequence
(CF)

IandA Channels - Usage

Via ChannelMgmt interface:
1-2) Create a Hash Channel. This allocates cryptographic resources for the hash function and returns the channel Id to use.

Via HashChannel interface:
3-4) Get the max data size for the channel. Data packets pushed to the channel cannot exceed this max size.
5) Loop to push the data to be hashed to the IRSS using the channel Id returned in step 2.
6-7) When all the data has been pushed, the hash results can be retrieved.



: WF - PT Side
: IRSS
1: CreateHashChannel(cryptoModuleId, endpointId, hashAlgorithmId)
2: hashChannelId
3: GetMaxDataSize(hashChannelId)
4: maxDataSize
loop
[while more data]
5: PushData(hashChannelId, data)
6: GetHash(hashChannelId)
7: hash

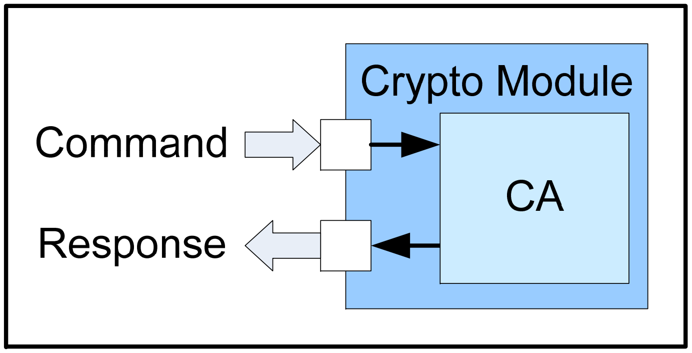
Driving the future of radio communications and systems worldwide

Protocol Module:

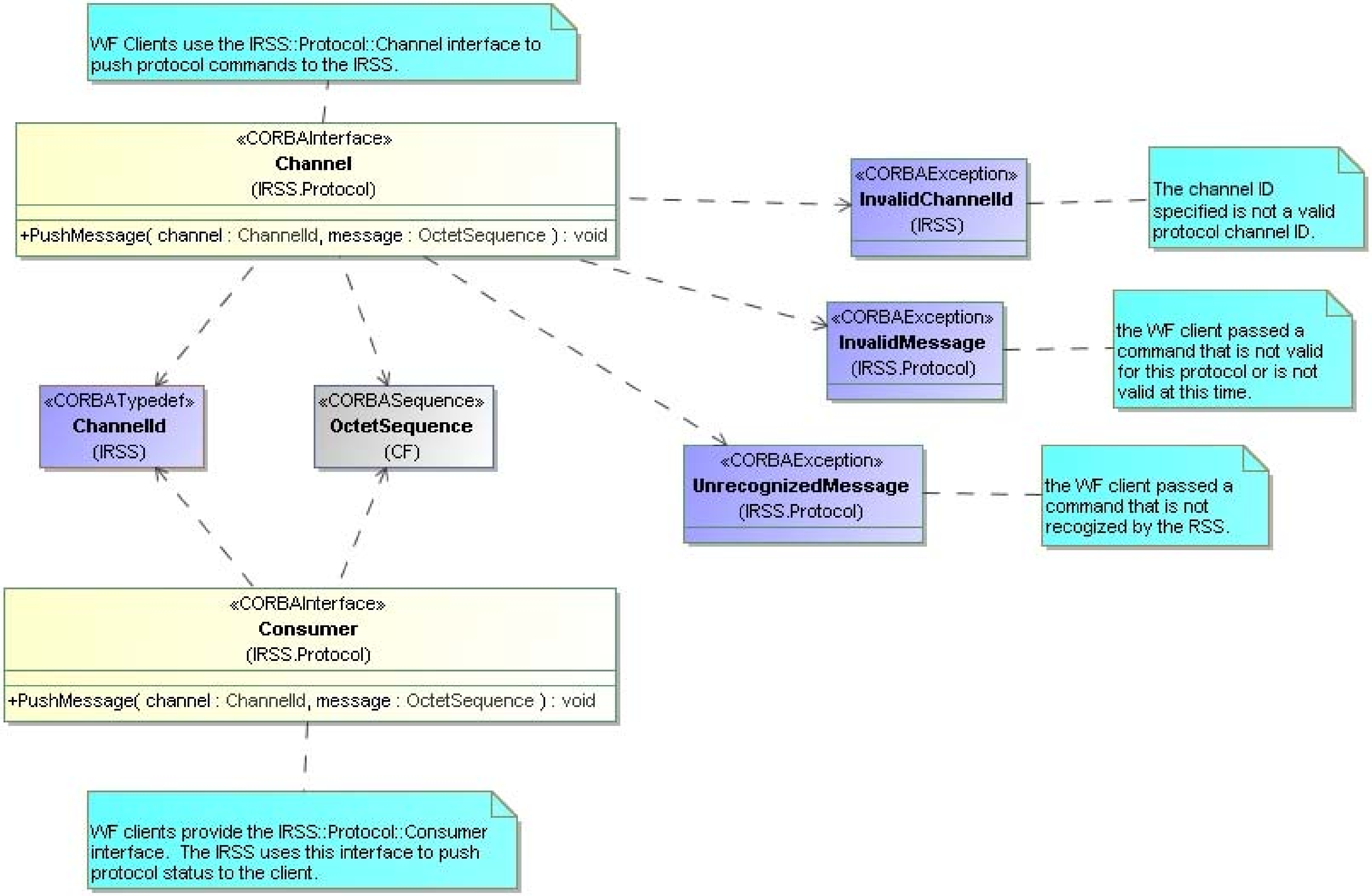
Protocol Interfaces

Protocol Channels

- **Used to exchange protocol messages with a cryptographic application (CA)**
  - Generic messaging API
  - Message definition is protocol dependent
    - Appendices will define the format for a specific protocol
  - Example, used to support an IKE protocol
- **API defines *Channel* and *Consumer* interfaces for both the IRSS and waveform clients, respectively**
  - Clients invoke push operations on the IRSS to send messages to the CA
  - The CA invokes push operations on the clients to send messages to the waveform



Protocol Channels



WF Clients use the IRSS::Protocol::Channel interface to push protocol commands to the IRSS.

«CORBAInterface»
Channel
(IRSS.Protocol)
+PushMessage(channel : ChannelId, message : OctetSequence) : void

«CORBAException»
InvalidChannelId
(IRSS)

The channel ID specified is not a valid protocol channel ID.

«CORBAException»
InvalidMessage
(IRSS.Protocol)

the WF client passed a command that is not valid for this protocol or is not valid at this time.

«CORBATypedef»
ChannelId
(IRSS)

«CORBASequence»
OctetSequence
(CF)

«CORBAException»
UnrecognizedMessage
(IRSS.Protocol)

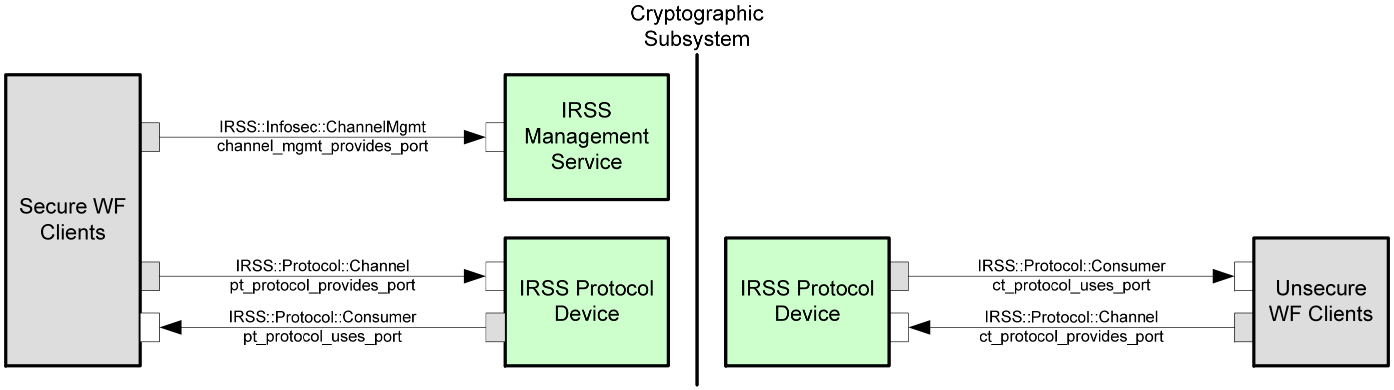
the WF client passed a command that is not recognized by the RSS.

«CORBAInterface»
Consumer
(IRSS.Protocol)
+PushMessage(channel : ChannelId, message : OctetSequence) : void

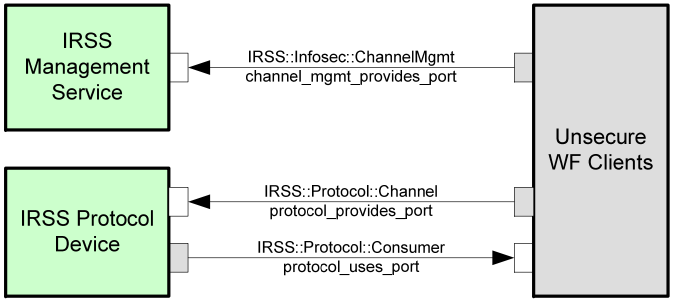
WF clients provide the IRSS::Protocol::Consumer interface. The IRSS uses this interface to push protocol status to the client.

Protocol Channels

Two-Sided Port Diagram



One-Sided Port Diagram



Protocol Channels - Usage

Via the ChannelMgmt interface:
1-2) Create a protocol channel. This initializes the protocol for that channel using the specified protocol ID and returns a channel ID to use.

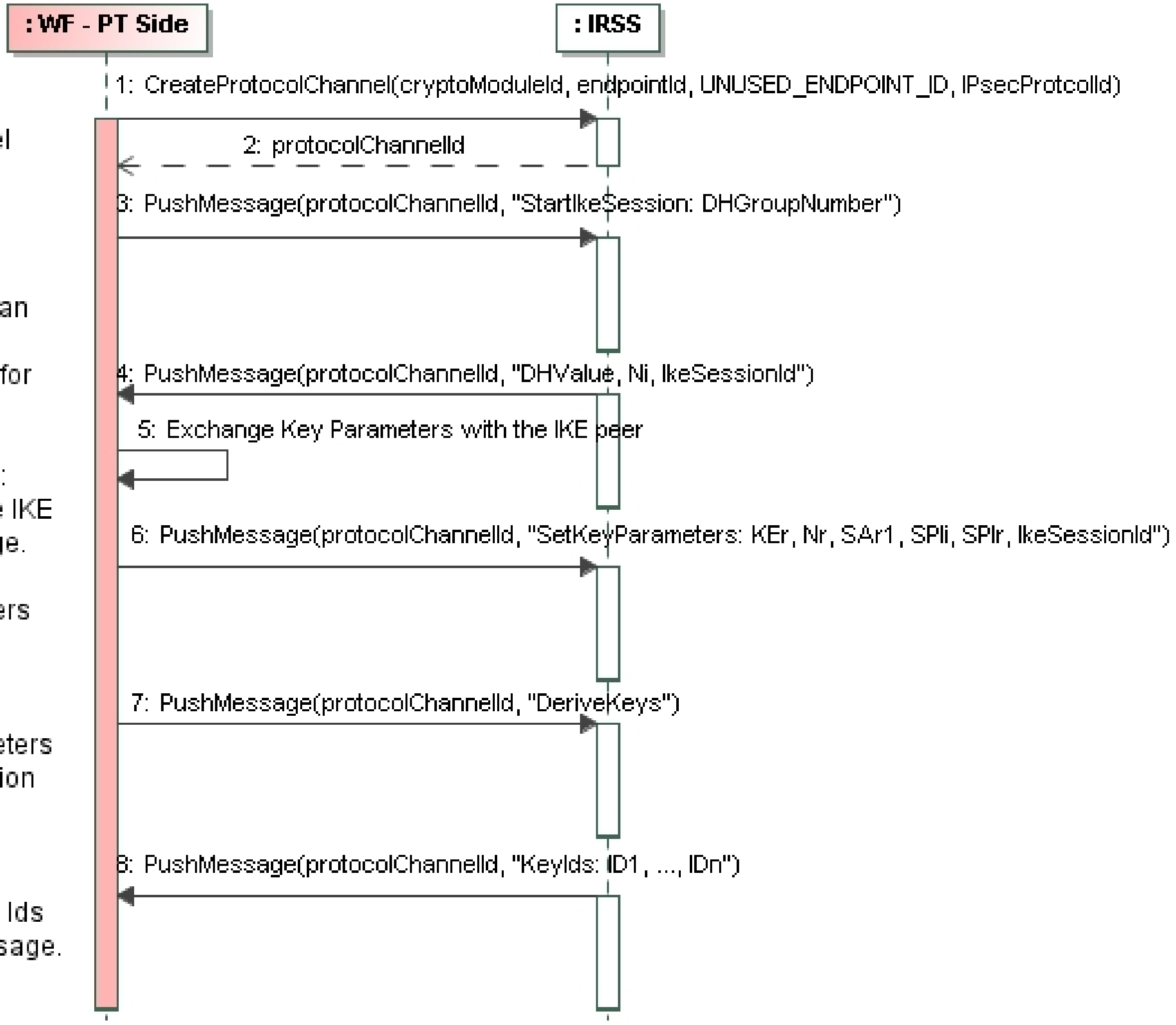
Via the Protocol::Channel interface:
3) Send a protocol message to start an IKE session. The client specifies the Diffie-Hellman group number to use for the session.

Via the Protocol::Consumer interface:
4)The IRSS pushes the results of the IKE initiation in a protocol status message.

5) the client exchanges key parameters with its remote IKE peer.

Via the Protocol::Channel interface:
6-7) The client sends the key parameters to the IRSS and requests the deriviation of keys.

Via the Protocol Consumer interface:
8) The IRSS returns the resulting key Ids to the client in a protocol status message.



: WF - PT Side

: IRSS

1: CreateProtocolChannel(cryptoModuleId, endpointId, UNUSED_ENDPOINT_ID, IPsecProtcolId)

2: protocolChannelId

3: PushMessage(protocolChannelId, "StartIkeSession: DHGroupNumber")

4: PushMessage(protocolChannelId, "DHValue, Ni, IkeSessionId")

5: Exchange Key Parameters with the IKE peer

6: PushMessage(protocolChannelId, "SetKeyParameters: KEr, Nr, SAr1, SPIi, SPIr, IkeSessionId")

7: PushMessage(protocolChannelId, "DeriveKeys")

8: PushMessage(protocolChannelId, "KeyIds: ID1, ..., IDn")

WIRELESS INNOVATION FORUM Driving the future of radio communications and systems worldwide

SDR forum version 2.0

Questions?