Transport API C Edition V3.0.2

VALUE ADDED COMPONENTS

DEVELOPERS GUIDE C EDITION

Document Version: 3.0.2
Date of issue: 30 April 2016
Document ID: ETAC302UMVAC.160

THOMSON REUTERS

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 About this Manual

This document is authored by Transport API architects and programmers who encountered and resolved many of issues the reader might face. Several of its authors have designed, developed, and maintained the Transport API product and other Thomson Reuters products which leverage it. As such, this document is concise and addresses realistic scenarios and use cases.

This guide documents the functionality and capabilities of the Transport API C Edition Value Added Components. In addition to connecting to itself, the Transport API can also connect to and leverage many different Thomson Reuters and customer components. If you want the Transport API to interact with other components, consult that specific component's documentation to determine the best way to configure and interact with these other devices.

1.2 Audience

This manual provides information and examples that aid programmers using the Transport API C Edition Value Added Components. The level of material covered assumes that the reader is a user or a member of the programming staff involved in the design, coding, and test phases for applications which will use the Transport API or its Value Added Components. It is assumed that the reader is familiar with the data types, classes, operational characteristics, and user requirements of real-time data delivery networks, and has experience developing products using the C programming language in a networked environment. Although Transport API Value Added Components offer alternate entry points to Transport API functionality, it is recommended that users are familiar with general Transport API usage and interfaces.

1.3 Programming Language

The Transport API Value added Components are is written to both the C and Java languages. This guide discusses concepts related to the C Edition. All code samples in this document, value added component source, and all example applications provided with the product are written accordingly.

1.4 Acronyms and Abbreviations

ACRONYM	MEANING
ADH	Advanced Data Hub
ADS	Advanced Distribution Server
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ATS	Advanced Transformation System
DACS	Data Access Control System
DMM	Domain Message Model
EDF	Elektron Data Feeds

Table 1: Acronyms and Abbreviations

ACRONYM	MEANING
EED	Elektron Edge Device
EMA	Elektron Message API, referred to simply as the Message API
EOA	Elektron Object API, referred to simply as the Object API.
ETA	Elektron Transport API, referred to simply as the Transport API
EWA	Elektron Web API
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol (Secure)
IDN	Integrated Data Network
NIP	Non-Interactive Provider
OMM	Open Message Model
QoS	Quality of Service
RDF Direct	Reuters Data Feed Direct
RDM	Reuters Domain Model
RFA	Robust Foundation API
RMTES	Reuters Multi-Lingual Text Encoding Standard
RSSL	Reuters Source Sink Library
RWF	Reuters Wire Format
SOA	Service Oriented Architecture
SSL	Source Sink Library
TREP	Thomson Reuters Enterprise Platform
UML	Unified Modeling Language
UTF-8	8-bit Unicode Transformation Format

Table 1: Acronyms and Abbreviations

1.5 References

1. *Transport API C Edition RDM Usage Guide*
2. *API Concepts Guide*
3. Transport API *ANSI Library Reference Manual*
4. Transport API *DACS LOCK Library Reference Manuals*
5. *Transport API C Edition Value Added Components Developers Guide*
6. *Reuters Multilingual Text Encoding Standard Specification*
7. *Transport API Java C Edition Developers Guide*

1.6 Documentation Feedback

While we make every effort to ensure the documentation is accurate and up-to-date, if you notice any errors, or would like to see more details on a particular topic, you have the following options:

- Send us your comments via email at apidocumentation@thomsonreuters.com.
- Add your comments to the PDF using Adobe's **Comment** feature. After adding your comments, submit the entire PDF to Thomson Reuters by clicking **Send File** in the **File** menu. Use the apidocumentation@thomsonreuters.com address.

1.7 Document Conventions

This document uses the following types of conventions:

- Typographic
- Diagrams
- Document Structure

1.7.1 Typographic

- Structures, methods, in-line code snippets, and types are shown in `orange, Courier New` font.
- Parameters, filenames, tools, utilities, and directories are shown in **Bold** font.
- Document titles and variable values are shown in *italics*.
- When initially introduced, concepts are shown in ***Bold, Italics***.
- Longer code examples are shown in Courier New font against an orange background. For example:

```
/* decode contents into the filter list structure */
if ((retVal = rsslDecodeFilterList(&decIter, &filterList)) >= RSSL_RET_SUCCESS)
{
    /* create single filter entry and reuse while decoding each entry */
    RsslFilterEntry filterEntry = RSSL_INIT_FILTER_ENTRY;
```

1.7.2 Document Structure

- General Concepts
- Detailed Concepts
- Interface Definitions
- Example Code

1.7.3 Diagrams

Diagrams that depict the interaction between components on a network use the following notation:

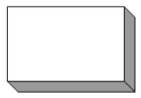
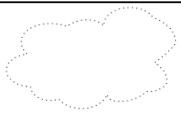
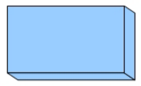
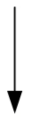
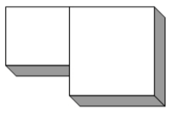
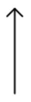
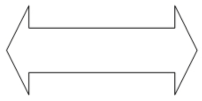
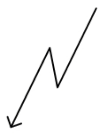
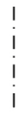
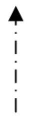
	Feed Handler, Enterprise Platform server, or other application		Network of multiple servers
	Transport API application		Point-to-point connection showing direction of primary data flow
	Application with local daemon		Point-to-point connection showing direction of client connecting to server
	Multicast network		Data from external source (e.g. consolidated network or exchange)
	Connection to Multicast network, no primary data flow direction		Connection to Multicast network showing direction of primary data flow

Figure 1. Network Diagram Notation

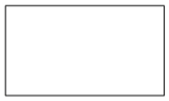
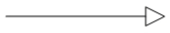
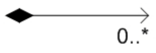
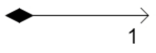
	Object
	Inheritance: object on left is like object on right
	Composition: object on left is made up of some number of objects on right
	Composition: object on left is made up of one object on right

Figure 2. UML Diagram Notation

1.8 What's New in this Document

For a list of 8.0 changes made to this document, refer to Appendix C. For changes made to the Transport API in previous versions, refer to the last 7.X version release Developer's Guide.

Chapter 2 Product Description and Overview

2.1 What is the Transport API?

The Transport API is a low-level Transport API that provides the most flexible development environment to the application developer. It is the foundation on which all Thomson Reuters OMM-based components are built. The Transport API allows applications to achieve the highest throughput and lowest latency available with any OMM API, but requires applications to perform all message encoding/decoding and manage all aspects of network connectivity. The Transport API, Elektron Message API, and the Robust Foundation API (RFA) make up the set of OMM API offerings.

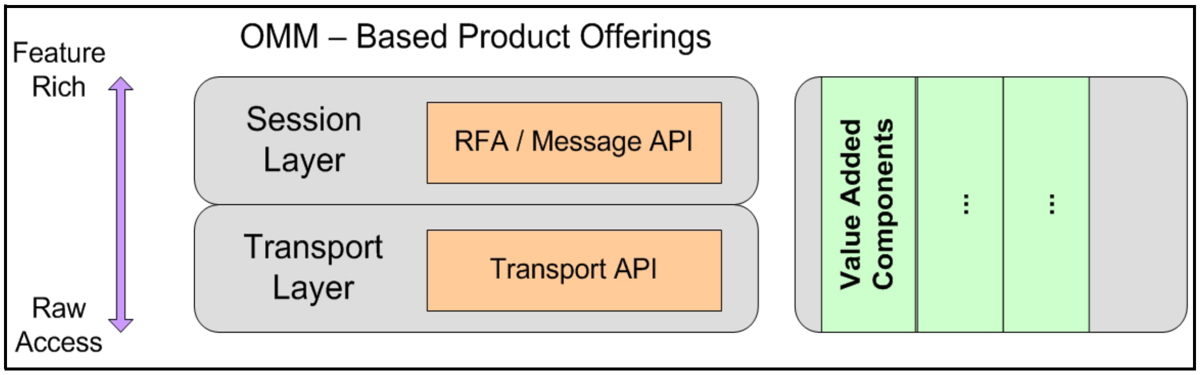


Figure 3. OMM APIs with Value Added Components

The Transport API Value Added Components provide alternate entry points for applications to leverage OMM-Based APIs with more ease and simplicity. These optional components help to offload much of the connection management code and perform encoding and decoding of some key OMM domain representations. Unlike older domain-based APIs that lock the user into capabilities or ease-of-use into the highest layer of API, Value Added components are independently implemented for use with the Transport API and RFA in their native languages (Example: Transport API in C and Java, RFA in C++ and Java). These implementations are then shipped with their respective API products, as options for the application developer that may want these additional capabilities.

This release of the Transport API includes the Value Added Components, an overview of which is given in this chapter. Future Transport API releases may provide additional Value Added Components and domain model representations.

2.2 What are Transport API Value Added Components?

The Value Added Components simplify and compliment the use of the Transport API. These components (depicted in green in the figure below) are offered along side of the Transport API in order to maximize the user experience and allow for more intuitive and straight forward, rapid creation of Transport API applications. Applications can write directly to the Transport API interfaces or commingle some or all Value Added Components. The choice to leverage these components is up to the application developer; Value Added Component use is not required to use the Transport API. By using Transport API Value Added Components, it allows the application to choose and customize the balance between ultra high performance raw access and ease of use feature functionality. Value Added Components are written to the Transport API interfaces and are designed to work alongside the Transport API. Their interfaces have a similar look and feel to Transport API interfaces to provide simple migration and consistent use between all components and the Transport API.

All value added components provide fully supported library and header files ready to build into new or existing Transport API applications. Examples and documentation are provided to show the full power and capability of the component.

Some value added components provide buildable source code[1] to allow for customization and modification to suit specific user needs. This source code serves the following purposes:

- Clients may want to provide their own implementation of the component in a slightly different way than most clients. Rather than starting from scratch, clients can modify the component (as their own code) to jump start their development efforts. This code is then theirs to support and maintain.
- Clients may want to aid in troubleshooting or suggest improvements to the component to benefit everyone.
- Clients might want to build a new component that has similar behaviors to an existing component. Clients can leverage the code of one component to jump start their development efforts.

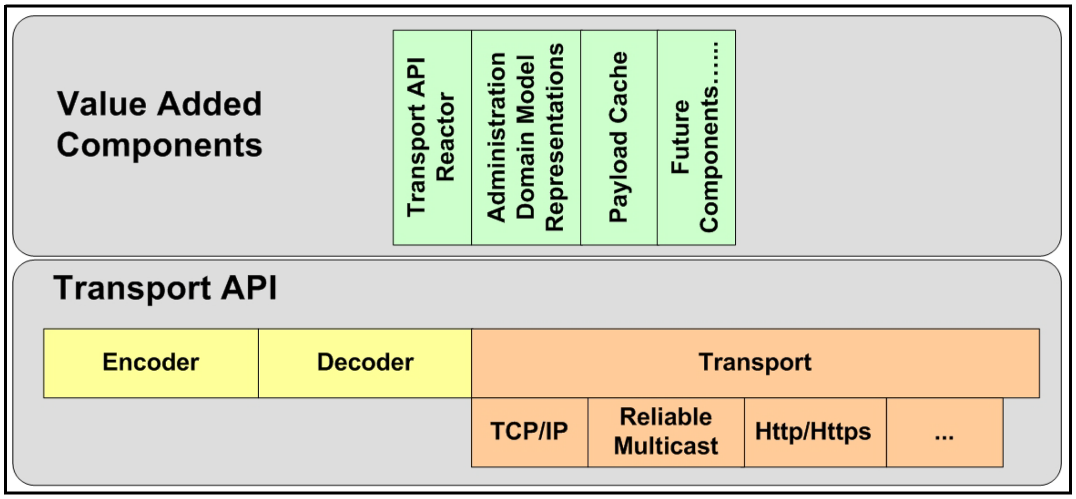


Figure 4. Transport API Value Added Components

2.3 Transport API Reactor

The ***Transport API Reactor*** is a connection management and event processing component that can significantly reduce the amount of code an application must write to leverage OMM in its own functions and to connect to other OMM-based devices. Consumer, interactive provider, and non-interactive provider applications can use the reactor and leverage it in managing consumer and non-interactive provider start-up processes, including user log in, source directory establishment, and dictionary download. The reactor also supports dispatching of events to user-implemented callback functions. In addition, it handles flushing of user written content and manages network pings on the user's behalf. The connection recovery feature allows the Value Added Reactor to automatically recover from disconnects. Value Added domain representations are coupled with the reactor, allowing domain specific callbacks to be presented with their respective domain representation for easier, more logical access to content. For more information, refer to Chapter 6, Reactor Detailed View. This component depends on the Value Added Administration Domain Model Representation component, the Value Added Utilities, Transport API Reliable Transport Package, Transport API Message Package, and Transport API Data Package.

To access all Transport API reactor functionality, including the Administration Domain Model Representations, an application must include **rsslReactor.h**.

2.4 OMM Consumer Watchlist

The `RsslReactor` features a per-channel watchlist that provides a wealth of functionality for OMM Consumer applications. The watchlist automatically performs various recovery behaviors for which developers would normally need to account.

The watchlist supports consuming from TCP-based connections (**RSSL_CONN_TYPE_SOCKET**) and multicast networks (**RSSL_CONN_TYPE_RELIABLE_MULTICAST**). The reactor uses the watchlist to provide the same interaction model for both TCP and Multicast communications, so that application developers need not write code specific to either system.

1. Thomson Reuters fully supports the use of its pre-built library and header files. Provided source code can help with user troubleshooting and debugging. However, the user, not Thomson Reuters, is responsible for supporting any modifications to the provided source.

For details on configuring the `RsslReactor` to enable the consumer watchlist, refer to Section 6.3.2.

2.4.1 Data Stream Aggregation and Recovery

The watchlist automatically recovers data streams in response to failure conditions, such as disconnects and unavailable services, so that applications do not need special handling for these conditions. As conditions are resolved, the watchlist will re-request items on the application's behalf. Applications can also use this function to request data before a connection is fully established.

To use watchlist recovery from disconnects, enable the Reactor's connection recovery. Options to reconnect disconnected channels are detailed in Section 6.4.1.2.

For efficient bandwidth usage, the watchlist also combines multiple requests for the same item into a single stream and forwards response messages to each requested stream as appropriate.

2.4.2 ADS Multicast Consumption

The watchlist can request and consume data from an ADS configured to provide data over a multicast network.

When multicasting data, the ADS provides data through two paths: a broadcast network that sends messages intended for multiple consumers (such as item updates), and a unicast channel, for messages directed at a particular consumer (such as refreshes to satisfy an item request). The watchlist synchronizes messages delivered over these paths with each other and provides them to streams opened by the application.

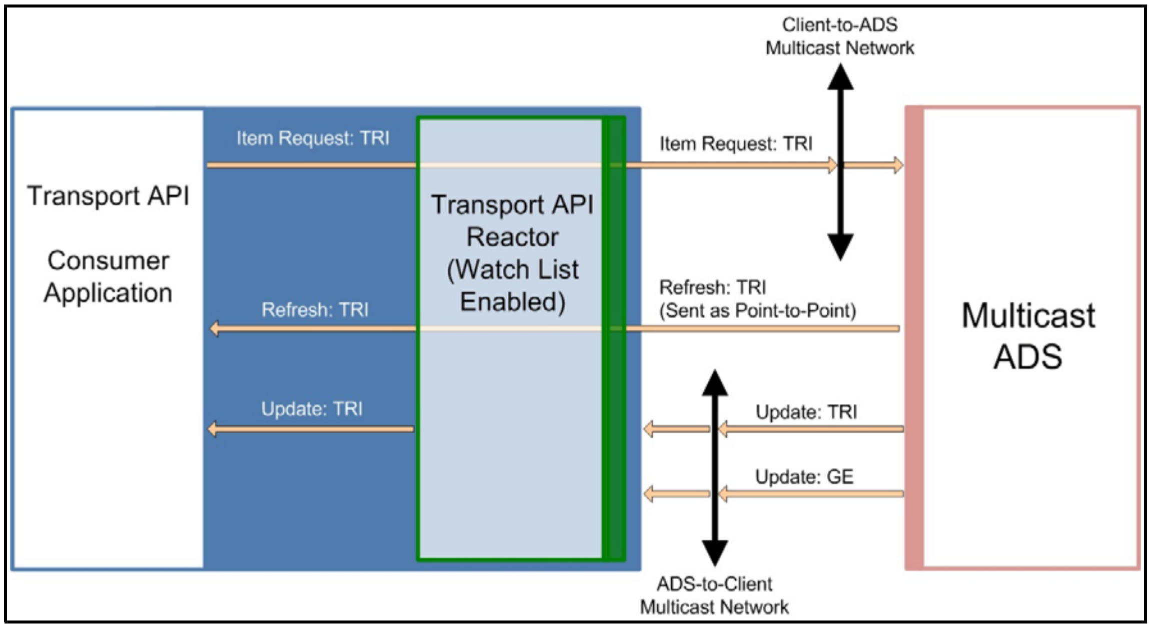


Figure 5. Consuming Multicast Data with the Transport API Reactor

The watchlist also provides additional recovery for lost data by:

- Periodically re-requesting unanswered requests.
- Detecting and recovering from gaps in sequenced data streams.
- Identifying disconnects with the ADS and recovering streams provided by the ADS.

For notes on configuring multicast, refer to Section 2.4.5.

2.4.3 Additional Features

The watchlist provides additional features for convenience:

- Group and Service Status Fanout: The `RsslReactor` maintains a directory stream to receive service updates. As group status messages or service status messages are received, the `RsslReactor` forwards the status to all affected streams via `RsslStatusMsgs`.
- QoS Range Matching: The `RsslReactor` will accept and aggregate item requests that specify a range of `RsslQos`, or requests that do not specify an `RsslQos`. After comparing these requests with the QoS from the providing service, the watchlist uses the best matching QoS.
- Support for Enhanced Symbol List Behaviors: The `RsslReactor` supports data streams when requesting a Symbol List item. For details on requesting Symbol list data streams, refer to the *Transport API C Edition RDM Usage Guide*.
- Support for Batch Requests: The `RsslReactor` will accept batch requests regardless of whether the connected provider supports them.

2.4.4 Usage Notes

Applications should note the following when enabling the watchlist:

- The application must use the `rsslReactorSubmitMsg` function to send messages. It cannot use `rsslReactorSubmit`.
- Only one login stream should be opened per `RsslReactorChannel`.
- To prevent unnecessary bandwidth use, the watchlist will not recover a dictionary request after a complete refresh is received.
- As private streams are intended for content delivery between two specific points, the watchlist does not aggregate nor recover them.
- The `RsslReactorOMMConsumerRole.dictionaryDownloadMode` option is not supported when the watchlist is enabled.

2.4.5 Configuring Multicast Connections

The watchlist supports consuming traffic only from a segmented network, where the ADS multicast network is separated from the consumer's (as illustrated in Figure 5). When configuring the connection, specify both `RsslReactorConnectOpts.rsslConnectOptions.sendAddress` and `recvAddress`.

ADS multicast provides a hash with many messages so that consumers can filter unwanted content. To improve performance, consumer applications that share networks with other consumers might want to enable this filtering. Filtering can be enabled by setting the **RSSL_MCAST_FILTERING_ON** flag in the `RsslReactorConnectOpts.rsslConnectOptions.multicastOpts`. The watchlist automatically handles filter registration for relevant data.

ADSs may configure multiple multicast networks to balance the traffic load. You can configure `RsslReactorChannels` to receive from multiple multicast addresses (by specifying a comma-separated list of addresses to the `recvAddress`).

For information on creating consumer connections, refer to Section 6.4.1.1 and `RsslConnectOptions` in the *Transport API C Edition Developers Guide*. Section 6.4.1.5 provides code that configures a connection to consume multicast from an ADS.

2.5 Administration Domain Model Representations

The ***Administration Domain Model Representations*** are RDM specific representations of the OMM administrative domain models. This Value Added Component contains structures that represent the messages within the Login, Source Directory, and Dictionary domains. All structures follow the formatting and naming specified in the *Transport API C Edition RDM Usage Guide*, so access to content is logical and specific to the content being represented. This component also handles all encoding and decoding functionality for these domain models, so the application needs only to manipulate the message's structure members to send or receive this content. This not only significantly reduces the amount of code an application needs to interact with OMM devices (i.e., TREP), but also ensures that encoding/decoding for these domain models follow OMM specified formatting rules. Applications can use this Value Added Component directly to help with encoding, decoding, and representation of these domain models. When using the Transport API Reactor, this component is embedded to manage and present callbacks with a domain specific representation of content. For more information, refer to Chapter 7, Administration

Domain Models Detailed View. This component depends on the Value Added Utilities, Transport API Message Package, and Transport API Data Package.

To access all data package functionality, an application must include **rsslRDMMsg.h**.

2.6 Value Added Utilities

The Value Added Utilities are a collection of helper constructs, mainly used by the Transport API Reactor. Included is a multi-purpose memory buffer type that can help with flexible, reusable memory - this is leveraged by the Administration Domain Model Representations when encoding or decoding messages. Other Value Added Utilities include a simple queue, mutex locks, thread helper functionality, and a simple event alerting component.

2.7 Value Added Cache

Any application type (consumer, interactive provider, or non-interactive provider) can leverage the OMM payload cache feature. Using payload cache, an application can maintain a local store of the OMM container data it consumes, publishes, or transforms. The cache maintains the latest values of the OMM data entries: container values update to reflect the most recent refresh and update message payloads whenever the application receives them. Data is retrieved from the cache entry in the form of an encoded OMM container. The cache does not depend on other Value Added components, and only requires the Transport API Data Package and Transport API Message Package. Only library and API header files are available for the cache component.

Chapter 3 Building an OMM Consumer

3.1 Overview

This chapter provides an overview of how to create an OMM Consumer application using the Transport API Reactor and Administration Domain Model Representation Value Added Components. The Value Added Components simplify the work done by an OMM Consumer application when establishing a connection to other OMM Interactive Provider applications, including the Enterprise Platform, Data Feed Direct, and Elektron. After the Reactor indicates that the connection is ready, an OMM Consumer can then consume (i.e., send data requests and receive responses) and publish data (i.e., post data).

The general process can be summarized by the following steps.

- Leverage Existing or Create New `RsslReactor`
- Implement Callbacks and Populate Role
- Establish connection using `rsslReactorConnect`
- Issue Requests and/or Post information
- Log out and shut down

The **rsslVAConsumer** example application, included with the Transport API product, provides one implementation of an OMM Consumer application that uses the Transport API Value Added Components. The application is written with simplicity in mind and demonstrates usage of the Transport API and Transport API Value Added Components. Portions of functionality have been abstracted and can easily be reused, though you might need to modify it to achieve your own unique performance and functionality goals.

3.2 Leverage Existing or Create New RsslReactor

The `RsslReactor` can manage one or multiple `RsslReactorChannel` structures. This allows the application to choose to associate OMM Consumer connections with an existing `RsslReactor`, having it manage more than one connection, or to create a new `RsslReactor` to use with the connection.

If the application is creating a new `RsslReactor`, the `rsslCreateReactor` function is used. This will create any necessary memory and threads that the `RsslReactor` uses to manage `RsslReactorChannel`s and their content flow. If the application is using an existing `RsslReactor`, there is nothing additional to do.

Detailed information about the `RsslReactor` and its creation are available in Section 6.2.1.

3.3 Implement Callbacks and Populate Role

Before creating the OMM Consumer connection, the application needs to specify callback functions to use for all inbound content. The callback functions are specified on a per `RsslReactorChannel` basis so each channel can have its own unique callback functions or existing callback functions can be specified and shared across multiple `RsslReactorChannel`s.

Several of the callback functions are required for use with an `RsslReactor`. The application must have an

- `RsslReactorChannelEventCallback`, which returns information about the `RsslReactorChannel` and its state (e.g., connection up)
- `RsslDefaultMsgCallback`, which processes all data not handled by other optional callbacks.

In addition to the required callbacks, an OMM Consumer can specify several administrative domain specific callback functions. The available domain specific callbacks are

- `RsslRDMLoginMsgCallback`, which processes all data for the RDM Login domain.
- `RsslRDMDirectoryMsgCallback`, which processes all data for the RDM Source Directory domain.
- `RsslRDMDictionaryMsgCallback`, which processes all data for the RDM Dictionary domain.

The `RsslReactorOMMConsumerRole` structure should be populated with all callback information for the `RsslReactorChannel`.

The `RsslReactorOMMConsumerRole` allows the application to provide login, directory and dictionary request information. This can be initialized with default information. The callback functions are specified on the `RsslReactorOMMConsumerRole` structure or with specific information according to the application and user. The `RsslReactor` will use this information when starting up the `RsslReactorChannel`.

Detailed information about the `RsslReactorOMMConsumerRole` is in Section 6.3.1. Information about the various callback functions and their specifications are available in Section 6.5.2.

3.4 Establish Connection using rsslReactorConnect

Once the `RsslReactorOMMConsumerRole` is populated, the application can use `rsslReactorConnect` to create a new outbound connection. `rsslReactorConnect` will create an OMM Consumer type connection using the provided configuration and role information.

Once the underlying connection is established a channel event will be returned to the application's `RsslReactorChannelEventCallback`; this will provide the `RsslReactorChannel` and to indicate the current connection state. At this point, the application can begin using the `rsslReactorDispatch` function to dispatch directly on this `RsslReactorChannel`, or continue using `rsslReactorDispatch` to dispatch across all channels associated with the `RsslReactor`.

The `RsslReactor` will use the login, directory, and dictionary information specified on the `RsslReactorOMMConsumerRole` to perform all channel initialization for the user. After the user is logged in, has received a source directory response, and downloaded field dictionaries, a channel event is returned to inform the application that the connection is ready.

The `rsslReactorConnect` function is described in Section 6.4.1.1. Dispatching is described in Section 6.5.

3.5 Issue Requests and/or Post Information

After the `RsslReactorChannel` is established, it can be used to request additional content. When issuing the request, the consuming application can use the `serviceId` of the desired service, along with the stream's identifying information. Requests can be sent for any domain using the formats defined in that domain model specification. Domains provided by Thomson Reuters are defined in the *Transport API C Edition RDM Usage Guide*. This content will be returned to the application via the `RsslDefaultMsgCallback`.

At this point, an OMM Consumer application can also post information to capable provider applications. All content requested, received, or posted is encoded and decoded using the Transport API Message Package and the Transport API Data Package described in the *Transport API C Edition Developers Guide*.

3.6 Log Out and Shut Down

When the Consumer application is done retrieving or posting content, it can close the `RsslReactorChannel` by calling `rsslReactorCloseChannel`. This will close all item streams and log out the user. Prior to closing the `RsslReactorChannel`, the application should release any unwritten pool buffers to ensure proper memory cleanup.

If the application is done with the `RsslReactor`, the `rsslDestroyReactor` function can be used to shutdown and cleanup any `RsslReactor` resources.

- Closing an `RsslReactorChannel` is described in Section 6.4.2.

- Shutting down an `RsslReactor` is described in Section 6.2.2.

3.7 Additional Consumer Details

The following locations provide specific details about using OMM Consumers, the Transport API, and Transport API Value Added Components:

- The **rsslVAConsumer** application demonstrates one way of implementing of an OMM Consumer application that uses the Transport API Value Added Components. The application's source code and ReadMe file contain additional information about specific implementation and behaviors.
- Chapter 6 provides a detailed look at the Transport API Reactor.
- Chapter 7 provides more information about the Administration Domain Model Representations.
- The *Transport API C Edition Developers Guide* provides specific Transport API encoder/decoder and transport usage information.
- The *Transport API C Edition RDM Usage Guide* provides specific information about the DMMs used by this application type.

Chapter 4 Building an OMM Interactive Provider

4.1 Overview

This chapter provides a high-level description of how to create an OMM Interactive Provider application using the Transport API Reactor and Administration Domain Model Representation Value Added Components. An OMM Interactive Provider application opens a listening socket on a well-known port allowing OMM Consumer applications to connect. The Transport API Value Added Components simplify the work done by an OMM Interactive Provider application when accepting connections and handling requests from OMM Consumers.

The following steps summarize this process:

- Leverage Existing or Create New `RsslReactor`
- Create an `RsslServer`
- Implement Callbacks and Populate Role
- Associate incoming connections using `rsslReactorAccept`
- Perform Login Process
- Provide Source Directory Information
- Provide Necessary Dictionaries
- Handle Requests and Post messages
- Disconnect Consumers and shut down

Included with the Transport API package, the **rsslVAProvider** example application provides one way of implementing an OMM Interactive Provider application that uses the Transport API Value Added Components. The application is written with simplicity in mind and demonstrates the use of the Transport API and Transport API Value Added Components. Portions of the functionality are abstracted for easy reuse, though you might need to customize it to achieve your own unique performance and functionality goals.

4.2 Leverage Existing or Create New RsslReactor

The `RsslReactor` can manage one or multiple `RsslReactorChannel` structures. This allows the application to choose to associate OMM Provider connections with an existing `RsslReactor`, having it manage more than one connection, or to create a new `RsslReactor` to use with the connection.

If the application is creating a new `RsslReactor`, the `rsslCreateReactor` function is used. This will create any necessary memory and threads that the `RsslReactor` uses to manage `RsslReactorChannel`s and their content flow. If the application is using an existing `RsslReactor`, there is nothing additional to do.

Detailed information about the `RsslReactor` and its creation are available in Section 6.2.1.

4.3 Create an RsslServer

The first step of any Transport API Interactive Provider application is to establish a listening socket, usually on a well-known port so that consumer applications can easily connect. The provider uses the `rsslBind` function to open the port and listen for incoming connection attempts. This uses the standard Transport API Transport functionality described in the *Transport API C Edition Developers Guide*.

Whenever an OMM consumer application attempts to connect, the provider will use the **RsslServer** and associate the incoming connections with an **RsslReactor**, which will accept the connection and perform any initialization. This process is described in the following sections.

4.4 Implement Callbacks and Populate Role

Before accepting an incoming connection with the OMM Provider, the application needs to specify callback functions to use for all inbound content. The callback functions are specified on a per **RsslReactorChannel** basis so each channel can have its own unique callback functions or existing callback functions can be specified and shared across multiple **RsslReactorChannel**s.

Several of the callback functions are required for use with an **RsslReactor**. The application must have an

- **RsslReactorChannelEventCallback**, which returns information about the **RsslReactorChannel** and its state (e.g., connection up)
- **RsslDefaultMsgCallback**, which processes all data not handled by other optional callbacks.
- In addition to the required callbacks, an OMM Provider can specify several administrative domain specific callback functions. The available domain specific callbacks are
- **RsslRDMLoginMsgCallback**, which processes all data for the RDM Login domain.
- **RsslRDMDirectoryMsgCallback**, which processes all data for the RDM Source Directory domain.
- **RsslRDMDictionaryMsgCallback**, which processes all data for the RDM Dictionary domain.

The **RsslReactorOMMProviderRole** structure should be populated with all callback information for the **RsslReactorChannel**.

Detailed information about the **RsslReactorOMMProviderRole** is in Section 6.3.1. Information about the various callback functions and their specifications are available in Section 6.5.2.

4.5 Associate Incoming Connections Using rsslReactorAccept

Once the **RsslReactorOMMProviderRole** is populated, the application can use **rsslReactorAccept** to accept a new inbound connection. **rsslReactorAccept** will accept an OMM Provider connection from the passed in **RsslServer** using the provided configuration and role information.

Once the underlying connection is established a channel event will be returned to the application's **RsslReactorChannelEventCallback**; this will provide the **RsslReactorChannel** and to indicate the current connection state. At this point, the application can begin using the **rsslReactorDispatch** function to dispatch directly on this **RsslReactorChannel**, or continue using **rsslReactorDispatch** to dispatch across all channels associated with the RsslReactor.

The **RsslReactor** will perform all channel initialization and pass any administrative domain information to the application via the callbacks specified with the **RsslReactorOMMProviderRole**.

The **rsslReactorAccept** function is described in Section 6.4.1.6. Dispatching is described in Section 6.5.

4.6 Perform Login Process

Applications authenticate with one another using the Login domain model. An OMM Interactive Provider must handle the consumer's Login request messages and supply appropriate responses. Login information will be provided to the application via the **RsslRDMLoginMsgCallback**, when specified on the **RsslReactorOMMProviderRole**.

After receiving a Login request, the Interactive Provider can perform any necessary authentication and permissioning.

- If the Interactive Provider grants access, it should send an **`RsslRDMLoginRefresh`** to convey that the user successfully connected. This message should indicate the feature set supported by the provider application.
- If the Interactive Provider denies access, it should send an **`RsslRDMLoginStatus`**, closing the connection and informing the user of the reason for denial.

Login messages can be encoded and decoded using the **`RsslRDMLoginMsg`**. More details and code examples are in Section 7.3.

All content requested, received, or posted is encoded and decoded using the Transport API Message Package and the Transport API Data Package described in the *Transport API C Edition Developers Guide*.

Information about the Login domain and expected content formatting is available in the *Transport API C Edition RDM Usage Guide*.

4.7 Provide Source Directory Information

The Source Directory domain model conveys information about all available services in the system. An OMM consumer typically requests a Source Directory to retrieve information about available services and their capabilities. This includes information about supported domain types, the service's state, the QoS, and any item group information associated with the service. Thomson Reuters recommends that at a minimum, an Interactive Provider supply the Info, State, and Group filters for the Source Directory.

- The Source Directory Info filter contains the name and **`serviceId`** for each available service. The Interactive Provider should populate the filter with information specific to the services it provides.
- The Source Directory State filter contains status information for the service informing the consumer whether the service is Up (available), or Down (unavailable).
- The Source Directory Group filter conveys item group status information, including information about group states, as well as the merging of groups. If a provider determines that a group of items is no longer available, it can convey this information by sending either individual item status messages (for each affected stream) or a Directory message containing the item group status information. Additional information about item groups is available in the *Transport API C Edition Developers Guide*.

Source Directory messages can be encoded and decoded using the **`RsslRDMDirectoryMsg`**. More details and code examples are in Section 7.4.

All content requested, received, or posted is encoded and decoded using the Transport API Message Package and the Transport API Data Package described in the *Transport API C Edition Developers Guide*.

Information about the Source Directory domain and expected content formatting is available in the *Transport API C Edition RDM Usage Guide*.

4.8 Provide or Dowload Necessary Dictionaries

Some data requires the use of a dictionary for encoding or decoding. The dictionary typically defines type and formatting information, and tells the application how to encode or decode information. Content that uses the **`RsslFieldList`** type requires the use of a field dictionary (usually the Thomson Reuters **RDMFieldDictionary**, though it can instead be a user-defined or modified field dictionary).

The Source Directory message should notify the consumer about dictionaries needed to decode content sent by the provider. If the consumer needs a dictionary to decode content, it is ideal that the Interactive Provider application also make this dictionary available to consumers for download. The provider can inform the consumer whether the dictionary is available via the Source Directory.

If connected to a supporting ADH, a provider application can also download the RWFFld and RWFEnum dictionaries to retrieve appropriate dictionary information for providing field list content. A provider can use this feature to ensure they are using the appropriate version of the dictionary or to encode data. An ADH supporting the Provider Dictionary Download feature sends a Login request message containing the **SupportProviderDictionaryDownload** login element. For details on using the Login domain and expected message content, refer to the *Transport API C Edition RDM Usage Guide*. The dictionary request is sent using the Dictionary domain model[1].

Dictionary messages can be encoded and decoded using the `RsslRDMDictionaryMsg`. More details and code examples are in Section 7.5. Dictionary requests will be provided via the `RsslRDMDictionaryMsgCallback`, when specified on the `RsslReactorOMMProviderRole`.

Whether loading a dictionary from file or requesting it from an ADH, the Transport API offers several utility functions for loading, downloading, and managing a properly-formatted field dictionary. There are also utility functions provided to help the provider encode into an appropriate format for downloading or decoding downloaded dictionaries. The *Transport API C Edition RDM Usage Guide* describes available dictionary utility functions.

Dictionary messages can be encoded and decoded using the `RsslRDMDictionaryMsg`. More details and code examples are in Section 7.5.

All content requested, received, or posted is encoded and decoded using the Transport API Message Package and the Transport API Data Package described in the *Transport API C Edition Developers Guide*. Information about the Dictionary domain and expected content formatting is available in the *Transport API C Edition RDM Usage Guide*.

4.9 Handle Requests and Post Messages

A provider can receive a request for any domain, though this should typically be limited to the domain capabilities indicated in the Source Directory. When a request is received, the provider application must determine if it can satisfy the request by:

- Comparing `msgKey` identification information
- Determining whether it can provide the requested QoS
- Ensuring that the consumer does not already have a stream open for the requested information

If a provider can service a request, it should send appropriate responses. However, if the provider cannot satisfy the request, the provider should send an `RsslStatusMsg` to indicate the reason and close the stream. All requests and responses should follow specific formatting as defined in the domain model specification. The *Transport API C Edition RDM Usage Guide* defines all domains provided by Thomson Reuters. This content will be returned to the application via the `RsslDefaultMsgCallback`.

The provider can specify that it supports post messages via the `RsslRDMLoginRefresh`. If a provider application receives a Post message, the provider should determine the correct handling for the post. This depends on the application's role in the system and might involve storing the post in its cache or passing it farther up into the system. If the provider is the destination for the Post, the provider should send any requested acknowledgments, following the guidelines described in the *Transport API C Edition Developers Guide*. Any posted content will be returned to the application via the `RsslDefaultMsgCallback`.

All content requested, received, or posted is encoded and decoded using the Transport API Message Package and the Transport API Data Package described in the *Transport API C Edition Developers Guide*.

4.10 Disconnect Consumers and Shut Down

If the `RsslReactor` application must shut down, it can either leave consumer connections intact or shut them down. If the provider decides to close consumer connections, the provider should send an `RsslStatusMsg` on each connection's Login stream closing the stream. At this point, the consumer should assume that its other open streams are also closed.

1. Because this is instantiated by the provider, the application should use a `streamId` with a negative value. Additional details are provided in subsequent chapters.

It can then close the `RsslReactorChannel`s by calling `rsslReactorCloseChannel`. Prior to closing the `RsslReactorChannel`, the application should release any unwritten pool buffers to ensure proper memory cleanup.

If the application is done with the `RsslReactor`, the `rsslDestroyReactor` function can be used to shutdown and cleanup any `RsslReactor` resources.

Closing an `RsslReactorChannel` is described in Section 6.4.2. Shutting down an `RsslReactor` is described in Section 6.2.2.

4.11 Additional Interactive Provider Details

For specific details about OMM Interactive Providers, the Transport API and Transport API Value Added Component use, refer to the following locations:

- The **rsslVAProvider** application demonstrates one implementation of an OMM Interactive Provider application that uses Transport API Value Added Components. The application's source code and ReadMe file have additional information about specific implementation and behaviors.
- Chapter 6 provides a detailed look at the Transport API Reactor.
- Chapter 7 provides more information about the Administration Domain Model Representations.
- The *Transport API C Edition Developers Guide* provides specific Transport API encoder/decoder and transport usage information.
- The *Transport API C Edition RDM Usage Guide* provides specific information about the DMMs used by this application type.

Chapter 5 Building an OMM Non-Interactive Provider

5.1 Building an OMM Non-Interactive Provider Overview

This chapter provides an overview of how to create an OMM Non-Interactive Provider application using the Transport API Reactor and Administration Domain Model Representation Value Added Components. The Value Added Components simplify the work done by an OMM Non-Interactive Provider application when establishing a connection to ADH devices. After the Reactor indicates that the connection is ready, an OMM Non-Interactive Provider can publish information into the ADH cache without needing to handle requests for the information. The ADH can cache the information and along with other Enterprise Platform components, provide the information to any OMM Consumer applications that indicate interest.

The general process can be summarized by the following steps.

- Leverage Existing or Create New `RsslReactor`
- Implement Callbacks and Populate Role
- Establish connection using `rsslReactorConnect`
- Perform Dictionary Download
- Provide content
- Log out and shut down

The **rsslVANIProvider** example application, included with the Transport API product, provides one implementation of an OMM Consumer application that uses the Transport API Value Added Components. The application is written with simplicity in mind and demonstrates usage of the Transport API and Transport API Value Added Components. Portions of functionality have been abstracted and can easily be reused, though you might need to modify it to achieve your own unique performance and functionality goals.

5.2 Leverage Existing or Create New RsslReactor

The `RsslReactor` can manage one or multiple `RsslReactorChannel` structures. This allows the application to choose to associate OMM Non-Interactive Provider connections with an existing `RsslReactor`, having it manage more than one connection, or to create a new `RsslReactor` to use with the connection.

If the application is creating a new `RsslReactor`, the `rsslCreateReactor` function is used. This will create any necessary memory and threads that the `RsslReactor` uses to manage `RsslReactorChannel`s and their content flow. If the application is using an existing `RsslReactor`, there is nothing additional to do.

Detailed information about the `RsslReactor` and its creation are available in Section 6.2.1.

5.3 Implement Callbacks and Populate Role

Before creating the OMM Non-Interactive Provider connection, the application needs to specify callback functions to use for all inbound content. The callback functions are specified on a per `RsslReactorChannel` basis so each channel can have its own unique callback functions or existing callback functions can be specified and shared across multiple `RsslReactorChannel`s.

Several of the callback functions are required for use with an `RsslReactor`. The application must have an:

- `RsslReactorChannelEventCallback`, which returns information about the `RsslReactorChannel` and its state (e.g., connection up)
- `RsslDefaultMsgCallback`, which processes all data not handled by other optional callbacks.

- In addition to the required callbacks, an OMM Non-Interactive Provider can specify administrative domain specific callback functions. The available domain specific callback is `RsslRDMLoginMsgCallback`, which processes all data for the RDM Login domain.

The `RsslReactorOMMNIProviderRole` structure should be populated with all callback information for the `RsslReactorChannel`.

The `RsslReactorOMMNIProviderRole` allows the application to provide login request and initial directory refresh information. This can be initialized with default information. The callback functions are specified on the `RsslReactorOMMConsumerRole` structure or with specific information according to the application and user. The `RsslReactor` will use this information when starting up the `RsslReactorChannel`.

Detailed information about the `RsslReactorOMMNIProviderRole` is in Section 6.3.1. Information about the various callback functions and their specifications are available in Section 6.5.2.

5.4 Establish Connection using rsslReactorConnect

Once the `RsslReactorOMMNIProviderRole` is populated, the application can use `rsslReactorConnect` to create a new outbound connection. `rsslReactorConnect` will create an OMM Non-Interactive Provider type connection using the provided configuration and role information.

Once the underlying connection is established a channel event will be returned to the application's `RsslReactorChannelEventCallback`; this will provide the `RsslReactorChannel` and to indicate the current connection state. At this point, the application can begin using the `rsslReactorDispatch` function to dispatch directly on this `RsslReactorChannel`, or continue using `rsslReactorDispatch` to dispatch across all channels associated with the `RsslReactor`.

The `RsslReactor` will use the login and directory information specified on the `RsslReactorOMMNIProviderRole` to perform all channel initialization for the user. After the user is logged in and has sent a source directory response, a channel event is returned to inform the application that the connection is ready.

The `rsslReactorConnect` function is described in Section 6.4.1.1. Dispatching is described in Section 6.5.

5.5 Perform Dictionary Download

If connected to a supporting ADH, an OMM NIP can download the RWFFld and RWFEnum dictionaries to retrieve the appropriate dictionary information for providing field list content. An OMM NIP can use this feature to ensure they are using the appropriate version of the dictionary or to encode data. To support the Provider Dictionary Download feature, the ADH sends a Login response message containing the **SupportProviderDictionaryDownload** login element. The dictionary request is sent using the Dictionary domain model[1].

The Transport API offers several utility functions for downloading and managing a properly-formatted field dictionary. There are also utility functions that the provider can use to encode the dictionary into an appropriate format for downloading or decoding.

For details on using the Login domain, expected message content, and available dictionary utility functions, refer to the *Transport API C Edition RDM Usage Guide*.

5.6 Provide Content

After the `RsslReactorChannel` is established, it can begin pushing content to the ADH. Each unique information stream should begin with an `RsslRefreshMsg`, conveying all necessary identification information for the content. Because the provider

1. Because the provider instantiates this request, the application should use a streamId with a negative value. Additional details are provided in subsequent chapters.

instantiates this information, a negative value `streamId` should be used for all streams. The initial identifying refresh can be followed by other status or update messages.

All content is encoded and decoded using the Transport API Message Package and the Transport API Data Package described in the *Transport API C Edition Developers Guide*.

5.7 Log Out and Shut Down

When the Consumer application is done retrieving or posting content, it can close the `RsslReactorChannel` by calling `rsslReactorCloseChannel`. This will close all item streams and log out the user. Prior to closing the `RsslReactorChannel`, the application should release any unwritten pool buffers to ensure proper memory cleanup.

If the application is done with the `RsslReactor`, the `rsslDestroyReactor` function can be used to shutdown and cleanup any `RsslReactor` resources.

Closing an `RsslReactorChannel` is described in Section 6.4.2. Shutting down an `RsslReactor` is described in Section 6.2.2.

5.8 Additional Non-Interactive Provider Details

The following locations discuss specific details about using OMM Non-Interactive Providers and the Transport API:

- The **rsslVANIProvider** application demonstrates one implementation of an OMM Non-Interactive Provider application that uses Transport API Value Added Components. The application’s source code and ReadMe file have additional information about the specific implementation and behaviors.
- Chapter 6 provides a detailed look at the Transport API Reactor.
- Chapter 7 provides more information about the Administration Domain Model Representations.
- The *Transport API C Edition Developers Guide* provides specific Transport API encoder/decoder and transport usage information.
- The *Transport API C Edition RDM Usage Guide* provides specific information about the DMMs used by this application type.

Chapter 6 Reactor Detailed View

6.1 Concepts

The ***Transport API Reactor*** is a connection management and event processing component that can significantly reduce the amount of code an application must write to leverage OMM. This component helps simplify many aspects of a typical Transport API application, regardless of whether the application is an OMM Consumer, OMM Interactive Provider, or OMM Non-Interactive Provider. The Transport API Reactor can help manage Consumer and Non-Interactive Provider start-up processing, including user log in, source directory establishment, and dictionary download. It also allows for dispatching of events to user implemented callback functions, handles flushing of user written content, and manages network pings on the user's behalf. Value Added domain representations are coupled with the Reactor, allowing domain specific callbacks to be presented with their respective domain representation for easier, more logical access to content. For a list and comparison of Transport API and Transport API Reactor functionalities, refer to Section 6.1.1.

The Transport API Reactor internally depends on the Administration Domain Model Representation component. This allows the user to provide and consume the administrative RDM types in a more logical format. This additionally hides encoding and decoding of these domains from the Reactor user, all interaction is via a simple structural representation. More information about the Administration Domain Model Representation value added component is available in Chapter 7. The Transport API Reactor also leverages several utility components, contained in the Value Added Utilities. This includes constructs like mutex locks, a simple queue, and memory buffers.

The Transport API Reactor helps to manage the life-cycle of a connection on the user's behalf. When a channel is associated with a Reactor, the Reactor will perform all necessary transport level initialization and alert the user, via a callback, when the connection is up, ready for use, and down. An application can simultaneously run multiple unique reactor instances, where each Reactor instance can associate and manage a single channel or multiple channels. This functionality allows users to quickly and easily horizontally scale their application to leverage multi-core systems or distribute content across multiple connections.

Each instance of the Transport API Reactor leverages multiple threads to help manage inbound and outbound data efficiently. The following figure illustrates a high-level view of the Reactor threading model.

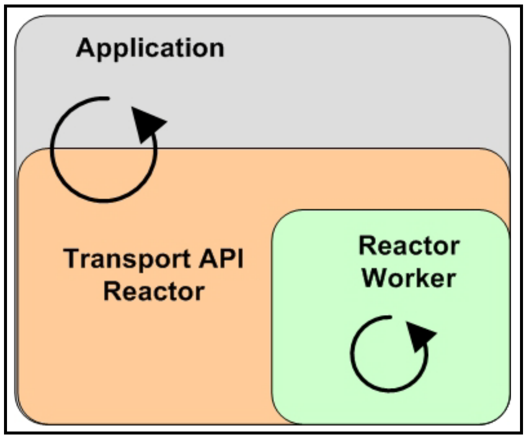


Figure 6. Transport API Reactor Thread Model

There are two main threads associated with each Transport API Reactor instance. The application thread is the main driver of the Reactor; all event dispatching (e.g., reading), callback processing, and submitting of data to the Transport API is done from this thread. This is done to reduce latency and simplify any threading model associated with the user defined callback functions – because callbacks happen from the application thread, a single threaded application does not need to have additional mutex locking. The Transport API Reactor also leverages an internal worker thread. The worker thread flushes any queued outbound data and manages outbound network pings for all channels associated with the Reactor.

The application drives the Reactor with the use of a dispatch function. The dispatch function reads content from the network, performs some light processing to handle inbound network pings, and provides the information to the user through a series of per-channel, user defined callback functions. Callback functions are separated based on whether they are Reactor callbacks or channel callbacks. Channel callbacks are separated by domain, with a default callback where all unhandled domains or non-OMM content are provided to the user. The application can choose whether to dispatch on a single channel or across all channels being managed by the Reactor. The application can leverage an I/O notification mechanism (e.g. select, poll) or periodically call dispatch – it is all up to the user.

6.1.1 Functionality: Transport API Versus Transport API Reactor

FUNCTIONALITY	TRANSPORT API	TRANSPORT API REACTOR
Programmatic Configuration	X	X
Programmatic Logging	X	X
Controlled Fragmentation and Assembly of Large Messages	X	X
Controlled Locking / Threading Model	X	X
Controlled Message Buffers with Ability to Change During Runtime	X	X
Controlled Message Packing	X	X
Support for Unified and Segmented Network Connection Types	X	X
Network Ping Management		X
Automatic Flushing of Data		X
User-Defined Callbacks for Data		X
User Login		X
Requesting Source Directory		X
Downloading Field Dictionary		X
Loading Field Dictionary File		X
***: Transport API users can implement this functionality themselves. They can also use or modify the Transport API Reactor functionality.		

Table 2: Transport API Functionality

6.1.2 Reactor Error Handling

The `RsslErrorInfo` structure is used to return error or warning information to the application. This can be returned from the various Reactor functions as well as part of a callback function. When returned directly from a Reactor function, this indicates that an error occurred while processing in that function. If returned as part of a callback function, this indicates that an error has occurred on one of the channels that the Reactor is managing.

The `RsslErrorInfo` members are described in the following table.

STRUCTURE MEMBER	DESCRIPTION
rsslErrorInfoCode	An informational code about this error. Indicates whether it reports a failure condition or is intended to provide non-failure related information to the user. For details on available codes, refer to Table 4.
rsslError	The underlying error information from the Transport API. This includes a pointer to the `RsslChannel` on which the error occurred, both a Transport API and a system error number, and more descriptive error text. The `RsslError` and its values are described in the *Transport API C Edition Developers Guide*.
errorLocation	Provides information about the file and line that the error occurred at. Detailed error text is provided via the `rsslError` portion of this structure. `RsslErrorInfo.errorLocation` length is limited to 1,024 bytes.

Table 3: `RsslErrorInfo` Structure Members

6.1.3 Reactor Error Info Codes

It is important that the application monitors return values from the `RsslReactor` callbacks and functions. The error info codes indicate whether the returned `RsslErrorInfo` indicates a failure condition or is providing information of a successful operation.

RETURN CODE	DESCRIPTION
RSSL_EIC_SUCCESS	Indicates a success code. Used to inform the user of the success and provide additional information.
RSSL_EIC_FAILURE	A general failure has occurred. The `RsslErrorInfo` code contains more information about the specific error.

Table 4: Reactor Error Info Codes

6.1.4 Transport API Reactor Application Lifecycle

The following figure depicts the typical lifecycle of an application using the Transport API Reactor, as well as the associated function calls. The subsequent sections in this document provide more detailed information.

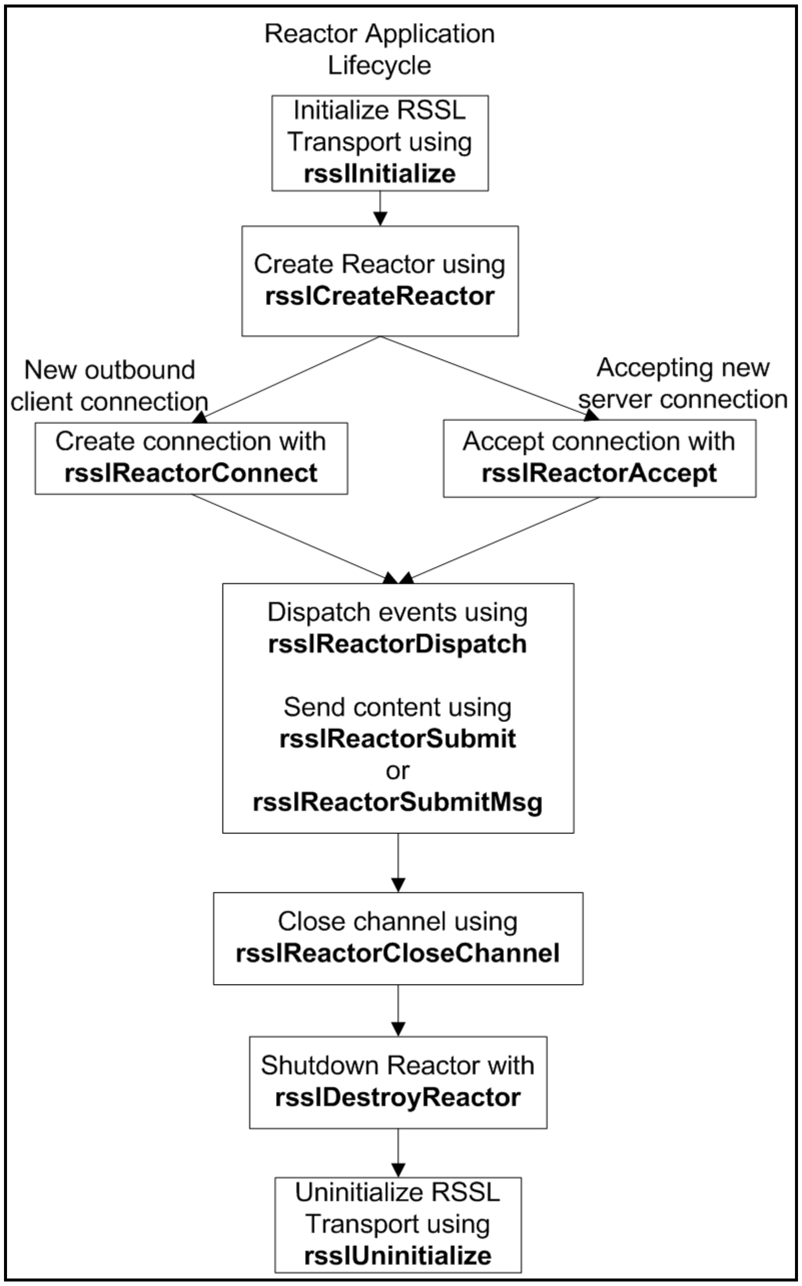

Reactor Application Lifecycle
Initialize RSSL Transport using **rsslInitialize**
Create Reactor using **rsslCreateReactor**
New outbound client connection
Accepting new server connection
Create connection with **rsslReactorConnect**
Accept connection with **rsslReactorAccept**
Dispatch events using **rsslReactorDispatch**
Send content using **rsslReactorSubmit** or **rsslReactorSubmitMsg**
Close channel using **rsslReactorCloseChannel**
Shutdown Reactor with **rsslDestroyReactor**
Uninitialize RSSL Transport using **rsslUninitialize**

Figure 7. Transport API Reactor Application Lifecycle

6.2 Reactor Use

This section describes use of the Reactor, or `RsslReactor`. The `RsslReactor` manages `RsslReactorChannel`s which are described in Section 6.3. An understanding of both constructs is necessary for application writers.

Before creating any `RsslReactor` instance, the user must ensure that the Transport API has been properly initialized. This is accomplished through the use of the `rsslInitialize` function, as documented in the *Transport API C Edition Developers Guide*. Because the `RsslReactor` internally leverages multiple threads, the **RSSL_LOCK_GLOBAL_AND_CHANNEL** option must be specified in the call to `rsslInitialize`. After the Transport API has been properly initialized, the application can create an `RsslReactor` instance. The `RsslReactor` is represented by a structure as defined in the following table.

Note: An application can leverage multiple `RsslReactor` instances to scale across multiple cores and distribute their `RsslReactorChannel`s as needed.

STRUCTURE MEMBER	DESCRIPTION
eventFd	Represents a file descriptor that can be used in some kind of I/O notification mechanism (e.g. select, poll). This file descriptor is associated with `RsslReactorChannel` connection events or `RsslReactor` specific events, for example an `RsslReactorChannel` **up** or **down** notification. All `RsslReactorChannel` data event notification occurs on the `RsslReactorChannel`'s specific `socketId`, as detailed in Section 6.3.
userSpecPtr	A pointer that can be set by the user of the `RsslReactor`. This value can be set directly or via the creation options. This information can be useful for identifying a specific instance of an `RsslReactor` or coupling this `RsslReactor` with other user created information.

Table 5: `RsslReactor` Structure Members

6.2.1 Creating a Reactor

The lifecycle of an `RsslReactor` is controlled by the application, which controls creation and destruction of each reactor instance. The following sections describe the creation functionality in more detail.

6.2.1.1 Reactor Creation

The creation of an `RsslReactor` instance can be accomplished through the use of the following function.

Note: Before the first use of any Transport API Reactor functionality, the application must ensure that `rsslInitialize` has been called with the **RSSL_LOCK_GLOBAL_AND_CHANNEL** option.

FUNCTION NAME	DESCRIPTION
rsslCreateReactor	Creates an `RsslReactor` instance, including all necessary internal memory and threads. After creating the `RsslReactor`, `RsslReactorChannel`s can be associated, as described in Section 6.3. Options are passed in via the `RsslCreateReactorOptions`, as defined in Table 7.

Table 6: `RsslReactor` Creation Function

6.2.1.2 RsslCreateReactorOptions Structure Members

STRUCTURE MEMBER	DESCRIPTION
dispatchDecodeMemoryBufferSize	The size, in bytes, of an internally created memory buffer. The memory buffer is used by the `RsslReactor` when performing any necessary message decoding required for callbacks. When cleared, defaults to 65,536 bytes.
port	**Deprecated**. RsslReactor now chooses an ephemeral port upon creation. Any values specified in this parameter are ignored.

Table 7: `RsslCreateReactorOptions` Structure Members

STRUCTURE MEMBER	DESCRIPTION
userSpecPtr	A pointer that can be set by the application. This value is preserved and stored in the `userSpecPtr` of the `RsslReactor` returned from `rsslCreateReactor`. This information can be useful for coupling this `RsslChannel` with other user created information, such as a watch list associated with this connection.

Table 7: `RsslCreateReactorOptions` Structure Members (Continued)

6.2.1.3 RsslCreateReactorOptions Utility Function

The Transport API provides the following utility function for use with the `RsslCreateReactorOptions`.

FUNCTION NAME	DESCRIPTION
rsslClearCreateReactorOptions	Clears the `RsslCreateReactorOptions` structure. Useful for structure reuse.

Table 8: `RsslCreateReactorOptions` Utility Function

6.2.2 Destroying a Reactor

The lifecycle of an `RsslReactor` is controlled by the application, which controls creation and destruction of each reactor instance. The following sections describe the destruction functionality in more detail.

6.2.2.1 Reactor Destruction

When the application no longer requires an `RsslReactor` instance, it can destroy it using the following function.

FUNCTION NAME	DESCRIPTION
rsslDestroyReactor	Destroys an `RsslReactor` instance, including all internal memory and threads. This also sends `RsslReactorChannelEvent`s, indicating channel down, to all `RsslReactorChannel`s associated with this `RsslReactor`.

Table 9: `RsslReactor` Destruction Function

6.2.2.2 Reactor Creation and Destruction Example

```
RsslCreateReactorOptions reactorCreateOptions;

/* Use of reactors requires that RSSL be initialized with both global
 * and per-channel locks. */
ret = rsslInitialize(RSSL_LOCK_GLOBAL_AND_CHANNEL, &rsslError);

rsslClearCreateReactorOptions(&reactorCreateOptions);

/* Create the RsslReactor. */
pReactor = rsslCreateReactor(&reactorCreateOptions, &rsslErrorInfo);

/* Any use of the reactor occurs here -- see following sections for all other functionality */

/* Destroy the RsslReactor. */
ret = rsslDestroyReactor(pReactor, &rsslErrorInfo);
```

```
/* Uninitialize RSSL. */
ret = rsslUninitialize();
```

Code Example 1: Reactor Creation and Destruction Example

6.3 Reactor Channels

The **`RsslReactorChannel`** structure is used to represent a connection that can send or receive information across a network. This structure is used to represent a connection, regardless of if that was an outbound connection or a connection that was accepted by a listening socket via an **`RsslServer`**. The **`RsslReactorChannel`** is the application's point of access, used to perform any action on the connection that it represents (e.g. dispatching events, writing, disconnecting, etc). See the subsequent sections for more information about **`RsslReactorChannel`** and how to associate with an **`RsslReactor`**.

Note: Only Transport API Reactor functions, like those defined in this chapter, should be called on a channel managed by an RsslReactor.

The following table describes the members of the **`RsslReactorChannel`** structure.

STRUCTURE MEMBER	DESCRIPTION
pRsslChannel	A pointer to the underlying **`RsslChannel`** structure, as defined in the *Transport API C Edition Developers Guide*, mainly for reference purposes. All operations should be performed using the Transport API Reactor functionality; the application should not use this **`RsslChannel`** directly with any RSSL Transport functionality.
pRsslServer	A pointer to the underlying **`RsslServer`** structure, as defined in the *Transport API C Edition Developers Guide*, mainly for reference purposes. This will only be populated when the channel was created via the **`rsslReactorAccept`** function, as described in Section 6.4.1.6.
socketId	Represents a file descriptor that can be used in some kind of I/O notification mechanism (e.g. select, poll) to alert users when dispatch is required on a specific **`RsslReactorChannel`**. This is the file descriptor associated with this end of the network connection; the file descriptor value may be different from the other end of the connection.
oldSocketId	It is possible for a file descriptor to change over time, typically due to some kind of connection keep-alive mechanism. If this occurs, this is typically communicated via a callback indicating **RSSL_RC_CET_FD_CHANGE**. The previous **`socketId`** is stored in **`oldSocketId`** so the application can properly unregister and then register the new **`socketId`** with their I/O notification mechanism.
protocolType	When an **`RsslReactorChannel`** is up (**RSSL_RC_CET_CHANNEL_UP**), this is populated with the **`protocolType`** associated with the content being sent on this connection. If the **`protocolType`** indicated by a server does not match the **`protocolType`** that a client specifies, the connection will be rejected. The Transport API Reactor will leverage the versioning information for any content it is encoding or decoding. Proper use of versioning should be handled by the application for any other application encoded or decoded content. See the *Transport API C Edition Developers Guide* for more information about use of versioning.

Table 10: `RsslReactorChannel` Structure Members

STRUCTURE MEMBER	DESCRIPTION
majorVersion	When an `RsslReactorChannel` is up (**RSSL_RC_CET_CHANNEL_UP**), this is populated with the major version number that is associated with the content being sent on this connection. Typically, a minor version increase is associated with a fully backward compatible change or extension. The Transport API Reactor will leverage the versioning information for any content it is encoding or decoding. Proper use of versioning should be handled by the application for any other application encoded or decoded content. See the *Transport API C Edition Developers Guide* for more information about use of versioning.
minorVersion	When an `RsslReactorChannel` is up (**RSSL_RC_CET_CHANNEL_UP**), this is populated with the minor version number that is associated with the content being sent on this connection. Typically, a minor version increase is associated with a fully backward compatible change or extension. The Transport API Reactor will leverage the versioning information for any content it is encoding or decoding. Proper use of versioning should be handled by the application for any other application encoded or decoded content. For more information about use of versioning, refer to the *Transport API C Edition Developers Guide*.
userSpecPtr	A pointer that can be set by the user of the `RsslChannel`. This value can be set directly or via `RsslReactorConnectOptions` and `RsslReactorAcceptOptions`. This information can be useful for coupling this `RsslReactorChannel` with other user created information, such as a watch list associated with this connection.

Table 10: `RsslReactorChannel` Structure Members (Continued)

6.3.1 Reactor Channel Roles

An `RsslReactorChannel` can be configured to fulfill several specific roles, which overlap with the typical OMM application types. The provided role definitions are:

- `RsslReactorOMMConsumerRole` for OMM Consumer applications
- `RsslReactorOMMProviderRole` for OMM Interactive Provider applications
- `RsslReactorOMMNIProviderRole` for OMM Non-Interactive Provider applications

All roles have the same common element, the `RsslReactorChannelRoleBase`.

6.3.1.1 RsslReactorChannelRoleBase Structure

`RsslReactorChannelRoleBase` contains information and callback functions common to all role types and consists of the following members:

STRUCTURE MEMBER	DESCRIPTION
roleType	The role type enumeration value, as defined in Section 6.3.1.2.
channelEventCallback	This `RsslReactorChannel`'s user-defined callback function to handle all `RsslReactorChannel` specific events, like **RSSL_RC_CET_CHANNEL_UP** or **RSSL_RC_CET_CHANNEL_DOWN**. This callback function is required for all role types. This callback is defined in more detail in Section 6.5.2.

Table 11: `RsslReactorChannelRoleBase` Structure Members

STRUCTURE MEMBER	DESCRIPTION
defaultMsgCallback	This `RsslReactorChannel`'s user-defined callback function to handle `RsslMsg` content not handled by another domain-specific callback function. This callback function is required for all role types. This callback is defined in more detail inSection 6.5.2.

Table 11: `RsslReactorChannelRoleBase` Structure Members (Continued)

6.3.1.2 roleType Enumerations

ENUMERATED NAME	DESCRIPTION
RSSL_RC_RT_INIT	Role is not specified. This is intended for structure initialization only.
RSSL_RC_RT_OMM_CONSUMER	Indicates that the `RsslReactorChannel` should act as an OMM Consumer.
RSSL_RC_RT_OMM_PROVIDER	Indicates that the `RsslReactorChannel` should act as an OMM Interactive Provider.
RSSL_RC_RT_OMM_NI_PROVIDER	Indicates that the `RsslReactorChannel` should act as an OMM Non-Interactive Provider.

Table 12: `RsslReactorChannelRoleBase.role` Enumerated Values

6.3.2 Reactor Channel Role: OMM Consumer

When an `RsslReactorChannel` is acting as an OMM Consumer application, it connects to an OMM Interactive Provider. As part of this process it is expected to perform a login to the system. Once the login is completed, the consumer acquires a source directory, which provides information about the available services and their capabilities. Additionally, a consumer can download or load field dictionaries, providing information to help decode some types of content. The messages that are exchanged during this connection establishment process are administrative RDMs and are described in the *Transport API C Edition RDM Usage Guide*.

An `RsslReactorChannel` in a consumer role helps to simplify this connection process by exchanging these messages on the user's behalf. The user can choose to provide specific information or leverage a default populated messages, which uses the information of the user currently logged into the machine running the application. In addition, the Transport API Reactor allows the application to specify user-defined callback functions to handle the processing of received messages on a per-domain basis.

6.3.2.1 OMM Consumer Role

When creating an `RsslReactorChannel`, this information can be specified with the `RsslReactorOMMConsumerRole` structure, as defined below.

STRUCTURE MEMBER	DESCRIPTION
base	The role base structure, as defined in Table 11.
pLoginRequest	The `RsslRDMLoginRequest`, defined in Section 6.5.2, to be sent during the connection establishment process. This can be populated with a user's specific information or invoke the `rsslInitDefaultRDMLoginRequest` function to populate with default information. If this parameter is left empty no login will be sent to the system; useful for systems that do not require a login.

Table 13: `RsslReactorOMMConsumerRole` Structure Members

STRUCTURE MEMBER	DESCRIPTION
pDirectoryRequest	The **`RsslRDMDirectoryRequest`**, defined in Section 6.5.2, to be sent during the connection establishment process. This can be populated with specific source directory request information or invoke the **`rsslInitDefaultRDMDirectoryRequest`** function to populate with default information. If this parameter is specified, a **`pLoginRequest`** is required. If this parameter is left empty, no directory request will be sent to the system.
dictionaryDownloadMode	Informs the **`RsslReactorChannel`** on the method to use when requesting dictionaries. The allowable modes are defined in Table 14.
loginMsgCallback	This **`RsslReactorChannel`**'s user-defined callback function to handle login message content. If not specified, all received login messages will be passed to the **`defaultMsgCallback`**. • This callback is defined in more detail in Section 6.5.2. • The login messages are described in Section 7.3.
directoryMsgCallback	This **`RsslReactorChannel`**'s user-defined callback function to handle directory message content. If not specified, all received directory messages will be passed to the **`defaultMsgCallback`**. • This callback is defined in more detail in Section 6.5.2. • The directory messages are described in Section 7.4.
dictionaryMsgCallback	This **`RsslReactorChannel`**'s user-defined callback function to handle dictionary message content. If not specified, all received dictionary messages will be passed to the **`defaultMsgCallback`**. • This callback is defined in more detail in Section 6.5.2. • The dictionary messages are described in Section 7.5.
watchlistOptions	Configurable options for the consumer watchlist. The options are described in more detail in Section 6.3.2.3.

Table 13: `RsslReactorOMMConsumerRole` Structure Members (Continued)

6.3.2.2 OMM Consumer Role Dictionary Download Modes

There are several dictionary download options available to an **`RsslReactorChannel`**. The application can determine which option is desired and specify via the **`RsslReactorOMMConsumerRole.dictionaryDownloadMode`** parameter.

ENUMERATED NAME	DESCRIPTION
RSSL_RC_DICTIONARY_DOWNLOAD_NONE	The **`RsslReactor`** will not request any dictionaries for this **`RsslReactorChannel`**. This is typically used when the application has loaded a file based dictionary or has acquired the dictionary elsewhere.
RSSL_RC_DICTIONARY_DOWNLOAD_FIRST_AVAILABLE	The **`RsslReactor`** will search received directory messages for the RDMFieldDictionary (RWFFld) and the **enumtype.def** (RWFEnum) dictionaries. Once found, the **`RsslReactor`** will request these dictionaries for the application. After transmission is completed, the streams will be closed as this content does not update.

Table 14: RsslReactorOMMConsumerRole.dictionaryDownloadMode Enumerated Values

6.3.2.3 OMM Consumer Role Watchlist Options

The consumer may enable an internal watchlist and configure behaviors. For more detail on the consumer watchlist feature, refer to Section 2.4.

OPTION	DESCRIPTION
enableWatchlist	Enables the watchlist.
itemCountHint	Can improve performance when used with the watchlist. If possible, set this to the approximate number of item requests the application expects to open.
obeyOpenWindow	Sets whether the `RsslReactor` obeys the OpenWindow of services advertised in a provider's Source Directory response.
maxOutstandingPosts	Sets the maximum allowable number of on-stream posts waiting for acknowledgment before the reactor disconnects.
postAckTimeout	Sets the time (in milliseconds) a stream waits to receive an ACK for an outstanding post before forwarding a negative acknowledgment `RsslAckMsg` to the application.
requestTimeout	Sets the time (in milliseconds) the watchlist waits for a response to a request.

Table 15: OMM Consumer Role Watchlist Options

6.3.2.4 OMM Consumer Role Utility Function

The Transport API provides the following utility function for use with the `RsslReactorOMMConsumerRole`.

FUNCTION NAME	DESCRIPTION
rsslClearOMMConsumerRole	Clears the `RsslReactorOMMConsumerRole` structure. Useful for structure reuse.

Table 16: `RsslReactorOMMConsumerRole` Utility Function

6.3.3 Reactor Channel Role: OMM Provider

When an `RsslReactorChannel` is acting as an OMM Provider application, it allows connections from OMM Consumer applications. As part of this process it is expected to respond to login requests and source directory information requests. Additionally, a provider can optionally allow consumers to download field dictionaries. The messages that are exchanged during this connection establishment process are administrative RDMs and are described in the *Transport API C Edition RDM Usage Guide*.

An `RsslReactorChannel` in an interactive provider role allows the application to specify user-defined callback functions to handle the processing of received messages on a per-domain basis.

6.3.3.1 OMM Provider Role Members

When creating an `RsslReactorChannel`, this information can be specified with the `RsslReactorOMMProviderRole` structure, as defined below.

STRUCTURE MEMBER	DESCRIPTION
base	The role base structure, as defined in Table 11.

Table 17: `RsslReactorOMMProviderRole` Structure Members

STRUCTURE MEMBER	DESCRIPTION
loginMsgCallback	This `RsslReactorChannel`'s user-defined callback function to handle login message content. If not specified, all received login messages will be passed to the `defaultMsgCallback`. • This callback is defined in more detail in Section 6.5.2. • The login messages are described in Section 7.3.
directoryMsgCallback	This `RsslReactorChannel`'s user-defined callback function to handle directory message content. If not specified, all received login messages will be passed to the `defaultMsgCallback`. • This callback is defined in more detail in Section 6.5.2. • The directory messages are described in Section 7.4.
dictionaryMsgCallback	This `RsslReactorChannel`'s user-defined callback function to handle dictionary message content. If not specified, all received login messages will be passed to the `defaultMsgCallback`. • This callback is defined in more detail in Section 6.5.2. • The dictionary messages are described in Section 7.5.
tunnelStreamListenerCallback	This `RsslReactorChannel`'s user-defined callback for accepting or rejecting tunnel streams. For further details on this callback, refer to Section 6.7.6.

Table 17: `RsslReactorOMMProviderRole` Structure Members (Continued)

6.3.3.2 OMM Provider Role Utility Function

The Transport API provides the following utility function for use with the `RsslReactorOMMProviderRole`.

FUNCTION NAME	DESCRIPTION
rsslClearOMMProviderRole	Clears the `RsslReactorOMMProviderRole` structure. Useful for structure reuse.

Table 18: `RsslReactorOMMProviderRole` Utility Function

6.3.4 Reactor Channel Role: OMM Non-Interactive Provider

When an `RsslReactorChannel` is acting as an OMM Non-Interactive Provider application, it connects to an Enterprise Platform ADH. As part of this process it is expected to perform a login to the system. Once the login is completed, the non-interactive provider publishes a source directory, which provides information about the available services and their capabilities. The messages that are exchanged during this connection establishment process are administrative RDMs and are described in the *Transport API C Edition RDM Usage Guide*.

An `RsslReactorChannel` in a non-interactive provider role helps to simplify this connection process by exchanging these messages on the user's behalf. The user can choose to provide specific information or leverage a default populated messages, which uses the information of the user currently logged into the machine running the application. In addition, the Transport API Reactor allows the application to specify user-defined callback functions to handle the processing of received messages on a per-domain basis.

6.3.4.1 OMM Non-Interactive Role Members

When creating an `RsslReactorChannel`, this information can be specified with the `RsslReactorOMMNIProviderRole` structure, as defined below.

STRUCTURE MEMBER	DESCRIPTION
base	The role base structure, as defined in Table 11.
pLoginRequest	The `RsslRDMLoginRequest`, defined in Section 7.3.1, to be sent during the connection establishment process. This can be populated with a user's specific information or invoke the `rsslInitDefaultRDMLoginRequest` function to populate with default information. If this parameter is left empty no login will be sent to the system; useful for systems that do not require a login.
pDirectoryRefresh	The `RsslRDMDirectoryRefresh`, defined in Section 7.4.2, to be sent during the connection establishment process. This can be populated with specific source directory refresh information. If this parameter is specified, a `pLoginRequest` is required. If this parameter is left empty, no directory request will be sent to the system.
loginMsgCallback	This `RsslReactorChannel`'s user-defined callback function to handle login message content. If not specified, all received login messages will be passed to the `defaultMsgCallback`. This callback is defined in more detail in Section 6.5.2.

Table 19: RsslReactorOMMNIProviderRole Structure Members

6.3.4.2 OMM Non-Interactive Provider Role Utility Function

The Transport API provides the following utility function for use with the `RsslReactorOMMNIProviderRole`.

FUNCTION NAME	DESCRIPTION
rsslClearOMMNIProviderRole	Clears the `RsslReactorOMMNIProviderRole` structure. Useful for structure reuse.

Table 20: `RsslReactorOMMNIProviderRole` Utility Function

6.3.5 Reactor Channel: Role Union

A union is provided that allows use of any of the role structures. This is mainly for use within the `RsslReactor` implementation; however it is documented in the event that it can be useful to an application.

6.3.5.1 Union Members

UNION MEMBER	DESCRIPTION
base	The role's base structure, as defined in Table 11.
ommConsumerRole	The `RsslReactorOMMConsumerRole`, as defined in Section 6.3.2.
ommProviderRole	The `RsslReactorOMMProviderRole`, as defined in Section Section 6.3.3.
ommNIProviderRole	The `RsslReactorOMMNIProviderRole`, as defined in Section 6.3.4.

Table 21: `RsslReactorChannelRole` Union Members

6.3.5.2 Union Utility Function

The Transport API provides the following utility function for use with the `RsslReactorOMMProviderRole`.

FUNCTION NAME	DESCRIPTION
rsslClearReactorChannelRole	Clears the `RsslReactorChannelRole` union.

Table 22: `RsslReactorChannelRole` Utility Function

6.4 Managing Reactor Channels

6.4.1 Adding Reactor Channels

A single `RsslReactor` instance can manage one or many `RsslReactorChannel`s. An `RsslReactorChannel` can be instantiated as an outbound client style connection or as a connection that is accepted from an `RsslServer`. This allows a user to mix connection styles within or across Reactors and have a consistent usage and behavior.

Note: A single `RsslReactor` can simultaneously manage `RsslReactorChannel`s from `rsslReactorConnect` and `rsslReactorAccept`.

6.4.1.1 Reactor Connect

The `rsslReactorConnect` function will create a new `RsslReactorChannel` and associate it with an `RsslReactor`. This function creates a new outbound connection. The `RsslReactorChannel` will be returned to the application via a callback, as described in Section 6.5.2, at which point it should begin dispatching.

Client applications can specify that `RsslReactor` automatically reconnect an `RsslReactorChannel` whenever a connection fails. To enable this, the application sets the appropriate members of the `RsslReactorConnectOptions` structure. The application can specify that `RsslReactor` reconnect `RsslReactorChannel` to the same host, or to one of multiple hosts.

Consumer applications can combine this feature with the watchlist feature to enable recovery of item streams across connections. For more information on the watchlist feature, refer to Section 2.4.

FUNCTION NAME	DESCRIPTION
rsslReactorConnect	Creates an `RsslReactorChannel` that makes an outbound connection to the configured host. This establishes a connection in a manner similar to the `rsslConnect` function, as described in the *Transport API C Edition Developers Guide*. Connection options are passed in via the `RsslReactorConnectOptions`, as defined in Table 24. `RsslReactorChannel` specific information, such as the per-channel callback functions, the type of behavior, default RDM messages, and such are passed in via the `RsslReactorChannelRole`, as defined in Section 6.3.1.

Table 23: `rsslReactorConnect` Function

6.4.1.2 RsslReactorConnectOptions Structure Members

STRUCTURE MEMBER	DESCRIPTION
rsslConnectOptions	The `RsslConnectOptions` associated with the underlying `rsslConnect` function. This includes information about the host or network to connect to, the type of connection to use, and other transport specific configuration information. This is described in more detail in the *Transport API C Edition Developers Guide*.
initializationTimeout	The amount of time (in seconds) to wait for the successful connection establishment of an `RsslReactorChannel`. If a timeout occurs, an event is dispatched to the application to indicate that the `RsslReactorChannel` is down.
reconnectAttemptLimit	The maximum number of times the `RsslReactor` attempts to reconnect a channel when it fails. If set to **-1**, there is no limit.
reconnectMinDelay	The minimum time the `RsslReactor` waits (in milliseconds) before attempting to reconnect a failed channel. The time increases with each reconnection attempt, from `reconnectMinDelay` to `reconnectMaxDelay`.
reconnectMaxDelay	The maximum time the `RsslReactor` waits (in milliseconds) before attempting to reconnect a failed channel. The time increases with each reconnection attempt, from `reconnectMinDelay` to `reconnectMaxDelay`.
reactorConnectionList	Specifies an array of connection information. When used with `reconnectAttemptLimit`, the `RsslReactor` attempts to connect to each host in the list with each reconnection attempt.
connectionCount	The number of connections listed in `reactorConnectionList`. If set to **0**, `rsslConnectOptions` is used.

Table 24: `RsslReactorConnectOptions` Structure Members

6.4.1.3 RsslReactorConnectOptions Utility Function

The Transport API provides the following utility function for use with the `RsslReactorConnectOptions`.

FUNCTION NAME	DESCRIPTION
rsslClearReactorConnectOptions	Clears the `RsslReactorConnectOptions` structure. Useful for structure reuse.

Table 25: `RsslReactorConnectOptions` Utility Function

6.4.1.4 rsslReactorConnect Example

```
RsslReactorConnectOptions reactorConnectOpts;
RsslReactorOMMConsumerRole consumerRole;

RsslRDMLoginRequest loginRequest;
RsslRDMDirectoryRequest directoryRequest;

/* Configure connection options.*/
rsslClearReactorConnectOptions(&reactorConnectOpts);
reactorConnectOpts.rsslConnectOptions.connectionInfo.unified.address = "localhost";
reactorConnectOpts.rsslConnectOptions.connectionInfo.unified.serviceName = "14002";
```

```
/* Configure a role for this connection as an OMM Consumer. */
rsslClearOMMConsumerRole(&consumerRole);

/* Set the functions to which rsslDispatch will deliver events. */
consumerRole.base.channelEventCallback = channelEventCallback;
consumerRole.base.defaultMsgCallback = defaultMsgCallback;
consumerRole.loginMsgCallback = loginMsgCallback;
consumerRole.directoryMsgCallback = directoryMsgCallback;
consumerRole.dictionaryMsgCallback = dictionaryMsgCallback;

/* Prepare a login request. Once the channel is initialized this message will be sent. */
rsslInitDefaultRDMLoginRequest(&loginRequest, 1);
consumerRole.pLoginRequest = &loginRequest;

/* Prepare a directory request. Once the application has logged in, this message will be sent. */
rsslInitDefaultRDMDirectoryRequest(&directoryRequest, 2);
consumerRole.pDirectoryRequest = &directoryRequest;

/* Add the connection to the RsslReactor. */
ret = rsslReactorConnect(pReactor, &reactorConnectOpts, (RsslReactorChannelRole*)&consumerRole,
        &rsslErrorInfo);
```

Code Example 2: `rsslReactorConnect` Example

6.4.1.5 rsslReactorConnect Segmented Multicast Consumer Example

```
RsslReactorConnectOptions reactorConnectOpts;
RsslReactorOMMConsumerRole consumerRole;

RsslRDMLoginRequest loginRequest;
RsslRDMDirectoryRequest directoryRequest;

/* Configure connection options.*/
rsslClearReactorConnectOptions(&reactorConnectOpts);
reactorConnectInfo.rsslConnectOptions.connectionType = RSSL_CONN_TYPE_RELIABLE_MCAST;

/* Configure outgoing network */
reactorConnectOpts.rsslConnectOptions.connectionInfo.segmented.sendAddress = “232.6.6.1”;
reactorConnectOpts.rsslConnectOptions.connectionInfo.segmented.sendServiceName = “30010”;

/* Configure incoming network. This example listens to two multicast networks. */
reactorConnectOpts.rsslConnectOptions.connectionInfo.segmented.recvAddress = “232.6.6.2,232.6.6.4”;
reactorConnectOpts.rsslConnectOptions.connectionInfo.segmented.recvServiceName = “30011”;
reactorConnectOpts.rsslConnectOptions.connectionInfo.segmented.unicastServiceName = “55555”;

/* Enable filtering of incoming multicast traffic. */
reactorConnectOpts.rsslConnectOptions.multicastOpts.flags = RSSL_MCAST_FILTERING_ON;
```

```
/* Configure a role for this connection as an OMM Consumer. */
rsslClearOMMConsumerRole(&consumerRole);

/* Set the functions to which rsslDispatch will deliver events. */
consumerRole.base.channelEventCallback = channelEventCallback;
consumerRole.base.defaultMsgCallback = defaultMsgCallback;
consumerRole.loginMsgCallback = loginMsgCallback;
consumerRole.directoryMsgCallback = directoryMsgCallback;
consumerRole.dictionaryMsgCallback = dictionaryMsgCallback;

/* Prepare a login request. Once the channel is initialized this message will be sent. */
rsslInitDefaultRDMLoginRequest(&loginRequest, 1);
consumerRole.pLoginRequest = &loginRequest;

/* Prepare a directory request. Once the application has logged in, this message will be sent. */
rsslInitDefaultRDMDirectoryRequest(&directoryRequest, 2);
consumerRole.pDirectoryRequest = &directoryRequest;

/* Add the connection to the RsslReactor. */
ret = rsslReactorConnect(pReactor, &reactorConnectOpts, (RsslReactorChannelRole*)&consumerRole,
        &rsslErrorInfo);
```

Code Example 3: `rsslReactorConnect` Segmented Multicast Consumer Example

6.4.1.6 Reactor Accept

The `rsslReactorAccept` function will create a new `RsslReactorChannel` and associate it with an `RsslReactor`. This function accepts the connection from an already running `RsslServer`. The `RsslReactorChannel` will be returned to the application via a callback, as described in Section 6.5.2, at which point it can begin dispatching on the channel.

FUNCTION NAME	DESCRIPTION
rsslReactorAccept	Creates an `RsslReactorChannel` by accepting it from an `RsslServer`. This establishes a connection in a manner similar to the `rsslAccept` function, as described in the *Transport API C Edition Developers Guide*. • Connection options are passed in via the `RsslReactorAcceptOptions`, as defined in Section 6.4.1.2. • `RsslReactorChannel`-specific information (such as the per-channel callback functions, the type of behavior, default RDM messages, and etc.) are passed in via the `RsslReactorChannelRole`, as defined in Section 6.3.1.

Table 26: `rsslReactorAccept` Function

6.4.1.7 RsslReactorAcceptOptions Structure Members

STRUCTURE MEMBER	DESCRIPTION
rsslAcceptOptions	The `RsslAcceptOptions` associated with the underlying `rsslAccept` function. This includes an option to reject the connection as well as a `userSpecPtr`. This is described in more detail in the *Transport API C Edition Developers Guide*.
initializationTimeout	The amount of time (in seconds) to wait for the successful connection establishment of an `RsslReactorChannel`. If a timeout occurs, an event is dispatched to the application to indicate that the `RsslReactorChannel` is down.

Table 27: `RsslReactorAcceptOptions` Structure Members

6.4.1.8 RsslReactorAcceptOptions Utility Function

The Transport API provides the following utility function for use with the `RsslReactorAcceptOptions`.

FUNCTION NAME	DESCRIPTION
rsslClearReactorAcceptOptions	Clears the `RsslReactorAcceptOptions` structure. Useful for structure reuse.

Table 28: `RsslReactorAcceptOptions` Utility Function

6.4.1.9 rsslReactorAccept Example

```
RsslReactorAcceptOptions reactorAcceptOpts;
RsslReactorOMMProviderRole providerRole;

/* Configure accept options.*/
rsslClearReactorAcceptOptions(&reactorAcceptOpts);

/* Configure a role for this connection as an OMM Provider. */
rsslClearOMMProviderRole(&providerRole);
providerRole.base.channelEventCallback = channelEventCallback;
providerRole.base.defaultMsgCallback = defaultMsgCallback;
providerRole.loginMsgCallback = loginMsgCallback;
providerRole.directoryMsgCallback = directoryMsgCallback;
providerRole.dictionaryMsgCallback = dictionaryMsgCallback;

/* Add the connection to the RsslReactor. */
rsslClearReactorAcceptOptions(&reactorAcceptOpts);
ret = rsslReactorAccept(pReactor, pRsslServer, &reactorAcceptOpts,
        (RsslReactorChannelRole*)&providerRole, &rsslErrorInfo);
```

Code Example 4: `rsslReactorAccept` Example

6.4.2 Removing Reactor Channels

6.4.2.1 rsslReactorClose Function

The following function can be used to remove an `RsslReactorChannel` from an `RsslReactor` instance. It will also close and clean up any resources associated with the `RsslReactorChannel`.

FUNCTION NAME	DESCRIPTION
rsslReactorCloseChannel	Removes an `RsslReactorChannel` from the passed in `RsslReactor` instance and cleans up associated resources. This will additionally invoke the `rsslCloseChannel` function, as described in the *Transport API C Edition Developers Guide*, to clean up any resources associated with the underlying `RsslChannel`. This function can be called from either outside or within a callback.

Table 29: rsslReactorCloseChannel Function

6.4.2.2 rsslReactorClose Example

```
RsslErrorInfo rsslErrorInfo;
/* Can be used inside or outside of a callback */
ret = rsslReactorCloseChannel(pReactor, pReactorChannel, &rsslErrorInfo);
```

Code Example 5: `rsslReactorClose` Example

6.5 Dispatching Data

Once an application has an `RsslReactor`, it can begin dispatching messages. Until there is at least one associated `RsslReactorChannel`, there is nothing to dispatch. Once there are `RsslReactorChannel`s to dispatch over, each channel will begin seeing its user-defined per-channel callbacks get invoked. For more information about the available callbacks and their specifications, refer to Section 6.5.2.

An application can choose to dispatch across all associated `RsslReactorChannel`s or to dispatch on a particular `RsslReactorChannel`. If dispatching on a single `RsslReactorChannel`, only data for this channel will be processed and returned via the channel's callback. If dispatching across multiple `RsslReactorChannel`s, the `RsslReactor` attempts to fairly dispatch over all channels. In either case, the dispatch call allows the application to specify the maximum number of messages that will be processed and returned via callback.

Typically, an application will register both the `RsslReactor.eventFd` and each `RsslReactorChannel`'s socketId with an I/O notifier (e.g., select, poll). The I/O notifier can help inform the application when data is available on particular `RsslReactorChannel`s or when channel information is available from the `RsslReactor`. An application can also forego notifier use and periodically call the dispatch function to ensure that data is processed.

The dispatch function is described in Section 6.5.1.

6.5.1 rsslReactorDispatch Function

Note: Applications should not call `rsslDestroyReactor` or `rsslReactorDispatch` from within a callback function. All other RsslReactor functionality is safe to use from within a callback.

Events received in callback functions should be assumed to be invalid once the callback function returns. For callbacks that provide `RsslMsg` or `RsslRDMMsg` structures, a deep copy of the object should be made if the application wishes to preserve it.

To copy an `RsslMsg`, see the `rsslCopyMsg` function in the *Transport API C Edition Developers Guide*; for copying an `RsslRDMMsg`, see the copy utility function for the appropriate `RsslRDMMsg` structure.

FUNCTION NAME	DESCRIPTION
rsslReactorDispatch	This function process events and messages across the provided `RsslReactor` and all of its associated `RsslReactorChannel`. When channel information or data is available for an `RsslReactorChannel`, the channels user-defined callback function will be invoked. The application can dispatch on a specified channel or over all channels associated with the `RsslReactor`. The application can also control the maximum number of messages dispatched with a single call to `rsslReactorDispatch`. This can be controlled through the passed in `RsslReactorDispatchOptions`, as described in Section 6.5.1.1.

Table 30: `rsslReactorDispatch` Function

6.5.1.1 Reactor Dispatch Options

The `RsslReactorDispatchOptions` allow the application control various aspects of the call to `rsslReactorDispatch`.

STRUCTURE MEMBER	DESCRIPTION
pReactorChannel	The specific `RsslReactorChannel` to dispatch on in this call. If **NULL**, `rsslReactorDispatch` will process across all `RsslReactorChannel`s associated with the passed in `RsslReactor`.
maxMessages	Controls the maximum number of events or messages processed in this call. If this is larger than the number of available messages, `rsslReactorDispatch` will return when there is no more data to process. This value is initialized to allow for up to 100 messages to be returned with a single call to `rsslReactorDispatch`.

Table 31: `RsslReactorDispatchOptions` Structure Members

6.5.1.2 RsslReactorDispatchOptions Utility Function

The Transport API provides the following utility function for use with the `RsslReactorDispatchOptions`.

FUNCTION NAME	DESCRIPTION
rsslClearReactorDispatchOptions	Clears the `RsslReactorDispatchOptions` structure. Useful for structure reuse.

Table 32: `RsslReactorDispatchOptions` Structure Members

6.5.1.3 rsslReactorDispatch Example

```
RsslReactorDispatchOptions dispatchOpts;

/* Set dispatching options. */
rsslClearReactorDispatchOptions(&dispatchOpts);
dispatchOpts.maxMessages = 200;

/* Call rsslReactorDispatch(). It will keep dispatching events until there is nothing to read or
 * maxMessages is reached. */
ret = rsslReactorDispatch(pReactor, &dispatchOpts, &rsslErrorInfo);
```

Code Example 6: `rsslReactorDispatch` Example

6.5.2 Reactor Callback Functions

Information about the state of the **`RsslReactorChannel`** connection as well as any messages for that channel are returned to the application via a series of callback functions. Each **`RsslReactorChannel`** can define its own unique callback functions or specify callback functions that can be shared across several or all channels.

There are several values that can be returned from a callback function implementation. These can trigger specific **`RsslReactor`** behavior based on the outcome of the callback function. The callback return values are described in the following table.

RETURN CODE	DESCRIPTION
RSSL_RC_CRET_SUCCESS	Indicates that the callback function was successful and the message or event has been handled.
RSSL_RC_CRET_FAILURE	Indicates that the message or event has failed to be handled. Returning this code from any callback function will cause the **`RsslReactor`** to shutdown.
RSSL_RC_CRET_RAISE	Can be returned from any domain-specific callback (e.g., **`RsslRDMLoginMsgCallback`**). This will cause the **`RsslReactor`** to invoke the **`RsslDefaultMsgCallback`** for this message upon the domain-specific callbacks return.

Table 33: `RsslReactorCallbackRet` Callback Return Codes

6.5.3 Reactor Callback: Channel Event

The **`RsslReactor`** channel event callback is used to communicate **`RsslReactorChannel`** and connection state information to the application. This callback function has the following prototype:

```
RsslReactorChannelEventCallback(RsslReactor*, RsslReactorChannel*, RsslReactorChannelEvent*)
```

When invoked, this will return the **`RsslReactor`** and the **`RsslReactorChannel`** that the event has occurred on. In addition, an **`RsslReactorChannelEvent`** structure is returned, containing more information about the event information.

6.5.3.1 Reactor Channel Event

The **`RsslReactorChannelEvent`** is returned to the application via the **`RsslReactorChannelEventCallback`**.

STRUCTURE MEMBER	DESCRIPTION
channelEventType	The type of event that has occurred on the **`RsslReactorChannel`**. For a list of enumeration values, refer to Section 6.5.3.2.
pReactorChannel	The **`RsslReactorChannel`** on which the event occurred.
pError	An **`RsslErrorInfo`** structure that is populated with error and warning information that occurred. This is only populated for **RSSL_RC_CET_CHANNEL_DOWN** and **RSSL_RC_CET_WARNING** event types.

Table 34: `RsslReactorChannelEvent` Structure Members

6.5.3.2 Reactor Channel Event Type Enumeration Values

FLAG ENUMERATION	MEANING
RSSL_RC_CET_INIT	Channel event initialization value. This should not be used by the application or returned to the application.
RSSL_RC_CET_CHANNEL_UP	Indicates that the `RsslReactorChannel` has been successfully initialized and can be directly dispatched on. Where applicable, any specified Login, Directory, or Dictionary messages will now be exchanged by the `RsslReactor`.
RSSL_RC_CET_CHANNEL_DOWN	Indicates that the `RsslReactorChannel` is not available for use. This could be a result of an initialization failure, a ping timeout, or some other kind of connection related issue. The `RsslErrorInfo` will contain more detailed information about what occurred. The application should call `rsslReactorCloseChannel` to clean up the failed `RsslReactorChannel`.
RSSL_RC_CET_CHANNEL_DOWN_ RECONNECTING	Indicates that the `RsslReactorChannel` is temporarily unavailable for use. The Reactor will attempt to reconnect the channel according to the values specified in `RsslReactorConnectOptions` when `rsslReactorConnect` was called. If the watchlist is enabled, requests are recovered as appropriate when the channel successfully reconnects. Before exiting the `channelEventCallback`, the application should release any resources associated with the channel, such as `RsslBuffers`, and remove its file descriptor, if valid, from any notification sets.
RSSL_RC_CET_CHANNEL_READY	Indicates that the `RsslReactorChannel` has successfully completed any necessary initialization process. Where applicable, this includes exchange of any provided Login, Directory, or Dictionary content. The application should now be able to consume or provide any content.
RSSL_RC_CET_WARNING	Indicates that the `RsslReactorChannel` has experienced an event that did not result in connection failure, but may require the attention of the application. The `RsslErrorInfo` will contain more detailed information about what occurred.
RSSL_RC_CET_FD_CHANGE	Indicates that a file-descriptor change occurred on the `RsslReactorChannel`. If the application is using its own I/O notification mechanism, it should replace the `oldSocketId` with the `socketId`, both of which can be found on the `RsslReactorChannel`.
RSSL_RC_CET_CHANNEL_OPEN	This event occurs only when the watchlist is enabled and only via the optional `channelOpenCallback` function. Indicates that a channel has been created via `rsslReactorConnect`. Though the channel is still not ready to be dispatched, the application can begin submitting request messages which will be sent when the channel successfully initializes.

Table 35: `RsslReactorChannelEventType` Enumeration Values

6.5.3.3 Reactor Channel Event Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearReactorChannelEvent	Clears an `RsslReactorChannelEvent` structure.

Table 36: `RsslReactorChannelEvent` Utility Functions

6.5.3.4 Reactor Channel Event Callback Example

```
RsslReactorCallbackRet channelEventCallback(RsslReactor *pReactor, RsslReactorChannel
        *pReactorChannel, RsslReactorChannelEvent *pChannelEvent)
{
    switch(pChannelEvent->channelEventType)
    {
        case RSSL_RC_CET_CHANNEL_UP:
            /* Channel has successfully initialized, add its descriptors to our notification
                    mechanism. */
            FD_SET(pReactorChannel->socketId, &readFds);
            FD_SET(pReactorChannel->socketId, &exceptFds);
        break;
        case RSSL_RC_CET_CHANNEL_DOWN:
            /* Channel has failed. Clean up all references and close the channel. */
            FD_CLR(pReactorChannel->socketId, &readFds);
            FD_CLR(pReactorChannel->socketId, &exceptFds);

            /* If all references are already clean up, channel can be closed now. Otherwise the
            * application can wait for a more appropriate time. */
            ret = rsslReactorCloseChannel(pReactor, pReactorChannel, &rsslErrorInfo);
        break;

        case RSSL_RC_CET_CHANNEL_READY:
            /* Channel has exchanged its initial messages(if any were provided on the role object)
            * and is ready for use. */
            sendItemRequests(pReactorChannel);
        break;

        case RSSL_RC_CET_FD_CHANGE:
            /* The descriptor representing this channel has changed. Normally the application only needs
            * to update its notification mechanism in response to this event. */
            FD_CLR(pReactorChannel->oldSocketId, &readFds);
            FD_CLR(pReactorChannel->oldSocketId, &exceptFds);
            FD_SET(pReactorChannel->socketId, &readFds);
            FD_SET(pReactorChannel->socketId, &exceptFds);
        break;

        case RSSL_RC_CET_WARNING:
            /* Received a warning about the channel. The channel is still active, but the event may
             * require the application's attention. */
            printf("Received channel warning event: %d(%s) ",
                    pChannelEvent->pError->rsslError.rsslErrorId,
                    pChannelEvent->pError->rsslError.text);
        break;

    }
    return RSSL_RC_CRET_SUCCESS;
}
```

Code Example 7: Reactor Channel Event Callback Example

6.5.4 Reactor Callback: Default Message

The `RsslReactor` default message callback communicates all received content that is not handled directly by a domain-specific callback function. This callback will also be invoked after any domain-specific callback that returns the **RSSL_RC_CET_RAISE** value. This callback function has the following prototype:

```
RsslDefaultMsgCallback(RsslReactor*, RsslReactorChannel*, RsslMsgEvent*)
```

When invoked, this will return the `RsslReactor` and the `RsslReactorChannel` that the event has occurred on. In addition, an `RsslMsgEvent` structure is returned, containing more information about the event information.

6.5.4.1 Reactor Message Event

The `RsslMsgEvent` is returned to the application via the `RsslDefaultMsgCallback`.

STRUCTURE MEMBER	DESCRIPTION
pRsslMsgBuffer	An `RsslBuffer` containing the raw, undecoded message that was read and processed by the callback. **Note:** When the consumer watchlist is enabled, this is not provided, as the message present might not match this buffer, or the message might be internally generated.
pRsslMsg	An `RsslMsg` structure that has been populated with the message content by calling `rsslDecodeMsg`. If not present, an error was encountered while processing the information. **Note:** When the consumer watchlist is enabled, this is not provided to callback functions that provide `RsslRDMMsg`s.
pError	An `RsslErrorInfo` structure that is populated with error and warning information that occurred, likely related to message decoding or processing.
pStreamInfo	Any information associated with a stream (only when the watchlist is enabled).
pSeqNum	The sequence number associated with a message, if present (only when using multicast).
pFTGroupId	The fault-tolerant group associated with a message, if present (only when using multicast).

Table 37: `RsslMsgEvent` Structure Members

6.5.4.2 Reactor Message Event Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearMsgEvent	Clears an `RsslMsgEvent` structure.

Table 38: `RsslMsgEvent` Utility Functions

6.5.4.3 Reactor Message Event Callback Example

```
RsslReactorCallbackRet defaultMsgCallback(RsslReactor *pReactor, RsslReactorChannel *pReactorChannel,
        RsslMsgEvent *pMsgEvent)
{
    RsslMsg *pRsslMsg = pMsgEvent->pRsslMsg;

    /* Received an RsslMsg --- or, if the decode failed, an error. */
    /* The RsslMsg will have already been passed through rsslDecodeMsg. Only the payload requires
            additional decoding */
    if (pRsslMsg)
        processRsslMsg(pRsslMsg);
    else
        printf("Error: %s(%s)\n", pMsgEvent->pErrorInfo->rsslError.text,
                pMsgEvent->pErrorInfo->errorLocation);
}
```

Code Example 8: Reactor Message Event Callback Example

6.5.5 Reactor Callback: RDM Login Message

The **`RsslReactor`** RDM Login Message callback is used to communicate all received RDM Login messages. This callback function has the following prototype:

```
RsslRDMLoginMsgCallback(RsslReactor*, RsslReactorChannel*, RsslRDMLoginMsgEvent*)
```

When invoked, this will return the **`RsslReactor`** and the **`RsslReactorChannel`** that the event has occurred on. In addition, an **`RsslRDMLoginMsgEvent`** structure is returned, containing more information about the event information.

6.5.5.1 Reactor RDM Login Message Event

The **`RsslRDMLoginMsgEvent`** is returned to the application via the **`RsslRDMLoginMsgCallback`**.

STRUCTURE MEMBER	DESCRIPTION
baseMsgEvent	An **`RsslMsgEvent`** populated with the raw buffer, **`RsslMsg`**, and any error information. This structure is defined in Section 6.5.4.1.
pRDMLoginMsg	The RDM representation of the decoded Login message. If not present, an error was encountered while processing the information. This message is presented as the **`RsslRDMLoginMsg`**, described in Section 7.3.

Table 39: `RsslRDMLoginMsgEvent` Structure Members

6.5.5.2 Reactor RDM Login Message Event Utility Function

FUNCTION NAME	DESCRIPTION
rsslClearRDMLoginMsgEvent	Clears an **`RsslRDMLoginMsgEvent`** structure.

Table 40: `RsslRDMLoginMsgEvent` Utility Function

6.5.5.3 Reactor RDM Login Message Event Callback Example

```
RsslReactorCallbackRet loginMsgCallback(RsslReactor *pReactor, RsslReactorChannel *pReactorChannel,
        RsslRDMLoginMsgEvent *pLoginMsgEvent)
{
    RsslRDMLoginMsg *pLoginMsg = pLoginMsgEvent->pRDMLoginMsg;

    /* Received an RsslRDMLoginMsg --- or, if the decode failed, an error. */
    /* The login message will already be fully decoded */
    if (pLoginMsg)
    {
        switch(pLoginMsg->rdmMsgBase.rdmMsgType)
        {
            case RDM_LG_MT_REFRESH:
                RsslRDMLoginRefresh *pRefresh = &pLoginMsg->refresh;
                    break;
            case RDM_LG_MT_STATUS:
                RsslRDMLoginStatus *pStatus = &pLoginMsg->status;
                    break;
            default:
                printf("Received unhandled login message.\n"); break;
        }
    }
    else
        printf("Error: %s(%s)\n", pLoginMsgEvent->baseMsgEvent.pErrorInfo->rsslError.text,
                pLoginMsgEvent->baseMsgEvent.pErrorInfo->errorLocation);
}
```

Code Example 9: Reactor RDM Login Message Event Callback Example

6.5.6 Reactor Callback: RDM Directory Message

The **`RsslReactor`** RDM Directory Message callback is used to communicate all received RDM Directory messages. This callback function has the following prototype:

```
RsslRDMDirectoryMsgCallback(RsslReactor*, RsslReactorChannel*, RsslRDMDirectoryMsgEvent*)
```

When invoked, this will return the **`RsslReactor`** and the **`RsslReactorChannel`** that the event has occurred on. In addition, an **`RsslRDMDirectoryMsgEvent`** structure is returned, which contains more information about the event information.

6.5.6.1 Reactor RDM Directory Message Event

The `RsslRDMDirectoryMsgEvent` is returned to the application via the `RsslRDMDirectoryMsgCallback`.

STRUCTURE MEMBER	DESCRIPTION
baseMsgEvent	An `RsslMsgEvent` populated with the raw buffer, `RsslMsg`, and any error information. This structure is defined in Section 6.5.4.1.
pRDMDirectoryMsg	The RDM representation of the decoded Source Directory message. If not present, an error was encountered while processing the information. This message is presented as the `RsslRDMDirectoryMsg`, described in Section 7.4.

Table 41: `RsslRDMDirectoryMsgEvent` Structure Members

6.5.6.2 Reactor RDM Directory Message Event Utility Function

FUNCTION NAME	DESCRIPTION
rsslClearRDMDirectoryMsgEvent	Clears an `RsslRDMDirectoryMsgEvent` structure.

Table 42: `RsslRDMDirectoryMsgEvent` Utility Function

6.5.6.3 Reactor RDM Directory Message Event Callback Example

```
RsslReactorCallbackRet directoryMsgCallback(RsslReactor *pReactor, RsslReactorChannel
        *pReactorChannel, RsslRDMDirectoryMsgEvent *pDirectoryMsgEvent)
{
    RsslRDMDirectoryMsg *pDirectoryMsg = pDirectoryMsgEvent->pRDMDirectoryMsg;

    /* Received an RsslRDMDirectoryMsg --- or, if the decode failed, an error. */
    /* The directory message will already be fully decoded */
    if (pDirectoryMsg)
    {
        switch(pDirectoryMsg->rdmMsgBase.rdmMsgType)
        {
            case RDM_DR_MT_REFRESH:
                RsslRDMDirectoryRefresh *pRefresh = &pDirectoryMsg->refresh;
                    break;
            case RDM_DR_MT_UPDATE:
                RsslRDMDirectoryUpdate *pUpdate = &pDirectoryMsg->update;
                    break;
            case RDM_DR_MT_STATUS:
                RsslRDMDirectoryStatus *pStatus = &pDirectoryMsg->status;
                    break;
            default:
                printf("Received unhandled directory message.\n");
        }
    }
    else
        printf("Error: %s(%s)\n", pDirectoryMsgEvent->baseMsgEvent.pErrorInfo->rsslError.text,
            pDirectoryMsgEvent->baseMsgEvent.pErrorInfo->errorLocation);
}
```

Code Example 10: Reactor RDM Directory Message Event Callback Example

6.5.7 Reactor Callback: RDM Dictionary Message

The `RsslReactor` RDM Dictionary Message callback is used to communicate all received RDM Dictionary messages. This callback function has the following prototype:

```
RsslRDMDictionaryMsgCallback(RsslReactor*, RsslReactorChannel*, RsslRDMDictionaryMsgEvent*)
```

When invoked, this will return the `RsslReactor` and the `RsslReactorChannel` that the event has occurred on. In addition, an `RsslRDMDictionaryMsgEvent` structure is returned, containing more information about the event information.

6.5.7.1 Reactor RDM Dictionary Message Event

The `RsslRDMDictionaryMsgEvent` is returned to the application via the `RsslRDMDictionaryMsgCallback`.

STRUCTURE MEMBER	DESCRIPTION
baseMsgEvent	An `RsslMsgEvent` populated with the raw buffer, `RsslMsg`, and any error information. This structure is defined in Section 6.5.4.1.
pRDMDictionaryMsg	The RDM representation of the decoded Dictionary message. If not present, an error was encountered while processing the information. This message is presented as the `RsslRDMDictionaryMsg`, described in Section 7.5.

Table 43: `RsslRDMDictionaryMsgEvent` Structure Members

6.5.7.2 Reactor RDM Dictionary Message Event Utility Function

FUNCTION NAME	DESCRIPTION
rsslClearRDMDictionaryMsgEvent	Clears an `RsslRDMDictionaryMsgEvent` structure.

Table 44: `RsslRDMDictionaryMsgEvent` Utility Function

6.5.7.3 Reactor RDM Dictionary Message Event Callback Example

```
RsslReactorCallbackRet dictionaryMsgCallback(RsslReactor *pReactor, RsslReactorChannel
        *pReactorChannel, RsslRDMDictionaryMsgEvent *pDictionaryMsgEvent)
{
    RsslRDMDictionaryMsg *pDictionaryMsg = pDictionaryMsgEvent->pRDMDictionaryMsg;

    /* Received an RsslRDMDictionaryMsg --- or, if the decode failed, an error. */
    if (pDictionaryMsg)
    {
        switch(pDictionaryMsg->rdmMsgBase.rdmMsgType)
        {
            case RDM_DC_MT_REFRESH:
                RsslRDMDictionaryRefresh *pRefresh = &pDictionaryMsg->refresh;
                    break;
            case RDM_DC_MT_STATUS:
                RsslRDMDictionaryStatus *pStatus = &pDictionaryMsg->status;
                    break;
            default:
                printf("Received unhandled dictionary message.\n");
        }
    }
    else
        printf("Error: %s(%s)\n", pDictionaryMsgEvent->baseMsgEvent.pErrorInfo->rsslError.text,
            pDictionaryMsgEvent->baseMsgEvent.pErrorInfo->errorLocation);
}
```

Code Example 11: Reactor RDM Dictionary Message Event Callback Example

6.6 Writing Data

The Transport API Reactor helps streamline the high performance writing of content. The **`RsslReactor`** flushes content to the network so the application does not need to. The **`RsslReactor`** does so through the use of a separate worker thread that becomes active whenever there is queued content that needs to be passed to the connection.

The Transport API Reactor offers two methods for writing content: **`rsslReactorSubmitMsg`** and **`rsslReactorSubmit`**. When writing applications to the Reactor, consider which is most appropriate for your needs:

rsslReactorSubmitMsg

- Takes an **`RsslMsg`** structure as part of its options; does not require retrieval of an **`RsslBuffer`** from the channel.
- Must be used when the consumer watchlist is enabled.

rsslReactorSubmit

- Takes an **`RsslBuffer`** which the application retrieves from the channel.
- More efficient: the application encodes directly into the buffer, and can use buffer packing.
- Cannot be used when the consumer watchlist is enabled.

6.6.1 Writing Data using rsslReactorSubmitMsg()

`rsslReactorSubmitMsg` provides a simple interface for writing **`RsslMsg`**s. To send a message, the application populates an RsslMsg structure, sets it (along with any other desired options) on an **`RsslReactorSubmitMsgOptions`** structure, and calls **`rsslReactorSubmitMsg`** with the structure.

A buffer is not needed to use **`rsslReactorSubmitMsg`**. If the application needs to include any encoded content, it can encode the content into any available memory, and set the appropriate member of the **`RsslMsg`** to point to the memory (as well as set the length of the encoded content).

6.6.1.1 rsslReactorSubmitMsg Function

FUNCTION NAME	DESCRIPTION
rsslReactorSubmitMsg	Encodes and submits an **`RsslMsg`** to the Reactor. This function expects a properly populated **`RsslMsg`**. This function allows for several modifications and additional parameters to be specified via the **`RsslReactorSubmitMsgOptions`** structure.

Table 45: `rsslReactorSubmit` Function

6.6.1.2 Reactor Submit Message Options

Using **`RsslReactorSubmitMsgOptions`**, the application can control various aspects of the call to **`rsslReactorSubmitMsg`**.

STRUCTURE MEMBER	DESCRIPTION
pRsslMsg	The **`RsslMsg`** structure to submit. Use only one instance of either pRsslMsg or pRDMMsg.
pRDMMsg	The **`RsslRDMMsg`** structure to submit. Use only one instance of either pRsslMsg or pRDMMsg.
pServiceName	The application may use this in place of the **`serviceId`** member specified on the **`RsslMsgKey`** of an **`RsslMsg`**. When used to open streams via request messages, the **`RsslReactor`** will recover using this service name. When used for other message types such as **`RsslPostMsg`** or **`RsslGenericMsg`**, the **`RsslReactor`** converts the name to its corresponding ID before writing the message. **Note:** This option is only supported when the watchlist is enabled.
requestMsgOptions	Provides additional functionality that may be used when using **`RsslRequestMsgs`** to send requests.
majorVersion	The RWF major version of any encoded content in the message.
minorVersion	The RWF minor version of any encoded content in the message.

Table 46: `RsslReactorSubmitMsgOptions` Structure Members

6.6.1.3 RsslReactorRequestMsgOptions

The **`RsslReactorRequestMsgOptions`** provide additional functionality when requesting items. These options are only available when the watchlist is enabled.

STRUCTURE MEMBER	DESCRIPTION
pUserSpec	A user-specified pointer that will be associated with the stream. This pointer will be provided in responses to this stream via the **`RsslStreamInfo`** provided with each message event.

Table 47: `RsslReactorSubmitOptions` Structure Members

6.6.1.4 RsslReactorSubmitMsgOptions Utility Function

The Transport API provides the following utility function for use with **`RsslReactorSubmitMsgOptions`**.

FUNCTION NAME	DESCRIPTION

Table 48: `RsslReactorSubmitOptions` Utility Function

6.6.1.5 rsslReactorSubmitMsg Return Codes

The following table defines the return codes that can occur when using `rsslReactorSubmitMsg`.

RETURN CODE	DESCRIPTION
RSSL_RET_SUCCESS	Indicates that the `rsslReactorSubmitMsg` function has succeeded.
RSSL_RET_BUFFER_NO_BUFFERS	Indicates that the message was currently unable to be written due to a lack of available pool buffers. The application can try to submit the message later, or it can optionally use `rsslReactorChannelIoctl` to increase the number of available pool buffers and try again.
RSSL_RET_FAILURE	Indicates that a general failure has occurred and the message was not submitted. The `RsslErrorInfo` structure passed to the function will contain more details.

Table 49: `rsslReactorSubmitMsg` Return Codes

6.6.1.6 rsslReactorSubmitMsg Example

The following example shows typical use of `rsslReactorSubmitMsg`.

```
RsslMsg requestMsg;
RsslReactorSubmitMsgOptions opts;
RsslErrorInfo errorInfo;
RsslRet ret;

rsslClearRequestMsg(&requestMsg);
requestMsg.msgBase.streamId = 2;
requestMsg.msgBase.domainType = RSSL_DMT_MARKET_PRICE;
requestMsg.msgBase.containerType = RSSL_DT_NO_DATA;
requestMsg.flags = RSSL_RQMF_STREAMING | RSSL_RQMF_HAS_QOS;
requestMsg.qos.timeliness = RSSL_QOS_TIME_REALTIME;
requestMsg.qos.rate = RSSL_QOS_RATE_TICK_BY_TICK;
requestMsg.msgBase.msgKey.flags = RSSL_MKF_HAS_NAME | RSSL_MKF_HAS_SERVICE_ID;
requestMsg.msgBase.msgKey.name.data = “TRI.N”;
requestMsg.msgBase.msgKey.name.length = 5;
requestMsg.msgBase.msgKey.serviceId = 1;

rsslClearReactorSubmitMsgOptions(&opts);
opts.pRsslMsg = (RsslMsg*)&requestMsg;
ret = rsslReactorSubmitMsg(pReactor, pReactorChannel, &opts, &errorInfo);
```

Code Example 12: `rsslReactorSubmitMsg` Example

6.6.2 Writing data using rsslReactorSubmit()

The `rsslReactorSubmit` function offers efficient writing of data by using buffers retrieved directly from the Transport API transport buffer pool. It also provides additional features not normally available from `rsslReactorSubmitMsg`, such as buffer packing or the Transport API priority queue. When ready to send data, the application acquires a buffer from the Transport API pool. This allows the content to be encoded directly into the output buffer, reducing the number of times the content needs to be copied. Once content is encoded and the buffer is properly populated, the application can submit the data to the Reactor. The Transport API will ensure that successfully submitted buffers reach the network. Applications can also pack multiple messages into a single buffer by following a similar process as described above, however instead of getting a new buffer for each message the application uses the Reactor's pack function instead. The following flow chart depicts the typical write process.

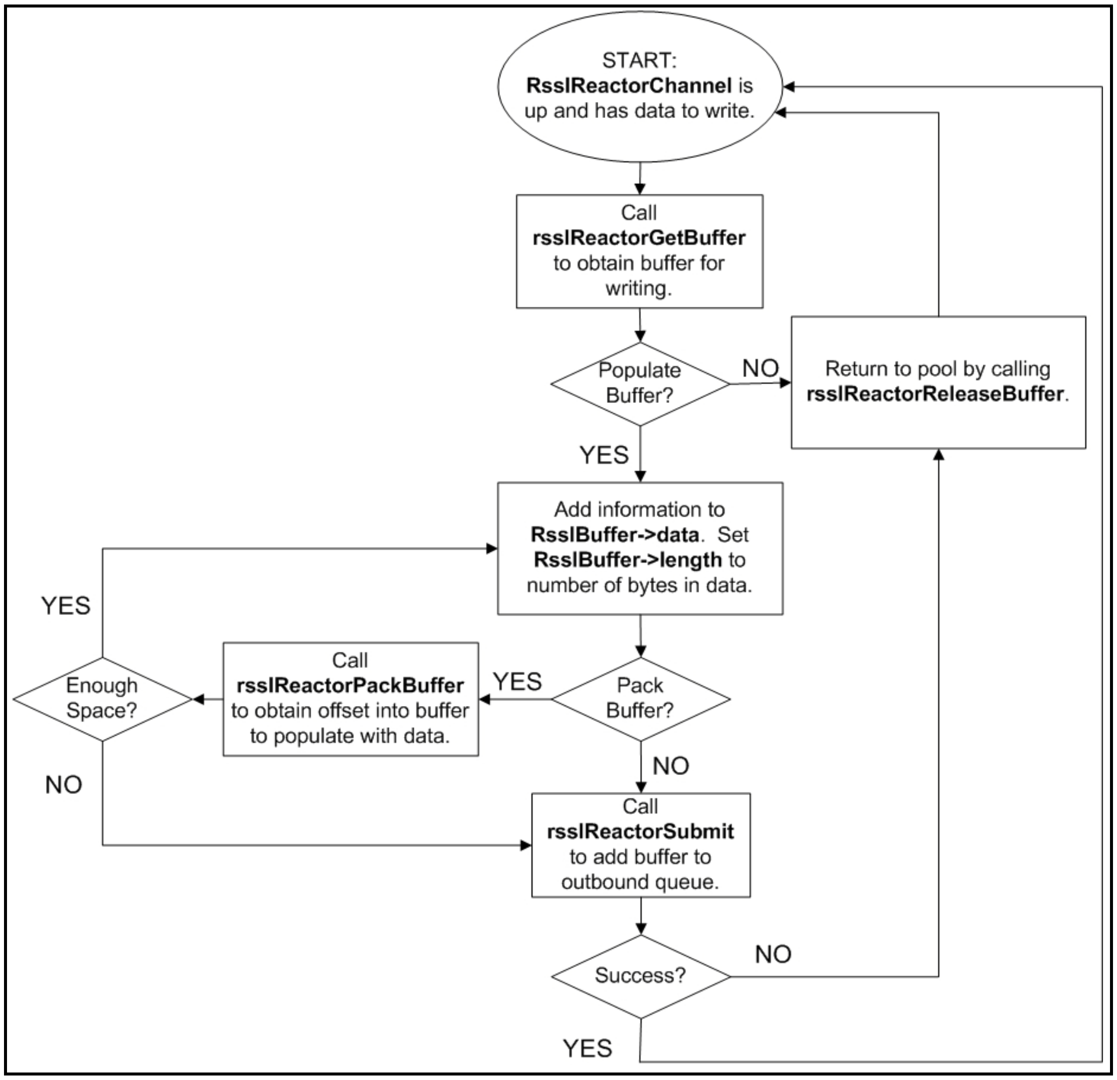

START: RsslReactorChannel is up and has data to write.
Call rsslReactorGetBuffer to obtain buffer for writing.
Populate Buffer?
NO
Return to pool by calling rsslReactorReleaseBuffer.
YES
Add information to RsslBuffer->data. Set RsslBuffer->length to number of bytes in data.
YES
Enough Space?
Call rsslReactorPackBuffer to obtain offset into buffer to populate with data.
YES
Pack Buffer?
NO
NO
Call rsslReactorSubmit to add buffer to outbound queue.
Success?
NO
YES

Figure 8. Flow Chart for writing data via `rsslReactorSubmit`

6.6.2.1 Obtaining a Buffer: Overview

Before information can be submitted, the user is required to obtain a buffer from the internal Transport API buffer pool, as described in the *Transport API C Edition Developers Guide*. After a buffer is acquired, the user can populate the `RsslBuffer.data` and set the `RsslBuffer.length` to the number of bytes referred to by `data`. If the buffer is not used or the `rsslReactorSubmit` function call fails, the buffer must be released back into the pool to ensure proper reuse and cleanup. If the buffer is successfully passed to `rsslReactorSubmit`, the Transport API Reactor will return the buffer to the pool.

The number of buffers made available to an `RsslReactorChannel` is configurable through the `RsslReactorConnectOptions` or `RsslReactorAcceptOptions`. For more information about available `rsslReactorConnect` and `rsslReactorAccept` options, refer to Section 6.4.1.2 and Section 6.4.1.7.

6.6.2.2 Obtaining a Buffer: Buffer Management Functions

FUNCTION NAME	DESCRIPTION
rsslReactorGetBuffer	Obtains a buffer of the requested size from the buffer pool. When the `RsslBuffer` is returned, the `length` member indicates the number of bytes available in the buffer (this should match the amount the application requested). When populating, it is required that the application set `length` to the number of bytes actually used. This ensures that only the required bytes are written to the network. If the requested size is larger than the `maxFragmentSize`, the transport will create and return the buffer to the user. When written, this buffer will be fragmented by the `rsslReactorSubmit` function (for further details, refer to Section 6.6.2.4). Because of some additional book keeping required when packing, the application must specify whether a buffer should be ‘packable’ when calling `rsslReactorGetBuffer`. For more information on packing, refer to Section 6.6.2.9. For performance purposes, an application is not permitted to request a buffer larger than `maxFragmentSize` and have the buffer be ‘packable.’ If the buffer is not used or the `rsslReactorSubmit` call fails, the buffer must be returned to the pool using `rsslReactorReleaseBuffer`. If the `rsslReactorSubmit` call is successful, the buffer will be returned to the correct pool by the transport. This function calls the Transport API `rsslGetBuffer` function which has its use and return values described in the *Transport API C Edition Developers Guide*.
rsslReactorReleaseBuffer	Releases a buffer back to the correct pool. This should only be called with buffers that originate from `rsslReactorGetBuffer` and are not successfully passed to `rsslReactorSubmit`. This function calls the Transport API `rsslReleaseBuffer` function which has its use and return values described in the *Transport API C Edition Developers Guide*.
rsslReactorChannelBufferUsage	Returns the number of buffers currently in use by the `RsslReactorChannel`, this includes buffers that the application holds and buffers internally queued and waiting to be flushed to the connection by the `RsslReactor`. This function calls the Transport API `rsslBufferUsage` function which has its use and return values described in the *Transport API C Edition Developers Guide*.

Table 50: Reactor Buffer Management Functions

6.6.2.3 Obtaining a Buffer: rsslReactorGetBuffer Return Values

The following table defines return and error code values that can occur while using `rsslReactorGetBuffer`.

RETURN CODE	DESCRIPTION
Valid buffer returned Success Case	An `RsslBuffer` is returned to the user. The `RsslBuffer.length` indicates the number of bytes available to populate and the `RsslBuffer.data` provides a starting location for population.

Table 51: `rsslReactorGetBuffer` Return Values

RETURN CODE	DESCRIPTION
NULL buffer returned Error Code: RSSL_RET_BUFFER_NO_BUFFERS	NULL is returned to the user. This value indicates that there are no buffers available to the user. See `RsslErrorInfo` content for more details. This typically occurs because all available buffers are queued and pending flushing to the connection. The `rsslReactorIoctl` function can be used to increase the number of `guaranteedOutputBuffers` (for details, refer to Section 6.8).
NULL buffer returned Error Code: RSSL_RET_FAILURE	NULL is returned to the user. This value indicates that some type of general failure has occurred. The `RsslChannel` should be closed.
NULL buffer returned Error Code: RSSL_RET_INIT_NOT_INITIALIZED	Indicates that the underlying RSSL Transport has not been initialized. See the `RsslErrorInfo` content for more details.

Table 51: `rsslReactorGetBuffer` Return Values (Continued)

6.6.2.4 Writing Data: Overview

After an `RsslBuffer` is obtained from `rsslReactorGetBuffer` and populated with the user's data, the buffer can be passed to the `rsslReactorSubmit` function. This function manages queuing and flushing of user content. It will also perform any fragmentation or compression. If an unrecoverable error occurs, any `RsslBuffer` that has not been successfully passed to `rsslReactorSubmit` should be released to the pool using `rsslReactorReleaseBuffer`. The following table describes the `rsslReactorSubmit` function as well as some additional parameters associated with it.

6.6.2.5 Writing Data: rsslReactorSubmit Function

Note: Before passing a buffer to `rsslReactorSubmit`, it is required that the application set `length` to the number of bytes actually used. This ensures that only the required bytes are written to the network.

FUNCTION NAME	DESCRIPTION
rsslReactorSubmit	Performs writing of data. This function expects the buffer to be properly populated, where length reflects the actual number of bytes used. This function calls the Transport API `rsslWrite` function and also triggers the `rsslFlush` function (both are described in the *Transport API C Edition Developers Guide*). This function allows for several modifications and additional parameters to be specified via the `RsslReactorSubmitOptions` structure, defined in Section 6.6.2.6. For a list of return values, refer to Section 6.6.2.8.

Table 52: `rsslReactorSubmit` Function

6.6.2.6 Writing Data: Reactor Submit Options

The application uses `RsslReactorSubmitOptions` to control various aspects of the call to `rsslReactorSubmit`.

STRUCTURE MEMBER	DESCRIPTION
Priority	Controls the priority at which the data will be written. Valid priorities are • **RSSL_HIGH_PRIORITY** • **RSSL_MEDIUM_PRIORITY** • **RSSL_LOW_PRIORITY** More information about write priorities, including an example scenario, are available in the *Transport API C Edition Developers Guide*.
writeFlags	Flag values that allow the application to modify the behavior of this rsslReactorSubmit call. This includes options to bypass queuing or compression. More information about the specific flag values are available in the *Transport API C Edition Developers Guide*.
pBytesWritten	If specified, will return the number of bytes to be written, including any transport header overhead and taking into account any savings from compression.
pUncompressedBytesWritten	If specified, will return the number of bytes to be written, including any transport header overhead but not taking into account any compression savings.

Table 53: `RsslReactorSubmitOptions` Structure Members

6.6.2.7 Writing Data: RsslReactorSubmitOptions Utility Function

The Transport API provides the following utility function for use with the **`RsslReactorDispatchOptions`**.

FUNCTION NAME	DESCRIPTION
rsslClearReactorSubmitOptions	Clears the **`RsslReactorSubmitOptions`** structure. Useful for structure reuse.

Table 54: `RsslReactorSubmitOptions` Utility Function

6.6.2.8 Writing Data: rsslReactorSubmit Return Codes

The following table defines the return codes that can occur when using **`rsslReactorSubmit`**.

RETURN CODE	DESCRIPTION
RSSL_RET_SUCCESS	Indicates that the **`rsslReactorSubmit`** function has succeeded. The **`RsslBuffer`** will be released by the Transport API Reactor.
RSSL_RET_WRITE_CALL_AGAIN	Indicates that a large buffer could not be fully written with this **`rsslReactorSubmit`** call. This is typically due to all pool buffers becoming unavailable. The **`RsslReactor`** will flush for the user in order to free up buffers. The application can optionally use **`rsslReactorIoctl`** to increase the number of available pool buffers. After pool buffers become available again, the same buffer should be used to call **`rsslReactorSubmit`** an additional time (the same priority level must be used to ensure proper ordering of each fragment). This will continue the fragmentation process from where it left off. If the application does not subsequently pass the buffer to **`rsslReactorSubmit`**, the application should release it by calling **`rsslReactorReleaseBuffer`**.

Table 55: `rsslReactorSubmit` Return Codes

RETURN CODE	DESCRIPTION
RSSL_RET_FAILURE	Indicates that a general write failure has occurred. The `RsslReactorChannel` should be closed. The application should release the `RsslBuffer` by calling `rsslReactorReleaseBuffer`.

Table 55: `rsslReactorSubmit` Return Codes (Continued)

6.6.2.9 Packing Additional Data into a Buffer

If an application is writing many small buffers, it may be advantageous to combine the small buffers into one larger buffer. This can increase efficiency of the transport layer by reducing the overhead associated with each write operation, although it may add to the latency associated with each smaller buffer.

It is up to the writing application to determine when to stop packing, and the mechanism used can vary greatly. One simple algorithm that can be used is to pack a fixed number of messages each time. A slightly more complex technique could use returned `RsslBuffer.length` to determine the amount of space remaining, and pack until the buffer is nearly full. Both of these mechanisms can introduce a variable amount of latency as they both depend on the rate of arrival of data (e.g. the packed buffer will not be written until enough data arrives to fill it). One method that can balance this would also employ a timer, used to limit the amount of time a packed buffer is held. If the buffer is full prior to the timer expiring, the data is written however when the timer expires the buffer will be written regardless of the amount of data it contains. This can help to limit the latency introduced and hold it to a maximum amount associated with the duration of the timer.

FUNCTION NAME	DESCRIPTION
rsslReactorPackBuffer	Packs contents of passed in `RsslBuffer` and returns a new `RsslBuffer` to continue packing new data into. For a buffer to allow packing, it must be requested from `rsslReactorGetBuffer` as ‘packable’ and cannot exceed the `maxFragmentSize`. The returned buffer provides a `data` pointer for populating and the `length` conveys number of bytes available in the buffer. An application can use the `RsslBuffer.length` to determine the amount of space available to continue packing buffers into. After each buffer is populated, the length should be set to reflect the actual number of bytes contained in the buffer. This will ensure that only the necessary space is reserved while packing. `rsslReactorPackBuffer` return values are defined in Section 6.6.2.10. This function calls the Transport API `rsslPackBuffer` function which has its use and return values described in the *Transport API C Edition Developers Guide*.

Table 56: `rsslReactorPackBuffer` Function

6.6.2.10 rsslReactorPackBuffer Return Values

The following table defines return and error code values that can occur when using `rsslReactorPackBuffer`.

RETURN CODE	DESCRIPTION
Valid buffer returned Success Case	An `RsslBuffer` is returned to the user. The `RsslBuffer.length` indicates the number of bytes available to populate and the `RsslBuffer.data` provides a starting location for population.
NULL buffer returned Error Code: RSSL_RET_FAILURE	NULL is returned to the user. This value indicates that some type of general failure has occurred. The `RsslReactorChannel` should be closed.
NULL buffer returned Error Code: RSSL_RET_INIT_NOT_INITIALIZED	Indicates that the underlying RSSL Transport has not been initialized. See the `RsslErrorInfo` content for more details.

Table 57: `rsslReactorPackBuffer` Return Values

6.6.2.11 Example: rsslReactorGetBuffer and rsslReactorSubmit Example

The following example shows typical use of `rsslReactorGetBuffer` and `rsslReactorSubmit`.

```
RsslBuffer *pMsgBuffer;
RsslEncodeIterator encodeIter;
RsslReactorSubmitOptions submitOpts;

pMsgBuffer = rsslReactorGetBuffer(pReactorChannel, 1024, RSSL_FALSE, &rsslErrorInfo);

rsslClearEncodeIterator(&encodeIter);
rsslSetEncodeIteratorRWFVersion(&encodeIter, pReactorChannel->majorVersion, pReactorChannel-
        >minorVersion);
rsslSetEncodeIteratorBuffer(&encodeIter, pMsgBuffer);
encodeMsgIntoBuffer(&encodeIter, pMsgBuffer);

pMsgBuffer->length = rsslGetEncodedBufferLength(&encodeIter);
rsslClearReactorSubmitOptions(&submitOpts);
ret = rsslReactorSubmit(pReactor, pReactorChannel, pMsgBuffer, &submitOpts, &rsslErrorInfo);
/* check return code */
switch (ret)
{
    case RSSL_RET_SUCCESS:
        /* successful write, nothing left to do */
        return 0;
    break;
    case RSSL_RET_FAILURE:
        /* an error occurred, need to release buffer */
        rsslReactorReleaseBuffer(pReactorChannel,pMsgBuffer, &rsslErrorInfo);
    break;
    case RSSL_RET_WRITE_CALL_AGAIN:
        /* large message couldn’t be fully written with one call, pass it to submit again */
        ret = rsslReactorSubmit(pReactor, pReactorChannel, pMsgBuffer, &rsslErrorInfo);
    break;
}
```

Code Example 13: Writing Data Using `rsslReactorSubmit`, `rsslReactorGetBuffer`, and `rsslReactor-ReleaseBuffer`

6.6.2.12 Example: rsslReactorGetBuffer, rsslReactorPackBuffer, and rsslReactorSubmit

The following example shows typical use of **`rsslReactorGetBuffer`**, **`rsslReactorPackBuffer`**, and **`rsslReactorSubmit`**.

```
RsslBuffer *pMsgBuffer;
RsslReactorSubmitOptions submitOpts;
RsslEncodeIterator encodeIter;

/* get a packable buffer */
pMsgBuffer = rsslReactorGetBuffer(pReactorChannel, 1024, RSSL_TRUE, &rsslErrorInfo);

rsslClearEncodeIterator(&encodeIter);
rsslSetEncodeIteratorRWFVersion(&encodeIter, pReactorChannel->majorVersion, pReactorChannel-
        >minorVersion);
rsslSetEncodeIteratorBuffer(&encodeIter, pMsgBuffer);
encodeMsgIntoBuffer(&encodeIter, pMsgBuffer);

/* pack first encoded message into buffer */
pMsgBuffer->length = rsslGetEncodedBufferLength(&encodeIter);
pMsgBuffer = rsslReactorPackBuffer(pReactorChannel, pMsgBuffer, &rsslErrorInfo);

rsslClearEncodeIterator(&encodeIter);
rsslSetEncodeIteratorRWFVersion(&encodeIter, pReactorChannel->majorVersion, pReactorChannel-
        >minorVersion);
rsslSetEncodeIteratorBuffer(&encodeIter, pMsgBuffer);
encodeMsgIntoBuffer(&encodeIter, pMsgBuffer);

/* pack second encoded message into buffer */
pMsgBuffer->length = rsslGetEncodedBufferLength(&encodeIter);
pMsgBuffer = rsslReactorPackBuffer(pReactorChannel, pMsgBuffer, &rsslErrorInfo);

rsslClearEncodeIterator(&encodeIter);
rsslSetEncodeIteratorRWFVersion(&encodeIter, pReactorChannel->majorVersion, pReactorChannel-
        >minorVersion);
rsslSetEncodeIteratorBuffer(&encodeIter, pMsgBuffer);

/* now write packed buffer by passing third buffer to rsslSubmit */
encodeMsgIntoBuffer(&encodeIter, pMsgBuffer);
pMsgBuffer->length = rsslGetEncodedBufferLength(&encodeIter);

rsslClearReactorSubmitOptions(&submitOpts);
ret = rsslReactorSubmit(pReactor, pReactorChannel, pMsgBuffer, &submitOpts, &rsslErrorInfo);
```

Code Example 14: Message Packing using `rsslReactorPackBuffer`

6.7 Creating and Using Tunnel Streams

The Reactor allows users to create and use special tunnel streams. A tunnel stream is a private stream that has additional behaviors associated with it, such as end-to-end line of sight for authentication and guaranteed delivery. Because tunnel streams are founded on the private streams concept, these will be established between consumer and provider endpoints and will pass through any intermediate components, such as TREP or EED.

The user creating the tunnel stream indicates the additional behaviors they want enforced, which is exchanged with the provider application end point. The provider end point acknowledges creation of the stream as well as the behaviors that it will also enforce on the stream. Once this is accomplished, the user is able to exchange any content they want, where the negotiated behaviors will be enforced on the content exchanged via the tunnel stream.

The tunnel stream allows for multiple substreams to exist, where substreams follow from the same the Transport API stream concept, however they flow and coexist within the confines of a tunnel stream.

In the following diagram, imagine the tunnel stream as the orange cylinder that connects the Consumer application and the Provider application. Notice that this passes directly through any intermediate components. The tunnel stream has end-to-end line of sight so the Provider and Consumer are effectively talking to each other directly, although they are traversing multiple devices in the system. Each of the black lines flowing through the cylinder represent a different substream, where each substream is its own independent stream of information. Each of these could be for different market content, for example one could be a Time Series request while another could be a request for Market Price content. A substream can also be a connection to a special provider application called a Queue Provider. A Queue Provider allows for persistence of content exchanged over the tunnel stream and substream, and can help provide the content beyond the end point visible to the consumer. To interact with a Queue Provider, additional addressing information is required. This is described in more detail in Section 7.6.

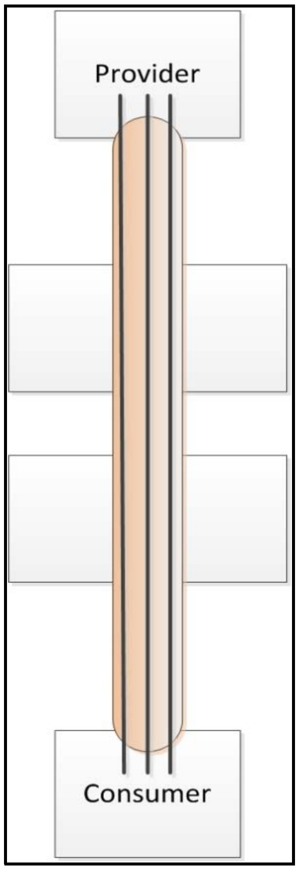


Figure 9. Tunnel Stream Illustration

6.7.1 Authenticating a Tunnel Stream

Providers might require the consumer to authenticate itself when establishing the tunnel stream. The type of authentication, if any, is given by the `RsslClassOfService.authentication.type`. For more information, refer to Section 6.7.3.

The `RsslClassOfService.authentication.type` may be set to **RDM_COS_AU_OMM_LOGIN**. When an OMM Consumer expects this type of authentication, it should set an `RsslRDMLoginRequest` message on the `RsslTunnelStreamOpenOptions.pAuthLoginRequest` member. If the OMM Consumer application does not provide it, the API will use the login request provided on the `RsslReactorOMMConsumerRole.pLoginRequest` when the consumer connected (refer to Section 6.3.2). The consumer must provide one of these for authentication of this type.

The login request will be sent to the provider. When the provider sends a Login response to complete the authentication, the `RsslTunnelStreamStatusEvent` event given to the consumer will include an `RsslTunnelStreamAuthInfo` structure with more details. OMM Provider applications will see the login request as a normal message within the `RsslTunnelStream` and should respond with a login response message via `rsslTunnelStreamSubmit` or `rsslTunnelStreamSubmitMsg`.

Other types of authentication might be specified, but must be performed by both the Provider and Consumer applications by submitting normal `RsslTunnelStream` messages via `rsslTunnelStreamSubmit` or `rsslTunnelStreamSubmitMsg`.

The `RsslTunnelStreamAuthInfo` structure contains the following member:

MEMBER	DESCRIPTION
pLoginMsg	The Login message sent by the tunnel stream's provider application, which resulted in this event.

Table 58: `RsslTunnelStreamAuthInfo` Structure Members

6.7.2 Opening a Tunnel Stream

The user can create one or more tunnel streams and associate them with any `RsslReactorChannel`. This will begin the private stream connection as well negotiation of any specified behaviors. Prior to opening a tunnel stream, it is required to implement the `RsslTunnelStreamStatusEventCallback`, which is described in Section 6.7.4.

6.7.2.1 rsslReactorOpenTunnelStream Method

METHOD NAME	DESCRIPTION
rsslReactorOpenTunnelStream	Begins the establishment of a tunnel stream. The `RsslTunnelStream` will be returned via the `RsslTunnelStreamStatusEventCallback` specified on the `RsslTunnelStreamOpenOptions`. For more information, refer to Section 6.7.2.2.

Table 59: `rsslReactorOpenTunnelStream` Method

6.7.2.2 RsslTunnelStreamOpenOptions

The `RsslTunnelStreamOpenOptions` contain necessary event handler associations and options used for creation of a tunnel stream.

CLASS MEMBER	DESCRIPTION
domainType	Indicates the domain the tunnel stream is established for. This should be set to the domain specified on the service the tunnel stream is being opened on.

Table 60: `RsslTunnelStreamOpenOptions` Class Members

CLASS MEMBER	DESCRIPTION
streamId	Indicates the stream ID to use for the tunnel stream itself. All substreams will flow within this stream ID, but will each have their own independent stream IDs assigned. For example, the tunnel stream stream ID can be 10. If a substream is opened for retrieving TRI, this can have stream ID 5 and it will be encapsulated within the tunnel stream with stream ID 10.
serviceId	Indicates the service ID associated with the service the tunnel stream is being opened on.
userSpecPtr	Indicates a user specified-object that is passed in via these options and then associated with the `RsslTunnelStream`.
statusEventCallback	Specifies an instance of the callback for `RsslTunnelStreamStatusEvents`, which provides the `RsslTunnelStream` upon initial connection and after the tunnel stream is established. For details on the `RsslTunnelStreamStatusEventCallback`, refer to Section 6.7.4.
queueMsgCallback	Specifies the instance of the callback used to handle Queue Messages received on this `RsslTunnelStream`. • For details on the `RsslTunnelStreamQueueMsgCallback`, refer to Section 6.7.4. • For details on various Queue Messages, refer to Section 7.6.
defaultMsgCallback	Specifies the instance of the callback that handles all other content received on this `RsslTunnelStream`. For details on the `RsslTunnelStreamMsgCallback`, refer to Section 6.7.4.
name	Specifies the tunnel stream name, which is provided to the remote application.
responseTimeout	Sets the duration (in seconds) to wait for a provider to respond to a tunnel stream open request. If the provider does not respond in time, an `RsslTunnelStreamStatusEvent` is sent to the application to indicate that the tunnel stream was not opened.
guaranteedOutputBuffers	Sets the number of guaranteed output buffers available for the tunnel stream.
pAuthLoginRequest	Specifies the `RsslRDMLoginRequest` to send if `RsslClassOfService.authentication.type` is set to **RDM_COS_AU_OMM_LOGIN**. If absent, the API uses the login request provided on its `RsslReactorOMMConsumerRole.pLoginRequest`.
classOfService	Specifies the tunnel stream's class of service to open. For details on `RsslClassOfService`, refer to Section 6.7.3.

Table 60: `RsslTunnelStreamOpenOptions` Class Members (Continued)

6.7.3 Negotiating Stream Behaviors: Class of Service

The `RsslClassOfService` is used to negotiate behaviors of an `RsslTunnelStream`. Negotiated behaviors are divided into five categories: common, authentication, flow control, data integrity, and guarantee.

- When an OMM Consumer application calls `rsslReactorOpenTunnelStream`, it sets the `RsslTunnelStreamOpenOptions.classOfService` members to manage and control tunnel stream behaviors. The consumer passes these settings to the connected OMM Provider.
- When the OMM Provider application receives an `RsslTunnelStreamRequestEvent`, the provider calls `rsslTunnelStreamRequestGetCos` to retrieve the behaviors requested by the consumer.

After tunnel stream negotiation is complete, the provider and consumer each receives an `RsslTunnelStreamStatusEvent` where each can view the negotiated behaviors on the `RsslTunnelStream` structure.

Note: Do not modify the `RsslClassOfService` member of the `RsslTunnelStream`.

The enumerations given for members described in this section can be found in **rsslRDM.h**.

6.7.3.1 ClassOfService Common Member

Common elements describe options related to the exchange of messages, such as the maximum message size and desired exchange protocol.

MEMBER	DEFAULT	RANGE/ ENUMERATIONS	DESCRIPTION
maxMsgSize	6144	1 – 2,147,483,647	The maximum size of messages exchanged on the tunnel stream. This value is only set by providers when accepting a tunnel stream.
protocolType	RSSL_RWF_PROTOCOL_TYPE	0 – 255	Identifies the protocol of the messages exchanged on the tunnel stream.
protocolMajorVersion	RSSL_RWF_MAJOR_VERSION	0 – 255	The major version of the protocol specified by `protocolType`.
protocolMinorVersion	RSSL_RWF_MINOR_VERSION	0 – 255	The minor version of the protocol specified by `protocolType`.

Table 61: `RsslClassOfService.common` Structure Members

6.7.3.2 ClassOfService Authentication Members

The authentication member contains options to authenticate a consumer to the corresponding provider.

MEMBER	DEFAULT	RANGE/ ENUMERATIONS	DESCRIPTION
type	RDM_COS_AU_NOT_REQUIRED	RDM_COS_AU_NOT_REQUIRED == 0, RDM_COS_AU_OMM_LOGIN == 1	Indicates the type of authentication, if any, to perform to the tunnel stream. For more information on authentication, refer to Section 6.7.1.

Table 62: `RsslClassOfService.authentication` Structure Members

6.7.3.3 ClassOfService Flow Control Members

The flow control member contains options related to flow control, such as the type and the allowed window of outstanding data.

MEMBER	DEFAULT	RANGE/ ENUMERATIONS	DESCRIPTION
type	RDM_COS_FC_NONE	RDM_COS_FC_NONE == 0, RDM_COS_FC_BIDIRECTIONAL == 1	Indicates the type of flow control (if any) to apply to the tunnel stream.
recvWindowSize	-1	0 – 2,147,483,647	Sets the amount of data (in bytes) that the remote peer can send to the application over a reliable tunnel stream. If `type` is set to **RDM_COS_FC_NONE**, this parameter has no effect. **-1** indicates that the application wishes to use the default value for the negotiated flow control type. In this case, if `type` is set to **RDM_COS_FC_BIDIRECTIONAL**, the default is **12288**.
sendWindowSize	None	0 – 2,147,483,647	Indicates the amount of data (in bytes) the application can send to the remote peer on a reliable tunnel stream. This value is provided on the `RsslTunnelStream` object and does not need to be set when opening or accepting a tunnel stream. This value is retrieved from the remote end and is informational, as flow control is performed by the API. When room is available in the window, the API transmits more content as submitted by the application. If `type` is set to **RDM_COS_FC_NONE**, this parameter has no effect.

Table 63: `RsslClassOfService.flowControl` Structure Members

6.7.3.4 ClassOfService Data Integrity Member

The data integrity member contains options related to the reliability of content exchanged over the tunnel stream.

MEMBER	DEFAULT	RANGE	DESCRIPTION
type	RDM_COS_DI_BEST_EFFORT	RDM_COS_DI_BEST_EFFORT == 0, RDM_COS_DI_RELIABLE == 1	Sets the level of reliability for message transmission on the tunnel stream. If set to **RDM_COS_DI_RELIABLE**, data is retransmitted as needed over the tunnel stream to ensure that all messages are delivered in the correct order. **Note:** At this time, **RDM_COS_DI_RELIABLE** is the only supported option.

Table 64: `RsslClassOfService.dataIntegrity` Structure Members

6.7.3.5 ClassOfService Guarantee Members

The guarantee member contains options related to the guarantee of content submitted over the tunnel stream.

OMM Consumer applications performing Queue Messaging to a Queue Provider should set the `ClassOfService.guarantee.type` to **RDM_COS_GU_PERSISTENT_QUEUE**.

MEMBER	DEFAULT	RANGE	DESCRIPTION
type	RDM_COS_GU_NONE	RDM_COS_GU_NONE == 0, RDM_COS_GU_PERSISTENT_QUEUE == 1	Indicates the level of guarantee that will be performed on this stream. **RDM_COS_GU_PERSISTENT_QUEUE** is not supported for provider applications. **Note:** If `type` is set to **RDM_COS_GU_PERSISTENT_QUEUE** for a consumer application, the data integrity `type` must also be set to **RDM_COS_DI_RELIABLE** and the flow control `type` to **RDM_COS_FC_BIDIRECTIONAL**.
persistLocally	RSSL_TRUE	RSSL_FALSE, RSSL_TRUE	Indicates whether messages are persisted locally on the tunnel stream. When `type` is **RDM_COS_GU_NONE**, this member has no effect.
persistenceFilePath	NULL	n/a	File path where files containing persistent messages may be stored. If set to NULL, the current working directory is used. When `type` is **RDM_COS_GU_NONE** or `persistLocally` is set to **RSSL_FALSE**, this member has no effect.

Table 65: `RsslClassOfService.guarantee` Structure Members

6.7.4 Tunnel Stream Callback Functions and Event Types

6.7.4.1 Tunnel Stream Callback Functions

The **`RsslTunnelStream`** delivers events via several user implemented callback functions. The callback interfaces are described below and the event objects that they return are defined in Section 6.7.4.2. Each callback returns the **`RsslTunnelStream`** that the event occurred on along with the event itself.

CALLBACK FUNCTION	DESCRIPTION
RsslTunnelStreamStatusEventCallback	Communicates status information about the tunnel stream. Additionally, this callback delivers the **`RsslTunnelStream`** object when the enhanced private stream establishment completes. This callback provides an **`RsslTunnelStreamStatusEvent`** to the application. Details about this event are available in Section 6.7.4.2.
RsslTunnelStreamDefaultMsgCallback	Similar to the **`ReactorChannels defaultMsgCallback`**, content received by the tunnel stream will be returned via this callback if it is not handled by a more specific content handler, such as the **`RsslTunnelStreamQueueMsgCallback`**. This callback provides an **`RsslTunnelStreamMsgEvent`** to the application. Details about this event are available in Section 6.7.4.2.
RsslTunnelStreamQueueMsgCallback	Any Queue Messages will be delivered via this callback and presented to the user in their native Queue Message formats. If not specified, queue messages will be delivered via the **`RsslTunnelStreamDefaultMsgCallback`**, however they will not be presented in a Queue Message format. This callback provides a **`RsslTunnelStreamQueueMsgEvent`** to the application. Details about this event are available in Section 6.7.4.2.

Table 66: Tunnel Stream Callback Functions

6.7.4.2 Tunnel Stream Callback Event Types

The various tunnel stream callbacks return their information via specific event objects. The following table defines these events.

EVENT	EVENT DESCRIPTION	CLASS MEMBER	CLASS MEMBER DESCRIPTION
RsslTunnelStreamStatusEvent	This event is used to present the tunnel stream and its status.	pReactorChannel	A pointer to the `RsslReactorChannelunnelStream` with which this tunnel stream is associated.
		pState	Indicates status information associated with the `RsslTunnelStream`. For instance, a state of **OPEN** and **OK** indicates that the tunnel stream is established and content should be flowing as expected. If **CLOSED_RECOVER** and **SUSPECT**, this indicates that the connection or tunnel stream may be lost, however if performing guaranteed messaging content may still be able to be persisted by the Reactor and communicated upon recovery of the tunnel stream.
		pRsslMsg	A pointer to an `RsslMsg` structure.
		pAuthInfo	If the event was produced by an authentication message, `pAuthInfo` is populated by an `RsslTunnelStreamAuthInfo` structure. For more information, refer to Section 6.7.1.
RsslTunnelStreamMsgEvent	This event presents content received on the `RsslTunnelStream`. If a more specific handler, such as the `RsslTunnelStreamQueueMsgEvent`, is also configured, messages of that type will go to their specific handler.	pReactorChannel	A pointer to the `RsslReactorChannelunnelStream` with which this tunnel stream is associated.
		pRsslMsg	A pointer to an RsslMsg structure, used to deliver any OMM content or opaque content.
		pErrorInfo	Used to convey error information, when applicable.

Table 67: Tunnel Stream Callback Event Types

EVENT	EVENT DESCRIPTION	CLASS MEMBER	CLASS MEMBER DESCRIPTION
RsslTunnelStreamQueueMsgEvent	This event is used to present any Queue Message content received on the `RsslTunnelStream`.	base	An `RsslTunnelStreamMsgEvent`. Refer to Section 6.7.4.1.
		pQueueMsg	A pointer to a Queue Message containing OMM content or opaque content exchanged with a Queue Provider. Refer to subsequent chapters for information about Queue Messages.

Table 67: Tunnel Stream Callback Event Types (Continued)

6.7.5 Opening a Tunnel Stream Code Sample

The following code sample illustrates how to open a tunnel stream. The example assumes that a Reactor and ReactorChannel are already open and properly established.

```
// Basic sample for event handlers

// RsslTunnelStreamStatusEventCallback
RsslReactorCallbackRet tunnelStreamStatusEventCallback(RsslTunnelStream
        *pTunnelStream,RsslTunnelStreamStatusEvent *pEvent)
{
    printf(“Status of Tunnel Stream %d is %d:%d\n”, pTunnelStream->streamId, pEvent->pState-
            >streamState, pEvent->pState->dataState);
    return RSSL_RC_CRET_SUCCESS;
}

// RsslTunnelStreamDefaultMsgCallback
RsslReactorCallbackRet tunnelStreamDefaultMsgCallback(RsslTunnelStream *pTunnelStream,
        RsslTunnelStreamMsgEvent *pEvent)
{
    printf(“Received content on Tunnel Stream %d\n”, pTunnelStream->streamId);
    return RSSL_RC_CRET_SUCCESS;
}

// RsslTunnelStreamQueueMsgCallback
RsslReactorCallbackRet tunnelStreamQueueMsgCallback(RsslTunnelStream *pTunnelStream,
        RsslTunnelStreamQueueMsgEvent *pEvent)
{
    printf(“Received Queue Message on Tunnel Stream %d\n”, pTunnelStream->streamId);
    return RSSL_RC_CRET_SUCCESS;
}

int openTunnelStream()
{
    RsslTunnelStreamOpenOptions _openOptions;
    RsslErrorInfo _errorInfo;
    rsslClearTunnelStreamOpenOptions(&_openOptions);

    // populate the options and enable guaranteed delivery for communication with a Queue Provider
    _openOptions.classOfService.guarantee.type = RDM_COS_GU_PERSISTENT_QUEUE;
    _openOptions.classOfService.dataIntegrity = RDM_COS_DI_RELIABLE;
    _openOptions.classOfService.flowControl = RDM_COS_FC_BIDIRECTIONAL;
    _openOptions.classOfService.guarantee.persistLocally = RSSL_TRUE;
    _openOptions.streamId = TUNNEL_STREAM_ID;
    _openOptions.domainType = RSSL_DMT_QUEUE_MESSAGING;
    _openOptions.serviceId = QUEUE_MESSAGING_SERVICE_ID;
    // specify the event handlers
    _openOptions.statusEventCallback = tunnelStreamStatusEventCallback;
    _openOptions.defaultMsgCallback = tunnelStreamDefaultMsgCallback;
    _openOptions.queueMsgCallback = tunnelStreamQueueMsgCallback;

    if ((rsslReactorOpenTunnelStream(_pReactorChannel, &_openOptions, &_errorInfo)) !=
            RSSL_RET_SUCCESS)
    {
        printf(“rsslReactorOpenTunnelStream failed!”);
        return RSSL_RET_FAILURE;
    }

    printf(“rsslReactorOpenTunnelStream succeeded!”);
    return RSSL_RET_SUCCESS;
}
```

Code Example 15: Opening a Tunnel Stream

6.7.6 Accepting Tunnel Streams

OMM provider applications can accept tunnel streams provided on an `RsslReactorChannel` (enabled by specifying a `tunnelStreamListenerCallback` on the `RsslReactorOMMProviderRole`).

When a consumer opens a tunnel stream, the `tunnelStreamListenerCallback` receives an `RsslTunnelStreamRequestEvent`. At this point, the provider should call `rsslTunnelStreamRequestGetCos` to retrieve the `RsslClassOfService` requested by the tunnel stream, and ensure that the parameters indicated by the members of that class of service match what the provider allows. The provider can also check the `RsslTunnelStreamRequestEvent.classOfServiceFilter` to determine which behaviors are supported by the consumer. For more information on this filter, refer to Section 6.7.6.1.

- To accept a tunnel stream, the provider must call `rsslReactorAcceptTunnelStream` with the given `RsslTunnelStreamRequestEvent`. Further events regarding the accepted stream are provided in the specified `RsslReactorAcceptTunnelStreamOptions.statusEventCallback`.
- To reject a tunnel stream, the provider calls `rsslReactorRejectTunnelStream` with the given `RsslTunnelStreamRequestEvent`. No further events are received for that tunnel stream.

Queue messaging (an `RsslClassOfService.guarantee.type` setting of **RDM_COS_GU_PERSISTENT_QUEUE**) is not supported for provider applications.

The API will automatically reject tunnel streams that contain invalid information. When this occurs, the provider application will receive warnings via an `RsslReactorChannelEvent`. The type will be set to **RSSL_RC_CET_WARNING** and the `RsslErrorInfo` in the event will contain text describing the reason for the rejection.

Warning! Ensure that the provider application calls `rsslReactorAcceptTunnelStream` or `rsslReactorRejectTunnelStream` before returning from the `tunnelStreamListenerCallback`. If not, the provider application will receive a warning via an `RsslReactorChannelEvent` similar to the above, and the stream will be automatically rejected.

6.7.6.1 Reactor Tunnel Stream Listener Callback and Tunnel Stream Request Event

OMM providers that want to handle tunnel streams from connected consumers can specify a `tunnelStreamListenerCallback`. This callback informs the provider application of any consumer tunnel stream requests. The provider can specify this callback on the `RsslReactorOMMProviderRole`. It has the following signature:

```
RsslTunnelStreamListenerCallback(RsslTunnelStreamRequestEvent*, RsslErrorInfo*)
```

For more information on the `RsslReactorOMMProviderRole`, refer to Section 6.3.3.

An `RsslTunnelStreamRequestEvent` is returned to the application via the `RsslTunnelStreamListenerCallback`.

STRUCTURE MEMBER	DESCRIPTION
pReactorChannel	Specifies the `RsslReactorChannel` on which the event was received.
streamId	Specifies the stream ID of the requested tunnel stream.
domainType	Specifies the domain type of the requested tunnel stream.
serviceId	Specifies the service ID of the requested tunnel stream.
name	Specifies the name of the requested tunnel stream.
classOfServiceFilter	Sets a filter that indicates which `RsslClassOfService` members were present. The provider can use this filter to determine whether behaviors are supported by the consumer, and if needed reject the tunnel stream before calling `rsslTunnelStreamRequestGetCos` to get the full `RsslClassOfService`. For enumerations of the flags present in this filter, refer to `RsslTunnelStreamCoSFilterFlags` in **rsslRDM.h**.

Table 68: `RsslTunnelStreamRequestEvent` Structure Members

6.7.6.2 rsslReactorAcceptTunnelStream Function

FUNCTION NAME	DESCRIPTION
rsslReactorAcceptTunnelStream	Accepts a tunnel stream requested by a consumer. The `RsslTunnelStream` is returned in the `RsslTunnelStreamStatusEventCallback` specified on the `RsslReactorAcceptTunnelStreamOptions`. For more information, refer to Section 6.7.6.3.

Table 69: `rsslReactorAcceptTunnelStream` Function

6.7.6.3 RsslReactorAcceptTunnelStreamOptions

OPTION	DESCRIPTION
statusEventCallback	Specifies the instance of the callback for `RsslTunnelStreamStatusEvents`, which provides the `RsslTunnelStream` on initial connection and then communicates state information about the tunnel afterwards. For details on the `RsslTunnelStreamStatusEventCallback`, refer to Section 6.7.4.1.
defaultMsgCallback	Specifies the instance of the callback used to handle all other content received on this `RsslTunnelStream`. For details on `RsslTunnelStreamDefaultMsgCallback`, refer to Section 6.7.4.1.
userSpecPtr	Specifies a user-defined pointer passed in via these options and then associated with the `RsslTunnelStream`.
classOfService	Specifies an `RsslClassOfService` with members indicating behaviors that the application wants to apply to the `RsslTunnelStream`. For more information on class of service, refer to Section 6.7.3.
guaranteedOutputBuffers	Sets the number of pooled buffers available to the application when writing content to `RsslTunnelStream`.

Table 70: `RsslReactorAcceptTunnelStreamOptions` Options

6.7.6.4 rsslReactorRejectTunnelStream Function

FUNCTION NAME	DESCRIPTION
rsslReactorRejectTunnelStream	Rejects a tunnel stream requested by a consumer. No further events will be received for this tunnel stream. For more information, refer to Section 6.7.6.5.

Table 71: `rsslReactorRejectTunnelStream` Function

6.7.6.5 RsslReactorRejectTunnelStreamOptions

OPTION	DESCRIPTION
state	An `RsslState` to send to the consumer. The application can use the `state.streamState`, `state.dataState`, and `state.text` to indicate the nature of the rejection.
pCos	An optional `RsslClassOfService` to send to the consumer. If rejecting the stream due to a problem with the `RsslClassOfService` parameters from the `RsslTunnelStreamRequestEvent`, the provider application should populate this with the associated parameters.

Table 72: `RsslReactorRejectTunnelStreamOptions` Options

6.7.6.6 Accepting a Tunnel Stream Code Sample

The following code illustrates how to accept a tunnel stream requested by a consumer. The example presumes that a Reactor and Reactor Channel are already open and properly established.

```
RsslReactorCallbackRet tunnelStreamListenerCallback(RsslTunnelStreamRequestEvent *pEvent,
        RsslErrorInfo *pErrorInfo)
{
    RsslErrorInfo errorInfo;
    RsslRet ret;
    RsslClassOfService cos;
    RsslReactorAcceptTunnelStreamOptions acceptOpts;
    ret = rsslTunnelStreamRequestGetCos(pEvent, &cos, &errorInfo);

    /* Now presuming that the application wishes to accept the tunnel stream. */
    rsslClearReactorAcceptTunnelStreamOptions(&acceptOpts);
    acceptOpts.statusEventCallback = tunnelStreamStatusEventCallback;
    acceptOpts.defaultMsgCallback = tunnelStreamDefaultMsgCallback;

    /* Set desired ClassOfService options. */
    /* For this sample, set authentication to match consumer. */
    acceptOpts.classOfService.authentication.type = cos.authentication.type;
    acceptOpts.classOfService.flowControl.type = RDM_COS_FC_BIDIRECTIONAL;
    acceptOpts.classOfService.dataIntegrity.type = RDM_COS_DI_RELIABLE;
    /* ... (set additional members, based on what is desired by the provider) */

    ret = rsslReactorAcceptTunnelStream(pEvent, &acceptOpts, &errorInfo);

    return RSSL_RC_CRET_SUCCESS;
}
```

Code Example 16: Accepting a Tunnel Stream Code Example

6.7.6.7 Rejecting a Tunnel Stream Code Sample

The following code illustrates how to reject a tunnel stream requested by a consumer. The example presumes that a Reactor and Reactor Channel are already open and properly established.

```
RsslReactorCallbackRet tunnelStreamListenerCallback(RsslTunnelStreamRequestEvent *pEvent,
        RsslErrorInfo *pErrorInfo)
{
    RsslErrorInfo errorInfo;
    RsslRet ret;
    RsslClassOfService cos;

    ret = rsslTunnelStreamRequestGetCos(pEvent, &cos, &errorInfo);

    /* Now presuming that the application wishes to reject the tunnel stream
     * Because it only communicates using the RWF protocol type. */

    if (cos.common.protocolType != RSSL_RWF_PROTOCOL_TYPE)
    {
        RsslReactorRejectTunnelStreamOptions rejectOpts;
        RsslClassOfService expectedCos;
        rsslClearReactorRejectTunnelStreamOptions(&rejectOpts);

        rejectOpts.state.streamState = RSSL_STREAM_CLOSED;
        rejectOpts.state.dataState = RSSL_DATA_SUSPECT;
        rejectOpts.state.text.data = "This provider only communicates using the RWF protocol.";
        rejectOpts.state.text.length = (RsslUInt32)strlen(rejectOpts.state.text.data);

        /* Set what the class of service is expected to be. */
        rsslClearClassOfService(&expectedCos);
        expectedCos.common.protocolType = RSSL_RWF_PROTOCOL_TYPE;
        expectedCos.common.protocolMajorVersion = RSSL_RWF_MAJOR_VERSION;
        expectedCos.common.protocolMinorVersion = RSSL_RWF_MINOR_VERSION;
        expectedCos.authentication.type = RDM_COS_AU_NOT_REQUIRED;
        expectedCos.flowControl.type = RDM_COS_FC_BIDIRECTIONAL;
        expectedCos.dataIntegrity.type = RDM_COS_DI_RELIABLE;
        /* ... (set additional members, based on what is desired by the provider) */

        rejectOpts.pCos = &expectedCos;

        ret = rsslReactorRejectTunnelStream(pEvent, &rejectOpts, &errorInfo);
    }

    return RSSL_RC_CRET_SUCCESS;
}
```

Code Example 17: Rejecting a Tunnel Stream Code Example

6.7.7 Receiving Content on a TunnelStream

Invoking the **`RsslReactor.dispatch`** method will read and process inbound content, where any information received on this **`RsslTunnelStream`** will be delivered to the application via the tunnel stream callback methods specified via **`rsslReactorOpenTunnelStream`** or **`rsslReactorAcceptTunnelStream`**.

The tunnel stream callback methods are described in Section 6.7.4 and the events they deliver are defined in Section 6.7.4.2. Dispatching this content works in the same manner as dispatching any other content on the Reactor.

6.7.8 Sending Content on a TunnelStream

When sending content on an **`RsslTunnelStream`**, the user should get a buffer from the **`RsslTunnelStream`**, encode their content into the buffer, and then use the **`rsslTunnelStreamSubmitMsg`** method to push the content out over the **`RsslTunnelStream`**. By obtaining a buffer from the **`RsslTunnelStream`**, the behaviors that were negotiated on the enhanced private stream can be properly handled by the Reactor, making this functionality appear nearly transparent to the user.

6.7.8.1 Tunnel Stream Buffer Methods

METHOD NAME	DESCRIPTION
rsslTunnelStreamGetBuffer	Obtains a buffer from the `RsslTunnelStream`. The buffer is associated with the tunnel stream it is obtained from, ensuring that the correct behaviors are enforced on the content within the buffer.
rsslTunnelStreamReleaseBuffer	If a buffer is not submitted by the user, it should be returned to the `RsslTunnelStream` that it came from. This will ensure it is properly recycled and can be reused. If it is submitted properly, the user should not release it, as the submit method will handle this once content is sent on the `RsslTunnelStream`.

Table 73: Tunnel Stream Buffer Methods

6.7.8.2 Tunnel Stream Submit

The submit method is used to write content to the `RsslTunnelStream`. This method will ensure that any specified behaviors are enforced on the content that is submitted, for example if guaranteed messaging is specified this content will follow the persistence options that were configured by the user.

METHOD NAME	DESCRIPTION
rsslTunnelStreamSubmitMsg	Allows the user to pass in RDM Message content, including Queue Messages, that will be processed and sent over the `RsslTunnelStream`. This method has additional options that can be specified via the `RsslTunnelStreamSubmitOptions`. Currently, the only available members of the option structure allow the user to pass in an RDM Message or an RsslMsg structure containing their content.
rsslTunnelStreamSubmit	Allows the user to pass in a buffer populated with content that will be processed and sent over the `RsslTunnelStream`.

Table 74: Tunnel Stream Submit Message

6.7.8.3 RsslTunnelStreamSubmitOptions

When calling `rsslTunnelStreamSubmit`, you can use `RsslTunnelStreamSubmitOptions` to provide the `containerType` option.

MEMBER	DESCRIPTION
containerType	Specifies the type of data in the buffer being submitted. For example: • If the submitted buffer contains an `RsslMsg`, set `containerType` **RSSL_DT_MSG**. • If non-RWF data is sent, this should be set to a non-RWF type such as **RSSL_DT_OPAQUE**. • For more information on possible container types, refer to the *Transport API C Edition Developers Guide*.

Table 75: `RsslTunnelStreamSubmitOptions` Structure Members

6.7.8.4 RsslTunnelStreamSubmitMsgOptions

When calling `rsslTunnelStreamSubmitMsg`, you can use `RsslTunnelStreamSubmitMsgOptions` to provide options the following options:

MEMBER	DESCRIPTION
pRsslMsg	Specifies an `RsslMsg` populated by the application, which the API encodes and sends over the `RsslTunnelStream`; mutually exclusive with `pRDMMsg`.
pRDMMsg	Specifies an `RsslRDMMsg` populated by the application, which the API encodes and sends over the `RsslTunnelStream`; mutually exclusive with `pRsslMsg`.

Table 76: `RsslTunnelStreamSubmitMsgOptions` Structure Members

6.7.8.5 Submitting Content on a Tunnel Stream Code Sample

The following code sample is a basic example of writing opaque content to a tunnel stream. This can be combined with the QueueData message samples in subsequent chapters to send content to a Queue Provider.

```
int submitMessage()
{
    RsslErrorInfo _errorInfo;
    RsslBuffer *pBuffer;
    RsslTunnelStreamGetBufferOptions _getBufferOpts;
    RsslTunnelStreamSubmitOptions _submitOpts;

    // gets a buffer of 50 bytes to put content into.
    rsslClearTunnelStreamGetBufferOptions(&_getBufferOptions);
    _getBufferOptions.size = 50;
    pBuffer = rsslTunnelStreamGetBuffer(pTunnelStream, &_getBufferOptions, _errorInfo);

    // put generic content into the buffer
    pBuffer->data = “Hello World!”;
    pBuffer->length = 12;

    rsslClearTunnelStreamSubmitOptions(&_submitOpts);
    _submitOpts.containerType = RSSL_DT_OPAQUE;
    if ((rsslTunnelStreamSubmit(pTunnelStream, pBuffer, &_submitOpts, &_errorInfo)) !=
            RSSL_RET_SUCCESS)
    {
        printf(“Content submission failed!”);
        // Because submission failed, we need to return the buffer to the tunnel stream
        rsslTunnelStreamReleaseBuffer(&_buffer, &_errorInfo);

        return RSSL_RET_FAILURE;
    }

    printf(“Content submission succeeded!”);
    // Thanks to successful submission, we do not need to release the buffer because the Reactor will.
    return RSSL_RET_SUCCESS;
}
```

Code Example 18: Submitting Content on a Tunnel Stream

6.7.8.6 Closing a Tunnel Stream

When an application has completed its use of an **`RsslTunnelStream`**, it can be closed.

METHOD NAME	DESCRIPTION
rsslReactorCloseTunnelStream	Closes a tunnel stream. Once closed, any content stored for guaranteed messaging or reliable delivery will be cleaned up.

Table 77: `rsslReactorCloseTunnelStream`

6.7.8.7 RsslTunnelStreamCloseOptions

When calling `rsslTunnelStreamClose`, you can use `RsslTunnelStreamCloseOptions` to provide the `finalStatusEvent` option.

MEMBER	DESCRIPTION
finalStatusEvent	Indicates that the application wants to receive a final `RsslTunnelStreamStatusEvent` whenever the tunnel stream closes. If set to **RSSL_TRUE**, the tunnel stream is cleaned up after the application receives the final `RsslTunnelStreamStatusEvent`.

Table 78: `RsslTunnelStreamCloseOptions` Structure Members

6.7.8.8 Closing a Tunnel Stream Code Sample

The following code sample illustrates how to close a tunnel stream.

```
int closeTunnelStream()
{
    RsslTunnelStreamCloseOptions _closeOpts;

    rsslClearTunnelStreamCloseOptions(&_closeOpts);
    _closeOpts.finalStatusEvent = RSSL_TRUE;

    if ((rsslReactorCloseTunnelStream(pTunnelStream, &_closeOpts, &_errorInfo)) != RSSL_RET_SUCCESS)
    {
        printf(“Closing tunnel stream failed!”);
        return RSSL_RET_FAILURE;
    }

    printf(“Tunnel Stream closed successfully.”);
    return RSSL_RET_SUCCESS;
}
```

Code Example 19: Closing a Tunnel Stream

6.8 Reactor Utility Functions

The Transport API Reactor provides several additional utility functions. These functions can be used to query more detailed information for a specific connection or change certain `RsslReactorChannel` parameters during run-time. These functions are described in the following tables.

6.8.1 General Reactor Utility Functions

FUNCTION NAME	DESCRIPTION
rsslReactorGetChannelInfo	Allows the application to query `RsslReactorChannel` negotiated parameters and settings and retrieve all current settings. This includes `maxFragmentSize` and negotiated compression information as well as many other values. See `RsslReactorChannelInfo` structure, defined in Section 6.8.2, for a full list of available settings. This function calls the Transport API `rsslGetChannelInfo` function which has its use and return values described in the *Transport API C Edition Developers Guide*.
rsslReactorIoctl	Allows the application to change various settings associated with the `RsslReactorChannel`. The available options are defined in Section 6.8.3. This function calls the Transport API `rsslIoctl` function which has its use and return values described in the *Transport API C Edition Developers Guide*.

Table 79: Reactor Utility Functions

6.8.2 RsslReactorChannelInfo Structure Members

The following table describes the values available to the user through using the `rsslReactorGetChannelInfo` function. This information is returned as part of the `RsslReactorChannelInfo` structure.

STRUCTURE MEMBER	DESCRIPTION
rsslChannelInfo	Returns the underlying `RsslChannel` information. This includes `maxFragmentSize`, number of output buffers, compression information, and more. The `RsslChannelInfo` function structure is fully described in the *Transport API C Edition Developers Guide*.

Table 80: `RsslReactorChannelInfo` Structure Members

6.8.3 rsslReactorIoctl Option Values

There are currently no `RsslReactor` or `RsslReactorChannel` specific codes for use with the rsslReactorIoctl. Reactor specific codes may be added in the future. The application can still use any of the codes allowed with `rsslIoctl`, which are documented in the *Transport API C Edition Developers Guide*.

Chapter 7 Administration Domain Models Detailed View

7.1 Concepts

Administration Domain Model Representations are RDM-specific representations of OMM administrative domain models. This Value Added Component contains structures that represent messages within the Login, Source Directory, and Dictionary domains (as discussed in Table 81). All structures follow the formatting and naming specified in the *Transport API C Edition RDM Usage Guide*, so access to content is logical and specific to the content being represented. This component also handles all encoding and decoding functionality for these domain models, so the application needs only to manipulate the message's structure members to send or receive this content. Such functionality significantly reduces the amount of code an application needs to interact with OMM devices (i.e., Enterprise Platform for Real-time), and also ensures that encoding/decoding for these domain models follow OMM-specified formatting rules. Applications can use this Value Added Component directly to help with encoding, decoding, and representation of these domain models. When using the Transport API Reactor, this component is embedded to manage and present callbacks with a domain-specific representation of content.

Where possible, the members of an Administration Domain Model Representation structure are represented in the structure with the same **`RsslDataType`** that is specified for the element by the Domain Model. In cases where multiple elements are part of a more complex container such as an **`RsslMap`** or **`RsslElementList`**, the elements are represented with a C-style array with an associated count indicating the number of structures in the array.

The *Transport API C Edition RDM Usage Guide* defines and describes all domain-specific behaviors, usage, and details.

DOMAIN	PURPOSE
Login	Authenticates users and advertise/request features that are not specific to a particular domain. Use of and support for this domain is required for all OMM applications. This is considered an administrative domain, content is required and expected by many Thomson Reuters components and conformance to the domain model definition is expected. For further details refer to Section 7.3.
Source Directory	Advertises information about available services and their state, QoS, and capabilities. This domain also conveys any group status and group merge information. Interactive and Non-Interactive OMM Provider applications require support for this domain. Thomson Reuters strongly recommends that OMM Consumers request this domain. This is considered an administrative domain, and many Thomson Reuters components expect and require content to conform to the domain model definition. For further details, refer to Section 7.4.
Dictionary	Provides dictionaries that may be needed when decoding data. Though use of the Dictionary domain is optional, Thomson Reuters recommends that Provider applications support the domain's use. Considered an administrative domain, content is required and expected by many Thomson Reuters components and following the domain model definition is expected. For further details refer to Section 7.5.

Table 81: Domains Representations in the Administration Domain Model Value Added Component

7.2 RDM Message Base

All Administration Domain Model Representation structures contain a common base structure that provides members common to all representations and identifies the specific message.

7.2.1 RSSL RDM Message Base Structure Members

All domain representation structures have several common members used for stream and domain identification. These are available in the `RsslRDMMsgBase` structure, as described in the following table.

STRUCTURE MEMBER	DESCRIPTION
streamId	**Required**. Unique signed-integer identifier associated with all messages flowing within a stream. Positive values indicate a consumer-instantiated stream, typically via a request message. Negative values indicate a provider-instantiated stream, often associated with Non-Interactive Providers. This is required on all messages.
domainType	**Required**. Identifies the specific domain message model type. If value is less than **128**, domain is a Thomson Reuters defined domain model. If value is **128 - 255**, domain is a user defined domain model. Domain model definition is decoupled from the API and domain models are typically defined in some type of specification document. Thomson Reuters defined domain models are specified in the *Transport API C Edition RDM Usage Guide*. This is required on all messages.
rdmMsgType	Required. Identifies the specific representation for a given domain. The currently supported `rdmMsgTypes` are defined in Table 7.2.2.

Table 82: `RsslRDMMsgBase` Structure Members

7.2.2 RSSL RDM Message Types

The following table provides a reference mapping between the administrative domain type and the structural representations provided in this component.

DOMAIN TYPE	RDM MESSAGE TYPE	RDM MESSAGE STRUCTURE
RSSL_DMT_LOGIN (`RsslRDMLoginMsg`) Refer to Section 7.3	RDM_LG_MT_REQUEST	RsslRDMLoginRequest
	RDM_LG_MT_REFRESH	RsslRDMLoginRefresh
	RDM_LG_MT_STATUS	RsslRDMLoginStatus
	RDM_LG_MT_CLOSE	RsslRDMLoginClose
	RDM_LG_MT_CONSUMER_CONNECTION_STATUS	RsslRDMLoginConsumerConnectionStatus
RSSL_DMT_SOURCE (`RsslRDMDirectoryMsg`) Refer to Section 7.4	RDM_DR_MT_REQUEST	RsslRDMDirectoryRequest
	RDM_DR_MT_REFRESH	RsslRDMDirectoryRefresh
	RDM_DR_MT_UPDATE	RsslRDMDirectoryUpdate
	RDM_DR_MT_STATUS	RsslRDMDirectoryStatus
	RDM_DR_MT_CLOSE	RsslRDMDirectoryClose
	RDM_DR_MT_CONSUMER_STATUS	RsslRDMDirectoryConsumerStatus
RSSL_DMT_DICTIONARY (`RsslRDMDictionaryMsg`) Refer to Section 7.5	RDM_DC_MT_REQUEST	RsslRDMDictionaryRequest
	RDM_DC_MT_REFRESH	RsslRDMDictionaryRefresh
	RDM_DC_MT_STATUS	RsslRDMDictionaryStatus
	RDM_DC_MT_CLOSE	RsslRDMDictionaryClose

Table 83: `RsslRDMMsg` Types

7.2.3 RSSL RDM Encoding and Decoding Functions

Encode and decode functionality is provided that can take the `RsslRDMMsg` union. This allows users to encode or decode from a general type that can represent any of the domain messages. Encode and decode functions are also provided for each specific domain type, as documented in the following chapters.

FUNCTION NAME	DESCRIPTION
rsslEncodeRDMMsg	Used to encode any message that the `RsslRDMMsg` can represent. This function takes the `RsslRDMMsg` as a parameter.
rsslDecodeRDMMsg	Used to decode any message that the `RsslRDMMsg` can represent. This function populates the `RsslRDMMsg` and leverages the Value Added Utility message buffer (refer to Section 8.2).
	Note: The decoded message may refer to encoded data from the original `RsslMsg`. If the message is to be stored, the appropriate copy function for the decoded `RsslRDMMsg` should be used to create a full copy.

Table 84: RDM Encoding and Decoding Functions

7.3 RDM Login Domain

The Login domain registers a user with the system, after which the user can request[1], post[2], or provide[3] OMM content. A Login request may also be used to authenticate a user with the system.

- A consumer application must log into the system before it can request or post content.
- A non-interactive provider application must log into the system before providing any content. An interactive provider application is required to handle log in requests and provide Login response messages, possibly using DACS to authenticate users.

The following sections detail layout and use of each message structure within the Login portion of the Administration Domain Message Component.

7.3.1 RSSL RDM Login Request

A ***Login Request*** message is encoded and sent by OMM Consumer and OMM non-interactive provider applications. This message registers a user with the system. After receiving a successful login response, applications can then begin consuming or providing additional content. An OMM Provider can use the Login request information to authenticate users with DACS.

The `RsslRDMLoginRequest` represents all members of a login request message and allows for simplified use in OMM applications that leverage RDM. This structure follows the behavior and layout that is defined in the *Transport API C Edition RDM Usage Guide*.

7.3.1.1 RSSL RDM Login Request Structure Members

STRUCTURE MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Contains general message information like streamId and domainType. For more information, refer to Section 7.2.

Table 85: `RsslRDMLoginRequest` Structure Members

1. Consumer applications can request content after logging into the system.
2. Consumer applications can post content, which is similar to contribution or unmanaged publication, after logging into the system.
3. Non-interactive provider applications.

STRUCTURE MEMBER	DESCRIPTION
flags	**Required**. Indicate presence of optional login request members. For details, refer to Section 7.3.1.2.
userNameType	**Optional**. If present, **`flags`** value of **RDM_LG_RQF_HAS_USERNAME_TYPE** should be specified. If absent, a default value of **RDM_LOGIN_USER_NAME** is assumed. Possible values: • RDM_LOGIN_USER_NAME == 1 • RDM_LOGIN_USER_EMAIL_ADDRESS == 2 • RDM_LOGIN_USER_TOKEN == 3 A type of **RDM_LOGIN_USER_NAME** typically corresponds to a DACS user name. This can be used to authenticate and permission a user. **RDM_LOGIN_USER_TOKEN** is specified when using AAAAPI The user token is retrieved from a AAAAPI gateway. To validate users, a provider application can pass this user token to an Authentication Manager application. This type of token periodically changes: when it changes, an application can send a login reissue to pass information upstream. For more information, refer to specific AAAAPI documentation.
userName	**Required**. Should be populated with user name, e-mail address, or user token based on the **`userNameType`** specification. If initializing **`RsslRDMLoginRequest`** using **`rsslInitDefaultRDMLoginRequest`**, the name of the user that is currently logged into the system the application is running on will be used.
applicationId	**Optional**. If present, **`flags`** value of **RDM_LG_RQF_HAS_APPLICATION_ID** should be specified. When populated, should contain the DACS **`applicationId`**. If the server authenticates with DACS, the consumer application may be required to pass in a valid application id. If initializing **`RsslRDMLoginRequest`** using **`rsslInitDefaultRDMLoginRequest`**, an **`applicationId`** of **256** will be used.
applicationName	**Optional**. If present, **`flags`** value of **RDM_LG_RQF_HAS_APPLICATION_NAME** should be specified. When present, the **`applicationName`** in the login request identifies the OMM consumer or OMM non-interactive provider. If initializing **`RsslRDMLoginRequest`** using **`rsslInitDefaultRDMLoginRequest`**, the name upa will be used.
position	**Optional**. If present, **`flags`** value of **RDM_LG_RQF_HAS_POSITION** should be specified. When populated, should contain the DACS **`position`**. If the server is authenticating with DACS, the consumer application might be required to pass in a valid position. If initializing **`RsslRDMLoginRequest`** using **`rsslInitDefaultRDMLoginRequest`**, the IP address of the system the application is running on will be used.
password	**Optional**. If present, **`flags`** value of **RDM_LG_RQF_HAS_PASSWORD** should be specified. When necessary, this should be set to the **`password`** for logging into the system. See specific component documentation to determine password requirements and how to obtain one.

Table 85: `RsslRDMLoginRequest` Structure Members (Continued)

STRUCTURE MEMBER	DESCRIPTION
instanceId	**Optional**. If present, `flags` value of **RDM_LG_RQF_HAS_INSTANCE_ID** should be specified. The `instanceId` can be used to differentiate applications running on the same machine. However, because `instanceId` is set by the user logging into the system, it does not guarantee uniqueness across different applications on the same machine.
providePermissionProfile	**Optional**. If present, `flags` value of **RDM_LG_RQF_HAS_PROVIDE_PERM_PROFILE** should be specified. If not present, a default value of **1** is assumed. When **1**, this indicates that a consumer desires the permission profile. The permission profile can be used by an application to perform proxy permissioning.
providePermissionExpressions	**Optional**. If present, `flags` value of **RDM_LG_RQF_HAS_PROVIDE_PERM_EXPR** should be specified. If absent, a default value of **1** is assumed. When **1**, this indicates a consumer wants permission expression information to be sent with responses. Permission expressions allow for items to be proxy permissioned by a consumer via content-based entitlements.
singleOpen	**Optional**. If present, `flags` value of **RDM_LG_RQF_HAS_SINGLE_OPEN** should be specified. If absent, a default value of **1** is assumed. • **1**: Indicates the consumer application wants the provider to drive stream recovery. • **0**: Indicates that the consumer application will drive stream recovery.
allowSuspectData	**Optional**. If present, `flags` value of **RDM_LG_RQF_HAS_ALLOW_SUSPECT_DATA** should be specified. If absent, a default value of **1** is assumed. • **1**: Indicates that the consumer application allows for suspect `streamState` information. • **0**: Indicates that the consumer application prefers any suspect data to result in the stream being closed with an **RSSL_STREAM_CLOSED_RECOVER** state.
role	**Optional**. If present, `flags` value of **RDM_LG_RQF_HAS_ROLE** should be specified. If absent, a default value of **RDM_LOGIN_ROLE_CONS** is assumed. Indicates the role of the application logging onto the system. • **0**: **RDM_LOGIN_ROLE_CONS**, indicates application is a consumer. • **1**: **RDM_LOGIN_ROLE_PROV**, indicates application is a provider.
downloadConnectionConfig	**Optional**. If present, `flags` value of **RDM_LG_RQF_HAS_DOWNLOAD_CONN_CONFIG** should be specified. If absent, a default value of **0** is assumed. Enabling this option allows the application to download information about other providers on the network. This downloaded information can be used to load balance connections across multiple providers. • **1**: Indicates the user wants to download connection configuration information. • **0**: Indicates that no connection information should be downloaded.

Table 85: `RsslRDMLoginRequest` Structure Members (Continued)

STRUCTURE MEMBER	DESCRIPTION
supportProviderDictionaryDownload	**Optional**. If present, `flags` value of **RDM_LG_RQF_HAS_SUPPORT_PROV_DIC_DOWNLOAD** should be specified. If absent, a default value of **0** is assumed. Indicates whether the ADH supports the Provider Dictionary Download feature, which allows the application to request RWFFld and RFFEnum dictionaries from ADH. • **1**: The ADH supports the Provider Dictionary Download feature. • **0**: The ADH does not support the Provider Dictionary Download feature. For details on the Provider Dictionary Download feature, refer to the *Transport API C Edition Developers Guide*.

Table 85: `RsslRDMLoginRequest` Structure Members (Continued)

7.3.1.2 RSSL RDM Login Request Flag Enumeration Values

FLAG ENUMERATION	MEANING
RDM_LG_RQF_HAS_ALLOW_SUSPECT_DATA	Indicates presence of `allowSuspectData`. If not present, a value of **1** should be assumed.
RDM_LG_RQF_HAS_APPLICATION_ID	Indicates presence of `applicationId`.
RDM_LG_RQF_HAS_APPLICATION_NAME	Indicates presence of `applicationName`.
RDM_LG_RQF_HAS_DOWNLOAD_CONN_CONFIG	Indicates presence of `downloadConnectionConfig`. If not present, a value of **0** should be assumed.
RDM_LG_RQF_HAS_INSTANCE_ID	Indicates presence of `instanceId`.
RDM_LG_RQF_HAS_PASSWORD	Indicates presence of `password`.
RDM_LG_RQF_HAS_POSITION	Indicates presence of `position`.
RDM_LG_RQF_HAS_PROVIDE_PERM_EXPR	Indicates presence of `providePermissionExpressions`. If not present, a value of **1** should be assumed.
RDM_LG_RQF_HAS_PROVIDE_PERM_PROFILE	Indicates presence of `providePermissionProfile`. If not present, a value of **1** should be assumed.
RDM_LG_RQF_HAS_ROLE	Indicates presence of `role`. If not present, a role of **RDM_LOGIN_ROLE_CONS** should be assumed.
RDM_LG_RQF_HAS_SINGLE_OPEN	Indicates presence of `singleOpen`. If not present, a value of 1 should be assumed.
RDM_LG_RQF_HAS_USERNAME_TYPE	Indicates presence of `userNameType`. If not present a `userNameType` of **RDM_LOGIN_USER_NAME** should be assumed.
RDM_LG_RQF_PAUSE_ALL	Indicates that the consumer would like to pause all streams associated with the logged in user. For more information on pause and resume behavior, refer to the *Transport API C Edition Developers Guide*.

Table 86: `RsslRDMLoginRequest` Flags

FLAG ENUMERATION	MEANING
RDM_LG_RQF_NO_REFRESH	Indicates that the consumer application does not require a login refresh for this request. This typically occurs when resuming a stream or changing a AAA token. In some instances, a provider may still see fit to deliver a refresh message, however if not explicitly asked for by the consumer it should be considered unsolicited.
RDM_LG_RQF_HAS_SUPPORT_PROV_DIC_DOWNLOAD	Indicates presence of **supportProviderDictionaryDownload**. If absent, a value of **0** should be assumed. For more information on Provider Dictionary Download, refer to the *Transport API C Edition Developers Guide*.

Table 86: RsslRDMLoginRequest Flags (Continued)

7.3.1.3 RSSL RDM Login Request Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMLoginRequest	Clears an **RsslRDMLoginRequest** structure. Useful for structure reuse.
rsslInitDefaultRDMLoginRequest	Clears an **RsslRDMLoginRequest** structure and populates **userName**, **position**, **applicationId**, and **applicationName** with default values.
rsslCopyRDMLoginRequest	Performs a deep copy of an **RsslRDMLoginRequest** structure.

Table 87: RsslRDMLoginRequest Utility Functions

7.3.2 RSSL RDM Login Refresh

A ***Login Refresh*** message is encoded and sent by OMM interactive provider applications. This message is used to respond to a Login Request message and to indicate that the user's Login is accepted. An OMM Provider can use the Login request information to authenticate users with DACS. After authentication, a refresh message is sent to convey that the login was accepted. If the login is rejected, a Login status message should be sent as described in Section 7.3.3.

The **RsslRDMLoginRefresh** represents all members of a login refresh message and allows for simplified use in OMM applications that leverage RDM. This structure follows the behavior and layout that is defined in the *Transport API C Edition RDM Usage Guide*.

7.3.2.1 RSSL RDM Login Refresh Structure Members

STRUCTURE MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Contains general message information like streamId and domainType.
flags	**Required**. Indicate presence of optional login refresh members. For details, see Section 7.3.2.2.

Table 88: RsslRDMLoginRefresh Structure Members

STRUCTURE MEMBER	DESCRIPTION
state	**Required**. Indicates the state of the login stream. Defaults to a `streamState` of **RSSL_STREAM_OPEN** and a `dataState` of **RSSL_DATA_OK**. For more information on `RsslState`, refer to the *Transport API C Edition Developers Guide*.
userNameType	If present, `flags` value of **RDM_LG_RFF_HAS_USERNAME_TYPE** should be specified. If absent, a default value of **RDM_LOGIN_USER_NAME** is assumed. Possible values: • **RDM_LOGIN_USER_NAME == 1** • **RDM_LOGIN_USER_EMAIL_ADDRESS == 2** • **RDM_LOGIN_USER_TOKEN == 3** A type of **RDM_LOGIN_USER_NAME** typically corresponds to a DACS user name. This can be used to authenticate and permission a user. **RDM_LOGIN_USER_TOKEN** is specified when using AAAAPI The user token is retrieved from a AAAAPI gateway. To validate users, a provider application can pass this user token to an Authentication Manager application. This type of token periodically changes: when it changes, an application can send a login reissue to pass information upstream. For more information, refer to specific AAAAPI documentation.
userName	If present, `flags` value of **RDM_LG_RFF_HAS_USERNAME** should be specified. When populated, this should match the `userName` contained in the login request.
applicationId	If present, `flags` value of **RDM_LG_RFF_HAS_APPLICATION_ID** should be specified. When populated, this should match the `applicationId` contained in the login request.
applicationName	If present, `flags` value of **RDM_LG_RFF_HAS_APPLICATION_NAME** should be specified. When populated, the `applicationName` in the login refresh identifies the OMM provider.
position	If present, `flags` value of **RDM_LG_RFF_HAS_POSITION** should be specified. When populated, this should match the `position` contained in the login request.
providePermissionProfile	If present, `flags` value of **RDM_LG_RFF_HAS_PROVIDE_PERM_PROFILE** should be specified. If absent, a default value of **1** is assumed. When **1**, this indicates that the permission profile is provided. The permission profile can be used by an application to perform proxy permissioning.
providePermissionExpressions	If present, `flags` value of **RDM_LG_RFF_HAS_PROVIDE_PERM_EXPR** should be specified. If absent, a default value of **1** is assumed. When **1**, this indicates a provider will provide permission expression information with responses. Permission expressions allow for items to be proxy permissioned by a consumer via content-based entitlements.

Table 88: `RsslRDMLoginRefresh` Structure Members (Continued)

STRUCTURE MEMBER	DESCRIPTION
singleOpen	If present, `flags` value of **RDM_LG_RFF_HAS_SINGLE_OPEN** should be specified. If absent, a default value of **1** is assumed. • **1**: Indicates the consumer application wants the provider to drive stream recovery. • **0**: Indicates that the consumer application will drive stream recovery.
allowSuspectData	If present, `flags` value of **RDM_LG_RFF_HAS_ALLOW_SUSPECT_DATA** should be specified. If absent, a default value of 1 is assumed. • **1**: Indicates that the consumer application allows for suspect `streamState` information. • **0**: Indicates that the consumer application prefers any suspect data to result in the stream being closed with an **RSSL_STREAM_CLOSED_RECOVER** state.
supportOMMPost	If present, `flags` value of **RDM_LG_RFF_HAS_SUPP_POST** should be specified. If absent, a default value of **0** is assumed. Indicates whether the provider supports OMM Posting: • **1**: The provider supports OMM Posting and the user is permissioned. • **0**: The provider supports the OMM Post feature, but the user is not permissioned. • If this element is not present, then the server does not support OMM Post feature. For more information on Posting, refer to the *Transport API C Edition Developers Guide*.
supportStandby	If present, `flags` value of **RDM_LG_RFF_HAS_SUPP_STANDBY** should be specified. If absent, a default value of **0** is assumed. Indicates whether the provider supports Warm Standby functionality. If supported, a provider can be told to run as an Active or a Standby server, where the Active will behave as usual. The Standby will respond to item requests only with the message header and will forward any state changing information. When informed of an Active's failure, the Standby begins sending responses and assumes Active functionality. • **1**: The provider can support a role of Active or Standby in a Warm Standby group. • **0**: The provider does not support warm standby functionality.
supportEnhancedSymbolList	If present, a `flags` value of **RDM_LG_RFF_HAS_SUPPORT_ENH_SL** should be specified. If absent, a default value of **0x0** is assumed. Advertises, via flags, additonal features that the provider supports for the **Symbol List** domain, such as providing data streams for the items present in a requested **Symbol List** item. • **0x0**: The provider does not support any Symbol List enhancements. • **0x1**: The provider supports providing Symbol List data streams. For more information on **Symbol List** requestable behaviors, refer to the *Transport API C Edition RDM Usage Guide*.

Table 88: `RsslRDMLoginRefresh` Structure Members (Continued)

STRUCTURE MEMBER	DESCRIPTION
supportBatchRequests	If present, `flags` value of **RDM_LG_RFF_HAS_SUPP_BATCH** should be specified. If absent, a default value of **0** is assumed. Indicates whether the provider supports batch functionality. Batch functionality allows a consumer to specify multiple items, all with matching attributes, in the same request message. • **1**: The provider supports batch requesting. • **0**: The provider does not support batch requesting. For more information on batch requesting, refer to the *Transport API C Edition Developers Guide*.
supportViewRequests	If present, `flags` value of **RDM_LG_RFF_HAS_SUPP_VIEW** should be specified. If absent, a default value of **0** is assumed. Indicates whether the provider supports Dynamic View functionality. A Dynamic View allows a user to request only the specific contents of the response information in which they are interested. • **1**: The provider supports Dynamic View functionality. • **0**: The provider does not support Dynamic View functionality. For more information on Dynamic View use, refer to the *Transport API C Edition Developers Guide*.
supportOptimizedPauseResume	If present, `flags` value of **RDM_LG_RFF_HAS_SUPP_OPT_PAR** should be specified. If not present, a default value of **0** is assumed. Indicates whether the provider supports Optimized Pause and Resume. Optimized Pause and Resume allows for pausing/resuming of individual item streams or pausing all item streams via a pause of the login stream. • **1**: The server supports optimized pause and resume. • **0**: The server does not support optimized pause and resume. For more information on Pause and Resume, refer to the *Transport API C Edition Developers Guide*.
supportProviderDictionaryDownload	If present, a `flags` value of **RDM_LG_RFF_HAS_SUPPORT_PROV_DIC_DOWNLOAD** should be specified. If absent, a default value of **0** is assumed. Indicates whether the ADH supports the Provider Dictionary Download feature, which allows a user to request RWFFld and RFFEnum dictionaries from ADH. • **1**: The ADH supports the Provider Dictionary Download feature. • **0**: The ADH does not support the Provider Dictionary Download feature. For more information on Provider Dictionary Download, refer to the *Transport API C Edition Developers Guide*.
numStandbyServers	If present, `flags` value of **RDM_LG_RFF_HAS_CONN_CONFIG** should be specified and the `serverList` member should also be specified. If not present, a default value of **0** is assumed. Indicates the number of servers in the `serverList` that the consumer is expected to use as standby servers when using Warm Standby functionality.

Table 88: `RsslRDMLoginRefresh` Structure Members (Continued)

STRUCTURE MEMBER	DESCRIPTION
serverCount	If present, **`flags`** value of **RDM_LG_RFF_HAS_CONN_CONFIG** should be specified and the **`serverList`** member should also be specified. If not present, a default value of **0** is assumed. Indicates the number of servers present in the **`serverList`** parameter.
serverList	If present, **`flags`** value of **RDM_LG_RFF_HAS_CONN_CONFIG** should be specified and the **`serverCount`** and **`numStandbyServers`** members should also be specified. An array of servers that the consumer may connect to when using Warm Standby functionality.
sequenceNumber	A user-specified, item-level sequence number which can be used by the application for sequencing messages within this stream.

Table 88: `RsslRDMLoginRefresh` Structure Members (Continued)

7.3.2.2 RSSL RDM Login Refresh Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
RDM_LG_RFF_HAS_ALLOW_SUSPECT_DATA	Indicates presence of **`allowSuspectData`**. If absent, a value of **1** should be assumed.
RDM_LG_RFF_HAS_APPLICATION_ID	Indicates presence of **`applicationId`**.
RDM_LG_RFF_HAS_APPLICATION_NAME	Indicates presence of **`applicationName`**.
RDM_LG_RFF_HAS_POSITION	Indicates presence of **`position`**.
RDM_LG_RFF_HAS_PROVIDE_PERM_EXPR	Indicates presence of **`providePermissionExpressions`**. If absent, a value of **1** should be assumed.
RDM_LG_RFF_HAS_PROVIDE_PERM_PROFILE	Indicates presence of **`providePermissionProfile`**. If absent, a value of **1** should be assumed.
RDM_LG_RFF_HAS_SINGLE_OPEN	Indicates presence of **`singleOpen`**. If absent, a value of **1** should be assumed.
RDM_LG_RFF_HAS_SUPP_BATCH	Indicates presence of **`supportBatchRequests`**. If absent, a value of **0** should be assumed. For more information on Batch functionality, refer to the *Transport API C Edition Developers Guide*.
RDM_LG_RFF_HAS_SUPP_POST	Indicates presence of **`supportOMMPost`**. If absent, a value of **0** should be assumed. For more information on Posting, refer to the *Transport API C Edition Developers Guide*.
RDM_LG_RFF_HAS_SUPPORT_PROV_DIC_DOWNLOAD	Indicates presence of **`supportProviderDictionaryDownload`**. If absent, a value of **0** should be assumed. For more information on Provider Dictionary Download, refer to the *Transport API C Edition Developers Guide*.

Table 89: `RsslRDMLoginRefresh` Flags

FLAG ENUMERATION	DESCRIPTION
RDM_LG_RFF_HAS_SUPP_OPT_PAR	Indicates presence of `supportOptimizedPauseResume`. If absent, a value of **0** should be assumed. For more information on Pause and Resume, refer to the *Transport API C Edition Developers Guide*.
RDM_LG_RFF_HAS_SUPP_VIEW	Indicates presence of `supportViewRequests`. If absent, a value of **0** should be assumed. For more information on View functionality, refer to the *Transport API C Edition Developers Guide*.
RDM_LG_RFF_HAS_SUPP_STANDBY	Indicates presence of `supportStandby`. If absent, a value of 0 should be assumed.
RDM_LG_RFF_SOLICITED	If this flag is present, it indicates that the login refresh is solicited (e.g., it is in response to a request). If the flag is absent, this refresh is unsolicited.
RDM_LG_RFF_HAS_USERNAME	Indicates presence of `userName`.
RDM_LG_RFF_HAS_USERNAME_TYPE	Indicates presence of `userNameType`. If absent, a `userNameType` of **RDM_LOGIN_USER_NAME** should be assumed.
RDM_LG_RFF_HAS_SEQ_NUM	Indicates presence of `numStandbyServers`, `serverCount`, and `serverList`.
RDM_LG_RFF_HAS_CONN_CONFIG	Indicates presence of the connection configuration information.
RDM_LG_RFF_CLEAR_CACHE	Indicates that any stored payload information associated with the login stream should be cleared. This may occur if some portion of data is known to be invalid.

Table 89: `RsslRDMLoginRefresh` Flags (Continued)

7.3.2.3 RSSL RDM Login Refresh Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMLoginRefresh	Clears an `RsslRDMLoginRefresh` structure. Useful for structure reuse.
rsslCopyRDMLoginRefresh	Performs a deep copy of an `RsslRDMLoginRefresh` structure.

Table 90: `RsslRDMLoginRefresh` Utility Functions

7.3.2.4 RSSL RDM Server Info Structure Members

STRUCTURE MEMBER	DESCRIPTION
flags	Required. Indicate presence of optional server info members. For details, refer to Section 7.3.2.5.
serverIndex	Required. Provides the index value to this server.

Table 91: RsslRDMServerInfo Structure Members

STRUCTURE MEMBER	DESCRIPTION
hostname	Required. The `hostname` information for this server.
port	Required. The port number to connect to for this server.
loadFactor	The load information for this server. If present, `flags` value of RDM_LG_RFFSIF_HAS_LOAD_FACTOR should be specified.
serverType	Indicates whether this server is an active or standby server. If present, specify a `flags` value of **RDM_LG_RFFSIF_HAS_TYPE**. Populated by `RDMLoginServerTypes`.

Table 91: RsslRDMServerInfo Structure Members (Continued)

7.3.2.5 RSSL RDM Server Info Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
RDM_LG_SIF_HAS_LOAD_FACTOR	Indicates presence of `loadFactor` information.
RDM_LG_SIF_HAS_TYPE	Indicates presence of `serverType`.

Table 92: `RsslRDMServerInfo` Flags

7.3.2.6 RSSL RDM Server Info Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMServerInfo	Clears an `RsslRDMServerInfo` structure. Useful for structure reuse.

Table 93: `RsslRDMServerInfo` Utility Functions

7.3.3 RSSL RDM Login Status

OMM Provider and OMM non-interactive provider applications use the ***Login Status*** message to convey state information associated with the login stream. Such state information can indicate that a login stream cannot be established or to inform a consumer of a state change associated with an open login stream.

The Login status message can also be used to reject a login request or close an existing login stream. When a login stream is closed via a status, any other open streams associated with the user are also closed as a result.

The `RsslRDMLoginStatus` represents all members of a login refresh message and allows for simplified use in OMM applications that leverage RDM. This structure follows the behavior and layout that is defined in the *Transport API C Edition RDM Usage Guide*.

7.3.3.1 RSSL RDM Login Status Structure Members

STRUCTURE MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Contains general message information like streamId and domainType.
flags	**Required**. Indicate presence of optional login status members. For details, refer to Section 7.3.3.2.

Table 94: `RsslRDMLoginStatus` Structure Members

STRUCTURE MEMBER	DESCRIPTION
state	If present, flags value of **RDM_LG_STF_HAS_STATE** should be specified. Indicates the state of the login stream. When rejecting a login the state should be: • `streamState` = **RSSL_STREAM_CLOSED** or **RSSL_STREAM_CLOSED_RECOVER** • `dataState` = **RSSL_DATA_SUSPECT** • `stateCode` = **RSSL_SC_NOT_ENTITLED** For more information on `RsslState`, refer to the *Transport API C Edition Developers Guide*.
userNameType	If present, `flags` value of **RDM_LG_STF_HAS_USERNAME_TYPE** should be specified. If not present, a default value of **RDM_LOGIN_USER_NAME** is assumed. Possible values: • **RDM_LOGIN_USER_NAME == 1** • **RDM_LOGIN_USER_EMAIL_ADDRESS == 2** • **RDM_LOGIN_USER_TOKEN == 3** A type of **RDM_LOGIN_USER_NAME** typically corresponds to a DACS user name. This can be used to authenticate and permission a user. **RDM_LOGIN_USER_TOKEN** is specified when using AAAAPI The user token is retrieved from an AAAAPI gateway. To validate users, a provider application can pass this user token to an Authentication Manager application. This type of token periodically changes: when it changes, an application can send a login reissue to pass information upstream. For more information, refer to specific AAAAPI documentation.
userName	If present, `flags` value of **RDM_LG_STF_HAS_USERNAME** should be specified. When populated, this should match the `userName` contained in the login request.

Table 94: `RsslRDMLoginStatus` Structure Members (Continued)

7.3.3.2 RSSL RDM Login Status Flag Enumeration Values

FLAG ENUMERATION	MEANING
RDM_LG_STF_HAS_STATE	Indicates presence of `state`. If not present, any previously conveyed state should continue to apply.
RDM_LG_STF_HAS_USERNAME	Indicates presence of `userName`.
RDM_LG_STF_HAS_USERNAME_TYPE	Indicates presence of `userNameType`. If not present a `userNameType` of **RDM_LOGIN_USER_NAME** should be assumed.

Table 95: `RsslRDMLoginStatus` Flags

7.3.3.3 RSSL RDM Login Status Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMLoginStatus	Clears an `RsslRDMLoginStatus` structure. Useful for structure reuse.

Table 96: `RsslRDMLoginStatus` Utility Functions

FUNCTION NAME	DESCRIPTION
rsslCopyRDMLoginStatus	Performs a deep copy of an `RsslRDMLoginStatus` structure.

Table 96: `RsslRDMLoginStatus` Utility Functions (Continued)

7.3.4 RSSL RDM Login Close

A ***Login Close*** message is encoded and sent by OMM consumer applications. This message allows a consumer to log out of the system. Closing a login stream is equivalent to a ***Close All*** type of message, where all open streams are closed (thus all other streams associated with the user are closed). A provider can log off a user and close all of that user's streams via a Login Status message, see Section 7.3.3.

7.3.4.1 RSSL RDM Login Close Structure Members

STRUCTURE MEMBER	DESCRIPTION
rdmMsgBase	Contains general message information like streamId and domainType.

Table 97: `RsslRDMLoginClose` Structure Members

7.3.4.2 RSSL RDM Login Close Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMLoginClose	Clears an `RsslRDMLoginClose` structure. Useful for structure reuse.
rsslCopyRDMLoginClose	Performs a deep copy of an `RsslRDMLoginClose` structure.

Table 98: `RsslRDMLoginClose` Utility Functions

7.3.5 RSSL RDM Consumer Connection Status

The ***Login Consumer Connection Status*** informs an interactive provider of its role in a ***Warm Standby*** group, either as an ***Active*** or ***Standby*** provider. When Active a provider behaves normally, however if a provider is Standby it responds to requests only with a message header (intended to allow a consumer application to confirm the availability of their requested data across active and standby servers), and forwards any state-related messages (i.e., unsolicited refresh messages, status messages). While in Standby mode, a provider should aggregate changes to item streams whenever possible. If the provider is changed from Standby to Active via this message, all aggregated update messages are passed along. When aggregation is not possible, a full, unsolicited refresh message is passed along.

The consumer application is responsible for ensuring that items are available and equivalent across all providers in a warm standby group. This includes managing state and availability differences as well as item group differences.

The `RsslRDMLoginConsumerConnectionStatus` relies on the `RsslGenericMsg` and represents all members necessary for applications that leverage RDM. This structure follows the behavior and layout that is defined in the *Transport API C Edition RDM Usage Guide*.

7.3.5.1 RSSL RDM Login Consumer Connection Status Structure Members

STRUCTURE MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Contains general message information like streamId and domainType.

Table 99: `RsslRDMLoginConsumerConnectionStatus` Structure Members

STRUCTURE MEMBER	DESCRIPTION
flags	**Required**. Indicate presence of optional login consumer connection status members. For details, refer to Section 7.3.5.2.
warmStandbyInfo	If present, flags value of **RDM_LG_CCSF_HAS_WARM_STANDBY_INFO** should be specified. Includes **`RsslRDMLoginWarmStandbyInfo`** to convey the state of the upstream provider. For details, refer to Section 7.3.5.3.

Table 99: `RsslRDMLoginConsumerConnectionStatus` Structure Members (Continued)

7.3.5.2 RSSL RDM Login Consumer Connection Status Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
RDM_LG_CCSF_HAS_WARM_STANDBY_INFO	Indicates presence of **`warmStandbyInfo`**.

Table 100: `RsslRDMLoginConsumerConnectionStatus` Flags

7.3.5.3 RSSL RDM Login Warm Standby Info Structure Members

STRUCTURE MEMBER	DESCRIPTION
action	**Required**. Indicates how a cache of Warm Standby content should apply this information. See the *Transport API C Edition Developers Guide* for information on **`RsslMapEntry`** actions.
warmStandbyMode	**Required**. Indicate presence of optional login consumer connection status members. For details, refer to Section 7.3.5.4.

Table 101: `RsslRDMLoginWarmStandbyInfo` Structure Members

7.3.5.4 RSSL RDM Login Warm Standby Mode Enumeration Values

ENUMERATION	DESCRIPTION
RDM_LOGIN_SERVER_TYPE_ACTIVE	Indicates that the server is acting as the **active** or primary server in a warm standby configuration.
RDM_LOGIN_SERVER_TYPE_STANDBY	Indicates that the server is acting as the **standby** or backup server in a warm standby configuration.

Table 102: `RDMLoginServerTypes` Enumeration Values

7.3.5.5 RSSL RDM Login Consumer Connection Status Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMLoginConsumerConnectionStatus	Clears an **`RsslRDMLoginConsumerConnectionStatus`** structure. Useful for structure reuse.
rsslClearRDMLoginWarmStandbyInfo	Clears the **`RsslRDMLoginWarmStandbyInfo`** structure.

Table 103: `RsslRDMLoginConsumerConnectionStatus` Utility Functions

FUNCTION NAME	DESCRIPTION
rsslCopyRDMLoginConsumerConnectionStatus	Performs a deep copy of an `RsslRDMLoginConsumerConnectionStatus` structure.

Table 103: `RsslRDMLoginConsumerConnectionStatus` Utility Functions (Continued)

7.3.6 Login Post Message Use

OMM consumer applications can encode and send data for any item via Post messages on the item's Login stream. This is known as off-stream posting because items are posted without using that item's dedicated stream. Posting an item on its own dedicated stream is referred to as on-stream posting.

When an application is off-stream posting, msgKey information is required on the RsslPostMsg. For more details on posting, refer to the *Transport API C Edition Developers Guide*.

7.3.7 Login Ack Message Use

OMM Provider applications encode and send Ack messages to acknowledge the receipt of Post messages. This message is used whenever a consumer posts and asks for acknowledgments. For more details on posting, see the *Transport API C Edition Developers Guide*.

7.3.8 RSSL RDM Login Message Union

This union can contain any of the RDM Login message types. This is provided for use with Login specific functionality.

7.3.8.1 RSSL RDM Login Union

UNION MEMBERS	DESCRIPTION
rdmMsgBase	The message base information.
request	The `RsslRDMLoginRequest` as described in Section 7.3.1.
close	The `RsslRDMLoginClose` as described in Section 7.3.4.
refresh	The `RsslRDMLoginRefresh` as described in Section 7.3.2.
status	The `RsslRDMLoginStatus` as described in Section 7.3.3.
consumerConnectionStatus	The `RsslRDMLoginConsumerConnectionStatus` as described in Section 7.3.5.

Table 104: `RsslRDMLoginMsg` Union Members

7.3.8.2 RSSL RDM Login Message Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMLoginMsg	Clears an `RsslRDMLoginMsg` union. Useful for reuse.
rsslCopyRDMLoginMsg	Performs a deep copy of an `RsslRDMLoginMsg` structure.

Table 105: `RsslRDMLoginMsg` Utility Functions

7.3.9 Login Encoding and Decoding

7.3.9.1 RSSL RDM Directory Login Encoding and Decoding Functions

FUNCTION NAME	DESCRIPTION
rsslEncodeRDMLoginMsg	Used to encode an RDM Login message. This function takes the `RsslRDMLoginMsg` as a parameter. Alternately, `rsslEncodeRDMMsg` can be used if encoding from an `RsslRDMMsg`.
rsslDecodeRDMLoginMsg	Used to decode an RDM Login message. This function populates the `RsslRDMLoginMsg` and leverages the Value Added Utility message buffer (refer to Section 8.2). Alternately, `rsslDecodeRDMMsg` can be used to decode into an `RsslRDMMsg`.

Table 106: RDM Login Encoding and Decoding Functions

7.3.9.2 Encoding a Login Request

```
RsslEncodeIterator encodeIter;
RsslRDMLoginRequest loginRequest;

/* Clear the Login Request structure. */
rsslClearRDMLoginRequest(&loginRequest);

/* Set flags indicating presence of optional members. */
loginRequest.flags =
    RDM_LG_RQF_HAS_APPLICATION_NAME
    | RDM_LG_RQF_HAS_APPLICATION_ID
    | RDM_LG_RQF_HAS_POSITION;

/* Set UserName. */
loginRequest.userName.data = "username";
loginRequest.userName.length = 8;

/* Set ApplicationName */
loginRequest.applicationName.data = "upa";
loginRequest.applicationName.length = 3;

/* Set ApplicationId */
loginRequest.applicationId.data = "256";
loginRequest.applicationId.length = 3;

/* Set Position */
loginRequest.position.data = "127.0.0.1/net";
loginRequest.position.length = 13;

/* Clear the encode iterator, set its RWF Version, and set it to a buffer for encoding into. */
rsslClearEncodeIterator(&encodeIter);
ret = rsslSetEncodeIteratorRWFVersion(&encodeIter, channelMajorVersion, channelMinorVersion);
ret = rsslSetEncodeIteratorBuffer(&encodeIter, &msgBuffer);

/* Encode the message. */
```

```
ret = rsslEncodeRDMMsg(&encodeIter, (RsslRDMMsg*)&loginRequest, &msgBuffer.length, &rsslErrorInfo);
```

Code Example 20: Login Request Encoding Example

7.3.9.3 Decoding a Login Request

```
/* The decoder may require additional space to store things such as lists. */
char memoryArray[1024];
RsslBuffer memoryBuffer = { 1024, memoryArray };

RsslDecodeIterator decodeIter;
RsslMsg msg;
RsslRDMMsg rdmMsg;
RsslRDMLoginRequest *pLoginRequest;

/* Clear the decode iterator, set its RWF Version, and set it to the encoded buffer. */
rsslClearDecodeIterator(&decodeIter);
ret = rsslSetDecodeIteratorRWFVersion(&decodeIter, channelMajorVersion, channelMinorVersion);
ret = rsslSetDecodeIteratorBuffer(&decodeIter, &msgBuffer);

/* Decode the message to an RsslMsg structure and RsslRDMMsg structure. */
ret = rsslDecodeRDMMsg(&decodeIter, &msg, &rdmMsg, &memoryBuffer, &rsslErrorInfo);

if (ret == RSSL_RET_SUCCESS
        && rdmMsg.rdmMsgBase.domainType == RSSL_DMT_LOGIN && rdmMsg.rdmMsgBase.rdmMsgType ==
        RDM_LG_MT_REQUEST)
{
    /* The message we decoded is an RsslRDMLoginRequest. */
    pLoginRequest = &rdmMsg.loginMsg.request;

    /* Print username. */
    printf("Username: %.*s\n", pLoginRequest->userName.length, pLoginRequest->userName.data);

    /* Print ApplicationName if present. */
    if (pLoginRequest->flags & RDM_LG_RQF_HAS_APPLICATION_NAME)
            printf("ApplicationName: %.*s\n", pLoginRequest->applicationName.length, pLoginRequest-
            >applicationName.data);

    /* Print ApplicationId if present. */
    if (pLoginRequest->flags & RDM_LG_RQF_HAS_APPLICATION_ID)
            printf("ApplicationId: %.*s\n", pLoginRequest->applicationId.length, pLoginRequest-
            >applicationId.data);

    /* Print Position if present. */
    if (pLoginRequest->flags & RDM_LG_RQF_HAS_POSITION)
            printf("Position: %.*s\n", pLoginRequest->position.length, pLoginRequest->position.data);
}
```

Code Example 21: Login Request Decoding Example

7.3.9.4 Encoding a Login Refresh

```
RsslEncodeIterator encodeIter;
RsslRDMLoginRefresh loginRefresh;

/* Clear the Login Refresh structure. */
rsslClearRDMLoginRefresh(&loginRefresh);

/* Set flags indicating presence of optional members. */
loginRefresh.flags =
    RDM_LG_RFF_HAS_USERNAME
    | RDM_LG_RFF_HAS_APPLICATION_NAME
    | RDM_LG_RFF_HAS_APPLICATION_ID
    | RDM_LG_RFF_HAS_POSITION;

/* Set UserName(should match request). */
loginRefresh.userName.data = "username";
loginRefresh.userName.length = 8;

/* Set ApplicationName(should match request). */
loginRefresh.applicationName.data = "upa";
loginRefresh.applicationName.length = 3;

/* Set ApplicationId(should match request). */
loginRefresh.applicationId.data = "256";
loginRefresh.applicationId.length = 3;

/* Set Position(should match request). */
loginRefresh.position.data = "127.0.0.1/net";
loginRefresh.position.length = 13;

/* Clear the encode iterator, set its RWF Version, and set it to a buffer for encoding into. */
rsslClearEncodeIterator(&encodeIter);
ret = rsslSetEncodeIteratorRWFVersion(&encodeIter, channelMajorVersion, channelMinorVersion);
ret = rsslSetEncodeIteratorBuffer(&encodeIter, &msgBuffer);

/* Encode the message. */
ret = rsslEncodeRDMMsg(&encodeIter, (RsslRDMMsg*)&loginRefresh, &msgBuffer.length, &rsslErrorInfo);
```

Code Example 22: Login Refresh Encoding Example

7.3.9.5 Decoding a Login Refresh

```
/* The decoder may require additional space to store things such as lists. */
char memoryArray[1024];
RsslBuffer memoryBuffer = { 1024, memoryArray };
```

```
RsslDecodeIterator decodeIter;
RsslMsg msg;
RsslRDMMsg rdmMsg;
RsslRDMLoginRefresh *pLoginRefresh;

/* Clear the decode iterator, set its RWF Version, and set it to the encoded buffer. */
rsslClearDecodeIterator(&decodeIter);
ret = rsslSetDecodeIteratorRWFVersion(&decodeIter, channelMajorVersion, channelMinorVersion);
ret = rsslSetDecodeIteratorBuffer(&decodeIter, &msgBuffer);

/* Decode the message to an RsslMsg structure and RsslRDMMsg structure. */
ret = rsslDecodeRDMMsg(&decodeIter, &msg, &rdmMsg, &memoryBuffer, &rsslErrorInfo);

if (ret == RSSL_RET_SUCCESS
        && rdmMsg.rdmMsgBase.domainType == RSSL_DMT_LOGIN && rdmMsg.rdmMsgBase.rdmMsgType ==
        RDM_LG_MT_REFRESH)
{
    /* The message we decoded is an RsslRDMLoginRefresh. */
    pLoginRefresh = &rdmMsg.loginMsg.refresh;

    /* Print username if present. */
    if (pLoginRefresh->flags & RDM_LG_RFF_HAS_APPLICATION_NAME)
        printf("Username: %.*s\n", pLoginRefresh->userName.length, pLoginRefresh->userName.data);

    /* Print ApplicationName if present. */
    if (pLoginRefresh->flags & RDM_LG_RFF_HAS_APPLICATION_NAME)
        printf("ApplicationName: %.*s\n", pLoginRefresh->applicationName.length, pLoginRefresh-
                >applicationName.data);

    /* Print ApplicationId if present. */
    if (pLoginRefresh->flags & RDM_LG_RFF_HAS_APPLICATION_ID)
        printf("ApplicationId: %.*s\n", pLoginRefresh->applicationId.length, pLoginRefresh-
                >applicationId.data);

    /* Print Position if present. */
    if (pLoginRefresh->flags & RDM_LG_RFF_HAS_POSITION)
        printf("Position: %.*s\n", pLoginRefresh->position.length, pLoginRefresh->position.data);
}
```

Code Example 23: Login Refresh Decoding Example

7.4 Source Directory Domain

The Source Directory domain model conveys:

- Information about all available services and their capabilities. This includes information about domain types supported within a service, the service's state, the QoS, and any item group information associated with the service. Each service is associated with a unique **`serviceId`**.

- Status information associated with item groups. This allows a single message to change the state of all associated items, avoiding the need to send a status message for each individual item. The consumer is responsible for applying any changes to its open items. For details, refer to Section 7.4.10.
- Source Mirroring information between an ADH and OMM interactive provider applications exchanged via a specifically-formatted generic message as described in Section 7.4.6.

7.4.1 RSSL RDM Directory Request

A ***Directory Request*** message is encoded and sent by OMM Consumer applications. This message is used to request information from an OMM Provider about available services. A consumer may request information about all services by omitting the **`serviceId`** member, or request information about a specific service by setting it to the ID of the desired service.

The **`RsslRDMDirectoryRequest`** represents all members of a directory request message and allows for simplified use in OMM applications that leverage RDM. This structure follows the behavior and layout that is defined in the *Transport API C Edition RDM Usage Guide*.

7.4.1.1 RSSL RDM Directory Request Structure Members

STRUCTURE MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Contains general message information like streamId and domainType.
flags	**Required**. Indicate presence of optional directory request members. For details, refer to Section 7.4.1.2.
serviceId	If present, **`flags`** value of **RDM_DR_RQF_HAS_SERVICE_ID** should be specified. • If not present, this indicates the consumer wants information about all available services. • If present, this indicates the consumer only wants information about the service that has this ID.
filter	**Required**. Should be populated with flags indicating what information about the service the consumer is interested in. The available flags are: • **RDM_DIRECTORY_SERVICE_INFO_FILTER == 0x01** • **RDM_DIRECTORY_SERVICE_STATE_FILTER == 0x02** • **RDM_DIRECTORY_SERVICE_GROUP_FILTER == 0x04** • **RDM_DIRECTORY_SERVICE_LOAD_FILTER == 0x08** • **RDM_DIRECTORY_SERVICE_DATA_FILTER == 0x10** • **RDM_DIRECTORY_SERVICE_LINK_FILTER == 0x20** In most cases, the **RDM_DIRECTORY_SERVICE_INFO_FILTER**, **RDM_DIRECTORY_SERVICE_STATE_FILTER**, and **RDM_DIRECTORY_SERVICE_GROUP_FILTER** should be set.

Table 107: `RsslRDMDirectoryRequest` Structure Members

7.4.1.2 RSSL RDM Directory Request Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
RDM_DR_RQF_HAS_SERVICE_ID	Indicates presence of **`serviceId`**.
RDM_DR_RQF_STREAMING	Indicates that the consumer wants to receive updates about directory information after the initial refresh.

Table 108: `RsslRDMDirectoryRequest` Flags

7.4.1.3 RSSL RDM Directory Request Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMDirectoryRequest	Clears an `RsslRDMDirectoryRequest` structure. Useful for structure reuse.
rsslInitDefaultRDMDirectoryRequest	Clears an `RsslRDMDirectoryRequest`, sets the structure to request all services and receive updates for them, and populates `filter` with default values.
rsslCopyRDMDirectoryRequest	Performs a deep copy of an `RsslRDMDirectoryRequest` structure.

Table 109: `RsslRDMDirectoryRequest` Utility Functions

7.4.2 RSSL RDM Directory Refresh

A ***Directory Refresh*** message is encoded and sent by OMM Provider and OMM non-interactive provider applications. This message may be used to provide information about services it supports.

The `RsslRDMDirectoryRefresh` represents all members of a directory refresh message and allows for simplified use in OMM applications that leverage RDM. This structure follows the behavior and layout that is defined in the *Transport API C Edition RDM Usage Guide*.

7.4.2.1 RSSL RDM Directory Refresh Structure Members

STRUCTURE MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Contains general message information like streamId and domainType.
flags	**Required**. Indicates presence of optional directory refresh members. Refer to Section 7.4.2.2.
state	**Required**. Indicates stream and data state information. See the *Transport API C Edition Developers Guide* for information about `RsslState`.
filter	**Required**. Indicates what information is being provided about supported services. This should match the `filter` of the consumer's `RsslRDMDirectoryRequest`. The available flags are: • **RDM_DIRECTORY_SERVICE_INFO_FILTER == 0x01** • **RDM_DIRECTORY_SERVICE_STATE_FILTER == 0x02** • **RDM_DIRECTORY_SERVICE_GROUP_FILTER == 0x04** • **RDM_DIRECTORY_SERVICE_LOAD_FILTER == 0x08** • **RDM_DIRECTORY_SERVICE_DATA_FILTER == 0x10** • **RDM_DIRECTORY_SERVICE_LINK_FILTER == 0x20**
serviceCount	**Required**. Indicates the number of services present in the `serviceList`.
serviceList	Presence indicated by `serviceCount`. Contains an array of information about available services.
serviceId	If present, `flags` value of **RDM_DR_RFF_HAS_SERVICE_ID** should be specified. This should match the `serviceId` of the consumer's `RsslRDMDirectoryRequest`.
sequenceNumber	If present, `flags` value of **RDM_DR_RFF_HAS_SEQ_NUM** should be specified. A user-specified, item-level sequence number which can be used by the application for sequencing messages within this stream.

Table 110: `RsslRDMDirectoryRefresh` Structure Members

7.4.2.2 RSSL RDM Directory Refresh Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
RDM_DR_RFF_HAS_SERVICE_ID	Indicates presence of `serviceId`.
RDM_DR_RFF_SOLICITED	If this flag is present, it indicates that the login refresh is solicited (e.g., it is in response to a request). If the flag is not present, this refresh is unsolicited.
RDM_DR_RFF_HAS_SEQ_NUM	Indicates presence of `sequenceNumber`.
RDM_DR_RFF_CLEAR_CACHE	Indicates that any stored payload information associated with the login stream should be cleared. This may occur if some portion of data is known to be invalid.

Table 111: `RsslRDMDirectoryRefresh` Flags

7.4.2.3 RSSL RDM Directory Refresh Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMDirectoryRefresh	Clears an `RsslRDMDirectoryRefresh` structure. Useful for structure reuse.
rsslCopyRDMDirectoryRefresh	Performs a deep copy of an `RsslRDMDirectoryRefresh` structure.

Table 112: `RsslRDMDirectoryRefresh` Utility Functions

7.4.3 RSSL RDM Directory Update

A ***Directory Update*** message is encoded and sent by OMM Provider and OMM non-interactive provider applications. This message may be used to provide about new or removed services, or changes to existing services.

The `RsslRDMDirectoryUpdate` represents all members of a directory update message and allows for simplified use in OMM applications that leverage RDM. This structure follows the behavior and layout that is defined in the *Transport API C Edition RDM Usage Guide*.

7.4.3.1 RSSL RDM Directory Update Structure Members

STRUCTURE MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Contains general message information like streamId and domainType.
flags	**Required**. Indicates presence of optional directory update members. For details refer to Section 7.4.3.2.
filter	If present, `flags` value of **RDM_DR_UPF_HAS_FILTER** should be specified. Indicates what information is being provided about supported services. This should match the `filter` of the consumer's `RsslRDMDirectoryRequest`. Available flags are: • **RDM_DIRECTORY_SERVICE_INFO_FILTER == 0x01** • **RDM_DIRECTORY_SERVICE_STATE_FILTER == 0x02** • **RDM_DIRECTORY_SERVICE_GROUP_FILTER == 0x04** • **RDM_DIRECTORY_SERVICE_LOAD_FILTER == 0x08** • **RDM_DIRECTORY_SERVICE_DATA_FILTER == 0x10** • **RDM_DIRECTORY_SERVICE_LINK_FILTER == 0x20**

Table 113: `RsslRDMDirectoryUpdate` Structure Members

STRUCTURE MEMBER	DESCRIPTION
serviceCount	**Required**. Indicates the number of services present in the `serviceList`.
serviceList	Presence indicated by `serviceCount`. Contains an array of information about available services.
serviceId	If present, `flags` value of **RDM_DR_UPF_HAS_SERVICE_ID** should be specified. This should match the `serviceId` of the consumer's `RsslRDMDirectoryRequest`.
sequenceNumber	If present, `flags` value of **RDM_DR_UPF_HAS_SEQ_NUM** should be specified. A user-specified, item-level sequence number which can be used by the application for sequencing messages within this stream.

Table 113: `RsslRDMDirectoryUpdate` Structure Members (Continued)

7.4.3.2 RSSL RDM Directory Update Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
RDM_DR_UPF_HAS_SERVICE_ID	Indicates presence of `serviceId`.
RDM_DR_UPF_HAS_FILTER	Indicates presence of `filter`.
RDM_DR_UPF_HAS_SEQ_NUM	Indicates presence of `sequenceNumber`.

Table 114: `RsslRDMDirectoryUpdate` Flags

7.4.3.3 RSSL RDM Directory Update Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMDirectoryUpdate	Clears an `RsslRDMDirectoryUpdate` structure. Useful for structure reuse.
rsslCopyRDMDirectoryUpdate	Performs a deep copy of an `RsslRDMDirectoryUpdate` structure.

Table 115: `RsslRDMDirectoryUpdate` Utility Functions

7.4.4 RSSL RDM Directory Status

OMM Provider and OMM non-interactive provider applications use the ***Directory Status*** message to convey state information associated with the directory stream. Such state information can indicate that a directory stream cannot be established or to inform a consumer of a state change associated with an open directory stream. The Directory Status message can also be used to close an existing directory stream.

The `RsslRDMDirectoryStatus` represents all members of a directory refresh message and allows for simplified use in OMM applications that leverage RDM. This structure follows the behavior and layout that is defined in the *Transport API C Edition RDM Usage Guide*.

7.4.4.1 RSSL RDM Directory Status Structure Members

STRUCTURE MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Contains general message information like streamId and domainType.

Table 116: `RsslRDMDirectoryStatus` Structure Members

STRUCTURE MEMBER	DESCRIPTION
flags	**Required**. Indicate presence of optional directory status members. For details, refer to Section 7.4.4.2.
filter	If present, `flags` value of **RDM_DR_STF_HAS_FILTER** should be specified. Indicates what information is being provided about supported services. This should match the `filter` of the consumer's `RsslRDMDirectoryRequest`. The available flags are: • **RDM_DIRECTORY_SERVICE_INFO_FILTER == 0x01** • **RDM_DIRECTORY_SERVICE_STATE_FILTER == 0x02** • **RDM_DIRECTORY_SERVICE_GROUP_FILTER == 0x04** • **RDM_DIRECTORY_SERVICE_LOAD_FILTER == 0x08** • **RDM_DIRECTORY_SERVICE_DATA_FILTER == 0x10** • **RDM_DIRECTORY_SERVICE_LINK_FILTER == 0x20**
state	If present, `flags` value of **RDM_DR_STF_HAS_STATE** should be specified. Indicates the state of the directory stream. For more information on `RsslState`, refer to the *Transport API C Edition Developers Guide*.
serviceId	If present, `flags` value of **RDM_DR_STF_HAS_SERVICE_ID** should be specified. This should match the `serviceId` of the consumer's `RsslRDMDirectoryRequest`.

Table 116: `RsslRDMDirectoryStatus` Structure Members (Continued)

7.4.4.2 RSSL RDM Directory Status Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
RDM_DR_STF_HAS_STATE	Indicates presence of `state`. If not present, any previously conveyed state should continue to apply.
RDM_DR_STF_HAS_FILTER	Indicates presence of `filter`.
RDM_DR_STF_HAS_SERVICE_ID	Indicates presence of `serviceId`.

Table 117: `RsslRDMDirectoryStatus` Flags

7.4.4.3 RSSL RDM Directory Status Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMDirectoryStatus	Clears an `RsslRDMDirectoryStatus` structure. Useful for structure reuse.
rsslCopyRDMDirectoryStatus	Performs a deep copy of an `RsslRDMDirectoryStatus` structure.

Table 118: `RsslRDMDirectoryStatus` Utility Functions

7.4.5 RSSL RDM Directory Close

A ***Directory Close*** message is encoded and sent by OMM consumer applications. This message allows a consumer to close an open directory stream. A provider can close the directory stream via a Directory Status message, refer to Section 7.4.4.

7.4.5.1 RSSL RDM Directory Close Structure Members

STRUCTURE MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Contains general message information like streamId and domainType.

Table 119: `RsslRDMDirectoryClose` Structure Members

7.4.5.2 RSSL RDM Directory Close Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMDirectoryClose	Clears an **`RsslRDMDirectoryClose`** structure. Useful for structure reuse.
rsslCopyRDMDirectoryClose	Performs a deep copy of an **`RsslRDMDirectoryClose`** structure.

Table 120: `RsslRDMDirectoryClose` Utility Functions

7.4.6 RSSL RDM Consumer Status

The ***Directory Consumer Status*** is sent by OMM Consumer applications to inform a service of how it is being used for ***Source Mirroring***. This message is primarily informational.

The **`RsslRDMDirectoryConsumerStatus`** relies on the **`RsslGenericMsg`** and represents all members necessary for applications that leverage RDM. This structure follows the behavior and layout that is defined in the *Transport API C Edition RDM Usage Guide*.

7.4.6.1 RSSL RDM Directory Consumer Status Structure Members

STRUCTURE MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Contains general message information like streamId and domainType.
consumerServiceStatusCount	**Required**. Indicates the number of services present in the **`serviceList`**.
consumerServiceStatusList	Presence indicated by **`consumerServiceStatusCount`**. Contains an array of **`RsslRDMConsumerStatusService`** structures.

Table 121: `RsslRDMDirectoryConsumerStatus` Structure Members

7.4.6.2 RSSL RDM Directory Consumer Status Service Structure Members

STRUCTURE MEMBER	DESCRIPTION
serviceId	**Required**. Indicates the service associated with this status.
action	**Required**. Indicates how a cache of Source Mirroring content should apply this information. For information on **`RsslMapEntry`** actions, refer to the *Transport API C Edition Developers Guide*.

Table 122: `RsslRDMConsumerStatusService` Structure Members

STRUCTURE MEMBER	DESCRIPTION
sourceMirroringMode	**Required**. Indicates how the consumer is using the service. The available enumerations are: • **RDM_DIRECTORY_SOURCE_MIRROR_MODE_ACTIVE_NO_STANDBY == 0**, • **RDM_DIRECTORY_SOURCE_MIRROR_MODE_ACTIVE_WITH_STANDBY == 1**, • **RDM_DIRECTORY_SOURCE_MIRROR_MODE_STANDBY == 2**

Table 122: `RsslRDMConsumerStatusService` Structure Members (Continued)

7.4.6.3 RSSL RDM Directory Consumer Status Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMDirectoryConsumerStatus	Clears an **`RsslRDMDirectoryConsumerStatus`** structure. Useful for structure reuse.
rsslClearRDMConsumerStatusService	Clears the **`RsslRDMConsumerStatusService`** structure.
rsslCopyRDMDirectoryConsumerStatus	Performs a deep copy of an **`RsslRDMDirectoryConsumerStatus`** structure.

Table 123: `RsslRDMDirectoryConsumerStatus` Utility Functions

7.4.7 Source Directory RDM Service

An RsslRDMService structure is used to convey information about a service. An array of **`RsslRDMServices`** forms the **`serviceList`** member of the **`RsslRDMDirectoryRefresh`** and **`RsslRDMDirectoryUpdate`** messages.

The members of an **`RsslRDMService`** represent the different filters used to categorize service information.

7.4.7.1 RSSL RDM Service Structure Members

STRUCTURE MEMBER	DESCRIPTION
flags	**Required**. Indicate presence of optional service members. For details, refer to Section 7.4.7.2.
action	**Required**. Indicates how a cache of the service should apply this information. See the *Transport API C Edition Developers Guide* for information on **`RsslMapEntry`** actions.
serviceId	**Required**. Indicates the service associated with this **`RsslRDMService`**.
info	If present, **`flags`** value of **RDM_SVCF_HAS_INFO** should be specified. Contains information related to the Source Directory Info Filter.
state	If present, **`flags`** value of **RDM_SVCF_HAS_STATE** should be specified. Contains information related to the Source Directory State Filter.
groupStateCount	**Required**. Indicates the number of elements present in **`groupStateList`**.
groupStateList	Presence indicated by **`groupStateCount`**. Contains an array of elements indicating changes to item groups. Represents the Source Directory Group filter.

Table 124: `RsslRDMService` Structure Members

STRUCTURE MEMBER	DESCRIPTION
load	If present, **`flags`** value of **RDM_SVCF_HAS_LOAD** should be specified. Contains information about the service's operating workload. Represents the Source Directory Load Filter.
data	If present, **`flags`** value of **RDM_SVCF_HAS_DATA** should be specified. Contains data that applies to the items requested from the service. Represents the Source Directory Data Filter.
linkInfo	If present, **`flags`** value of **RDM_SVCF_HAS_LINK** should be specified. Contains information about upstream sources that provide data to this service. Represents the Source Directory Link Filter.

Table 124: `RsslRDMService` Structure Members (Continued)

7.4.7.2 RSSL RDM Service Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
RDM_SVCF_HAS_INFO	Indicates presence of **`info`**.
RDM_SVCF_HAS_STATE	Indicates presence of **`state`**.
RDM_SVCF_HAS_LOAD	Indicates presence of **`load`**.
RDM_SVCF_HAS_DATA	Indicates presence of **`data`**.
RDM_SVCF_HAS_LINK	Indicates presence of **`linkInfo`**.

Table 125: `RsslRDMService` Flags

7.4.7.3 RSSL RDM Service Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMService	Clears an **`RsslRDMService`** structure. Useful for structure reuse.

Table 126: `RsslRDMService` Utility Functions

7.4.8 Source Directory RDM Service Info

An **`RsslRDMServiceInfo`** structure is used to convey information that identifies the service and the content it can provide. It represents the Source Directory Info filter. More information about the Info filter is available in the *Transport API C Edition RDM Usage Guide*.

7.4.8.1 RSSL RDM Service Info Members

STRUCTURE MEMBER	DESCRIPTION
flags	**Required**. Indicate presence of optional service info members. For details, refer to Section 7.4.8.2.

Table 127: `RsslRDMServiceInfo` Structure Members

STRUCTURE MEMBER	DESCRIPTION
action	**Required**. Indicates how a cache of the service info should apply this information. For information on **RsslFilterEntry** actions, refer to the *Transport API C Edition Developers Guide*.
serviceName	**Required**. Indicates the name of the service.
vendor	If present, **flags** value of **RDM_SVC_IFF_HAS_VENDOR** should be specified. Identifies the vendor of the data.
isSource	If present, **flags** value of **RDM_SVC_IFF_HAS_IS_SOURCE** should be specified. Indicates whether the service is provided directly by a source or represents a group of sources. • **1**: The service is provided directly by a source • **0**: The service represents a group of sources. If not present, a value of **0** is assumed.
dictionariesProvidedCount	If present, **flags** value of **RDM_SVC_IFF_HAS_DICTS_PROVIDED** should be specified. Indicates the number of elements present in **dictionariesProvided**.
dictionariesProvidedList	If present, **flags** value of **RDM_SVC_IFF_HAS_DICTS_PROVIDED** and the **dictionariesProvidedCount** should be specified. Contains an array of elements that identify dictionaries that can be requested from this service.
dictionariesUsedCount	If present, **flags** value of **RDM_SVC_IFF_HAS_DICTS_USED** should be specified. Indicates the number of elements present in **dictionariesUsed**.
dictionariesUsedList	If present, **flags** value of **RDM_SVC_IFF_HAS_DICTS_USED** and the **dictionariesUsedCount** should be specified. Contains an array of elements that identify dictionaries that are used when decoding data from this service.
qosCount	If present, **flags** value of **RDM_SVC_IFF_HAS_QOS** should be specified. Indicates the number of elements present in **qosList**.
qosList	If present, **flags** value of **RDM_SVC_IFF_HAS_DICTS_USED** and the **qosCount** should be specified. Contains an array of elements that identify Qualities of Service available.
itemList	If present, **flags** value of **RDM_SVC_IFF_HAS_ITEM_LIST** should be specified. Specifies a name that can be requested on the **RSSL_DMT_SYMBOL_LIST** domain to get a list of all items available from this service.
supportsQosRange	If present, **flags** value of **RDM_SVC_IFF_HAS_SUPPORT_QOS_RANGE** should be specified. Indicates whether this service supports specifying a range of Qualities of Service when requesting an item. For information, see the **qos** and **worstQos** members of the **RsslRequestMsg** in the *Transport API C Edition Developers Guide*. • **1**: QoS Range requests are supported. • **0**: QoS Range requests are not supported. If not present, a value of **0** is assumed.

Table 127: RsslRDMServiceInfo Structure Members (Continued)

STRUCTURE MEMBER	DESCRIPTION
supportsOutOfBandSnapshots	If present, `flags` value of **RDM_SVC_IFF_HAS_SUPPORT_OOB_SNAPSHOTS** should be specified. Indicates whether this service supports making snapshot requests even when the **OpenLimit** is reached. • **1**: QoS Range requests are allowed. • **0**: QoS Range requests are not allowed. If not present, a value of **1** is assumed.
acceptingConsumerStatus	If present, `flags` value of **RDM_SVC_IFF_HAS_ACCEPTING_CONS_STATUS** should be specified. Indicates whether this service supports accepting `RsslRDMDirectoryConsumerStatus` messages for Source Mirroring. • **1**: The service will accept Consumer Status messages. • **0**: The service will not accept Consumer Status messages. If not present, a value of **1** is assumed.

Table 127: `RsslRDMServiceInfo` Structure Members (Continued)

7.4.8.2 RSSL RDM Service Info Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
RDM_SVC_IFF_HAS_VENDOR	Indicates presence of `vendor`.
RDM_SVC_IFF_HAS_IS_SOURCE	Indicates presence of `isSource`
RDM_SVC_IFF_HAS_DICTS_PROVIDED	Indicates presence of `dictionariesProvidedList` and `dictionariesProvidedCount`.
RDM_SVC_IFF_HAS_DICTS_USED	Indicates presence of `dictionariesUsedList` and `dictionariesUsedCount`.
RDM_SVC_IFF_HAS_QOS	Indicates presence of `qosList` and `qosCount`.
RDM_SVC_IFF_HAS_SUPPORT_QOS_RANGE	Indicates presence of `supportsQosRange`.
RDM_SVC_IFF_HAS_ITEM_LIST	Indicates presence of `itemList`.
RDM_SVC_IFF_HAS_SUPPORT_OOB_SNAPSHOTS	Indicates presence of `supportsOutOfBandSnapshots`.
RDM_SVC_IFF_HAS_ACCEPTING_CONS_STATUS	Indicates presence of `acceptingConsumerStatus`.

Table 128: `RsslRDMServiceInfo` Flags

7.4.8.3 RSSL RDM Service Info Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMServiceInfo	Clears an `RsslRDMServiceInfo` structure. Useful for structure reuse.

Table 129: `RsslRDMServiceInfo` Utility Functions

7.4.9 Source Directory RDM Service State

An `RsslRDMServiceState` structure is used to convey information about the current state of a service. It represents the Source Directory State filter. More information about the State filter is available in the *Transport API C Edition RDM Usage Guide*.

7.4.9.1 RSSL RDM Service State Members

STRUCTURE MEMBER	DESCRIPTION
flags	**Required**. Indicate presence of optional service info members. For details refer to Section 7.4.9.2.
action	**Required**. Indicates how a cache of the service state should apply this information. See the Transport API C Edition Developers Guide for information on `RsslFilterEntry` actions.
serviceState	**Required**. Indicates whether the original provider of the data can respond to new requests. It may still be possible to make requests if indicated by `acceptingRequests`. • **1**: The original provider of the data is available. • **0**: The original provider of the data is not currently available.
acceptingRequests	If present, `flags` value of **RDM_SVC_STF_HAS_ACCEPTING_REQS** should be specified. Indicates whether the immediate provider (to which the consumer is directly connected) can handle the request. • **1**: The service will accept new requests. • **0**: The service is not currently accepting new requests.
status	If present, `flags` value of **RDM_SVC_STF_HAS_STATUS** should be specified. This status should be applied to all open items associated with this service.

Table 130: `RsslRDMServiceState` Structure Members

7.4.9.2 RSSL RDM Service State Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
RDM_SVC_STF_HAS_ACCEPTING_REQS	Indicates presence of `acceptingRequests`.
RDM_SVC_STF_HAS_STATUS	Indicates presence of `status`.

Table 131: `RsslRDMServiceState` Flags

7.4.9.3 RSSL RDM Service State Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMServiceState	Clears an `RsslRDMServiceState` structure. Useful for structure reuse.

Table 132: `RsslRDMServiceState` Utility Functions

7.4.10 Source Directory RDM Service Group State

An **`RsslRDMServiceGroupState`** structure is used to convey status and name changes for an item group. It represents the Source Directory Group filter. More information about the Group State filter is available in the *Transport API C Edition RDM Usage Guide*.

7.4.10.1 RSSL RDM Service Group State Members

STRUCTURE MEMBER	DESCRIPTION
flags	**Required**. Indicate presence of optional service info members. For details, refer to Section 7.4.10.2.
action	**Required**. Indicates how a cache of the service group state should apply this information. For information on **`RsslFilterEntry`** actions, refer to the *Transport API C Edition Developers Guide*.
group	**Required**. Identifies the name of the item group that is being changed.
mergedToGroup	If present, **`flags`** value of **RDM_SVC_GRF_HAS_MERGED_TO_GROUP** should be specified. Specifies the new name to which items with the group name specified by **`group`** should be changed.
status	If present, **`flags`** value of **RDM_SVC_GRF_HAS_STATUS** should be specified. This status should be applied to all open items associated with the group specified by **`group`**.

Table 133: `RsslRDMServiceGroupState` Structure Members

7.4.10.2 RSSL RDM Service Group State Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
RDM_SVC_GRF_HAS_MERGED_TO_GROUP	Indicates presence of **`mergedToGroup`**.
RDM_SVC_GRF_HAS_STATUS	Indicates presence of **`status`**.

Table 134: `RsslRDMServiceGroupState` Flags

7.4.10.3 RSSL RDM Service Group State Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMServiceGroupState	Clears an **`RsslRDMServiceGroupState`** structure. Useful for structure reuse.

Table 135: `RsslRDMServiceGroupState` Utility Functions

7.4.11 Source Directory RDM Service Load

An **`RsslRDMServiceLoad`** structure is used to convey the workload of a service. It represents the Source Directory Load filter. More information about the Service Load filter is available in the *Transport API C Edition RDM Usage Guide*.

7.4.11.1 RSSL RDM Service Load Members

STRUCTURE MEMBER	DESCRIPTION
flags	**Required**. Indicates presence of optional service info members. For details, refer to Section 7.4.11.2.
action	**Required**. Indicates how a cache of the service load should apply this information. For information on **RsslFilterEntry** actions, refer to the *Transport API C Edition Developers Guide*.
openLimit	If present, **flags** value of **RDM_SVC_LDF_HAS_OPEN_LIMIT** should be specified. Specifies the maximum number of streaming requests allowed for this service.
openWindow	If present, **flags** value of **RDM_SVC_LDF_HAS_OPEN_WINDOW** should be specified. Specifies the maximum number of outstanding requests (i.e., requests awaiting a refresh) that the service allows at any given time.
loadFactor	If present, **flags** value of **RDM_SVC_LDF_HAS_LOAD_FACTOR** should be specified. Indicates the current workload on the source providing the data. A higher load factor indicates a higher workload. For more information, refer to the *Transport API C Edition RDM Usage Guide*.

Table 136: RsslRDMServiceLoad Structure Members

7.4.11.2 RSSL RDM Service Load Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
RDM_SVC_LDF_HAS_OPEN_LIMIT	Indicates presence of **openLimit**.
RDM_SVC_LDF_HAS_OPEN_WINDOW	Indicates presence of **openWindow**.
RDM_SVC_LDF_HAS_LOAD_FACTOR	Indicates presence of **loadFactor**.

Table 137: RsslRDMServiceLoad Flags

7.4.11.3 RSSL RDM Service Load Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMServiceLoad	Clears an **RsslRDMServiceLoad** structure. Useful for structure reuse.

Table 138: RsslRDMServiceLoad Utility Functions

7.4.12 Source Directory RDM Service Data

An **RsslRDMServiceData** structure is used to convey data that should be applied to all items of a service. It represents the Source Directory Data filter. More information about the Data filter is available in the *Transport API C Edition RDM Usage Guide*.

7.4.12.1 RSSL RDM Service Data Members

STRUCTURE MEMBER	DESCRIPTION
flags	**Required**. Indicate presence of optional service data members. For details, refer to Section 7.4.12.2.
action	**Required**. Indicates how a cache of the service data should apply this information. For further details on `RsslFilterEntry` actions, refer to the *Transport API C Edition Developers Guide*.
type	If present, `flags` value of **RDM_SVC_DTF_HAS_DATA** should be specified. Indicates the type of content present in `data`. Available enumerations are: • **RDM_DIRECTORY_DATA_TYPE_TIME == 1** • **RDM_DIRECTORY_DATA_TYPE_ALERT == 2** • **RDM_DIRECTORY_DATA_TYPE_HEADLINE == 3** • **RDM_DIRECTORY_DATA_TYPE_STATUS == 4**
dataType	If present, `flags` value of **RDM_SVC_DTF_HAS_DATA** should be specified. Specifies the `RsslDataType` of the data. For information on `RsslDataTypes`, refer to the *Transport API C Edition Developers Guide*.
data	If present, `flags` value of **RDM_SVC_DTF_HAS_DATA** should be specified. Contains the encoded `RsslBuffer` representing the data. The type of the data is given by `dataType`.

Table 139: `RsslRDMServiceData` Structure Members

7.4.12.2 RSSL RDM Service Load Data Enumeration Values

FLAG ENUMERATION	DESCRIPTION
RDM_SVC_DTF_HAS_DATA	Indicates presence of `type`, `dataType`, and `data`.

Table 140: `RsslRDMServiceData` Flags

7.4.12.3 RSSL RDM Service Data Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMServiceData	Clears an `RsslRDMServiceData` structure. Useful for structure reuse.

Table 141: `RsslRDMServiceData` Utility Functions

7.4.13 Source Directory RDM Service Link Information

An `RsslRDMServiceLinkInfo` structure is used to convey information about upstream sources that form a service. It represents the Source Directory Link filter. More information about the Service Link filter content is available in the *Transport API C Edition RDM Usage Guide*.

The `RsslRDMServiceLinkInfo` structure contains an array of `RsslRDMServiceLink` structures that each represents an upstream source.

7.4.13.1 RSSL RDM Service Link Info Members

STRUCTURE MEMBER	DESCRIPTION
action	**Required**. Indicates how a cache of the service link information should apply this information. For further information on `RsslFilterEntry` actions, refer to the *Transport API C Edition Developers Guide*.
linkCount	**Required**. Indicates the number of link elements present in `linkList`.
linkList	Presence indicated by `linkCount`. Contains an array of `RsslRDMServiceLink` structures that each represents a source.

Table 142: `RsslRDMServiceLinkInfo` Structure Members

7.4.13.2 RSSL RDM Service Link Info Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMServiceLinkInfo	Clears an `RsslRDMServiceLinkInfo` structure. Useful for structure reuse.

Table 143: `RsslRDMServiceLinkInfo` Utility Functions

7.4.14 Source Directory RDM Service Link

An `RsslRDMServiceLink` structure is used to convey information about an upstream source. It represents an entry in the Source Directory Link filter and is used by the `linkList` member of the `RsslRDMServiceLinkInfo` structure. More information about the Service Link filter content is available in the *Transport API C Edition RDM Usage Guide*.

7.4.14.1 RSSL RDM Service Link Members

STRUCTURE MEMBER	DESCRIPTION
flags	**Required**. Indicate presence of optional service link members. For details, refer to Section 7.4.14.2.
action	**Required**. Indicates how a cache of the service link should apply this information. For information on `RsslMapEntry` actions, refer to the *Transport API C Edition Developers Guide*.
name	**Required**. Specifies the name of the source. Sources with identical names may indicate sources that are load-balanced.
type	If present, `flags` value of **RDM_SVC_LKF_HAS_TYPE** should be specified. Specifies if the source is interactive or broadcast. Available enumerations are: • **RDM_DIRECTORY_LINK_TYPE_INTERACTIVE == 1** • **RDM_DIRECTORY_LINK_TYPE_BROADCAST == 2**
linkState	**Required**. Indicates whether the source is up or down.

Table 144: `RsslRDMServiceLink` Structure Members

STRUCTURE MEMBER	DESCRIPTION
linkCode	If present, `flags` value of **RDM_SVC_LKF_HAS_CODE** should be specified. Indicates additional information about the status of a source. Available enumerations are: • **RDM_DIRECTORY_LINK_CODE_NONE == 0** • **RDM_DIRECTORY_LINK_CODE_OK == 1** • **RDM_DIRECTORY_LINK_CODE_RECOVERY_STARTED == 2** • **RDM_DIRECTORY_LINK_CODE_RECOVERY_COMPLETED == 3**
text	If present, `flags` value of **RDM_SVC_LKF_HAS_TEXT** should be specified. Further describes the status of a source.

Table 144: `RsslRDMServiceLink` Structure Members (Continued)

7.4.14.2 RSSL RDM Service Link Enumeration Values

FLAG ENUMERATION	DESCRIPTION
RDM_SVC_LKF_HAS_TYPE	Indicates presence of `type`.
RDM_SVC_LKF_HAS_CODE	Indicates presence of `code`.
RDM_SVC_LKF_HAS_TEXT	Indicates presence of `text`.

Table 145: `RsslRDMServiceLink` Flags

7.4.14.3 RSSL RDM Service Link Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMServiceLink	Clears an `RsslRDMServiceLink` structure. Useful for structure reuse.

Table 146: `RsslRDMServiceLink` Utility Functions

7.4.15 Source Directory RDM Sequenced Multicast Information

An `RsslRDMServiceSeqMcastInfo` structure is included in the services advertised by the Reference Data Server component of an Elektron Direct Feed (EDF) system. It identifies components in the system to which an OMM Consumer application connects for content.

- For further information on the service sequenced multicast information filter, refer to the *Transport API C Edition RDM Usage Guide*.
- For further information on Elektron Direct Feed, refer to the *EDF Developers Guide*.

7.4.15.1 RSSL RDM Service Sequenced Multicast Information Structure

STRUCTURE MEMBER	DESCRIPTION
flags	**Required**. Sets any optional members. For details, refer to Section 7.4.15.2.
action	**Required**. Sets how a cache should apply this information. For details, refer to the *Transport API C Edition Developers Guide*.

Table 147: `RsslRDMServiceSeqMcastInfo` Structure Members

STRUCTURE MEMBER	DESCRIPTION
snapshotServer	Sets the network address/port information for the Snapshot Server, if one is present.
gapRecoveryServer	Sets the network address/port information for the Gap Recovery Server, if one is present.
refDataServer	Sets the network address/port information for the Reference Data Server, if one is present.
StreamingMCastChanServerCount	The number of real time stream components in `StreamingMCastChanServerList`.
StreamingMCastChanServerList	Sets the network address/port information for real time stream components.
GapMCastChanServerCount	Number of Gap Fill Server components in `GapMCastServerList`.
GapMCastChanServerList	Sets the network address/port information for Gap Fill Server components.

Table 147: `RsslRDMServiceSeqMcastInfo` Structure Members (Continued)

7.4.15.2 RSSL RDM Service Sequenced Multicast Info Enumeration Values

FLAG ENUMERATION	DESCRIPTION
RDM_SVC_SMF_HAS_SNAPSHOT_SERV	Indicates the presence of `snapshotServer`.
RDM_SVC_SMF_HAS_GAP_REC_SERV	Indicates the presence of `gapRecoveryServer`.
RDM_SVC_SMF_HAS_REF_DATA_SERV	Indicates the presence of `refDataServer`.
RDM_SVC_SMF_HAS_SMC_SERV	Indicates the presence of `StreamingMCastChanServerList`.
RDM_SVC_SMF_HAS_GMC_SERV	Indicates the presence of `GapMCastChanServerList`.

Table 148: RSSL RDM Service Sequenced Multicast Info Enumeration Values

7.4.15.3 RSSL RDM Address/Port Information

STRUCTURE MEMBER	DESCRIPTION
address	The network address of the component.
port	The network port of the component.
domain	The item domain associated with this component (e.g.: 6 (MarketPrice)).

Table 149: RSSL RDM Address/Port Information Structure Members

7.4.15.4 RSSL RDM Sequenced Multicast Info Utility Functions

UTILITY	DESCRIPTION
rsslClearRDMMCAddressPortInfo	Clears an `RsslRDMAddressPortInfo` structure.
rsslClearRDMServiceSeqMCastInfo	Clears an `RsslRDMServiceSeqMCastInfo` structure.

Table 150: `RsslRDMServiceSeqMcastInfo` Utility Functions

7.4.16 RSSL RDM Directory Message Union

This union can contain any of the RDM Directory message types. This is provided for use with directory-specific functionality.

7.4.16.1 RSSL RDM Directory Union

UNION MEMBERS	DESCRIPTION
rdmMsgBase	The message base information.
request	The `RsslRDMDirectoryRequest` as described in Section 7.4.1.
close	The `RsslRDMDirectoryClose` as described in Section 7.4.5.
refresh	The `RsslRDMDirectoryRefresh` as described in Section 7.4.2.
status	The `RsslRDMDirectoryStatus` as described in Section 7.4.4.
update	The `RsslRDMDirectoryUpdate` as described in Section 7.4.3.
consumerStatus	The `RsslRDMDirectoryConsumerStatus` as described in Section 7.4.6.

Table 151: `RsslRDMDirectoryMsg` Union Members

7.4.16.2 RSSL RDM Directory Message Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMDirectoryMsg	Clears an `RsslRDMDirectoryMsg` union. Useful for reuse.
rsslCopyRDMDirectoryMsg	Performs a deep copy of an `RsslRDMDirectoryMsg` structure.

Table 152: `RsslRDMDirectoryMsg` Utility Functions

7.4.17 Source Directory Encoding and Decoding

7.4.17.1 RSSL RDM Directory Encoding and Decoding Functions

FUNCTION NAME	DESCRIPTION
rsslEncodeRDMDirectoryMsg	Used to encode an RDM Directory message. This function takes the `RsslRDMDirectoryMsg` as a parameter. Alternately, `rsslEncodeRDMMsg` can be used if encoding from an `RsslRDMMsg`.
rsslDecodeRDMDirectoryMsg	Used to decode an RDM Directory message. This function populates the `RsslRDMDirectoryMsg` and leverages the Value Added Utility message buffer (refer to Section 8.2). Alternately, `rsslDecodeRDMMsg` can be used to decode into an `RsslRDMMsg`.

Table 153: RDM Directory Encoding and Decoding Functions

7.4.17.2 Encoding a Source Directory Request

```
RsslEncodeIterator encodeIter;
RsslRDMDirectoryRequest directoryRequest;
```

```
/* Clear the Directory Request structure. */
rsslClearRDMDirectoryRequest(&directoryRequest);

/* Set flags indicating presence of optional members. */
directoryRequest.flags =
    RDM_DR_RQF_HAS_SERVICE_ID
    | RDM_DR_RQF_STREAMING;

/* Set Service ID. */
directoryRequest.serviceId = 273;

/* Set ApplicationName. */
directoryRequest.filter =
    RDM_DIRECTORY_SERVICE_INFO_FILTER
    | RDM_DIRECTORY_SERVICE_STATE_FILTER
    | RDM_DIRECTORY_SERVICE_GROUP_FILTER;

/* Clear the encode iterator, set its RWF Version, and set it to a buffer for encoding into. */
rsslClearEncodeIterator(&encodeIter);
ret = rsslSetEncodeIteratorRWFVersion(&encodeIter, channelMajorVersion, channelMinorVersion);
ret = rsslSetEncodeIteratorBuffer(&encodeIter, &msgBuffer);

/* Encode the message. */
ret = rsslEncodeRDMMsg(&encodeIter, (RsslRDMMsg*)&directoryRequest, &msgBuffer.length,
        &rsslErrorInfo);
```

Code Example 24: Directory Request Encoding Example

7.4.17.3 Decoding a Source Directory Request

```
/* The decoder may require additional space to store things such as lists. */
char memoryArray[1024];
RsslBuffer memoryBuffer = { 1024, memoryArray };

RsslDecodeIterator decodeIter;
RsslMsg msg;
RsslRDMMsg rdmMsg;
RsslRDMDirectoryRequest *pDirectoryRequest;

/* Clear the decode iterator, set its RWF Version, and set it to the encoded buffer. */
rsslClearDecodeIterator(&decodeIter);
ret = rsslSetDecodeIteratorRWFVersion(&decodeIter, channelMajorVersion, channelMinorVersion);
ret = rsslSetDecodeIteratorBuffer(&decodeIter, &msgBuffer);

/* Decode the message to an RsslMsg structure and RsslRDMMsg structure. */
ret = rsslDecodeRDMMsg(&decodeIter, &msg, &rdmMsg, &memoryBuffer, &rsslErrorInfo);

if (ret == RSSL_RET_SUCCESS
        && rdmMsg.rdmMsgBase.domainType == RSSL_DMT_SOURCE && rdmMsg.rdmMsgBase.rdmMsgType ==
        RDM_DR_MT_REQUEST)
{
    /* The message we decoded is an RsslRDMDirectoryRequest. */
    pDirectoryRequest = &rdmMsg.directoryMsg.request;

    /* Print if Info filter was requested. */
    if (pDirectoryRequest->filter & RDM_DIRECTORY_SERVICE_INFO_FILTER)
        printf("Info filter requested.\n");

    /* Print if State filter was requested. */
    if (pDirectoryRequest->filter & RDM_DIRECTORY_SERVICE_STATE_FILTER)
        printf("State filter requested.\n");

    /* Print if Group filter was requested. */
    if (pDirectoryRequest->filter & RDM_DIRECTORY_SERVICE_GROUP_FILTER)
        printf("Group filter requested.\n");

    /* Print service ID if present. */
    if (pDirectoryRequest->flags & RDM_DR_RQF_HAS_SERVICE_ID)
        printf("Service ID: %u\n", pDirectoryRequest->serviceId);
```

Code Example 25: Directory Request Decoding Example

7.4.17.4 Encoding a Source Directory Refresh

```
RsslEncodeIterator encodeIter;
RsslRDMDirectoryRefresh directoryRefresh;

/* List of services to be used.
 * This example will show encoding of one service. Additional services
 * can be set up using the same method shown below. */

RsslRDMService serviceList[1];
/* Lists to be used with MY_SERVICE. */
RsslUInt capabilitiesList[3];
RsslBuffer dictionariesList[2];
RsslQos qosList[2];

/* Clear the Directory Refresh structure. */
rsslClearRDMDirectoryRefresh(&directoryRefresh);

/* Set flags */
directoryRefresh.flags = RDM_DR_RFF_SOLICITED;

/* Set state. */
directoryRefresh.state.streamState = RSSL_STREAM_OPEN;
directoryRefresh.state.dataState = RSSL_DATA_OK;
directoryRefresh.state.code = RSSL_SC_NONE;

/* Set filter to say the Info, State, and Group filters are supported. */
directoryRefresh.filter =
    RDM_DIRECTORY_SERVICE_INFO_FILTER
    | RDM_DIRECTORY_SERVICE_STATE_FILTER
    | RDM_DIRECTORY_SERVICE_GROUP_FILTER;

/*** Build Service MY_SERVICE. ***/

rsslClearRDMService(&serviceList[0]);

/* Set flags to indicate Info and State filter are present. */
serviceList[0].flags =
    RDM_SVCF_HAS_INFO
    | RDM_SVCF_HAS_STATE;

/* Set action to indicate adding a new service. */
serviceList[0].info.action = RSSL_MPEA_ADD_ENTRY;

/*** Build Info for MY_SERVICE. ***/

/* Set flags to indicate optional members. */
serviceList[0].info.flags =
    RDM_SVC_IFF_HAS_VENDOR
    | RDM_SVC_IFF_HAS_DICTS_PROVIDED
    | RDM_SVC_IFF_HAS_DICTS_USED
    | RDM_SVC_IFF_HAS_QOS;

/* Set service name. */
serviceList[0].info.serviceName.data = "MY_SERVICE";
serviceList[0].info.serviceName.length = 10;

/* Set vendor name. */
serviceList[0].info.vendor.data = "Thomson Reuters";
serviceList[0].info.vendor.length = 15;

/* Build capabilities list. */
capabilitiesList[0] = RSSL_DMT_DICTIONARY;
capabilitiesList[1] = RSSL_DMT_MARKET_PRICE;
capabilitiesList[2] = RSSL_DMT_MARKET_BY_ORDER;

/* Set capabilities list. */
serviceList[0].info.capabilitiesList = capabilitiesList;
serviceList[0].info.capabilitiesCount = 3;

/* Build dictionary list to use with dictionariesProvidedList and dictionariesUsedList. */
dictionariesList[0].data = "RWFFld";
dictionariesList[0].length = 6;
dictionariesList[1].data = "RWFEnum";
```

```
dictionariesList[1].length = 7;

/* Set dictionaries provided. */
serviceList[0].info.dictionariesProvidedList = dictionariesList;
serviceList[0].info.dictionariesProvidedCount = 2;

/* Set dictionaries used. */
serviceList[0].info.dictionariesUsedList = dictionariesList;
serviceList[0].info.dictionariesUsedCount = 2;

/* Build QoS list. */
qosList[0].timeliness = RSSL_QOS_TIME_REALTIME;
qosList[0].rate = RSSL_QOS_RATE_TICK_BY_TICK;
qosList[1].timeliness = RSSL_QOS_TIME_REALTIME;
qosList[1].rate = RSSL_QOS_RATE_JIT_CONFLATED;

/* Set QoS list. */
serviceList[0].info.qosList = qosList;
serviceList[0].info.qosCount = 2;

/*** Build Service State for MY_SERVICE ***/
serviceList[0].state.flags = RDM_SVC_STF_HAS_ACCEPTING_REQS;
serviceList[0].state.serviceState = 1;
serviceList[0].state.acceptingRequests = 1;

/*** Finish and encode. ***/

/* Set the array of services on the message. The refresh will information about 2 services*/
directoryRefresh.serviceList = serviceList;
directoryRefresh.serviceCount = 1;

/* Clear the encode iterator, set its RWF Version, and set it to a buffer for encoding into. */
rsslClearEncodeIterator(&encodeIter);
ret = rsslSetEncodeIteratorRWFVersion(&encodeIter, channelMajorVersion, channelMinorVersion);
ret = rsslSetEncodeIteratorBuffer(&encodeIter, &msgBuffer);

/* Encode the message. */
ret = rsslEncodeRDMMsg(&encodeIter, (RsslRDMMsg*)&directoryRefresh, &msgBuffer.length,
        &rsslErrorInfo);
```

Code Example 26: Directory Refresh Decoding Example

7.4.17.5 Decoding a Source Directory Refresh

```
/* The decoder may require additional space to store things such as lists. */
char memoryArray[4096];
RsslBuffer memoryBuffer = { 4096, memoryArray };
```

```
RsslDecodeIterator decodeIter;
RsslMsg msg;
RsslRDMMsg rdmMsg;
RsslRDMDirectoryRefresh *pDirectoryRefresh;

/* Clear the decode iterator, set its RWF Version, and set it to the encoded buffer. */
rsslClearDecodeIterator(&decodeIter);
ret = rsslSetDecodeIteratorRWFVersion(&decodeIter, channelMajorVersion, channelMinorVersion);
ret = rsslSetDecodeIteratorBuffer(&decodeIter, &msgBuffer);

/* Decode the message to an RsslMsg structure and RsslRDMMsg structure. */
ret = rsslDecodeRDMMsg(&decodeIter, &msg, &rdmMsg, &memoryBuffer, &rsslErrorInfo);

if (ret == RSSL_RET_SUCCESS && rdmMsg.rdmMsgBase.domainType == RSSL_DMT_SOURCE &&
        rdmMsg.rdmMsgBase.rdmMsgType == RDM_DR_MT_REFRESH)
{
    RsslUInt32 i;

    /* The message we decoded is an RsslRDMDirectoryRefresh. */
    pDirectoryRefresh = &rdmMsg.directoryMsg.refresh;

    /* Print serviceId if present. */
    if (pDirectoryRefresh->flags & RDM_DR_RFF_HAS_SERVICE_ID)
        printf("Service ID: %u\n", pDirectoryRefresh->serviceId);

    /* Print information about each service present in the refresh. */
    for(i = 0; i < pDirectoryRefresh->serviceCount; ++i)
    {
        /* Print Service Info if present */
        if (pDirectoryRefresh->serviceList[i].flags & RDM_SVCF_HAS_INFO)
        {
            RsslUInt32 j;
            RsslRDMServiceInfo *pInfo = &pDirectoryRefresh->serviceList[i].info;

            /* Print service name. */
            printf("Service Name: %.*s\n", pInfo->serviceName.length, pInfo->serviceName.data);

            /* Print vendor name if present. */
            if (pInfo->flags & RDM_SVC_IFF_HAS_VENDOR)
                printf("Vendor: %.*s\n", pInfo->vendor.length, pInfo->vendor.data);

            /* Print supported domains if present. */
            for (j = 0; j < pInfo->capabilitiesCount; ++j)
                printf("Capability: %s\n", rsslDomainTypeToString(pInfo->capabilitiesList[j]));

            /* Print dictionaries provided if present. */
            if (pInfo->flags & RDM_SVC_IFF_HAS_DICTS_PROVIDED)
            {
                for (j = 0; j < pInfo->dictionariesProvidedCount; ++j)
                printf("Dictionary Provided: %.*s\n", pInfo->dictionariesProvidedList[j].length,
                        pInfo->dictionariesProvidedList[j].data);
            }

            /* Print dictionaries used if present. */
            if (pInfo->flags & RDM_SVC_IFF_HAS_DICTS_USED)
            {
                for (j = 0; j < pInfo->dictionariesUsedCount; ++j)
                printf("Dictionary Used: %.*s\n", pInfo->dictionariesUsedList[j].length,
                        pInfo->dictionariesUsedList[j].data);
            }

            /* Print qualities of service supported if present. */
            if (pInfo->flags & RDM_SVC_IFF_HAS_QOS)
            {
                for (j = 0; j < pInfo->qosCount; ++j)
                printf("QoS: %s,%s\n", rsslQosTimelinessToString(pInfo->
                        qosList[j].timeliness),rsslQosRateToString(pInfo->qosList[j].rate));
            }
        }

        /* Print Service State if present */
        if (pDirectoryRefresh->serviceList[i].flags & RDM_SVCF_HAS_STATE)
        {
            RsslRDMServiceState *pState = &pDirectoryRefresh->serviceList[i].state;

            printf("Service State: %llu\n", pState->serviceState);

            if (pState->flags & RDM_SVC_STF_HAS_ACCEPTING_REQS)
                printf("Accepting Requests: %llu\n", pState->acceptingRequests);

        }
    }
}
```

Code Example 27: Directory Refresh Decoding Example

7.5 Dictionary Domain

The Dictionary domain model conveys information for parsing data. Dictionaries provide additional information about data, such as information necessary for decoding the content of an `RsslFieldEntry` or additional content related to its `fieldId`. The *Transport API C Edition RDM Usage Guide* has more information about the different types of dictionaries and their usage.

The structures provided for this domain provide easier usage of the existing utilities for encoding, decoding, and caching dictionary information. For more information on these utilities, see the *Transport API C Edition RDM Usage Guide*.

7.5.1 RSSL RDM Dictionary Request

A ***Dictionary Request*** message is encoded and sent by OMM Consumer applications. This message is used to request a dictionary from a service.

The `RsslRDMDictionaryRequest` represents all members of a dictionary request message and allows for simplified use in OMM applications that leverage RDM. This structure follows the behavior and layout that is defined in the *Transport API C Edition RDM Usage Guide*.

7.5.1.1 RSSL RDM Dictionary Request Structure Members

STRUCTURE MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Contains general message information like streamId and domainType.
flags	**Required**. Indicates presence of optional login request members. For details, refer to Section 7.5.1.2.
serviceId	**Required**. Specifies the service from which to request the dictionary.
verbosity	**Required**. Indicates the desired amount of information desired from the dictionary. The available enumerations are: • **RDM_DICTIONARY_INFO == 0x00**: Version information only • **RDM_DICTIONARY_MINIMAL == 0x03**: Provides information needed for caching • **RDM_DICTIONARY_NORMAL == 0x07**: Provides all information needed for decoding • **RDM_DICTIONARY_VERBOSE == 0x0F**: Provides all information (including comments) Providers are not required to support the **MINIMAL** and **VERBOSE** filters.
dictionaryName	**Required**. Indicates the name of the dictionary being requested.

Table 154: `RsslRDMDictionaryRequest` Structure Members

7.5.1.2 RSSL RDM Login Request Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
RDM_DC_RQF_STREAMING	Indicates the dictionary stream should remain open after the initial refresh. An open stream may be used to listen for status messages that indicate changes to the dictionary version. For more information, see the *Transport API C Edition RDM Usage Guide*.

Table 155: `RsslRDMDictionaryRequest` Flags

7.5.1.3 RSSL RDM Dictionary Request Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMDictionaryRequest	Clears an `RsslRDMDictionaryRequest` structure. Useful for structure reuse.
rsslCopyRDMDictionaryRequest	Performs a deep copy of an `RsslRDMDictionaryRequest` structure.

Table 156: `RsslRDMDictionaryRequest` Utility Functions

7.5.2 RSSL RDM Dictionary Refresh

A ***Dictionary Refresh*** message is encoded and sent by OMM Provider applications. This message is used to send dictionary content in response to a request.

The `RsslRDMDictionaryRefresh` represents all members of a dictionary refresh message and allows for simplified use in OMM applications that leverage RDM. This structure follows the behavior and layout that is defined in the *Transport API C Edition RDM Usage Guide*.

7.5.2.1 RSSL RDM Dictionary Refresh Structure Members

STRUCTURE MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Contains general message information like `streamId` and `domainType`.
flags	**Required**. Indicates presence of optional login request members. For details, refer to Section 7.5.2.2.
state	**Required**. Indicates the state of the login stream. Defaults to a `streamState` of **RSSL_STREAM_OPEN** and a `dataState` of **RSSL_DATA_OK**. For more information on `RsslState`, refer to the *Transport API C Edition Developers Guide*.
dictionaryName	**Required**. Indicates the name of the dictionary being provided.
serviceId	**Required**. Indicates which service the dictionary is provided from.
verbosity	**Required**. Indicates the desired amount of information desired from the dictionary. The available enumerations are: • **RDM_DICTIONARY_INFO == 0x00**: Version information only • **RDM_DICTIONARY_MINIMAL == 0x03**: Provides information needed for caching • **RDM_DICTIONARY_NORMAL == 0x07**: Provides all information needed for decoding • **RDM_DICTIONARY_VERBOSE == 0x0F**: Provides all information (including comments) Providers are not required to support the **MINIMAL** and **VERBOSE** filters.
type	**Required**. Indicates the type of dictionary being provided. The types supported by the dictionary encoder and decoder are: • **RDM_DICTIONARY_FIELD_DEFINITIONS == 1** • **RDM_DICTIONARY_ENUM_TABLES == 2**
sequenceNumber	If present, `flags` value of **RDM_DC_RFF_HAS_SEQ_NUM** should be specified. A user-specified, item-level sequence number which can be used by the application for sequencing messages within this stream.

Table 157: `RsslRDMDictionaryRefresh` Structure Members

STRUCTURE MEMBER	DESCRIPTION
pDictionary	**Conditional (required when encoding)**. Points to an `RsslDataDictionary` object that contains the content to encode. For more information on the `RsslDataDictionary` structure, see the *Transport API C Edition RDM Usage Guide*. **Not used when decoding**.
startFid	This is used to maintain state when encoding a dictionary across multiple messages. To ensure that all dictionary content is correctly encoded, the application should not modify this.
dictionaryId	When decoding, this will be populated with the dictionary's ID. Presence is indicated by the **RDM_DC_RFF_HAS_INFO** flag. **Not used when encoding**. The Dictionary is retrieved from the `RsslDataDictionary` structure.
version	When decoding, this will be populated with the dictionary's version string. Presence is indicated by the **RDM_DC_RFF_HAS_INFO** flag. **Not used when encoding**. The Dictionary is retrieved from the `RsslDataDictionary` structure.
dataBody	When decoding, this will point to the encoded data buffer containing dictionary content. This buffer should be set on an `RsslDecodeIterator` and passed to the appropriate decode function according to the `type`. **Not used when encoding**. The Dictionary is retrieved from the `RsslDataDictionary` structure.

Table 157: `RsslRDMDictionaryRefresh` Structure Members (Continued)

7.5.2.2 RSSL RDM Dictionary Refresh Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
RDM_DC_RFF_HAS_INFO	When decoding, indicates presence of `dictionaryId`, `version`, and `type`. **Not used when encoding**. The encode function will add the information to the encoded message when appropriate.
RDM_DC_RFF_IS_COMPLETE	When decoding, if this flag is present, it indicates that this is the final fragment and that the consumer has received all content for this dictionary. **Not used when encoding**. The encode function will add the information to the encoded message when appropriate.
RDM_DC_RFF_SOLICITED	If this flag is present, it indicates that the directory refresh is solicited (e.g., it is in response to a request). If the flag is not present, this refresh is unsolicited.
RDM_DC_RFF_HAS_SEQ_NUM	Indicates presence of `sequenceNumber`.
RDM_DC_RFF_CLEAR_CACHE	Indicates that any stored payload information associated with the dictionary stream should be cleared. This may occur if some portion of data is known to be invalid.

Table 158: `RsslRDMDictionaryRefreshFlags` Flags

7.5.2.3 RSSL RDM Dictionary Refresh Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMDictionaryRefresh	Clears an `RsslRDMDictionaryRefresh` structure. Useful for structure reuse.

Table 159: `RsslRDMDictionaryRefresh` Utility Functions

FUNCTION NAME	DESCRIPTION
rsslCopyRDMDictionaryRefresh	Performs a deep copy of an `RsslRDMDictionaryRefresh` structure.

Table 159: `RsslRDMDictionaryRefresh` Utility Functions (Continued)

7.5.3 RSSL RDM Dictionary Status

OMM Provider and OMM non-interactive provider applications use the ***Dictionary Status*** message to convey state information associated with the dictionary stream. Such state information can indicate that a dictionary stream cannot be established or to inform a consumer of a state change associated with an open login stream. The Dictionary status message can also be used to indicate that a new dictionary should be retrieved. For more information on handling Dictionary versions, see the *Transport API C Edition RDM Usage Guide*.

The **`RsslRDMDictionaryStatus`** represents all members of a dictionary status message and allows for simplified use in OMM applications that leverage RDM. This structure follows the behavior and layout that is defined in the *Transport API C Edition RDM Usage Guide*.

7.5.3.1 RSSL RDM Dictionary Status Structure Members

STRUCTURE MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Contains general message information like streamId and domainType.
flags	**Required**. Indicate presence of optional login status members. For details, refer to Section 7.5.3.2.
state	If present, **`flags`** value of **RDM_DR_STF_HAS_STATE** should be specified. Indicates the state of the dictionary stream. For more information on **`RsslState`**, refer to the *Transport API C Edition Developers Guide*.

Table 160: `RsslRDMDictionaryStatusFlags` Structure Members

7.5.3.2 RSSL RDM Dictionary Status Flag Enumeration Values

FLAG ENUMERATION	DESCRIPTION
RDM_DC_STF_HAS_STATE	Indicates presence of **`state`**. If not present, any previously conveyed state should continue to apply.

Table 161: `RsslRDMDictionaryStatus` Flags

7.5.3.3 RSSL RDM Dictionary Status Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMLoginStatus	Clears an **`RsslRDMLoginStatus`** structure. Useful for structure reuse.
rsslCopyRDMLoginStatus	Performs a deep copy of an **`RsslRDMLoginStatus`** structure.

Table 162: `RsslRDMDictionaryStatus` Utility Functions

7.5.4 RSSL RDM Dictionary Close

A ***Dictionary Close*** message is encoded and sent by OMM consumer applications. This message allows a consumer to close an open dictionary stream. A provider can close the directory stream via a Dictionary Status message, refer to Section 7.5.3.

7.5.4.1 RSSL RDM Dictionary Close Structure Members

STRUCTURE MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Contains general message information like `streamId` and `domainType`.

Table 163: `RsslRDMDictionaryClose` Structure Members

7.5.4.2 RSSL RDM Dictionary Close Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMDictionaryClose	Clears an `RsslRDMDictionaryClose` structure. Useful for structure reuse.
rsslCopyRDMDictionaryClose	Performs a deep copy of an `RsslRDMDictionaryClose` structure.

Table 164: `RsslRDMDictionaryClose` Utility Functions

7.5.5 RSSL RDM Dictionary Message Union

This union can contain any of the RDM Dictionary message types. This is provided for use with Dictionary specific functionality.

7.5.5.1 RSSL RDM Dictionary Union

UNION MEMBERS	DESCRIPTION
rdmMsgBase	The message base information.
request	The `RsslRDMDictionaryRequest` as described in Section 7.5.1.
close	The `RsslRDMDictionaryClose` as described in Section 7.5.4.
refresh	The `RsslRDMDictionaryRefresh` as described in Section 7.5.2.
status	The `RsslRDMDictionaryStatus` as described in Section 7.5.3.

Table 165: `RsslRDMDictionaryMsg` Union Members

7.5.5.2 RSSL RDM Dictionary Message Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearRDMDictionaryMsg	Clears an `RsslRDMDictionaryMsg` union. Useful for reuse.
rsslCopyRDMDictionaryMsg	Performs a deep copy of an `RsslRDMDictionaryMsg` structure.

Table 166: `RsslRDMDictionaryMsg` Utility Functions

7.5.6 Dictionary Encoding and Decoding

7.5.6.1 RSSL RDM Dictionary Encoding and Decoding Functions

FUNCTION NAME	DESCRIPTION
rsslEncodeRDMDictionaryMsg	Used to encode an RDM Dictionary message. This function takes the `RsslRDMDictionaryMsg` as a parameter. Alternately, `rsslEncodeRDMMsg` can be used if encoding from an `RsslRDMMsg`.
rsslDecodeRDMDictionaryMsg	Used to decode an RDM Directory message. This function populates the `RsslRDMDictionaryMsg` and leverages the Value Added Utility message buffer (refer to Section 8.2). Alternately, `rsslDecodeRDMMsg` can be used to decode into an `RsslRDMMsg`.

Table 167: RDM Dictionary Encoding and Decoding Functions

7.5.6.2 Encoding a Dictionary Request

```
RsslEncodeIterator encodeIter;
RsslRDMDictionaryRequest dictionaryRequest;

/* Clear the Dictionary Request structure. */
rsslClearRDMDictionaryRequest(&dictionaryRequest);

/* Set flags. */
dictionaryRequest.flags = RDM_DC_RQF_STREAMING;

/* Set serviceId. */
dictionaryRequest.serviceId = 273;

/* Set verbosity. */
dictionaryRequest.verbosity = RDM_DICTIONARY_NORMAL;

/* Set dictionary name. */
dictionaryRequest.dictionaryName.data = "RWFFld";
dictionaryRequest.dictionaryName.length = 6;

/* Clear the encode iterator, set its RWF Version, and set it to a buffer for encoding into. */
rsslClearEncodeIterator(&encodeIter);
ret = rsslSetEncodeIteratorRWFVersion(&encodeIter, channelMajorVersion, channelMinorVersion);
ret = rsslSetEncodeIteratorBuffer(&encodeIter, &msgBuffer);

/* Encode the message. */
ret = rsslEncodeRDMMsg(&encodeIter, (RsslRDMMsg*)&dictionaryRequest, &msgBuffer.length,
        &rsslErrorInfo);
```

Code Example 28: Dictionary Request Encoding Example

7.5.6.3 Decoding a Dictionary Request

```
/* The decoder may require additional space to store things such as lists. */
char memoryArray[1024];
RsslBuffer memoryBuffer = { 1024, memoryArray };

RsslDecodeIterator decodeIter;
RsslMsg msg;
RsslRDMMsg rdmMsg;
RsslRDMDictionaryRequest *pDictionaryRequest;

/* Clear the decode iterator, set its RWF Version, and set it to the encoded buffer. */
rsslClearDecodeIterator(&decodeIter);
ret = rsslSetDecodeIteratorRWFVersion(&decodeIter, channelMajorVersion, channelMinorVersion);
ret = rsslSetDecodeIteratorBuffer(&decodeIter, &msgBuffer);

/* Decode the message to an RsslMsg structure and RsslRDMMsg structure. */
ret = rsslDecodeRDMMsg(&decodeIter, &msg, &rdmMsg, &memoryBuffer, &rsslErrorInfo);

if (ret == RSSL_RET_SUCCESS && rdmMsg.rdmMsgBase.domainType == RSSL_DMT_DICTIONARY &&
        rdmMsg.rdmMsgBase.rdmMsgType == RDM_DC_MT_REQUEST)
{
    /* The message we decoded is an RsslRDMDictionaryRequest. */
    pDictionaryRequest = &rdmMsg.dictionaryMsg.request;

    /* Print if streaming. */
    if (pDictionaryRequest->flags & RDM_DC_RQF_STREAMING)
        printf("Request is streaming.\n");

    /* Print serviceId. */
    printf("Service ID: %u\n", pDictionaryRequest->serviceId);

    /* Print verbosity. */
    printf("Verbosity: %u\n", pDictionaryRequest->verbosity);

    /* Print dictionary name. */
    printf("Dictionary Name: %.*s\n", pDictionaryRequest->dictionaryName.length, pDictionaryRequest->
            dictionaryName.data);
}
```

Code Example 29: Dictionary Request Decoding Example

7.5.6.4 Encoding a Dictionary Refresh

```
RsslEncodeIterator encodeIter;
RsslRDMDictionaryRefresh dictionaryRefresh;
RsslDataDictionary dataDictionary;
```

```
rsslClearDataDictionary(&dataDictionary);
ret = rsslLoadFieldDictionary("RDMFieldDictionary", &dataDictionary, &errorText);

/* Clear the Dictionary Refresh structure. */
rsslClearRDMDictionaryRefresh(&dictionaryRefresh);

/* Set flags. */
dictionaryRefresh.flags = RDM_DC_RFF_SOLICITED;

/* Set dictionary name. */
dictionaryRefresh.dictionaryName.data = "RWFFld";
dictionaryRefresh.dictionaryName.length = 6;

/* Set type. */
dictionaryRefresh.type = RDM_DICTIONARY_FIELD_DEFINITIONS;

/* Set the dictionary. */
dictionaryRefresh.pDictionary = &dataDictionary;

/* Set serviceId. */
dictionaryRefresh.serviceId = 273;

/* Set verbosity. */
dictionaryRefresh.verbosity = RDM_DICTIONARY_NORMAL;

do
{
    /* (Represents the application getting a new buffer to encode the message into.) */
    getNextEncodeBuffer(&msgBuffer);

    /* Clear the encode iterator, set its RWF Version, and set it to a buffer for encoding into. */
    rsslClearEncodeIterator(&encodeIter);
    ret = rsslSetEncodeIteratorRWFVersion(&encodeIter, channelMajorVersion, channelMinorVersion);
    ret = rsslSetEncodeIteratorBuffer(&encodeIter, &msgBuffer);

    /* Encode the message. This will return RSSL_RET_DICT_PART_ENCODED if it only a part
    * was encoded. We must keep encoding the message until RSSL_RET_SUCCESS is returned. */
    ret = rsslEncodeRDMMsg(&encodeIter, (RsslRDMMsg*)&dictionaryRefresh, &msgBuffer.length,
            &rsslErrorInfo);
} while (ret == RSSL_RET_DICT_PART_ENCODED);
```

Code Example 30: Dictionary Refresh Encoding Example

7.5.6.5 Decoding a Dictionary Refresh

```
/* The decoder may require additional space to store things such as lists. */
char memoryArray[4096];
RsslBuffer memoryBuffer = { 4096, memoryArray };
```

```
RsslDecodeIterator decodeIter;
RsslMsg msg;
RsslRDMMsg rdmMsg;
RsslRDMDictionaryRefresh *pDictionaryRefresh;
RsslInt32 dictionaryTypeForThisStreamId = 0;

RsslDataDictionary dataDictionary;

rsslClearDataDictionary(&dataDictionary);

do
{
    /* (Represents the application getting the next buffer to decode.) */
    getNextDecodeBuffer(&msgBuffer);

    /* Reset our memory buffer. */
    memoryBuffer.length = 4096;
    memoryBuffer.data = memoryArray;

    /* Clear the decode iterator, set its RWF Version, and set it to the encoded buffer. */
    rsslClearDecodeIterator(&decodeIter);
    ret = rsslSetDecodeIteratorRWFVersion(&decodeIter, channelMajorVersion, channelMinorVersion);
    ret = rsslSetDecodeIteratorBuffer(&decodeIter, &msgBuffer);

    /* Decode the message to an RsslMsg structure and RsslRDMMsg structure. */
    ret = rsslDecodeRDMMsg(&decodeIter, &msg, &rdmMsg, &memoryBuffer, &rsslErrorInfo);

    if (ret == RSSL_RET_SUCCESS && rdmMsg.rdmMsgBase.domainType == RSSL_DMT_DICTIONARY &&
            rdmMsg.rdmMsgBase.rdmMsgType == RDM_DC_MT_REFRESH)
    {
        /* The message we decoded is an RsslRDMDictionaryRefresh. */
        pDictionaryRefresh = &rdmMsg.dictionaryMsg.refresh;

        /* Print if request is streaming. */
        if (pDictionaryRefresh->flags & RDM_DC_RFF_SOLICITED)
            printf("Refresh is solicited.\n");

        /* Print info if present. If the dictionary is split into parts, this is normally only present
        * on the first part. */
        if (pDictionaryRefresh->flags & RDM_DC_RFF_HAS_INFO)
        {
            /* Remember the dictionary type for this stream since subsequent parts will not indicate it.
                    */
            dictionaryTypeForThisStreamId = pDictionaryRefresh->type;

            /* Print version. */
            printf("Version: %.*s\n", pDictionaryRefresh->version.length, pDictionaryRefresh->
                    version.data);
            /* Print dictionary ID. */
            printf("Dictionary ID: %lld\n", pDictionaryRefresh->dictionaryId);
        }

        /* Print serviceId. */
        printf("Service ID: %u\n", pDictionaryRefresh->serviceId);

        /* Print verbosity. */
        printf("Verbosity: %u\n", pDictionaryRefresh->verbosity);

        /* Print dictionary name. */
        printf("Dictionary Name: %.*s\n", pDictionaryRefresh->dictionaryName.length,
            pDictionaryRefresh->dictionaryName.data);

        if (dictionaryTypeForThisStreamId == RDM_DICTIONARY_FIELD_DEFINITIONS)
        {
            /* Decode the dictionary content into the RsslDataDictionary structure. */
            rsslClearDecodeIterator(&decodeIter);
            ret = rsslSetDecodeIteratorRWFVersion(&decodeIter, channelMajorVersion,
                    channelMinorVersion);
            ret = rsslSetDecodeIteratorBuffer(&decodeIter, &pDictionaryRefresh->dataBody);
            ret = rsslDecodeFieldDictionary(&decodeIter, &dataDictionary, RDM_DICTIONARY_NORMAL,
                    &errorText);
        }
    }
} while(!(pDictionaryRefresh->flags & RDM_DC_RFF_IS_COMPLETE));
```

Code Example 31: Dictionary Refresh Decoding Example

7.6 RDM Queue Messages

The Queue Messaging domain model is a series of message constructs that you use to interact with a Queue Provider. A Queue Provider can persist content for which users want to have guaranteed delivery and can also help send content to destinations with which users cannot directly communicate.

7.6.1 Queue Data Message Persistence

When using a queue messaging stream with a queue provider, use of a persistence file can guarantee delivery of messages sent by the OMM consumer on that queue stream. The queue file will be named after the name of the queue stream (as specified in the `RsslRDMQueueRequest` message that opened the stream). When the consumer submits `RsslRDMQueueData` messages, the consumer stores these messages in the persistence file in case the tunnel stream to the queue provider is lost and reconnected. As `RsslRDMQueueAck` messages are received from the queue provider, space in the persistence file is freed for additional messages. If at any time the application submits an `RsslRDMQueueData` message but the persistence file has no room for it, the application receives the **RSSL_RET_PERSISTENCE_FULL** return code.

The `RsslClassOfService.guarantee.persistLocally` option (set when opening the tunnel stream) specifies whether to create and maintain persistence files. The location for storage of persistent files is specified by the `RsslClassOfService.guarantee.persistenceFilePath` option. For more information on these options, refer to Section 6.7.3.

Note: Thomson Reuters recommends that the `RsslClassOfService.guarantee.persistenceFilePath` be set to a local storage device.

If a particular queue stream is no longer needed, the user may delete the persistence file with that queue stream's name.

⚠ **Warning!** If you delete a persistence file that stores messages that were not successfully transmitted, the messages will be lost.

7.6.2 Queue Request

The OMM application encodes and sends a ***Queue Request*** message to a Queue Provider to open a user queue. By opening a queue with an `RsslRDMQueueRequest`, the user receives any content previously sent to and persisted on a Queue Provider. To send content to other users' queues, a user must first open their own queue.

MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Sets the message type (i.e., **RDM_QMSG_MT_REQUEST**) and contains general message information, including the stream's ID (**streamId**) and domain type (**domainType**).
sourceName	**Required**. Specifies the name of the queue you want to open.

Table 168: `RsslRDMQueueRequest` Members

7.6.3 Queue Refresh

A Queue Provider encodes and sends a ***Queue Refresh*** message to OMM applications to inform users about queue open requests and give state information pertaining to specific queue refresh attempts.

MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Sets the message type (i.e., **RDM_QMSG_MT_REFRESH**) and contains general message information, including the stream's ID (**streamId**) and domain type (**domainType**).
sourceName	**Required**. Specifies the name of a queue to open, which should match the `sourceName` specified in the initial queue request.
state	**Required**. Indicates the state of the queue. • States of **Open** and **Ok** indicate the queue was successfully opened. • Other state combinations indicate an issue, for which additional code and text provide supplemental information. For more information on `RsslState`, refer to the *Transport API C Edition Developers Guide*.
queueDepth	**Required**. Indicates how many Queue Data or Queue Data Expired messages are inbound on this Queue Stream.

Table 169: `RsslRDMQueueRefresh` Members

7.6.4 Queue Status

A Queue Provider encodes and sends a ***Queue Status*** message to an OMM application, conveying state information about a user's queue.

MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Sets the message type (i.e., **RDM_QMSG_MT_STATUS**) and contains general message information, including the stream's ID (**streamId**) and domain type (**domainType**).
flags	**Required**. Indicates the presence of optional queue status members. `flags` has only one enumeration: **HAS_STATE**, which indicates the presence of the `state` member. If `flags` is absent (or has no value), any previously conveyed state continues to apply.
state	Indicates the state of the queue: • States of **Open** and **Ok** indicate the queue is in a good state. • Other state combinations indicate an issue, for which additional code and text provide supplemental information. For more information on `RsslState`, refer to the *Transport API C Edition Developers Guide*.

Table 170: `RsslRDMQueueStatus` Members

7.6.5 Queue Close

An OMM application encodes and sends a ***Queue Close*** message to a Queue Provider, closing the user's queue.

MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Sets the message type (i.e., **RDM_QMSG_MT_CLOSE**) and contains general message information, including the stream's ID (**streamId**) and domain type (**domainType**).

Table 171: `RsslRDMQueueClose` Members

7.6.6 Queue Data

Both OMM applications and queue providers can send and receive ***Queue Data*** messages, which exchange data content between queue users and also communicates whether content was undeliverable.

7.6.6.1 Queue Data Members

MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Sets the message type (i.e., **RDM_QMSG_MT_DATA**) and contains general message information, including the stream ID (**streamId**) and domain type (**domainType**).
sourceName	**Required**. Specifies the name of the queue from which content is sourced, which should match the `sourceName` specified in the Queue Request for this substream.
destName	**Required**. Specifies the name of the queue to which content is sent.
identifier	**Required**. A user-specified unique identifier for the message. `identifier` is used when acknowledging this content via a Queue Ack message.
timeout	Specifies the desired timeout for this content (which can be any of the **RsslRDMQueueTimeCodes** in Section 7.6.6.3 or a specific time interval in milliseconds). If a timeout value expires during the course of delivery, the content is returned as an **RsslRDMQueueDataExpired** message. If not specified, this defaults to **QueueMsgTimeoutCodes.INFINITE** (content never times out).
containerType	**Required**. Indicates the type of contents in this queue message.

Table 172: `RsslRDMQueueData` Members

MEMBER	DESCRIPTION
encDataBody	• If sending a message, populate **`encDataBody`** with pre-encoded content. If sending a message without pre-encoded contents, you can use the encoding methods described in Section 7.6.6.4. • If receiving a message, **`encDataBody`** can be used to access payload contents for decoding.
flags	**Required**. Specifies any flags that indicate more information about this message. For details on available flags, refer to Section 7.6.6.2. **`flags`** is only for decoding, and OMM Consumer applications do not need to set it.
queueDepth	**Required**. Indicates the number of Queue Data or Queue Data Expired messages still inbound on this queue stream, following this message. **`queueDepth`** is only for reading, and OMM Consumer applications do not need to set it.

Table 172: `RsslRDMQueueData` Members (Continued)

7.6.6.2 Queue Data Flags

`RsslRDMQueueData` messages and **`RsslRDMQueueDataExpired`** messages use the following flag:

FLAG	DESCRIPTION
RDM_QMSG_DF_POSSIBLE_DUPLICATE== 0x1	Indicates that the message was retransmitted, and the application might have already received it.

Table 173: Queue Data Flags

7.6.6.3 Queue Message Timeout Codes

Queue message timeout codes are special codes that can be set on the **`RsslRDMQueueData.timeout`** member to specify timeout behavior.

ENUMERATION	DESCRIPTION
RDM_QMSG_TC_INFINITE	This message persists in the system for an infinite amount of time.
RDM_QMSG_TC_IMMEDIATE	This message immediately times out if any portion of its delivery path is unavailable.
RDM_QMSG_TC_PROVIDER_DEFAULT	This message persists in the system for a duration set by the provider.

Table 174: `RsslRDMQueueTimeoutCodes`

7.6.6.4 Queue Data Encoding

The **`RsslRDMQueueData`** message allows users to encode both OMM and non-OMM/opaque content. There are several methods available to help with encoding.

FUNCTION NAME	DESCRIPTION
rsslEncodeRDMQueueMsgInit	Begins the process for content to be encoded into this **`RsslRDMQueueData`** message. This method takes an **`EncodeIterator`** as a parameter, where the **`EncodeIterator`** is associated with the buffer content will be encoded into. After this method returns, users should call additional methods required to encode the content. Once remaining encoding is completed, the **`encodeComplete`** method should be called.
rsslEncodeRDMQueueMsgComplete	Completes the content encoding into this **`RsslRDMQueueData`** message.
rsslEncodeRDMQueueMsg	When sending no payload or payload content is preencoded and specified on the **`RsslRDMQueueData.encDataBody`** buffer, this method will encode the **`RsslRDMQueueData`** message in a single call.

Table 175: Queue Data Message Encoding Methods

7.6.6.5 Queue Data Message Encoding Code Sample

```
RsslEncodeIterator _msgEncIter;
RsslRDMQueueData _queueData;

// initialize the QueueData encoding
rsslClearRDMQueueData(&_queueData);
_queueData.rdmMsgBase.streamId = QUEUE_MSG_STREAM_ID;
_queueData.identifier= 124;
_queueData.sourceName.data = “MY_QUEUE”;
_queueData.sourceName.length = 8;
_queueData.destName.data = “DESTINATION_QUEUE”;
_queueData.destName.length = 17;
_queueData.timeout = RDM_QMSG_TC_INFINITE;
_queueData.containerType = RSSL_DT_FIELD_LIST;

_msgEncIter.clear();
rsslClearEncodeIterator(&_msgEncIter);
rsslSetEncodeIteratorRWFVersion(&_msgEncIter, pTunnelStream-
        >classOfService.common.protocolMajorVersion, pTunnelStream-
        >classOfService.common.protocolMinorVersion);
rsslSetEncodeIteratorBuffer(&_msgEncIter, buffer);

// begin encoding content into RsslRDMQueueData message
if ((ret = rsslEncodeRDMQueueMsgInit(&_msgEncIter, &_queueData, &error)) < RSSL_RET_SUCCESS)
{
    printf("rsslEncodeRDMQueueMsgInit() failed”);
    return;
}

// Start Content Encoding – follow standard field list encoding as shown in the
// Transport API C Edition Developers Guide examples.
// When content encoding is done, complete the RsslRDMQueueData encoding
```

```
if ((ret = rsslEncodeRDMQueueMsgComplete(&_msgEncIter, RSSL_TRUE, &buffer->length &error)) <
        RSSL_RET_SUCCESS)
{
    printf("rsslEncodeRDMQueueMsgComplete() failed");
    return;
}
```

Code Example 32: Queue Data Message Encoding Example

7.6.7 QueueDataExpired

If queue data messages sent on a queue stream cannot be successfully delivered, the queue provider sends **`RsslRDMQueueDataExpired`** messages on the queue stream to OMM consumer applications.

OMM consumer applications do not send this message.

7.6.7.1 RsslRDMQueueDataExpired Structure Members

`RsslRDMQueueDataExpired` uses the following members.

MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Specifies the queue message type (i.e., **RDM_QMSG_MT_DATA_EXPIRED**) and contains general message information, including the stream's ID (**streamId**) and domain type (**domainType**).
flags	**Required**. **`flags`** indicate more information about this message. For details, refer to Section 7.6.6.2.
sourceName	**Required**. **`sourceName`** specifies the name of the queue to which content was sent (i.e., the value of **`destName`** as set in the original **`RsslRDMQueueData`** message).
destName	**Required**. **`destName`** specifies the name of the queue from which content is sourced (i.e., the value of **`sourceName`** as set in the original **`RsslRDMQueueData`** message).
identifier	**Required**. A user-specified, unique identifier for the message (which is the same as the **`identifier`** from the original **`RsslRDMQueueData`** message).
undeliverableCode	**Required**. Specifies a code explaining why the content was undeliverable. For more information on undeliverable codes and their meanings, refer to Section 7.6.7.2.
containerType	**Required**. Indicates the type of contents in the message.
encDataBody	Optional. Contains the payload contents (if any) of the original Queue Data message.
queueDepth	**Required**. Indicates how many Queue Data or Queue Data Expired messages are still inbound on this queue stream (following this message).

Table 176: RsslRDMQueueDataExpired Structure Members

7.6.7.2 Queue Message Undeliverable Codes

Undeliverable codes are used in the **`QueueDataExpired.undeliverable`** member, and specify why the message could not be delivered.

ENUMERATION	REASON FOR DELIVERY FAILURE
RDM_QMSG_UC_EXPIRED	The timeout value specified for this message has expired.
RDM_QMSG_UC_NO_PERMISSION	The source/sender of this message is not permitted to send or is not permitted to send to the specified destination.
RDM_QMSG_UC_INVALID_TARGET	The specified destination of this message does not exist.
RDM_QMSG_UC_QUEUE_FULL	The specified destination of this message has a full queue and cannot receive any additional content.
RDM_QMSG_UC_QUEUE_DISABLED	The specified destination of this message has a disabled queue.
RDM_QMSG_UC_UNSPECIFIED	Delivery failed for unspecified reasons.
RDM_QMSG_UC_MAX_MSG_SIZE	The message was too large.
RDM_QMSG_UC_INVALID_SENDER	The sender of this message has now become invalid.
RDM_QMSG_UC_TARGET_DELETED	The target queue was deleted before the message was delivered.

Table 177: `RsslRDMQueueDataUndeliverableCodes`

7.6.8 Queue Ack

A Queue Provider encodes and sends a ***Queue Ack*** message to OMM applications, acknowledging that a Queue Data message is persisted on the Queue Provider. Once acknowledged by a Queue Provider, the OMM application no longer needs to persist the acknowledged content.

MEMBER	DESCRIPTION
rdmMsgBase	**Required**. Sets the message type (i.e., **RDM_QMSG_MT_ACK**) and contains general message information, including the stream's ID (**streamId**) and domain type (**domainType**).
identifier	**Required**. The identifier of the message being acknowledged. This should match the `RsslRDMQueueData.identifier` for the message being acknowledged.
sourceName	**Required**. Specifies the name of the queue to which content was originally sent (i.e., the value of `destName` as set in the original Queue Data message).
destName	Specifies the name of the queue from which content is sourced (i.e., the value of `sourceName` as set in the original Queue Data message).

Table 178: `RsslRDMQueueAck` Members

Chapter 8 Value Added Utilities

8.1 Utility Overview

The Value Added Utilities are a collection of helper constructs, mainly used by the Transport API Reactor. Included is a multi-purpose memory buffer type that can help with flexible, reusable memory - this is leveraged by the Administration Domain Model Representations when encoding or decoding messages. Other Value Added Utilities include a simple queue, mutex locks, thread helper functionality, and a simple event alerting component.

Only the Memory Buffer utility is described in this document as it is leveraged by the provided example applications. The other Value Added Utilities are internally leveraged by the Transport API Reactor so applications need not be familiar with their use.

8.2 Memory Buffer

The Memory Buffer utilities provide a simple method to apportion space from a block of memory space. This allows for the creation of complex objects without expensively requesting and releasing memory from the operating system. This also allows for easy reuse of the memory block. The memory is provided to the functions via an **`RsslBuffer`** that has its **`data`** member set to the memory block and **`length`** member indicating the byte length of the memory.

FUNCTION NAME	DESCRIPTION
rsslReserveBufferMemory	Reserves memory from an **`RsslBuffer`**. The buffer passed in is modified to point to the unused portion of the memory block. Subsequent calls reserve adjacent memory, so this can be called multiple times to generate a C-style array of objects without knowing the full length in advance.
rsslReserveAlignedBufferMemory	Reserves memory from an RsslBuffer. Similar to **`rsslReserveBufferMemory`**, but will ensure that the memory is aligned on an appropriate word boundary. Subsequent calls to the non-aligned **`rsslReserveBufferMemory`** will reserve adjacent memory, so this can be called multiple times to generate a C-style array of objects without knowing the full length in advance.
rsslCopyBufferMemory	Requires an input **`RsslBuffer`**, output **`RsslBuffer`**, and an **`RsslBuffer`** pointing to an available memory block. Sets the output buffer to a deep copy of the input **`RsslBuffer`**, using the space provided by the memory block.

Table 179: Memory Buffer Functions

8.3 Using the Memory Buffer

The following example reserves an aligned block of memory to represent an array of five user-defined **`MyStruct`** structures. The memory for the first structure is reserved and aligned. Each subsequent member of the array is then reserved in a loop, wherein memory is reserved based on the initial aligned offset. The memory associated with each **`MyStruct`** is initialized after it is reserved. Once completed, **`myStructArray`** can be accessed like an array of **`MyStruct`**s (**`myStructArray[index]`**, etc.).

```
/* Represents some complex user-defined struct.
 * This example will create an array of these structs. */
typedef struct
{
    int number;
    char letter;
} MyStruct;

int i = 0;

/* The block of memory that we will use as storage of the array. */
char memoryBlock[128];
RsslBuffer memoryBuffer;

MyStruct *myStructArray, *myStructElem;

memoryBuffer.data = memoryBlock;
memoryBuffer.length = 128;

/* Create first element on a word boundary, then initialize */
myStructArray = (MyStruct*)rsslReserveAlignedBufferMemory(&memoryBuffer, 1, sizeof(MyStruct));
myStructArray->number = i;
myStructArray->letter = ‘a’;

for(i = 1; i < 5; ++i)
{
    /* Reserve space for subsequent elements and initialize them in. */
    myStructElem = (MyStruct*)rsslReserveBufferMemory(&memoryBuffer, 1, sizeof(MyStruct));

    myStructElem->number = i;
    myStructElem->letter = 'a';
}

/* Change the letter of MyStruct in position 2, can access just like an array */
myStructArray[2]->letter = ‘b’;
```

Code Example 33: Memory Buffer Example

Chapter 9 Payload Cache Detailed View

9.1 Concepts

The Value Added Payload Cache component provides a facility for storing OMM containers (the data payload of OMM messages). Typical use of a payload cache is to store the current image of OMM data streams, where each entry in the cache corresponds to a single data stream. The initial content of a cache entry is defined by the payload of a refresh message. The current (or last) value of the entry is defined by the cumulative application of all refresh and update messages applied to the cache entry container. Values are stored in and retrieved from the cache as encoded OMM containers.

A cache is defined as a collection of OMM data containers. An application may create multiple cache collections, or instances, depending on how it wants to organize the data. The only restriction on cache organization is that all entries in a cache must use the same RDM Field Dictionary to define the set of field definitions it will use. At minimum, a separate cache would be required for each field dictionary in use by the application. However, since cache instances can also share the same field dictionary, partitioning is not restricted to dictionary usage. Some examples of how cache instances can be organized in an application include: all item streams on an RSSL connection; all items belonging to a particular service; all items across the entire application.

The application is responsible for organizing cache instances, managing the lifecycle of all entries in each cache, and applying and retrieving data from the cache. Figure 1 shows an example consumer type application which has created two cache instances to store data from two services on an OMM provider.

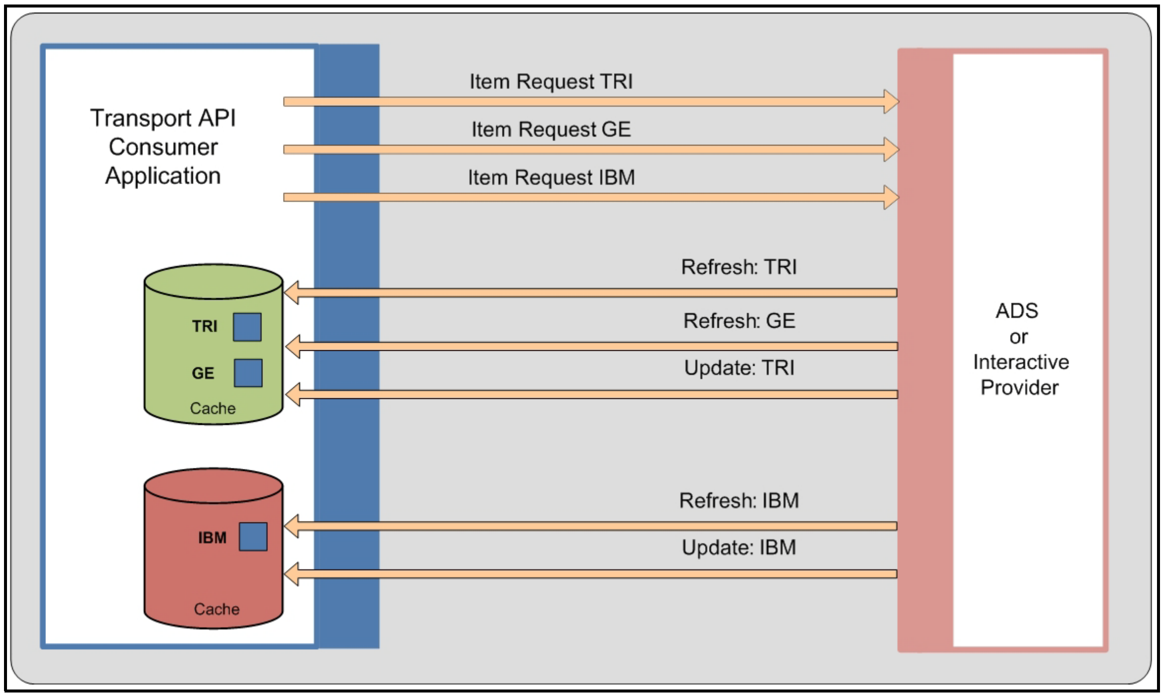


Figure 10. Consumer Application using Cache to Store Payload Data for Item Streams

9.2 Payload Cache

This section describes how the payload cache is managed (initialization and uninitialization), and how instances of cache (collections of payload entries) are created and destroyed.

9.2.1 Payload Cache Management

To use the Value Added Payload Cache, the application must first call the `rsslPayloadCacheInitialize` function for global static resource initialization. When the payload cache is no longer needed, the application should call `rsslPayloadCacheUninitialize` to cleanup and release all resources used by the cache.

Use the following functions to manage the cache:

FUNCTION NAME	DESCRIPTION
rsslPayloadCacheInitialize	The first function the application must call prior to using the payload cache. The method only needs to be called one time by the application, but may be called more than once. A reference count is incremented for each call to this function. An equal number of calls to `rsslPayloadCacheUninitialize` must be made before the component is uninitialized.
rsslPayloadCacheUninitialize	The last call an application should make when it is finished using the payload cache interface. The initialization reference count is decremented for each call to this function. Uninitialization only occurs if the initialization reference count is zero. During uninitialization, all remaining cache instances, entries, and resources will be destroyed.
rsslPayloadCacheIsInitialized	This function can be used by an application to determine if the payload cache component has already been initialized (by a call to `rsslPayloadCacheInitialize`).

Table 180: Payload Cache Management Functions

9.2.2 Cache Error Handling

Some of the functions on the payload cache interface use the `RsslCacheError` structure to return error information. This structure will be populated with additional information if an error occurred during the function call. The application should check the return value from functions. The application can optionally provide the `RsslCacheError` structure to obtain additional information.

9.2.2.1 Cache Error Structure Members

The `RsslCacheError` has the following structure members:

STRUCTURE MEMBER	DESCRIPTION
rsslErrorId	Range of values is defined by the set of Transport API return codes (from the `RsslReturnCodes` enumeration).
text	This `char[]` will contain text with additional information when a function call returns a non-successful result. The size of the buffer is fixed to **MAX_OMM_CACHE_ERROR_TEXT** as defined on the cache interface.

Table 181: `RsslCacheError` Structure Members

9.2.2.2 Clearing a Cache Error

The following function handles the RsslCacheError.

STRUCTURE MEMBER	DESCRIPTION
rsslCacheErrorClear	Clears the `RsslCacheError` structure. Use this function prior to passing the structure to a cache interface function.

Table 182: Function for Cache Error Handling

9.2.3 Payload Cache Instances

A payload cache instance is a collection of payload data containers. An empty cache instance must be created before any data can be stored in the cache. When a cache or its entries are no longer needed, it can be destroyed. For functions used to create and destroy a cache, refer to Section 9.2.1.

9.2.3.1 Payload Cache Structure Member

STRUCTURE MEMBER	DESCRIPTION
maxItems	Sets the maximum number of entries allowed in the cache. There is no limit on the number of items when the value is zero. When the maximum number of items is reached, no new entries can be created in the cache until entries are removed. The `rsslPayloadEntryCreate` function will return a null `RsslPayloadEntryHandle` when the maximum number of items is reached. Refer to Section 9.3.1.

Table 183: RsslPayloadCacheConfigOptions Structure Members

9.2.3.2 Managing the Payload Cache

FUNCTION NAME	DESCRIPTION
rsslPayloadCacheCreate	Creates a payload cache instance, and returns the `RsslPayloadCacheHandle`. All operations on the cache require this handle. Options are passed in via the `RsslPayloadCacheConfigOptions` defined in Section 9.2.3.1.
rsslPayloadCacheDestroy	Destroys a payload cache instance. Any entries remaining in the cache are also destroyed at this time.

Table 184: Functions for Managing Cache Instances

9.2.4 Managing RDM Field Dictionaries for Payload Cache

Each cache instance requires an RDM Field Dictionary, to define the set of fields that may be encoded in the OMM containers stored in the cache.

A cache is associated with a field dictionary through a binding process, which requires an `RsslDataDictionary` structure loaded with the field dictionary. The dictionary structure can be loaded from a file (using the `rsslLoadFieldDictionary` function), or from an encoded dictionary message from a provider (using the `rsslDecodeFieldDictionary` function). The cache does not use the enumerated dictionary content, so loading the enumeration dictionary is not required for cache use. See the Transport API Reference Manual for more information on using `RsslDataDictionary`.

Once the `RsslDataDictionary` is loaded, it is bound to a cache instance using a key (an arbitrary string identifier assigned by the application to name the dictionary). The key allows multiple cache instances to share the same dictionary. After the first binding of a dictionary, it can be bound to additional cache instances by simply providing the same key on additional bindings. For a list of functions used to bind a dictionary to a cache, refer to Section 9.2.4.1.

The cache builds its own field definition database from the `RsslDataDictionary` definitions. After binding, the application does not need to retain the dictionary structure, since the cache does not refer to the `RsslDataDictionary` used during the binding. In typical usage, the application will likely retain the dictionary for use with other encoding and decoding operations.

Note: A cache can only be bound to a dictionary once during its lifetime. While a cache cannot be switched to a new dictionary, the dictionary in use may be extended with new definitions. Refer to Section 9.2.4.3.

9.2.4.1 Setting Functions

FUNCTION NAME	DESCRIPTION
rsslPayloadCacheBindDictionary	**Deprecated**. For details on using this function, refer to the *Transport API C Edition Developers Guide*. Otherwise, for equivalent functionality, refer to **rsslPayloadCacheSetDictionary** in this table.
rsslPayloadCacheBindSharedDictionaryKey	**Deprecated**. For details on using this function, refer to the *Transport API C Edition Developers Guide*. Otherwise, for equivalent functionality, refer to **rsslPayloadCacheSetSharedDictionaryKey** in this table.
rsslPayloadCacheSetDictionary	This function sets an **`RsslDataDictionary`** to a cache instance (identified by **`RsslPayloadCacheHandle`**). Use this function the first time a dictionary is set to a cache. The application must provide a key parameter to this function to name the dictionary for future reference. This key will be used in future setting operations when the application wants to share a dictionary between cache instances, or to extend the definitions in the dictionary. The first time a particular key is used with this function will be the initial setting of that dictionary to a cache. The second time the same key is used in this function; it will reload the field definitions from the given **`RsslDataDictionary`** structure, enabling the dictionary to be extended. Refer to Section 9.2.4.3.
rsslPayloadCacheSetSharedDictionaryKey	Use this function when sharing a dictionary among multiple caches. This function will set a cache (identified by the **`RsslPayloadCacheHandle`**) to a previously set dictionary (identified by the dictionary key name). In order to share a dictionary, the dictionary named by the key passed to this function must have previously had an initial setting to another cache using the **`rsslPayloadCacheSetDictionary`** function. This function does not require the **`RsslDataDictionary`** structure, since that was already loaded during the initial setting with this dictionary key.

Table 185: Functions for Setting Dictionary to Cache

9.2.4.2 Setting Example

In the following example, two cache instances are created and set to a single shared field dictionary.

```
RsslRet ret;
RsslCacheError cacheError;
RsslPayloadCacheConfigOptions cacheConfig;
const char* dictionaryKey = “SharedKey1”;
char errorDataArray[256];
RsslBuffer errorBuffer = {256, &errorDataArray[0]};
RsslPayloadCacheHandle cacheHandle1 = 0;
RsslPayloadCacheHandle cacheHandle2 = 0;
RsslDataDictionary dataDictionary;

/* For simplicity in this code fragment, CHK is assumed to be a macro for error handling
        (performing cleanup and returning from function). */

/* Initialize cache component and create cache instances */
ret = rsslPayloadCacheInitialize(); CHK(ret)
```

```
cacheConfig.maxItems = 0; /* unlimited */
cacheHandle1 = rsslPayloadCacheCreate(&cacheConfig, &cacheError);
if (cacheHandle1 == 0)
{
    printf("rsslPayloadCacheCreate failure: %s\n", cacheError.text);
    CHK(cacheError.rsslErrorId)
}
cacheHandle2 = rsslPayloadCacheCreate(&cacheConfig, &cacheError);
if (cacheHandle2 == 0)
{
    printf("rsslPayloadCacheCreate failure: %s\n", cacheError.text);
    CHK(cacheError.rsslErrorId)
}

/* Load an RDM Field Dictionary structure from file: set to each cache. */
rsslClearDataDictionary(&dataDictionary);
ret = rsslLoadFieldDictionary("RDMFieldDictionary", &dataDictionary, &errorBuffer); CHK(ret)
/* Initial setting of the dictionary to the first cache */
ret = rsslPayloadCacheSetDictionary(cacheHandle1, &dataDictionary, dictionaryKey, &cacheError);
        CHK(ret)
/* Shared setting of the same dictionary to the second cache */
ret = rsslPayloadCacheSetSharedDictionaryKey(cacheHandle2, dictionaryKey, &cacheError); CHK(ret)
/* The dataDictionary can be destroyed after setting, but is typically retained by the application
        for encoding and decoding. */

/* Two cache instances are now ready for applying and retrieving data */
/* ... */

/* Cleanup */
rsslPayloadCacheDestroy(cacheHandle1); /* destroys all entries and the cache instance */
rsslPayloadCacheDestroy(cacheHandle2);
/* After all cache instances bound to a dictionary are destroyed, the cache API will clean up the
        internal field dictionary database used by the cache. */
rsslPayloadCacheUninitialize(); /* final call to cache interface */
rsslDeleteDataDictionary(&dataDictionary);
```

Code Example 34: Creating Cache and Setting to Dictionary

9.2.4.3 Extending the Cache Field Dictionary

While a cache can only be set to a single dictionary during it's lifetime, the set of field definitions defined by the dictionary can be extended. This is accomplished by reloading the cache field definition database with another call to the **`rsslPayloadCacheSetDictionary`** function. When extending the field dictionary, the **`RsslDataDictionary`** must contain the original field definitions, plus any new definitions the application wishes to use. Changes or deletions to the original field definitions are not supported; only additions are allowed. Using the same **`RsslPayloadCacheHandle`** and dictionary key that were previously set, call the **`rsslPayloadCacheSetDictionary`** function again with extended dictionary structure.

Note: When extending a field dictionary that is shared, all caches sharing that same dictionary key will see the extension, with only a single call to **`rsslPayloadCacheSetDictionary`**. There is no need to set the shared dictionary key again to each cache after a dictionary is extended.

9.2.5 Payload Cache Utilities

The functions in the following table describe additional tools available to the application for managing each cache instance. These utilities provide a count of the cache entries, and a list of handles to each cache entry.

FUNCTION NAME	DESCRIPTION
rsslPayloadCacheGetEntryCount	Returns the number of item payload entries in this cache instance (`RsslPayloadCacheHandle`).
rsslPayloadCacheGetEntryList	Populates an array provided by the caller with entry handles (`RsslPayloadEntryHandle`) for this cache instance (`RsslPayloadCacheHandle`). An application would typically manage the set of entry handles, since each cache entry will likely be associated with an entry in the application's item list. This utility provides access to the entire entry handle list if needed.
rsslPayloadCacheClearAll	Destroys all entries in the cache instance (`RsslPayloadCacheHandle`). The empty cache can be reused and remains bound to it's data dictionary.

Table 186: Payload Cache Utility Functions

9.3 Payload Cache Entries

A payload cache entry stores a single OMM container (whose data types are defined by `RsslContainerType`). While any arbitrary OMM data can be stored in the cache entry, the primary use case is to maintain the last known value of an item data stream by applying the sequence of refresh and update messages in the stream to the cache entry. The initial data applied to a container must be a refresh message payload, which will define the container type to be stored (e.g. Map). As refresh and update messages from the item stream are applied to the cache entry, the cache decodes the OMM data and sets the current value by following the OMM rules for the container (e.g. adding, deleting or updating map entries in a Map, or updating fields in a Field List). The last value of the data stream can be retrieved from cache at any time as an encoded OMM container.

9.3.1 Managing Payload Cache Entries

Payload cache entries are created within a cache instance. A cache entry is defined by the `RsslPayloadEntryHandle` returned from the `rsslPayloadEntryCreate` function. Entries cannot be moved between different cache instances, due to their dependency on the field dictionary bound to the cache where they are created.

Cache entries only store the payload container of an item. Other information related to items (e.g. message key attributes, domain, state) would need to be maintained as needed in an item list managed by the application, which will identify the source or sink associated with the cache entry data. This item list will likely include the `RsslPayloadEntryHandle` if the payload of the item is cached.

For a list of basic utilities provided by the payload cache to manage the collection of entries in the cache, refer to Section 9.2.5.

Use the following functions for managing cache entries:

FUNCTION NAME	DESCRIPTION
rsslPayloadEntryCreate	Returns a `RsslPayloadEntryHandle` to the newly created entry in the cache defined by the given `RsslPayloadCacheHandle`. The `RsslPayloadEntryHandle` is required for all operations on this entry. This function will return a null handle if the entry could not be created (e.g. if the maximum number of entries as defined in `RsslPayloadCacheConfigOptions` would be exceeded).

Table 187: Payload Cache Entry Management Functions

FUNCTION NAME	DESCRIPTION
rsslPayloadEntryDestroy	Destroys the cache entry defined by **`RsslPayloadEntryHandle`** and removes it from it's cache.
rsslPayloadEntryClear	Any data contained in the cache entry **`RsslPayloadEntryHandle`** is deleted, and the entry is returned to its initial state. The entry itself remains in the cache and can be re-used.

Table 187: Payload Cache Entry Management Functions (Continued)

9.3.2 Applying Data

Data is applied to a cache entry from the payload of an OMM message by using the **`rsslPayloadEntryApply`** function. The decoded **`RsslMsg`** and an **`RsslDecodeIterator`** are passed to the apply function. The iterator (positioned at the start of the encoded payload data **`RsslMsgBase.encDataBody`**) will be used to decode the OMM data so that the cache entry data can be set or updated.

Some caching behaviors are controlled by flags in the **`RsslMsg`**. When an **`RsslRefreshMsg`** is applied to the cache entry, the following **`RsslRefreshFlags`** will be followed during the application:

- **RSSL_RFMF_CLEAR_CACHE**: the cache entry data will be cleared prior to applying this message.
- **RSSL_RFMF_DO_NOT_CACHE**: the payload will not be applied to the cache entry.

When an **`RsslUpdateMsg`** is applied to cache, the following **`RsslUpdateFlags`** will be followed:

- **RSSL_UPMF_DO_NOT_CACHE**: the payload data will not be applied to the cache entry.
- **RSSL_UPMF_DO_NOT_RIPPLE**: entry rippling will not be performed when this data is applied.

The following example demonstrates how to create a payload entry in a cache instance and apply the payload of an **`RsslMsg`** to the cache entry.

```
/* Apply buffer containing an encoded RsslMsg to cache entry */
RsslRet applyBufferToCache(RsslChannel *pChannel, RsslBuffer *pBuffer, RsslPayloadCacheHandle
        cacheHandle, RsslPayloadEntryHandle *pEntryHandle)
{
    RsslDecodeIterator dIter;
    RsslMsg msg;
    RsslRet ret;
    RsslCacheError cacheError;

    /* If the caller did not provide a cache entry handle, create a new entry */
    if (*pEntryHandle == 0)
    {
        rsslCacheErrorClear(&cacheError);
        *pEntryHandle = rsslPayloadEntryCreate(cacheHandle, &cacheError);
        if (*pEntryHandle == 0)
        {
            printf("Error (%d) creating cache entry: %s\n", cacheError.rsslErrorId, cacheError.text);
            return cacheError.rsslErrorId;
        }
    }

    /* Perform message decoding. */
    rsslClearDecodeIterator(&dIter);
    rsslSetDecodeIteratorRwfVersion(&dIter, pChannel->majorVersion, pChannel->minorVersion);
    rsslSetDecodeIteratorBuffer(&dIter, pBuffer);
    rsslClearMsg(&msg);
    ret = rsslDecodeMsg(&dIter, &msg);
    if (ret < RSSL_RET_SUCCESS)
    {
        printf("Failure (%d) decoding message from buffer\n");
        return ret;
    }

    /* Apply the decoded RsslMsg to cache, with iterator positioned at the start of the payload */
    rsslCacheErrorClear(&cacheError);
    ret = rsslPayloadEntryApply(*pEntryHandle, &dIter, &msg, &cacheError);
    if (ret < RSSL_RET_SUCCESS)
    {
        printf("Error (%d) applying data to cache entry: %s\n", cacheError.rsslErrorId,
                cacheError.text);
        return ret;
    }

    return ret;
}
```

Code Example 35: Applying Data to a Payload Cache Entry

9.3.3 Retrieving Data

Data is retrieved from a cache entry as an encoded OMM container by using the **`rsslPayloadEntryRetrieve`** function. The application provides the data buffer (via an **`RsslEncodeIterator`**) where the container will be encoded. The retrieve function supports both encoding scenarios. When using **`rsslEncodeMsg`**, the encoded content retrieved from the cache entry can be set on the **`RsslMsgBase.encDataBody`**. If using **`rsslEncodeMsgInit`** and **`rsslEncodeMsgComplete`** encoding, the cache retrieve function can be used to encode the message payload prior to **`rsslEncodeMsgComplete`**.

There are two options for using the **`rsslPayloadEntryRetrieve`** function. For single-part retrieval, the buffer provided by the application must be large enough to hold the entire encoded container. For multi-part retrieval, the application makes a series of calls to **`rsslPayloadEntryRetrieve`** to get the OMM container in fragments (e.g. a sequence of maps are retrieved which together contain the entire set of map entries for the OMM container). In this usage, the optional **`RsslPayloadCursorHandle`** is required to maintain the state of the multi-part retrieval. Container types **`FieldList`** and **`ElementList`** cannot be fragmented, so the buffer size must be large enough to retrieve the entire container.

The following functions describe data-related operations on a cache entry.

FUNCTION NAME	DESCRIPTION
rsslPayloadEntryGetDataType	Returns the **`RsslContainerType`** stored in the cache entry (**`RsslPayloadEntryHandle`**). When initially created (or after the entry is cleared), the data type will be **RSSL_DT_UNKNOWN**. The data type is defined by the container type of the first refresh message applied to the entry.

Table 188: Functions for Applying and Retrieving Cache Entry Data

FUNCTION NAME	DESCRIPTION
rsslPayloadEntryApply	Applies the OMM data in the payload of the `RsslMsg` to the cache entry (`RsslPayloadEntryHandle`). The first message applied must be a refresh message (class **RSSL_MC_REFRESH**).
rsslPayloadEntryRetrieve	Retrieves data from the cache entry by encoding the OMM container into the buffer provided with the `RsslEncodeIterator` given by the application. For single-part retrieval, the `RsslPayloadCursorHandle` parameter is optional. For details on multi-part retrieval, refer to Section 9.3.3.1.

Table 188: Functions for Applying and Retrieving Cache Entry Data (Continued)

9.3.3.1 Multi-Part Retrieval

For data types that support fragmentation, the container can be retrieved in multiple parts by calling `rsslPayloadEntryRetrieve` until the complete container is returned. To support multi-part retrieval, the optional `RsslPayloadCursorHandle` parameter is required when calling `rsslPayloadEntryRetrieve`. The cursor is used to maintain the position where the next retrieval will resume. The application must check the state of the cursor after each call to `rsslPayloadEntryRetrieve` to determine when the retrieval is complete. The following functions are needed when using the payload cursor.

FUNCTION NAME	DESCRIPTION
rsslPayloadCursorCreate	Creates a cursor for optional use in the `rsslPayloadEntryRetrieve` function (required for multi-part retrieval). Returns the `RsslPayloadCursorHandle`.
rsslPayloadCursorDestroy	Destroys the cursor referenced by the `RsslPayloadCursorHandle`.
rsslPayloadCursorClear	Clears the state of the cursor for the given `RsslPayloadCursorHandle`. Whenever retrieving data from a cache entry, the cursor must be cleared prior to the first call to `rsslPayloadEntryRetrieve`. Clearing the cursor also allows it to be reused with a retrieval on a different container.
rsslPayloadCursorIsComplete	Returns the completion state of a retrieval where the `RsslPayloadCursorHandle` was used. The state must be checked after each call to `rsslPayloadEntryRetrieve` to determine if there is any additional data to be encoded for the cache entry container. When the cursor state is complete, the entire container of the cache entry has been retrieved.

Table 189: Functions for Using the Payload Cursor

9.3.3.2 Buffer Management

In multi-part usage, the size of the buffer used in the calls to `rsslPayloadEntryRetrieve` will affect how many fragments are required to retrieve the entire image of the cache entry. The retrieve function will continue to encode OMM entries from the cache container until it runs out of room in the buffer to encode the next entry. To progress during a multi-part retrieval, the buffer size must be at least large enough to encode a single OMM entry from the payload container. For example, if retrieving a Map in multiple parts, the buffer must be large enough to encode at least one MapEntry on each retrieval.

There are three general outcomes when using the `rsslPayloadEntryRetrieve` function:

- Full cache container is encoded into the buffer. This can occur with or without the use of the optional `RsslPayloadCursorHandle`. If used in this scenario, the cursor state would indicate the retrieval is complete.
- Partial container encoded into the buffer. This is only possible when using the `RsslPayloadCursorHandle` for container types that support fragmentation. The application must check the cursor to test if this is the final part.

- No data encoded into container due to insufficient buffer size. This can occur with or without the use of the optional `RsslPayloadCursorHandle`. The application may retrieve again with a larger buffer.

9.3.3.3 Example: Cache Retrieval with Multi-Part Support

The following example illustrates data retrieval from a cache entry, which supports multi-part encoding of a container.

```
/* Code fragment showing use of rsslPayloadEntryRetrieve for multi-part retrieval. */

RsslRet ret;
RsslCacheError cacheError;
RsslBuffer buffer;
RsslEncodeIterator eIter;
int arraySize = DEFAULT_BUFFER_SIZE;
unsigned char* bufferArray = (char*) malloc(arraySize);
buffer.data = bufferArray;
buffer.length = arraySize;
RsslPayloadCursorHandle cursorHandle = rsslPayloadCursorCreate();
rsslPayloadCursorClear(cursorHandle);
while (!rsslPayloadCursorIsComplete(cursorHandle))
{
    buffer.length = arraySize;
    rsslClearEncodeIterator(&eIter);
    rsslSetEncodeIteratorBuffer(&eIter, &buffer);
    rsslCacheErrorClear(&cacheError);
    /* _entryHandle created outside the scope of this code fragment */
    ret = rsslPayloadEntryRetrieve(_entryHandle, &eIter, cursorHandle, &cacheError);
    if (ret == RSSL_RET_SUCCESS)
        /* Number of bytes encoded is buffer.length. Application can used encoded data, e.g. set the
                payload on RsslMsgBase.encDataBody and encode a message to be transmitted. */
    else if (ret == RSSL_RET_BUFFER_TOO_SMALL)
        /* Increase arraySize and reallocate bufferArray. */
    else
        /* Handle terminal error condition. See cacheError.text[] for additional information. */
}
rsslPayloadCursorDestroy(cursorHandle);
free(bufferArray);
```

Code Example 36: Cache Retrieval with Multi-Part Support

Document ID: ETAC302UMVAC.160
Date of issue: 30 April 2016