

Transport API C Edition

V3.1

DEVELOPERS GUIDE

C EDITION

© Thomson Reuters 2015 - 2017. All rights reserved.

Thomson Reuters, by publishing this document, does not guarantee that any information contained herein is and will remain accurate or that use of the information will ensure correct and faultless operation of the relevant service or equipment. Thomson Reuters, its agents and employees, shall not be held liable to or through any user for any loss or damage whatsoever resulting from reliance on the information contained herein.

This document contains information proprietary to Thomson Reuters and may not be reproduced, disclosed, or used in whole or part without the express written permission of Thomson Reuters.

Any Software, including but not limited to, the code, screen, structure, sequence, and organization thereof, and Documentation are protected by national copyright laws and international treaty provisions. This manual is subject to U.S. and other national export regulations.

Nothing in this document is intended, nor does it, alter the legal obligations, responsibilities or relationship between yourself and Thomson Reuters as set out in the contract existing between us.

Contents

Chapter 1	Transport API Developers Guide Introduction	1
1.1	About this Manual	1
1.2	Audience	1
1.3	Programming Language.....	1
1.4	Acronyms and Abbreviations	1
1.5	References	3
1.6	Documentation Feedback	3
1.7	Document Conventions	3
1.7.1	<i>Typographic</i>	3
1.7.2	<i>Diagrams</i>	4
1.8	What's New in this Document	4
Chapter 2	Product Description	5
2.1	What is the Transport API?	5
2.2	Transport API Features	6
2.2.1	<i>General Capabilities</i>	6
2.2.2	<i>Consumer Applications.....</i>	6
2.2.3	<i>Provider Applications: Interactive</i>	7
2.2.4	<i>Provider Applications: Non-Interactive.....</i>	7
2.3	Performance and Feature Comparison	7
2.4	Functionality: Which API to Choose?	8
2.4.1	<i>General Capability Comparison.....</i>	8
2.4.2	<i>Layer-Specific Capability Comparison</i>	9
Chapter 3	Consumers and Providers	11
3.1	Overview	11
3.2	Consumers	12
3.2.1	<i>Subscriptions: Request/Response.....</i>	13
3.2.2	<i>Batches.....</i>	13
3.2.3	<i>Views</i>	14
3.2.4	<i>Pause and Resume</i>	15
3.2.5	<i>Symbol Lists</i>	16
3.2.6	<i>Posting.....</i>	19
3.2.7	<i>Generic Message.....</i>	20
3.2.8	<i>Private Streams</i>	20
3.3	Providers	22
3.3.1	<i>Interactive Providers</i>	23
3.3.2	<i>Non-Interactive Providers</i>	24
Chapter 4	System View	26
4.1	System Architecture Overview	26
4.2	Advanced Distribution Server (ADS)	27
4.3	Advanced Data Hub (ADH)	28
4.4	Elektron	29
4.5	Data Feed Direct	30
4.6	Internet Connectivity via HTTP and HTTPS.....	31
4.7	Direct Connect	32
Chapter 5	Model and Package Overviews	33
5.1	Transport API Models	33

5.1.1	<i>Open Message Model (OMM)</i>	33
5.1.2	<i>Reuters Wire Format (RWF)</i>	33
5.1.3	<i>Domain Message Model</i>	33
5.2	Packages	34
5.2.1	<i>Transport Package</i>	34
5.2.2	<i>Data Package</i>	34
5.2.3	<i>Message Package</i>	34
Chapter 6	Building an OMM Consumer	35
6.1	Overview	35
6.2	Establish Network Communication	35
6.3	Perform Login Process	36
6.4	Obtain Source Directory Information	36
6.5	Load or Download Necessary Dictionary Information	37
6.6	Issue Requests and/or Post Information	37
6.7	Log Out and Shut Down	37
6.8	Additional Consumer Details	38
Chapter 7	Building an OMM Interactive Provider	39
7.1	Overview	39
7.2	Establish Network Communication	39
7.3	Perform Login Process	40
7.4	Provide Source Directory Information	40
7.5	Provide or Download Necessary Dictionaries	40
7.6	Handle Requests and Post Messages	41
7.7	Disconnect Consumers and Shut Down	41
7.8	Additional Interactive Provider Details	42
Chapter 8	Building an OMM NIP	43
8.1	Overview	43
8.2	Establish Network Communication	43
8.3	Perform Login Process	44
8.4	Perform Dictionary Download	44
8.5	Provide Source Directory Information	44
8.6	Provide Content	45
8.7	Log Out and Shut Down	45
8.8	Additional NIP Details	45
Chapter 9	Encoding and Decoding Conventions	46
9.1	Concepts	46
9.1.1	<i>Data Types</i>	46
9.1.2	<i>Composite Pattern of Data Types</i>	47
9.2	Encoding Semantics	47
9.2.1	<i>Init and Complete Suffixes</i>	47
9.2.2	<i>The Encode Iterator: RsslEncodeIterator</i>	48
9.2.3	<i>Content Roll Back with Example</i>	51
9.3	Decoding Semantics and RsslDecodeIterator	52
9.3.1	<i>The Decode Iterator: RsslDecodeIterator</i>	52
9.3.2	<i>Functions for use with RsslDecodeIterator</i>	52
9.3.3	<i>RsslDecodeIterator: Basic Use Example</i>	53
9.4	Return Code Values	54
9.4.1	<i>Success Codes</i>	54
9.4.2	<i>Failure Codes</i>	56
9.5	Versioning	57

9.5.1	<i>Protocol Versioning</i>	57
9.5.2	<i>Library Versioning</i>	58
Chapter 10	Transport Package Detailed View.....	59
10.1	Concepts	59
10.1.1	<i>Transport Types</i>	60
10.1.2	<i>RSSL Channel Structure</i>	60
10.1.3	<i>RSSL Server Structure</i>	65
10.1.4	<i>Transport Error Handling</i>	65
10.1.5	<i>General Transport Return Codes</i>	66
10.1.6	<i>Application Lifecycle</i>	67
10.2	Initializing and Uninitializing the Transport.....	68
10.2.1	<i>RSSL Initialization and Uninitialization Functions</i>	68
10.2.2	<i>Initialization Reference Counting with Example</i>	69
10.2.3	<i>Transport Locking Models</i>	70
10.3	Creating the Connection	71
10.3.1	<i>Network Topologies</i>	71
10.3.2	<i>Creating the Outbound Connection: rsslConnect</i>	74
10.3.3	<i>rsslConnect Outbound Connection Creation Example</i>	81
10.3.4	<i>Tunneling Connection Keep Alive</i>	82
10.4	Server Creation and Accepting Connections	83
10.4.1	<i>Creating a Listening Socket</i>	83
10.4.2	<i>Accepting Connection Requests</i>	89
10.4.3	<i>Compression Support</i>	91
10.5	Channel Initialization	93
10.5.1	<i>rsslInitChannel Function</i>	93
10.5.2	<i>RsslInProglInfo Structure</i>	94
10.5.3	<i>Calling rsslInitChannel</i>	94
10.5.4	<i>rsslInitChannel Return Codes</i>	94
10.5.5	<i>rsslInitChannel Example</i>	95
10.6	Reading Data	96
10.6.1	<i>rsslRead Function</i>	97
10.6.2	<i>rsslRead Return Codes</i>	97
10.6.3	<i>rsslRead Example</i>	99
10.6.4	<i>rsslReadEx Function</i>	101
10.7	Writing Data: Overview	102
10.8	Writing Data: Obtaining a Buffer	103
10.8.1	<i>Buffer Management Functions</i>	104
10.8.2	<i>rsslGetBuffer Return Values</i>	104
10.9	Writing Data to a Buffer	106
10.9.1	<i>rsslWrite Function</i>	106
10.9.2	<i>rsslWrite Flag Enumeration Values</i>	107
10.9.3	<i>rsslWriteEx Function</i>	107
10.9.4	<i>Compression</i>	108
10.9.5	<i>Fragmentation</i>	109
10.9.6	<i>rsslWrite Return Codes</i>	110
10.9.7	<i>rsslGetBuffer and rsslWrite Example</i>	111
10.10	Managing Outbound Queues	114
10.10.1	<i>Ordering Queued Data: rsslWrite Priorities</i>	114
10.10.2	<i>rsslFlush Function</i>	116
10.10.3	<i>rsslFlush Return Codes</i>	116
10.10.4	<i>rsslFlush Example</i>	117
10.11	Packing Additional Data into a Buffer.....	118
10.11.1	<i>RsslPackBuffer Return Values</i>	118
10.11.2	<i>Example: rsslGetBuffer, RsslPackBuffer, and rsslWrite</i>	119

10.12	Ping Management	122
10.12.1	<i>Ping Timeout</i>	122
10.12.2	<i>rsslPing Function</i>	123
10.12.3	<i>rsslPing Return Values</i>	123
10.12.4	<i>rsslPing Example</i>	124
10.13	Closing Connections	125
10.13.1	<i>Functions for Closing Connections</i>	125
10.13.2	<i>Close Connections Example</i>	125
10.14	Utility Functions	126
10.14.1	<i>General Transport Utility Functions</i>	126
10.14.2	<i>RsslChannelInfo Structure Members</i>	127
10.14.3	<i>multicastStats Options</i>	129
10.14.4	<i>ComponentInfo Option</i>	129
10.14.5	<i>RsslServerInfo Structure Members</i>	130
10.14.6	<i>rsslIoctl Option Values</i>	130
10.14.7	<i>rsslServerIoctl Option Values</i>	131
10.15	XML Tracing	132
10.15.1	<i>RsslTraceOptions Structure Members</i>	132
10.15.2	<i>RsslTraceCodes Flag Enumeration Values</i>	132

Chapter 11 Data Package Detailed View 133

11.1	Concepts	133
11.2	Primitive Types.....	133
11.2.1	<i>RsslReal</i>	137
11.2.2	<i>RsslDate</i>	141
11.2.3	<i>RsslTime</i>	142
11.2.4	<i>RsslDateTime</i>	143
11.2.5	<i>RsslQos</i>	145
11.2.6	<i>RsslState</i>	148
11.2.7	<i>RsslArray</i>	153
11.2.8	<i>RsslBuffer</i>	158
11.2.9	<i>RMTES Decoding</i>	159
11.2.10	<i>General Primitive Type Utility Functions</i>	163
11.3	Container Types	164
11.3.1	<i>RsslFieldList</i>	167
11.3.2	<i>RsslElementList</i>	176
11.3.3	<i>RsslMap</i>	184
11.3.4	<i>RsslSeries</i>	195
11.3.5	<i>RsslVector</i>	203
11.3.6	<i>RsslFilterList</i>	213
11.3.7	<i>Non-RWF Container Types</i>	221
11.4	Permission Data	223
11.5	Summary Data	223
11.6	Set Definitions and Set-Defined Data	224
11.6.1	<i>Set-Defined Primitive Types</i>	225
11.6.2	<i>Set Definition Use</i>	228
11.6.3	<i>Set Definition Database</i>	231

Chapter 12 Message Package Detailed View 244

12.1	Concepts	244
12.1.1	<i>Common Message Base</i>	244
12.1.2	<i>Message Key</i>	247
12.1.3	<i>Stream Identification</i>	250
12.2	RSSL Messages	252
12.2.1	<i>RSSL Request Message Class</i>	252

12.2.2	<i>RSSL Refresh Message Class</i>	256
12.2.3	<i>RSSL Update Message Class</i>	260
12.2.4	<i>RSSL Status Message Class</i>	263
12.2.5	<i>RSSL Close Message Class</i>	265
12.2.6	<i>RSSL Generic Message Class</i>	266
12.2.7	<i>RSSL Post Message Class</i>	268
12.2.8	<i>RSSL Acknowledgment Message Class</i>	271
12.2.9	<i>The RSSL Message Union</i>	273
Chapter 13	Advanced Messaging Concepts	284
13.1	Multi-Part Message Handling	284
13.2	Stream Priority	284
13.3	Stream Quality of Service	285
13.4	Item Group Use	286
13.4.1	<i>Item Group Buffer Contents</i>	286
13.4.2	<i>Item Group Utility Functions</i>	287
13.4.3	<i>Group Status Message Information</i>	287
13.4.4	<i>Group Status Responsibilities by Application Type</i>	287
13.5	Single Open and Allow Suspect Data Behavior	288
13.6	Pause and Resume	289
13.7	Batch Messages	290
13.7.1	<i>Batch Request</i>	290
13.7.2	<i>Batch Reissue</i>	291
13.7.3	<i>Batch Close</i>	292
13.7.4	<i>Batch Request Encoding Example</i>	293
13.7.5	<i>Batch Reissue Encoding Example</i>	295
13.7.6	<i>Batch Close Encoding Example</i>	296
13.8	Dynamic View Use	297
13.8.1	<i>RDMViewTypes Enumerated Names</i>	299
13.8.2	<i>Dynamic View RsslRequestMsg Encoding Example</i>	299
13.9	Posting	301
13.9.1	<i>Post Message Encoding Example</i>	302
13.9.2	<i>Post Acknowledgement Encoding Example</i>	303
13.10	Visible Publisher Identifier (VPI)	304
13.10.1	<i>VPI Example: Using RsslPostUserInfo to Obtain VPI Data</i>	305
13.10.2	<i>VPI Example: Populating VPI in Post Messages from Consumer Applications</i>	305
13.10.3	<i>VPI Example: Getting VPI from Post Messages</i>	306
13.11	TREP Authentication	307
13.12	Private Streams	308
Appendix A	Item and Group State Decision Table	310
Appendix B	Error Codes	312
Appendix C	Document Revision History	315

List of Figures

Figure 1.	Network Diagram Notation	4
Figure 2.	UML Diagram Notation	4
Figure 3.	OMM-Based Product Offerings	5
Figure 4.	Transport API: Core Diagram.....	5
Figure 5.	TREP Infrastructure	11
Figure 6.	Transport API as a Consumer.....	12
Figure 7.	Batch Request.....	13
Figure 8.	View Request Diagram	14
Figure 9.	Symbol List: Basic Scenario.....	16
Figure 10.	Symbol List: Accessing the Entire ADS Cache	16
Figure 11.	Symbol List: Requesting Symbol List Streams via the Transport API Reactor	17
Figure 12.	Server Symbol List	18
Figure 13.	Posting into a Cache	19
Figure 14.	OMM Post with Legacy Inserts	20
Figure 15.	Private Stream Scenarios	21
Figure 16.	Provider Access Point	22
Figure 17.	Interactive Providers	23
Figure 18.	NIP: Point-To-Point	25
Figure 19.	NIP: Multicast	25
Figure 20.	Typical TREP Components	26
Figure 21.	Transport API and Advanced Distribution Server	27
Figure 22.	Transport API and the Advanced Data Hub.....	28
Figure 23.	Transport API and Elektron.....	29
Figure 24.	Transport API and Data Feed Direct.....	30
Figure 25.	Transport API and Internet Connectivity	31
Figure 26.	Transport API and Direct Connect	32
Figure 27.	Transport API and the Composite Pattern	47
Figure 28.	Application Lifecycle.....	67
Figure 29.	Unified TCP Network.....	71
Figure 30.	TCP Connection Creation	72
Figure 31.	Unified Multicast Network.....	72
Figure 32.	Segmented Multicast Network	73
Figure 33.	Multicast Connection Creation	73
Figure 34.	Consuming Multicast Data	74
Figure 35.	Transport API Server Creation	83
Figure 36.	Transport API Writing Flow Chart	103
Figure 37.	rss1Write Priority Scenario	114
Figure 38.	Item Group Example	286
Figure 39.	Batch Request Interaction Example'	291
Figure 40.	Batch Reissue (Pause) Interaction Example	292
Figure 41.	Batch Close Interaction Example	293

List of Tables

Table 1:	Acronyms and Abbreviations	1
Table 2:	API Performance Comparison	8
Table 3:	Capabilities by API	8
Table 4:	Layer-Specific Capabilities	10
Table 5:	Rssl Encoder Utility Functions.....	49
Table 6:	Rssl Decoder Utility Functions.....	52
Table 7:	Data and Message Package Success Return Codes	54
Table 8:	Data and Message Package Failure Return Codes	56
Table 9:	Rssl LibraryVersionInfo Structure Members.....	57
Table 10:	Rssl LibraryVersionInfo Structure Members.....	58
Table 11:	Library Version Utility Functions.....	58
Table 12:	Rssl Channel Structure Members.....	61
Table 13:	RSSL Connection State Values	62
Table 14:	RSSL ConnectionType Values.....	63
Table 15:	Rssl Server Structure Members.....	65
Table 16:	Rssl Error Structure Members.....	65
Table 17:	General Transport Return Codes	66
Table 18:	RSSL Initialization and Uninitialization Functions	68
Table 19:	RSSL Initialize Locking Options	70
Table 20:	rssl Connect Function	74
Table 21:	Rssl ConnectOptions Structure Members.....	75
Table 22:	Rssl ConnectOptions.connectInfo Options.....	77
Table 23:	Rssl ConnectOptions.tcpOpts Options.....	78
Table 24:	Rssl ConnectOptions.mulicastOpts Options.....	78
Table 25:	Rssl ConnectOptions.shmemOpts Options.....	80
Table 26:	Rssl ConnectOptions.seqMulicastOpts Options.....	80
Table 27:	Rssl ConnectOptions Utility Function.....	80
Table 28:	rssl Bind Function	83
Table 29:	Rssl BindOptions Structure Members.....	83
Table 30:	Rssl BindOptions.tcpOpts Options.....	87
Table 31:	Rssl BindOptions Utility Function	87
Table 32:	rssl Accept Function	89
Table 33:	Rssl AcceptOptions Structure Members.....	89
Table 34:	Rssl AcceptOptions Utility Functions	89
Table 35:	RSSL Compression Types	91
Table 36:	rssl InitChannel Function	93
Table 37:	Rssl InProgInfo Structure Members.....	94
Table 38:	rssl InitChannel Return Codes.....	95
Table 39:	Rssl Channel Function	97
Table 40:	rssl Read Return Codes	97
Table 41:	rssl ReadEx Function.....	101
Table 42:	rssl ReadOutArgs Options	101
Table 43:	RsslReadFlagsOut Enumerations	102
Table 44:	rssl ReadInArgs Option	102
Table 45:	Buffer Management Functions	104
Table 46:	rssl GetBuffer Return Values	105
Table 47:	rssl Write Function	106
Table 48:	rssl WriteFlags	107
Table 49:	rssl WriteEx Function	107
Table 50:	rssl ReadInArgs Options	107
Table 51:	rssl ReadInArgs Options	108

Table 52: rssiWriteFlagsIn Enumerations	108
Table 53: rssiWrite Return Codes.....	110
Table 54: rssiWrite Priority Value Enumerations	115
Table 55: rssiFlush Function	116
Table 56: rssiFlush Return Codes.....	116
Table 57: RssiPackBuffer Function	118
Table 58: RssiPackBuffer Return Values.....	118
Table 59: rssiPing function	123
Table 60: rssiPing Return Codes.....	123
Table 61: RSSL Connection Closing Functionality	125
Table 62: Transport Utility Functions	126
Table 63: RssiChannelInfo Structure Members.....	127
Table 64: multiicastStats Options.....	129
Table 65: componentInfo Option.....	129
Table 66: RssiServerInfo Structure Members.....	130
Table 67: rssiLocl Option Values.....	130
Table 68: rssiServerLocl Option Values.....	131
Table 69: RssiTraceOptions Structure Members.....	132
Table 70: RssiTraceCodes Option Values	132
Table 71: Transport API Primitive Types	134
Table 72: RssiReal Structure Members	137
Table 73: RssiRealHints Enumeration Values.....	137
Table 74: RssiReal Utility Functions	140
Table 75: RssiDate Structure Members	141
Table 76: RssiDate Utility Functions	141
Table 77: RssiTime Structure Members.....	142
Table 78: RssiTime Utility Functions	142
Table 79: RssiDateTime Structure Members.....	143
Table 80: RssiDateTime Utility Functions	144
Table 81: RssiQos Structure Members	145
Table 82: RssiQos Timeliness Values	146
Table 83: RssiQos Rate Values	147
Table 84: RssiQos Utility Functions.....	147
Table 85: RssiState Structure Members	148
Table 86: RssiState Stream State Values.....	149
Table 87: RssiState Data State Values.....	149
Table 88: RssiState Code Values.....	150
Table 89: RssiState Utility Functions	152
Table 90: RssiArray Structure Members	153
Table 91: RssiArray Encode Functions	154
Table 92: RssiArray Decode Functions	156
Table 93: RssiArray Utility Functions	157
Table 94: RssiBuffer Structure Members	158
Table 95: RssiRmtesCacheBuffer Structure Members	159
Table 96: RssiRmtesCacheBuffer Decode Functions.....	159
Table 97: RssiRmtesCacheBuffer Utility Functions	160
Table 98: RMTES to Unicode Conversion Functions.....	160
Table 99: RssiU16Buffer Structure Members	160
Table 100: General Primitive Type Utility Functions	163
Table 101: Transport API Container Types.....	164
Table 102: RssiFielddList Structure Members	167
Table 103: RssiFielddList Flags.....	168
Table 104: RssiFielddEntry Structure Members	169
Table 105: RssiFielddList Encode Functions	170
Table 106: RssiFielddList Decode Functions	174

Table 107: Rssi FieldList Utility Functions.....	176
Table 108: Rssi ElementList Structure Members	176
Table 109: Rssi ElementList Flags.....	177
Table 110: Rssi ElementEntry Structure Members.....	177
Table 111: Rssi ElementList Encoding Interfaces.....	178
Table 112: Rssi ElementList Decode Functions.....	182
Table 113: Rssi ElementList Utility Functions	184
Table 114: Rssi Map Structure Members	184
Table 115: Rssi Map Flags	186
Table 116: Rssi MapEntry Structure Members	186
Table 117: Rssi MapEntry Flags.....	187
Table 118: Rssi MapEntry Actions.....	187
Table 119: Rssi MapEntry Encode Functions	188
Table 120: Rssi MapEntry Decode Functions	192
Table 121: Rssi Map Utility Functions.....	194
Table 122: Rssi Series Structure Members	195
Table 123: Rssi Series Flags.....	196
Table 124: Rssi SeriesEntry Structure Members	196
Table 125: Rssi SeriesEntry Encode Functions.....	197
Table 126: Rssi SeriesEntry Decode Functions.....	201
Table 127: Rssi Series Utility Functions	202
Table 128: Rssi Vector Structure Members	203
Table 129: Rssi Vector Flags.....	204
Table 130: Rssi VectorEntry Structure Members	205
Table 131: Rssi VectorEntry Flag.....	205
Table 132: Rssi VectorEntry Actions.....	206
Table 133: Rssi Vector Encode Functions	207
Table 134: Rssi Vector Decode Functions	211
Table 135: Rssi Vector Utility Functions	212
Table 136: Rssi FilterList Structure Members	213
Table 137: Rssi FilterList Flags	214
Table 138: Rssi FilterEntry Structure Members	214
Table 139: Rssi FilterEntry Flags.....	215
Table 140: Rssi FilterEntry Actions.....	215
Table 141: Rssi FilterList Encode Functions	216
Table 142: Rssi FilterList Decode Functions	219
Table 143: Rssi FilterList Utility Functions	220
Table 144: Non-RWF Type Encode Functions	221
Table 145: Set-Defined Primitive Types.....	225
Table 146: Rssi FieldSetDef Structure Member	228
Table 147: Rssi FieldSetDefEntry Structure Members.....	229
Table 148: Rssi ElementSetDef Structure Members	229
Table 149: Rssi ElementSetDefEntry Structure Members	230
Table 150: Rssi Local FieldSetDefDb Structure Members	231
Table 151: Rssi Local ElementSetDefDb Structure Members	232
Table 152: Local Set Definition Database Encode Functions	232
Table 153: Local Set Definition Database Decode Functions	233
Table 154: Local Set Definition Database Utility Functions	234
Table 155: Message Base Structure Members	244
Table 156: Message Class Information	246
Table 157: msgKey Structure Members.....	247
Table 158: Message Key Flags	248
Table 159: MsgKey Utility Functions.....	249
Table 160: Rssi RequestMsg Structure Members.....	252
Table 161: Rssi RequestMsg Flags.....	254

Table 162: Rssi RequestMsg Utility Functions	255
Table 163: Rssi RefreshMsg Structure Members	256
Table 164: Rssi RefreshMsg Flags.....	258
Table 165: Rssi RefreshMsg Utility Functions	259
Table 166: Rssi UpdateMsg Structure Members	260
Table 167: Rssi UpdateMsg Flags.....	262
Table 168: Rssi UpdateMsg Utility Functions.....	262
Table 169: Rssi StatusMsg Structure Members	263
Table 170: Rssi StatusMsg Flags.....	264
Table 171: Rssi StatusMsg Utility Functions.....	264
Table 172: Rssi CloseMsg Structure Members.....	265
Table 173: Rssi CloseMsg Flags.....	265
Table 174: Rssi CloseMsg Utility Functions.....	265
Table 175: Rssi GenericMsg Structure Members	266
Table 176: Rssi GenericMsg Flags.....	267
Table 177: Rssi GenericMsg Utility Functions	267
Table 178: Rssi PostMsg Structure Members.....	268
Table 179: Rssi PostMsg Flags	269
Table 180: Rssi PostRights Flags	270
Table 181: Rssi PostMsg Utility Functions	270
Table 182: Rssi AckMsg Structure Members	271
Table 183: Rssi AckMsg Flags.....	272
Table 184: Rssi AckMsg NakCode Values.....	272
Table 185: Rssi AckMsg Utility Functions	273
Table 186: Rssi Msg Encode Functions	273
Table 187: Rssi Msg Decode Functions	279
Table 188: Rssi Msg Utility Functions.....	281
Table 189: Item Group Utility Functions	287
Table 190: Single eOpen and All LowSuspectData Effects.....	289
Table 191: RDMViewTypes Values	299
Table 192: Item and Group State Decision Table	310
Table 193: Error Codes.....	312
Table 194: Transport API C Edition Document Revision History	315

Chapter 1 Transport API Developers Guide Introduction

1.1 About this Manual

This document is authored by Transport API architects and programmers who encountered and resolved many of the 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. 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. 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. 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.

1.3 Programming Language

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

1.4 Acronyms and Abbreviations

ACRONYM	Meaning
ADH	Advanced Data Hub is the horizontally scalable service component within Thomson Reuters Enterprise Platform (TREP) providing high availability for publication and contribution messaging, subscription management with optional persistence, conflation and delay capabilities.
ADS	Advanced Distribution Server is the horizontally scalable distribution component within Thomson Reuters Enterprise Platform (TREP) providing highly available services for tailored streaming and snapshot data, publication and contribution messaging with optional persistence, conflation and delay capabilities.
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ATS	Advanced Transformation System
DACS	Data Access Control System

Table 1: Acronyms and Abbreviations

ACRONYM	Meaning
DMM	Domain Message Model
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. Formerly referred to as UPA.
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
RDM	Reuters Domain Model
Reactor	The Reactor is a low-level, open-source, easy-to-use layer above ETA. It offers heartbeat management, connection and item recovery, and many other features to help simplify application code for users.
RFA	Robust Foundation API
RMTES	Reuters Multi-Lingual Text Encoding Standard
RSSL	Reuters Source Sink Library
RWF	Reuters Wire Format, a Thomson Reuters proprietary 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 Manuals
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. The [Thomson Reuters Professional Developer Community](#)

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

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 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

Added the TREP Authentication feature, which provides enhanced authentication functionality when used with TREP and DACS. This feature requires TREP 3.1 or later. For further details, refer to Section 13.11.

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

2.1 What is the Transport API?

The Transport API (also known as the RSSL API) is the customer release of Thomson Reuters's low-level internal API, currently used by the Thomson Reuters Enterprise Platform (TREP) and its dependent APIs for optimal distribution of OMM/RWF data.

The Transport API is currently used by products such as the Advanced Distribution Server (ADS), Advanced Data Hub (ADH), Robust Foundation API (RFA), EDF-D, Elektron, and Eikon. Due to its well-integrated and common usage across these products, the Transport API allows clients to write applications for use with Thomson Reuters Enterprise Platform (TREP) to achieve the highest performance, highest throughput, and lowest latency.

The Transport API supports all constructs available as part of the Open Message Model. It complements RFA and the Message API by allowing users to choose the type of functionality and layer (Session or Transport) at which they want to access the TREP. With the addition of the Transport API, customers have a choice between a feature-loaded session-level API (i.e., the Message API) and high-performance transport-level API (i.e., the Transport API).

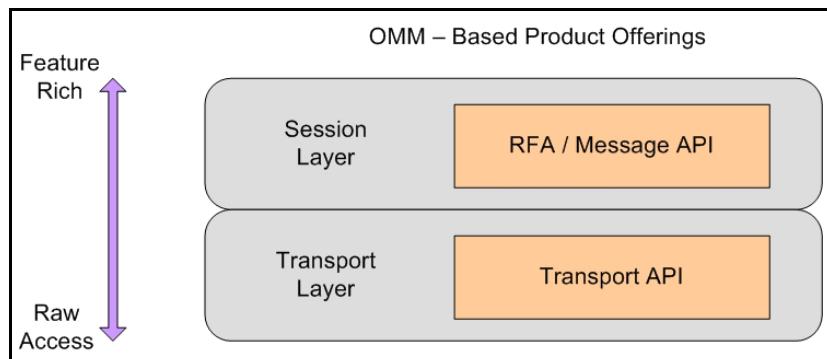


Figure 3. OMM-Based Product Offerings

The Transport API is a low-level API that provides application developers with the most flexible development environment and is the foundation on which all Thomson Reuters OMM-based components are built. By utilizing an API at the transport level, a client can write to the same API as the ADS / ADH and achieve the same levels of performance.

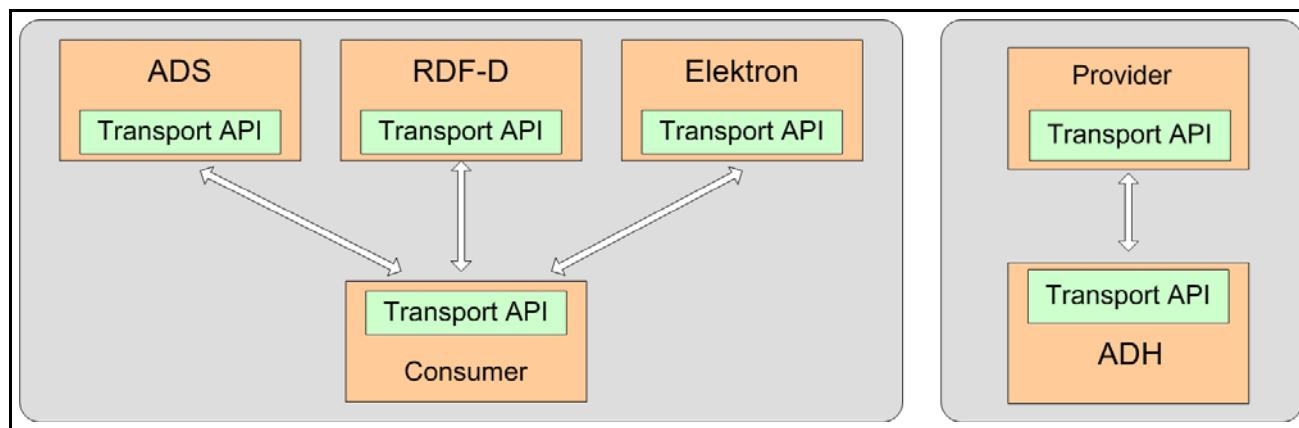


Figure 4. Transport API: Core Diagram

2.2 Transport API Features

The Transport API is:

- Available as both a C-based and Java-based API.
- 64-bit.
- Thread-safe and thread-aware.
- Capable of handling:
 - Any and all OMM primitives and containers.
 - All Domain Models, including those defined by Thomson Reuters as well as other user-defined models.
- A reliable, transport-level API which includes OMM encoders/decoders.

Additionally, the Transport API provides an ANSI Page parser to encode/decode ANSI sequences and a DACS Library to allow generation of DACS Locks.

2.2.1 General Capabilities

The Transport API provides general capabilities independent of the type of application. The Transport API:

- Supports fully connected or unified network topologies as well as segmented topologies.
- Supports multiple network session types, including TCP, HTTP, and multicast-based networks.
- Can internally fragment and reassemble large messages.
- Can pack multiple, small messages into the same network buffer.
- Can perform data compression and decompression internally.
- Can choose its locking model based on need. Locking can be enabled globally, within a connection, or disabled entirely, thus allowing clients to develop single-threaded, multi-threaded, thread-safe, or thread-aware solutions.
- Has full control over the number of message buffers and can dynamically increase or decrease this quantity during runtime.
- Does not have external configuration, log file, or message file dependencies: everything is programmatically supplied, where the user can define any external configuration or logging according to their needs.
- Allows users to write messages at different priority levels, allowing higher priority messages to be sent before lower priority messages.

2.2.2 Consumer Applications

You can use the Transport API to create consumer-based applications that can:

- Make streaming and snapshot-based subscription requests to the ADS.
- Send batch, views, and symbol list requests to the ADS.
- Support pause and resume on active data streams with the ADS.
- Send post messages to the ADS (for consumer-based publishing and contributions).
- Send and receive generic messages with ADS.
- Establish a private stream.
- Transparently use HTTP to communicate with an ADS by tunneling through the Internet.

2.2.3 Provider Applications: Interactive

You can use the Transport API to create interactive providers that can:

- Receive requests and respond to streaming and snapshot-based Requests from ADH (previously known as Managed or Sink-Driven Server applications).
- Receive and respond to batch, views, and symbol list requests from ADH.
- Receive and respond to requests for a Private Stream from the ADH.
- Receive requests for pause and resume on active data streams.
- Receive and acknowledge post messages (used receiving consumer- based Publishing and Contributions) from ADH.
- Send and receive Generic Messages with ADH.

Additionally, you can use the Transport API to create server-based applications that can accept multiple connections from ADH, or allows multiple ADHs to connect to a provider.

2.2.4 Provider Applications: Non-Interactive

Using the Transport API, you can write non-interactive applications that start up and begin publishing data to ADH (previously known as Source-Driven (Src-Driven) or broadcast-style server applications). This includes both TCP and UDP multicast-based Non-Interactive Provider (NIP) applications.

2.3 Performance and Feature Comparison

Though TREP's core infrastructure can achieve great performance numbers, such performance can suffer from bottlenecks caused by using the rich features offered in certain APIs (i.e., RFA) when developing high-performance applications. By writing to the Transport API, a client can leverage the full throughput and low latency of the core infrastructure while bypassing the full set of RFA's features. For a comparison of API capabilities and features, refer to Section 2.4.

As illustrated in Figure 2, core infrastructure components (as well as their performance test tools, such as `rmdstestclient` and `sink_driven_src`) are all written to the Transport API. A Transport API-based application's maximum achievable performance (latency, throughput, etc) is determined by the infrastructure component to which it connects. Thus, to know performance metrics, you should look at the performance numbers for the associated infrastructure component. For example:

- If a Transport API consumer application talks to the ADS and you want to know the maximum throughput and latency of the consumer, look at the performance numbers for the ADS configuration you use.
- If a Transport API provider application talks to an ADH and you want to know the maximum throughput and latency of the Transport API provider, look at the performance numbers for the ADH Configuration you use.



Tip: The Transport API now ships with API performance tools and additional documentation to which you can refer which you can use to arrive at more-specific results for your environment.

When referring to TREP infrastructure documentation, look for Transport API or RSSL numbers (TREP documentation often refers to the Transport API as RSSL), which will give the performance and latency of the Transport API and the associated core infrastructure component.

The following table compares existing API products and their performance. Key factors are latency, throughput, memory, and thread safety. Results may vary depending on whether you use of watch lists and memory queues and according to your hardware and operating system. Typically, when measuring performance on the same hardware and operating system, these comparisons remain consistent.

API	THREAD SAFETY	THROUGHPUT	LATENCY	MEMORY FOOTPRINT
Transport API	Safe and Aware	Very High	Lowest	Lowest
Message API	Safe and Aware	High	Low	Medium
RFA	Safe and Aware	High	Low	Medium (watch list, allows optional queues)
SFC C++	None	Medium	High	Medium – High (watch list, cache)
SSL 4.5	Big Lock	Medium - High	Medium	Low (watch list optional)
SSL 4.0 Classic Edition	Big Lock	Low - Medium	Medium - High	Medium (watch list)

Table 2: API Performance Comparison

2.4 Functionality: Which API to Choose?

To make an informed decision on which API to use, you should balance the tradeoffs between performance and functionality (for performance comparisons, refer to Section 2.3).

2.4.1 General Capability Comparison

The following table compares the general capabilities of RFA, the Message API, and the Transport API.

CAPABILITY TYPE	CAPABILITY	RFA	MESSAGE API	TRANSPORT API
Transport	Compression via OMM	X	X	X
	HTTP Tunneling (RWF)	X	X	X
	RV Multicast			
	TCP/IP: RWF	X		X
	Unidirectional Shared Memory			X
	Reliable Multicast	X	X	X
Application Type	Consumer	X	X	X
	Provider: Interactive	X		X
	Provider: Non-Interactive	X		X

Table 3: Capabilities by API

Capability Type	Capability	RFA	Message API	Transport API
General	Batch Support	X	X	X
	Generic Messages	X	X	X
	Pause/Resume	X	X	X
	Posting	X	X	X
	Snapshot Requests	X	X	X
	Streaming Requests	X	X	X
	Private Streams	X	X	X
	Views	X	X	X
Domain Models	Custom Data Model Support	X	X	X
	RDM: Dictionary	X	X	X
	RDM: Login	X	X	X
	RDM: Market Price	X	X	X
	RDM: MarketByOrder	X	X	X
	RDM: MarketByPrice	X	X	X
	RDM: Market Maker	X	X	X
	RDM: Service Directory	X	X	X
	RDM: Symbol List	X	X	X
	RDM: Yield Curve	X	X	X
Encoders/Decoders	AnsiPage	X	The Message API supports passing AnsiPage data, but does not currently have an ANSI parser.	X
	DACS Lock	X	The Message API might include this capability in a future release	X
	OMM	X	X	X
	RMTEs	X	X	X
	TS1 Parser	X	X	

Table 3: Capabilities by API (Continued)

2.4.2 Layer-Specific Capability Comparison

The following table lists capabilities specific to the individual session-layer (RFA and Message API) or transport-layer (Transport API).

RFA uses information provided from the Transport API and creates specific implementations of capabilities. Though these capabilities are not implemented in the Transport API, Transport API-based applications can use the information provided by

the Transport API to implement the same functionality (i.e., as provided by RFA). Additionally, Transport API Value Added Components offer fully-supported reference implementations for much of this functionality.

CAPABILITY	RFA	MESSAGE API	TRANSPORT API
Config: file-based	X	X	*
Config: programmatic	X	X	X
Group fanout to items	X	X	*
Logging: file-based	X	X	*
Logging: programmatic	X		X
QoS Management	X		*
Network Pings: automatic	X	X	*
Recovery: connection	X	X	*
Recover: items	X	X	*
Request routing	X		*
Session management	X		*
Service Groups	X		*
Single Open: API-based	X	X	*
Warm Standby: API-based	X	Planned for future release	*
Watchlist	X	X	*
Controlled fragmentation and assembly of large messages			X
Controlled locking / threading model			X
Controlled dynamic message buffers with ability to programmatically modify during runtime			X
Controlled message packing			X
Messages can be written at different priority levels			X

* Transport API users can get the same functionality but must implement it themselves or use the Transport API Value Added Component libraries or source code.

Table 4: Layer-Specific Capabilities

Chapter 3 Consumers and Providers

3.1 Overview

For those familiar with previous API products or concepts from TREP, Rendezvous, or Triarch, we map how the Transport API implements the same functionality.

At a very high level, the TREP system facilitates controlled and managed interactions between many different service **providers** and **consumers**. Thus, TREP is a real-time, streaming Service Oriented Architecture (SOA) used extensively as middleware integrating financial-service applications. While providers implement services and expose a certain set of capabilities (e.g. content, workflow, etc.), consumers use the capabilities offered by providers for a specific purpose (e.g., trading screen applications, black-box algorithmic trading applications, etc.). In some cases, a single application can function as both a consumer and a provider (e.g., a computation engine, value-add server, etc.).

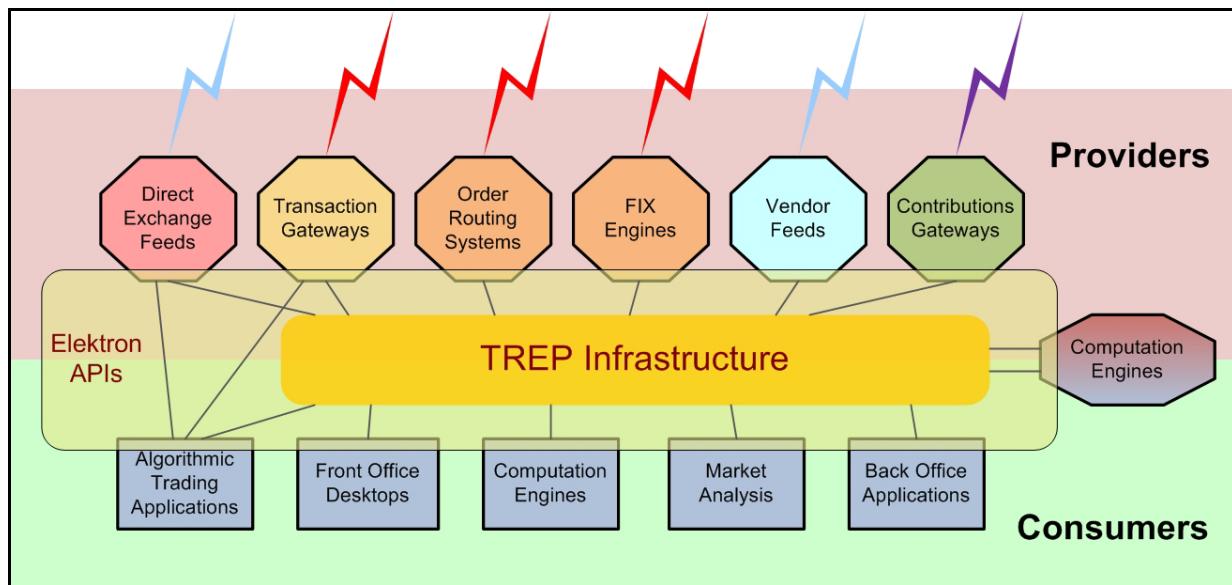


Figure 5. TREP Infrastructure

To access needed capabilities, consumers always interact with a provider, either directly and/or via TREP. Consumer applications that want the lowest possible latency can communicate directly via TREP APIs with the appropriate service providers. However, you can implement more complex deployments (i.e., integrating multiple providers, managing local content, automated resiliency, scalability, control, and protection) by placing the TREP infrastructure between provider and consumer applications.

3.2 Consumers

Consumers make use of capabilities offered by providers through access points. To interact with a provider, the consumer must attach to a consumer access point. Access points manifest themselves in two different forms:

- A **concrete access point**. A concrete access point is implemented by the service-provider application if it supports direct connections from consumers. The right-side diagram in Figure 6 illustrates a Transport API consumer connecting to Elektron via a direct access point.
- A **proxy access point**. A proxy access point is point-to-point based or multicast (according to your needs) and implemented by a TREP Infrastructure component (i.e., an ADS). Figure 6 also illustrates a Transport API consumer connecting to the provider by first passing through a proxy access point.

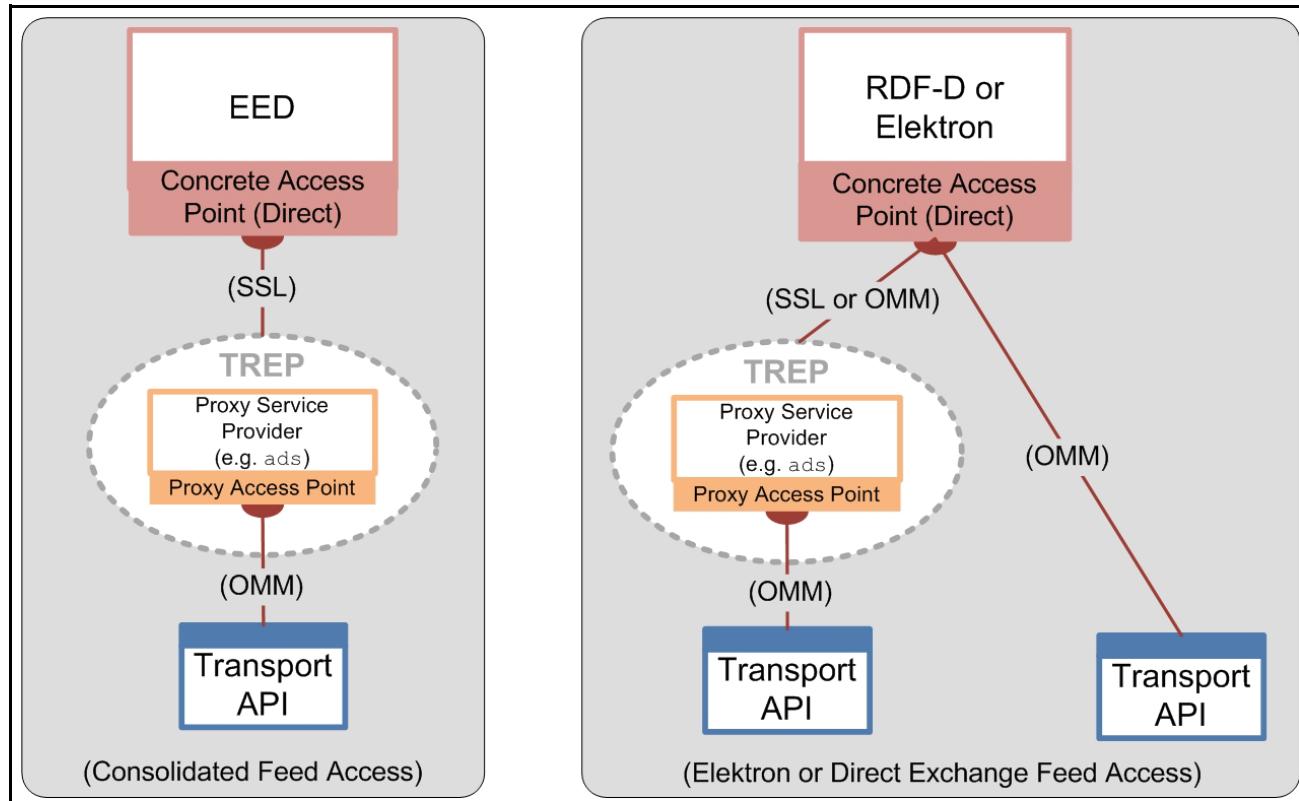


Figure 6. Transport API as a Consumer

Examples of consumers include:

- An application that subscribes to data via TREP, EDF, or Elektron.
- An application that posts data to TREP or Elektron (e.g., contributions/inserts or local publication into a cache).
- An application that communicates via generic messages with TREP or Elektron.
- An application that does any of the above via a private stream.

3.2.1 Subscriptions: Request/Response

After a consumer successfully logs into a provider (i.e., ADS or Elektron) and obtains a list of available sources, the consumer can then subscribe and receive data for various services. A consumer subscribes to a service or service ID that in turn maps to a service name in the Source Directory. Any service or service ID provides a set of items to its clients.

- If a consumer's request does not specify interest in future changes (i.e., after receiving a full response), the request is a classic ***snapshot request***. The data stream is considered closed after a full response of data (possibly delivered in multiple parts) is sent to the consumer. This is typical behavior when a user sends a non-streaming request. Because the response contains all current information, the stream is considered complete as soon as the data is sent.
- If a consumer's request specifies interest in receiving future changes (i.e., after receiving a full response), the request is considered to be a ***streaming request***. After such a request, the provider sends the consumer an initial set of data and then sends additional changes or "updates" to the data as they occur. The data stream is considered open until either the consumer or provider closes it. A consumer typically sends a streaming request when a user subscribes for an item and wants to receive every change to that item for the life of the stream.

Specialized cases of request / response include:

- Batches
- Views
- Symbol Lists
- Server Symbol Lists

3.2.2 Batches

A consumer can request multiple items using a single, client-based, request called a ***batch*** request. After the Transport API consumer sends an optimized batch request to the ADS, the ADS responds by sending the items as if they were opened individually so the items can be managed individually.

Figure 7 illustrates a Transport API consumer issuing a batch request for "TRI", "GE", and "INTC.O" and the resulting ADS responses.

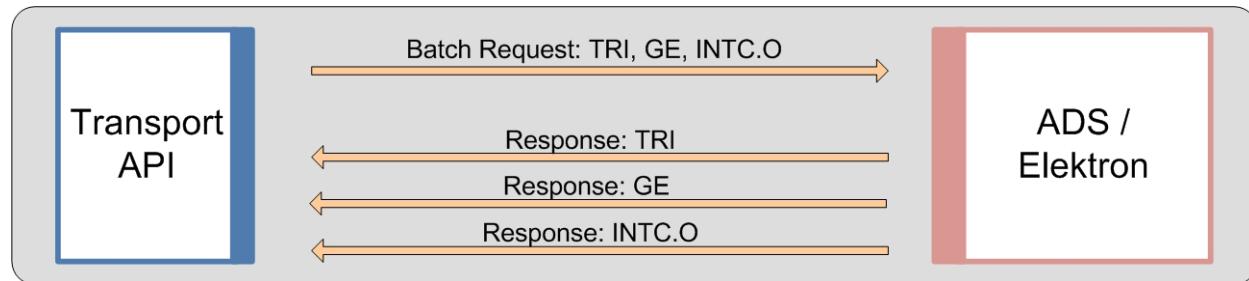


Figure 7. Batch Request

3.2.3 Views

The system reduces the amount of data that flows across the network by filtering out content in which the user is not interested. To improve performance and maximize bandwidth, you can configure the TREP to filter out certain fields to downstream users. When filtering, all consumer applications see the same subset of fields for a given item.

Another way of controlling filtering is to configure the consumer application to use **Views**. Using a view, a consumer requests a subset of fields with a single, client-based request (refer to Figure 8). The API then requests (from the ADS/Elektron) only the fields of interest. When the API receives the requested fields, it sends the subset back to the consumer. This is also called consumer-side (or request-side) filtering.

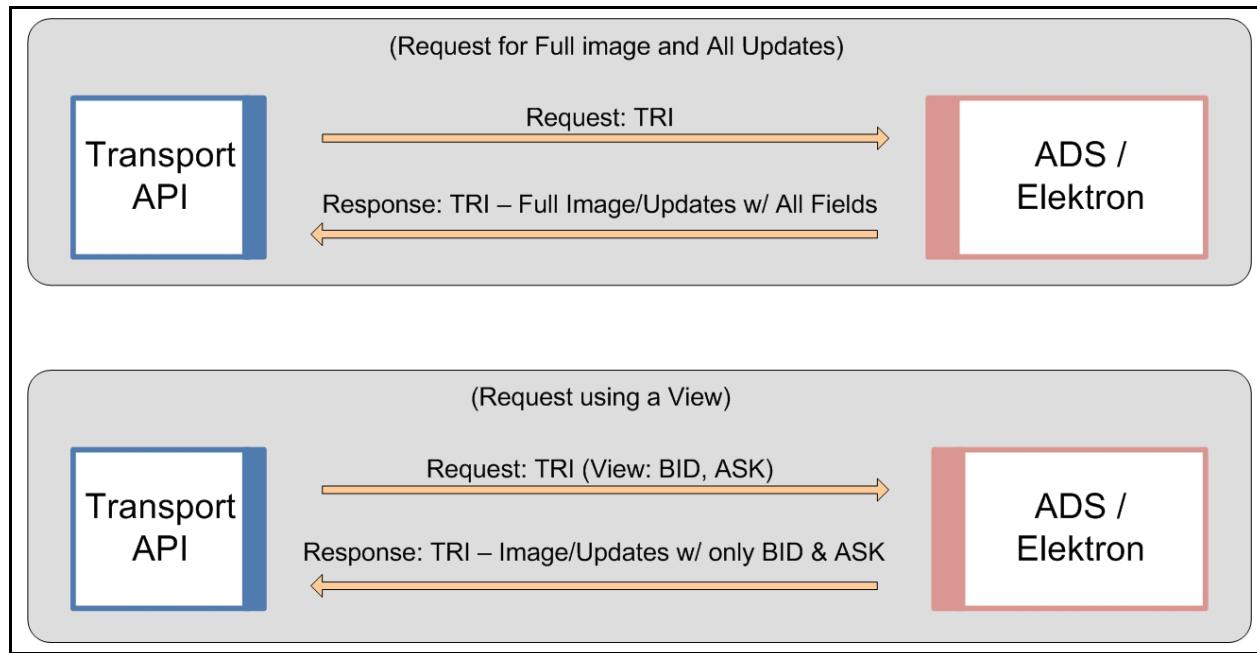


Figure 8. View Request Diagram

Views were designed to provide the same filtering functionality as the Legacy STIC device and SFC (based on its own internal cache) while optimizing network traffic.

Views, in conjunction with server-side filtering, can be a powerful tool for bandwidth optimization on a network. Users can combine a view with a batch request to send a single request to open multiple items using the same view.

3.2.4 Pause and Resume

The **Pause/Resume** feature optimizes network bandwidth. You can use Pause/Resume to reduce the amount of data flowing across the network for a single item or for many items that might already be openly streaming data to a client.

To pause/resume data, the client first sends a request to pause an item to the ADS. The ADS receives the pause request and stops sending new data to the client for that item, though the item remains open and in the ADS Cache. The ADS continues to receive messages from the upstream device (or feed) and continues to update the item in its cache (but because of the client's pause request, does not send the new data to the client). When the client wants to start receiving messages for the item again, the client sends a resume to the ADS, which then responds by sending an aggregated update or a refresh (a current image) to the client. After the ADS resumes sending data, the ADS sends all subsequent messages.

By using the Pause/Resume feature a client can avoid issuing multiple open/close requests which can disrupt the ADS and prolong recovery times. There are two main use-case scenarios for this feature:

- Clients with intensive back-end processing
- Clients that display a lot of data

3.2.4.1 Pause / Resume Use Case 1: Back-end Processing

In this use-case, a client application performs heavy back-end processing and has too many items open, such that the client is at the threshold for lowering the downstream update rate. The client now needs to run a specialized report, or do some other back-end processing. Such an increase in workload on the client application will negatively impact its downstream message traffic. The client does not want to back up its messages from the ADS and risk having ADS abruptly cut its connection, nor does the client want to close its own connection (or close all the items on the ADS) which would require the client to re-open all items after finishing its back-end processing.

In this case, the client application:

- Sends a single PAUSE message to the ADS to pause all the items it has open.
- Performs all needed back-end processing.
- Sends a Resume request to resume all the items it had paused.

After receiving the Resume request, the ADS sends a refresh (i.e., current image), to the client for all paused items and then continues to send any subsequent messages.

3.2.4.2 Pause / Resume Use Case 2: Display Applications

The second use case assumes the application displays a lot of data. In this scenario, the user has two windows open. One window has item "TRI" open and is updating (Window 1). The other has "INTC.O" open and is updating (Window 2). On his screen, the user moves Window 1 to cover Window 2 and the user can no longer see the contents of Window 2. In this case, the user might not need updates for "INTC.O" because the contents are obstructed from view. In this case, the client application can:

- Pause "INTC.O" as long as Window 2 is covered and out of view.
- Resume the stream for "INTC.O" when Window 2 moves back into view.

When Window 2 is again visible, the ADS sends a refresh, or current image, to the client for the item "INTC.O" and then continues to send any subsequent messages.

3.2.5 Symbol Lists

If a consumer wants to open multiple items but doesn't know their names, the consumer can first issue a request using a **Symbol List**. However, the consumer can issue such a request only if a provider exists that can resolve the symbol list name into a set of item names.

This replaces the functionality for clients that previously used Criteria-Based Requests (CBR) with the SSL 4.5 API.

The following diagram illustrates issuing a basic symbol list request. In this diagram, the consumer issues the request using a particular key name (**FRED**). The request flows through the platform to a provider capable of resolving the symbol list name (the interactive provider with **FRED** in its cache). The provider sends back all names that map to **FRED** (**TRI** and **GE**). After receiving the response, the client can then choose whether to open items; individually or by making a batch request for multiple items. A subsequent request is resolved by the first cache that contains the data (listed in the diagram as optional caches).

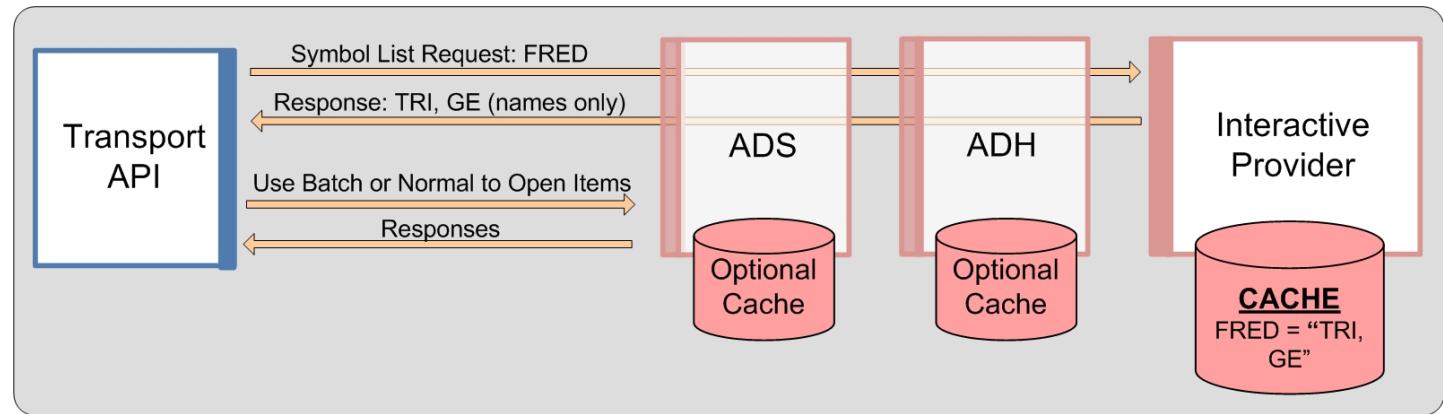


Figure 9. Symbol List: Basic Scenario

The following diagram illustrates how a consumer can access all items in the ADS Cache, effectively dumping the cache to the OMM client. In this scenario, the client requests the symbol list **_ADS_CACHE_LIST**. The ADS receives the request and responds with the names of all items in its cache. The client can then choose to open items individually, or make a batch request to open multiple items. The ADS provides an additional symbol list (**_SERVER_LIST**) for obtaining lists of items stored in specific ADH instances. For details on this symbol list, refer to the *ADS and ADH Software Installation Manuals*.

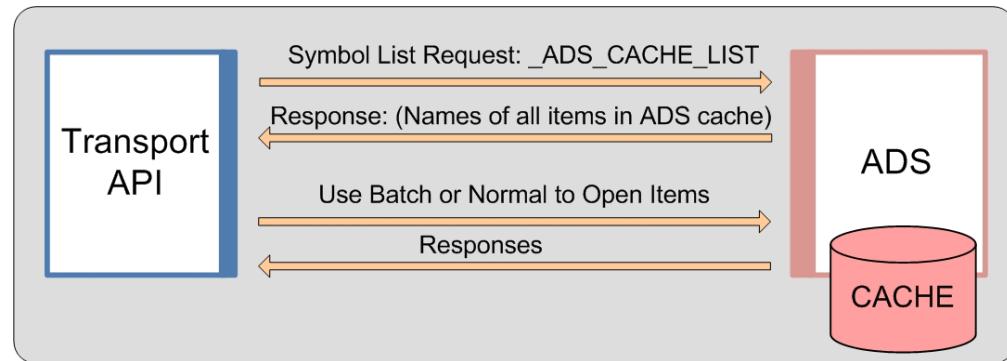


Figure 10. Symbol List: Accessing the Entire ADS Cache

3.2.5.1 Requesting Symbol List Data Streams

For consumer applications using the Transport API reactor value-add component on certain APIs: if the consumer watchlist is enabled, an application can indicate in its request that it wants streams for the items in the symbol list to be opened on its behalf. The reactor will internally process responses on the symbol list stream and open requests as new items appear in the list. The responses to these item requests will be provided to the application using negative **streamId** values.

The reactor supports this method with the ADS or in direct connections with interactive providers. For details on the model for requesting symbol list data streams, see the *Transport API RDM Usage Guide* specific to the API that you use.

Note: The reactor opens items from the symbol list as market price items, and uses the best available quality of service (QoS) advertised by the service in the provider's source directory response.

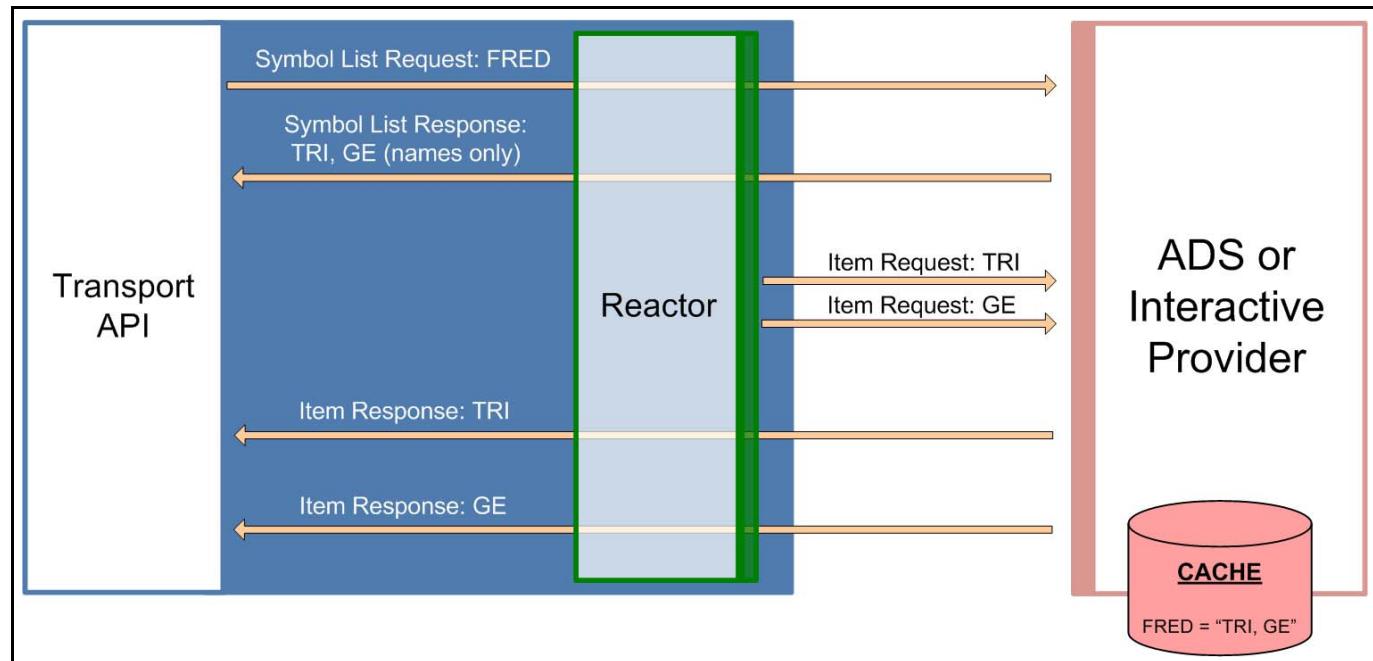


Figure 11. Symbol List: Requesting Symbol List Streams via the Transport API Reactor

3.2.5.2 Server Symbol Lists

Using certain Elektron APIs, client applications can request a list of all symbols maintained in the cache of all ADH servers across the network. Client applications start by first requesting a symbol list item `_SERVER_LIST` which will return a list of all servers and their supported domains. Each entry on that list is a symbol list item name formatted as follows

`_CACHE_LIST.serverId.domain`. Client applications can then spawn individual symbol list requests for servers and domains of interest using the symbol name `_CACHE_LIST.serverId.domain`. If `domain` is not provided, it defaults to 6.

The symbol list response for `_CACHE_LIST.serverId.domain` will include a list of all Level 1 or Level 2 items in the server cache. It will also include opened non-cached items but not items opened on private streams. The symbol list response will provide only item names, not item data.

The streams for `_SERVER_LIST` and `_CACHE_LIST.serverId.domain` requests will be kept open and updates will be sent to modify list of servers or list of items in server cache. These streams will be closed if a server is no longer available or it no longer supports a particular domain.

If the ADH is configured for source mirroring, a failover will trigger a server id change and will lead to closing of the relevant `_CACHE_LIST.serverId.domain` request and updating of the `_SERVER_LIST` to show the new server id after the failover. Clients will need to make a new symbol list request to the new server.

This feature provides the symbol list of all items in the ADH cache for both interactive and non-interactive services and is supported for both RSSL (symbol list) and SSL 4.5 (criteria) clients.

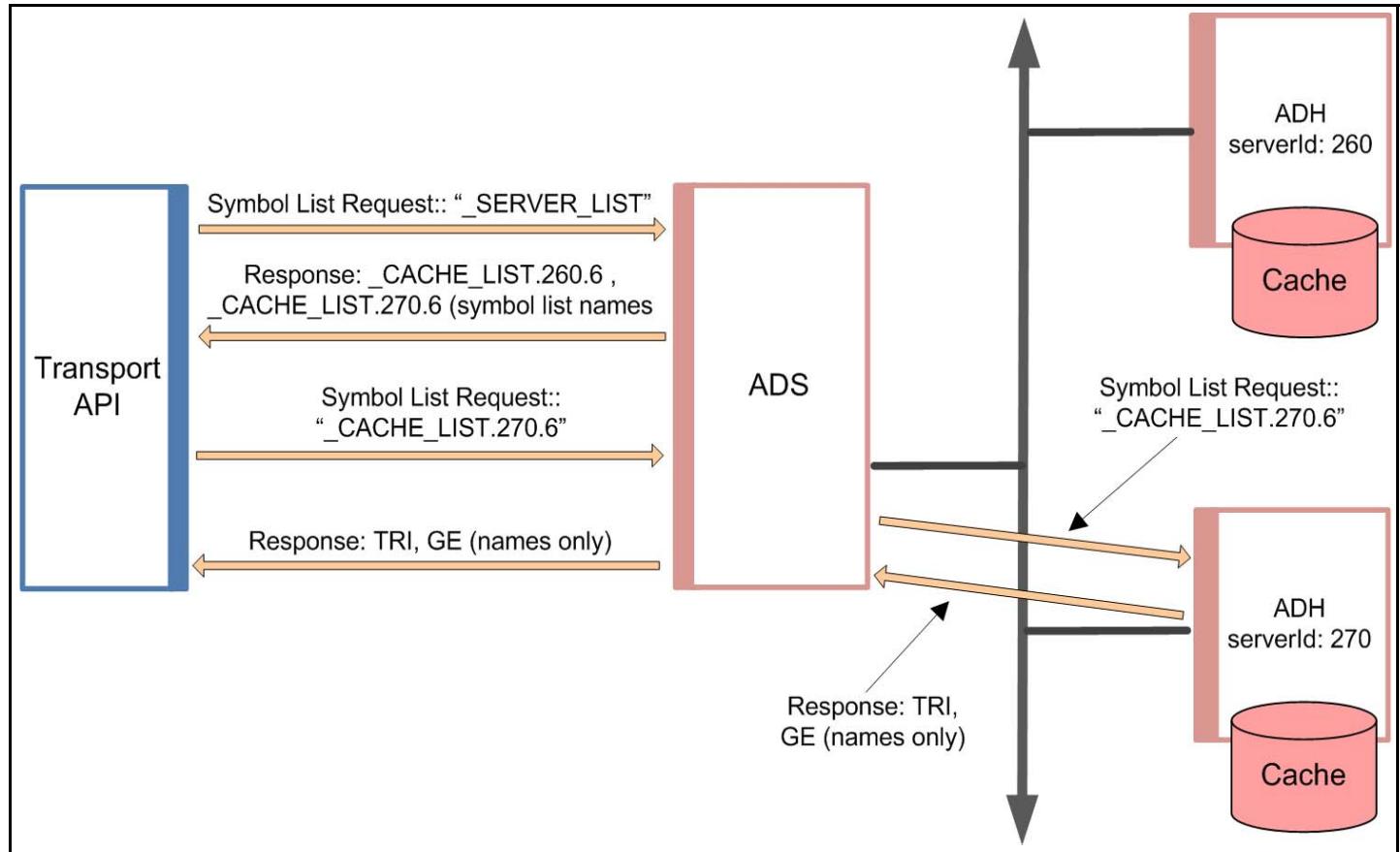


Figure 12. Server Symbol List

3.2.6 Posting

Through posting, API consumers can easily push content into any cache within the TREP (i.e., an HTTP POST request). Data contributions/inserts into the ATS or publishing into a cache offer similar capabilities today. When posting, API consumer applications reuse their existing sessions to publish content to any cache(s) residing within the TREP (i.e., service provider(s) and/or infrastructure components). When compared to spreadsheets or other applications, posting offers a more efficient form of publishing, because the application does not need to create a separate provider session or manage event streams. The posting capability, unlike unmanaged publishing or inserts, offers optional acknowledgments per posted message. The two types of posting are on-stream and off-stream:

- **On-Stream Post:** Before sending an on-stream post, the client must first open (request) a data stream for an item. After opening the data stream, the client application can then send a post. The route of the post is determined by the route of the data stream.
- **Off-Stream Post:** In an off-stream post, the client application can send a post for an item via a Login stream, regardless of whether a data stream first exists. The route of the post is determined by the Core Infrastructure (i.e., ADS, ADH, etc.) configuration.

3.2.6.1 Local Publication

The following diagram illustrates the benefits of posting.

Green and Red services support internal posting and are fully implemented within the ADH. In both cases the ADH receives posted messages and then distributes these messages to interested consumers. In the right-side segment, the ADS component has enabled caching (for the Red service). In this case posted messages received from connected applications are cached and distributed to these local applications before being forwarded (re-posted) up into the ADH cache. The Transport API can even post to provider applications (i.e., the Purple service in this diagram) that support posting.

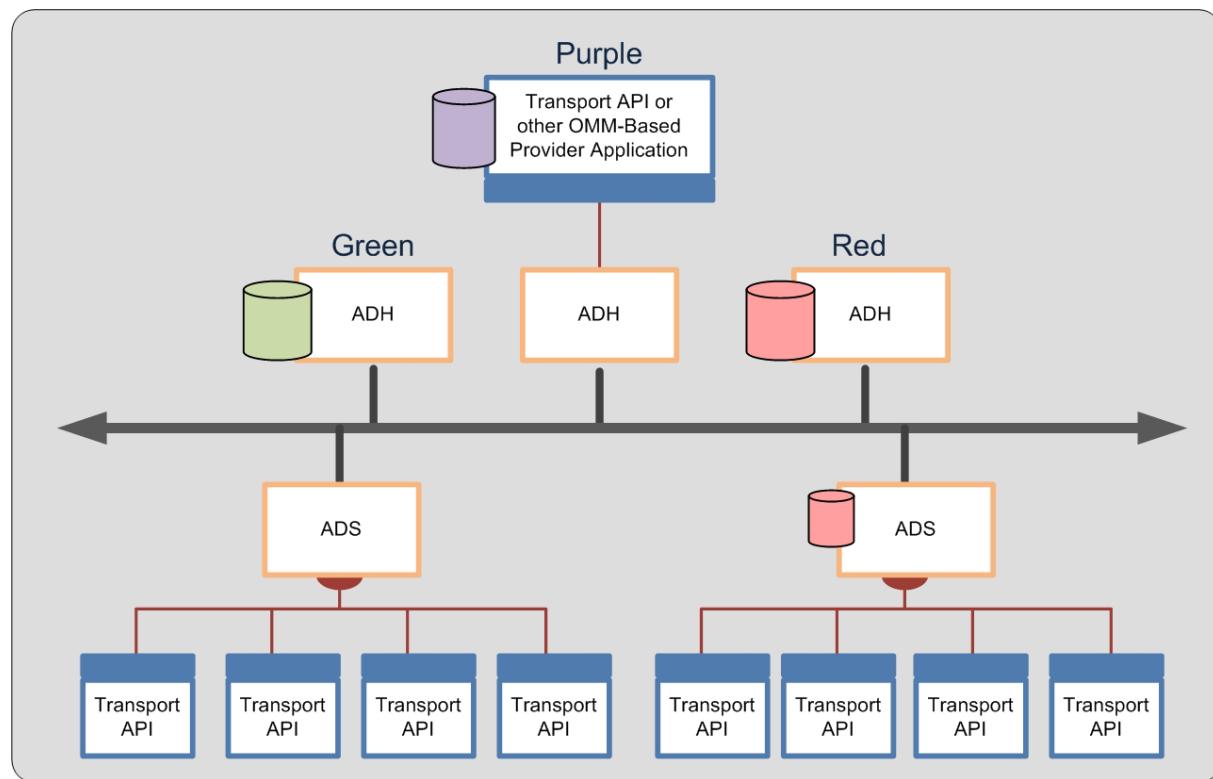


Figure 13. Posting into a Cache

You can use the Transport API to post into an ADH cache. If a cache exists in the ADS (the Red service), the ADS cache is also populated by responses from the ADH cache. If you configure TREP to allow such behavior, posts can be sent beyond

the ADH (to the Provider Application in the Purple service). Such posting flexibility is a good solution if one's applications are restricted to a LAN which hosts an ADS but allows publishing up the network to a cache with items to which other clients subscribe.

3.2.6.2 Contribution/Inserts

Posting also allows OMM-based contributions. Through such posting, clients can contribute data to a device on the head end or to a custom-provider. In the following example, the Transport API sends an OMM post to a provider application that supports such functionality.

While this diagram is similar to the example in Figure 13, the difference is that core components (such as the ADS/ADH) in TREP can convert a post into an SSL Insert for legacy connectivity. This functionality is provided for migration purposes.

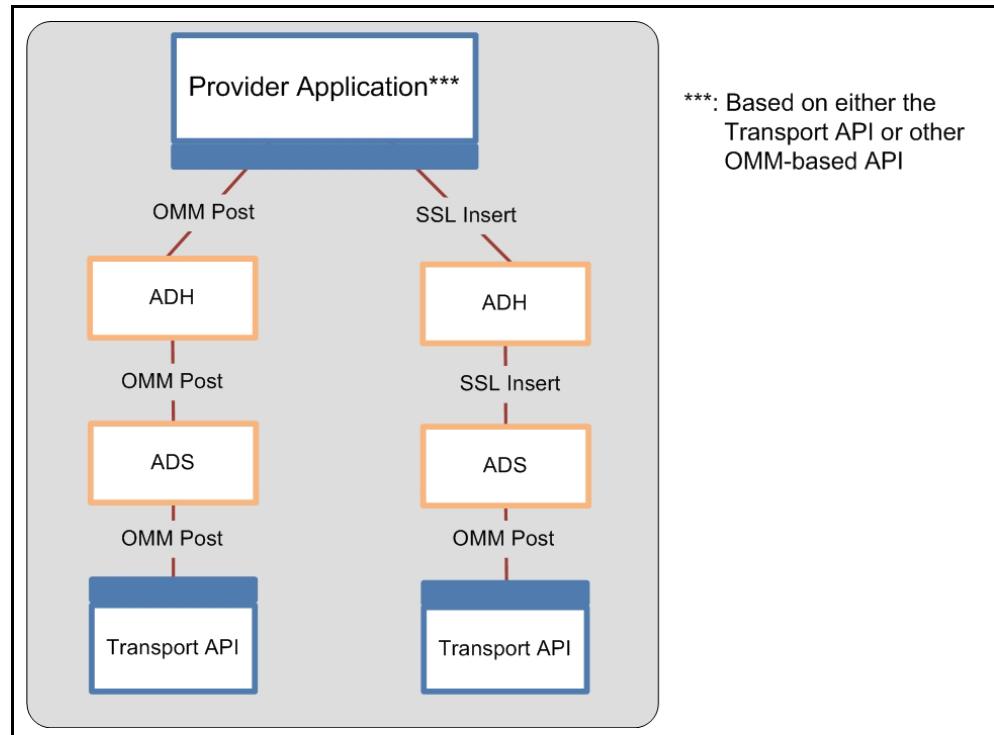


Figure 14. OMM Post with Legacy Inserts

3.2.7 Generic Message

Using a **Generic Message**, an application can send or receive a bi-directional message. A generic message can contain any OMM primitive type. Whereas the request/response type message flows from TREP to a consumer application, a generic message can flow in any direction, and a response is not required or expected. One advantage to using generic messages is its freedom from the traditional request/response data flow.

In a generic message scenario, the consumer sends a generic message to an ADS, while the ADS also publishes a generic message to the consumer application. All domains support this type of generic message behavior, not just market data-based domains (such as Market Price, etc). If a generic message is sent to a component that does not understand generic messages, the component ignores the message.

3.2.8 Private Streams

Using a **Private Stream**, a consumer application can create a virtual private connection with an interactive provider. This virtual private connection can be either a direct connection, through the TREP, or via a cascaded set of platforms. The following diagram illustrates these different configurations.

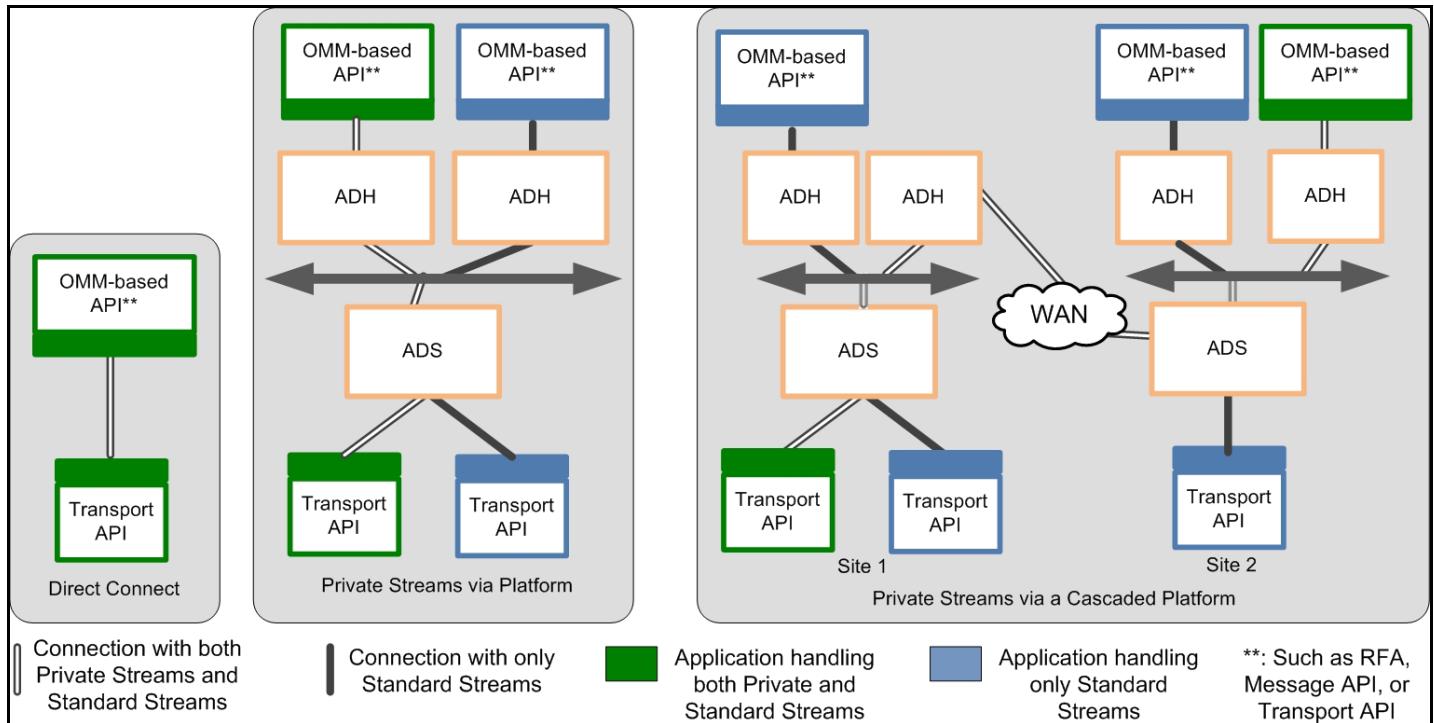


Figure 15. Private Stream Scenarios

A virtual private connection piggy backs on existing, individual point-to-point and multicast connections in the system (Figure 15 illustrates this behavior using a white connector). Messages exchanged via a Private Stream flow between a Consumer and an Interactive Provider using these existing underlying connections. However, unlike a regular stream, the Transport API or TREP components do not fan out these messages to other consumers or providers.

In Figure 15, each diagram shows a green consumer creating a private stream with a green provider. The private stream, using existing infrastructure and network connections, is illustrated as a white path in each of the diagrams. When established, communications sent on a private stream flow only between the green consumer and the green provider to which it connects. Blue providers and consumers do not see messages sent via the private stream.

Any break in a “virtual connection” causes the provider and consumer to be notified of the loss of connection. In such a scenario, the consumer is responsible for re-establishing the connection and re-requesting any data it might have missed from the provider. All types of requests, functionality, and Domain Models can flow across a private stream, including (but not limited to):

- Streaming Requests
- Snapshot Requests
- Posting
- Generic Messages
- Batch Requests
- Views
- All Thomson Reuters Domain Models & Custom Domain Models

3.3 Providers

Providers make their services available to consumers through TREP infrastructure components. Every provider-based application must attach to a provider access point to inter-operate with consumers. All provider access points are considered concrete and are implemented by an TREP infrastructure component (like the ADH).

Examples of providers include:

- A user who receives a subscription request from TREP.
- A user who publishes data into TREP, whether in response to a request or using a broadcast-publishing style.
- A user who receives post data from TREP. Providers can handle such concepts as receiving requests for contributions/inserts, or receiving publication requests.
- A user who sends and/or receives generic messages with TREP.

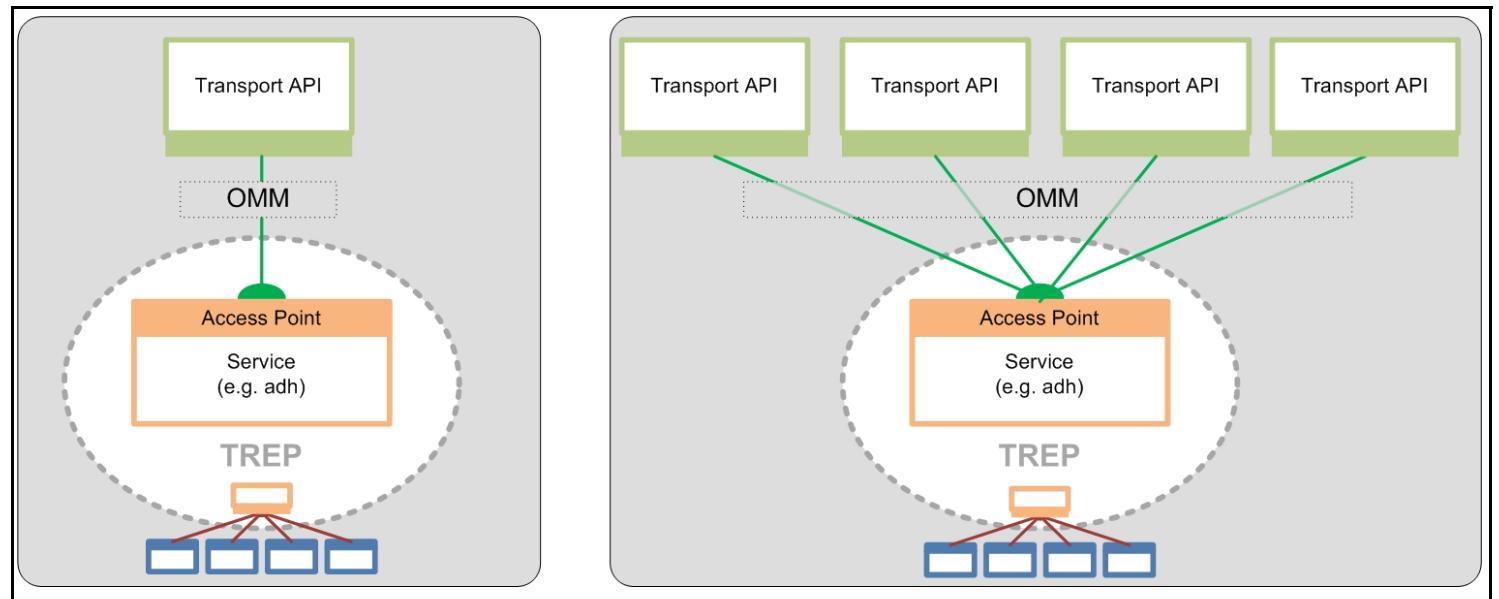


Figure 16. Provider Access Point

3.3.1 Interactive Providers

An **interactive provider** is one that communicates with the TREP, accepting and managing multiple connections with TREP components. The following diagram illustrates this concept.

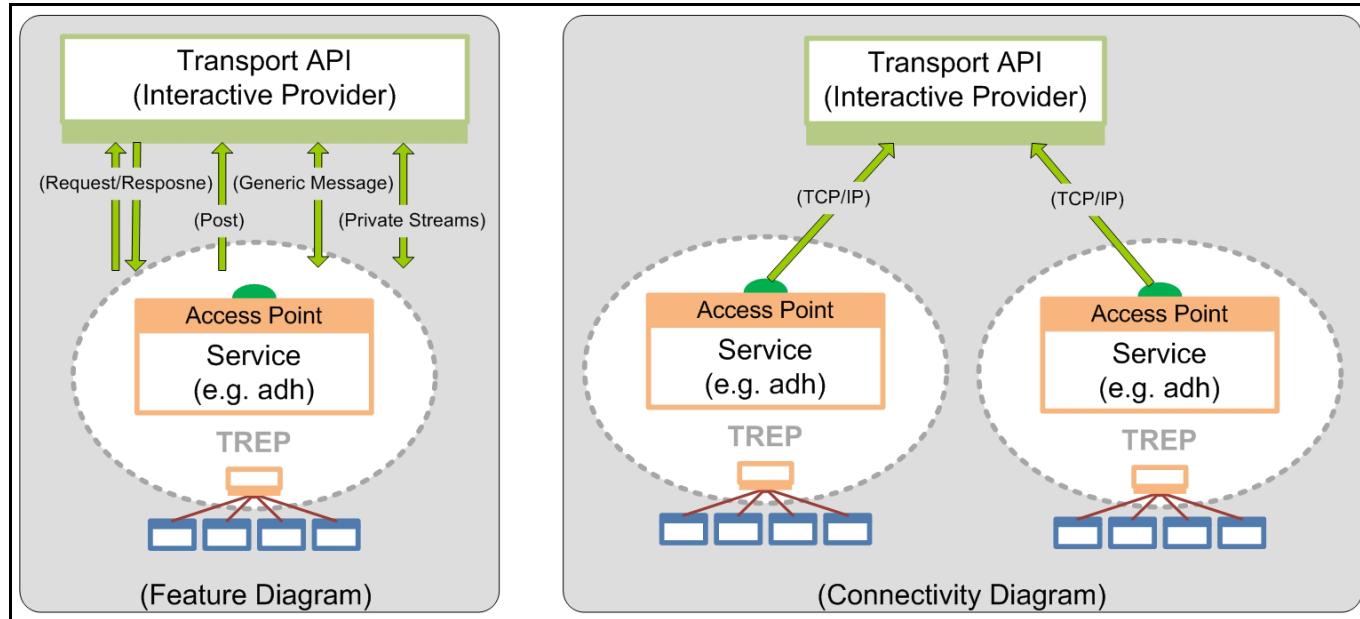


Figure 17. Interactive Providers

An interactive provider receives connection requests from the TREP. The Interactive Provider responds to requests for information as to what services, domains, and capabilities it can provide or for which it can receive requests. It may also receive and respond to requests for information about its data dictionary, describing the format of expected data types. After this is completed, its behavior is interactive.

For legacy Triarch users or early TREP adopters, the Interactive Provider is similar in concept to the legacy Sink-Driven Server or Managed Server Application. Interactive Providers act like servers in a client-server relationship. A Transport API interactive provider can accept and manage connections from multiple TREP components.

3.3.1.1 Request /Response

In a standard request/response scenario, the interactive provider receives requests from consumers on TREP (e.g., “Provide data for item TRI”). The consumer then expects the interactive provider to provide a response, status, and possible updates whenever the information changes. If the item cannot be provided by the interactive provider, the consumer expects the provider to reject the request by providing an appropriate response - commonly a status message with state and text information describing the reason. Request and response behavior is supported in all domains, not simply Market-Data-based domains.

Interactive providers can receive any consumer-style request described in the consumer section of this document, including batch requests, views, symbol lists, pause/resume, etc. Provider applications should respond with a negative acknowledgment or response if the interactive application cannot provide the expected response to a request.

3.3.1.2 Posts

The interactive provider can receive post messages via TREP. Post messages will state whether an acknowledgment is required. If required, TREP will expect the interactive provider to provide a response, in the form of a positive or negative acknowledgment. Post behavior is supported in all domains, not simply Market-Data-based domains. Whenever an interactive provider connects to TREP and publishes the supported domains, the provider states whether it supports post messages.

Further discussion on posting can be found in Section 13.9.

3.3.1.3 Generic Messages

Using generic messages, an application can send or receive bi-directional messages. Whereas a request/response type message flows from TREP to an interactive provider, generic messages can flow in any direction and do not expect a response. When using generic messages, the application need not conform to the request/response flow. A generic message can contain any OMM data type.

Interactive providers can receive a generic message from and publish a generic message to TREP.

Generic message behavior is supported in all domains, not simply Market-Data-based domains. If a generic message is sent to a component (e.g., a legacy application) which does not understand generic messages, the component ignores it.

Additional details on generic messages can be found in Section 12.2.6.

3.3.1.4 Private Streams

In a typical private stream scenario, the interactive provider can receive requests for a private stream. Once established, interactive providers can receive any consumer-style request via a private stream, described in the consumer section of this document, including Batch requests, Views, Symbol Lists, Pause/Resume, Posting, etc. Provider applications should respond with a negative acknowledgment or response if the interactive application cannot provide the expected response to a request.

3.3.1.5 Tunnel Streams (Available Only in ETA Reactor and EMA)

An interactive provider can receive requests for a tunnel stream when using the ETA Reactor or EMA. When creating a tunnel stream, the consumer indicates any additional behaviors to enforce, 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 enforce on the stream. After the stream is established, the consumer can exchange any content it wants, though the tunnel stream will enforce behaviors on the transmitted content as negotiated with the provider.

A tunnel stream allows for multiple substreams to exist, where substreams follow from the same general stream concept, except that they flow and coexist within the confines of a tunnel stream.

3.3.2 Non-Interactive Providers

A **non-interactive provider** (NIP) writes a provider application that connects to TREP and sends a specific set of non-interactive data (services, domains, and capabilities).

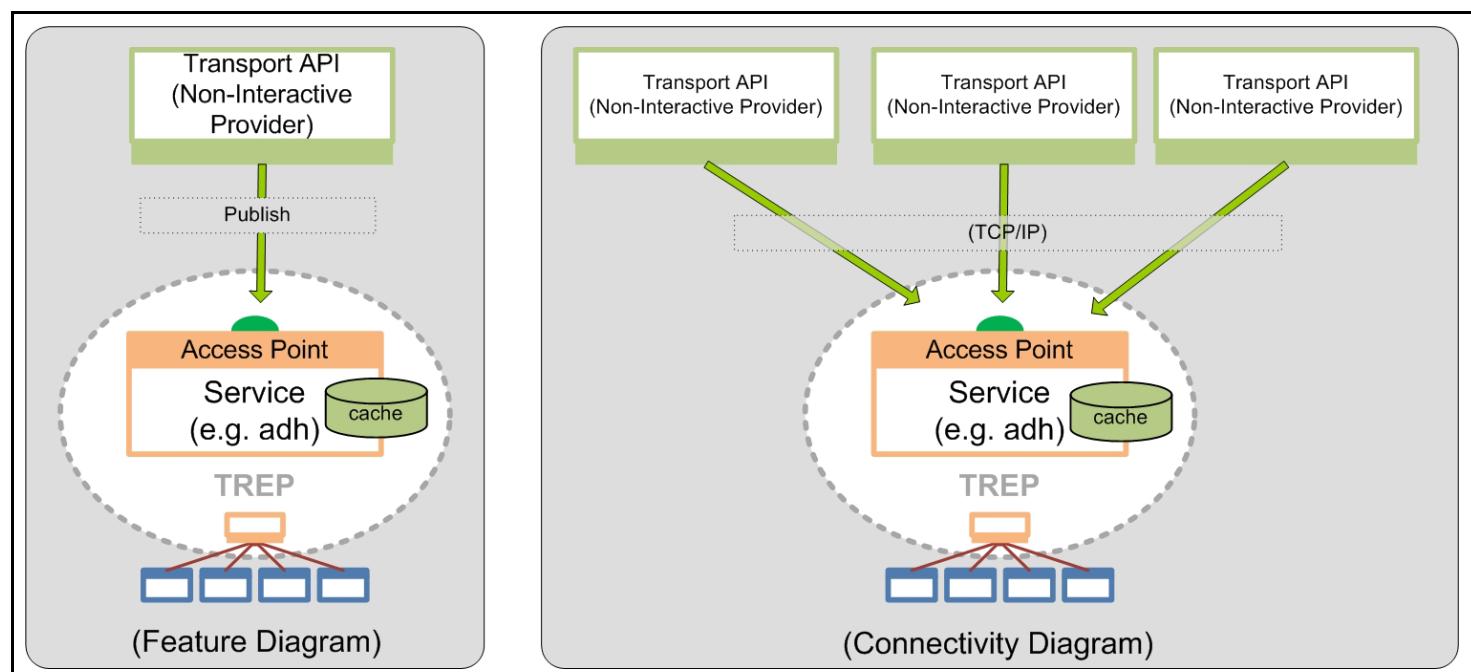
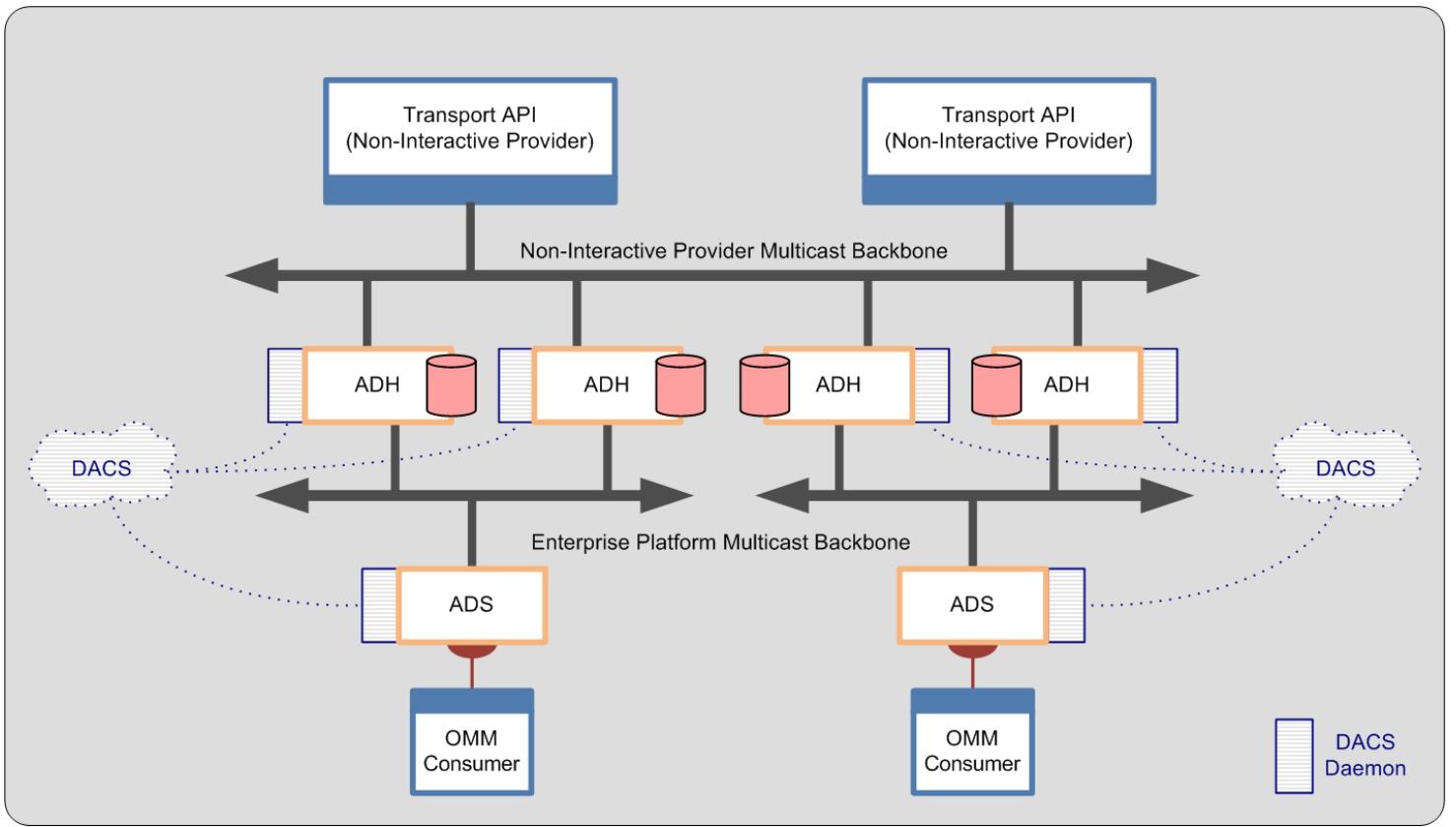


Figure 18. NIP: Point-To-Point**Figure 19. NIP: Multicast**

After a NIP connects to TREP, the NIP can start sending information for any supported item and domain. For legacy Triarch users or early TREP adopters, the NIP is similar in concept to what was once called the Src-Driven, or Broadcast Server Application.

Non-interactive providers act like clients in a client-server relationship. Multiple NIPs can connect to the same TREP and publish the same items and content. For example, two NIPs can publish the same or different fields for the same item "INTC.O" to the same TREP.

NIP applications can connect using a point-to-point TCP-based transport as shown in Figure 18, or using a multicast transport as shown in Figure 19.

The main benefit of this scenario is that all publishing traffic flows from top to bottom: the way a system normally expects updating data to flow. In the local publishing scenario, posting is frequently done upstream and must contend with a potential Infrastructure bias in prioritization of upstream versus downstream traffic.

Chapter 4 System View

4.1 System Architecture Overview

A TREP network typically hosts the following:

- Core Infrastructure (i.e., ADS, ADH, etc.)
- Consumer applications that typically request and receive information from the network
- Provider applications that typically write information to the network. Provider applications fall into one of two categories:
 - Interactive provider applications which receive and interpret request messages and reply back with any needed information.
 - NIP applications which publish data, regardless of user requests or which applications consume the data.
- Permissioning infrastructure (i.e., DACS)
- Devices which interact with the markets (i.e., Data Feed Direct and the Elektron Edge Device)

The following figure illustrates a typical deployment of a TREP network and some of its possible components. Components that use the Transport API could alternatively choose to leverage RFA, depending on user needs and required access levels. The remainder of this chapter briefly describes the components pictured in the diagram and explains how the Transport API integrates with each.

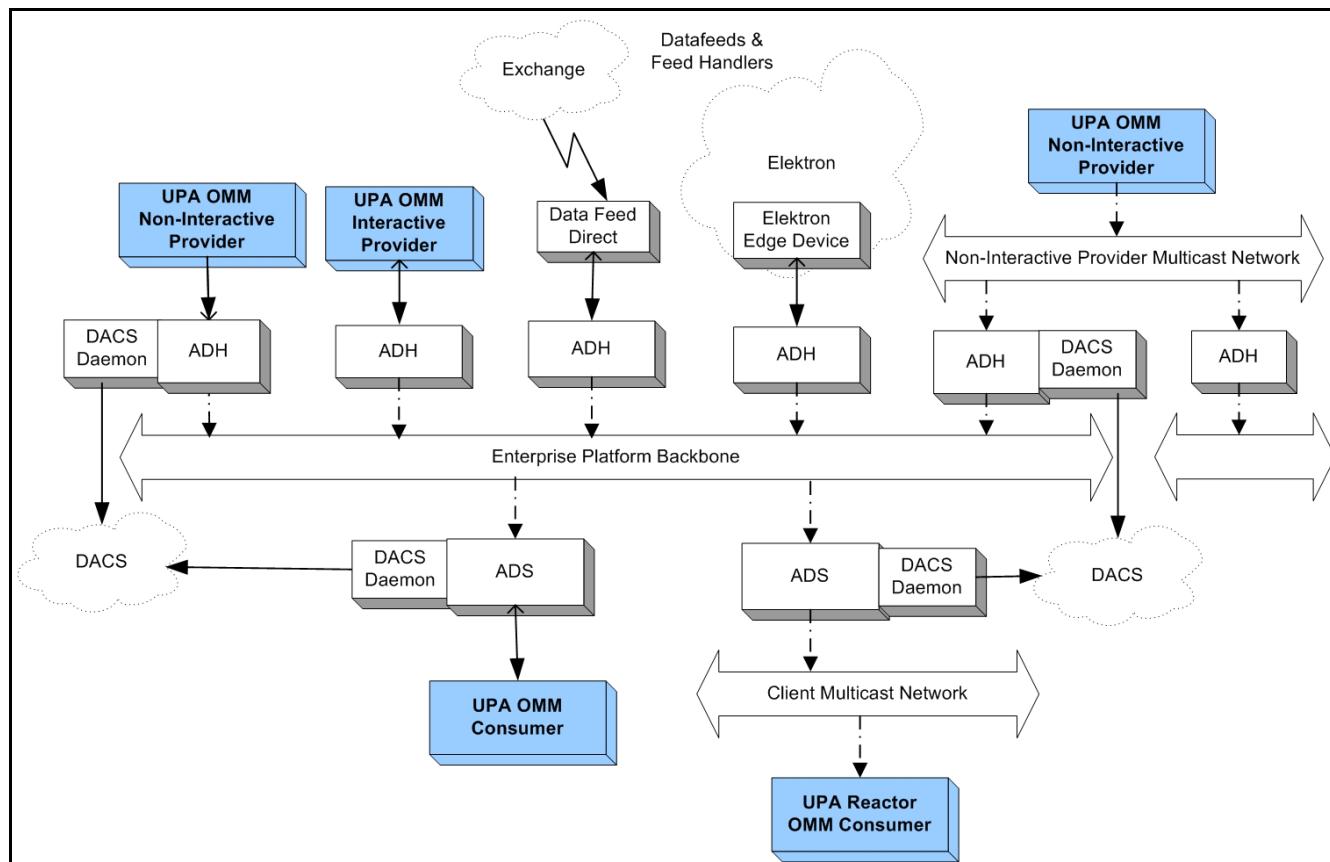


Figure 20. Typical TREP Components

4.2 Advanced Distribution Server (ADS)

The ADS provides a consolidated distribution solution for Thomson Reuters, value-added, and third-party data for trading-room systems. It distributes information using the same OMM and RWF protocols exposed by the Transport API.

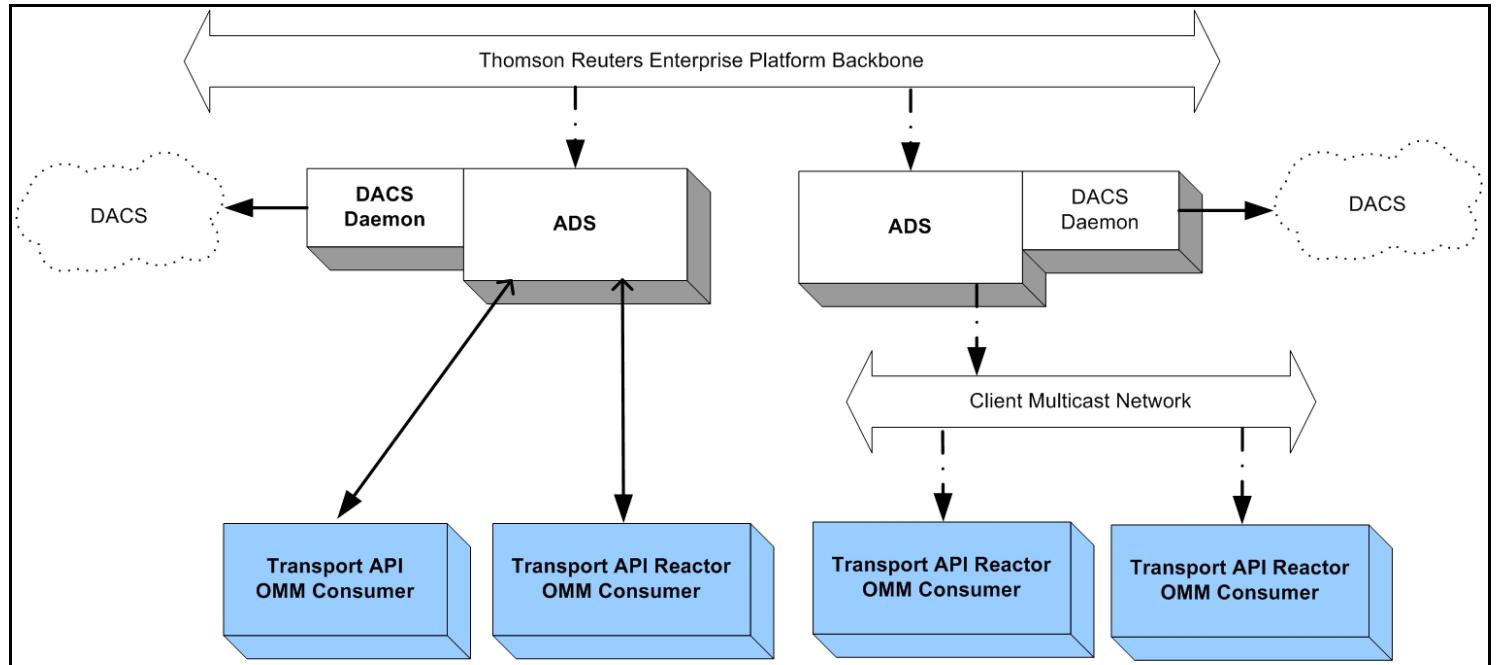


Figure 21. Transport API and Advanced Distribution Server

As a distribution device for market data, the ADS delivers data from the Advanced Data Hub (ADH). Because the ADS leverages multiple threads, it can offload the encoding, fan out, and writing of client data. By distributing its tasks in this fashion, ADS can support far more client applications than could any previous Thomson Reuters distribution solution.

The ADS supports two types of data delivery when communicating with API clients:

- Via point-to-point communication.
- Via multicast communication.

To take advantage of multicast communications, consumers must use a Value-Add component. For further information:

- On the Transport API Reactor component, refer to the *Transport API C Edition Value Added Components Developers Guide*.
- On network topologies as they relate to the Transport API, refer to Section 10.3.1.

4.3 Advanced Data Hub (ADH)

The **ADH** is a networked, data distribution server that runs in the TREP. It consumes data from a variety of content providers and reliably fans this data out to multiple ADSs over a backbone network (using either multicast or broadcast technology). Transport API-based non-interactive or interactive provider applications can publish content directly into an ADH, thus distributing data more widely across the network. NIP applications can publish content to an ADH via TCP or multicast connection types.

The ADH leverages multiple threads, both for inbound traffic processing and outbound data fanout. By leveraging multiple threads, ADH can offload the overhead associated with request and response processing, caching, data conflation, and fault tolerance management. By offloading overhead in such a fashion, the ADH can support higher throughputs than could previous Thomson Reuters data hub solutions.

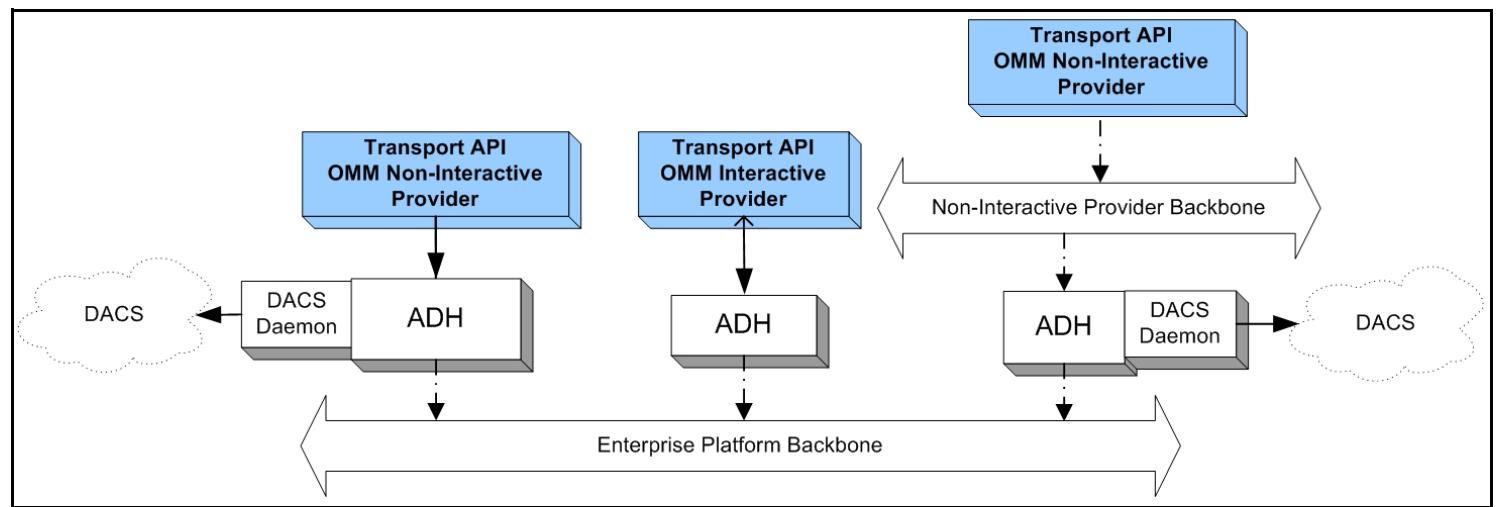


Figure 22. Transport API and the Advanced Data Hub

4.4 Elektron

Elektron is an open, global, ultra-high-speed network and hosting environment, which allows users to access and share various types of content. Elektron allows access to information from a wide network of content providers, including exchanges, where all exchange data is normalized using the OMM.

The Elektron Edge Device, based on ADS technology, is the access point for consuming this data. To access this content, a Transport API consumer application can connect directly to the Edge Device or via a cascaded Enterprise Platform architecture (as illustrated in the following diagram).

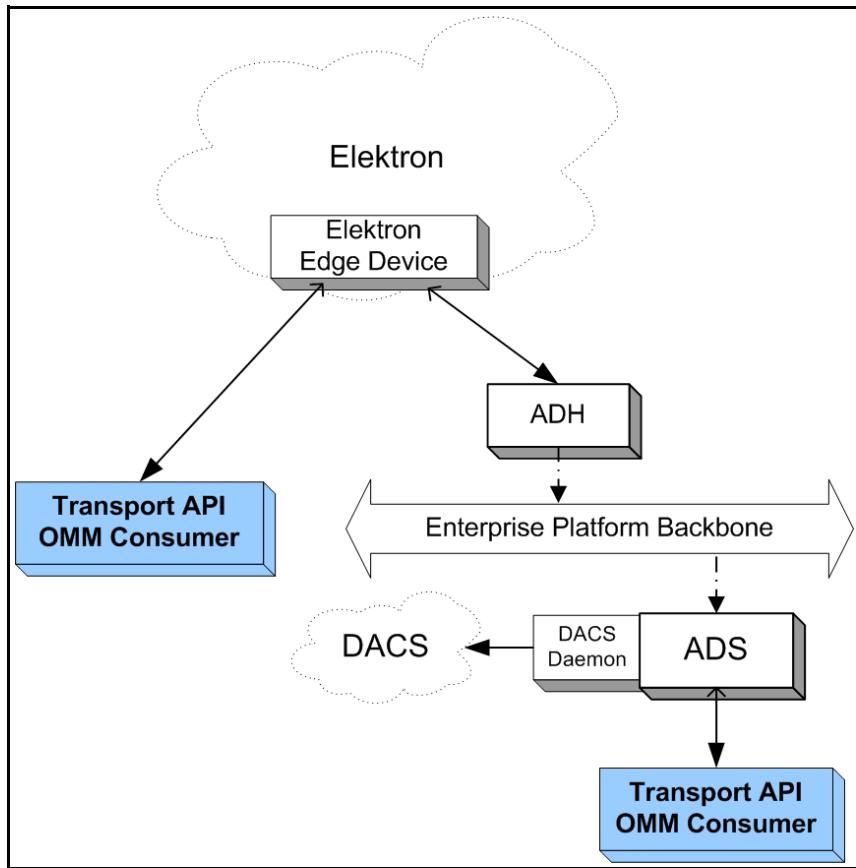


Figure 23. Transport API and Elektron

4.5 Data Feed Direct

Thomson Reuters Data Feed Direct is a fully managed Thomson Reuters exchange feed providing an ultra-low-latency solution for consuming data from specific exchanges. The Data Feed Direct normalizes all exchange data using the OMM.

To access this content, a Transport API consumer application can connect directly to the Data Feed Direct or via a cascaded TREP architecture.

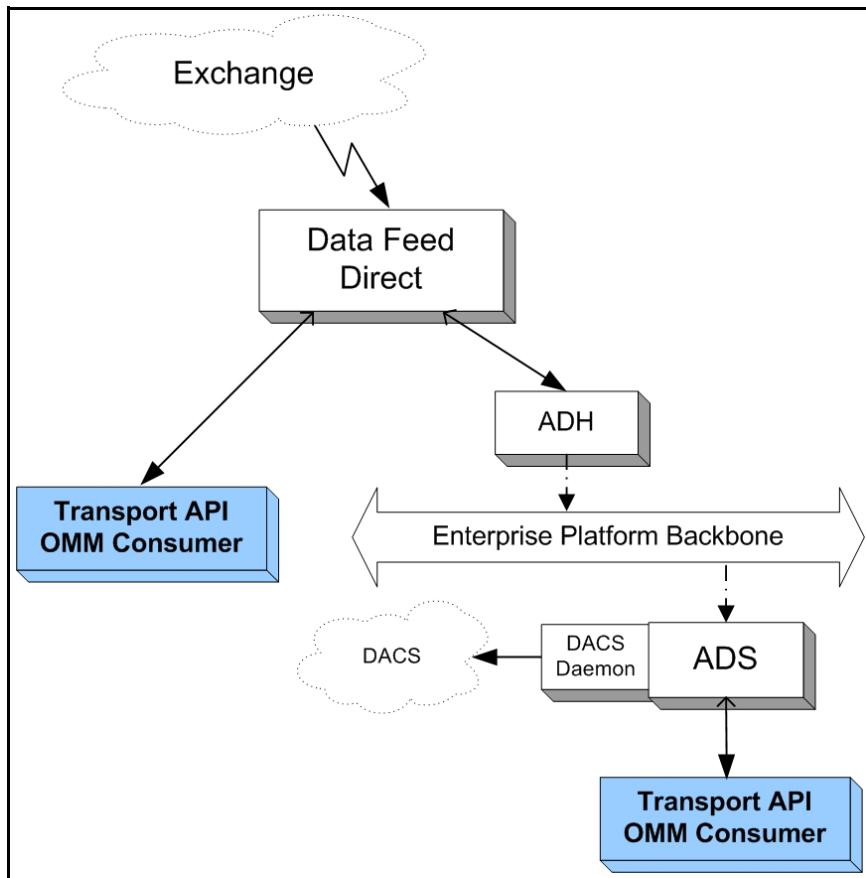


Figure 24. Transport API and Data Feed Direct

4.6 Internet Connectivity via HTTP and HTTPS

OMM consumer and Provider applications can use the Transport API to establish connections by tunneling through the Internet.

- OMM consumer and NIP applications can establish connections via HTTP tunneling.
- ADS and OMM Interactive Provider applications can accept incoming Transport API connections tunneled via HTTP (such functionality is available across all supported platforms).
- Consumer applications can leverage HTTPS to establish an encrypted tunnel to certain Thomson Reuters Hosted Solutions, performing key and certificate exchange.
- Consumer-side functionality leverages Microsoft WinINET. Users can configure certificate and proxy use via Internet Explorer. Because of its dependency on the Microsoft WinINET library, consumer HTTP and HTTPS tunneling are available only on supported Windows platforms.

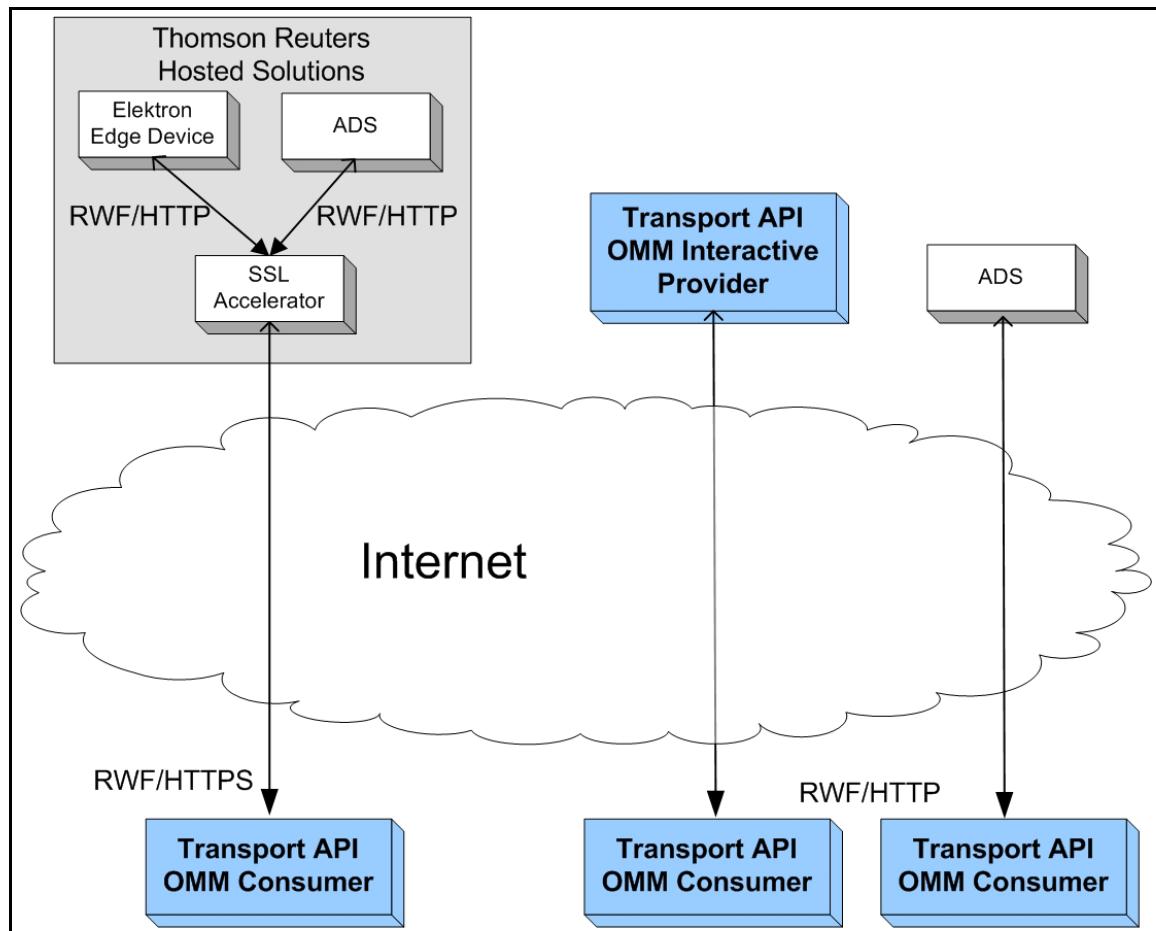


Figure 25. Transport API and Internet Connectivity

4.7 Direct Connect

The Transport API allows OMM Interactive Provider applications and OMM consumer applications to directly connect to one another. This includes OMM applications written to RFA. The following diagram illustrates various direct connect combinations.

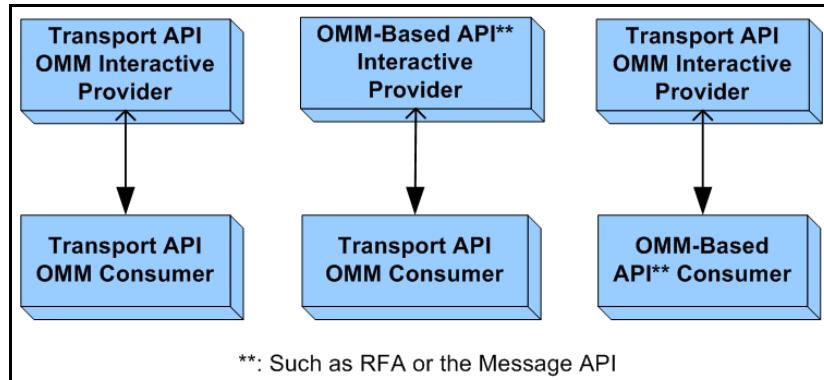


Figure 26. Transport API and Direct Connect

Chapter 5 Model and Package Overviews

5.1 Transport API Models

5.1.1 Open Message Model (OMM)

The **Open Message Model (OMM)** is a collection of message header and data constructs. Some OMM message header constructs (such as the Update message) have implicit market logic associated with them, while others (such as the Generic message) allow for free-flowing bi-directional messaging. You can combine OMM data constructs in various ways to model data ranging from simple (i.e., flat) primitive types to complex multi-level hierachal data.

The layout and interpretation of any specific OMM model (also referred to as a domain model) is described within that model's definition and is not coupled with the API. The OMM is a flexible and simple tool that provides the building blocks to design and produce domain models to meet the needs of the system and its users. The Transport API provides structural representations of OMM constructs and manages the RWF binary-encoded representation of the OMM. Users can leverage Thomson Reuters-provided OMM constructs to consume or provide OMM data throughout the Enterprise Platform.

5.1.2 Reuters Wire Format (RWF)

RWF is the encoded representation of the OMM; a highly-optimized, binary format designed to reduce the cost of data distribution compared to previous wire formats. Binary encoding represents data in the machine's native manner, enabling further use in calculations or data manipulations. RWF allows for serializing OMM message and data constructs in an efficient manner while still allowing you to model rich content types. You can use RWF to distribute field identifier-value pair data (similar to Marketfeed), self-describing data (similar to Qform), as well as more complex, nested hierachal content.

5.1.3 Domain Message Model

A Domain Message Model (DMM) describes a specific arrangement of OMM message and data constructs. A DMM defines any:

- Specialized behavior associated with the domain
- Specific meanings or semantics associated with the message data

Unless a DMM specifies otherwise, any implicit market logic associated with a message still applies (e.g., an Update message indicates that previously received data is being modified by corresponding data from the Update message).

5.1.3.1 Reuters Domain Model

A **Reuters Domain Model (RDM)** is a domain message model typically provided or consumed by a Thomson Reuters product (i.e., the Enterprise Platform, Data Feed Direct, or Elektron). Some currently-defined RDMS allow for authenticating to a provider (e.g., Login), exchanging field or enumeration dictionaries (e.g., Dictionary), and providing or consuming various types of market data (e.g., Market Price, Market by Order, Market by Price). Thomson Reuters's defined models have a domain value of less than 128. For extended definitions of the currently-defined Reuters Domain Models, refer to the *Transport API RDM Usage Guide*.

5.1.3.2 User-Defined Domain Model

A **User-Defined Domain Model** is a DMM defined by a third party. These might be defined to solve a need specific to a user or system in a particular deployment and which is not resolved through the use of an RDM. Any user-defined model must use a domain value between 128 and 255.

Customers can have their domain model designer work with Thomson Reuters to define their model as a standard RDM. Working directly with Thomson Reuters can help ensure interoperability with future RDM definitions and with other Thomson Reuters products.

5.2 Packages

The Transport API consists of several packages, each serving a different purpose within an application. While some packages are interdependent, others can be used alone or with other packages. Each package serves a distinct purpose as described in the following sections.

As needs evolve, additional packages can be added to the Transport API.

5.2.1 Transport Package

The **Transport Package** provides a mechanism to efficiently distribute messages across a variety of communication protocols. This package provides a receiver-transparent way for senders to combine or pack multiple messages into one outbound packet, and it will internally fragment and reassemble messages which exceed the size of an outbound packet. This package exposes structural representations to manage connection properties and information. The Transport Package includes interface functions that assist with establishing connections and the sending or receiving of data. This package utilizes some header files from the Data Package, but has no other dependencies other than system libraries.

To access all transport functionality, an application must include `rssITransport.h`.

The Transport Package is described in more detail in Chapter 10.

5.2.2 Data Package

The **Data Package** defines primitive and container types, which make up components representing OMM data. Primitive types are simple, atomically updating constructs, usually provided by the operating system (e.g., Integer, Date). Container types can model more complex data and be modified more granularly than a primitive type (e.g., field identifier-value pairs, key-value pairs, self-describing name-value pairs). This package exposes typedef and structural representations of Transport API primitive and container types and manages their RWF binary representation. The Data Package provides interface functions for encoding and decoding data, along with additional helper utility functionality. The Data Package is described with more detail in Chapter 11. This package requires no outside dependencies.

To access all data package functionality, an application must include `rssIDataPackage.h`.

5.2.3 Message Package

The **Message Package** defines message header elements that flow between various applications in the TREP (e.g., Update messages). Some of these header elements are standard to the market data environment, (such as conflation information, state information, permission information, and item key elements used for stream identification). Message headers also contain generic attributes in which usage and meaning are defined within specific DMMs (e.g., Market Price, Market By Order). All messages can carry payload information of varying format and layouts. This package exposes structural representations of the UPA message classes and manages the RWF binary-encoded representation of these messages. The Message Package provides interface functions for encoding and decoding messages, along with additional helper utility functionality. The Message Package is described with more detail in Chapter 12. This package depends on the Data Package.

To access all message package functionality, an application must include `rssIMessagePackage.h`.

Chapter 6 Building an OMM Consumer

6.1 Overview

This chapter provides an overview of how to create an OMM consumer application. An OMM consumer application can establish a connection to other OMM interactive provider applications, including the TREP, Data Feed Direct, and Elektron. After connecting successfully, 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:

- Establish network communication
- Log in
- Obtain source directory information
- Load or download all necessary dictionary information
- Issue requests, process responses, and/or post information
- Log out and shut down

The **rsslConsumer** example application, included with the Transport API products, provides an example implementation of an OMM consumer application. The application is written with simplicity in mind and demonstrates the uses of the Transport API. 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.

6.2 Establish Network Communication

The first step of any Transport API consumer application is to establish a network connection with its peer component (i.e., another application with which to interact). An OMM consumer typically creates an outbound connection to the well-known hostname and port of an Interactive Provider. The consumer uses the **rsslConnect** function to initiate the connection and then performs any additional connection initialization processes as described in this document.

After the consumer's connection is active, ping messages might need to be exchanged. The negotiated ping timeout is available via the **RsslChannel**. The connection can be terminated if ping heartbeats are not sent or received within the expected time frame. Thomson Reuters recommends sending ping messages at intervals one third the size of the ping timeout.

Detailed information and use case examples for using RSSL Transport are provided in Chapter 10, Transport Package Detailed View .

6.3 Perform Login Process

Applications authenticate with one another using the Login domain model. An OMM consumer must register with the system using a Login request prior to issuing any other requests or opening any other streams.

After receiving a Login request, an interactive provider determines whether a user is permissioned to access the system. The interactive provider sends back a Login response, indicating to the consumer whether access is granted.

- If the application is denied, the Login stream is closed, and the consumer application cannot send additional requests.
- If the application is granted access, the Login response contains information about available features, such as Posting, Pause and Resume, and the use of Dynamic Views. The consumer application can use this information to tailor its interaction with the provider.

Content is encoded and decoded using the Message Package (described in Chapter 12, Message Package Detailed View) and the Data Package (described in Chapter 11, Data Package Detailed View). Further information about Login domain usage and messaging is available in the *Transport API RDM Usage Guide*.

6.4 Obtain 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 quality of service (QoS), and any item group information associated with the service. At minimum, Thomson Reuters recommends that the application requests the Info, State, and Group filters for the Source Directory.

- The Source Directory Info filter contains the service name and **serviceId** information for all available services. When the OMM consumer discovers an appropriate service, it uses the service's **serviceId** on all subsequent requests to that service.
- The Source Directory State filter contains status information for service, which informs the consumer whether the service is Up and available, or Down and unavailable.
- The Source Directory Group filter conveys item group status information, including information about group states, as well as the merging of groups. Additional information on item groups is available in Section 13.4.

Content is encoded and decoded using the Transport API's Message Package (as described in Chapter 12, Message Package Detailed View) and Data Package (as described in Chapter 11, Data Package Detailed View). Information about the Source Directory domain and its associated filter entry content is available in the *Transport API RDM Usage Guide*.

6.5 Load or Download Necessary Dictionary Information

Some data requires the use of a dictionary for encoding or decoding. This dictionary typically defines type and formatting information and directs the application as to how to encode or decode specific pieces of information. Content that uses the **Rss1FieldList** type requires the use of a field dictionary (usually the Thomson Reuters **RDMFieldDictionary**, though it could also be a user-defined or modified field dictionary).

A source directory message should provide information about:

- Any dictionaries required to decode the content provided on a service.
- Which dictionaries are available for download.

A consumer application can determine whether to load necessary dictionary information from a local file or download the information from the provider if available.

- If loading from a file, the Transport API offers several utility functions to load and manage a properly-formatted field dictionary.
- If downloading information, the application issues a request using the Dictionary domain model. The provider application should respond with a dictionary response. Because a dictionary response often contains a large amount of content, it is typically broken into a multi-part message. the Transport API offers several utility functions for encoding and decoding of the Dictionary domain content.

For information on the utility functions used in both instances and for information about the Dictionary domain and its expected content formats, refer to the *Transport API RDM Usage Guide*.

Content is encoded and decoded using the Transport API Message Package (as described in Chapter 12, Message Package Detailed View) and the Transport API Data Package (as described in Chapter 11, Data Package Detailed View).

6.6 Issue Requests and/or Post Information

After the consumer application successfully logs in and obtains Source Directory and Dictionary information, it can 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 RDM Usage Guide*.

At this point, an OMM consumer application can also post information to capable provider applications. For more information, refer to Section 13.9.

Content is encoded and decoded using the Transport API Message Package (as described in Chapter 12, Message Package Detailed View) and Data Package (as described in Chapter 11, Data Package Detailed View).

6.7 Log Out and Shut Down

When the consumer application is done retrieving or posting content, it should close all open streams and shut down the network connection. Issuing an **Rss1CloseMsg** for the **streamId** associated with the Login closes all streams opened by the consumer.

- For more information on closing streams, refer to Section 12.2.5.
- For information on the Message Package, refer to Chapter 12, Message Package Detailed View.

When shutting down the consumer, the application should release any unwritten pool buffers. Calling **rss1CloseChannel** terminates the connection to the provider application. Detailed information and transport code examples are provided in Chapter 10, Transport Package Detailed View.

6.8 Additional Consumer Details

The following locations provide specific details about using OMM consumers and the Transport API:

- The **rssIConsumer** application demonstrates one way of implementing of an OMM consumer application. The application's source code and **ReadMe** file contain additional information about specific implementation and behaviors.
- For reviewing high-level encoding and decoding concepts, refer to Chapter 9, Encoding and Decoding Conventions.
- For a detailed look at the Data Package, typically used for encoding and decoding payload content, refer to Chapter 11, Data Package Detailed View.
- For a detailed look at the Message Package, used for all message encoding and decoding, refer to Chapter 12, Message Package Detailed View.
- For a detailed look at the Transport Package, used for the application's network communication, refer to Chapter 10, Transport Package Detailed View.
- For specific information about the DMMs required by this application type, refer to the *Transport API RDM Usage Guide*.

Chapter 7 Building an OMM Interactive Provider

7.1 Overview

This chapter provides a high-level description of how to create an OMM interactive provider application. An OMM interactive provider application opens a listening socket on a well-known port allowing OMM consumer applications to connect. After connecting, consumers can request data from the interactive provider.

The following steps summarize this process:

- Establish network communication
- Accept incoming connections
- Handle login requests
- Provide source directory information
- Provide or download necessary dictionaries
- Handle requests and post messages
- Disconnect consumers and shut down

The **rsslProvider** example application included with the Transport API package provides one way of implementing an OMM interactive provider. The application is written with simplicity in mind and demonstrates the use of the Transport API. 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.

7.2 Establish Network Communication

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.

Whenever an OMM consumer application attempts to connect, the provider uses the **rsslAccept** function to begin the connection initialization process.

Once the connection is active, the consumer and provider applications might need to exchange ping messages. A negotiated ping timeout is available via **Rss1Channel** corresponding to each connection (this value might differ on a per-connection basis). The provider may choose to terminate a connection if ping heartbeats are not sent or received within the expected time frame. Thomson Reuters recommends sending ping messages at intervals one-third the size of the ping timeout.

For detailed information and use cases for the RSSL Transport, refer to Chapter 10, Transport Package Detailed View.

7.3 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.

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 **Rss1RefreshMsg** 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 **Rss1StatusMsg**, closing the connection and informing the user of the reason for denial.

Content is encoded and decoded using the Transport API Message Package (as described in Chapter 12, Message Package Detailed View) and the Transport API Data Package (as described in Chapter 11, Data Package Detailed View). For further information on Login domain usage and messaging, refer to the *Transport API RDM Usage Guide*.

7.4 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 Section 13.4.

Content is encoded and decoded using the Transport API's Message Package (as described in Chapter 12, Message Package Detailed View) and Data Package (as described in Chapter 11, Data Package Detailed View). For details on the Source Directory domain and all of its associated filter entry content, refer to the *Transport API RDM Usage Guide*.

7.5 Provide or Download 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 **Rss1FieldList** type requires the use of a field dictionary (usually the Thomson Reuters **RDMFieldDictionary**, though it can instead be user-defined or a 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 may also download the RWFFId and RWFEnum dictionaries to retrieve the appropriate dictionary information for providing field list content. A provider can use this feature to ensure it has the appropriate version of the dictionary or to encode data. The ADH supporting the Provider Dictionary Download feature sends a

Login request 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 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 dictionary.

Content is encoded and decoded using the Transport API Message Package (as described in Chapter 12, Message Package Detailed View) and the Transport API Data Package (as described in Chapter 11, Data Package Detailed View).

Information about the Login and Dictionary domains, their expected content and formatting, and dictionary utility functions, is available in the *Transport API RDM Usage Guide*.

7.6 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 received against the content available from the provider
- 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 `Rss1StatusMsg` to indicate the reason and close the stream. All requests and responses should follow specific formatting as defined in the domain model specification. For details on all domains provided by Thomson Reuters, refer to the *Transport API RDM Usage Guide*.

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 Section 13.9.

Content is typically encoded and decoded using the Transport API's Message Package (as described in Chapter 12, Message Package Detailed View) and Data Package (as described in Chapter 11, Data Package Detailed View).

7.7 Disconnect Consumers and Shut Down

When shutting down, the provider application should close the listening socket by calling `rss1CloseServer`. Closing the listening socket prevents new connection attempts. The provider application can either leave consumer connections intact or shut them down.

If the provider decides to close consumer connections, the provider should send an `Rss1StatusMsg` on each connection's login stream, thus closing the stream. At this point, the consumer should assume that its other open streams are also closed. The provider should then release any unwritten pool buffers it has obtained from `rss1GetBuffer` and call `rss1CloseChannel` for each connected client.

For detailed information and use case examples for the transport, refer to Chapter 10, Transport Package Detailed View.

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.

7.8 Additional Interactive Provider Details

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

- The **rssIPublisher** application demonstrates one implementation of an OMM interactive provider application. The application's source code and **ReadMe** file have additional information about specific implementation and behaviors.
- To review high-level encoding and decoding concepts, refer to Chapter 9, Encoding and Decoding Conventions.
- For a detailed look at the Data Package, typically used for encoding and decoding payload content, refer to Chapter 11, Data Package Detailed View.
- For a detailed look at the Message Package, used for all message encoding and decoding, refer to Chapter 12, Message Package Detailed View.
- For a detailed look at the Transport Package, used for the application's network communication, refer to Chapter 10, Transport Package Detailed View.
- For specific information about DMMs required by this application type, refer to the *Transport API C Edition RDM Usage Guide*.

Chapter 8 Building an OMM NIP

8.1 Overview

This chapter provides an outline of how to create an OMM NIP application which can establish a connection to an ADH server. Once connected, an OMM NIP 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:

- Establish network communication
- Perform Login process
- Perform Dictionary Download
- Provide Source Directory information
- Provide content
- Log out and shut down

Included with the Transport API package, the **rsslNIPProvider** example application provides an implementation of an NIP written with simplicity in mind and demonstrates the use of the Transport API. Portions of the functionality are abstracted for easy reuse, though you might need to modify it to achieve your own performance and functionality goals.

Content is encoded and decoded using the Transport API Message Package (as described in Chapter 12, Message Package Detailed View) and the Transport API Data Package (as described in Chapter 11, Data Package Detailed View).

8.2 Establish Network Communication

The first step of any NIP application is to establish network communication with an ADH server. To do so, the OMM NIP typically creates an outbound connection to the well-known hostname and port of an ADH. The NIP application uses the **rsslConnect** function to initiate the connection process and then performs connection initialization processes as described in this document.

After establishing a connection, ping messages might need to be exchanged. The negotiated ping timeout is available via the **RsslChannel**. If ping heartbeats are not sent or received within the expected time frame, the connection can be terminated. Thomson Reuters recommends sending ping messages at intervals one-third the size of the ping timeout.

For detailed information on RSSL Transport and associated use case examples, refer to Chapter 10, Transport Package Detailed View.

8.3 Perform Login Process

Applications authenticate with one another using the Login domain model. An OMM NIP must register with the system using a Login request¹ prior to providing any content.

After receiving a Login request, the ADH determines whether the NIP is permissioned to access the system. The ADH sends a Login response to the NIP which indicates whether the ADH has granted it access. If the application is denied, the ADH closes the Login stream and the NIP application cannot perform any additional communication. If the application gains access to the ADH, the Login response informs the application of this. The provider must now provide a Source Directory and/or download Dictionary.

For details on using the Login domain and expected message content, refer to the *Transport API RDM Usage Guide*.

8.4 Perform Dictionary Download

If connected to an ADH that support dictionary downloads, an OMM NIP can download the RWFFId and RWFEnum dictionaries to retrieve appropriate information when providing field list content. An OMM NIP can use this feature to ensure they are using the correct version of the dictionary or to encode data. The ADH supporting the Provider Dictionary Download feature sends a Login response message containing the **SupportProviderDictionaryDownload** login element. The dictionary request is send using the Dictionary domain model².

The Transport API offers several utility functions you can use to download and manage a properly-formatted field dictionary. The API also includes other utility functions that help the provider encode into an appropriate format for downloading or decoding a downloaded dictionary.

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

8.5 Provide Source Directory Information

The Source Directory domain model conveys information about all available services in the system. After completing the Login process, an OMM NIP must provide a Source Directory refresh³ indicating:

- Service, service state, QoS, and capability information associated with the NIP
- Supported domain types and any item group information associated with the service.

At a minimum, Thomson Reuters recommends that the NIP send the Info, State, and Group filters for the Source Directory.

- The Source Directory Info filter contains service name and **serviceId** information for all available services, though NIPs typically provide data on only one service.
- The Source Directory State filter contains status information for service. This informs the ADH whether the service is Up and available or Down and unavailable.
- The Source Directory Group filter conveys item group status information, including information about group states as well as the merging of groups. For additional information about item groups, refer to Section 13.4.

For details on the Source Directory domain and all of its associated filter entry content, refer to the *Transport API RDM Usage Guide*.

1. Because this is done in an interactive manner, the NIP should assign a **streamId** with a positive value (which the ADH will reference) when sending its response.

2. Because this is instantiated by the provider, the application should use a **streamId** with a negative value.

3. Because this is instantiated by the provider, the NIP should use a **streamId** with a negative value.

8.6 Provide Content

After providing a Source Directory, the NIP application can begin pushing content to the ADH. Each unique information stream should begin with an `RsslRefreshMsg`, conveying all necessary identification information for the content⁴. The initial identifying refresh can be followed by other status or update messages. Some ADH functionality, such as cache rebuilding, may require that NIP applications publish the message key on all `RsslRefreshMsgs`. For more information, refer to component-specific documentation.

Note: Some components, depending on their specific functionality and configuration, require that NIP applications publish the `msgKey` in `RsslUpdateMsgs`. To avoid component or transport migration issues, NIP applications can choose to always include this information, however this incurs additional bandwidth use and overhead. When designing your application, read the documentation for your other components to ensure that you take into account any other requirements.

Content is typically encoded and decoded using the Transport API Message Package (as described in Chapter 12, Message Package Detailed View) and the Transport API Data Package (as described in Chapter 11, Data Package Detailed View).

8.7 Log Out and Shut Down

After publishing content to the system, the NIP application should close all open streams and shut down the network connection.

- For more information about closing streams, refer to Section 12.2.5.
- For information about the Message Package, refer to Chapter 12, Message Package Detailed View.

When shutting down the provider, the application should release all unwritten pool buffers. Calling `rsslCloseChannel` terminates the connection to the ADH. Detailed information for transport and associated use cases are provided in Chapter 10, Transport Package Detailed View.

8.8 Additional NIP Details

For specific details about OMM Non-Interactive Providers and the Transport API use, refer to the following locations:

- The `rsslNIPProvider` application demonstrates one implementation of an OMM NIP application. The application's source code and `ReadMe` file have additional information about specific implementation and behaviors.
- For reviewing high-level encoding and decoding concepts, refer to Chapter 9, Encoding and Decoding Conventions.
- For a detailed look at the Data Package, typically used for encoding and decoding payload content, refer to Chapter 11, Data Package Detailed View.
- For a detailed look at the Message Package, used for all message encoding and decoding, refer to Chapter 12, Message Package Detailed View.
- For a detailed look at the Transport Package, used for the application's network communication, refer to Chapter 10, Transport Package Detailed View.
- For specific information about the DMMs required by the application, refer to the *Transport API C Edition RDM Usage Guide*.

4. Because the provider instantiates these information streams, a negative value `streamId` should be used for each stream. Additional details are provided in subsequent chapters.

Chapter 9 Encoding and Decoding Conventions

9.1 Concepts

Both the Transport API Message Package and Data Package allow the user to encode and decode constructs and various content. The Data Package defines a single encode iterator type and a single decode iterator type, which are used in both packages to encode and decode. The Transport API supports single-iterator encoding and decoding such that a single instance can encode or decode the full depth and breadth of a user's content. The application controls the depth of decoding, so you can skip content of no interest. Less efficiently, you can continue to leverage the Transport API to use separate iterator instances and hence allow the user to separate portions of content across iterators when encoding or decoding.

To provide consistency and ease of use when moving between data and message content, functions provided across both packages follow the same naming and usage conventions.

Data and Message packages do not provide inherent threading or locking capability. Separate iterator and type instances do not cause contention and do not share resources between instances. Any needed threading, locking, or thread-model implementation is at the discretion of the application. Different application threads can encode or decode different messages without requiring a lock; thus each thread must use its own iterator instance and each message should be encoded or decoded using unique and independent buffers. Though possible, Thomson Reuters recommends that you do not encode or decode related messages (ones that flow on the same stream) on different threads as this can scramble the delivery order.

9.1.1 Data Types

The Transport API offers a wide variety of data types categorized into two groups:

- **Primitive Types:** A primitive type represents simple, atomically updating information such as values like integers, dates, and ASCII string buffers (refer to Section 11.2).
- **Container Types:** A container type can model data representations more intricately and manage dynamic content at a more granular level than primitive types. Container types represent complex information such as field identifier-value, name-value, or key-value pairs (refer to Section 11.3). The Transport API offers several uniform, homogeneous container types (i.e., all entries house the same type of data). Additionally, there are several non-uniform, heterogeneous container types in which different entries can hold different types of data.

9.1.2 Composite Pattern of Data Types

The following diagram illustrates the use of Transport API data types to resemble a composite pattern.

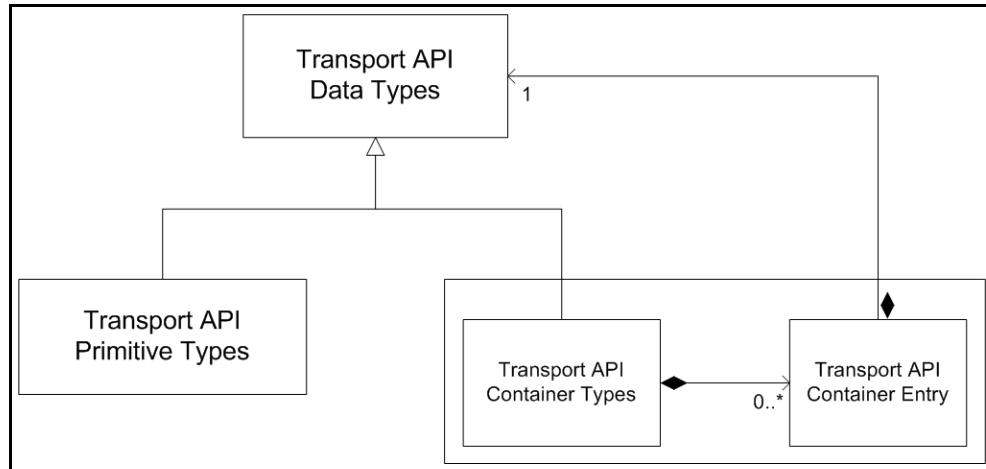


Figure 27. Transport API and the Composite Pattern

The diagram highlights the following:

- Being made up of both primitive and container types, Transport API data type values mirror the composite pattern's component.
- Transport API primitive types mimic the composite pattern's leaf, conveying concrete information for the user.
- The Transport API container type and its entries are similar to the composite pattern's composite. This allows for housing other container types and, in some cases such as field and element lists, housing primitive types.

The housing of other types is also referred to as **nesting**. Nesting allows:

- Messages to house other messages or container types
- Container types to house other messages, container, or primitive types

This provides the flexibility for domain model definitions and applications to arrange and nest data types in whatever way best achieves their goals.

9.2 Encoding Semantics

Because the Transport API supports several styles of encoding, the user can choose whichever method best fits their needs.

9.2.1 Init and Complete Suffixes

Encoding functions that have a suffix of **Init** or **Complete** (e.g. `rsslEncodeFieldEntryInit` and `rsslEncodeFieldEntryComplete`) allow the user to encode the type part-by-part, serializing each portion of data with each called function.

Functions without a suffix of **Init** or **Complete** (e.g. `rsslEncodeFieldEntry`, `rsslEncodeInt`, or `rsslEncodeMsg`) perform encoding within a single call, typically used for encoding simple types like Integer or incorporating previously encoded data (referred to as **pre-encoded** data).

9.2.2 The Encode Iterator: `RsslEncodeIterator`

To encode content you must use an `RsslEncodeIterator` and can use a single encode iterator to manage the entire encoding process¹ (including state and position information).

For example, if you want to encode a message that contains an `RsslFieldList` composed of various primitive types, you can use the same `RsslEncodeIterator` to encode all contents. In this case, initialize the iterator before encoding the message, and then pass the iterator as a parameter when encoding each portion. You do not need to perform additional initialization or clearing. When encoding finishes, you can determine the total encoded length and clear the iterator, reusing it for another encoding. If needed, you can use individual iterators for each level of encoding or for pre-encoding portions of data. However, when using separate iterators, you must initialize each iterator before starting the associated encoding process.

Initialization of an `RsslEncodeIterator` consists of several steps. After creating the iterator (typically stack allocated), clear it using `rsslClearEncodeIterator`. Each `RsslEncodeIterator` requires an `RsslBuffer` (provided via `rsslSetEncodeIteratorBuffer`) into which it encodes. RWF version information can also be populated on the iterator, ensuring that the proper version of the wire format is encoded (refer to Section 9.5.1).

1. A single `RsslEncodeIterator` can support up to sixteen levels of nesting, allowing for sixteen `Init` calls without a single `Complete` call. Because the most complex RDM currently requires only five levels, sixteen is believed to be sufficient. Should an encoding require more than sixteen levels of nesting, multiple iterators can be used.

9.2.2.1 Rssl Encode Iterator Functions

Note: Additional encoding examples are provided throughout this manual as well as in the Transport API package's example applications.

The following table describes functions that you can use with `RsslEncodeIterator`.

FUNCTION NAME	DESCRIPTION
<code>rsslClearEncodeIterator</code>	Clears members necessary for encoding and readies the iterator for reuse. You must clear the <code>RsslEncodeIterator</code> prior to starting any encoding process. For performance purposes, only those members necessary for proper functionality are cleared.
<code>rsslSetEncodeIteratorBuffer</code>	Associates an <code>RsslEncodeIterator</code> with the <code>RsslBuffer</code> into which it encodes. <code>RsslBuffer.data</code> should refer to sufficient space for encoding, and <code>RsslBuffer.length</code> should be set to the number of bytes available in <code>RsslBuffer.data</code> . This information ensures that encoding does not exceed allowable buffer space.
<code>rsslSetEncodeIteratorRWFVersion</code>	Associates RWF Versioning information to the <code>RsslEncodeIterator</code> , ensuring that The Transport API uses the appropriate wire format version while encoding. Wire format information is typically available on the connection between applications. Refer to Section 9.5.1.
<code>rsslGetEncodedBufferLength</code>	Returns the size (in bytes) of content encoded with the <code>RsslEncodeIterator</code> . After encoding is complete, use this function to set <code>RsslBuffer.length</code> to the size of data contained in the buffer.
<code>rsslRealignEncodeIteratorBuffer</code>	If an encoding process exceeds the space allocated in the current <code>RsslBuffer</code> , this function dynamically associates a new, larger buffer with the encoding process, allowing encoding to continue.

Table 5: `Rssl Encode Iterator` Utility Functions

9.2.2.2 Rssl Encode Iterator: Basic Use Example

The following example illustrates how to initialize `RsslEncodeIterator` in a typical fashion and set the buffer's length once encoding completes.

```

/* create and clear iterator to prepare for encoding */
RsslEncodeIterator encodeIter;
rsslClearEncodeIterator(&encodeIter);
/* associate buffer and iterator, code assumes that pBuffer->data points to sufficient
   memory and pBuffer->length indicates number of bytes available in pBuffer->data */
if (rsslSetEncodeIteratorBuffer(&encodeIter, pBuffer) < RSSL_RET_SUCCESS)
{
    printf("Error %s (%d) encountered with rsslSetEncodeIteratorBuffer. Error Text: %s\n",
           rsslRetCodeToString(retval), retval, rsslRetCodeInfo(retval));
    return;
}
/* set proper protocol version information on iterator, this can typically be obtained from
   the RsslChannel associated with the connection once it becomes active */
if (rsslSetEncodeIteratorRWFVersion(&encodeIter, majorVersion, minorVersion) <
    RSSL_RET_SUCCESS)
{
    printf("Error %s (%d) encountered with rsslSetEncodeIteratorRWFVersion. Error Text:
           %s\n", rsslRetCodeToString(retval), retval, rsslRetCodeInfo(retval));
}

/* Perform all content encoding now that iterator is prepared. */

/* When encoding is complete, set the pBuffer->length to the number of bytes Encoded */
pBuffer->length = rsslGetEncodedBufferLength(&encodeIter);

```

Code Example 1: Rssl Encode Iterator Usage Example

9.2.3 Content Roll Back with Example

Every **Complete** method has a **success** parameter, which allows you to discard unsuccessfully encoded content and roll back to the last successfully encoded portion.

For example, you begin encoding a list that contains multiple entries, but the tenth entry in the list fails to encode. To salvage the successful portion of the encoding, pass the **success** parameter as **RSSL_FALSE** when calling the failed entry's **Complete** method. This rolls back encoding to the end of the last successful entry. The remaining **complete** methods should be called, after which the application can use the encoded content. You can begin a new encoding for the remaining entries.

The following example demonstrates the use of the roll back procedure. This example encodes an **RsslMap** with two entries. The first entry succeeds; so **success** is passed in as **RSSL_TRUE**. However, encoding the second entry's contents fails, so the second map entry is rolled back, and the map is completed. To highlight the rollback feature, only those portions relevant to the example are included.

```
/* example shows encoding a map with two entries, where second entry content fails so it is
rolled back */
 retVal = rsslEncodeMapInit(&encIter, &rsslMap, 0, 0);

/* Encode the first map entry - this one succeeds */
 retVal = rsslEncodeMapEntryInit(&encIter, &mapEntry, &entryKey, 0);
/* encode contents - assume this succeeds */
/* Passing true for the success parameter completes encoding of this entry */
 retVal = rsslEncodeMapEntryComplete(&encIter, RSSL_TRUE);

/* Encode the second map entry - this one fails */
 retVal = rsslEncodeMapEntryInit(&encIter, &mapEntry, &entryKey, 0);
/* encode contents - assume this fails */
/* Passing false for the success parameter rolls back the encoding to the end of the previous
entry */
 retVal = rsslEncodeMapEntryComplete(&encIter, RSSL_FALSE);

/* Now complete encoding of the map - this results in only one entry being contained in the map
*/
 retVal = rsslEncodeMapComplete(&encIter, RSSL_TRUE);
```

Code Example 2: Encoding Rollback Example

9.3 Decoding Semantics and RsslDecodeIterator

Using the Transport API, applications can decode the full depth of the content or skip over portions in which the application is not interested. Each container type provided by the Transport API includes functionality for decoding the container header and decoding each entry in the container. If an application wishes to decode information present in a container entry, it can invoke the specific decode function associated with the nested type. When nested content is completely decoded, the next container entry can be decoded. If an application wishes to skip decoding data nested in a container entry, it can simply call the container entry decode function again without invoking the decoder for nested content. A decoding application will typically loop on decode until `RSSL_RET_END_OF_CONTAINER` is returned.



Tip: Decoding examples are provided throughout this manual as well as in the example applications provided with the Transport API package.

9.3.1 The Decode Iterator: RsslDecodeIterator

All decoding requires the use of an `RsslDecodeIterator`. You can use a single decode iterator to manage the full decoding process, internally managing various state and position information while decoding.

For example, when decoding a message that contains an `RsslFieldList` composed of various primitive types, you can use the same `RsslDecodeIterator` to decode all contents, including primitive types. In this case, you want to initialize the iterator before decoding the message and then pass the iterator as a parameter when decoding other portions (without additional initialization or clearing). After you completely decode all needed content, you can clear the iterator and reuse it for another decoding. If needed, you can use individual iterators for each level of decoding. However, if you use separate iterators, you must initialize each iterator before the decoding process that it manages.

Initialization of an `RsslDecodeIterator` consists of several steps. After the iterator is created (typically stack allocated), use `rsslClearDecodeIterator` to clear `RsslDecodeIterator`. Each `RsslDecodeIterator` requires an `RsslBuffer` (provided via `rsslSetDecodeIteratorBuffer`) from which to decode. RWF version information can also be populated on the iterator, thus decoding the appropriate version of the wire format (refer to Section 9.5.1).



Warning! The Transport API decodes directly from the `RsslBuffer` associated with the `RsslDecodeIterator`. If this `RsslBuffer.data` is adjacent to protected memory, it is possible that decoding content in the last bytes will result in attempted access to that protected memory due to optimized byte swapping routines. Padding the end of the `RsslBuffer` with an additional 7 bytes of space allows optimized swap routines to function properly without accessing protected memory.

9.3.2 Functions for use with RsslDecodeIterator

The following table describes the functions that you can use with `RsslDecodeIterator`.

function NAME	DESCRIPTION
<code>rsslClearDecodeIterator</code>	Clears members necessary for decoding and readies the iterator for reuse. You must clear <code>RsslDecodeIterator</code> before decoding content. For performance purposes, only those members required for proper functionality are cleared.

Table 6: Rssl Decode Iterator Utility Functions

function NAME	DESCRIPTION
rsslSetDecodeIteratorBuffer	Associates the <code>RsslDecodeIterator</code> with the <code>RsslBuffer</code> from which to decode. Set <code>RsslBuffer.data</code> to refer to the content to be decoded and <code>RsslBuffer.length</code> to the number of bytes contained in <code>RsslBuffer.data</code> . These settings ensure that decoding does not provide content beyond what is contained in the buffer space.
rsslSetDecodeIteratorRWFVersion	Sets the RWF Version to use with the <code>RsslDecodeIterator</code> . The appropriate RWF Version is typically available on the connection between applications. Refer to Section 9.5.1.
rsslFinishDecodeEntries	The decoding process typically runs until the end of each container, indicated by <code>RSSL_RET_END_OF_CONTAINER</code> . This function will skip past remaining entries in the container and perform necessary synchronization between the content and iterator so that decoding can continue.

Table 6: Rssl Decode Iterator Utility Functions (Continued)

9.3.3 RsslDecodeIterator: Basic Use Example

The following example demonstrates a typical `RsslDecodeIterator` initialization process.

```

/* create and clear iterator to prepare for decoding */
RsslDecodeIterator decodeIter;
rsslClearDecodeIterator(&decodeIter);
/* associate buffer and iterator, code assumes that pBuffer->data points to encoded contents
   to decode */
if (rsslSetDecodeIteratorBuffer(&decodeIter, pBuffer) < RSSL_RET_SUCCESS)
{
    printf("Error %s (%d) encountered with rsslSetDecodeIteratorBuffer. Error Text: %s\n",
           rsslRetCodeToString(retval), retval, rsslRetCodeInfo(retval));
    return;
}
/* set proper protocol version information on iterator, this can typically be obtained from
   the RsslChannel associated with the connection once it becomes active */
if (rsslSetDecodeIteratorRWFVersion(&decodeIter, majorVersion, minorVersion) <
    RSSL_RET_SUCCESS)
{
    printf("Error %s (%d) encountered with rsslSetDecodeIteratorRWFVersion. Error Text:
           %s\n", rsslRetCodeToString(retval), retval, rsslRetCodeInfo(retval));
}

/* Perform all content decoding now that iterator is prepared. */

```

Code Example 3: Rssl Decode Iterator Usage Example

9.4 Return Code Values

RSSL Data and Message functionality returns codes indicating success or failure.

- On failure conditions, these codes inform the user of the error.
- On success conditions, these codes provide the application additional direction regarding the next encoding steps.

When using RSSL Data and Message packages, return codes greater than or equal to **RSSL_RET_SUCCESS** indicate some type of specific success code, while codes less than **RSSL_RET_SUCCESS** indicate some type of specific failure.

Note: The Transport Layer has special semantics associated with its return codes. It does not follow the same semantics as the Data and Message Packages. For detailed handling instructions and return code information, refer to Chapter 10, Transport Package Detailed View.

9.4.1 Success Codes

The following table describes success value return codes associated with RSSL Data and Message packages.

RETURN CODE	DESCRIPTION
RSSL_RET_SUCCESS	Indicates operational success. Does not indicate next steps, though additional encoding or decoding might be required.
RSSL_RET_ENCODE_MSG_KEY_ATTRIB	Indicates that initial message encoding was successful and now the application should encode <code>msgKey</code> attributes. This return occurs if the application indicates that the message should include <code>msgKey</code> attributes when calling <code>rsslEncodeMsgInit</code> (RSSL_MKF_HAS_ATTRIB) without populating pre-encoded data into <code>msgKey.encAttrib</code> . For further details, refer to Section 12.1.2 and Code Example 41.
RSSL_RET_ENCODE_EXTENDED_HEADER	Indicates that initial message encoding (and <code>msgKey</code> attribute encoding) was successful, and the application should now encode <code>extendedHeader</code> content. This return occurs if an application indicates that the message should include <code>extendedHeader</code> content when calling <code>rsslEncodeMsgInit</code> without populating pre-encoded data into the <code>extendedHeader</code> . For further details on message encoding information, refer to Chapter 12, Message Package Detailed View.
RSSL_RET_ENCODE_CONTAINER	Indicates that initial encoding succeeded and that the application should now encode the specified <code>containerType</code> . <ul style="list-style-type: none"> For details on container types, refer to Section 11.3. For details on encoding messages, refer to Chapter 12, Message Package Detailed View.
RSSL_RET_SET_COMPLETE	Indicates that <code>RsslFieldList</code> or <code>RsslElementList</code> encoding is complete. Additionally encoded entries are encoded in the standard way with no additional data optimizations. For further information, refer to Section 11.6.

Table 7: Data and Message Package Success Return Codes

RETURN CODE	DESCRIPTION
RSSL_RET_DICT_PART_ENCODED	<p>Indicates that the dictionary encoding utility function successfully encoded part of a dictionary message (because dictionary messages tend to be large, they might be segmented into a multi-part message).</p> <ul style="list-style-type: none"> • For specific information about the Dictionary domain and the utility functions provided by the Transport API, refer to the <i>Transport API C Edition RDM Usage Guide</i>. • For more details on message fragmentation, refer to Section 13.1.
RSSL_RET_BLANK_DATA	<p>Indicates that the decoded primitiveType is a blank value. The contents of the primitive type should be ignored; any display or calculation should treat the value as blank.</p> <p>For further details on primitive types, refer to Section 11.2.</p>
RSSL_RET_NO_DATA	<p>Indicates that the containerType being decoded contains no data and was decoded from an empty payload. Informs the application not to continue to decode container entries (as none exist).</p>
RSSL_RET_END_OF_CONTAINER	<p>Indicates that the decoding process has reached the end of the current container. If decoding nested content, additional decoding might still be needed. The application can move back up the nesting stack and continue decoding the next container entry by calling the container's specific entry decode function.</p> <p>For example, if decoding an RsslFieldList contained in an RsslMapEntry, when this code is returned, it signifies that the contained field list decoding is complete.</p> <p>For details on container types, refer to Section 11.3.</p>
RSSL_RET_SET_SKIPPED	<p>Indicates that RsslFieldList or RsslElementList decoding skipped over contained, set-defined data because a set definition database was not provided. Any standard encoded entries will still be decoded.</p> <p>For further information on set definitions, refer to Section 11.6.</p>
RSSL_RET_SET_DEF_DB_EMPTY	<p>Indicates that decoding of a set definition database completed successfully, but the database was empty.</p> <p>For further information, refer to Section 11.6.</p>

Table 7: Data and Message Package Success Return Codes (Continued)

9.4.2 Failure Codes

RETURN CODE	Description
RSSL_RET_FAILURE	Indicates a general failure, used when no specific details are available.
RSSL_RET_BUFFER_TOO_SMALL	Indicates that the <code>RsslBuffer</code> on the <code>RsslEncodeIterator</code> lacks sufficient space for encoding.
RSSL_RET_INVALID_ARGUMENT	Indicates an invalid argument was provided to an encoding or decoding function.
RSSL_RET_ENCODING_UNAVAILABLE	Indicates that the invoked function does not contain encoding functionality for the specified type. There might be other ways to encode content or the type might be invalid in the combination being performed.
RSSL_RET_UNSUPPORTED_DATA_TYPE	Indicates that the type is not supported for the operation being performed. This might indicate a <code>primitiveType</code> is used where a <code>containerType</code> is expected or the opposite.
RSSL_RET_UNEXPECTED_ENCODER_CALL	Indicates that encoding functionality was used in an unexpected sequence or the called function is not expected in this encoding.
RSSL_RET_INCOMPLETE_DATA	Indicates that the <code>RsslBuffer</code> on the <code>RsslDecodeIterator</code> does not have enough data for proper decoding.
RSSL_RET_INVALID_DATA	Indicates that invalid data was provided to the invoked function.
RSSL_RET_ITERATOR_OVERRUN	Indicates that the application is attempting to nest more levels of content than is supported by a single <code>RsslEncodeIterator</code> ^a . If this occurs, you should use multiple iterators for encoding.
RSSL_RET_VALUE_OUT_OF_RANGE	Indicates that a value being encoded using a set definition exceeds the allowable range for the type as specified in the definition. For further information on set definitions, refer to Section 11.6.
RSSL_RET_SET_DEF_NOT_PROVIDED	Indicates that <code>RsslFieldList</code> or <code>RsslElementList</code> encoding requires a set definition database which was not provided. For more information, refer to Section 11.6.
RSSL_RET_TOO_MANY_LOCAL_SET_DEFS	Indicates that encoding exceeds the maximum number of allowed local set definitions. Currently 15 local set definitions are allowed per database. For more information, refer to Section 11.6.
RSSL_RET_DUPLICATE_LOCAL_SET_DEFS	Indicates that content includes a duplicate set definition that collides with a definition already stored in the database. For more information, refer to Section 11.6.
RSSL_RET_ILLEGAL_LOCAL_SET_DEF	Indicates that the <code>setId</code> associated with a contained definition exceeds the allowable value. Currently <code>setId</code> values up to 15 are allowed. For more information, refer to Section 11.6.

Table 8: Data and Message Package Failure Return Codes

- a. A single **Rssl EncodeIterator** can support up to sixteen levels of nesting (this allows for sixteen **InIt** calls without a single **Complete** call). Currently, the most complex RDM requires five levels, so sixteen is sufficient. If an encoding requires more than sixteen levels of nesting, multiple iterators can be employed.

9.5 Versioning

The Transport API supports two types of versioning:

- Protocol Versioning: Allows for the exchange of protocol type and version information across a connection established with the RSSL Transport Package. Protocol and version information can be provided to the **RsslEncodeIterator** and **RsslDecodeIterator** to ensure the proper handling and use of the appropriate wire format version.

Note: Thomson Reuters strongly recommends that you write all Transport API applications to leverage wire format versioning.

- Library Versioning: Allows for applications to programmatically query library version information. Library versioning ensures that expected libraries are used and that all versions match in the application.

9.5.1 Protocol Versioning

Consumer and provider applications using the RSSL Transport can provide protocol type and version information. This data is supplied as part of **RsslConnectOptions** or **RsslBindOptions** and populated via the **protocolType**, **majorVersion**, and **minorVersion** members. When establishing a connection, data is exchanged and negotiated between client and server:

- If the client's specified **protocolType** does not match the server's specified **protocolType**, the connection is refused.
- If the **protocolType** information matches, version information is compared and a compatible version determined.

After a connection becomes active, negotiated version information is available via the **RsslChannel** from both client and server and can be used for encoding and decoding:

- To populate version information on an **RsslEncodeIterator**, call the **rsslSetEncodeIteratorRWFVersion** function.
- To populate version information on an **RsslDecodeIterator**, call the **rsslSetDecodeIteratorRWFVersion** function.

The RSSL Transport layer is data neutral and does not change or depend on data distribution. Versioning information is provided only to help client and server applications manage the data they communicate. For further details on the RSSL Transport, refer to Chapter 10, Transport Package Detailed View.

Note: Properly using Transport API's versioning functionality helps minimize future impacts associated with underlying format modifications and enhancements, ensuring compatibility with other Transport API-enabled components.

Typically, an increase in the major version is associated with the introduction of an incompatible change. An increase in the minor version tends to signify the introduction of a compatible change or extension.

The Data Package contains several defined values that you can use with protocol versioning:

DEFINE NAME	DESCRIPTION
RSSL_RWF_PROTOCOL_TYPE	Defines the protocolType value associated with RWF. Define other protocols using different protocolType values.
RSSL_RWF_MAJOR_VERSION	Sets the value associated with the current major version. If incompatible changes are introduced, this value is incremented.

Table 9: Rssl LibraryVersionInfo Structure Members

DEFINE NAME	DESCRIPTION
RSSL_RWF_MINOR_VERSION	Sets the value associated with the current minor version. If extensions or compatible changes are introduced, this value is incremented.
RSSL_RWF_MAX_SUPPORTED_MAJOR_VERSION	Defines the maximum RWF major version for which this product release supports encoding or decoding. Any higher value requires the use of libraries from a more current release.
RSSL_RWF_MIN_SUPPORTED_MAJOR_VERSION	Defines the minimum RWF major version for which this product release supports encoding or decoding. Any lower value requires the use of libraries from an older release.

Table 9: Rssl LibraryVersionInfo Structure Members (Continued)

9.5.2 Library Versioning

Each Transport API library embeds its own version data as well as internal Thomson Reuters build version data. There are several ways in which you can obtain this data. From a console, you can use the **strings** command to search for **PACKAGE**, which provides Transport API package version data, and **VERSION**, which provides the internal Thomson Reuters build version data. Any issues raised to support should include this version data.

MEMBER	DESCRIPTION
productVersion	Contains the library's version.
internalVersion	Contains the internal Thomson Reuters build data.
productDate	Contains the build date for the product release.

Table 10: Rssl LibraryVersionInfo Structure Members

Additionally, each Transport API library includes a utility function. Using utility functions you can programmatically extract library version information. Each function populates an **RsslLibraryVersionInfo** structure, as defined in the following table.

FUNCTION NAME	DESCRIPTION
rsslQueryDataLibraryVersion	Retrieves version data associated with the RSSL Data Package library.
rsslQueryMessagesLibraryVersion	Retrieves version data associated with the RSSL Message Package library.
rsslQueryTransportLibraryVersion	Retrieves version data associated with the RSSL Transport Package library.

Table 11: Library Version Utility Functions

Chapter 10 Transport Package Detailed View

10.1 Concepts

The Transport API offers a Transport Package capable of communicating with other OMM-based components, including but not limited to TREP, Elektron, EDF Direct, and other TREP API OMM-based applications. The Transport Package efficiently sends and receives data across TCP/IP-based networks, leverages HTTP or HTTPS connection types, and presents a message-based interface to applications for ease of reading and writing data.

The package exposes a feature set that includes a receiver-transparent way for senders to combine or pack multiple messages into one outbound packet, as well as transparent fragmentation and reassembly of messages which exceed the size of an outbound packet. Structural representations are provided for managing connections (referred to as channels).

The transport layer offers multiple degrees of thread safety, all programmatically configurable by the application. This ranges from a fully thread-safe option¹ to the ability for an application to turn off all protective locking². Threading implementation and thread-model selection is managed by the application. The transport provides different locking options to provide maximum flexibility to the user. For more information, refer to Section 10.2.3.

The transport supports both non-blocking and blocking I/O models, however use of blocking I/O is not recommended. When a blocking operation is occurring, control will not be returned to the application until the operation has fully completed (e.g. all information is written). This prevents the application from performing additional tasks, including heartbeat sending and monitoring, while the transport operation may be waiting for the operating system. By employing an I/O notification mechanism (e.g. select, poll), an application can leverage a non-blocking I/O model, using the I/O notification to alert the application when data is available to read or when output space is available for writing to. The following sections are written with an emphasis on non-blocking I/O use, though blocking behavior is also described. All examples are written from a non-blocking I/O perspective.

1. When this option is enabled, RSSL Transport can function correctly during simultaneous execution by multiple application threads.

2. When this option is enabled, all locking is disabled for additional performance. If required, the application must provide any necessary thread safety.

10.1.1 Transport Types

The transport supports configuration of multiple connection types for different systems, while providing a single interface for a look and feel that is similar among all connections and components. Developers should ensure that the components to which they intend to connect are configured to support the appropriate transport type.

10.1.1.1 Socket Transport

The Transport API provides a transport for efficiently distributing messages across a TCP/IP-based reliable network (**RSSL_CONN_TYPE_SOCKET**). This transport is capable of connecting to various OMM-based components, including but not limited to Enterprise Platform, Elektron, RDF Direct, and other Transport API or RFA OMM-based applications. On specific platforms, applications can also leverage tunneling through HTTP (**RSSL_CONN_TYPE_HTTP**) or HTTPS (**RSSL_CONN_TYPE_ENCRYPTED**) connection types for internet connectivity.

The socket transport allows for both establishing outbound connections and for creating listening sockets to accept inbound connections. Once a connection is established, both connected components can send and receive information. Outbound connections are typically created by OMM Consumer applications to connect to an ADS or OMM Interactive Provider, or by OMM Non-Interactive Provider applications to connect to an ADH. Listening sockets are typically created by OMM Interactive Provider applications to allow OMM Consumer applications or ADSs to instantiate connections to it and request data.

10.1.1.2 Reliable Multicast Transport

The Transport API provides an efficient transport for exchanging messages over a UDP Multicast-based network (**RSSL_CONN_TYPE_RELIABLE_MCAST**). This transport leverages the same technology used on the Enterprise Platform Backbone to improve reliability of message delivery and automatically re-sequence out-of-order messages.

OMM Non-Interactive Provider applications may create multicast connections for publishing to an ADH. OMM Consumer applications may leverage the Transport API Reactor and its watchlist feature to create connections to an ADS. For more information on the Transport API Reactor, refer to the *Transport API C Edition Value Added Components Developers Guide*.

10.1.1.3 Sequenced Multicast Transport

The Transport API provides an efficient transport for reading messages over the UDP Multicast-based network (**RSSL_CONN_TYPE_SEQ_MCAST**). The Sequenced Multicast protocol is a special, unreliable UDP multicast with built-in sequence numbers that allow the user to ensure order and identify gaps in their applications.

10.1.2 RSSL Channel Structure

The channel structure represents a connection that can send or receive information across a network, regardless of whether the connection is outbound or accepted by a listening socket. The Transport Package internally manages any memory associated with an **Rss1Channel** structure, and the application does not need to create nor destroy memory (associated with the channel). The **Rss1Channel** is typically used to perform any action on the connection that it represents (e.g. reading, writing, disconnecting, etc). See the subsequent sections for more information about **Rss1Channel** use within the transport.

The following table describes the members of the **Rss1Channel** structure.

Structure Member	Description
socketId	Represents a file descriptor that can be used in some kind of I/O notification mechanism (e.g. select, poll). 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 return code of RSSL_RET_READ_FD_CHANGE (for further information, refer to Section 10.6). 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.
state	The state associated with the RsslChannel . Until the channel has completed its initialization handshake and has transitioned to an active state, no reading or writing can be performed. Table 13 describes channel state values.
connectionType	An enumerated value that indicates the type of underlying connection being used. For more information, refer to Table 14.
clientIP	When a server completes the connection handshake with an accepted connection and the RsslChannel becomes active, this is populated with a string representation of the IP address of the connecting client. This value is not populated for clients calling the rsslConnect function.
clientHostname	When a server completes the connection handshake with an accepted connection and the RsslChannel becomes active, this is populated with a string representation of the hostname of the connecting client. This value is not populated for clients calling the rsslConnect function.
pingTimeout	When an RsslChannel becomes active for a client or server, this is populated with the negotiated ping timeout value. This is the number of seconds after which no communication can result in a connection being terminated. Both client and server applications should send heartbeat information within this interval. The typically used rule of thumb is to send a heartbeat every pingTimeout /3 seconds. For more information, refer to Section 10.12.
protocolType	When an RsslChannel becomes active for a client or server, 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 layer is data-neutral and allows the flow of any type of content. protocolType is provided to help client and server applications manage the information they communicate. For more details, refer to Section 9.5.1.
majorVersion	When an RsslChannel becomes active for a client or server, this is populated with the negotiated major version number that is associated with the content being sent on this connection. Typically, a major version increase is associated with the introduction of incompatible change. The transport layer is data-neutral and allows the flow of any type of content. majorVersion is provided to help client and server applications manage the information they communicate. For more details, refer to Section 9.5.1.

Table 12: Rssl Channel Structure Members

Structure Member	Description
minorVersion	<p>When an RsslChannel becomes active for a client or server, this is populated with the negotiated 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.</p> <p>The transport layer is data-neutral and allows the flow of any type of content. minorVersion is provided to help client and server applications manage the information they communicate. For more details, refer to Section 9.5.1.</p>
userSpecPtr	A pointer that can be set by the user of the RsslChannel . This value can be set directly or via the connection options and is not modified by the transport. This information can be useful for coupling this RsslChannel with other user created information, such as a watch list associated with this connection.

Table 12: **Rssl Channel** Structure Members (Continued)

10.1.2.1 RsslChannel Enumerated Name Values

ENUMERATED NAME	DESCRIPTION
RSSL_CH_STATE_INACTIVE	Indicates that an RsslChannel is inactive. This channel cannot be used. This state typically occurs after a channel is closed by the user.
RSSL_CH_STATE_INITIALIZING	Indicates that an RsslChannel requires additional initialization. This initialization is typically additional connection handshake messages that need to be exchanged. When using blocking I/O, an RsslChannel is typically active when it is returned and no additional initialization is required by the user.
RSSL_CH_STATE_ACTIVE	Indicates that an RsslChannel is active. This channel can perform any connection-related actions, such as reading or writing.
RSSL_CH_STATE_CLOSED	Indicates that an RsslChannel has been closed. This typically occurs as a result of an error inside of a transport function call and is often related to a socket being closed or becoming unavailable. Appropriate error value return codes and RsslError information should be available for the user.

Table 13: RSSL Connection State Values

10.1.2.2 RSSL Connection Enumerated Names Values

RSSL Connection types are used in several areas of the transport. When creating a connection, an application can specify which connection type to use (refer to Section 10.3). Additionally, after a connection is established, the `RsslChannel.connectionType` will indicate the connection type being used.

ENUMERATED NAME	DESCRIPTION
RSSL_CONN_TYPE_INIT	Indicates that the <code>RsslChannel</code> is not connected.
RSSL_CONN_TYPE_SOCKET	Indicates that the <code>RsslChannel</code> uses a standard, TCP-based socket connection. This type can be used to connect between any RSSL Transport-based applications.
RSSL_CONN_TYPE_HTTP_TUNNEL	Indicates that the <code>RsslChannel</code> tunnels using HTTP. This type can be used to connect between any RSSL Transport-based applications. For more information, refer to Section 4.6.
RSSL_CONN_TYPE_ENCRYPTED	Indicates that the <code>RsslChannel</code> tunnels using encryption. The encryption use is transparent to the client application. For a server to accept encrypted connection types the use of an external encryption/decryption device is required (encryption / decryption is not performed by the server). Because data will already be decrypted when it arrives at the server, an <code>RsslChannel</code> may indicate that a connection type is HTTP or SOCKET, even if the connection was established by specifying ENCRYPTED . The client leverages the Microsoft WinINET library, which requires use of multiple underlying connections managed by the RSSL Transport. For more information, refer to Section 4.6.
RSSL_CONN_TYPE_RELIABLE_MCAST	Indicates that the <code>RsslChannel</code> uses a UDP-based, reliable multicast connection type. This connection type is available only to applications using the <code>rsslConnect</code> function to establish their connection. The reliable multicast connection type ensures proper ordering of content across the network and, through the use of an acknowledgment and retransmission mechanism, backfills recent packet gaps. In situations where a packet gap cannot be filled, the application is notified of the gap situation. The default behavior for this connection type is to stay connected to the multicast, even in a gap situation. This allows the application to attempt recovery in a manner that might minimize any affect on the network. You can control this behavior via the <code>disconnectOnGaps</code> option described in Table 24.
RSSL_CONN_TYPE_SEQ_MCAST	Indicates that the <code>RsslChannel</code> uses a UDP-based, sequenced multicast connection type. This connection type is available only to applications using the <code>rsslConnect</code> function to establish their connection. Though this connection type uses sequence numbers which enables gap detection, it only ensures the proper ordering of content across the network; it does not acknowledge or retransmit packets to fill a gap. The default behavior for this connection type is to stay connected to the multicast, even in a gap situation. This allows for the application to attempt recovery in a manner that might minimize any affect on the network. You can control this behavior via the <code>disconnectOnGaps</code> option described in Table 24.

Table 14: RSSL ConnectionType Values

ENUMERATED NAME	DESCRIPTION
RSSL_CONN_TYPE_UNIDIR_SHMEM	<p>Indicates that the <code>RsslChannel</code> is using a shared memory connection type. This connection type offers a one-way data flow from a single server to multiple clients using a shared memory segment for content delivery. However, the server and clients must run on the same machine.</p> <p>For compatibility purposes, this connection type provides an <code>RsslChannel.socketId</code> to the application. This <code>socketId</code> will always indicate that something is available to read, even when there is not. This ensures that the application is reading content with as little latency as possible. If needed, the application can implement alternate approaches that would allow for a less CPU intensive read algorithm.</p> <p> Warning! Transport API applications using this connection type require appropriate run-time permissions to create and lock memory on the system (e.g. <code>mlock()</code>). See operating system-specific information for details on ensuring applications have proper system access rights.</p>

Table 14: RSSL ConnectionType Values (Continued)

10.1.3 RSSL Server Structure

The RSSL Server structure is used to represent a server that is listening for incoming connection requests. Any memory associated with an **RsslServer** structure is internally managed by the RSSL Transport Package, and the application does not need to create nor destroy this type. The **RsslServer** is typically used to accept or reject incoming connection attempts. See the subsequent sections for more information about **RsslServer** use within the transport. The following table describes the members of the **RsslServer** structure.

STRUCTURE MEMBER	DESCRIPTION
socketId	Represents a file descriptor that can be used in some kind of I/O notification mechanism (e.g. select, poll). This is the file descriptor associated with listening socket. When triggered, this typically indicates that there is an incoming connection and rsslAccept should be called.
state	The state associated with the RsslServer . A server is typically returned as active unless an error occurred during the rsslBind call. Table 6 describes possible state values.
portNumber	The port number that this RsslServer is bound to and listening for incoming connections on.
userSpecPtr	A pointer that can be set by the user of the RsslServer . This value can be set directly or via the bind options and is not modified by the transport. This information can be useful for coupling this RsslServer with other user created information, such as a list of associated RsslChannel structures.

Table 15: **Rssl Server** Structure Members

10.1.4 Transport Error Handling

Many RSSL Transport Package functions take a parameter for returning detailed error information. This **RsslError** structure is populated only in the event of an error condition and should only be inspected when a specific failure code is returned from the function itself.

In several cases, positive return values are reserved or have special meaning, for example bytes remaining to write to the network. As a result, some negative return codes might be used to indicate success (e.g. **RSSL_RET_READ_PING**). Any specific transport-related success or failure error handling is described along with the function that requires it.

RsslError members are described in the following table.

Structure Member	DESCRIPTION
channel	A pointer to the RsslChannel structure on which the error occurred.
rsslErrorId	A Transport API-specific return code that specifies what error occurred. Refer to the following sections for specific error conditions that might arise.
sysError	Populated with the system errno or error number associated with the failure. This information is only available when the failure occurs as a result of a system function, and will be populated by 0 otherwise.
text ^a	Detailed text describing the error. This can include RSSL- specific error information, underlying library-specific error information, or a combination of both.

Table 16: **Rssl Error** Structure Members

a. **Rssl Error** text information is limited to 1,200 bytes in length.

10.1.5 General Transport Return Codes

It is important that the application monitors return values from all Transport API functions that provide return-codes. Where specific error values are returned or special handling is required, the subsequent sections describe the possible return codes from RSSL Transport functionality. The following table lists general error codes. For Transport return codes specific to a particular method, refer to that method's section:

- `rsslInitChannel` return codes: Section 10.5.4.
- `rsslRead` return codes: Section 10.6.2.
- `rsslWrite` return codes: Section 10.9.6.
- `rsslFlush` return codes: Section 10.10.3.

RETURN CODE	DESCRIPTION
RSSL_RET_SUCCESS	Indicates successful completion of the operation.
RSSL_RET_FAILURE	Indicates that initialization has failed and cannot progress. The <code>Rss1Channel.state</code> should be <code>RSSL_CH_STATE_CLOSED</code> . See the <code>RsslError</code> content for more information.
RSSL_RET_INIT_NOT_INITIALIZED	Indicates that the RSSL Transport has not been initialized. See the <code>RsslError</code> content for more details. For details on initializing, refer to Section 10.2.

Table 17: General Transport Return Codes

10.1.6 Application Lifecycle

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

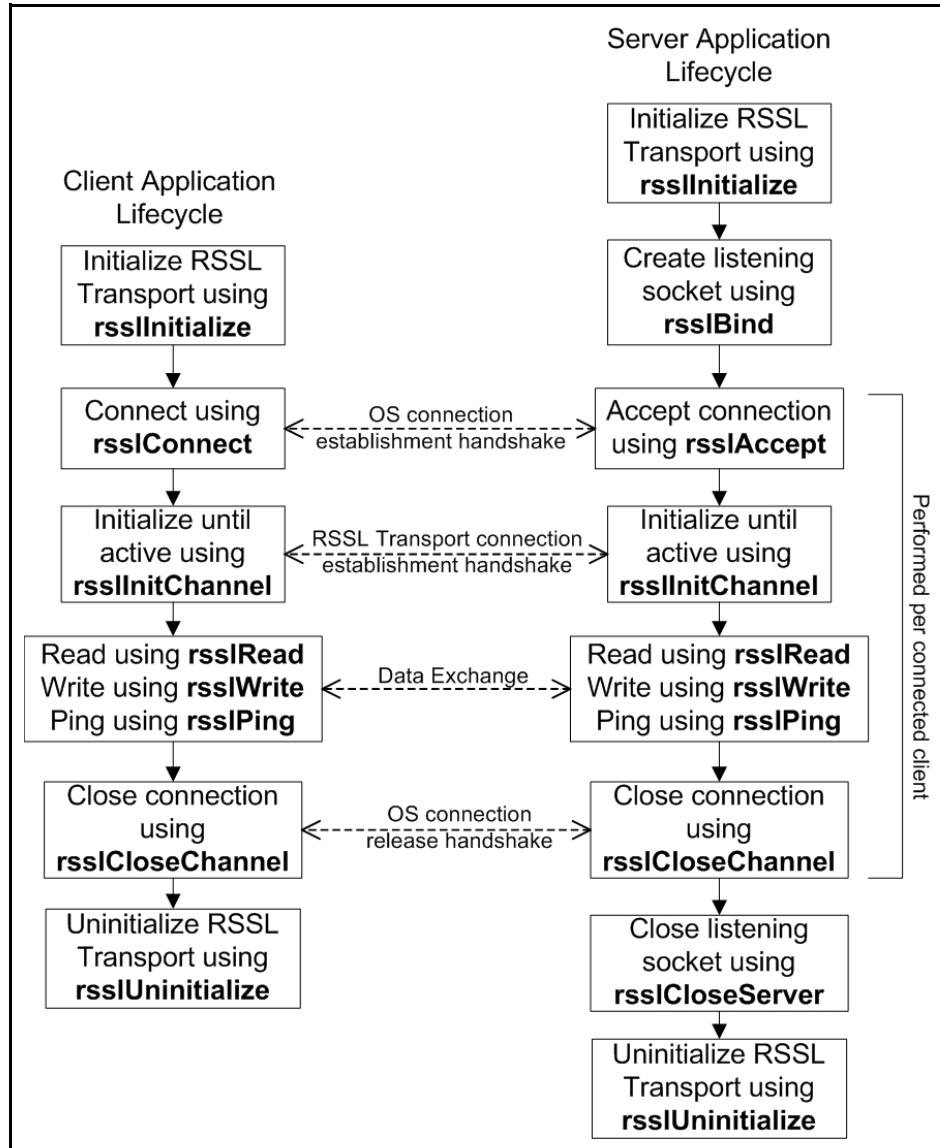


Figure 28. Application Lifecycle

10.2 Initializing and Uninitializing the Transport

Every application using the transport, client or server, must first initialize it. This initialization process allows the RSSL Transport to pre-allocate internal memory associated with buffering and channel management.

Similarly, when an application has completed its usage of the RSSL Transport, it must uninitialized it. The uninitialization process allows for any heap allocated memory to be cleaned up properly.

10.2.1 RSSL Initialization and Uninitialization Functions

The following table provides additional information about the RSSL Transport functions used for initializing and uninitialized.

function	Description
rsslInitialize	The first RSSL Transport function that an application should call. This creates and initializes internal memory and structures, as well as performing any boot strapping for underlying dependencies. The <code>rsslInitialize</code> function also allows the user to specify the locking model they want applied to the RSSL Transport. For more information, refer to Section 10.2.3.
rsslUninitialize	The last RSSL Transport function that an application should call. This uninitializeds internal data structures and deletes any allocated memory.

Table 18: RSSL Initialization and Uninitialization Functions

10.2.2 Initialization Reference Counting with Example

Both the `rsslInitialize` and `rsslUninitialize` functions use reference counting. This allows only the first call to `rsslInitialize` to perform any memory allocation or boot strapping and only the last necessary call to `rsslUninitialize` to undo the work of initialize. Only a single `rsslInitialize` call need be made within an application, however this call must be the first Transport function call performed.

The following example demonstrates the use of `rsslInitialize` and `rsslUninitialize`.

```
RsslError error;
/* Starting RSSL Transport use, must call initialize first */
if (rsslInitialize(RSSL_LOCK_GLOBAL_AND_CHANNEL, &error) < RSSL_RET_SUCCESS)
{
    printf("Error %s (%d) (errno: %d) encountered with rsslInitialize. Error Text: %s\n",
           rsslRetCodeToString(error.rsslErrorId), error.rsslErrorId, error.sysError,
           error.text);

    /* End application */
    return 0;
}

/* Any transport use occurs here - see following sections for all other functionality */
/* All RSSL Transport use is complete, must uninitialized */
rsslUninitialize();

/* End application */
return 0;
```

Code Example 4: Transport Initialization and Uninitialization

10.2.3 Transport Locking Models

The Rssl Transport offers the choice of several locking models. These locking models are designed to offer maximum flexibility and allow the transport to be used in the manner that best fits the application's design. There are three types of locking that occur in the RSSL transport. Global locking is used to protect any resources that are shared across connections or channels, such as connection pools. Channel locking is used to protect any resources that are shared within a single connection or channel, such as a channel's buffer pool. Shared pool locking is used to protect a server's shared buffer pool, which is used to share one pool of buffers across multiple connections.

All three types of locking can be enabled or disabled, depending on the needs of the application. Once a locking model is chosen, it cannot be changed without uninitialized and reinitializing the transport. This behavior is performed to ensure that there is no locking change pushed onto connections that may already be established. Shared pool locking is controlled on a per-server basis via [RsslBindOptions](#) (for more information, refer to Section 10.4.1.1). Global and channel locking are controlled by a parameter passed into the [rsslInitialize](#) function. The following table describes the valid options for use with [rsslInitialize](#).

ENUMERATED NAME	DESCRIPTION
RSSL_LOCK_NONE	Global and Channel locking will be disabled. This option can be used for single-threaded applications to remove any locking overhead since there is no risk of multiple thread access. It is additionally useful for multi-threaded applications that utilize the RSSL Transport from within a single thread. It is possible for an application to perform reading on an RsslChannel from one thread and writing to the same RsslChannel from a different thread - this requires synchronization while creating and destroying connections so use of RSSL_LOCK_GLOBAL is preferable.
RSSL_LOCK_GLOBAL_AND_CHANNEL	Both global locking and channel locking will be enabled. This, in addition to enabling shared pool locking, will provide full thread safety. This setting allows for accessing the same channel from multiple threads. Note that writing messages from multiple threads can result in ordering issues and it is not recommended to write related messages across different threads. Reading across multiple threads can also introduce ordering issues associated with information received, which may or may not impact ordering of related messages.
RSSL_LOCK_GLOBAL	Global locking is enabled and channel locking is disabled. This allows for any globally shared resources to be protected, but any channel related resources are not thread safe. This model allows for each channel to be handled by its own dedicated thread, but channel creation and destruction can occur across threads.

Table 19: RSSL Initialize Locking Options

10.3 Creating the Connection

The transport package allows for outbound connections to be established and managed. An outbound connection allows an application to connect to a listening socket or multicast network, often to some type of Provider running on a well known port number or multicast group address and port.

10.3.1 Network Topologies

The Transport API supports two types of network topologies:

- **unified**: A **unified** network topology is one where the **RsslChannel** uses the same connection information (**address:port**) to send and receive all content.
- **segmented**: A **segmented** network topology is one where the **RsslChannel** uses different connection information for sending and receiving. In the case of a **segmented** network, this allows for sent content and received content to be on different underlying **address:port** combinations.

On TCP-based networks, the Transport API supports only a **unified** topology (**RSSL_CONN_TYPE_SOCKET**, **RSSL_CONN_TYPE_HTTP**, and **RSSL_CONN_TYPE_ENCRYPTED**), but on multicast-based networks, the Transport API supports both **unified** and **segmented** topologies (**RSSL_CONN_TYPE_RELIABLE_MCAST** and **RSSL_CONN_TYPE_SEQ_MCAST**).

For configuration information on network topologies, refer to Table 22.

10.3.1.1 TCP-based Networks

If an application needs to communicate with multiple devices using a **RSSL_CONN_TYPE_SOCKET**, **RSSL_CONN_TYPE_HTTP**, and **RSSL_CONN_TYPE_ENCRYPTED** connection type, a unique (point-to-point) connection is required for each device. Any content that needs to go to all devices must be written (or “fanned out”) on all connections, which is the application’s responsibility. The following diagram illustrates this scenario:

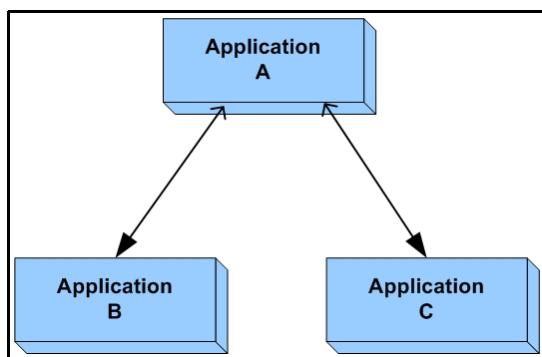


Figure 29. Unified TCP Network

In Figure 29, Application A has a unique, point-to-point connection with each of the applications B and C. If Application A wants to send the same content to both applications B and C, Application A must send the same content over each connection. In this scenario, if content is sent over only one connection, only the application on the corresponding end of that connection receives the content.

For TCP connections, OMM consumer and NIP applications connect as shown in Figure 30. The arrows used in the figure depict the directions in which connections are established. OMM consumers typically connect to a well known port number associated with some kind of Interactive Provider (e.g. ADS, Elektron), while OMM Non-Interactive Providers typically connect to a well known port on the ADH.

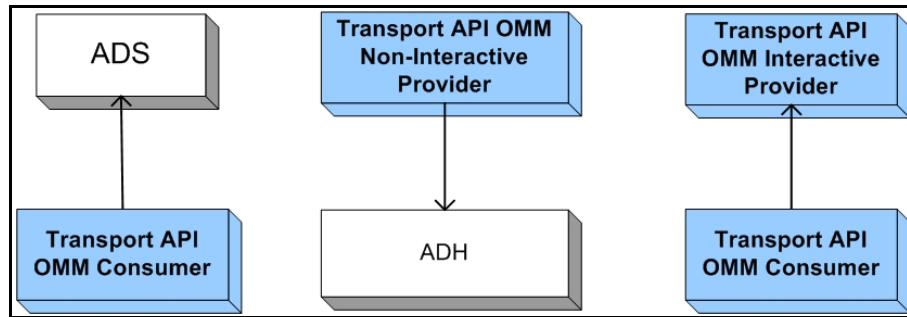


Figure 30. TCP Connection Creation

10.3.1.2 Multicast-based Networks: Unified

If an application wishes, it can communicate with multiple devices using a single connection to a multicast network (presuming the other devices access the same multicast network). In this case, a single transmission is sufficient to send data to all connected devices.

In the following diagram (Figure 31), all applications send and receive content on the same multicast network. Because the same network is used for sending and receiving traffic, all traffic is seen by all applications. Anything sent by one application will be received by all other applications on the network.

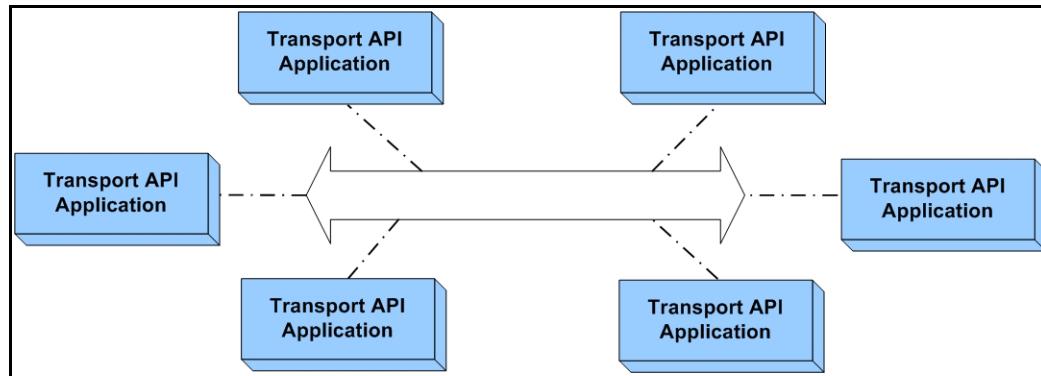


Figure 31. Unified Multicast Network

10.3.1.3 Multicast-based Networks: Segmented

In segmented multicast networks, applications transmit and receive data over different networks allowing users to separate applications based on the content they need to send or receive

In the following diagram (Figure 32):

- Applications A - C only send content on Network 1; they do not receive content from Network 1 (i.e., Application A does not see content sent by applications B or C). Applications A - C receive only the content sent on Network 2 (by applications D - F).
- Applications D - F only send content on Network 2; they do not receive content from Network 2 (i.e., Application D does not see content sent by applications E or F). Applications D - F receive only the content sent on Network 1 (by applications A - C).

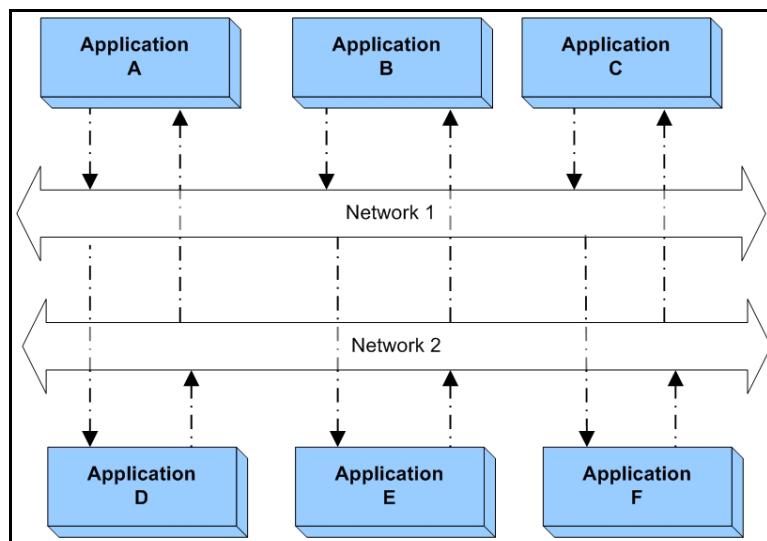


Figure 32. Segmented Multicast Network

The following diagram (Figure 33) illustrates OMM NIP applications using outbound multicast connections leveraging a segmented connection type. This allows the ADH to receive only content published by NIP applications (via the NIProv Send Network).

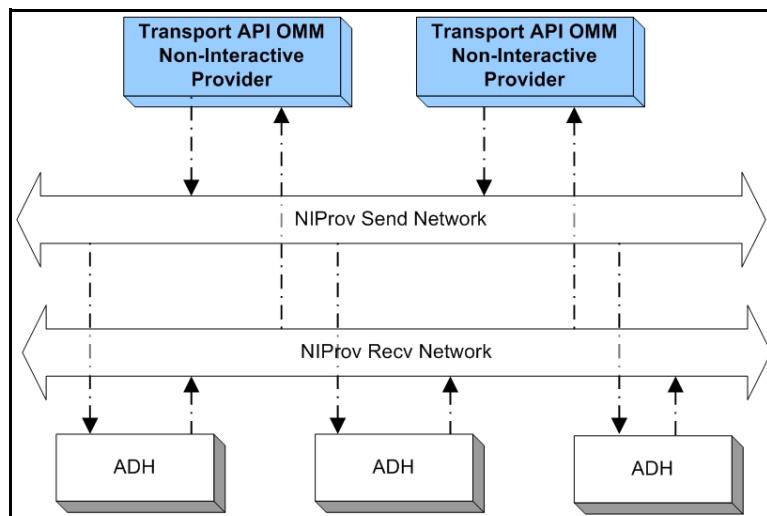


Figure 33. Multicast Connection Creation

The following diagram (Figure 34) illustrates Transport API Reactor Consumer applications leveraging a segmented network to connect to a ADS to consume multicast data. This allows a consumer to send to and receive data from multiple ADSs without receiving data from other consumers.

Note: Consuming data from an ADS multicast network is only supported when consuming through a watchlist-enabled Reactor. For more information, refer to the *Transport API C Edition Value Added Components Developers Guide*.

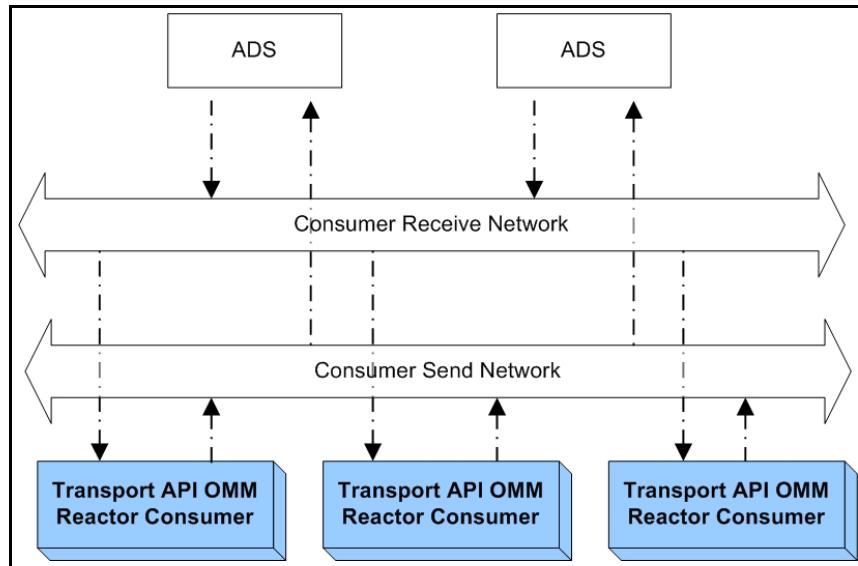


Figure 34. Consuming Multicast Data

10.3.2 Creating the Outbound Connection: `rsslConnect`

An application can create an outbound connection by using the `rsslConnect` function.

function NAME	DESCRIPTION
<code>rsslConnect</code>	<p>Establishes an outbound connection, which can leverage standard sockets, HTTP, or HTTPS. Returns an <code>RsslChannel</code> that represents the connection to the user. In the event of an error, NULL is returned and additional information can be found in the <code>RsslError</code> structure. Connection options are passed in via an <code>RsslConnectOptions</code> structure described in Table 21.</p> <p>Once a connection is established and transitions to the <code>RSSL_CH_STATE_ACTIVE</code> state, this <code>RsslChannel</code> can be used for other transport operations. For more information about channel initialization, refer to Section 10.5.</p>

Table 20: `rssl Connect` Function

10.3.2.1 RsslConnectOptions Structure Members

STRUCTURE MEMBER	DESCRIPTION
blocking	If set to RSSL_TRUE , blocking I/O will be used for this RsslChannel . When I/O is used in a blocking manner on an RsslChannel , any reading or writing will complete before control is returned to the application. In addition, the rsslConnect function will complete any initialization on the RsslChannel prior to returning it. Blocking I/O prevents the application from performing any operations until the I/O operation is completed. Blocking I/O is typically not recommended. An application can leverage an I/O notification mechanism to allow efficient reading and writing, while using other cycles to perform other necessary work in the application. An I/O notification mechanism enables the application to read when data is available, and write when output space is available.
componentVersion	An optional, user-defined component version string appended behind the standard Transport API component version information. If the combined component version length exceeds the maximum supported by the Transport API, the user-defined information will be truncated.
compressionType	The type of compression the client would like performed for this connection. Compression is negotiated between the client and server and may not be performed if only the client has it enabled. For more information about supported compression types and compression negotiation, refer to Section 10.4.3.
connectionInfo	Network configuration information. This includes unified and segmented network configuration parameters. For specific configuration information, refer to Table 22. This has replaced hostName and serviceName configuration.
connectionType	Specifies the type of connection to establish. Creation of TCP-based socket connection types or UDP-based multicast connection types are available across all supported platforms. Encrypted and HTTP connection types are available only for supported Windows platforms and depend on the Microsoft WinINET system library. Connection Types are described in more detail in Table 14.
guaranteedOutputBuffers	A guaranteed number of buffers made available for this RsslChannel to use while writing data. Guaranteed output buffers are allocated at initialization time. For more information, refer to Section 10.8.
hostName	DEPRECATED. For information on using connectionInfo.unified.address to configure equivalent functionality, refer to Table 22.
majorVersion	The major version of the protocol that the client intends to exchange over the connection. This value is negotiated with the server at connection time. The outcome of the negotiation is provided via the majorVersion information on the RsslChannel . Typically, a major version increase is associated with the introduction of incompatible change. The transport layer is data-neutral and allows the flow of any type of content. majorVersion is provided to help client and server applications manage the information they communicate. For more details, refer to Section 9.5.1.

Table 21: **Rssl ConnectOptions** Structure Members

STRUCTURE MEMBER	DESCRIPTION
minorVersion	The minor version of the protocol that the client intends to exchange over the connection. This value is negotiated with the server at connection time. The outcome of the negotiation is provided via the <code>minorVersion</code> information on the <code>RsslChannel</code> . Typically, a minor version increase is associated with a fully backward compatible change or extension. The transport layer is data-neutral and allows the flow of any type of content. <code>minorVersion</code> is provided to help client and server applications manage the information they communicate. For more details, refer to Section 9.5.1.
multicastOpts	A substructure containing multicast-based connection type specific options. These settings are used for <code>RSSL_CONN_TYPE_RELIABLE_MCAST</code> and <code>RSSL_CONN_TYPE_SEQ_MCAST</code> . For information about specific options, refer to Table 24.
numInputBuffers	The number of sequential input buffers to allocate for reading data into. This controls the maximum number of bytes that can be handled with a single network read operation. Input buffers are allocated at initialization time.
objectName	An optional object name to pass with a URL while tunneling. This option is only valid for HTTP and Encrypted connection types. For more information on connection types, refer to Table 14.
pingTimeout	The clients desired ping timeout value. This may change through the negotiation process between the client and the server. After the connection becomes active, the actual negotiated value becomes available through the <code>pingTimeout</code> value on the <code>RsslChannel</code> . When determining the desired ping timeout, the typically used rule of thumb is to send a heartbeat every <code>pingTimeout</code> /3 seconds. For more information, refer to Section 10.12.
protocolType	The protocol type that the client intends to exchange over the connection. If the <code>protocolType</code> indicated by a server does not match the <code>protocolType</code> that a client specifies, the connection will be rejected. When an <code>RsslChannel</code> becomes active for a client or server, this information becomes available via the <code>protocolType</code> on the <code>RsslChannel</code> . The transport layer is data-neutral and allows the flow of any type of content. <code>protocolType</code> is provided to help client and server applications manage the information they communicate. For more details, refer to Section 9.5.1.
serviceName	DEPRECATED. For information on using <code>connectionInfo.unified.serviceName</code> to configure equivalent functionality, refer to Table 22.
tcp_nodelay	DEPRECATED. To configure equivalent functionality, refer to <code>tcpOpts.tcp_nodelay</code> in Table 23.
tcpOpts	A substructure containing TCP-based connection type specific options. These settings are used for <code>RSSL_CONN_TYPE_SOCKET</code> , <code>RSSL_CONN_TYPE_HTTP</code> , and <code>RSSL_CONN_TYPE_ENCRYPTED</code> . For information about specific options, refer to Table 23.

Table 21: `RsslConnectOptions` Structure Members (Continued)

STRUCTURE MEMBER	DESCRIPTION
shmemOpts	A substructure containing shared memory based connection type specific options. These settings are used for RSSL_CONN_TYPE_UNIDIR_SHMEM . For information about specific options, refer to Table 25.
userSpecPtr	A pointer that can be set by the application. This value is not modified by the transport, but will be preserved and stored in the userSpecPtr of the RsslChannel returned from rsslConnect . This information can be useful for coupling this RsslChannel with other user created information, such as a watch list associated with this connection.

Table 21: **RsslConnectOptions** Structure Members (Continued)

10.3.2.2 RsslConnectOptions.connectionInfo Options

Option	Description
unified.address	Configures the address or hostname to use in a unified network configuration. All content will be sent and received on this address:serviceName pair. This replaces the RsslConnectOptions.hostName parameter.
unified.serviceName	Configures the numeric port number or service name (as defined in etc/services file) to use in a unified network configuration. All content will be sent and received on this address:serviceName pair. This replaces the RsslConnectOptions.serviceName parameter.
unified.interfaceName	A character representation of an IP address or hostname associated with the local network interface to use for sending and receiving content. This value is intended for use in systems which have multiple network interface cards, and if unspecified, the default network interface is used.
unified.unicastServiceName	Configures the numeric port number or service name (as defined in etc/services file) to use for all unicast UDP traffic in a unified network configuration. This parameter is only required for multicast connection types (RSSL_CONN_TYPE_RELIABLE_MCAST and RSSL_CONN_TYPE_SEQ_MCAST). If multiple connections or applications are running on the same host, this must be unique for each connection. This option also configures a TCP listening port for use with the rrdump tool. For more information on the rrdump tool, refer to the <i>ADS and ADH Software Installation Manuals</i> .
segmented.recvAddress	Configures the receive address or hostname to use in a segmented network configuration. All content is received on this recvAddress:recvServiceName pair. For multicast connections (RSSL_CONN_TYPE_RELIABLE_MULTICAST and RSSL_CONN_TYPE_SEQ_MCAST), you can specify multiple receive addresses using a comma-separated list.
segmented.recvServiceName	Configures the receive network's numeric port number or service name (as defined in etc/services file) to use in a segmented network configuration. All content is received on this recvAddress:recvServiceName pair.
segmented.sendAddress	Configures the send address or hostname to use in a segmented network configuration. All content is sent on this sendAddress:sendServiceName pair.

Table 22: **RsslConnectOptions.connectInfo** Options

Option	Description
segmented.sendServiceName	Configures the send network's numeric port number or service name (as defined in etc/services file) to use in a segmented network configuration. All content is sent on this sendAddress:sendServiceName pair.
segmented.interfaceName	A character representation of an IP address or hostname associated with the local network interface to use for sending and receiving content. This value is intended for use in systems which have multiple network interface cards, and if not specified the default network interface will be used.
segmented.unicastServiceName	Configures the numeric port number or service name (mapped to a port in etc/services file) to use for all unicast UDP traffic in a unified network configuration. This parameter is only required for multicast connection types (RSSL_CONN_TYPE_RELIABLE_MCAST and RSSL_CONN_TYPE_SEQ_MCAST). If multiple connections or applications are running on the same host, this must be unique for each connection. This option also configures a TCP listening port for use with the rrdump tool. For more information on the rrdump tool, refer to the <i>ADS and ADH Software Installation Manuals</i> .

Table 22: Rssi ConnectOptions. connectInfo Options (Continued)

10.3.2.3 RssiConnectOptions.tcpOpts Options

Option	Description
tcp_nodelay	If set to RSSL_TRUE , this disables Nagle's Algorithm for all accepted connections. Nagle's Algorithm allows more efficient use of TCP by delaying and combining small packets to reduce repeated overhead of TCP headers. Disabling Nagle's Algorithm can lead to lower latency by removing this delay, but can add increased bandwidth use as a result of the additional TCP header used with each small packet.

Table 23: Rssi ConnectOptions. tcpOpts Options

10.3.2.4 RssiConnectOptions.multicastOpts

Option	Description
disconnectOnGaps	Defaults to RSSL_FALSE , so if any multicast gap situation occur the underlying connection will not be closed. This allows the application to perform any item level recovery it may be able to do in order to reduce unnecessary bandwidth of full recovery on the multicast network. If set to RSSL_TRUE , the underlying connection will be closed if any multicast gap situation occurs. A multicast gap situation is reported as a return value of RSSL_RET_PACKET_GAP_DETECTED , RSSL_RET_SLOW_READER , or RSSL_RET_CONGESTION_DETECTED from rsslRead .
packetTTL	Controls the maximum number of components (network switches, etc.) a multicast datagram can traverse before it is removed from the network. Setting this to 0 , prevents packets from leaving the sending machine. When set to 255 , the packet is not limited in the number of components it can traverse and is not removed from the network.
nData	The maximum number of retransmissions that will be attempted for an unacknowledged point-to-point packet.

Table 24: Rssi ConnectOptions. multicastOpts Options

Option	Description
nrreq	Specifies the maximum number of retransmit requests that will be sent for a missing packet.
tdata	Specifies the time that RRCP must wait before retransmitting an unacknowledged point-to-point packet, in hundreds of milliseconds.
trreq	Specifies the minimum time that RRCP will wait before resending a retransmit request for a missed multicast packet, in hundreds of milliseconds.
twait	Specifies the time that RRCP will ignore additional retransmit requests for a data packet that it has already retransmitted, in hundreds of milliseconds. The time period starts with the receipt of the first request for retransmission.
tbchold	Specifies the maximum time that RRCP will hold a transmitted broadcast packet in case the packet needs to be retransmitted, in hundreds of milliseconds.
tpphold	Specifies the maximum time that RRCP will hold a transmitted point-to-point packet in case the packet needs to be retransmitted, in hundreds of milliseconds.
userQLimit	Specifies the maximum backlog of messages allowed on the channel's inbound message queue. Once this limit is exceeded RRCP will begin to discard messages until the backlog decreases. pktPoolLimitLow should be greater than three times userQLimit .
nmissing	Specifies the maximum number of missed consecutive multicast packets, from a particular node, from which RRCP will attempt to request retransmits.
pktPoolLimitHigh	Specifies the high-water mark for RRCP packet pool. If this limit is reached, no further RRCP packets will be allocated until and unless the usage falls below the low-water mark, the pktPoolLimitLow parameter.
pktPoolLimitLow	Specifies the low-water mark for RRCP packet pool. Additional RRCP packets will only be allocated if the usage falls from the high-water mark pktPoolLimitHigh to below this low-water mark value. pktPoolLimitLow should be greater than three times userQLimit .
hsmInterface	<p>A character representation of an IP address or hostname associated with the local network interface to use for sending host status message (HSM) packets. This value is intended for use in systems which have multiple network interface cards, and if not specified the default network interface will be used.</p> <p>This option is used to configure broadcasting of statistics messages to the Host Stat Message (HSM) Client tool. For more information on this tool, refer to the <i>ADS and ADH Software Installation Manuals</i>.</p>
hsmMultAddress	<p>Sets the multicast address over which to send host status message (HSM) packets.</p> <p>This option is used to configure broadcasting of statistics messages to the Host Stat Message (HSM) Client tool. For more information on this tool, refer to the <i>ADS and ADH Software Installation Manuals</i>.</p>
hsmPort	<p>Sets the multicast port on which to send host status message (HSM) packets.</p> <p>This option is used to configure broadcasting of statistics messages to the Host Stat Message (HSM) Client tool. For more information on this tool, refer to the <i>ADS and ADH Software Installation Manuals</i>.</p>

Table 24: **Rssi ConnectOptions.multicastOpts Options (Continued)**

Option	Description
hsmlInterval	The time interval over which HSM packets are sent, in seconds. Set this to 0 to disable sending host status messages. This setting may be adjusted by the <code>rrdump</code> tool (see the <code>unicastServiceName</code> option). This option is used to configure broadcasting of statistics messages to the Host Stat Message (HSM) Client tool. For more information on this tool, refer to the <i>ADS and ADH Software Installation Manuals</i> .
tcpControlPort	Specifies the port number to use when connecting <code>rrdump</code> (a monitoring tool available in the TREP Infrastructure Tools package). If set to or left as NULL, <code>tcpControlPort</code> uses the same port number as the <code>unicastServiceName</code> setting. If set to -1, a control port is not opened.
portRoamRange	Specifies the number of port numbers on which to attempt binding if the <code>unicastServiceName</code> fails to bind. Whichever port is specified in <code>unicastServiceName</code> is used as a starting point, with port numbers incrementing by 1 until it reaches the number specified in <code>portRoamRange</code> or successfully binds. If set to 0 , port roaming is disabled and the connection attempts to bind to only the <code>unicastServiceName</code> .

Table 24: `RsslConnectOptions.multicastOpts` Options (Continued)

10.3.2.5 `RsslConnectOptions.shmemOpts` Options

Option	Description
maxReaderLag	Maximum number of messages that the client can have waiting to be read. If the client "lags" the server by more than this amount, the client will be disconnected on its next attempt to read. The default is equal to 75% of the number of buffers in the shared memory segment.

Table 25: `RsslConnectOptions.shmemOpts` Options

10.3.2.6 `RsslConnectOptions.seqMulticastOpts` Options

Option	Description
maxMsgSize	Maximum size of messages that the SEQ_MCAST transport will read.
instanceId	The <code>instanceId</code> and originating IP address and port uniquely identify the sequenced multicast channel. When multiple applications run on the same host, unique <code>instanceId</code> values allow them to operate independently.

Table 26: `RsslConnectOptions.seqMultiCastOpts` Options

10.3.2.7 `RsslConnectOptions` Utility Function

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

Function Name	Description
<code>rsslClearConnectOpts</code>	Clears the <code>RsslConnectOptions</code> structure. Useful for structure reuse.

Table 27: `RsslConnectOptions` Utility Function

10.3.3 `rsslConnect` Outbound Connection Creation Example

The following example demonstrates basic `rsslConnect` use in a non-blocking manner. The application first populates the `RsslConnectOptions` and then attempts to connect. If the connection succeeds, the application then registers the `RsslChannel.socketId` with the I/O notification mechanism and continues with connection initialization (as described in Section 10.5).

```

RsslChannel *pChnl = 0;
RsslConnectOptions cOpts = RSSL_INIT_CONNECT_OPTS;
/* populate connect options, then pass to rsslConnect function - RSSL should already be
   initialized */

cOpts.connectionType = RSSL_CONN_TYPE_SOCKET; /* use standard socket connection */
cOpts.connectionInfo.unified.address = "localhost"; /* connect to server running on same
   machine */
cOpts.connectionInfo.unified.serviceName = "14002"; /* server is running on port number 14002
   */
cOpts.pingTimeout = 30; /* clients desired ping timeout is 30 seconds, pings should be sent
   every 10 */
cOpts.blocking = RSSL_FALSE; /* perform non-blocking I/O */
cOpts.compressionType = RSSL_COMP_NONE; /* client does not desire compression for this
   connection */

/* populate version and protocol with RWF information (found in rsslIterators.h) or protocol
   specific
   info */
cOpts.protocolType = RSSL_RWF_PROTOCOL_TYPE;
cOpts.majorVersion = RSSL_RWF_MAJOR_VERSION;
cOpts.minorVersion = RSSL_RWF_MINOR_VERSION;

if ((pChnl = rsslConnect(&cOpts, &error)) == 0)
{
    printf("Error %s (%d) (errno: %d) encountered with rsslConnect. Error Text: %s\n",
           rsslRetCodeToString(error.rsslErrorId), error.rsslErrorId, error.sysError,
           error.text);

    /* End application, uninitialized to clean up first */
    rsslUninitialize();
    return 0;
}

/* Connection was successful, add socketId to I/O notification mechanism and initialize
   connection */
/* Typical FD_SET use, this may vary depending on the I/O notification mechanism the
   application is using
   */
FD_SET(pChnl->socketId, &readfds);
FD_SET(pChnl->socketId, &exceptfds);
FD_SET(pChnl->socketId, &writefds);

```

```
/* Continue on with connection initialization process, refer to Section 10.5 for more details.  
 */
```

Code Example 5: Creating a Connection Using `rsslConnect`

10.3.4 Tunneling Connection Keep Alive

A client connection that is leveraging a `connectionType` of `RSSL_CONN_TYPE_HTTP_TUNNEL` or `RSSL_CONN_TYPE_ENCRYPTED` may be connecting through proxy devices as it tunnels through the Internet. Some proxy devices will force-close connections after certain elapsed time or time of day requirements are met. If one of these proxy devices is in a tunneling connections path, it can result in periodic connection loss. The RSSL Transport provides the `rsslReconnectClient` function which allows a tunneling client application to proactively create another connection and bridge data flow from the existing connection, which will be closed, to the new connection. An application can use this, along with knowledge of the proxy device's time requirements, to keep an application's connection alive beyond the time limits enforced by the proxy which helps to avoid data recovery scenarios. This function is not used to perform any kind of connection or data recovery after a connection is closed or disconnected or for any non-tunneled connection types.

10.4 Server Creation and Accepting Connections

10.4.1 Creating a Listening Socket

The Transport Package allows you to establish and manage listening sockets, typically associated with a server. Listening sockets can be leveraged to create an application that accepts connections created through the use of `rsslConnect`. Listening sockets are used mainly by OMM Interactive Provider applications and are typically established on a well-known port number (known by other connecting applications).

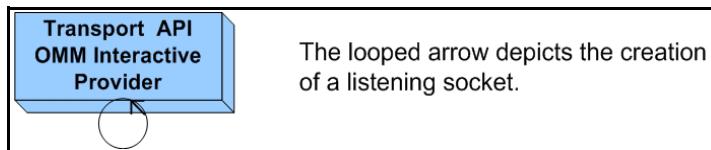


Figure 35. Transport API Server Creation

An application can create a listening socket connection by using the `rsslBind` function, described in the following table.

function NAME	DESCRIPTION
rsslBind	<p>Establishes a listening socket connection, which supports connections from standard socket and HTTP <code>rsslConnect</code> users. Returns an <code>RsslServer</code> that represents the listening socket connection to the user. In the event of an error, NULL is returned and additional information can be found in the <code>RsslError</code> structure.</p> <p>Options are passed in via an <code>RsslBindOptions</code> structure described in Section 10.4.1.1.</p> <p>Once a listening socket is established, this <code>RsslServer</code> can begin accepting connections. For more information, refer to Section 10.4.2.</p>

Table 28: `rsslBind` Function

10.4.1.1 RsslBindOptions Structure Members

Structure Member	DESCRIPTION
serviceName	A character representation of a numeric port number or service name (as defined in the <code>etc/services</code> file) on which to bind and open a listening socket.
interfaceName	A character representation of an IP address or hostname for the local network interface to which to bind. The RSSL Transport will establish connections on the specified interface. This value is intended for use in systems which have multiple network interface cards. If not populated, a connection can be accepted on all interfaces ^a . If the loopback address (127.0.0.1) is specified, connections can be accepted only when instantiating from the local machine ^b .
maxFragmentSize	The maximum size buffer that will be written to the network. If a larger buffer is required, the RSSL Transport will internally fragment the larger buffer into smaller <code>maxFragmentSize</code> buffers. This is different from application level message fragmentation done via the Message Package (as discussed in Section 13.1). Any guaranteed, shared, or input buffers created will use this size. This value is passed to all connected client applications and enforces a common message size between components. For more information about RSSL Transport buffer fragmentation, refer to Section 10.9.

Table 29: `RsslBindOptions` Structure Members

Structure Member	DESCRIPTION
numInputBuffers	The number of sequential input buffers used by each <code>RsslChannel</code> for data reading. This controls the maximum number of bytes that can be handled with a single network read operation on each channel. Each input buffer will be created to contain <code>maxFragmentSize</code> bytes. Input buffers are allocated at initialization time.
guaranteedOutputBuffers	A guaranteed number of buffers made available for each <code>RsslChannel</code> to use while writing data. Each buffer is created to contain <code>maxFragmentSize</code> bytes. Guaranteed output buffers are allocated at initialization time. For more information, refer to Section 10.8.
	Note: For <code>RSSL_CONN_TYPE_UNIDIR_SHMEM</code> , this parameter determines the number of buffers in the shared memory segment. The size of the shared memory segment will approximate <code>guaranteedOutputBuffers * maxFragmentSize</code> .
maxOutputBuffers	The maximum number of output buffers allowed for use by each <code>RsslChannel</code> . (<code>maxOutputBuffers - guaranteedOutputBuffers</code>) is equal to the number of shared pool buffers that each <code>RsslChannel</code> is allowed to use. Shared pool buffers are only used if all <code>guaranteedOutputBuffers</code> are unavailable. If equal to the <code>guaranteedOutputBuffers</code> value, no shared pool buffers are available.
sharedPoolSize	The maximum number of buffers to make available as part of the shared buffer pool. The shared buffer pool can be drawn upon by any connected <code>RsslChannel</code> , where each channel is allowed to use up to (<code>maxOutputBuffers - guaranteedOutputBuffers</code>) number of buffers. Each shared pool buffer will be created to contain <code>maxFragmentSize</code> bytes. If set to 0 , a default of 1,048,567 shared pool buffers will be allowed. The shared pool is not fully allocated at bind time. As needed, shared pool buffers are added and reused until the server is shut down. For more information, refer to Section 10.8.
	Note: It is considered an invalid configuration to allow more shared pool buffers (<code>maxOutputBuffers - guaranteedOutputBuffers</code>) than the <code>sharedPoolSize</code> . If this happens, an error is returned from <code>rsslBind</code> .
sharedPoolLock	If set to <code>RSSL_TRUE</code> , the shared buffer pool will have its own locking performed. This setting is independent of any other locking mode options. Enabling a shared pool lock allows shared pool use to remain thread safe while still disabling channel locking. For more information, refer to Section 10.2.3.
pingTimeout	The servers desired ping timeout value. This may change through the negotiation process between the client and the server. After the connection becomes active, the actual negotiated value becomes available through the <code>pingTimeout</code> value on the <code>RsslChannel</code> . When determining the desired ping timeout, the rule of thumb is to send a heartbeat every <code>pingTimeout/3</code> seconds. For more information, refer to Section 10.12.
minPingTimeout	The server's lowest allowable ping timeout value. This is the lowest possible value allowed in the negotiation between client and servers <code>pingTimeout</code> values. After the connection becomes active, the actual negotiated value becomes available through the <code>pingTimeout</code> value on the <code>RsslChannel</code> . When determining the desired ping timeout, the rule of thumb is to send a heartbeat every <code>pingTimeout/3</code> seconds. For more information, refer to Section 10.12.
serverToClientPings	If set to <code>RSSL_TRUE</code> , heartbeat messages are required to flow from the server to the client. If set to <code>RSSL_FALSE</code> , the server is not required to send heartbeats. TREP and other Thomson Reuters components typically require this to be set to <code>RSSL_TRUE</code> .

Table 29: `RsslBindOptions` Structure Members (Continued)

Structure Member	DESCRIPTION
clientToServerPings	If set to RSSL_TRUE , heartbeat messages are required to flow from the client to the server. If set to RSSL_FALSE , the client is not required to send heartbeats. TREP and other Thomson Reuters components typically require this to be set to RSSL_TRUE .
compressionType	The type of compression the server wants to apply for this connection. Compression is negotiated between the client and server and may not be performed if only the server has this enabled. The server can force compression, regardless of client settings, by using the forceCompression option. For more information about supported compression types and compression negotiation, refer to Section 10.4.3.
compressionLevel	<p>Sets the level of compression to apply. Allowable values are 0 to 9.</p> <ul style="list-style-type: none"> • A compressionLevel of 1 results in the fastest compression. • A compressionLevel of 9 results in the best compression. • A compressionLevel of 6 is a compromise between speed and compression. • A compressionLevel of 0 will copy the data with no compression applied. <p>For more information on supported compression levels, refer to Section 10.4.3.</p>
forceCompression	If set to RSSL_TRUE , this forcibly enables compression, regardless of client preference. When enabled, compression will use the compressionType and compressionLevel specified by the server. If set to RSSL_FALSE , compression is negotiated between the client and server. For more information about supported compression types and compression negotiation, refer to Section 10.4.3.
tcp_nodelay	DEPRECATED. To configure equivalent functionality, refer to tcpOpts.tcp_nodelay in Table 23.
serverBlocking	<p>If set to RSSL_TRUE, blocking I/O will be used for this RsslServer. When I/O is used in a blocking manner on an RsslServer, the rsslAccept function will complete any initialization on the RsslChannel prior to returning it. Blocking I/O prevents the application from performing any operations until the I/O operation is completed.</p> <p>Blocking I/O is typically not recommended. An application can leverage an I/O notification mechanism to allow efficient use, while using other cycles to perform other necessary work in the application.</p>
channelsBlocking	<p>If set to RSSL_TRUE, blocking I/O will be used for all connected RsslChannel structures. When I/O is used in a blocking manner on an RsslChannel, any reading or writing will complete before control is returned to the application. Blocking I/O prevents the application from performing any operations until the I/O operation is completed.</p> <p>Blocking I/O is typically not recommended. An application can leverage an I/O notification mechanism to allow efficient reading and writing, while using other cycles to perform other necessary work in the application. An I/O notification mechanism enables the application to read when data is available, and write when output space is available.</p>

Table 29: **RsslBindOptions** Structure Members (Continued)

Structure Member	DESCRIPTION
protocolType	<p>Sets the protocol type that the server uses on its connections. The server rejects connections from clients that do not use the specified <code>protocolType</code>. When an <code>RsslChannel</code> becomes active for a client or server, this information becomes available via the <code>protocolType</code> on the <code>RsslChannel</code>.</p> <p>The transport layer is data-neutral and allows the flow of any type of content. <code>protocolType</code> is provided to help client and server applications manage the information they communicate. For more details, refer to Section 9.5.1.</p>
majorVersion	<p>Specifies the major version of the protocol supported by the server. The actual major version used is negotiated with the client at connection time. The outcome of the negotiation is provided via <code>majorVersion</code> on the <code>RsslChannel</code>. Typically, the major version increases with the introduction of a significant (i.e., incompatible) change.</p> <p>The transport layer is data-neutral and allows the flow of any type of content. <code>majorVersion</code> is provided to help client and server applications manage the information they communicate. For more details, refer to Section 9.5.1.</p>
minorVersion	<p>The minor version of the protocol supported by the server. The actual minor version used is negotiated with the client at connection time. The outcome of the negotiation is provided via <code>minorVersion</code> on the <code>RsslChannel</code>. Typically, the minor version increases with the introduction of a fully backward-compatible change or extension.</p> <p>The transport layer is data-neutral and allows the flow of any type of content. <code>minorVersion</code> is provided to help client and server applications manage the information they communicate. For more details, refer to Section 9.5.1.</p>
sysRecvBufSize	<p>Sets the system's receive buffer size for this connection. A missing value, or a setting of 0 sets the buffer to the default size of 64K. Setting <code>sysSendBufSize</code> is done via <code>RsslAcceptOptions</code> (for details, refer to Table 23).</p> <p>This can also be set or changed via <code>rsslIoctl</code> for values less than or equal to 64K. For values larger than 64K, you must use this method to set <code>sysRecvBufSize</code> prior to the bind system call.</p>
userSpecPtr	<p>A pointer that can be set by the application. This value is not modified by the transport, but is preserved and stored in the <code>userSpecPtr</code> of the <code>RsslServer</code> returned from <code>rsslBind</code> if a <code>userSpecPtr</code> was not specified in the <code>RsslAcceptOptions</code>. This information can be useful for coupling this <code>RsslServer</code> with other user-created information, such as a list of connected <code>RsslChannel</code> structures.</p>
tcpOpts	<p>A substructure containing options specific to TCP-based connection types (i.e., these settings are used for <code>RSSL_CONN_TYPE_SOCKET</code> and <code>RSSL_CONN_TYPE_HTTP</code>). For information about specific options, refer to Table 23.</p>
componentVersion	<p>An optional, user-defined component version string appended behind the standard UPA component version information. If the combined component version length exceeds the maximum supported by the Transport API, the user-defined information will be truncated.</p>

Table 29: `RsslBindOptions` Structure Members (Continued)

- a. `INADDR_ANY` is used
- b. `INADDR_LOOPBACK` is used

10.4.1.2 RsslBindOptions.tcpOpts Structure Members

Option	DESCRIPTION
tcp_nodelay	If set to RSSL_TRUE , this disables Nagle's Algorithm for all accepted connections. Nagle's Algorithm allows more efficient use of TCP by delaying and combining small packets to reduce repeated overhead of TCP headers. Disabling Nagle's Algorithm can lead to lower latency by removing this delay, but can add increased bandwidth use as a result of the additional TCP header used with each small packet.

Table 30: **RsslBindOptions.tcpOpts** Options

10.4.1.3 RsslBindOptions Utility Function

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

Option	DESCRIPTION
rsslClearBindOpts	Clears the RsslBindOptions structure. Useful for structure reuse.

Table 31: **RsslBindOptions** Utility Function

10.4.1.4 rsslBind Listening Socket Connection Creation Example

The following example demonstrates basic **rsslBind** use in a non-blocking manner. The application first populates the **RsslBindOptions** and then attempts to create a listening socket. If the bind succeeds, the application then registers the **RsslServer.socketId** with the I/O notification mechanism and waits to be alerted of incoming connection attempts. For more details on accepting or rejecting incoming connection attempts, refer to Section 10.4.2.

```
RsslServer *pSrvr = 0;
RsslBindOptions b0pts = RSSL_INIT_BIND_OPTS;
/* populate bind options, then pass to rsslBind function - RSSL should already be initialized */
 */

b0pts.serviceName = "14002"; /* server is running on port number 14002 */
b0pts.pingTimeout = 45; /* servers desired ping timeout is 45 seconds, pings should be sent
   every 15 */
b0pts.minPingTimeout = 30; /* min acceptable ping timeout is 30 seconds, pings should be sent
   every 10 */

/* set up buffering, configure for shared and guaranteed pools */
b0pts.guaranteedOutputBuffers = 1000;
b0pts.maxOutputBuffers = 2000;
b0pts.sharedPoolSize = 50000;
b0pts.sharedPoolLock = RSSL_TRUE;

b0pts.serverBlocking = RSSL_FALSE; /* perform non-blocking I/O */
b0pts.channelsBlocking = RSSL_FALSE; /* perform non-blocking I/O */
b0pts.compressionType = RSSL_COMP_NONE; /* server does not desire compression for this
   connection */
```

```

/* populate version and protocol with RWF information (found in rsslIterators.h) or protocol
   specific
   info */

bOpts.protocolType = RSSL_RWF_PROTOCOL_TYPE;
bOpts.majorVersion = RSSL_RWF_MAJOR_VERSION;
bOpts.minorVersion = RSSL_RWF_MINOR_VERSION;

if ((pSrvr = rsslBind(&bOpts, &error)) == 0)
{
    printf("Error %s (%d) (errno: %d) encountered with rsslBind. Error Text: %s\n",
           rsslRetCodeToString(error.rsslErrorId), error.rsslErrorId, error.sysError,
           error.text);

    /* End application, uninitialized to clean up first */
    rsslUninitialize();
    return 0;
}

/* Connection was successful, add socketId to I/O notification mechanism and wait for
   connections */
/* Typical FD_SET use, this may vary depending on the I/O notification mechanism the
   application is using
 */
FD_SET(pSrvr->socketId, &readfds);
FD_SET(pSrvr->socketId, &exceptfds);

/* Use rsslAccept for incoming connections, read and write data to established connections,
   etc */

```

Code Example 6: Creating a Listening Socket Using `rsslBind`

10.4.2 Accepting Connection Requests

After establishing a listening socket, the `RsslServer.socketId` can be registered with an I/O notification mechanism. An alert from the I/O notification mechanism on the server's `socketId` indicates that a connection request has been detected. An application can begin the process of accepting or rejecting the connection by using the `rsslAccept` function.

function NAME	DESCRIPTION
<code>rsslAccept</code>	<p>Uses the <code>RsslServer</code> that represents the listening socket connection and begins the process of accepting the incoming connection request. Returns an <code>RsslChannel</code> that represents the client connection. In the event of an error, NULL is returned and additional information can be found in the <code>RsslError</code> structure.</p> <p>The <code>rsslAccept</code> function can also begin the rejection process for a connection through the use of the <code>RsslAcceptOptions</code> structure as described in Section 10.4.2.1.</p> <p>Once a connection is established and transitions to <code>RSSL_CH_STATE_ACTIVE</code>, this <code>RsslChannel</code> can be used for other transport operations. For more information about channel initialization, refer to Section 10.5.</p>

Table 32: `rsslAccept` Function

10.4.2.1 RsslAcceptOptions Structure Member

STRUCTURE MEMBER	DESCRIPTION
<code>nakMount</code>	Indicates that the server wants to reject the incoming connection. This may be due to some kind of connection limit being reached. For non-blocking connections to successfully complete rejection, the initialization process must still be completed. For more information about channel initialization, refer to Section 10.5.
<code>userSpecPtr</code>	A pointer that can be set by the application. This value is not modified by the transport, but will be preserved and stored in the <code>userSpecPtr</code> of the <code>RsslChannel</code> returned from <code>rsslAccept</code> . If this value is not set, the <code>RsslChannel.userSpecPtr</code> will be set to the <code>userSpecPtr</code> associated with the <code>RsslServer</code> that is accepting this connection.

Table 33: `RsslAcceptOptions` Structure Members

10.4.2.2 RsslAcceptOptions Utility Function

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

FUNCTION NAME	DESCRIPTION
<code>rsslClearAcceptOpts</code>	Clears the <code>RsslAcceptOptions</code> structure. Useful for structure reuse.

Table 34: `RsslAcceptOptions` Utility Functions

10.4.2.3 `rsslAccept` Accepting Connection Example

The following example demonstrates basic `rsslAccept` use. The application first populates the `RsslAcceptOptions` and then attempts to accept the incoming connection request. If the accept succeeds, the application then registers the new `RsslChannel.socketId` with the I/O notification mechanism and continues with connection initialization, described in Section 10.5.

```

/* Accept is typically called when servers socketId indicates activity */
RsslChannel *pChnl = 0;
RsslAcceptOptions aOpts = RSSL_INIT_ACCEPT_OPTS;
/* populate accept options, then pass to rsslAccept function - RSSL should already be
   initialized */

aOpts.nakMount = RSSL_FALSE; /* allow the connection */

if ((pChnl = rsslAccept(pSrvr, &aOpts, &error)) == 0)
{
    printf("Error %s (%d) (errno: %d) encountered with rsslAccept. Error Text: %s\n",
           rsslRetCodeToString(error.rsslErrorId), error.rsslErrorId, error.sysError,
           error.text);

    /* End application, uninitialized to clean up first */
    rsslUninitialize();
    return 0;
}

/* Connection was successful, add socketId to I/O notification mechanism and initialize
   connection */
/* Typical FD_SET use, this may vary depending on the I/O notification mechanism the
   application is using
 */
FD_SET(pChnl->socketId, &readfds);
FD_SET(pChnl->socketId, &exceptfds);

/* Continue on with connection initialization process, refer to Section 10.5 for more details
 */

```

Code Example 7: Accepting Connection Attempts using `rsslAccept`

10.4.3 Compression Support

As mentioned, the RSSL Transport supports the use of data compression. The client and server negotiate compression behavior during the connection establishment process, with the server determining supported compression methods by referencing the `RsslBindOptions.compressionType` parameter (refer to the ENUMs specified in Section 10.4.3.1).

Additionally:

- You can configure the server to support multiple compression types by including the appropriate bitmasks.
- When using zlib, you can configure the quality of compression by setting the `RsslBindOptions.compressionLevel` parameter.

A client requests compression by setting the `RsslConnectOptions.compressionType` parameter to one of the values specified in Section 10.4.3.1. If the client's configured compression type matches one of the types specified by the server, that compression type is used for the connection. After establishing a connection, the server or client can verify at any time the type of compression in use on a channel by calling `rsslGetChannelInfo` (refer to Section 10.14).

The server may also force compression for its connections by enabling the `RsslBindOptions.forceCompression` parameter, in which case the server's `compressionType` is used regardless of the client's configuration.

Note: If you set the server to force compression, use only one compression type in `RsslBindOptions.compressionType`.

10.4.3.1 Compression Types

The Transport API supports the following compression options:

ENUMERATED NAME	COMPRESSION LEVEL SUPPORTED	DEFAULT COMPRESSION THRESHOLD	DESCRIPTION
RSSL_COMP_NONE	n/a	n/a	No compression.
RSSL_COMP_ZLIB	Yes	30 bytes	Use zlib compression . Zlib, an open source utility, employs a variation of the LZ77 algorithm to compress and decompress data.
RSSL_COMP_LZ4	No	300 bytes	Use LZ4 compression. LZ4, an open source utility, employs a variation of the LZ77 algorithm to compress and decompress data. Note: Though LZ4 compression consumes less CPU than Zlib, LZ4 does not achieve the same reduction in size.

Table 35: RSSL Compression Types

10.4.3.2 Compression Level

The server's specified `compressionLevel` determines the quality of the compression, where:

- Lower values favor faster compression
- Higher values compress data into smaller sizes

Currently only zlib supports the use of compression levels.

10.4.3.3 Compression Threshold

Different compression types have different behaviors and compression efficiency can vary depending on buffer size. Because small buffer sizes might not compress well, the Transport API uses a compression threshold such that all buffers exceeding the threshold size are compressed. Default compression thresholds are specified in Section 10.4.3.1. You can change this threshold via the `rsslIoctl` function (refer to Section 10.14).

If a message is larger than the compression threshold, you can prevent its compression through the use of the `rsslWrite RSSL_WRITE_DO_NOT_COMPRESS` flag. For more information, refer to Section 10.9.

10.5 Channel Initialization

After an `RsslChannel` is returned from the client's `rsslConnect` or server's `rsslAccept` call, the channel may need to continue the initialization process using the `rsslInitChannel` function.

Note: For both client and server channels, to complete the channel initialization process, more than one call to `rsslInitChannel` might be required.

Additional initialization is required as long as the `RsslChannel.state` is `RSSL_CH_STATE_INITIALIZING`.

- If using a non-blocking I/O, this is the typical state from which an `RsslChannel` starts and multiple initialization calls might be needed to transition to active.
- If using a blocking I/O, when successful, `rsslConnect` and `rsslAccept` return a completely initialized channel in an active state.

Internally, the RSSL initialization process involves several actions. The initialization includes any necessary RSSL connection handshake exchanges, including any HTTP or HTTPS negotiation. Compression, ping timeout, and versioning related negotiations also take place during the initialization process. This process involves exchanging several messages across the connection, and once all message exchanges have completed the `RsslChannel.state` will transition.

- If the connection is accepted (i.e., all negotiations were successful), the `RsslChannel.state` will become `RSSL_CH_STATE_ACTIVE`.
- If the connection is rejected (i.e., due to either failed negotiation or an `RsslServer` rejection of the connection by setting `nakMount` to `RSSL_TRUE`), the `RsslChannel.state` will become `RSSL_CH_STATE_CLOSED`, and the application should close the channel to clean up any associated resources.

10.5.1 `rsslInitChannel` Function

function NAME	DESCRIPTION
<code>rsslInitChannel</code>	<p>Continues initialization of an <code>RsslChannel</code>. This channel could originate from <code>rsslConnect</code> or <code>rsslAccept</code>. This function exchanges various messages to perform necessary RSSL negotiations and handshakes to complete channel initialization. If using blocking I/O, this function is typically not used because <code>rsslConnect</code> and <code>rsslAccept</code> return active channels. Requires the use of the <code>RsslInProgInfo</code> structure, refer to Section 10.5.2.</p> <p>The <code>RsslChannel</code> can be used for all additional transport functionality (e.g. reading, writing) after the <code>state</code> transitions to <code>RSSL_CH_STATE_ACTIVE</code>. If a connection is rejected or initialization fails, the state transitions to <code>RSSL_CH_STATE_CLOSED</code>, and the application should close the channel to clean up any associated resources.</p> <p>The return values are described in Section 10.5.4.</p>

Table 36: `rsslInitChannel` Function

10.5.2 RsslInProgInfo Structure

Use the `RsslInProgInfo` structure with the `rsslInitChannel` function to initialize a channel.

In certain circumstances, the initialization process might need to create new or additional underlying connections. If this occurs, the application must unregister the previous `socketId` and register the new `socketId` with the I/O notification mechanism in use with associated information being conveyed by `RsslInProgInfo` and `RsslInProgFlags`.

Structure Member	DESCRIPTION
flags	<p>Combination of bit values to indicate special behaviors and presence of optional <code>RsslInProgInfo</code> content.</p> <p><code>flags</code> uses the following enumeration values:</p> <ul style="list-style-type: none"> • RSSL_IP_NONE: Indicates that channel initialization is still in progress and subsequent calls to <code>rsslInitChannel</code> are needed for completion. The call did not change the <code>socketId</code>. • RSSL_IP_FD_CHANGE: Indicates that the call changed the <code>socketId</code>. The previous <code>socketId</code> is now stored in <code>RsslInProgInfo.oldSocket</code> so it can be unregistered with the I/O notification mechanism. The new <code>socketId</code> is stored in <code>RsslInProgInfo.newSocket</code> so it can be registered with the I/O notification mechanism. However, channel initialization is still in progress and subsequent calls to <code>rsslInitChannel</code> are needed to complete it.
oldSocket	Populated if flags indicate that <code>rsslInitChannel</code> needs to perform a file descriptor change. If this occurs, the <code>oldSocket</code> contains the <code>socketId</code> associated with the previous connection so the application can unregister this with the I/O notification mechanism.
newSocket	Populated if flags indicate that <code>rsslInitChannel</code> needs to perform a file descriptor change. If this occurs, the <code>newSocket</code> contains the <code>socketId</code> associated with the new connection so the application can register this with the I/O notification mechanism.

Table 37: `RsslInProgInfo` Structure Members

10.5.3 Calling `rsslInitChannel`

Typically, calls to `rsslInitChannel` are driven by I/O on the connection, however this can also be accomplished by using a timer to periodically call the function or looping on a call until the channel transitions to active or a failure occurs. Other than any overhead associated with the function call, there is no harm in calling `rsslInitChannel` more frequently than required. If work is not required, the function returns, indicating that the connection is still in progress.

If using I/O, a client application should register the `RsslChannel.socketId` with the read, write, and exception file descriptor sets. When the write descriptor alerts the user that the `socketId` is ready for writing, `rsslInitChannel` is called (this sends the initial connection handshake message). When the read file descriptor alerts the user that the `socketId` has data to read, `rsslInitChannel` is called - this typically reads the next portion of the handshake. This process would continue until the connection is active.

A server application would typically register the `RsslChannel.socketId` with the read and exception file descriptor sets. When the read descriptor alerts the user that the `socketId` has data to read, `rsslInitChannel` is called, which typically reads the initial portion of the handshake and sends out any necessary response. This process continues until the connection is active.

10.5.4 `rsslInitChannel` Return Codes

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

RETURN CODE	DESCRIPTION
RSSL_RET_SUCCESS	Indicates the initialization process completed successfully. The <code>RsslChannel.state</code> should be <code>RSSL_CH_STATE_ACTIVE</code> .
RSSL_RET_FAILURE	Indicates that initialization has failed and cannot progress. The <code>RsslChannel.state</code> should be <code>RSSL_CH_STATE_CLOSED</code> , and the application should close the channel to clean up associated resources. For more details, refer to the <code>RsslError</code> content.
RSSL_RET_CHAN_INIT_IN_PROGRESS	Indicates that initialization is still in progress. Check <code>RsslInProgInfo.flags</code> to determine whether the <code>socketId</code> changed. The <code>RsslChannel.state</code> should be <code>RSSL_CH_STATE_INITIALIZING</code> .
RSSL_RET_CHAN_INIT_REFUSED	Indicates the connection was rejected. For more details, refer to the <code>RsslError</code> content.
RSSL_RET_INIT_NOT_INITIALIZED	Indicates that the RSSL Transport is not initialized. For more details, refer to the <code>RsslError</code> content. For information on initializing, refer to Section 10.2.

Table 38: `rsslInitChannel` Return Codes

10.5.5 `rsslInitChannel` Example

The example below shows general use of `rsslInitChannel`. Use of I/O notification is assumed, and the example assumes that the code is being executed due to some I/O notification.

```
/* rsslInitChannel is typically called based on activity on the socketId, though a timer or
   looping can be used - the rsslInitChannel function should continue to be called until the
   connection becomes active, at which point reading and writing can begin */
RsslInProgInfo inProgInfo = RSSL_INIT_IN_PROG_INFO;
if (pChnl->state == RSSL_CH_STATE_INITIALIZING)
{
    if ((retCode = rsslInitChannel(pChnl, &inProgInfo, &error)) < RSSL_RET_SUCCESS)
    {
        printf("Error %s (%d) (errno: %d) encountered with rsslInitChannel. Error Text: %s\n",
               rsslRetCodeToString(error.rsslErrorId), error.rsslErrorId, error.sysError,
               error.text);
    }
    else
    {
        /* Handle return code appropriately */
        switch (retCode)
        {
            case RSSL_RET_CHAN_INIT_IN_PROGRESS:
            {
                /* Initialization is still in progress, check the RsslInProgInfo for additional
                   information */
                if (inProgInfo.flags & RSSL_IP_FD_CHANGE)
                {

```

```

        /* File descriptor has changed, unregister old and register new */
        FD_CLR(inProgInfo.oldSocket, &readfds);
        FD_CLR(inProgInfo.oldSocket, &writefds);
        FD_CLR(inProgInfo.oldSocket, &exceptionfds);
        /* newSocket should equal pChnl->socketId */
        FD_SET(inProgInfo.newSocket, &readfds);
        FD_SET(inProgInfo.newSocket, &writefds);
        FD_SET(inProgInfo.newSocket, &exceptionfds);
    }
}
break;
case RSSL_RET_SUCCESS:
printf("Channel on fd %d is now active - reading and writing can begin.\n",
       pChnl->socketId);
break;
default:
printf("Unexpected return code (%d) encountered!", retCode);
/* Likely unrecoverable, connection should be closed */
break;
}
}
}
}

```

Code Example 8: `RsslChannel` Initialization Process Using `rsslInitChannel`

10.6 Reading Data

When a client or server `RsslChannel.state` is `RSSL_CH_STATE_ACTIVE`, an application can receive data from the connection by calling `rsslRead`. The arrival of this data is often announced by the I/O notification mechanism with which the `RsslChannel.socketId` is registered. The RSSL Transport reads data from the network as a byte stream, after which it determines `RsslBuffer` boundaries and returns each buffer one by one. The `numInputBuffers` connect or bind option controls the maximum length of the byte stream that the transport can internally process with each network read.

Note: When an `RsslBuffer` is returned from `rsslRead`, the contents are only valid until the next call to `rsslRead`.

To reduce potentially unnecessary copies, returned information simply points into the internal `rsslRead` input buffer. If the application requires the contents of the buffer beyond the next `rsslRead` call, the application can copy the contents of the buffer and allow the user to control the duration of the life cycle of the memory.

If the connection uses compression, the `rsslRead` function will perform any necessary decompression prior to returning information to the application. For available compression types, refer to Section 10.4.3.

It is possible for `rsslRead` to succeed and return a NULL buffer. When this occurs, it indicates that a portion of a fragmented buffer has been received. The Transport Package internally reassembles all parts of the fragmented buffer and after processing the last fragment, returns the entire buffer to the user through `rsslRead`.

If a packed buffer is received, each call to `rsslRead` returns an individual message (i.e., portion of contents) from the packed buffer. Every subsequent call to `rsslRead` continues to return portions of the packed buffer until the buffer is emptied.

Message packing is transparent to the application that receives a packed buffer. For more information about packing, refer to Section 10.11.

10.6.1 `rsslRead` Function

function Name	DESCRIPTION
RsslChannel	<p>Provides the user with data received from the connection. This function expects the <code>RsslChannel</code> to be in the active state. When data is available, an <code>RsslBuffer</code> referring to the information is returned, which is valid until the next call to <code>rsslRead</code>. If a blocking I/O is used, the <code>rsslRead</code> function will not return until there is information to return or an error has occurred.</p> <p>A return code parameter passed into the function is used to convey return code information as well as communicate whether there is additional information to read. An I/O notification mechanism may not inform the user of this additional information as it has already been read from the socket and is contained in the <code>rsslRead</code> input buffer.</p> <p>Return values are described in Section 10.6.2.</p>

Table 39: `Rssl Channel` Function

10.6.2 `rsslRead` Return Codes

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

RETURN CODE	BUFFER CONTENTS	DESCRIPTION
RSSL_RET_SUCCESS	Populated if the full buffer is available, NULL otherwise. The buffer's <code>length</code> indicates the number of bytes to which the <code>data</code> refers.	Indicates that the <code>rsslRead</code> call was successful and there are no remaining bytes in the input buffer. The I/O notification mechanism will notify the user when additional information arrives. The ping timer should be updated, refer to Section 10.12.
Any positive value > 0	Populated if full buffer is available, NULL otherwise. The buffer's <code>length</code> indicates the number of bytes to which the <code>data</code> refers.	Indicates that the <code>rsslRead</code> call was successful and there are remaining bytes in the input buffer. The I/O notification mechanism will not notify the user of these bytes. The <code>rsslRead</code> function should be called again to ensure that the remaining bytes are processed. The ping timer should be updated (for details, refer to Section 10.12).
RSSL_RET_READ_WOULD_BLOCK	NULL	Note: If there are additional bytes to process, you should call <code>rsslRead</code> again. Because the bytes are already contained in the transport input buffer, an I/O notification mechanism will not alert the user of their presence.

Table 40: `rssl Read` Return Codes

RETURN CODE	BUFFER CONTENTS	DESCRIPTION
RSSL_RET_READ_PING	NULL	Indicates that a heartbeat message was received. The ping timer should be updated (for details, refer to Section 10.12).
RSSL_RET_FAILURE	NULL	Indicates a failure condition, often that the connection is no longer available. The RsslChannel should be closed (for details, refer to Section 10.13). For more details, refer to RsslError content.
RSSL_RET_PACKET_GAP_DETECTED	NULL	Indicates that a packet gap was detected in the inbound transport content. This may be recoverable above the transport layer, so the RsslChannel is left in a connected state. If needed, an application can configure the transport to disconnect whenever this occurs by using the disconnectOnGaps option. For details on this option, refer to Section 10.3.2.4.
RSSL_RET_SLOW_READER	NULL	Indicates that the reader is not keeping up with the data rate and a packet gap was detected in the inbound transport content. This may be recoverable above the transport layer, so the RsslChannel is left in a connected state. If needed, an application can configure the transport to disconnect whenever this occurs by using the disconnectOnGaps option. For details on this option, refer to Section 10.3.2.4.
RSSL_RET_CONGESTION_DETECTED	NULL	Indicates network congestion and that a gap was detected in the inbound transport content. This may be recoverable above the transport layer, so the RsslChannel is left in a connected state. If needed, an application can configure the transport to disconnect whenever this occurs by using the disconnectOnGaps option. For details on this option, refer to Section 10.3.2.4.
RSSL_RET_READ_FD_CHANGE	NULL	Indicates that the connections socketId has changed. This can occur as a result of internal connection keep-alive mechanisms. The previous socketId is stored in the RsslChannel.oldSocket so it can be removed from the I/O notification mechanism. The RsslChannel.oldSocket contains the new file descriptor, which should be registered with the I/O notification mechanism.
RSSL_RET_READ_IN_PROGRESS	NULL	Indicates that an rsslRead call on the RsslChannel is already in progress. This can be due to another thread performing the same operation.
RSSL_RET_INIT_NOT_INITIALIZED	NULL	Indicates that the RSSL Transport has not been initialized. See the RsslError content for more details. For information on initializing, refer to Section 10.2.

Table 40: [rssl Read](#) Return Codes (Continued)

10.6.3 rsslRead Example

The following example shows typical use of `rsslRead` and assumes use of an I/O notification mechanism. This code would be similar for client or server based `RsslChannel` structures.

```

/* rsslRead use, be sure to keep track of the return values from read so data is not stranded
   in the
   input buffer */
RsslRet retCode = RSSL_RET_FAILURE;
RsslBuffer *pBuffer = 0;

if ((pBuffer = rsslRead(pChnl, &retCode, &error)) != 0)
{
    /* if a buffer is returned, we have data to process and code is success */
    /* Process data and update ping monitor (Section 10.12) since data was received */

    /* Check the return code to determine whether more data is available to read */
    if (retCode > RSSL_RET_SUCCESS)
    {
        /* There is more data to read and process and I/O notification may not trigger for it */
        /* Either schedule another call to read or loop on read until retCode ==
           RSSL_RET_SUCCESS */
        /* and there is no data left in internal input buffer */
    }
}
else
{
    /* Handle return codes appropriately, not all return values are failure conditions */
    switch(retCode)
    {
        case RSSL_RET_SUCCESS:
        {
            /* There is more data to read and process and I/O notification may not trigger for it */
            /* Either schedule another call to read or loop on read until retCode ==
               RSSL_RET_SUCCESS */
            /* and there is no data left in internal input buffer */
        }
        case RSSL_RET_READ_PING:
        {
            /* Update ping monitor (Section 10.12) */
        }
        break;
        case RSSL_RET_READ_FD_CHANGE:
        {
            /* File descriptor changed, typically due to tunneling keep-alive */
            /* Unregister old socketId and register new socketId */
            FD_CLR(pChnl->oldSocketId, &readfds);
        }
    }
}

```

```

FD_CLR(pChnl->oldSocketId, &writefds);
FD_CLR(pChnl->oldSocketId, &exceptionfds);
/* Up to application whether to register with write set - depends on need for write
notification */
FD_SET(pChnl->socketId, &readfds);
FD_SET(pChnl->socketId, &exceptionfds);
}
break;
case RSSL_RET_READ_WOULD_BLOCK: /* Nothing to read */
case RSSL_RET_READ_IN_PROGRESS: /* Reading from multiple threads - this is dangerous */
{
    /* Handle as application sees fit, output warning, etc */
}
break;
case RSSL_RET_INIT_NOT_INITIALIZED:
case RSSL_RET_FAILURE:
{
    printf("Error %s (%d) (errno: %d) encountered with rsslRead. Error Text: %s\n",
        rsslRetCodeToString(error.rsslErrorId), error.rsslErrorId, error.sysError,
        error.text);
    /* Connection should be closed */
}
break;
default:
printf("Unexpected return code (%d) encountered!", retCode);
/* Likely unrecoverable, connection should be closed */
}
}
}

```

Code Example 9: Receiving Data Using **rssl Read**

10.6.4 rsslReadEx Function

The following table describes the `rsslReadEx` function, which expands the functionality of `rsslRead` while preserving backwards compatibility.

 **Tip:** `rsslReadEx` can return compression statistics via `bytesRead` and `uncompressedBytesRead` (members of `RsslReadOutArgs`). This data gives the user a direct way to analyze compression ratios.

FUNCTION NAME	DESCRIPTION
<code>rsslReadEx</code>	<p>This function behaves similarly to <code>rsslRead</code>, but also provides the user with additional information about the data received.</p> <p><code>rsslReadEx</code> takes two additional parameters:</p> <ul style="list-style-type: none"> • <code>RsslReadOutArgs</code>, which consists of variables that hold data being returned to the user. For details on <code>RsslReadOutArgs</code>, refer to Section 10.6.4.1. • <code>RsslReadInArgs</code>, which passes in the <code>readInFlags</code> variable. For details on <code>readInFlags</code>, refer to Section 10.6.4.3.

Table 41: `rssl ReadEx` Function

10.6.4.1 RsslReadOutArgs Options

The following table describes `rsslReadOutArgs` options.

Option	DESCRIPTION
<code>readOutFlags</code>	Flags used for returning information about the outcome of the read.
<code>bytesRead</code>	The number of bytes read from the wire before decompression, during the call to <code>rsslReadEx</code> , including any transport overhead.
<code>uncompressedBytesRead</code>	The number of decompressed bytes read from the wire, including any transport overhead, processed during the call to <code>rsslReadEx</code> .
<code>hashId</code>	Reserved.
<code>nodeId</code>	Reserved.
<code>seqNum</code>	A sequence number used by Elektron Direct Feed data.
<code>FTGroupId</code>	Reserved.
<code>instanceId</code>	Sets the sender's <code>instanceId</code> . <code>instanceId</code> with the IP address and port from the <code>nodeId</code> uniquely identify the specific channel on which the message is sent.

Table 42: `rssl ReadOutArgs` Options

10.6.4.2 RsslReadFlagsOut Enumerations

Flag Enumeration	DESCRIPTION
RSSL_READ_OUT_NO_FLAGS	Channel data does not have associated read flags.
RSSL_READ_OUT_FTGROUP_ID	Channel data includes a valid FT Group ID.
RSSL_READ_OUT_NODE_ID	Channel data includes a valid node ID.
RSSL_READ_OUT_SEQNUM	Channel data includes a sequence number.
RSSL_READ_OUT_HASH_ID	Channel data includes a hash ID.
RSSL_READ_OUT_UNICAST	The message was sent unicast to this node.
RSSL_READ_OUT_INSTANCE_ID	The message includes an instance ID.
RSSL_READ_OUT_RETRANSMIT	Channel data is a retransmission of previous content.

Table 43: RsslReadFlagsOut Enumerations

10.6.4.3 RsslReadInArgs Option

The `rsslReadInArgs` structure has only one option:

Option	DESCRIPTION
readInFlags	Flags used when reading the buffer.

Table 44: rssl ReadInArgs Option

10.7 Writing Data: Overview

When a client or server `RsslChannel.state` is `RSSL_CH_STATE_ACTIVE`, it is possible for an application to write data to the connection. Writing involves a multi-step process. Because the RSSL Transport provides efficient buffer management, the user must obtain a `RsslBuffer` from the RSSL Transport buffer pool (refer to Section 10.8). This can be the guaranteed output buffer pool associated with an `RsslChannel` or the shared buffer pool associated with an `RsslServer`.

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`.

At this point, the user can choose to pack additional information into the same buffer (refer to Section 10.11) or add the buffer to the transports outbound queue (refer to Section 10.9). If queued information cannot be passed to the network, a function is provided to allow the application to continue attempts to flush data to the connection (refer to Section 10.10.2). An I/O notification mechanism can be used to help with determining when the network is able to accept additional bytes for writing. The RSSL Transport can continue to queue data, even if the network is unable to write. The following figure depicts this process and the following sections describe the functionality used to write information to the connection.

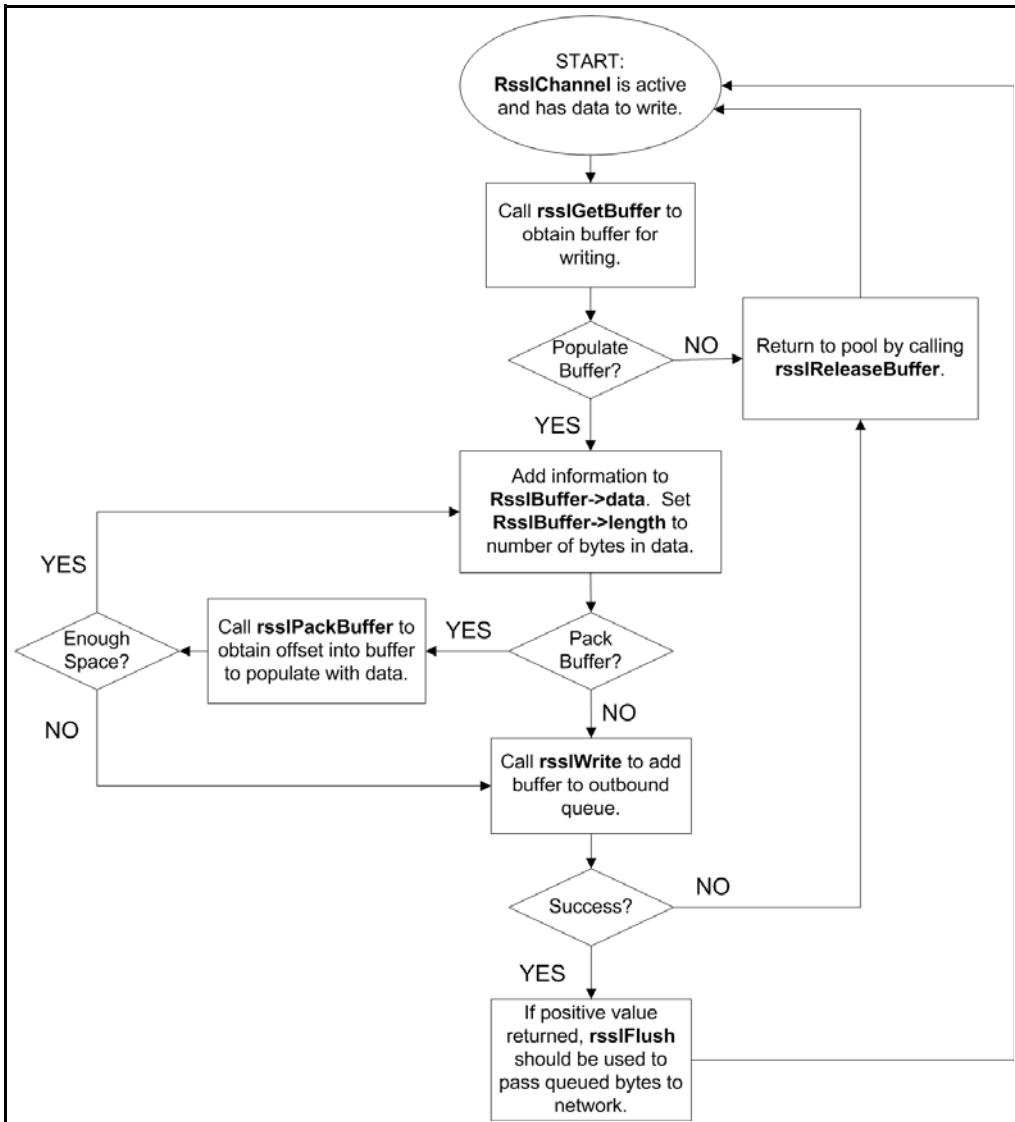


Figure 36. Transport API Writing Flow Chart

10.8 Writing Data: Obtaining a Buffer

To write information, the user must obtain a `RsslBuffer` from the RSSL Transport buffer pool. This buffer can originate from the guaranteed output buffer pool associated with the `RsslChannel` or the shared buffer pool associated with the `RsslServer`. After acquiring a buffer, 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 `rsslWrite` 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 `rsslWrite`, when flushed to the network the buffer will be returned to the correct pool by the transport.

The number of buffers made available to an `RsslChannel` is configurable through `RsslConnectOptions` or `RsslBindOptions`. When connecting, the `guaranteedOutputBuffers` setting controls the number of available buffers. When connections are accepted by an `RsslServer`, the `maxOutputBuffers` parameter controls the number of available buffers per connection. This value is the sum of the number of `guaranteedOutputBuffers` and any available shared pool buffers. For more information about available `rsslConnect` and `rsslBind` options, refer to Table 22 and Table 29.

10.8.1 Buffer Management Functions

FUNCTION NAME	DESCRIPTION
rsslGetBuffer	<p>Obtains a <code>RsslBuffer</code> of the requested size from the guaranteed or shared buffer pool. When the <code>RsslBuffer</code> is returned, the <code>length</code> 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 <code>length</code> to the number of bytes actually used. This ensures that only the required bytes are written to the network.</p> <p>If the requested size is larger than the <code>maxFragmentSize</code>, the transport will create and return the buffer to the user. When written, this buffer will be fragmented by the <code>rsslWrite</code> function (refer to Section 10.9).</p> <p>Because of some additional book keeping required when packing, the application must specify whether a buffer should be ‘packable’ when calling <code>rsslGetBuffer</code>. For more information on packing, refer to Section 10.11.</p> <p>For performance purposes, an application is not permitted to request a buffer larger than <code>maxFragmentSize</code> and have the buffer be ‘packable.’</p> <p>If the buffer is not used or the <code>rsslWrite</code> call fails, the buffer must be returned to the pool using <code>rsslReleaseBuffer</code>. If the <code>rsslWrite</code> call is successful, the buffer will be returned to the correct pool by the transport.</p> <p>Return values are described in Table 46.</p> <p>Note: For shared memory connection types (<code>RSSL_CONN_TYPE_UNIDIR_SHMEM</code>) only one buffer can be obtained at a time. The application must release or write the buffer it has before the application can obtain another buffer.</p>
rsslReleaseBuffer	Releases a <code>RsslBuffer</code> back to the correct pool. This should only be called with buffers that originate from <code>rsslGetBuffer</code> and are not successfully passed to <code>rsslWrite</code> .
rsslBufferUsage	Returns the number of buffers currently in use by the <code>RsslChannel</code> , this includes buffers that the application holds and buffers internally queued and waiting to be flushed to the connection.
rsslServerBufferUsage	Returns the number of shared pool buffers currently in use by all channels connected to the <code>RsslServer</code> , this includes shared pool buffers that the application holds and shared pool buffers internally queued and waiting to be flushed.

Table 45: Buffer Management Functions

10.8.2 rsslGetBuffer Return Values

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

RETURN CODE	DESCRIPTION
Valid buffer returned Success Case	An <code>RsslBuffer</code> is returned to the user. The <code>RsslBuffer.length</code> indicates the number of bytes available to populate, and <code>RsslBuffer.data</code> provides a starting location for population.
NULL buffer returned Error Code: <code>RSSL_RET_BUFFER_NO_BUFFERS</code>	NULL is returned to the user. This value indicates that there are no buffers available to the user. See <code>RsslError</code> content for more details. This typically occurs because all available buffers are queued and pending flushing to the connection. The application can use <code>rsslFlush</code> to attempt releasing buffers back to the pool (refer to Section 10.10.2). Additionally, the <code>rsslIoctl</code> function can be used to increase the number of <code>guaranteedOutputBuffers</code> (refer to Section 10.14).
NULL buffer returned Error Code: <code>RSSL_RET_FAILURE</code>	NULL is returned to the user. This value indicates that some type of general failure has occurred. The <code>RsslChannel</code> should be closed, refer to Section 10.13. See <code>RsslError</code> content for more details.
NULL buffer returned Error Code: <code>RSSL_RET_INIT_NOT_INITIALIZED</code>	Indicates that the RSSL Transport has not been initialized. See the <code>RsslError</code> content for more details. For information on initializing, refer to Section 10.2.

Table 46: `rssl GetBuffer` Return Values

10.9 Writing Data to a Buffer

After an `RsslBuffer` is obtained from `rsslGetBuffer` and populated with the user's data, the buffer can be passed to the `rsslWrite` function. Though the name seems to imply it, this function may not write the contents of the buffer to the connection. By queuing, the RSSL Transport can attempt to use the network layer more efficiently by combining multiple buffers into a single socket write operation. Additionally, queuing allows the application to continue to 'write' data, even while the network has no available space in the output buffer. If `rsslWrite` does not pass all data to the socket, unwritten data will remain in the outbound queue for future writing. If an error occurs, any `RsslBuffer` that has not been successfully passed to `rsslWrite` should be released to the pool using `rsslReleaseBuffer`. The following table describes the `rsslWrite` function as well as some additional parameters associated with it.

The example in Section 10.9.7 demonstrates the use of `rsslGetBuffer` and `rsslReleaseBuffer`.

10.9.1 rsslWrite Function

FUNCTION NAME	DESCRIPTION
rsslWrite	<p>Performs any writing or queuing of data. This function expects the <code>RsslChannel</code> to be in the active state and the buffer to be properly populated, where length reflects the actual number of bytes used. If blocking I/O is used, the <code>rsslWrite</code> function will not return until data was written to the connection or an error has occurred.</p> <p>This function allows for several modifications to be specified for this call. For more information, refer to Section 10.9.2.</p> <p>The RSSL Transport supports writing data at different priority levels. For more details on priority levels, refer to Section 10.10.1.</p> <p>The application can pass in two integer values used for reporting information about the number of bytes that will be written.</p> <ul style="list-style-type: none"> The <code>uncompressedBytesWritten</code> parameter will return the number of bytes to be written, including any transport header overhead but not taking into account any compression. The <code>bytesWritten</code> parameter will return the number of bytes to be written, including any transport header overhead and taking into account any compression. <p>If compression is disabled, <code>uncompressedBytesWritten</code> and <code>bytesWritten</code> should match. The number of bytes saved through the compression process can be calculated by <code>(uncompressedBytesWritten - bytesWritten)</code>.</p> <p>Return values are described in Section 10.9.6.</p> <p>Note: Before passing a buffer to <code>rsslWrite</code>, 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.</p>

Table 47: rssl Write Function

10.9.2 `rsslWrite` Flag Enumeration Values

Note: Before passing a buffer to `rsslWrite`, 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.

FLAG ENUMERATION	MEANING
RSSL_WRITE_NO_FLAGS	No modification will be performed to this <code>rsslWrite</code> operation.
RSSL_WRITE_DO_NOT_COMPRESS	Though the connection might have compression enabled, this flag value indicates that this message will not be compressed. This flag value applies only to the contents of the <code>RsslBuffer</code> passed in with this <code>rsslWrite</code> call.
RSSL_WRITE_DIRECT_SOCKET_WRITE	<p>When set, the <code>rsslWrite</code> function will attempt to pass the contents of the <code>RsslBuffer</code> directly to the socket write operation, bypassing any internal RSSL transport queuing. If any information is currently queued, this buffer will also be queued and the <code>rsslFlush</code> function will be invoked to ensure proper ordering of outbound data.</p> <p>Use of this modification will result in a higher CPU writing cost however it might decrease latency when internal queues are empty.</p> <p>This can be useful for writing at low data rates or when the return codes from <code>rsslWrite</code> and <code>rsslFlush</code> indicate that data is not queued.</p>

Table 48: `rsslWrite` Flags

10.9.3 `rsslWriteEx` Function

The following table describes the `rsslWriteEx` function.

FUNCTION NAME	DESCRIPTION
<code>rsslWriteEx</code>	<p>The <code>rsslWriteEx</code> function expands the functionality of <code>rsslWrite</code> while preserving backwards compatibility. The parameter list of this function is based on a restructuring of the one from <code>rsslWrite</code>.</p> <ul style="list-style-type: none"> Variables which hold output values are options in the <code>RsslWriteOutArgs</code> structure (described in Section 10.9.3.2). Input variables that affect the call to <code>rsslWriteEx</code> are options in the <code>RsslWriteInArgs</code> structure (described in Section 10.9.3.1).

Table 49: `rsslWriteEx` Function

10.9.3.1 `RsslWriteInArgs`

Option	DESCRIPTION
<code>writelnFlags</code>	Sets any flags for use in writing the buffer.
<code>rsslPriority</code>	Sets the priority in flushing the message.
<code>seqNum</code>	Specifies the message's sequence number.

Table 50: `rsslReadInArgs` Options

10.9.3.2 RsslWriteOutArgs

Option	DESCRIPTION
writeOutFlags	Flags that return information about the write's outcome.
bytesWritten	The number of bytes written (taking compression into account) with the write call, including any transport overhead.
uncompressedBytesWritten	The number of bytes written (without taking compression into account) with the write call, including any transport overhead.

Table 51: `rssl ReadInArgs` Options

10.9.3.3 RsslWriteFlagsIn

Note: Before passing a buffer to `rsslWriteEx`, 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.

<code>rsslWriteFLAGSIn</code> ENUMERATION	Description
RSSL_WRITE_IN_NO_FLAGS	No modification will be performed to this <code>rsslWrite</code> operation.
RSSL_WRITE_IN_DO_NOT_COMPRESS	Though the connection might have compression enabled, this flag value indicates that this message will not be compressed. This flag value applies only to the contents of the <code>RsslBuffer</code> passed in with this <code>rsslWrite</code> call.
RSSL_WRITE_IN_DIRECT_SOCKET_WRITE	When set, the <code>rsslWrite</code> function will attempt to pass the contents of the <code>RsslBuffer</code> directly to the socket write operation, bypassing any internal RSSL transport queuing. If any information is currently queued, this buffer will also be queued and the <code>rsslFlush</code> function will be invoked to ensure proper ordering of outbound data. Use of this modification will result in a higher CPU writing cost however it might decrease latency when internal queues are empty. This can be useful for writing at low data rates or when the return codes from <code>rsslWrite</code> and <code>rsslFlush</code> indicate that data is not queued.
RSSL_WRITE_WRITE_IN_SEQNUM	Indicates that the writer wants to attach a sequence number to this message
RSSL_WRITE_WRITE_IN_RETRANSMIT	Indicates that this message is a retransmission of previous content and requires a user-supplied sequence number to indicate which packet is being retransmitted.

Table 52: `rssl WriteFlagsIn` Enumerations

10.9.4 Compression

The `rsslWrite` function performs all necessary compression associated with the connection. Because of information order changes, compression can only be applied to a single priority level. If writing data using different priorities, the first priority level used will leverage compression and all other priority levels will be sent uncompressed. For available compression types, refer to Section 10.4.3.

10.9.5 Fragmentation

In addition to compression, the `rsslWrite` function performs any necessary fragmentation of large buffers. This fragmentation process subdivides one large buffer into smaller `maxFragmentSize` portions, where each part is placed into a buffer acquired from the pool associated with the `RsslChannel`. If the fragmentation cannot fully complete, often due to a shortage of pool buffers, this is indicated by the `RSSL_RET_WRITE_CALL AGAIN` return code. In this situation, the application should use `rsslFlush` to write queued buffers to the connection - this will release buffers back to the pool. When additional pool buffers are available, the application can call `rsslWrite` with the same buffer to continue the fragmentation process from where it left off. The RSSL transport keeps track of necessary information to identify and track individual fragmented messages. This allows an application to write unrelated messages between portions of a fragmented buffer as well as writing multiple fragmented messages that may be interleaved.

Currently, shared memory (`RSSL_CONN_TYPE_UNIDIR_SHMEM`) connections do not support fragmentation.

Note: In the event that the connection is unable to accept additional bytes to write, the RSSL Transport queues on the user's behalf. The application can attempt to pass queued data to the network by using the `rsslWrite` function.

10.9.6 `rsslWrite` Return Codes

The following table lists all return codes that can occur when using the `rsslWrite` function.

Return Code	Description
RSSL_RET_SUCCESS	Indicates that the <code>rsslWrite</code> function was successful and additional bytes have not been internally queued. The <code>rsslFlush</code> function does not need to be called. The application should not release the <code>RsslBuffer</code> ; the Transport API will release it.
Any positive value > 0	Indicates that the <code>rsslWrite</code> function has succeeded and there is information internally queued by the transport. To pass internally queued information to the connection, the <code>rsslFlush</code> function must be called. This information can be queued because there is not sufficient space in the connections output buffer. An I/O notification mechanism can be used to indicate when the socketId has write availability. The application should not release the <code>RsslBuffer</code> ; the Transport API will release it.
RSSL_RET_WRITE_FLUSH_FAILED	Indicates that the <code>rsslWrite</code> function has succeeded, however an internal attempt to flush data to the socket has failed - the channel's state should be inspected. This might not be a failure condition and can occur if there is no available socket output buffer space. If the flush failure is unrecoverable, the <code>RsslChannel.state</code> will transition to <code>RSSL_CH_STATE_CLOSED</code> . If the connection closes, <code>RsslError</code> information will be populated. The application should not release the <code>RsslBuffer</code> ; the Transport API will release it.
RSSL_RET_WRITE_CALL AGAIN	Indicates that a large buffer could not be fully fragmented with this <code>rsslWrite</code> call. This is typically due to all pool buffers being unavailable. An application can use <code>rsslFlush</code> to free up pool buffers or use <code>rsslIoctl</code> to increase the number of available pool buffers. After pool buffers become available again, the same buffer should be used to call <code>rsslWrite</code> an additional time (the same priority level must be used to ensure fragments are ordered properly). This will continue the fragmentation process from where it left off. If the application does not subsequently pass the <code>RsslBuffer</code> to <code>rsslWrite</code> , the buffer should be released by calling <code>rsslReleaseBuffer</code> .
RSSL_RET_FAILURE	Indicates that a general write failure has occurred. The <code>RsslChannel</code> should be closed (refer to Section 10.13). For more details, refer to any <code>RsslError</code> content. The application should release the <code>RsslBuffer</code> by calling <code>rsslReleaseBuffer</code> .

Table 53: `rsslWrite` Return Codes

Return Code	Description
RSSL_RET_BUFFER_TOO_SMALL	<p>Indicates that either the buffer has been corrupted, possibly by exceeding the allowable length, or it is not a valid pool buffer. For more details, refer to any RsslError content.</p> <p>If this RsslBuffer was obtained from rsslGetBuffer, the application should release it by calling rsslReleaseBuffer.</p>
RSSL_RET_INIT_NOT_INITIALIZED	<p>Indicates that the RSSL Transport has not been initialized.</p> <ul style="list-style-type: none"> For more details, refer to any RsslError content. For information on initializing, refer to Section 10.2. <p>The application's attempt to call rsslGetBuffer should have failed for the same reason, so an RsslBuffer should not be present.</p>

Table 53: rssl Write Return Codes (Continued)

10.9.7 rsslGetBuffer and rsslWrite Example

The following example shows typical use of [rsslGetBuffer](#) and [rsslWrite](#). This code would be similar for client or server based [RsslChannel](#) structures.

```

/* rsslGetBuffer and rsslWrite use, be sure to keep track of the return values from write so
   data is not
   stranded in the output buffer - rsslFlush may be required to continue attempting to pass data
   to the
   connection */

RsslRet retCode = RSSL_RET_FAILURE;
RsslBuffer *pBuffer = 0;
RsslUInt32 outBytes = 0;
RsslUInt32 outUncompBytes = 0;

/* Ask for a 500 byte non-packable buffer to write into */
if ((pBuffer = rsslGetBuffer(pChnl, 500, RSSL_FALSE, &error)) != 0)
{
    /* if a buffer is returned, we can populate and write, encode an RsslMsg into the buffer */
    /* set the buffer on an RsslEncodeIterator */
    rsslSetEncodeIteratorBuffer(&encIter, pBuffer);
    /* set version information of the connection on the encode iterator so proper versioning
       can be
       performed */
    rsslSetEncodeIteratorRWFVersion(&encIter, pChnl->majorVersion, pChnl->minorVersion);
    /* populate message and encode it - see 11.6.3.5 for more message encoding information */
    retCode = rsslEncodeMsg(&encIter, &rsslMsg);
    /* set the buffer's encoded content length prior to writing, this can be obtained from the
       iterator.*/
    pBuffer->length = rsslGetEncodedBufferLength(&encIter);

    /* Now write the data - keep track of return code */
    /* this example writes buffer as high priority and no write modification flags */
}

```

```

retCode = rsslWrite(pChnl, pBuffer, RSSL_HIGH_PRIORITY, 0, &outBytes, &outUncompBytes,
&error);

if (retCode > RSSL_RET_SUCCESS)
{
    /* The write was successful and there is more data queued in RSSL Transport.
       The rsslFlush function (see Section 10.10.2) should be used to continue attempting to
       flush
       data to the connection. UPA will release buffer.*/
}
else
{
    /* Handle return codes appropriately, not all return values are failure conditions */
    switch(retCode)
    {
        case RSSL_RET_SUCCESS:
        {
            /* Successful write and all data has been passed to the connection */
            /* Continue with next operations. UPA will release buffer.*/
        }
        break;
        case RSSL_RET_WRITE_CALL AGAIN:
        {
            /* Large buffer is being split by transport, but out of output buffers */
            /* Schedule a call to rsslFlush (see Section 10.10.2) and then call the rsslWrite
               function
               Again with this same exact buffer to continue the fragmentation process. */
            /* Only release the buffer if not passing it to rsslWrite again. */
        }
        break;
        case RSSL_RET_WRITE_FLUSH FAILED:
        {
            /* The write was successful, but an attempt to flush failed. UPA will release
               buffer.*/
            /* Must check channel state to determine if this is unrecoverable or not */
            if (pChnl->state == RSSL_CH_STATE_CLOSED)
            {
                printf("Error %s (%d) (errno: %d) encountered with rsslWrite. Error Text:
                      %s\n",
                      rsslRetCodeToString(error.rsslErrorId),
                      error.rsslErrorId, error.sysError, error.text);
                /* Connection should be closed, return failure */
            }
            else
            {
                /* Successful write call, data is queued. The rsslFlush function
                   (see Section 10.10.2) should be used to continue attempting to flush data to
                   the connection. */
            }
        }
    }
}

```

```

        }

        break;
    case RSSL_RET_BUFFER_TOO_SMALL: /* Nothing to read */
    {
        /* Buffer somehow got corrupted, if it was from rsslGetBuffer, release it */
        rsslReleaseBuffer(pBuffer, &error);
    }
    break;
    case RSSL_RET_INIT_NOT_INITIALIZED:
    case RSSL_RET_FAILURE:
    {
        printf("Error %s (%d) (errno: %d) encountered with rsslWrite. Error Text: %s\n",
               rsslRetCodeToString(error.rsslErrorId), error.rsslErrorId, error.sysError,
               error.text);
        /* Buffer must be released - return code from rsslReleaseBuffer can be checked */
        rsslReleaseBuffer(pBuffer, &error);
        /* Connection should be closed, return failure */
    }
    break;
    default:
        printf("Unexpected return code (%d) encountered!", retCode);
        /* Likely unrecoverable, connection should be closed */
    }
}
else
{
    /* Check to see if this is just out of buffers or if it's unrecoverable */
    if (error.rsslErrorId == RSSL_RET_BUFFER_NO_BUFFERS)
    {
        /* The rsslFlush function (Section 10.10.2) should be used to attempt to free buffers
back to the
pool */
    }
    else
    {
        printf("Error %s (%d) (errno: %d) encountered with rsslGetBuffer. Error Text: %s\n",
               rsslRetCodeToString(error.rsslErrorId), error.rsslErrorId, error.sysError,
               error.text);
        /* Connection should be closed, return failure */
    }
}

```

Code Example 10: Writing Data Using `rsslWrite`, `rsslGetBuffer`, and `rsslReleaseBuffer`

10.10 Managing Outbound Queues

Because it may not be possible for the `rsslWrite` function to pass all data to the underlying socket, some data may be queued by the RSSL Transport. Applications can use the `rsslFlush` function to continue attempting to pass queued data to the connection.

10.10.1 Ordering Queued Data: `rsslWrite` Priorities

Using the `rsslWrite` function, an application can associate a priority with each `RsslBuffer`. Priority information is used to determine outbound ordering of data, and can allow for higher priority information to be written to the connection before lower priority data, even if the lower priority data was passed to `rsslWrite` first. Only queued data will incur any ordering changes due to priority, and data directly written to the socket by `rsslWrite` will not be impacted.

Priority ordering occurs as part of the `rsslFlush` call (refer to Section 10.10.2), where the `priorityFlushStrategy` determines how to handle each priority level. The default `priorityFlushStrategy` writes buffers in the order: High, Medium, High, Low, High, Medium. This provides a slight advantage to the medium priority level and a greater advantage to high priority data. Data order is preserved within each priority level (thus, if all buffers are written with the same priority, data is not reordered). If a particular priority level being flushed does not have content, `rsslFlush` will move to the next priority in the `priorityFlushStrategy`. The `priorityFlushStrategy` can be changed for each `RsslChannel` by using the `rsslIoctl` function (refer to Section 10.14).

10.10.1.1 Priority Ordering

The following figure presents an example of a possible priority write ordering. On the left, there are three queues and each queue is associated with one of the available `rsslWrite` priority values. As the user calls `rsslWrite` and assigns priorities to their buffers, they will be queued at the appropriate priority level. As the `rsslFlush` function is called, buffers are removed from the queues in a manner that follows the `priorityFlushStrategy`.

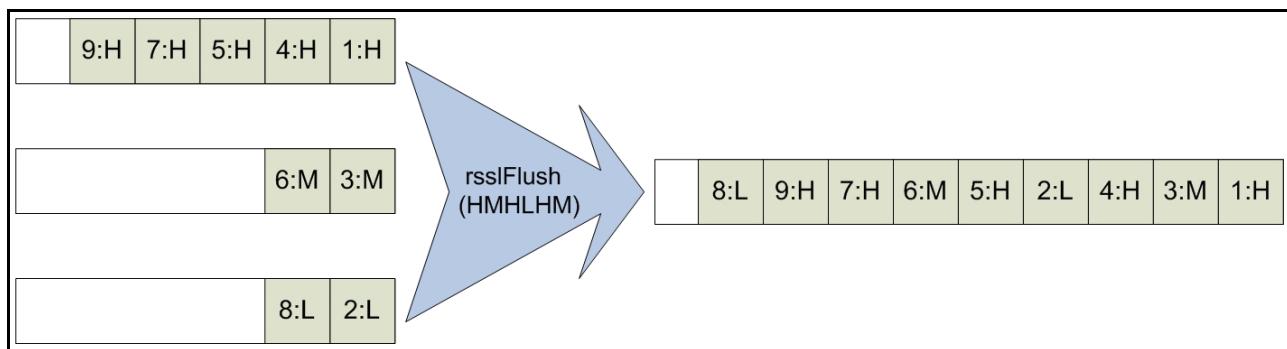


Figure 37. `rsslWrite` Priority Scenario

On the left side of the figure there are three outbound queues, one for each priority value. As buffers enter the queues (as a result of an `rsslWrite` call), they are marked with a number and the priority value associated with their queue. The number indicates the order the buffers were passed to `rsslWrite`, so the buffer marked **1** was the first buffer into `rsslWrite`, the buffer marked **5** was the 5th buffer into `rsslWrite`. Buffers are marked **H** if they are in the high priority queue, **M** if they are in the medium priority queue, or **L** if they are in the low priority queue. Buffers leave the queue (as a result of an `rsslFlush` call) in the order specified by the `priorityFlushStrategy`, which by default is **HMHLHM**. In Figure 37, the queue on the right side represents the order in which buffers are written to the network and the order that they will be returned when `rsslRead` is called. The buffers will still be marked with their `number:priority` information so it is easy to see how data is reordered by any priority writing.

Notice that though data was reordered between various priorities, individual priority levels are not reordered. Thus, all buffers in the high priority are written in the order they are queued, even though some medium and low buffers are sent as well.

10.10.1.2 Priority Value Enumerations

ENUMERATION	MEANING
RSSL_HIGH_PRIORITY	If not directly written to the socket, this RsslBuffer will be flushed at the high priority.
RSSL_MEDIUM_PRIORITY	If not directly written to the socket, this RsslBuffer will be flushed at the medium priority.
RSSL_LOW_PRIORITY	If not directly written to the socket, this RsslBuffer will be flushed at the low priority.

Table 54: **rsslWrite** Priority Value Enumerations

10.10.2 `rsslFlush` Function

If all available output space is used for a connection, data might be queued as a result. An I/O notification mechanism can be used to alert the application when output space becomes available on a connection.

Note: The return value from `rsslFlush` indicates whether there are any queued bytes left to pass to the connection. If this is a positive value (typical when operating system output buffers lack space), the application should continue to call `rsslFlush` until all bytes have been written.

FUNCTION NAME	DESCRIPTION
<code>rsslFlush</code>	<p>Writes queued data to the connection. This function expects the <code>RsslChannel</code> to be in the active state. If data is not queued, the <code>rsslFlush</code> function is not required and should return immediately.</p> <p>This function performs any buffer reordering that might occur due to priorities passed in on the <code>rsslWrite</code> function. For more information about priority writing, refer to Section 10.10.1. Return values are described in Table 56.</p>

Table 55: `rsslFlush` Function

10.10.3 `rsslFlush` Return Codes

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

RETURN CODE	DESCRIPTION
<code>RSSL_RET_SUCCESS</code>	Indicates that the <code>rsslFlush</code> function has succeeded and additional bytes are not internally queued. The <code>rsslFlush</code> function need not be called.
Any positive value > 0	Indicates that the <code>rsslFlush</code> function has succeeded, however data is still internally queued by the transport. The <code>rsslFlush</code> function must be called again. Data might still be queued because the connections output buffer does not have sufficient space. An I/O notification mechanism can indicate when the <code>socketId</code> has write availability.
<code>RSSL_RET_FAILURE</code>	Indicates that a general failure has occurred, often because the underlying connection is unavailable or closed. The <code>RsslChannel</code> should be closed (refer to Section 10.13). For more details, refer to the <code>RsslError</code> content.
<code>RSSL_RET_INIT_NOT_INITIALIZED</code>	Indicates that the RSSL Transport is not initialized. For more details, refer to the <code>RsslError</code> content. For information on initializing, refer to Section 10.2.

Table 56: `rsslFlush` Return Codes

10.10.4 rsslFlush Example

The following example shows typical use of **rsslFlush**. This example assumes use of an I/O notification mechanism. This code would be similar for client or server based **RsslChannel** structures.

```

/* rsslFlush use, be sure to keep track of the return values from rsslFlush so data is not
stranded in the output buffer - rsslFlush may need to be called again to continue attempting
to
pass data to the connection */
RsslRet retCode = RSSL_RET_FAILURE;

/* Assuming this section of code was called because of a write file descriptor alert */
if ((retCode = rsslFlush(pChnl, &error)) > RSSL_RET_SUCCESS)
{
    /* There is still data left to flush, leave our write notification enabled so we get called
again,
    If everything wasn't flushed, it usually indicates that the TCP output buffer cannot
accept more
    yet */
}
else
{
    switch (retCode)
    {
        case RSSL_RET_SUCCESS:
        {
            /* Everything has been flushed, no data is left to send - unset write notification */
            FD_CLR(pChnl->socketId, &writefds);
        }
        break;
        case RSSL_RET_INIT_NOT_INITIALIZED:
        case RSSL_RET_FAILURE:
        {
            printf("Error %s (%d) (errno: %d) encountered with rsslFlush. Error Text: %s\n",
                   rsslRetCodeToString(error.rsslErrorId), error.rsslErrorId, error.sysError,
                   error.text);
            /* Connection should be closed, return failure */
        }
        break;
        default:
            printf("Unexpected return code (%d) encountered!", retCode);
            /* Likely unrecoverable, connection should be closed */
    }
}

```

Code Example 11: **rssl Flush** Use

10.11 Packing Additional Data into a Buffer

If an application is writing many small buffers, it might be advantageous to combine the small buffers into one larger buffer. This can increase the efficiency of the transport layer by reducing overhead associated with each write operation, though it might increase 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 is to pack a fixed number of messages each time. A slightly more complex technique could use the returned `Rss1Buffer.length` to determine the amount of remaining space 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 at which data arrives (i.e., the packed buffer will not be written until enough data arrives to fill it). One method that can balance this is to use a timer 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, otherwise whenever the timer expires, whatever is in the buffer will be written (regardless of the amount of data in the buffer). This limits latency to a maximum, acceptable amount as set by the duration of the timer.

The `Rss1PackBuffer` method packs multiple messages into one `Rss1Buffer`.

function NAME	DESCRIPTION
Rss1PackBuffer	<p>Packs the contents of a passed-in <code>Rss1Buffer</code> and returns a new <code>Rss1Buffer</code> to continue packing. The returned buffer provides a <code>data</code> pointer for populating and the <code>length</code> conveys number of bytes available in the buffer. An application can use the <code>Rss1Buffer.length</code> to determine the amount of space available to continue packing buffers.</p> <p>For a buffer to allow packing, it must be requested from <code>rss1GetBuffer</code> as ‘packable’ and cannot exceed the <code>maxFragmentSize</code>.</p> <p>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.</p> <p>Return values are described in Table 58.</p> <p>Packing is not supported for shared memory (<code>RSSL_CONN_TYPE_UNIDIR_SHMEM</code>) connections.</p>

Table 57: Rss1 PackBuffer Function

10.11.1 Rss1PackBuffer Return Values

The following table defines return and error code values that can occur when using `Rss1PackBuffer`.

RETURN CODE	DESCRIPTION
Valid buffer returned Success Case	An <code>Rss1Buffer</code> is returned to the user. The <code>Rss1Buffer.length</code> indicates the number of bytes available to populate and the <code>Rss1Buffer.data</code> 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 <code>Rss1Channel</code> should be closed (refer to Section 10.13). For more details, refer to <code>Rss1Error</code> content.
NULL buffer returned Error Code: RSSL_RET_INIT_NOT_INITIALIZED	Indicates that the RSSL Transport has not been initialized. See the <code>Rss1Error</code> content for more details. For information on initializing, refer to Section 10.2.

Table 58: Rss1 PackBuffer Return Values

10.11.2 Example: `rsslGetBuffer`, `RsslPackBuffer`, and `rsslWrite`

The following example shows typical use of `rsslGetBuffer`, `RsslPackBuffer`, and `rsslWrite`. This code would be similar for client or server based `RsslChannel` structures.

```

/* rsslGetBuffer, RsslPackBuffer and rsslWrite use, be sure to keep track of the return values
   from
   write so data is not stranded in the output buffer - rsslFlush may be required to continue
   attempting to pass data to the connection */
RsslRet retCode = RSSL_RET_FAILURE;
RsslBuffer *pBuffer = 0;
RsslBuffer *pOrigBuffer = 0;
RsslUInt32 outBytes = 0;
RsslUInt32 outUncompBytes = 0;

/* Ask for a 6000 byte packable buffer to write multiple messages into */
if ((pBuffer = rsslGetBuffer(pChnl, 6000, RSSL_TRUE, &error)) != 0)
{
    /* if a buffer is returned, we can populate and write, encode an RsslMsg into the buffer */
    /* store a copy of the original buffer pointer. This can be used to properly release if
       an error occurs in rsslPackBuffer */
    pOrigBuffer = pBuffer;
    /* set the buffer on an RsslEncodeIterator */
    rsslSetEncodeIteratorBuffer(&encIter, pBuffer);
    /* set version information of the connection on the encode iterator so proper versioning
       can be
       performed */
    rsslSetEncodeIteratorRWFVersion(&encIter, pChnl->majorVersion, pChnl->minorVersion);
    /* populate message and encode it - see (Section 12.2.9.1) for more message encoding
       information */
    retCode = rsslEncodeMsg(&encIter, &rsslMsg);
    /* set the buffer's encoded content length prior to writing, this can be obtained from the
       iterator.*/
    pBuffer->length = rsslGetEncodedBufferLength(&encIter);

    /* Instead of writing, let's continue packing messages into the buffer */
    /* This will take the existing buffer and return a new location to continue encoding into */
    if ((pBuffer = rsslPackBuffer(pChnl, pBuffer, &error)) == 0)
    {
        printf("Error %s (%d) (errno: %d) encountered with rsslPackBuffer. Error Text: %s\n",
               rsslRetCodeToString(error.rsslErrorId), error.rsslErrorId, error.sysError,
               error.text);
        /* Buffer must be released - return code from rsslReleaseBuffer can be checked */
        rsslReleaseBuffer(pOrigBuffer, &error);
        /* Connection should be closed, return failure */

    }
}

```

```

/* if a buffer is returned, encode an additional message */
/* set the buffer on an RsslEncodeIterator */
rsslSetEncodeIteratorBuffer(&encIter, pBuffer);
/* set version information of the connection on the encode iterator so proper versioning
can be performed */
rsslSetEncodeIteratorRWFVersion(&encIter, pChnl->majorVersion, pChnl->minorVersion);
/* populate message and encode it - see (Section 12.2.9.1) for more message encoding
information */
retCode = rsslEncodeMsg(&encIter, &rsslMsg);
/* set the buffer's encoded content length prior to writing, this can be obtained from the
iterator.*/
pBuffer->length = rsslGetEncodedBufferLength(&encIter);
/* Instead of writing, lets continue packing messages into the buffer */
/* This will take the existing buffer and return a new location to continue encoding into */
if ((pBuffer = rsslPackBuffer(pChnl, pBuffer, &error)) == 0)
{
    printf("Error %s (%d) (errno: %d) encountered with rsslPackBuffer. Error Text: %s\n",
           rsslRetCodeToString(error.rsslErrorId), error.rsslErrorId, error.sysError,
           error.text);
    /* Buffer must be released - return code from rsslReleaseBuffer can be checked */
    rsslReleaseBuffer(pOrigBuffer, &error);
    /* Connection should be closed, return failure */

}

/* Packing can continue like this until the application determines its time to stop -
this can be due to the pBuffer->length not containing enough space for an additional
message,
a timer alerting that enough pack time has elapsed, etc */

/* If rsslPackBuffer is called and nothing is put into the last buffer before rsslWrite is
called,
buffer.length should be set to 0. If content is encoded into last buffer before
rsslWrite,
buffer.length should be set to encoded length of content. */
pBuffer->length = 0;

/* After packing is complete, write the buffer as normal */
retCode = rsslWrite(pChnl, pBuffer, RSSL_HIGH_PRIORITY, 0, &outBytes, &outUncompBytes,
&error);

/* See Example in Section 10.9 for full rsslWrite error handling example */
}
else
{
    /* Check to see if this is just out of buffers or if it's unrecoverable */
    if (error.rsslErrorId == RSSL_RET_BUFFER_NO_BUFFERS)
    {
}

```

```
/* The rsslFlush function (Section 10.10.2) should be used to attempt to free buffers
back to the
pool */
}
else
{
    printf("Error %s (%d) (errno: %d) encountered with rsslGetBuffer. Error Text: %s\n",
           rsslRetCodeToString(error.rsslErrorId), error.rsslErrorId, error.sysError,
           error.text);
    /* Connection should be closed, return failure */
}
}
```

Code Example 12: Message Packing Using [RsslPackBuffer](#)

10.12 Ping Management

Ping or heartbeat messages indicate the continued presence of an application. These are typically required only when no other data is exchanged. For example, there may be long periods of time that elapse between requests made from an OMM consumer application. In this situation, the consumer sends periodic heartbeat messages to inform the providing application that it is still connected. Because the provider application is likely sending data more frequently (providing updates on any streams the consumer has requested), the provider might not need to send heartbeats (as the other data sufficiently announces its continued presence). The application is responsible for managing the sending and receiving of heartbeat messages on each connection.

10.12.1 Ping Timeout

Applications are able to configure their desired `pingTimeout` values, where the *ping timeout* is the point at which a connection is terminated due to inactivity. Heartbeat messages are typically sent every one-third of the `pingTimeout`, ensuring that heartbeats are exchanged prior to a ping timeout. This can be useful for detecting a connection loss prior to any kind of network or operating system notification.

`pingTimeout` values are negotiated between a connecting client application and the server application, where the server can specify a minimum allowable ping timeout (via the `minPingTimeout` option) and the direction in which heartbeats flow (via `serverToClientPings` and `clientToServerPings`). For more information on specifying these options, refer to Section 10.3.2.1 and Section 10.4.1.1. During negotiation, the lowest `pingTimeout` value is selected. Because `minPingTimeout` sets the lowest possible value, if a client's specified `pingTimeout` value is less than `minPingTimeout`, the connection uses the `minPingTimeout` as its `pingTimeout` value. After a connection transitions to the active state, the negotiated `pingTimeout` is available through the `RsslChannel.pingTimeout`.

The RSSL Transport uses the following formula to determine the negotiated `pingTimeout` value:

```
/* Determine lesser of client or servers pingTimeout */
if (client.pingTimeout < server.pingTimeout)
    connection.pingTimeout = client.pingTimeout;
else
    connection.pingTimeout = server.pingTimeout;
/* Determine whether timeout is less than minimum allowable timeout */
if (connection.pingTimeout < server.minPingTimeout)
    connection.pingTimeout = server.minPingTimeout;
```

Code Example 13: Ping Negotiation Calculation

10.12.2 `rssIPing` Function

An application typically monitors both messages and heartbeats. If bytes are flushed to the network, this is considered sufficient as a heartbeat so any timer mechanism associated with sending heartbeats can be reset. When bytes are received or `rssIPing` returns **RSSL_RET_READ_PING** (refer to Section 9.6), this is comparable to receiving a heartbeat so any timer mechanism associated with receiving heartbeats can be reset. If either the sending or receiving heartbeat timer mechanism reaches or surpasses the `RssIPing.pingTimeout` value, the connection should be closed.

The following table describes the `rssIPing` function, used to send heartbeat messages.

function NAME	DESCRIPTION
<code>rssIPing</code>	<p>Attempts to write a heartbeat message on the connection. This function expects an active <code>RssIPing</code>.</p> <p>If an application calls the <code>rssIPing</code> function while other bytes are queued for output, the RSSL Transport layer suppresses the heartbeat message and attempts to flush bytes to the network on the user's behalf.</p> <p>When using a shared memory (RSSL_CONN_TYPE_UNIDIR_SHMEM) connection type, pings can only be sent from server to client.</p> <p>Return values are described in Table 60.</p>

Table 59: `rssIPing` function

10.12.3 `rssIPing` Return Values

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

RETURN CODE	DESCRIPTION
RSSL_RET_SUCCESS	Indicates that the <code>rssIPing</code> function succeeded and additional bytes are not internally queued.
Any positive value > 0	Indicates that queued data was sent as a heartbeat but data is still internally queued by the transport. The <code>rssIPing</code> function must be called to continue passing queued bytes to the connection. Data might still be queued because the connections output buffer does not have sufficient space. An I/O notification mechanism indicate when the <code>socketId</code> has write availability.
RSSL_RET_FAILURE	This value indicates that some type of general failure has occurred. The <code>RssIPing</code> should be closed (refer to Section 10.13). For more details, refer to the <code>RsslError</code> content.
RSSL_RET_INIT_NOT_INITIALIZED	Indicates that the RSSL Transport has not been initialized. <ul style="list-style-type: none"> • For more details, refer to the <code>RsslError</code> content. • For information on initializing the transport, refer to Section 10.2.

Table 60: `rssIPing` Return Codes

10.12.4 rsslPing Example

The following example shows typical use of `rsslPing`. This example assumes use of some kind of timer mechanism to execute when necessary. This code would be similar for client or server based `RsslChannel` structures.

```

/* rsslPing use - this demonstrates sending of heartbeats */
/* Additionally, an application should determine if data or pings have been received, if not
   the
   application should determine if pingTimeout has elapsed, and if so connection should be
   closed */
RsslRet retCode = RSSL_RET_FAILURE;

/* First, send our ping, if there is other data queued, that will be flushed instead */
if ((retCode = rsslPing(pChnl, &error)) > RSSL_RET_SUCCESS)
{
    /* There is still data left to flush, leave our write notification enabled so we get called
       again,
       If everything wasn't flushed, it usually indicates that the TCP output buffer cannot
       accept more
       yet */
}
else
{
    switch (retCode)
    {
        case RSSL_RET_SUCCESS:
        {
            /* Ping message has been sent successfully */
        }
        break;
        case RSSL_RET_INIT_NOT_INITIALIZED:
        case RSSL_RET_FAILURE:
        {
            printf("Error %s (%d) (errno: %d) encountered with rsslPing. Error Text: %s\n",
                   rsslRetCodeToString(error.rsslErrorId), error.rsslErrorId, error.sysError,
                   error.text);
            /* Connection should be closed, return failure */
        }
        break;
        default:
            printf("Unexpected return code (%d) encountered!", retCode);
            /* Likely unrecoverable, connection should be closed */
    }
}

```

Code Example 14: rssl Ping Use

10.13 Closing Connections

10.13.1 Functions for Closing Connections

When an error occurs on a connection or an `RsslChannel` is being disconnected, the `rsslCloseChannel` function should be called to perform any necessary cleanup and to shutdown the underlying socket. This will release any pool-based resources back to their respective pools. If the application is holding any buffers obtained from `rsslGetBuffer`, they should be released using `rsslReleaseBuffer` prior to closing the channel.

If a server is being shut down, use the `rsslCloseServer` function to close the listening socket and perform any necessary cleanup. All currently connected `RsslChannels` will remain open. This allows applications to continue sending and receiving data, while preventing new applications from connecting. The server has the option of calling `rsslCloseChannel` to shut down any currently connected applications.

function NAME	DESCRIPTION
<code>rsslCloseChannel</code>	Closes a client- or server-based <code>RsslChannel</code> . This releases any pool-based resources back to their respective pools, closes the connection, and performs any additional necessary cleanup.
	Note: If an application is multi-threaded, all other threads that depend on the closed channel should complete their use prior to calling <code>rsslCloseChannel</code> .
<code>rsslCloseServer</code>	Closes a listening socket associated with an <code>RsslServer</code> . <code>rsslCloseServer</code> releases any pool-based resources back to their respective pools, closes the listening socket, and performs any additional necessary cleanup. Established connections remain open, allowing for continued exchange of data. If needed, the server can use <code>rsslCloseChannel</code> to shutdown any remaining connections.

Table 61: RSSL Connection Closing Functionality

10.13.2 Close Connections Example

The following example shows typical use of `rsslCloseChannel` and `rsslCloseServer`.

```
/* rsslCloseChannel */
if (rsslCloseChannel(pChnl, &error) < RSSL_RET_SUCCESS)
{
    printf("Error %s (%d) (errno: %d) encountered with rsslCloseChannel. Error Text: %s\n",
           rsslRetCodeToString(error.rsslErrorId), error.rsslErrorId, error.sysError,
           error.text);
}

/* rsslCloseServer */
if (rsslCloseServer(pSrvr, &error) < RSSL_RET_SUCCESS)
{
    printf("Error %s (%d) (errno: %d) encountered with rsslCloseServer. Error Text: %s\n",
           rsslRetCodeToString(error.rsslErrorId), error.rsslErrorId, error.sysError,
           error.text);
}
```

Code Example 15: Closing a Connection Using `rsslCloseChannel` and `rsslCloseServer`

10.14 Utility Functions

The RSSL Transport layer provides several additional utility functions. These functions can be used to query more detailed information for a specific connection or change certain **RsslChannel** or **RsslServer** parameters during run-time. These functions are described in the following tables.

10.14.1 General Transport Utility Functions

FUNCTION NAME	DESCRIPTION
rsslGetChannelInfo	Allows the application to query RsslChannel negotiated parameters and settings and retrieve all current settings. This includes maxFragmentSize and negotiated compression information as well as many other values. See RsslChannelInfo structure, defined in Table 63, for a full list of available settings.
rsslGetServerInfo	Allows the application to query RsslServer related values, such as current and peak shared pool buffer usage statistics. This populates an RsslServerInfo structure, defined in Table 66.
rsslIoclt	Allows the application to change various settings associated with the RsslChannel . The available options are defined in Table 67.
rsslServerIoclt	Allows the application to change various settings associated with the RsslServer . The available options are defined in Table 68.
rsslHostByName	Takes an RsslBuffer populated with a hostname, where length is set to the length of the contained hostname. The hostname is used to look up and return a four-byte IP address, in host byte order.
rsslGetUserName	Takes an RsslBuffer with associated memory pointed to by data , where length is set to the amount of space available. Queries the username associated with the owner of the current process, and returns it in the provided buffer.
rsslIPAddrStringToInt	Takes a dotted-decimal IP address string (e.g. “127.0.0.1”) and converts to a host byte order integer equivalent.
rsslIPAddrUIntToString	Takes a host byte order integer representation of an IP address and converts to a dotted-decimal IP address string.

Table 62: Transport Utility Functions

10.14.2 RsslChannelInfo Structure Members

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

Structure Member	DESCRIPTION
maxFragmentSize	The maximum allowed buffer size which can be written to the network. If a larger buffer is required, the RSSL Transport will internally fragment the larger buffer into smaller buffers whose size is set to <code>maxFragmentSize</code> . This is the largest size a user can request while still being ‘packable.’
numInputBuffers	The number of sequential input buffers into which the <code>RsslChannel</code> reads data. This controls the maximum number of bytes that can be handled with a single network read operation on each channel. Each input buffer can contain <code>maxFragmentSize</code> bytes. Input buffers are allocated at initialization time.
guaranteedOutputBuffers	The guaranteed number of buffers which this <code>RsslChannel</code> can use while writing data. Each buffer can contain <code>maxFragmentSize</code> bytes. Guaranteed output buffers are allocated at initialization time. For more details on obtaining a buffer, refer to Section 10.8. You can configure <code>guaranteedOutputBuffers</code> using <code>rsslIoctl</code> , as described in Section 10.14.6.
maxOutputBuffers	The maximum number of output buffers which this <code>RsslChannel</code> can use. (<code>maxOutputBuffers - guaranteedOutputBuffers</code>) is equal to the number of shared pool buffers that this <code>RsslChannel</code> can use. Shared pool buffers are only used if all <code>guaranteedOutputBuffers</code> are unavailable. If <code>maxOutputBuffers</code> is equal to the <code>guaranteedOutputBuffers</code> value, shared pool buffers are unavailable. You can configure <code>maxOutputBuffers</code> using <code>rsslIoctl</code> , as described in Section 10.14.6.
pingTimeout	The negotiated ping timeout value. Typically, the rule of thumb in handling heartbeats is to send a heartbeat every <code>pingTimeout</code> /3 seconds. For more details on <code>pingTimeout</code> , refer to Section 10.12.1.
serverToClientPings	Sets whether server is expected to send heartbeat messages: <ul style="list-style-type: none"> If set to <code>RSSL_TRUE</code>, heartbeat messages must flow from server to client. If set to <code>RSSL_FALSE</code>, the server is not required to send heartbeats. TREP and other Thomson Reuters components typically require this value to be set to <code>RSSL_TRUE</code> .
clientToServerPings	Sets whether the client is expected to send heartbeat messages: <ul style="list-style-type: none"> If set to <code>RSSL_TRUE</code>, heartbeat messages must flow from client to server. If set to <code>RSSL_FALSE</code>, the client is not required to send heartbeats. TREP and other Thomson Reuters components typically require this value to be set to <code>RSSL_TRUE</code> .
tcpSendBufSize	DEPRECATED. To configure equivalent functionality, refer to <code>sysSendBufSize</code> (in this table).

Table 63: `Rssl Channel Info` Structure Members

Structure Member	DESCRIPTION
sysSendBufSize	Sets the size of the send or output buffer associated with the underlying transport. The RSSL Transport has additional output buffers, controlled by <code>maxOutputBuffers</code> and <code>guaranteedOutputBuffers</code> . For some connection types, you can configure <code>sysSendBufSize</code> using <code>rsslIoctl</code> , as described in Section 10.14.6.
tcpRecvBufSize	DEPRECATED. To configure equivalent functionality, refer to <code>sysRecvBufSize</code> in this table.
sysRecvBufSize	Sets the size of the receive or input buffer associated with the underlying transport. The RSSL Transport has an additional input buffer controlled by <code>numInputBuffers</code> . For some connection types, you can configure <code>sysRecvBufSize</code> using <code>rsslIoctl</code> , as described in Section 10.14.6.
compressionType	Sets the type of compression to use on this connection. Refer to Section 10.4.3 for more information about supported compression types.
compressionThreshold	Sets the compression threshold. Messages smaller than the threshold are not compressed; messages larger than the threshold are compressed.
priorityFlushStrategy ^a	The currently priority level order used when flushing buffers to the connection, where H = High priority, M = Medium priority, and L = Low priority. When passed to <code>rsslWrite</code> , each buffer is associated with the priority level at which it should be written. The default <code>priorityFlushStrategy</code> writes buffers in the order: High, Medium, High, Low, High, Medium. This provides a slight advantage to the medium-priority level and a greater advantage to high-priority data. Data order is preserved within each priority level and if all buffers are written with the same priority, the order of data does not change. You can configure <code>priorityFlushStrategy</code> using <code>rsslIoctl</code> , as described in Section 10.14.6.
multicastStats	If using a connection type of either <code>RSSL_CONN_TYPE_RELIABLE_MCAST</code> or <code>RSSL_CONN_TYPE_SEQ_MCAST</code> , this substructure reports information about sent and received packets, including any gap or retransmission information. For details on options used with <code>multicastStats</code> , refer to Section 10.14.3.
componentInfoCount	Indicates the number of <code>RsslComponentInfo</code> structures referred to by the <code>componentInfo</code> member. This information is only available when communicating with devices that support connected component versioning; otherwise this count is set to 0 .
componentInfo	A pointer to an array of <code>RsslComponentInfo</code> structures. One <code>RsslComponentInfo</code> structure will be present for each connected device that supports connected component versioning. The count is indicated by the <code>componentInfoCount</code> . For more detailed information on the <code>RsslComponentInfo</code> structure, refer to Section 10.14.4.

Table 63: Rssi Channel Info Structure Members (Continued)

a. Allows for up to 32 one-byte characters to be represented. ‘H’ = high priority, ‘M’ = medium priority, and ‘L’ = low priority.

10.14.3 multicastStats Options

Option	DESCRIPTION
mcastSent	The number of multicast packets sent by this <code>Rss1Channel</code> .
mcastRcvd	The number of multicast packets received by this <code>Rss1Channel</code> .
unicastSent	The number of unicast UDP packets sent by this <code>Rss1Channel</code> .
unicastRcvd	The number of unicast UDP packets received by this <code>Rss1Channel</code> .
retransReqSent	The number of retransmission requests sent by this <code>Rss1Channel</code> . Retransmission requests are sent in an attempt to recover a missed packet and may indicate a network problem if gaps are also detected. This is populated only for reliable multicast type connections.
retransReqRcvd	This is the number of retransmission requests received by this <code>Rss1Channel</code> . Retransmission requests are received if another component on the network missed a packet sent by this channel and may indicate a network problem if gaps are also being detected. This is populated only for reliable multicast type connections.
retransPktsSent	The number of retransmitted packets sent by this <code>Rss1Channel</code> . Packets are retransmitted in response to retransmission requests. If a packet cannot be retransmitted, this results in a gap occurring and indicates a network problem, which applications are notified of via <code>rss1Read</code> . This is populated only for reliable multicast type connections.
retransPktsRcvd	The number of retransmitted packets received by this <code>Rss1Channel</code> . This is populated only for reliable multicast type connections.
gapsDetected	Returns a count of the number of detected packet gaps detected and reported to the application. This is a result of packet loss on the network and may indicate a more serious network problem.

Table 64: `multicastStats` Options

10.14.4 ComponentInfo Option

Option	DESCRIPTION
componentVersion	An <code>Rss1Buffer</code> containing an ASCII string that indicates the product version of the connected component.

Table 65: `componentInfo` Option

10.14.5 RsslServerInfo Structure Members

The following table describes values available to the user through the use of the `rsslGetServerInfo` function. This information is returned as part of the `RsslServerInfo` structure.

Structure Member	DESCRIPTION
currentBufferUsage	The number of currently used shared pool buffers across all users connected to the <code>RsslServer</code> .
peakBufferUsage	The maximum achieved number of used shared pool buffers across all users connected to the <code>RsslServer</code> . This value can be reset through the use of <code>rsslServerIoctl</code> , as described in Section 10.14.7.

Table 66: `RsslServerInfo` Structure Members

10.14.6 `rsslIoctl` Option Values

The following table provides a description of the options available for use with the `rsslIoctl` function.

OPTION ENUMERATION	DESCRIPTION
RSSL_MAX_NUM_BUFFERS	Allows an <code>RsslChannel</code> to change its <code>maxOutputBuffers</code> setting. Value should be pointer to <code>RsslUInt32</code> .
RSSL_NUM_GUARANTEED_BUFFERS	Allows an <code>RsslChannel</code> to change its <code>guaranteedOutputBuffers</code> setting. Value should be pointer to <code>RsslUInt32</code> .
RSSL_HIGH_WATER_MARK	Allows an <code>RsslChannel</code> to change the internal RSSL output queue depth water mark, which has a default value of 6,144 bytes. When the RSSL output queue exceeds this number of bytes, the <code>rsslWrite</code> function internally attempts to flush content to the network. Value should be pointer to <code>RsslUInt32</code> .
RSSL_SYSTEM_READ_BUFFERS	Allows an <code>RsslChannel</code> to change the TCP receive buffer size associated with the connection. Value should be pointer to <code>RsslUInt32</code> .
RSSL_SYSTEM_WRITE_BUFFERS	Allows an <code>RsslChannel</code> to change the TCP send buffer size associated with the connection. Value should be pointer to <code>RsslUInt32</code> .
RSSL_COMPRESSION_THRESHOLD	Allows an <code>RsslChannel</code> to change the size (in bytes) at which buffer compression occurs, must be greater than 30 bytes. Value should be pointer to <code>RsslUInt32</code> .
RSSL_PRIORITY_FLUSH_ORDER	Allows an <code>RsslChannel</code> to change its <code>priorityFlushStrategy</code> . Value should be a character array, where each entry is either: <ul style="list-style-type: none"> • H for high priority • M for medium priority • L for low priority The array should not exceed 32 characters. At least one H and one M must be present, however no L is required. If low priority flushing is not specified, the low priority queue is flushed only when other data is not available for output.
RSSL_TRACE	Allows an <code>RsslChannel</code> to write incoming and outgoing messages in XML format to a file and/or the stdout. The value should be a pointer to <code>RsslTraceOptions</code> .

Table 67: `rsslIoctl` Option Values

10.14.7 `rsslServerIoctl` Option Values

The following table provides a description of the options available for use with the `rsslServerIoctl` function.

OPTION ENUMERATION	DESCRIPTION
RSSL_SERVER_NUM_POOL_BUFFERS	Allows an <code>RsslServer</code> to change its <code>sharedPoolSize</code> setting. Value should be pointer to <code>RsslUInt32</code>
RSSL_SERVER_PEAK_BUF_RESET	Allows an <code>RsslServer</code> to reset the <code>peakBufferUsage</code> statistic. Value is not required.

Table 68: `rssl ServerIoctl` Option Values

10.15 XML Tracing

When using the RSSL transport layer with RWF, you can configure XML tracing on a per channel basis. Once enabled, tracing logs the contents of incoming and outgoing messages in an XML format. This data can be written to a file of the user's choice and/or to the stdout. XML tracing is configurable through `rsslIoctl` by using **RSSL_TRACE** for the `RsslIoctlCodes`. The **Value** field should be an `RsslTraceOptions` pointer, which holds associated configuration parameters.

10.15.1 RsslTraceOptions Structure Members

The following table describes the XML trace options available for use with the `RsslTraceOptions` structure.

STRUCTURE MEMBER	DESCRIPTION
traceMsgFileName	Sets the base, user-defined trace message file name. The Transport API appends the number of milliseconds since January, 1, 1970 and the <code>.xml</code> extension to this file name. If tracing to a file is enabled, you must provide a non-null <code>traceMsgFileName</code> in the initial call to <code>rsslIoctl</code> when using the RSSL_TRACE option. Subsequent calls to <code>rsslIoctl</code> to modify RSSL_TRACE options do not require a file name.
traceMsgMaxFileSize	Sets the maximum file size (in bytes) for the trace message file. If you enable the RSSL_TRACE_TO_MULTIPLE_FILES trace flag, a new file begins when the current long reaches this size. Otherwise, tracing to a file stops when this size is reached.
traceFlags	Combination of bit values that indicate additional <code>RsslTraceOptions</code> settings.

Table 69: `RsslTraceOptions` Structure Members

10.15.2 RsslTraceCodes Flag Enumeration Values

The following table describes the flag enumeration values for the `traceFlags` member of the `RsslTraceOptions` structure.

ENUMERATION	DESCRIPTION
RSSL_TRACE_READ	Sets the Transport API channel to trace messages read from the wire.
RSSL_TRACE_WRITE	Sets the Transport API channel to trace messages written to the wire.
RSSL_TRACE_PING	Sets the Transport API channel to trace ping messages.
RSSL_TRACE_HEX	Sets the Transport API channel to display hex values for all messages.
RSSL_TRACE_TO_FILE_ENABLE	Sets the Transport API channel to write the XML trace to a file. Specify the file name in the <code>traceMsgFileName</code> member of the <code>RsslTraceOptions</code> structure.
RSSL_TRACE_TO_MULTIPLE_FILES	Sets the Transport API channel to break up trace files according to the <code>traceMsgMaxFileSize</code> member of the <code>RsslTraceOptions</code> structure. If set to true, the Transport API creates a new trace file whenever the current trace file reaches <code>traceMsgMaxFileSize</code> bytes in size. All trace files receive a time stamp (appended to the base name as set by the <code>traceMsgFileName</code> member of the <code>RsslTraceOptions</code> structure) to differentiate it from previous files.
RSSL_TRACE_TO_STDOUT	Sets the Transport API channel to write the XML trace to stdout.

Table 70: `RsslTraceCodes` Option Values

Chapter 11 Data Package Detailed View

11.1 Concepts

The Data Package exposes a collection of types that can combine in a variety of ways to assist with modeling user's data. These types are split into two categories:

- A **Primitive Type** represents simple, atomically updating information. Primitive types represent values like integers, dates, and ASCII string buffers (refer to Section 11.2).
- A **Container Type** models more intricate data representations than Transport API primitive types and can manage dynamic content at a more granular level. Container types represent complex types like field identifier-value, name-value, or key-value pairs (refer to Section 11.3). The Transport API offers several uniform (i.e., homogeneous) container types whose entries house the same type of data. Additionally, there are several non-uniform (i.e., heterogeneous) container types in which different entries can hold different types of data.

Some system-level types are provided as simple **typedef** values and more complex types are provided as structure definitions that represent type contents. Primitive and Container types are also presented as a part of the **Rss1DataTypes** enumeration in the ranges:

- 0 to 127 are Primitive Types as described in Section 11.2.
- 128 to 255 are Container Types as described in Section 11.3.

Each type represented with an enumeration has a corresponding system **typedef** or structural definition used when encoding or decoding that type.

11.2 Primitive Types

A primitive type represents some type of base, system information (such as integers, dates, or array values). If contained in a set of updating information, primitive types update atomically (incoming data replaces any previously held values). Primitive types support ranges from simple primitive types (e.g., an integer) to more complex primitive types (e.g., an array).

The **Rss1DataTypes** enumeration includes values that define the type of a primitive:

- Values between 0 and 63 are **base primitive types**. Base primitive types support the full range of values allowed by the primitive type and are discussed in Table 71.

When contained in an **Rss1FieldEntry** or **Rss1ElementEntry**, base primitive types can also represent a **blank value**. A blank value indicates that no value is currently present and any previously stored or displayed primitive value should be cleared. When decoding any base primitive value, the interface function (See Table 71) returns **RSSL_RET_BLANK_DATA**. To encode blank data into an **Rss1FieldEntry** or **Rss1ElementEntry**, refer to Section 11.3.1 and Section 11.3.2.

- Values between 64 and 127 are **set-defined primitive types**, which define fixed-length encodings for many of the base primitive types (e.g., **RSSL_DT_INT_1** is a one byte fixed-length encoding of **RSSL_DT_INT**). These types can be leveraged only within a Set Definition and encoded or decoded as part of an **Rss1FieldList** or **Rss1ElementList**. Only certain set-defined primitive types can represent blank values. For more details about set-defined primitive types, refer to Section 11.6.

The following table provides a brief description of each base primitive type, along with interface functions used for encoding and decoding. Several primitive types have a more detailed description following the table.

ENUM TYPE	PRIMITIVE TYPE	TYPE DESCRIPTION	ENCODE INTERFACE	DECODE INTERFACE
RSSL_DT_UNKNOWN	None	Indicates that the type is unknown. RSSL_DT_UNKNOWN is valid only when decoding a Field List type and a dictionary look-up is required to determine the type. This type cannot be passed into encoding or decoding functions.	None	None
RSSL_DT_INT	RsslInt ^a	A signed integer type. Can currently represent a value of up to 63 bits along with a one bit sign (positive or negative).	rsslEncodeInt	rsslDecodeInt
RSSL_DT_UINT	RsslUInt ^b	An unsigned integer type. Can currently represent an unsigned value with precision of up to 64 bits.	rsslEncodeUInt	rsslDecodeUInt
RSSL_DT_FLOAT	RsslFloat	A four-byte, floating point type. Can represent the same range of values allowed with the system RsslFloat type. Follows IEEE 754 specification.	rsslEncodeFloat	rsslDecodeFloat
RSSL_DT_DOUBLE	RsslDouble	An eight-byte, floating point type. Can represent the same range of values allowed with the system RsslDouble type. Follows IEEE 754 specification.	rsslEncodeDouble	rsslDecodeDouble
RSSL_DT_REAL	RsslReal ^c	An optimized RWF representation of a decimal or fractional value which typically requires less bytes on the wire than RsslFloat or RsslDouble types. The user specifies a value with a hint for converting to decimal or fractional representation. For more details on this type, refer to Section 11.2.1.	rsslEncodeReal	rsslDecodeReal
RSSL_DT_DATE	RsslDate	Defines a date with month, day, and year values. For more details on this type, refer to Section 11.2.2.	rsslEncodeDate	rsslDecodeDate
RSSL_DT_TIME	RsslTime	Defines a time with hour, minute, second, millisecond, microsecond, and nanosecond values. For more details on this type, refer to Section 11.2.3.	rsslEncodeTime	rsslDecodeTime
RSSL_DT_DATETIME	RsslDateTime	Combined representation of date and time. Contains all members of RSSL_DT_DATE and RSSL_DT_TIME . For more details on this type, refer to Section 11.2.4.	rsslEncodeDateTime	rsslDecodeDateTime

Table 71: Transport API Primitive Types

ENUM TYPE	PRIMITIVE TYPE	TYPE DESCRIPTION	ENCODE INTERFACE	DECODE INTERFACE
RSSL_DT_QOS	RsslQos	Defines QoS information such as data timeliness (e.g., real time) and rate (e.g., tick-by-tick). Allows a user to send QoS information as part of the data payload. Similar information can also be conveyed using multiple Transport API message headers. For more details on this type, refer to Section 11.2.5.	rsslEncodeQos	rsslDecodeQos
RSSL_DT_STATE	RsslState	Represents data and stream state information. Allows a user to send state information as part of data payload. Similar information can also be conveyed in several Transport API message headers. For more details on this type, refer to Section 11.2.6.	rsslEncodeState	rsslDecodeState
RSSL_DT_ENUM	RsslEnum ^d	Represents an enumeration type, defined as an unsigned, two-byte value. Many times, this enumeration value is cross-referenced with an enumeration dictionary (e.g., enumtype.def) or a well-known, enumeration definition (e.g., those contained in rssIRDM.h).	rsslEncodeEnum	rsslDecodeEnum
RSSL_DT_ARRAY	RsslArray	The array type allows users to represent a simple base primitive type list (all primitive types except RsslArray). The user can specify the base primitive type that an array carries and whether each is of a variable or fixed-length. Because the array is a primitive type, if any primitive value in the array updates, the entire array must be resent. For more details on this type, refer to Section 11.2.7.	For more information, refer to Section 11.2.7.2.	For more information, refer to Section 11.2.7.5.
RSSL_DT_BUFFER	RsslBuffer ^e	Represents a raw byte buffer type. Any semantics associated with the data in this buffer is provided from outside of the Transport API, either via a field dictionary (e.g., RDMFieldDictionary) or a DMM definition. For more details on this type, refer to Section 11.2.8.	rsslEncodeBuffer	rsslDecodeBuffer

Table 71: Transport API Primitive Types (Continued)

ENUM TYPE	PRIMITIVE TYPE	TYPE DESCRIPTION	ENCODE INTERFACE	DECODE INTERFACE
RSSL_DT_ASCII_STRING	RsslBuffer ^e	Represents an ASCII string which should contain only characters that are valid in ASCII specification. Because this might be NULL terminated, use the provided length when accessing content. The Transport API does not enforce or validate encoding standards: this is the user's responsibility. For more details on this type, refer to Section 11.2.8.	rsslEncodeBuffer	rsslDecodeBuffer
RSSL_DT_UTF8_STRING	RsslBuffer ^e	Represents a UTF8 string which should follow the UTF8 encoding standard and contain only characters valid within that set. Because this might be NULL terminated, use the provided length when accessing content. The Transport API does not enforce or validate encoding standards: this is the user's responsibility. For more details on this type, refer to Section 11.2.8.	rsslEncodeBuffer	rsslDecodeBuffer
RSSL_DT_RMTEES_STRING	RsslBuffer ^e	Represents an RMTEES (Reuters Multilingual Text Encoding Standard) string which should follow the RMTEES encoding standard and contain only characters valid within that set. For more details on this type, refer to Section 11.2.8. The Transport API provide utility functions to help with proper storage and converting RMTEES strings. For more information, including examples refer to Section 11.2.9.	rsslEncodeBuffer	rsslDecodeBuffer

Table 71: Transport API Primitive Types (Continued)

- a. This type allows a value ranging from (-2⁶³) to (2⁶³ - 1).
- b. This type allows a value ranging from 0 up to (2⁶⁴ - 1).
- c. This type allows a value ranging from (-2⁶³) to (2⁶³ - 1). This can be combined with hint values to add or remove up to seven trailing zeros, fourteen decimal places, or fractional denominators up to 256.
- d. This type allows a value ranging from 0 to 65,535.
- e. The Transport API handles this type as opaque data, simply passing the length specified by the user and that number of bytes, no additional encoding or processing is done to any information contained in this type. Any specific encoding or decoding required for the information contained in this type is done outside of the scope of the Transport API, before encoding or after decoding this type. This type allows for a length of up to 65,535 bytes.

11.2.1 RsslReal

RsslReal is a structure that represents decimals or fractional values in a bandwidth-optimized format.

The **RsslReal** preserves the precision of encoded numeric values by separating the numeric value from any decimal point or fractional denominator. Developers should note that in some conversion cases, there may be a loss of precision; this is an example of a narrowing precision conversion. Because the IEEE 754 specification (used for **float** and **double** types) cannot represent some values exactly, rounding (per the IEEE 754 specification) may occur when converting between **RsslReal** representation and **float** or **double** representations, either using the provided helper methods or manually (using the conversion formulas provided). In cases where precision may be lost, converting to a string or using the provided string conversion helper as an intermediate point can help avoid the rounding precision loss.

11.2.1.1 Structure Members

RsslReal contains the following members:

Member	Description
isBlank	A Boolean value. Indicates whether data is considered blank. If true, other members should be ignored, if false other members determine the resultant value. This allows RsslReal to be represented as blank when used as either a primitive type or a set-defined primitive type.
hint	A hint enumeration value which defines how to interpret the value contained in RsslReal . Hint values can add or remove up to seven trailing zeros, 14 decimal places, or fractional denominators up to 256. For more information about hint values, refer to Table 73.
value	The raw value represented by the RsslReal (omitting any decimal or denominator). Typically requires application of the hint before interpreting or performing any calculations. This member can currently represent up to 63 bits and a one-bit sign (positive or negative).

Table 72: **Rssl Real** Structure Members

11.2.1.2 **hint** Values

The following table defines the available **hint** values for use with **RsslReal**. The conversion routines described in Section 11.2.1.3 use **RsslReal**'s **hint** and **value**.

ENUM	DESCRIPTION
RSSL_RH_EXPONENT_14	Negative exponent operation, equivalent to 10^{-14} . Shifts decimal by 14 positions.
RSSL_RH_EXPONENT_13	Negative exponent operation, equivalent to 10^{-13} . Shifts decimal by 13 positions.
RSSL_RH_EXPONENT_12	Negative exponent operation, equivalent to 10^{-12} . Shifts decimal by 12 positions.
RSSL_RH_EXPONENT_11	Negative exponent operation, equivalent to 10^{-11} . Shifts decimal by 11 positions.
RSSL_RH_EXPONENT_10	Negative exponent operation, equivalent to 10^{-10} . Shifts decimal by ten positions.
RSSL_RH_EXPONENT_9	Negative exponent operation, equivalent to 10^{-9} . Shifts decimal by nine positions.
RSSL_RH_EXPONENT_8	Negative exponent operation, equivalent to 10^{-8} . Shifts decimal by eight positions.
RSSL_RH_EXPONENT_7	Negative exponent operation, equivalent to 10^{-7} . Shifts decimal by seven positions.
RSSL_RH_EXPONENT_6	Negative exponent operation, equivalent to 10^{-6} . Shifts decimal by six positions.
RSSL_RH_EXPONENT_5	Negative exponent operation, equivalent to 10^{-5} . Shifts decimal by five positions.

Table 73: **Rssl Real** **Hi nts** Enumeration Values

ENUM	DESCRIPTION
RSSL_RH_EXPONENT_4	Negative exponent operation, equivalent to 10^{-4} . Shifts decimal by four positions.
RSSL_RH_EXPONENT_3	Negative exponent operation, equivalent to 10^{-3} . Shifts decimal by three positions.
RSSL_RH_EXPONENT_2	Negative exponent operation, equivalent to 10^{-2} . Shifts decimal by two positions.
RSSL_RH_EXPONENT_1	Negative exponent operation, equivalent to 10^{-1} . Shifts decimal by one position.
RSSL_RH_EXPONENT0	Exponent operation, equivalent to 10^0 . value does not change.
RSSL_RH_EXPONENT1	Positive exponent operation, equivalent to 10^1 . Depending on the type of conversion, this adds or removes one trailing zero.
RSSL_RH_EXPONENT2	Positive exponent operation, equivalent to 10^2 . Depending on the type of conversion, this adds or removes two trailing zeros.
RSSL_RH_EXPONENT3	Positive exponent operation, equivalent to 10^3 . Depending on the type of conversion, this adds or removes three trailing zeros.
RSSL_RH_EXPONENT4	Positive exponent operation, equivalent to 10^4 . Depending on the type of conversion, this adds or removes four trailing zeros.
RSSL_RH_EXPONENT5	Positive exponent operation, equivalent to 10^5 . Depending on the type of conversion, this adds or removes five trailing zeros.
RSSL_RH_EXPONENT6	Positive exponent operation, equivalent to 10^6 . Depending on the type of conversion, this adds or removes six trailing zeros.
RSSL_RH_EXPONENT7	Positive exponent operation, equivalent to 10^7 . Depending on the type of conversion, this adds or removes seven trailing zeros.
RSSL_RH_FRACTION_1	Fractional denominator operation, equivalent to 1/1. Value does not change.
RSSL_RH_FRACTION_2	Fractional denominator operation, equivalent to 1/2. Depending on the type of conversion, this adds or removes a denominator of two.
RSSL_RH_FRACTION_4	Fractional denominator operation, equivalent to 1/4. Depending on the type of conversion, this adds or removes a denominator of four.
RSSL_RH_FRACTION_8	Fractional denominator operation, equivalent to 1/8. Depending on the type of conversion, this adds or removes a denominator of eight.
RSSL_RH_FRACTION_16	Fractional denominator operation, equivalent to 1/16. Depending on the type of conversion, this adds or removes a denominator of 16.
RSSL_RH_FRACTION_32	Fractional denominator operation, equivalent to 1/32. Depending on the type of conversion, this adds or removes a denominator of 32.
RSSL_RH_FRACTION_64	Fractional denominator operation, equivalent to 1/64. Depending on the type of conversion, this adds or removes a denominator of 64.
RSSL_RH_FRACTION_128	Fractional denominator operation, equivalent to 1/128. Depending on the type of conversion, this adds or removes a denominator of 128.
RSSL_RH_FRACTION_256	Fractional denominator operation, equivalent to 1/256. Depending on the type of conversion, this adds or removes a denominator of 256.

Table 73: Rssl Real Hints Enumeration Values (Continued)

11.2.1.3 **hi nt** Use Case: Converting an **RsslReal** to a **Float** or a **Double**

An application can convert between an **RsslReal** and a system **float** or **double** as needed. Converting an **RsslReal** to a **double** or **float** is typically done to perform calculations or display data after receiving it.

The conversion process adds or removes decimal or denominator information from the value to optimize transmission sizes. In an **RsslReal** type, the decimal or denominator information is indicated by the **RsslReal.hint**, and the **RsslReal.value** indicates the value (less any decimal or denominator). If the **RsslReal.isBlank** member is **true**, this is handled as blank regardless of information contained in the **RsslReal.hint** and **RsslReal.value** members.

For this conversion, both the hint and its value are stored in the **RsslReal** structure. You can use the following example to perform this conversion, where **outputValue** is a system **float** or **double** to store output:

```
/* perform calculation and assign output to outputValue - may require appropriate float or
   double casts
 * depending on type of outputValue */
if (RsslReal.hint < RSSL_RH_FRACTION_1)
{
    /* insert the decimal point back into a decimal value */
    outputValue = rsslReal.value*(pow(10,(RsslReal.hint - RSSL_RH_EXPONENT0)));
}
else
{
    /* apply the denominator to the value to convert back to fraction */
    outputValue = RsslReal.value/(pow(2,(RsslReal.hint - RSSL_RH_FRACTION_1)));
}
```

Code Example 16: Rssl Real Conversion to Double/Float

11.2.1.4 **hi nt** Use Case: Converting Double or Float to an **RsslReal**

To convert a **double** or **float** type to an **RsslReal** type (typically done to prepare for transmission), the user must determine which hint value to use based on the type of value used:

- When converting a decimal value, the chosen hint value must be less than **RSSL_RH_FRACTION_1**.
- When converting a fractional value, the chosen hint value must be greater than or equal to **RSSL_RH_FRACTION_1**.

You can use the following example to perform the conversion, where **inputValue** is the unmodified input **float** or **double** value and **inputHint** is the hint chosen by the user:

```
/* rsslReal value is not blank so set to false */
RsslReal.isBlank = RSSL_FALSE;
/* store input hint value in the rsslReal structure */
RsslReal_hint = inputHint;

/* perform calculation and store output in rsslReal structure - may require appropriate
   * float or double casts depending on type of inputValue */

if (inputHint < RSSL_RH_FRACTION_1)
```

```

{
    /* removing the decimal point from a decimal value */
    RsslReal.value = floor(((inputValue)/(pow(10,(inputHint - RSSL_RH_EXPONENT0)))) + 0.5);
}
else
{
    /* removing the denominator from a fractional value */
    RsslReal.value = floor(((inputValue)*(pow(2,(inputHint - RSSL_RH_FRACTION_1)))) + 0.5);
}

```

Code Example 17: Rssl Real Conversion from Double/Float

11.2.1.5 Utility Functions

The Transport API provides the following utility functions for use with the **RsslReal** type:

UTILITY	DESCRIPTION
rsslClearReal	Sets all members in RsslReal to 0. isBlank is set to false .
rsslBlankReal	Sets all members in RsslReal to 0. isBlank is set to true .
rsslRealsEqual	Compares two RsslReal structures. Returns true if equal, false otherwise.
rsslDoubleToReal	Uses the formulas described in Section 11.2.1.4 to convert a system double to an RsslReal type.
rsslFloatToReal	Uses the formulas described in Section 11.2.1.4 to convert a system float to an RsslReal type.
rsslRealtoDouble	Uses the formulas described above in Section 11.2.1.3 to convert an RsslReal to a system double type.
rsslNumericStringtoDouble	Converts a numeric string with denominator or decimal information to a system double type.
rsslNumericStringToReal	Converts a numeric string with denominator or decimal information to an RsslReal type. Interprets string of +0 as a blank RsslReal .
rsslRealToString	Converts an RsslReal type to a numeric string representation. Blank is output as an empty zero length string.

Table 74: Rssl Real Utility Functions

11.2.2 Rss1Date

Rss1Date represents the date (i.e., **day**, **month**, and **year**) in a bandwidth-optimized fashion.

11.2.2.1 Structure Members

Rss1Date represents the date (i.e., **day**, **month**, and **year**) in a bandwidth-optimized fashion.

If **day**, **month**, and **year** are all set to **0** the **Rss1Date** is blank. If any individual member is represented as a blank value (**0**), only that member is blank. This is useful for representing dates which specify **month** and **year**, but not **day**. The **Rss1Date** type can be represented as blank when used as a primitive type and a set-defined primitive type.

MEMBER	DESCRIPTION
day	Represents the day of the month, where 0 indicates a blank entry. day allows a range of 0 to 255 , though the value typically does not exceed 31 .
month	Represents the month of the year, where 0 indicates a blank entry. month allows a range of 0 to 255 , though the value typically does not exceed 12 .
year	Represents the year, where 0 indicates a blank entry. You can use this member to specify a two- or four-digit year (where specific usage is indicated outside of the Transport API). year allows a range of 0 to 65,535 .

Table 75: **Rss1 Date** Structure Members

11.2.2.2 Utility Functions

The Transport API provides the following utility functions for use with the **Rss1Date** type:

UTILITY	DESCRIPTION
rss1ClearDate	Sets all members in Rss1Date to 0 . Because 0 represents a blank date value, this performs the same functionality as rss1BlankDate .
rss1BlankDate	Sets all members in Rss1Date to 0 . Because 0 represents a clear date value, this performs the same functionality as rss1ClearDate .
rss1DateisValid	Verifies the contents of a populated Rss1Date structure. Determines whether the specified day is valid within the specified month (e.g., a day greater than 31 is considered invalid for any month). This function uses the year member to determine leap year validity of day numbers for February. If Rss1Date is blank or valid, true is returned; False otherwise.
rss1DateIsEqual	Compares two Rss1Date structures. If equal, returns true ; false otherwise.
rss1DateStringToDate	Converts a string representation of a date to a populated Rss1Date structure. This function supports strftime() %d %b %Y format (e.g., 30 NOV 2010) or %m/%d/%y format (e.g., 11/30/2010).
Rss1DateTimeToString	Converts part of an Rss1DateTime structure to string (the appropriate portion must be populated with data). For example, if only converting Rss1Date to string, populates Rss1DateTime.date portion and passes in the dataType parameter as RSS1_DT_DATE . Outputs Rss1DateTime as a string in strftime() "%d %b %Y" format (e.g., 30 NOV 2010).

Table 76: **Rss1 Date** Utility Functions

11.2.3 RsslTime

RsslTime represents time (hour, minute, second, millisecond, microsecond, and nanosecond) in a bandwidth-optimized fashion. This type is represented as Greenwich Mean Time (GMT) unless noted otherwise¹.

11.2.3.1 Structure Members

RsslTime represents time (hour, minute, second, millisecond, microsecond, and nanosecond) in a bandwidth-optimized fashion. This type is represented as Greenwich Mean Time (GMT) unless noted otherwise².

RsslTime is a structure that uses the members listed in Table 77 (hour, minute, second, millisecond, microsecond, and nanosecond).

If all members are set to their respective blank values, **RsslTime** is blank. If any individual member is set to a blank value, only that member is blank. This is useful for representing times without **second**, **millisecond**, **microsecond**, or **nanosecond** values. The **RsslTime** type can be represented as blank when it is used as a primitive type and a set-defined primitive type.

Structure Member	DESCRIPTION
hour	Represents the hour of the day using a range of 0 to 255 (255 represents a blank hour value), though the value does not typically exceed 23 .
minute	Represents the minute of the hour using a range of 0 to 255 (255 represents a blank minute value), though the value does not typically exceed 59 .
second	Represents the second of the minute using a range of 0 to 255 (255 represents a blank second value), though the value does not typically exceed 59 .
millisecond	Represents the millisecond of the second using a range of 0 - 65,535 (65535 represents a blank millisecond value), though the value does not typically exceed 999 .
microsecond	Represents the microsecond of the millisecond using a range of 0 - 2047 (2047 represents a blank microsecond value), though the value does not typically exceed 999 .
nanosecond	Represents the nanosecond of the microsecond using a range of 0 - 2047 (where 2047 represents a blank nanosecond value), though the value does not typically exceed 999 .

Table 77: **RsslTime** Structure Members

11.2.3.2 Utility Functions

The Transport API provides the following utility functions for use with the **RsslTime** type:

FUNCTION NAME	DESCRIPTION
rsslClearTime	Sets all members in RsslTime to 0 .
rsslBlankTime	Sets all members in RsslTime to the values used to signify blank. A blank RsslTime contains hour , minute , and second values of 255 , a millisecond value of 65535 , and microsecond and nanosecond values of 2047 .

Table 78: **RsslTime** Utility Functions

1. The provider's documentation should indicate whether the providing application provides times in another representation.

2. The provider's documentation should indicate whether the providing application provides times in another representation.

FUNCTION NAME	DESCRIPTION
rsslTimelsValid	Verifies contents of populated <code>RsslTime</code> structure. Validates the ranges of the <code>hour</code> , <code>minute</code> , <code>second</code> , <code>millisecond</code> , <code>microsecond</code> , and <code>nanosecond</code> members. If <code>RsslTime</code> is blank or valid, <code>true</code> is returned; <code>false</code> otherwise.
rsslTimelsEqual	Compares two <code>RsslTime</code> structures. If equal, returns <code>true</code> ; <code>false</code> otherwise.
rsslTimeStringToTime	Converts a string representation of a time to a populated <code>RsslTime</code> structure. This function supports <code>strftime()</code> “%H:%M” (e.g., 15:24) or “%H:%M:%S” (e.g., 15:24:54) formats as well as “hour:minute:second:milli:micro:nano” format (e.g. 15:24:54:627:843:143).
rsslDateTimeToString	Takes an <code>RsslDateTime</code> structure populated with appropriate portion to convert. If only converting <code>RsslTime</code> to string, this function populates the <code>RsslDateTime.time</code> portion and passes in the <code>dataType</code> parameter as <code>RSSL_DT_TIME</code> . Outputs <code>RsslTime</code> as a string in “hour:minute:second:milli:micro:nano” format (e.g. 15:24:54:627:843:143).

Table 78: Rssl Time Utility Functions (Continued)

11.2.4 RsslDateTime

`RsslDateTime` represents the date (`date`) and time (`time`) in a bandwidth-optimized fashion. This time value is represented as Greenwich Mean Time (GMT) unless noted otherwise³.

11.2.4.1 Structure Members

`RsslDateTime` represents the date (`date`) and time (`time`) in a bandwidth-optimized fashion. This time value is represented as Greenwich Mean Time (GMT) unless noted otherwise⁴.

If `date` and `time` values are set to their respective blank values, `RsslDateTime` is blank. If any individual member is set to a blank value, only that member is blank. The `RsslDateTime` type can be represented as blank when it is used as a primitive type and a set-defined primitive type.

`RsslDateTime` is a structure with the following members:

MEMBER	DESCRIPTION
date	Represents the date values as an <code>RsslDate</code> structure and conforms to the behaviors described in Section 11.2.2.
time	Represents the time values as an <code>RsslTime</code> structure and conforms to the behaviors described in Section 11.2.3.

Table 79: Rssl DateTi me Structure Members

3. The provider's documentation should indicate whether the providing application provides times in another representation.

4. The provider's documentation should indicate whether the providing application provides times in another representation.

11.2.4.2 Utility Functions

The Transport API provides the following utility functions for use with `Rss1DateTime`:

FUNCTION NAME	DESCRIPTION
<code>rss1ClearDateTime</code>	Sets all members in <code>Rss1DateTime</code> to 0 .
<code>rss1BlankDateTime</code>	Sets all members in <code>Rss1DateTime</code> to their respective blank values. <ul style="list-style-type: none"> For date, all values are set to 0. For time, the hour, minute, and second are set to 255, millisecond is set to 65535, and microsecond and nanosecond are set to 2047.
<code>rss1DateTimelsValid</code>	Verifies the contents of a populated <code>Rss1DateTime</code> structure. Determines whether <code>day</code> is valid for the specified <code>month</code> (e.g., a <code>day</code> greater than 31 is considered invalid for any month) as determined by the specified <code>year</code> (to calculate whether it is a leap year). Also validates the range of <code>hour</code> , <code>minute</code> , <code>second</code> , <code>millisecond</code> , <code>microsecond</code> , and <code>nanosecond</code> members. If <code>Rss1DateTime</code> is blank or valid, <code>true</code> is returned; <code>false</code> otherwise.
<code>rss1DateTimelsEqual</code>	Compares two <code>Rss1DateTime</code> structures. Returns <code>true</code> if equal; <code>false</code> otherwise.
<code>rss1DateTimeStringToDateTIme</code>	Converts a string representation of a date and time to a populated <code>Rss1DateTime</code> structure. This function supports <code>date</code> values following <code>strftime() %d %b %Y</code> format (e.g., 30 NOV 2010) or <code>%m/%d/%y</code> format (e.g., 11/30/2010). This function supports <code>time</code> values that conform to <code>strftime() %H:%M</code> (e.g., 15:24) or <code>%H:%M:%S</code> (e.g., 15:24:54) formats as well as <code>hour:minute:second:milli:micro:nano</code> format (e.g., 15:24:54:627:843:143).
<code>rss1DateTimeToString</code>	Takes an <code>Rss1DateTime</code> structure populated with the appropriate portion to convert to a string representation. If converting a date and time to string, populates all portions of <code>Rss1DateTime</code> and passes in the <code>dataType</code> parameter as <code>RSS1_DT_DATETIME</code> . Outputs <code>Rss1DateTime</code> as a string in <code>%d %b %Y</code> <code>hour:minute:second:milli:micro:nano</code> format (e.g., 30 NOV 2010 15:24:54:627:843:143).

Table 80: `Rss1 DateTime` Utility Functions

11.2.5 RssiQos

RssiQos classifies data into two attributes:

- **Timeliness:** Conveys the age of data.
- **Rate:** Conveys the rate at which data changes.

Some timeliness or rate values allow you to provide additional time or rate data, for more details refer to Section 11.2.5.1, Section 11.2.5.2, Section 11.2.5.3, and Section 11.2.5.4.

If present in a data payload, specific handling and interpretation associated with QoS information is provided from outside of the Transport API, possibly via the specific DMM definition.

Several Transport API message headers also contain QoS data. When present, this data is typically used to request or convey the QoS associated with a particular stream. For more information about QoS use within a message, refer to Section 12.2.1 and Section 12.2.2. When conflated data is sent, additional conflation data might be included with update messages. For further details on conflation, refer to Section 12.2.3.

11.2.5.1 Structure Members

RssiQos is a structure with the following Structure Members:

Structure Member	DESCRIPTION
timeliness	Describes the age of the data (e.g., real time). Timeliness values are described in Section 11.2.5.2.
rate	Describes the rate at which the data changes (e.g., tick-by-tick). Rate values are described in Section 11.2.5.3.
dynamic	Describes the changeability of the QoS within the requested range, typically over the life of a data stream. <ul style="list-style-type: none"> • If set to false, the QoS should not change following the initial establishment. • If set to true, the QoS can change over time to other values within the requested range. QoS can change due to permissioning information, stream availability, network congestion, or other reasons. Specific information about dynamically changing QoS should be described in documentation for components that support this behavior.
timeInfo	Conveys detailed information about data timeliness , typically the amount of time delay. timeInfo allows for a range of 0 to 65,535 . This information is present only when timeliness is set to RSSL_QOS_TIME_DELAYED .
rateInfo	Conveys detailed information about rate , typically the interval of time during which data are conflated. Conflation combines multiple information updates into a single update, usually reducing network traffic. rateInfo allows for a range of 0 to 65,535 . This information is present only when rate is set to RSSL_QOS_RATE_TIME_CONFLATED .

Table 81: **Rssi Qos** Structure Members

11.2.5.2 Timeliness Enum Values

ENUM	DESCRIPTION
RSSL_QOS_TIME_UNSPECIFIED	timeliness is unspecified. Typically used by QoS initialization functions and not intended to be encoded or decoded.
RSSL_QOS_TIME_REALTIME	timeliness is real time: data is updated as soon as new data is available. This is the highest-quality timeliness value. In conjunction with a rate of RSSL_QOS_RATE_TICK_BY_TICK , real time is the best overall QoS.
RSSL_QOS_TIME_DELAYED_UNKNOWN	timeliness is delayed, though the amount of delay is unknown. This is a lower quality than RSSL_QOS_TIME_REALTIME and might be worse than RSSL_QOS_TIME_DELAYED (in which case the delay is known).
RSSL_QOS_TIME_DELAYED	timeliness is delayed and the amount of delay is provided in Rss1Qos.timeInfo . This is lower quality than RSSL_QOS_TIME_REALTIME and might be better than RSSL_QOS_TIME_DELAYED_UNKNOWN .

Table 82: **Rssi Qos** Timeliness Values

11.2.5.3 Rate Enum Values

ENUM	DESCRIPTION
RSSL_QOS_RATE_UNSPECIFIED	<code>rate</code> is unspecified. Typically used by QoS initialization functions and not intended to be encoded or decoded.
RSSL_QOS_RATE_TICK_BY_TICK	<code>rate</code> is tick-by-tick (i.e., data is sent for every update). This is the highest quality <code>rate</code> value. The best overall QoS is a tick-by-tick <code>rate</code> with a <code>timeliness</code> of <code>RSSL_QOS_TIME_REALTIME</code> .
RSSL_QOS_RATE_JIT_CONFLATED	<p><code>rate</code> is Just-In-Time (JIT) Conflated, meaning that quality is typically tick-by-tick, but if a data burst occurs (or if a component cannot keep up with tick-by-tick delivery), multiple updates are combined into a single update to reduce traffic. This value is usually considered a lower quality than <code>RSSL_QOS_RATE_TICK_BY_TICK</code>.</p> <p>Because JIT conflation is triggered by an application's inability to keep up with data rates, the effective rate depends on whether the application can sustain full data rates.</p> <p>Use of this value typically results in a rate similar to <code>RSSL_QOS_RATE_TICK_BY_TICK</code>. However, when the application cannot keep up with data rates, it results in a rate similar to <code>RSSL_QOS_RATE_TIME_CONFLATED</code>, where <code>rateInfo</code> is determined by the provider. Specific information about <code>conflationTime</code> or <code>conflationCount</code> might be present in an <code>RsslUpdateMsg</code>. For further details, refer to Section 12.2.3.</p>
RSSL_QOS_RATE_TIME_CONFLATED	<code>rate</code> is time-conflated. The interval of time (usually in milliseconds) over which data are conflated is provided in <code>RsslQos.rateInfo</code> . This is lower quality than <code>RSSL_QOS_RATE_TICK_BY_TICK</code> and at times even lower than <code>RSSL_QOS_RATE_JIT_CONFLATED</code> . Specific information about the <code>conflationTime</code> or <code>conflationCount</code> might be present in the <code>RsslUpdateMsg</code> . For more details, refer to Section 12.2.3.

Table 83: Rssi Qos Rate Values

11.2.5.4 Utility Functions

The Transport API provides the following utility functions for use with `RsslQos`:

FUNCTION NAME	DESCRIPTION
<code>rsslClearQos</code>	Sets all members in <code>RsslQos</code> to an initial value of <code>0</code> . This includes setting <code>rate</code> and <code>timeliness</code> to their unspecified values (which are not intended to be encoded or decoded).
<code>rsslCopyQos</code>	Copies one <code>RsslQos</code> into another.
<code>rsslQosIsEqual</code>	Compares two <code>RsslQos</code> structures. <ul style="list-style-type: none"> Returns <code>true</code> if the values contained in the structure are identical. Returns <code>false</code> if the values contained in the structure differ.

Table 84: Rssi Qos Utility Functions

FUNCTION NAME	DESCRIPTION
rssiQosIsBetter	Compares a new <code>RssiQos</code> with a previous <code>RssiQos</code> to determine which has better overall quality. <ul style="list-style-type: none"> Returns <code>true</code> if the new <code>RssiQos</code> is better. Returns <code>false</code> if the new <code>RssiQos</code> is not better.
rssiQosInRange	Determines whether a specified <code>RssiQos</code> lies within a range from best <code>RssiQos</code> to worst <code>RssiQos</code> . <ul style="list-style-type: none"> Returns <code>true</code> if the specified value inclusively falls between best and worst <code>RssiQos</code> Returns <code>false</code> if the value falls outside of the best or worst <code>RssiQos</code> range.

Table 84: Rssi Qos Utility Functions (Continued)

11.2.6 RssiState

`RssiState` conveys data and stream health information. When present in a header, `RssiState` applies to the state of the stream and data. When present in a data payload, the meaning of `RssiState` should be defined by the DMM.

Several Transport API message headers also contain `RssiState` data. When present in a message header, `RssiState` typically conveys the overall data and stream health of messages flowing over a particular stream. For more information on using `RssiState` in a message, refer to Section 12.2.1, Section 12.2.2, and Section 12.2.4. A decision table that provides example behaviors for various state combinations is available in Appendix A, Item and Group State Decision Table.

11.2.6.1 Structure Members

`RssiState` contains the following members:

Structure Member	DESCRIPTION
streamState	Conveys data about the stream's health. <code>streamState</code> values are described in Section 11.2.6.2.
dataState	Conveys data about the health of data flowing within a stream. <code>dataState</code> values are described in Section 11.2.6.3.
code	An enumerated code value that conveys additional information about the current state. Typically indicates more specific information (e.g., pertaining to a condition occurring upstream causing current data and stream states). <code>code</code> is typically used for informational purposes. State Code values are described in Section 11.2.6.4. Note: An application should not trigger specific behavior based on this content.
text	Specific <code>text</code> regarding the current data and stream state. Typically used for informational purposes. Encoded <code>text</code> has a maximum allowed length of 32,767 bytes. Note: An application should not trigger specific behavior based on this content.

Table 85: Rssi State Structure Members

11.2.6.2 Stream State Enum Values

ENUM	DESCRIPTION
RSSL_STREAM_UNSPECIFIED	<code>streamState</code> is unspecified. Typically used as a structure initialization value and is not intended to be encoded or decoded.
RSSL_STREAM_OPEN	<code>streamState</code> is open. This typically means that data is streaming: as data changes, they are sent on the stream.
RSSL_STREAM_NON_STREAMING	<code>streamState</code> is non-streaming. After receiving a final <code>RsslRefreshMsg</code> or <code>RsslStatusMsg</code> , the stream is closed and updated data is not delivered without a subsequent re-request. Update messages might still be received between the first and final part of a multi-part refresh. For further details, refer to Section 13.1.
RSSL_STREAM_CLOSED_RECOVER	<code>streamState</code> is closed, however data can be recovered on this service and connection at a later time. This state can occur via either an <code>RsslRefreshMsg</code> or an <code>RsslStatusMsg</code> . Single Open behavior can modify this state (continuing to indicate a stream state of <code>RSSL_STREAM_OPEN</code>) and attempt to recover data on the user's behalf. For further details on Single Open behavior, refer to Section 13.5.
RSSL_STREAM_CLOSED	<code>streamState</code> is closed. Data is not available on this service and connection and is not likely to become available, though the data might be available on another service or connection. This state can result from either an <code>RsslRefreshMsg</code> or an <code>RsslStatusMsg</code> .
RSSL_STREAM_REDIRECTED	<code>streamState</code> is redirected. The current stream is closed and has new identifying information. The user can issue a new request for the data using the new message key data from the redirect message. This state can result from either an <code>RsslRefreshMsg</code> or an <code>RsslStatusMsg</code> . For further details, refer to Section 12.1.3.3.

Table 86: Rssl State Stream State Values

11.2.6.3 Data State Enum Values

ENUM	DESCRIPTION
RSSL_DATA_NO_CHANGE	Indicates there is no change in the current state of the data. When available, it is preferable to send more concrete state information (such as <code>OK</code> or <code>SUSPECT</code>) instead of <code>NO_CHANGE</code> . This typically conveys <code>code</code> or <code>text</code> information associated with an item group, but no change to the group's previous data and stream state has occurred.
RSSL_DATA_OK	<code>dataState</code> is <code>OK</code> . All data associated with the stream is healthy and current.
RSSL_DATA_SUSPECT	<code>dataState</code> is <code>SUSPECT</code> (also known as a stale-data state). A suspect data state means some or all of the data on a stream is out-of-date (or that it cannot be confirmed as current, e.g., the service is down). If an application does not allow suspect data, a stream might change from open to closed or closed recover as a result. For further details, refer to Section 13.5.

Table 87: Rssl State Data State Values

11.2.6.4 Code Values

ENUMERATED NAME	DESCRIPTION
RSSL_SC_NONE	Indicates that additional state code information is not required, nor present.
RSSL_SC_NOT_FOUND	Indicates that requested information was not found, though it might be available at a later time or through changing some parameters used in the request.
RSSL_SC_TIMEOUT	Indicates that a timeout occurred somewhere in the system while processing requested data.
RSSL_SC_NOT_ENTITLED	Indicates that the request was denied due to permissioning. Typically indicates that the requesting user does not have permission to request on the service, to receive requested data, or to receive data at the requested QoS.
RSSL_SC_INVALID_ARGUMENT	Indicates that the request includes an invalid or unrecognized parameter. Specific information should be contained in the text .
RSSL_SC_USAGE_ERROR	Indicates invalid usage within the system. Specific information should be contained in the text .
RSSL_SC_PREEMPTED	Indicates the stream was preempted, possibly by a caching device. Typically indicates the user has exceeded an item limit, whether specific to the user or a component in the system. Relevant information should be contained in the text .
RSSL_SC_JIT_CONFLATION_STARTED	Indicates that JIT conflation has started on the stream. User is notified when JIT Conflation ends via RSSL_SC_REALTIME_RESUMED .
RSSL_SC_REALTIME_RESUMED	Indicates that JIT conflation on the stream has finished.
RSSL_SC_FAILOVER_STARTED	Indicates that a component is recovering due to a failover condition. User is notified when recovery finishes via RSSL_SC_FAILOVER_COMPLETED .
RSSL_SC_FAILOVER_COMPLETED	Indicates that recovery from a failover condition has finished.
RSSL_SC_GAP_DETECTED	Indicates that a gap was detected between messages. A gap might be detected via an external reliability mechanism (e.g., transport) or using the seqNum present in Transport API messages.
RSSL_SC_NO_RESOURCES	Indicates that no resources are available to accommodate the stream.
RSSL_SC_TOO_MANY_ITEMS	Indicates that a request cannot be processed because too many other streams are already open.
RSSL_SC_ALREADY_OPEN	Indicates that a stream is already open on the connection for the requested data.

Table 88: Rssl State Code Values

ENUMERATED NAME	DESCRIPTION
RSSL_SC_SOURCE_UNKNOWN	Indicates that the requested service is not known, though the service might be available at a later point in time.
RSSL_SC_NOT_OPEN	Indicates that the stream was not opened. Additional information should be available in the text .
RSSL_SC_NON_UPDATING_ITEM	Indicates that a streaming request was made for non-updating data.
RSSL_SC_UNSUPPORTED_VIEW_TYPE	Indicates that the domain on which a request is made does not support the requested viewType . Section 13.8 discusses views in more detail.
RSSL_SC_INVALID_VIEW	Indicates that the requested view is invalid, possibly due to bad formatting. Additional information should be available in the text . Section 13.8 discusses views in more detail.
RSSL_SC_FULL_VIEW_PROVIDED	Indicates that the full view (e.g., all available fields) is being provided, even though only a specific view was requested. Section 13.8 discusses views in more detail.
RSSL_SC_UNABLE_TO_REQUEST_AS_BATCH	Indicates that a batch request cannot be used for this request. The user can instead split the batched items into individual requests. Section 13.7 discusses batch requesting in more detail.
RSSL_SC_NO_BATCH_VIEW_SUPPORT_IN_REQ	Indicates that the provider does not support batch and/or view functionality.
RSSL_SC_EXCEEDED_MAX_MOUNTS_PER_USER	Indicates that the login was rejected because the user exceeded their maximum number of allowed mounts.
RSSL_SC_ERROR	Indicates an internal error from the sender.
RSSL_SC_DACS_DOWN	Indicates that the connection to DACS is down and users are not allowed to connect.
RSSL_SC_USER_UNKNOWN_TO_PERM_SYS	Indicates that the user is unknown to the permissioning system and is not allowed to connect.
RSSL_SC_DACS_MAX_LOGINS_REACHED	Indicates that the maximum number of logins has been reached.
RSSL_SC_DACS_USER_ACCESS_TO_APP_DENIED	Indicates that the application is denied access to the system.
RSSL_SC_GAP_FILL	Indicates that the received content is meant to fill a recognized gap.
RSSL_SC_APP_AUTHORIZATION_FAILED	Indicates that application authorization using the secure token has failed.

Table 88: [Rssi State](#) Code Values(Continued)

11.2.6.5 Utility Functions

The Transport API provides the following utility functions for use with `RsslState`:

FUNCTION NAME	DESCRIPTION
<code>rsslClearState</code>	Sets all members in <code>RsslState</code> to an initial value. This includes setting <code>streamState</code> to its unspecified value, which is not intended to be encoded or decoded.
<code>rsslIsFinalState</code>	<ul style="list-style-type: none"> Returns <code>true</code> if the <code>RsslState</code> represents a final state for a stream (e.g. stream is Closed, Closed Recover, Redirected, or NonStreaming). Returns <code>false</code> if the <code>RsslState</code> is not final.
<code>rsslStreamStateToString</code>	Converts an <code>RsslState.streamState</code> enum to a string representation.
<code>rsslDataStateToString</code>	Converts an <code>RsslState.dataState</code> enum to a string representation.
<code>rsslToString</code>	Converts an <code>RsslState.code</code> enum to a string representation.
<code>rsslStreamStateInfo</code>	Converts an <code>RsslState.streamState</code> enum to text (e.g., <code>RSSL_DATA_OK</code> to "Data state is OK") and returns as a string for the <code>streamState</code> .
<code>rsslDataStateInfo</code>	Converts an <code>RsslState.dataState</code> enum to text (e.g., <code>RSSL_DATA_OK</code> to "Data state is OK") and returns as a string for the <code>dataState</code> .
<code>rsslStateCodeInfo</code>	Takes an <code>RsslState.code</code> enum to text (e.g., <code>RSSL_DATA_OK</code> to "Data state is OK") and returns as a string for the <code>code</code> .

Table 89: `Rssl State` Utility Functions

11.2.7 RsslArray

The **RsslArray** is a uniform primitive type that can contain multiple simple primitive entries. An **RsslArray** can contain zero to N primitive type entries⁵, where zero entries indicates an empty **RsslArray**.

Each **RsslArray** entry can house only simple primitive types such as **RsslInt**, **RsslReal**, or **RsslDate**. An **RsslArray** entry cannot house any container types or other **RsslArray** types. This is a uniform type, where **RsslArray.primitiveType** indicates the single, simple primitive type of each entry. **RsslArray** uses simple replacement rules for change management. When new entries are added, or any array entry requires a modification, all entries must be sent with the **RsslArray**. This new **RsslArray** entirely replaces any previously stored or displayed data.

An **RsslArray** entry can be encoded from pre-encoded data or by encoding individual pieces of data as provided. The **RsslArray** does not use a specific entry structure. When encoding, the application passes a pointer to the primitive type value (when data is not encoded) or an **RsslBuffer** (containing the pre-encoded primitive).

When decoding, an **RsslBuffer** structure is given, which provides access to the encoded content of the array entry. Further decoding of the entry's content can be skipped by invoking the entry decoder to move to the next **RsslArray** entry or the contents can be further decoded by invoking the specifically contained type's primitive decode function (refer to Section 11.2).

Note: Although it can house other primitive types, **RsslArray** is itself considered a primitive type and can be represented as a blank value.

11.2.7.1 Structure Members

STRUCTURE MEMBER	DESCRIPTION
primitiveType	Using an RsslDataTypes enum, primitiveType describes the base primitive type of each entry. RsslArray can only contain simple primitive types and cannot house container types or other RsslArrays .
itemLength ^a	Sets the expected length of all array entries. <ul style="list-style-type: none"> If set to 0, entries are variable length and each encoded entry can have a different length. If set to a non-zero value, each entry must be the specified length (e.g. sending primitiveType of RSSL_DT_ASCII_STRING with itemLength set to 3 indicates that each array entry will be a fixed-length three-byte string). When using a fixed length, the application still passes in the base primitive type when encoding (e.g., if encoding fixed length RSSL_DT_INT types, an RsslInt is passed in regardless of itemLength). When encoding buffer types as fixed length: <ul style="list-style-type: none"> Any content that exceeds itemLength will be truncated. Any content that is shorter than itemLength will be padded with the \0 (NULL) character.
encData	Length and pointer to all encoded primitive types in the contents (if any). This refers to encoded RsslArray payload and length information.

Table 90: **Rssl Array** Structure Members

a. Only specific types are allowed as fixed-length encodings. **RSSL_DT_INT** and **RSSL_DT_UINT** can support one-, two-, four-, or eight-byte fixed lengths. **RSSL_DT_TIME** supports three- or five-byte fixed lengths. **RSSL_DT_DATETIME** supports seven- or nine-byte fixed lengths. **RSSL_DT_ENUM** supports one- or two-byte fixed lengths. **RSSL_DT_BUFFER**, **RSSL_DT_ASCII_STRING**, **RSSL_DT_UTF8_STRING**, and **RSSL_DT_RMAP_STRING** support any legal length value; see those types for allowable lengths.

5. An **Rssl Array** currently has a maximum entry count of 65,535. This type has an approximate maximum encoded length of 4 gigabytes but may be limited to 65,535 bytes if housed inside of a container entry. The content of an **Rssl Array** entry is bound by the maximum encoded length of the primitive types being contained. These limitations can change in subsequent releases.

11.2.7.2 Encoding Interfaces

Encode Interface	Description
rsslEncodeArrayInit	Begins encoding an <code>RsslArray</code> . This function expects that the members <code>RsslArray.primitiveType</code> and <code>RsslArray.itemLength</code> are properly populated. The <code>RsslEncodeIterator</code> specifies the <code>RsslBuffer</code> into which it encodes data. Entries can be encoded after this function returns.
rsslEncodeArrayComplete	Completes encoding of an <code>RsslArray</code> . This function expects the same <code>RsslEncodeIterator</code> used with <code>rsslEncodeArrayInit</code> and <code>rsslEncodeArrayEntry</code> . Set the <code>RsslBool</code> parameter to: <ul style="list-style-type: none"> • <code>true</code> if encoding was successful, to finish encoding. • <code>false</code> if encoding of any entry failed, to roll back encoding to the last successfully-encoded point in the contents. All entries should be encoded before calling <code>rsslEncodeArrayComplete</code> .
rsslEncodeArrayEntry	Encodes an <code>RsslArray</code> entry. This function expects the <code>RsslEncodeIterator</code> which was used with <code>rsslEncodeArrayInit</code> . <ul style="list-style-type: none"> • If encoding from pre-encoded data, this can be passed in via the <code>RsslBuffer*</code> parameter, and the <code>void*</code> parameter should be passed in as NULL. • If encoding from a primitive type, a pointer to the populated primitive type should be passed via the <code>void*</code> and the <code>RsslBuffer*</code> should be passed in as NULL. This function should be called for each entry being encoded. The passed-in type must match <code>RsslArray.primitiveType</code> .

Table 91: `Rssl Array` Encode Functions

11.2.7.3 Encoding: Example 1

The following code samples demonstrate how to encode an `RsslArray`. In the first example, the array is set to encode unsigned integer entries, where the entries have a fixed length of two bytes each. The example encodes two array entries. The first entry is encoded from a primitive `RsslUInt` type; the second entry is encoded from an `RsslBuffer` containing a pre-encoded `RsslUInt` type. The example includes error handling for the initial encode function only, and omits additional error handling to simplify the sample code.

```
/* EXAMPLE 1 - Array of fixed length unsigned integer values */

/* populate array structure prior to call to rsslEncodeArrayInit encode unsigned integers in
   the array */
rsslArray.primitiveType = RSSL_DT_UINT;
/* send fixed length values where each uint is 2 bytes */
rsslArray.itemLength = 2;

/* begin encoding of array - assumes that encIter is already populated with buffer and version
   information, store return value to determine success or failure */
if ((RetVal = rsslEncodeArrayInit(&encIter, &rsslArray)) < RSSL_RET_SUCCESS)
{
    /* error condition - switch our success value to false so we can roll back */
    success = RSSL_FALSE;
    /* print out message with return value string, value, and text */
}
```

```

printf("Error %s (%d) encountered with rsslEncodeArrayInit. Error Text: %s\n",
       rsslRetCodeToString(retval), retval, rsslRetCodeInfo(retval));
}
else
{
    RsslUInt uInt = 23456;
    /* array encoding was successful */

    /* encode first entry from a UInt from a primitive type */
    retval = rsslEncodeArrayEntry(&encIter, NULL, &uInt);

    /* encode second entry from a UInt from pre-encoded integer contained in a buffer */
    /* this buffer.data should point to encoded int and the length should be number of bytes
     * encoded */
    retval = rsslEncodeArrayEntry(&encIter, pEncUInt, NULL);

}

/* complete array encoding. If success parameter is true, this will finalize encoding. */
/* If success parameter is false, this will roll back encoding prior to rsslEncodeArrayInit */
retval = rsslEncodeArrayComplete(&encIter, success);

```

Code Example 18: Rssl Array Encoding Example #1

11.2.7.4 Encoding: Example 2

This example demonstrates encoding an **RsslArray** containing ASCII string values. The example includes error handling for the initial encode function only, and omits additional error handling to simplify the sample code.

```

/* EXAMPLE 2 - Array of variable length ASCII string values populate array structure prior to
   call */
/* to rsslEncodeArrayInit encode ASCII Strings in the array */
RsslBuffer stringBuf = RSSL_INIT_BUFFER;

rsslArray.primitiveType = RSSL_DT_ASCII_STRING;
/* itemLength 0 indicates variable length entries */
rsslArray.itemLength = 0;

/* begin encoding of array - assumes that encIter is already populated with buffer and version
   information, store return value to determine success or failure */
if ((retval = rsslEncodeArrayInit(&encIter, &rsslArray)) < RSSL_RET_SUCCESS)
{
    /* error condition - switch our success value to false so we can roll back */
    success = RSSL_FALSE;
    /* print out message with return value string, value, and text */
    printf("Error %s (%d) encountered with rsslEncodeArrayInit. Error Text: %s\n",

```

```

        rsslRetCodeToString(retval), retval, rsslRetCodeInfo(retval));
}
else
{
    stringBuf.data = 'ENTRY 1';
    stringBuf.length = 7;
    /* array encoding was successful */

    /* encode first entry from a buffer containing an ASCII_STRING primitive type */
    retval = rsslEncodeArrayEntry(&encIter, NULL, &stringBuf);

}

/* complete array encoding. If success parameter is true, this will finalize encoding.
   If success parameter is false, this will roll back encoding prior to rsslEncodeArrayInit */
retval = rsslEncodeArrayComplete(&encIter, success);

```

Code Example 19: Rssl Array Encoding Example #2

11.2.7.5 Decoding Interfaces

DECODE INTERFACE	DESCRIPTION
rsslDecodeArray	Begins decoding an RsslArray . This function decodes from the RsslBuffer referred to by the passed-in RsslDecodeIterator .
rsslDecodeArrayEntry	Decodes an RsslArray entry and populates RsslBuffer with encoded entry contents. This function expects the same RsslDecodeIterator which was used with rsslDecodeArray . Any contained primitive type's decode function can be called based on RsslArray.primitiveType (refer to Table 71). Calling rsslDecodeArrayEntry again will decode and provide the next entry in the RsslArray until no more entries are available.

Table 92: Rssl Array Decode Functions

11.2.7.6 Decoding: Example

The following example decodes an `RsslArray` and each of its entries to the primitive value. This sample code assumes the contained primitive type is an `RsslUInt`. Typically an application invokes the specific primitive decoder for the contained type or uses a switch statement to allow for a more generic array entry decoder. This example uses the same `RsslDecodeIterator` when calling the primitive decoder function. An application could optionally use a new `RsslDecodeIterator` by setting the encoded entry buffer on a new iterator. To simplify the example, some error handling is omitted.

```

/* decode into the array structure header */
if ((retVal = rsslDecodeArray(&decIter, &rsslArray)) >= RSSL_RET_SUCCESS)
{
    /* decode each array entry */
    while ((retVal = rsslDecodeArrayEntry(&decIter, &entryBuffer)) != RSSL_RET_END_OF_CONTAINER)
    {
        if (retVal < RSSL_RET_SUCCESS)
        {
            /* decoding failure tends to be unrecoverable */
            printf("Error %s (%d) encountered with rsslDecodeArrayEntry. Error Text: %s\n",
                   rsslRetCodeToString(retVal), retVal, rsslRetCodeInfo(retVal));
        }
        else
        {
            /* decode array entry into primitive type we can use the same decode iterator, */
            /* or set the encoded entry buffer onto a new iterator */
            retVal = rsslDecodeUInt(&decIter, &uInt);
        }
    }
}
else
{
    /* decoding failure tends to be unrecoverable */
    printf("Error %s (%d) encountered with rsslDecodeArray. Error Text: %s\n",
           rsslRetCodeToString(retVal), retVal, rsslRetCodeInfo(retVal));
}

```

11.2.7.7 Utility Functions

FUNCTION NAME	DESCRIPTION
<code>rsslClearArray</code>	Sets all members in <code>RsslArray</code> to an initial value.

Table 93: **Rssl Array** Utility Functions

11.2.8 RsslBuffer

RsslBuffer represents some type of user-provided content along with the content's length. **RsslBuffer** can:

- Represent various buffer and string types, such as ASCII, RMTEs, or UTF8 strings.
- Contain or reference encoded data on both container and message header structures.

No validation or enforcement checks are performed on the contents of an **RsslBuffer**. Any desired validation can be performed by the user depending on the specific type of content represented by **RsslBuffer**. Null termination is not required with this type.

Blank buffers are conveyed as an **RsslBuffer.length** of **0**.

11.2.8.1 Structure Members

RsslBuffer is a structure with the following members:

STRUCTURE MEMBER	DESCRIPTION
length	The length, in bytes, of the content pointed to by data .
data	Points to some type of content, where the specific type description of the content is provided outside of Transport API via an external source (domain model definition, field dictionary, etc.). The length member should be set to the number of bytes pointed to by data .

Table 94: **Rssl Buffer** Structure Members

11.2.8.2 Example

For performance purposes contents are not copied while decoding **RsslBuffer** type. This may result in the **RsslBuffer.data** exposing additional encoded contents beyond the **RsslBuffer.length** to be exposed. The user can determine appropriate handling to suit their needs. Some options are to copy contents and add NULL termination or use appropriate **printf** modifiers to only display specified content length as illustrated in the following example:

```
/* display only the specified length of RsslBuffer contents */
printf("%.*s", RsslBuffer.length, RsslBuffer.data);
```

Code Example 20: Displaying Contents of an **Rssl Buffer**

11.2.9 RMTES Decoding

Use special consideration when handling and converting `Rss1Buffer`s that contain RMTES data. This allows for the application of partial content updates, used to efficiently change already received RMTES content by sending only those portions that need to be changed. For a more detailed description of RMTES, refer to the *Reuters Multilingual Text Encoding Standard Specification*.

The typical process for handling RMTES content contained in an `Rss1Buffer` involves storing content, applying partial updates, and converting to the desired character set. The Transport API provides several structures and functions to help with this storage and conversion as described in the following sections.

 **Warning!** RMTES processing is an expensive procedure that incurs multiple content copies. To avoid unnecessary processing, users should confirm that content providers are actually sending RMTES prior to using this function. If the content type is not RMTES, do not use this function^a.

- a. Although the type specified in the field dictionary may indicate RMTES, the actual content might not be encoded as such. Unless content uses RMTES encoding, this functionality is not necessary.

11.2.9.1 Rss1RmtesCacheBuffer: Structure

The `Rss1RmtesCacheBuffer` is a simple structure used to store initial RMTES content and when applying partial updates. Any character set conversions should be performed on the content stored in the `Rss1RmtesCacheBuffer`.

`Rss1RmtesCacheBuffer` includes the following members:

STRUCTURE MEMBER	DESCRIPTION
length	Returns the length (in bytes) of the content pointed to by <code>data</code> ; it represents the number of bytes used in the cache. For example, if <code>data</code> refers to 100 bytes and nothing is cached, <code>length</code> should be set to 0. If <code>data</code> refers to 100 bytes, and 50 bytes are currently in cache, <code>length</code> should be set to 50.
data	Points to the RMTES content. The <code>length</code> member should be set to the number of bytes pointed to by <code>data</code> .
allocatedLength	Returns the length (in bytes) allocated when creating <code>data</code> . This is typically larger than <code>length</code> to allow for the growth of <code>data</code> when applying future partial updates.

Table 95: `Rss1 RmtesCacheBuffer` Structure Members

11.2.9.2 Rss1RmtesCacheBuffer: Decoding Interfaces

DECODE INTERFACE	DESCRIPTION
<code>rssIRMTESSApplyToCache(Buffer, RmtesCacheBuffer)</code>	<p>Applies encoded RMTES content to the <code>Rss1RmtesCacheBuffer.data</code>. If any partial update commands reside in the given <code>Rss1Buffer</code>, these are applied as well. Generally, one of the character set conversion functions are invoked after content is applied to the cache buffer.</p> <p>Note: The <code>Rss1RmtesCacheBuffer.data</code> must refer to enough memory for storing and modifying the RMTES content</p>

Table 96: `Rss1 RmtesCacheBuffer` Decode Functions

11.2.9.3 RsslRmtesCacheBuffer: Utility Functions

FUNCTION NAME	DESCRIPTION
<code>rsslHasPartialRMTESUpdate</code>	<ul style="list-style-type: none"> If RMTES content in an <code>RsslBuffer</code> contains a partial update command, returns <code>true</code>. If RMTES content in an <code>RsslBuffer</code> does not contain a partial update command, returns <code>false</code>.

Table 97: `Rssl RmtesCacheBuffer` Utility Functions

11.2.9.4 Conversion Functionality: Interfaces and Structure Members

With the Transport API, users can convert RMTES content to either the [UTF-8](#) or [UCS-2](#) character sets. Both character sets can represent the full spectrum of RMTES content. The user can decide which character set best meets their requirements.

DECODE INTERFACE	DESCRIPTION
<code>rssIRMTESToUTF8</code>	Converts cached RMTES content into UTF-8 Unicode content. This converts from <code>RsslRmtesCacheBuffer.data</code> into the memory provided in the <code>RsslBuffer</code> . Note: The <code>RsslBuffer</code> must have access to enough memory for the conversion process. Conversion will typically need a similar size memory as required by the storage.
<code>rssIRMTESToUCS2</code>	Converts cached RMTES content into UCS-2 Unicode. This converts from <code>RsslRmtesCacheBuffer.data</code> into the memory provided in the <code>RsslU16Buffer.data</code> . The <code>RsslU16Buffer</code> members are defined in Table 99. Note: The <code>RsslU16Buffer</code> must have access to enough memory for the conversion process, which is similar in size as the memory required by storage.

Table 98: RMTES to Unicode Conversion Functions

The `RsslU16Buffer` is used only when converting from RMTES to UCS-2. The structure has the following members:

STRUCTURE MEMBER	DESCRIPTION
<code>length</code>	The length (in 16-bit unsigned integers) of the content pointed to by <code>data</code> .
<code>data</code>	Points to the memory to use prior to conversion and the UCS-2 content after conversion. The <code>length</code> member should be set to the number of bytes pointed to by <code>data</code> .

Table 99: `Rssl U16Buffer` Structure Members

11.2.9.5 Example: Converting RMTEs to UTF-8

The following example illustrates how to store and convert RMTEs content. This example converts from RMTEs to UTF-8 and assumes that:

- The input buffer is populated with RMTEs content.
- The allocated size of 100 bytes is sufficient for conversion and storage.

To simplify the example, some error handling is omitted.

```

/* create cache buffer for storing RMTEs and applying partial updates */
RsslRmtesCacheBuffer rmtesCache;
char cacheSpace[100];
/* create RsslBuffer to convert into */
RsslBuffer utf8Buffer;
char convertSpace[100];

/* populate cache and conversion buffers with created memory */
rmtesCache.data = cacheSpace;
rmtesCache.length = 0; /* this is the used length, since cache starts out empty it should
   start at 0 */
rmtesCache.allocatedLength = 100;

utf8Buffer.data = convertSpace;
utf8Buffer.length = 100;

/* apply RMTEs content to cache, if successful convert to UTF-8 */
if ((RetVal = rsslRMTESApplyToCache(&inputBuf, &rmtesCache)) < RSSL_RET_SUCCESS)
{
    /* error while applying to cache */
    printf("Error %s (%d) encountered while applying buffer to RMTEs cache. Error Text: %s\n",
           rsslRetCodeToString(RetVal), RetVal, rsslRetCodeInfo(RetVal));
}
else if ((RetVal = rsslRMTESToUTF8(&rmtesCache, &utf8Buffer)) < RSSL_RET_SUCCESS)
{
    /* error when converting */
    printf("Error %s (%d) encountered while converting from RMTEs to UTF-8. Error Text: %s\n",
           rsslRetCodeToString(RetVal), RetVal, rsslRetCodeInfo(RetVal));
}
else
{
    /* SUCCESS: conversion was successful - application can now use converted content */
}

```

Code Example 21: Converting RMTEs to UTF-8 Example

11.2.9.6 Example: Converting RMTES to UCS-2

The following example illustrates storing and converting RMTES content. This example converts from RMTES to UCS-2 and assumes that:

- The input buffer is populated with RMTES content.
- The allocated size of 100 bytes is sufficient for conversion and storage.

To simplify the example, some error handling is omitted.

```

/* create cache buffer for storing RMTES and applying partial updates */
RsslRmtesCacheBuffer rmtesCache;
char cacheSpace[100];
/* create RsslU16Buffer to convert into */
RsslU16Buffer ucs2Buffer;
RsslUInt16 convertSpace[100];

/* populate cache and conversion buffers with created memory */
rmtesCache.data = cacheSpace;
rmtesCache.length = 0; /* this is the used length, since cache starts out empty it should
   start at 0 */
rmtesCache.allocatedLength = 100;

ucs2Buffer.data = convertSpace;
ucs2Buffer.length = 100;

/* apply RMTES content to cache, if successful convert to UCS-2 */
if ((RetVal = rsslRMTESToUCS2(&rmtesCache, &ucs2Buffer)) < RSSL_RET_SUCCESS)
{
    /* error while applying to cache */
    printf("Error %s (%d) encountered while applying buffer to RMTES cache. Error Text: %s\n",
           rsslRetCodeToString(RetVal), RetVal, rsslRetCodeInfo(RetVal));
}
else if ((RetVal = rsslRMTESApplyToCache(&inputBuf, &rmtesCache)) < RSSL_RET_SUCCESS)
{
    /* error when converting */
    printf("Error %s (%d) encountered while converting from RMTES to UCS-2. Error Text: %s\n",
           rsslRetCodeToString(RetVal), RetVal, rsslRetCodeInfo(RetVal));
}
else
{
    /* SUCCESS: conversion was successful - application can now use converted content */
}

```

Code Example 22: Converting RMTES to UCS-2 Example

11.2.10 General Primitive Type Utility Functions

The Transport API provides the following utility functions for use with primitive types.

FUNCTION NAME	DESCRIPTION
<code>rsslDataTypeToString</code>	Converts an <code>RsslDataType</code> enum to a string representation.
<code>rsslPrimitiveTypeSize</code>	Returns the maximum encoded size for base and set-defined primitive types. If the type allows for content of varying length (e.g. <code>RsslArray</code> , <code>RsslBuffer</code>), a value of 255 is returned (though the maximum encoded length may exceed 255).
<code>rsslIsPrimitiveType</code>	<ul style="list-style-type: none"> If <code>RsslDataType</code> enum represents a primitive type, returns true If <code>RsslDataType</code> enum represents a container type, returns false.
<code>rsslIsContainerType</code>	<ul style="list-style-type: none"> If <code>RsslDataType</code> enum represents a container type, returns true. If <code>RsslDataType</code> enum represents a primitive type, returns false.
<code>rsslEncodePrimitiveType</code>	This is an abstraction of individual encode functions for base primitive types. The user can pass in a base primitive-type enumeration and a pointer to that primitive-type representation (e.g. <code>RsslInt</code> , <code>RsslReal</code>). This value will then be encoded using the buffer contained referenced by <code>RsslEncodeIterator</code> .
<code>rsslDecodePrimitiveType</code>	This is an abstraction of individual decode functions for base primitive types. The user can pass in an enumeration for a base primitive type and an <code>RsslDecodeIterator</code> that contains that encoded primitive type. The encoded primitive type will be decoded and stored in a representation of the provided primitive type (e.g. <code>RsslInt</code> , <code>RsslReal</code>).
<code>rsslPrimitiveToString</code>	Converts a primitive type to a string representation. The user can pass in a base primitive-type enumeration and a pointer to that primitive-type representation (e.g. <code>RsslInt</code> , <code>RsslReal</code>). This value will be converted to a string and stored in the provided <code>RsslBuffer</code> . The user should ensure that the provided <code>RsslBuffer</code> has enough memory to properly store the converted value.
<code>rsslEncodedPrimitiveToString</code>	Converts an encoded primitive type to a string representation. The user can pass in a base primitive type enumeration in addition to an <code>RsslDecodeIterator</code> that contains the encoded primitive type. This function decodes the primitive type and converts it to a string, which is stored in the provided <code>RsslBuffer</code> . The user should ensure that the provided <code>RsslBuffer</code> has enough memory to properly store the converted value.

Table 100: General Primitive Type Utility Functions

11.3 Container Types

Container Types can model more complex data representations and have their contents modified at a more granular level than primitive types. Some container types leverage simple entry replacement when changes occur, while other container types offer entry-specific actions to handle changes to individual entries. The Transport API offers several uniform (i.e., homogeneous) container types, meaning that all entries house the same type of data. Additionally, there are several non-uniform (i.e., heterogeneous) container types in which different entries can hold varying types of data.

The **Rss1DataTypes** enumeration exposes values that define the type of a container. For example, when a **containerType** is housed in an **Rss1Msg**, the message would indicate the **containerType**'s enumerated value. Values ranging from 128 to 224 represent container types. Transport API messages and container types can house other Transport API container types. Only the **Rss1FieldList** and **Rss1ElementList** container types can house both primitive types and other container types.

The following table provides a brief description of each container type and its housed entries.

ENUM TYPE Name	Container TYPE	DESCRIPTION	Entry Type Information
RSSL_DT_FIELD_LIST	Rss1FieldList	A highly optimized, non-uniform type, that contains field identifier-value paired entries. fieldId refers to specific name and type information as defined in an external field dictionary (such as RDMFieldDictionary). You can further optimize this type by using set-defined data as described in Section 11.6. For more details on this container, refer to Section 11.3.1.	<p>Entry type is Rss1FieldEntry, which can house any Rss1DataType, including set-defined data (Section 11.6), base primitive types (Section 11.2), and container types.</p> <ul style="list-style-type: none"> If the information and entry being updated contains a primitive type, previously stored or displayed data is replaced. If the entry contains another container type, action values associated with that type specify how to update the information.
RSSL_DT_ELEMENT_LIST	Rss1ElementList	A self-describing, non-uniform type, with each entry containing name , dataType , and a value. This type is equivalent to Rss1FieldList , but without the optimizations provided through fieldId use. Use of set-defined data allows for further optimization, as discussed in Section 11.6. For more details on this container, refer to Section 11.3.2.	<p>Entry type is Rss1ElementEntry, which can house any Rss1DataType, including set-defined data (Section 11.6), base primitive types (Section 11.2), and container types.</p> <ul style="list-style-type: none"> If the updating information and entry contain a primitive type, any previously stored or displayed data is replaced. If the entry contains another container type, action values associated with that type specify how to update the information.

Table 101: Transport API Container Types

ENUM TYPE Name	Container TYPE	DESCRIPTION	Entry Type Information
RSSL_DT_MAP	RsslMap	A container of key-value paired entries. RsslMap is a uniform type, where the base primitive type of each entry's key and the containerType of each entry's payload are specified on the RsslMap . <ul style="list-style-type: none"> For more information on base primitive types, refer to Section 11.2. For more details on this container, refer to Section 11.3.3. 	Entry type is RsslMapEntry , which can include only container types, as specified on the RsslMap . Each entry's key is a base primitive type, as specified on the RsslMap . Each entry has an associated action, which informs the user of how to apply the information stored in the entry.
RSSL_DT_SERIES	RsslSeries	A uniform type, where the containerType of each entry is specified on the RsslSeries . This container is often used to represent table-based information, where no explicit indexing is present or required. As entries are received, the user should append them to any previously-received entries. For more details on this container, refer to Section 11.3.4.	Entry type is RsslSeriesEntry , which can include only container types, as specified on the RsslSeries . RsslSeriesEntry types do not contain explicit actions; though as entries are received, the user should append them to any previously received entries.
RSSL_DT_VECTOR	RsslVector	A container of position index-value paired entries. This container is a uniform type, where the containerType of each entry's payload is specified on the RsslVector . Each entry's index is represented by an unsigned integer. For more details on this container, refer to Section 11.3.5.	Entry type is RsslVectorEntry , which can house only container types, as specified on the RsslVector . Each entry's index is an unsigned integer. Each entry has an associated action, which informs the user on how to apply the information stored in the entry.
RSSL_DT_FILTER_LIST	RsslFilterList	A non-uniform container of filterId -value paired entries. A filterId corresponds to one of 32 possible bit-value identifiers, typically defined by a domain model specification. FilterId 's can be used to indicate interest or presence of specific entries through the inclusion of the filterId in the message key's filter member. <ul style="list-style-type: none"> For more information about the message key, refer to Section 12.1.2. For more details on this container, refer to Section 11.3.6. 	Entry type is RsslFilterEntry , which can house only container types. Though the RsslFilterList can specify a containerType , each entry can override this specification to house a different type. Each entry has an associated action, which informs the user of how to apply the information stored in the entry.

Table 101: Transport API Container Types (Continued)

ENUM TYPE Name	Container TYPE	DESCRIPTION	Entry Type Information
RSSL_DT_MSG	Rss1Msg	Indicates that the contents are another message. This allows the application to house a message within a message or a message within another container's entries. This type is typically used with posting (described in Section 13.9). For more details on message encoding and decoding, refer to Chapter 12, Message Package Detailed View.	None
RSSL_DT_NO_DATA	None	Indicates there are no contents. <ul style="list-style-type: none"> When RSSL_DT_NO_DATA is housed in a message, the message has no payload. If RSSL_DT_NO_DATA is housed in a container type, each container entry has no payload.^a 	None
RSSL_DT_ANSI_PAGE	None	Indicates that contents are ANSI Page format. Though the Transport API does not natively support encoding and decoding for the ANSI Page format, the Transport API supports the use of a separate ANSI Page encoder/decoder. For further details, refer to the <i>Transport API ANSI Library Manual</i> . For more details on housing non-RWF types inside of container types, refer to Section 11.3.7.	None
RSSL_DT_XML	None	Indicates that contents are XML-formatted data. Though the Transport API does not natively support encoding and decoding XML, the Transport API supports the use of a separate XML encoder/decoder. For more details on housing non-RWF types inside of container types, refer to Section 11.3.7.	None
RSSL_DT_OPAQUE	None	Indicates that the contents are opaque and additional details are not provided through the Transport API. Any specific information about the concrete type housed in the opaque payload should be defined in the specific domain model associated with the message. For more details on housing non-RWF types inside of container types, refer to Section 11.3.7.	None

Table 101: Transport API Container Types (Continued)

a. An **Rssi Fi I terLi st** can indicate a type of **RSSL_DT_NO_DATA**, however an individual **Rssi Fi I terEntry** can override using the entry-specific **con-tai nerType**.

11.3.1 RsslFieldList

The **RsslFieldList** is a container of entries (known as **RsslFieldEntry**s) paired by the values of their field identifiers. A **field identifier** (known as a **fieldId**), is a signed, two-byte value that refers to specific name and type information defined by an external field dictionary (e.g., **RDMFieldDictionary**). A field list can contain zero to N^6 entries, where zero indicates an empty field list.

11.3.1.1 Structure Members

RsslFieldList includes the following structures:

STRUCTURE MEMBER	DESCRIPTION
flags	A combination of bit values that indicate the presence of optional field list content. For more information about flag values, refer to Section 11.3.1.2.
dictionaryId	<p>A two-byte, signed integer that refer to the external dictionary family for use when interpreting content in this RsslFieldEntry. The field dictionary contains specific name and type information which correlates to fieldId values present in each RsslFieldEntry. An example of this would be the RDMFieldDictionary, which has a dictionaryId value of 1.</p> <p>If not present, a value of 1 should be assumed. If using the default dictionary (RDMFieldDictionary), dictionaryId is not required and is assumed have an id value of 1. A dictionaryId should be provided as part of the initial refresh message on a stream or on the first refresh message after issuing a CLEAR_CACHE command.</p> <p>A dictionaryId can be changed in two ways.</p> <ul style="list-style-type: none"> If a dictionaryId is provided on a refresh message (solicited or unsolicited), the specified dictionary is used across all messages on the stream until a new dictionaryId is provided in a subsequent refresh. This new dictionary is now used for all messages on the stream until another dictionaryId is provided. If an RsslFieldEntry contains a fieldId of 0, this reserved value indicates a temporary dictionary change. In this situation, this entry's value is the new dictionaryId (encoded / decoded as an RsslInt). When a dictionaryId is changed in this manner, the change is only in effect on the remaining entries in the field list or until another fieldId of 0 is encountered. Any containerTypes housed inside the remaining entries also adopt this temporary dictionary. When the end of the field list is reached, the dictionaryId from the refresh takes precedence once again. <p>dictionaryId values have an allowed range of -16,384 to 16,383.</p>
fieldListNum	<p>A two-byte, signed integer referring to an external fieldlist template, also known as a record template. The record template contains information about all possible fields in a stream and is typically used by caching implementations to pre-allocate storage.</p> <p>fieldListNum values have an allowed range of -32,768 to 32,767.</p>

Table 102: **Rssl FieldList** Structure Members

6. A field list currently has a maximum entry count of 65,535, where the first 255 entries may contain set-defined types. This type has an approximate maximum encoded length of 5 gigabytes but may be limited to 65,535 bytes if housed inside of a container entry. The content of each field entry has a maximum encoded length of 65,535 bytes. These limitations could be changed in subsequent releases.

STRUCTURE MEMBER	DESCRIPTION
setId	A two-byte, unsigned integer corresponding to the set definition used for encoding or decoding the set-defined data in this RsslFieldList . <ul style="list-style-type: none"> When encoding, this is the set definition used to encode any set-defined content. When decoding, this is the set definition used for decoding any set-defined content. If a setId value is not present on a message containing set-defined data, a setId of 0 is implied. setId values have an allowed range of 0 to 32,767 . Currently, only values 0 to 15 are used. These indicate locally-defined set definition use. Refer to Section 11.6 for more information.
encSetData	Length and pointer to the encoded set-defined data, if any, contained in the message. If populated, contents are described by the set definition associated with the setId member. If this is populated while encoding, this is assumed to be pre-encoded set data. If this is populated while decoding, this represents encoded set data. For more information, refer to Section 11.6.
encEntries	Length and pointer to the encoded fieldId -value pair encoded data, if any, contained in the message. This would refer to encoded RsslFieldList payload and length information.

Table 102: **RsslFieldList** Structure Members (Continued)

11.3.1.2 Flag Enumerations

FLAG ENUMERATION	MEANING
RSSL_FLF_HAS_FIELD_LIST_INFO	Indicates that dictionaryId and fieldListNum members are present, which should be provided as part of the initial refresh message on a stream or on the first refresh message after issuance of a CLEAR_CACHE command.
RSSL_FLF_HAS_STANDARD_DATA	Indicates that the RsslFieldList contains standard fieldId -value pair encoded data. This value can be set in addition to RSSL_FLF_HAS_SET_DATA if both standard and set-defined data are present in this RsslFieldList . If no entries are present in the RsslFieldList , this flag value should not be set.
RSSL_FLF_HAS_SET_DATA	Indicates that the RsslFieldList contains set-defined data. This value can be set in addition to RSSL_FLF_HAS_STANDARD_DATA if both standard and set-defined data are present in this RsslFieldList . If no entries are present in the RsslFieldList , this flag value should not be set. For more information, refer to Section 11.6.
RSSL_FLF_HAS_SET_ID	Indicates the presence of a setId , used to determine the set definition used for encoding or decoding the set data on this RsslFieldList . For more information, refer to Section 11.6.

Table 103: **RsslFieldList** Flags

11.3.1.3 RsslFieldEntry Structure Members

An **RsslFieldList** can contain multiple **RsslFieldEntries**, and each **RsslFieldEntry** can house any **RsslDataType**. This includes primitive types (as described in Section 11.2), set-defined types (as described in Section 11.6), or container types. If updating information, when the **RsslFieldEntry** contains a primitive type, it replaces any previously stored or displayed data associated with the same **fieldId**. If the **RsslFieldEntry** contains another container type, action values associated with that type indicate how to modify the information.

Structure Member	DESCRIPTION
fieldId	A signed two-byte value that refers to specific name and type information defined by an external field dictionary, such as the RDMFieldDictionary . Negative fieldId values typically refer to user-defined values while positive fieldId values typically refer to Thomson Reuters-defined values. fieldId has an allowable range of -32,768 to 32,767 where Thomson Reuters defines positive values and the user defines negative values. A fieldId value of 0 is reserved to indicate dictionaryId changes, where the type of fieldId 0 is an RsslInt .
dataType	Defines the RsslDataType of this RsslFieldEntry 's contents. <ul style="list-style-type: none"> While encoding, dataType must be set to the enumerated value of the type being encoded. While decoding, if dataType is RSSL_DT_UNKNOWN, the user must determine the type of contained information from the associated field dictionary. If set-defined data is used, dataType will indicate specific RsslDataType information as indicated by the set definition.
encData	Length and pointer to the encoded content of this RsslFieldEntry . <ul style="list-style-type: none"> If populated on encode functions, this indicates that data is pre-encoded and encData will be copied while encoding. If populated on decoding functions, this refers to the encoded RsslFieldEntry's payload and length information.

Table 104: **Rssl FieldEntry** Structure Members

11.3.1.4 Encoding Interfaces

An **RsslFieldEntry** can be encoded from pre-encoded data or from individual pieces of data as they are provided.

Encode Interface	Description
rsslEncodeFieldListInit	<p>Begins encoding an RsslFieldList. The Transport API will encode all content into the RsslBuffer to which the passed in RsslEncodeIterator refers. Entries can be encoded after this function returns.</p> <ul style="list-style-type: none"> If encoding set-defined data, the set definition database should be passed into this function. The Transport API will use the specified definition to validate and optimize content while encoding. To reserve space for encoding, a maximum length hint value (associated with the expected maximum encoded length of set-defined content in this RsslFieldList) can be passed into this function. If the approximate encoded set data length is not known, a value of 0 can be passed in. <p>For more details on local set definitions, refer to Section 11.6.</p>
rsslEncodeFieldListComplete	<p>Completes encoding of an RsslFieldList. This function expects the same RsslEncodeIterator that was used with rsslEncodeFieldListInit and all entries.</p> <ul style="list-style-type: none"> If encoding of all entries was successful, an RsslBool success parameter setting of true finishes the encoding. If encoding of any entry failed, an RsslBool success parameter setting of false rolls back encoding to the last successfully encoded point in the contents. <p>Field entries should be encoded prior to this call.</p>
rsslEncodeFieldEntry	<p>Encodes an RsslFieldEntry from a primitive type or pre-encoded data, or encodes an RsslFieldEntry as a blank primitive. This function expects the same RsslEncodeIterator that was used with rsslEncodeFieldListInit. You must properly populate RsslFieldEntry.fieldId and RsslFieldEntry.dataType.</p> <ul style="list-style-type: none"> If encoding from pre-encoded data, pass in rsslEncodeFieldEntry via the RsslFieldEntry.encData parameter and set the void* parameter to NULL. If encoding from a primitive type, pass a pointer to the populated primitive type via the void* parameter and leave RsslFieldEntry.encData empty. If encoding a blank value, set the void* parameter to NULL and leave RsslFieldEntry.encData empty.
rsslEncodeFieldEntryInit	<p>Encodes an RsslFieldEntry from a complex type, such as a container type or an array. This function expects the same RsslEncodeIterator that was used with rsslEncodeFieldListInit. After this call, housed-type encode functions can be used to encode the contained type. You must properly populate RsslFieldEntry.fieldId and RsslFieldEntry.dataType.</p> <ul style="list-style-type: none"> A maximum-length hint value, associated with the expected maximum-encoded length of this field, can be passed into this function to reserve the space needed for encoding. If the approximate encoded length is not known, a value of 0 can be passed in which allows for up to the maximum content length.

Table 105: **Rssl FieldList Encode Functions**

ENCODE INTERFACE	DESCRIPTION
rsslEncodeFieldEntryComplete	<p>Completes the encoding of an RsslFieldEntry. This function expects the same RsslEncodeIterator that was used with rsslEncodeFieldListInit, rsslEncodeFieldEntryInit, and all other entry encoding.</p> <ul style="list-style-type: none"> • If encoding the entry succeeds, setting RsslBool success to true finishes entry encoding. • If encoding the entry fails, setting RsslBool success parameter to false rolls back the encoding of this particular RsslFieldEntry.

Table 105: **Rssl FieldList** Encode Functions (Continued)

11.3.1.5 Rippling

The **RsslFieldList** container supports rippling fields. When *rippling*, newly received content associated with a **fieldId** replaces previously received content associated with the same **fieldId**. The previously-received content is moved to a new **fieldId** (typically indicated in a field dictionary⁷). Rippling is typically used as a way to reduce bandwidth consumption. Normally, if previously-received data were still relevant, it would need to be sent with subsequent updates even though the value was not changing. Rippling allows this data to be removed from subsequent updates; however the consumer must use the ripple information from a field dictionary to correctly propagate previously received content. Rippling is the responsibility of the consumer application, and the Transport API does not perform entry rippling.

11.3.1.6 Encoding Example

The following example illustrates how to encode an **RsslFieldList**. The example encodes four **RsslFieldEntry** values:

- The first encodes an entry from a primitive **RsslDate** type
- The second from a pre-encoded buffer containing an encoded **RsslUInt**
- The third as a blank **RsslReal** value
- The fourth as an **RsslArray** complex type. The pattern followed while encoding the fourth entry can be used for encoding of any container type into an **RsslFieldEntry**.

This example demonstrates error handling for the initial encode function. To simplify the example, additional error handling is omitted (though it should be performed). This example shows encoding of standard **fieldId**-value data.

```

/* populate field list structure prior to call to rsslEncodeFieldListInit */
/* NOTE: the fieldId, dictionaryId and fieldListNum values used for this example do not
   correspond to
   actual id values

/* indicate that standard data will be encoded and that dictionaryId and fieldListNum are
   included */
fieldList.flags = RSSL_FLF_HAS_STANDARD_DATA | RSSL_FLF_HAS_FIELD_LIST_INFO;
/* populate dictionaryId and fieldListNum with info needed to cross-reference fieldIds and
   cache */
fieldList.dictionaryId = 2;
fieldList.fieldListNum = 5;

```

7. In the RDM Field Dictionary, the **RIPPLES TO** column defines the **fieldId** information to use when rippling.

```

/* begin encoding of field list - assumes that encIter is already populated with buffer and
version
/* information, store return value to determine success or failure */
if (( retVal = rsslEncodeFieldListInit(&encIter, &fieldList, 0, 0)) < RSSL_RET_SUCCESS)
{
    /* error condition - switch our success value to false so we can roll back */
    success = RSSL_FALSE;
    /* print out message with return value string, value, and text */
    printf("Error %s (%d) encountered with rsslEncodeFieldListInit. Error Text: %s\n",
        rsslRetCodeToString(retVal), retVal, rsslRetCodeInfo(retVal));
}
else
{
    /* fieldListInit encoding was successful */
    /* create a single RsslFieldEntry and reuse for each entry */
    RsslFieldEntry fieldEntry = RSSL_INIT_FIELD_ENTRY;
    /* stack allocate a date and populate {day, month, year} */
    RsslDate rsslDate = {30, 11, 2010};
    RsslArray rsslArray = RSSL_INIT_ARRAY;

    /* FIRST Field Entry: encode entry from the RsslDate primitive type */
    /* populate and encode field entry with fieldId and dataType information for this field */
    fieldEntry.fieldId = 16;
    fieldEntry.dataType = RSSL_DT_DATE;
    retVal = rsslEncodeFieldEntry(&encIter, &fieldEntry, &rsslDate);

    /* SECOND Field Entry: encode entry from preencoded buffer containing an encoded RsslUInt
type */
    /* populate and encode field entry with fieldId and dataType information for this field */
    /* because we are re-populating all values on RsslFieldEntry, there is no need to clear it
*/
    fieldEntry.fieldId = 1080;
    fieldEntry.dataType = RSSL_DT_UINT;
    /* assuming pEncUInt is an RsslBuffer with length and data properly populated */
    fieldEntry.encData.length = pEncUInt->length;
    fieldEntry.encData.data = pEncUInt->data;
    /* void* parameter is passed in as NULL because pre-encoded data is set on RsslFieldEntry
itself */
    retVal = rsslEncodeFieldEntry(&encIter, &fieldEntry, NULL);

    /* THIRD Field Entry: encode entry as a blank RsslReal primitive type */
    /* populate and encode field entry with fieldId and dataType information for this field */
    /* need to ensure that RsslFieldEntry is appropriately cleared
    /* - clearing will ensure that encData is properly emptied */
    rsslClearFieldEntry(&fieldEntry);

    fieldEntry.fieldId = 22;
}

```

```

fieldEntry.dataType = RSSL_DT_REAL;
/* void* parameter is passed in as NULL and encData is empty due to clearing */
retVal = rsslEncodeFieldEntry(&encIter, &fieldEntry, NULL);

/* FOURTH Field Entry: encode entry as a complex type, RsslArray primitive */
/* populate and encode field entry with fieldId and dataType information for this field */
/* need to ensure that RsslFieldEntry is appropriately cleared
/* - clearing will ensure that encData is properly emptied */
rsslClearFieldEntry(&fieldEntry);
fieldEntry.fieldId = 1021;
fieldEntry.dataType = RSSL_DT_ARRAY;
/* begin complex field entry encoding, we are not sure of the approximate max encoding
length */
retVal = rsslEncodeFieldEntryInit(&encIter, &fieldEntry, 0);
{
    /* now encode nested container using its own specific encode functions */
    /* encode RsslReal values into the array */
    rsslArray.primitiveType = RSSL_DT_REAL;
    /* values are variable length */
    rsslArray.itemLength = 0;
    /* begin encoding of array - using same encIterator as field list */
    if ((retVal = rsslEncodeArrayInit(&encIter, &rsslArray)) < RSSL_RET_SUCCESS)

        /*----- Continue encoding array entries. See example in Section 11.2.7 ----- */

        /* Complete nested container encoding */
        retVal = rsslEncodeArrayComplete(&encIter, success);
    }
    /* complete encoding of complex field entry. If any array encoding failed, success is false
*/
    retVal = rsslEncodeFieldEntryComplete(&encIter, success);
}
/* complete fieldList encoding. If success parameter is true, this will finalize encoding.
If success parameter is false, this will roll back encoding prior to
rsslEncodeFieldListInit */
retVal = rsslEncodeFieldListComplete(&encIter, success);

```

Code Example 23: Rssl FieldList Encoding Example

11.3.1.7 Decoding Interfaces

A decoded **RsslFieldEntry** structure provides access to the encoded content of the field entry. Further decoding of the entry's contents can be skipped by invoking the entry decoder to move to the next **RsslFieldEntry** or the contents can be further decoded by invoking the decode function of the contained type.

DECODE INTERFACE	DESCRIPTION
rsslDecodeFieldList	<p>Begins decoding of an <code>RsslFieldList</code> from the <code>RsslBuffer</code> referenced in the <code>RsslDecodeIterator</code>. This function allows the user to pass local set definitions.</p> <p>If the <code>RsslFieldList</code> structure contains set-defined data (e.g. if the <code>RSSL_FLF_HAS_SET_DATA</code> flag is present), the Transport API decodes the set-defined entries when definitions are present. Otherwise, set-defined entries are skipped while decoding entries.</p>
rsslDecodeFieldEntry	<p>Decodes an <code>RsslFieldEntry</code>, expecting the same <code>RsslDecodeIterator</code> that was used with <code>rsslDecodeFieldList</code>. This populates <code>encData</code> with the entry's encoded contents.</p> <ul style="list-style-type: none"> • If decoding set-defined entries, the <code>RsslFieldEntry.dataType</code> populates with the type from the set definition. • If decoding standard <code>fieldId</code>-value data, <code>RsslFieldEntry.dataType</code> is set to <code>RSSL_DT_UNKNOWN</code>, indicating that the user must determine the type from a field dictionary. <p>After determining the type, the specific decode function can be called if needed. Calling <code>rsslDecodeFieldEntry</code> again will begin decoding the next entry in the <code>RsslFieldList</code> until no more entries are available.</p>

Table 106: Rssl FieldList Decode Functions

11.3.1.8 Decoding Example

The following example demonstrates how to decode an **RsslFieldList** and is structured to decode each entry to the contained value. This example uses a switch statement to invoke the specific decoder for the contained type, however to simplify the example, necessary cases and some error handling are omitted. This example uses the same **RsslDecodeIterator** when calling the primitive decoder function. An application could optionally use a new **RsslDecodeIterator** by setting the **encData** on a new iterator.

```

/* decode into the field list structure */
if ((RetVal = rsslDecodeFieldList(&decIter, &fieldList, &localSetDefs)) >= RSSL_RET_SUCCESS)
{
    /* decode each field entry */
    while ((RetVal = rsslDecodeFieldEntry(&decIter, &fieldEntry)) != RSSL_RET_END_OF_CONTAINER)
    {
        if (RetVal < RSSL_RET_SUCCESS)
        {
            /* decoding failure tends to be unrecoverable */
            printf("Error %s (%d) encountered with rsslDecodeFieldEntry. Error Text: %s\n",
                   rsslRetCodeToString(RetVal), RetVal, rsslRetCodeInfo(RetVal));
        }
        else
        {
            /* look up type in field dictionary and call correct primitive decode function */
            switch (fieldDict->entriesArray[fieldEntry->fieldId]->rwfType)
            {
                case RSSL_DT_REAL:
                    RetVal = rsslDecodeReal(&decIter, &rsslReal);
                    break;
                case RSSL_DT_DATE:
                    RetVal = rsslDecodeDate(&decIter, &rsslDate);
                    break;
                /* full switch statement omitted to shorten sample code */

            }
        }
    }
}
else
{
    /* decoding failure tends to be unrecoverable */
    printf("Error %s (%d) encountered with rsslDecodeFieldList. Error Text: %s\n",
           rsslRetCodeToString(RetVal), RetVal, rsslRetCodeInfo(RetVal));
}

```

Code Example 24: RsslFieldList Decoding Example

11.3.1.9 Type Utility Functions

The Transport API provides the following utility functions for use with `Rss1FieldList`.

FUNCTION NAME	DESCRIPTION
<code>rss1ClearFieldList</code>	Clears members from an <code>Rss1FieldList</code> structure. Useful for structure reuse.
<code>rss1ClearFieldEntry</code>	Clears members from an <code>Rss1FieldEntry</code> structure. Useful for structure reuse.

Table 107: `Rss1 FieldList` Utility Functions

11.3.2 Rss1ElementList

`Rss1ElementList` is a self-describing container type. Each entry, known as an `Rss1ElementEntry`, contains an element `name`, `dataType` enumeration, and value. An element list is equivalent to `Rss1FieldList`, where name and type information is present in each element entry instead of optimized via a field dictionary. An element list can contain zero to N^8 entries, where zero indicates an empty element list.

11.3.2.1 Structure Members

Structure Member	DESCRIPTION
<code>flags</code>	Combination of bit values that indicate whether optional, element-list content is present. For more information about flag values, refer to Section 11.3.2.2.
<code>elementListNum</code>	A two-byte signed integer that refers to an external element-list template, also known as a record template . A record template contains information about all possible entries contained in the stream and is typically used by caching mechanisms to pre-allocate storage. <code>elementListNum</code> values have a range of -32,768 to 32,767 .
<code>setId</code>	A two-byte unsigned integer that corresponds to the set definition used for encoding or decoding the set-defined data in this <code>Rss1ElementList</code> . <ul style="list-style-type: none"> When encoding, this is the set definition used to encode any set-defined content. When decoding, this is the set definition used for decoding any set-defined content. <code>setId</code> values have an allowed range of 0 to 32,767 . Currently, only values 0 to 15 are used. These indicate locally-defined set definition use. If a <code>setId</code> value is not present on a message containing set-defined data, a <code>setId</code> of 0 is implied. For more information, refer to Section 11.6.
<code>encSetData</code>	Length and pointer to the encoded set-defined data, if any, contained in the message. If populated, contents are described by the set definition associated with the <code>setId</code> member. <ul style="list-style-type: none"> If this is populated while encoding, this is assumed to be pre-encoded set data. If this is populated while decoding, this represents encoded set data. For more information, refer to Section 11.6.

Table 108: `Rss1 ElementList` Structure Members

8. An element list currently has a maximum entry count of 65,535, where the first 255 entries may contain set-defined types. This type has an approximate maximum encoded length of 5 gigabytes but may be limited to 65,535 bytes if housed inside of a container entry. The content of element entry has a maximum encoded length of 65,535 bytes. These limitations can change in subsequent releases.

Structure Member	DESCRIPTION
encEntries	Length and pointer to all encoded element <code>name</code> , <code>dataType</code> , value encoded data, if any, contained in the message. This would refer to encoded <code>Rss1ElementList</code> payload and length information.

Table 108: `Rss1 ElementList` Structure Members (Continued)

11.3.2.2 Flag Enumerations

FLAG ENUMERATION	MEANING
<code>RSSL_ELF_HAS_ELEMENT_LIST_INFO</code>	Indicates the presence of the <code>elementListNum</code> member. This member is provided as part of the initial refresh message on a stream or on the first refresh message after a <code>CLEAR_CACHE</code> command.
<code>RSSL_ELF_HAS_STANDARD_DATA</code>	Indicates that the <code>Rss1ElementList</code> contains standard element <code>name</code> , <code>dataType</code> , value-encoded data. You can set this value in addition to <code>RSSL_ELF_HAS_SET_DATA</code> if both standard and set-defined data are present in this <code>Rss1ElementList</code> . If the <code>Rss1ElementList</code> does not have entries, do not set this flag value.
<code>RSSL_ELF_HAS_SET_DATA</code>	Indicates that <code>Rss1ElementList</code> contains set-defined data. <ul style="list-style-type: none"> If both standard and set-defined data are present in this <code>Rss1ElementList</code>, this value can be set in addition to <code>RSSL_ELF_HAS_STANDARD_DATA</code>. If the <code>Rss1ElementList</code> does not have entries, do not set this flag value. For more information, refer to Section 11.6.
<code>RSSL_ELF_HAS_SET_ID</code>	Indicates the presence of a <code>setId</code> and determines the set definition to use when encoding or decoding set data on this <code>Rss1ElementList</code> . For more information, refer to Section 11.6.

Table 109: `Rss1 ElementList` Flags

11.3.2.3 `Rss1ElementEntry` Structure Members

Each `Rss1ElementList` can contain multiple `Rss1ElementEntry`s and each `Rss1ElementEntry` can house any `Rss1DataType`, including primitive types (refer to Section 11.2), set-defined types (refer to Section 11.6), or container types. If an `Rss1ElementEntry` is a part of updating information and contains a primitive type, any previously stored or displayed data is replaced. If an `Rss1ElementEntry` contains another container type, action values associated with that type indicate how to modify data.

Structure Member	DESCRIPTION
<code>name</code>	An <code>Rss1Buffer</code> containing the <code>name</code> associated with this <code>Rss1ElementEntry</code> . Element names are defined outside of the Transport API, typically as part of a domain model specification or dictionary. A <code>name</code> can be empty; however this provides no identifying information for the element. The <code>name</code> buffer allows for content length ranging from 0 bytes to 32,767 bytes.

Table 110: `Rss1 ElementEntry` Structure Members

Structure Member	DESCRIPTION
dataType	<p>Defines the <code>Rss1DataType</code> of this <code>Rss1ElementEntry</code>'s contents.</p> <ul style="list-style-type: none"> While encoding, set this to the enumerated value of the target type. While decoding, <code>dataType</code> describes the type of contained data so that the correct decoder can be used. <p>If set-defined data is used, <code>dataType</code> will indicate any specific <code>Rss1DataType</code> information as defined in the set definition.</p>
encData	Length and pointer to the encoded content of this <code>Rss1ElementEntry</code> . If populated on encode functions, this indicates that data is pre-encoded and <code>encData</code> copies while encoding. While decoding, this refers to the encoded <code>Rss1ElementEntry</code> 's payload and length data.

Table 110: `Rss1ElementEntry` Structure Members (Continued)

11.3.2.4 Encoding Interfaces

`Rss1ElementEntry` can be encoded from pre-encoded data or by encoding individual data as they are provided.

ENCODE INTERFACE	DESCRIPTION
<code>rsslEncodeElementListInit</code>	<p>Begins encoding of an <code>Rss1ElementList</code>. The Transport API encodes data into the <code>Rss1Buffer</code> referred to by the <code>Rss1EncodeIterator</code>. Entries can be encoded after this function returns.</p> <ul style="list-style-type: none"> If encoding set-defined data, pass the set definition database into this function. The Transport API uses the specified definition to validate and optimize content while encoding. To reserve space for encoding, a maximum-length hint value (associated with the expected maximum-encoded length of set-defined content in this <code>Rss1ElementList</code>) can be passed into this function. If the approximate length of encoded set data is not known, you can pass in a value of 0. <p>For more details on local set definitions, refer to Section 11.6.</p>
<code>rsslEncodeElementListComplete</code>	<p>Completes encoding of an <code>Rss1ElementList</code>. This function expects the same <code>Rss1EncodeIterator</code> that was used with <code>rsslEncodeElementListInit</code> and all entries.</p> <ul style="list-style-type: none"> If all entries were encoded successfully, an <code>Rss1Bool</code> success parameter setting of true finishes encoding. If encoding of any entry failed, an <code>Rss1Bool</code> success parameter setting of false rolls back encoding to the last successfully encoded point in the contents. <p>Any element entries should be encoded prior to this call.</p>

Table 111: `Rss1ElementList` Encoding Interfaces

Encode Interface	Description
rsslEncodeElementEntry	<p>Encodes <code>RsslElementEntry</code> from primitive type representation or pre-encoded data or encodes an <code>RsslElementEntry</code> as a blank primitive. This function expects the same <code>RsslEncodeIterator</code> that was used with <code>rsslEncodeElementListInit</code>.</p> <ul style="list-style-type: none"> If encoding from pre-encoded data, pass in <code>rsslEncodeElementEntry</code> via the <code>RsslElementEntry.encData</code> parameter and set the <code>void*</code> parameter to <code>NULL</code>. If encoding from a primitive type, pass a pointer to the populated primitive type via the <code>void*</code> parameter and leave <code>RsslElementEntry.encData</code> empty. If encoding a blank value, set the <code>void*</code> parameter to <code>NULL</code> and leave <code>RsslElementEntry.encData</code> empty. <p>Call this function for each entry being encoded. <code>RsslElementEntry.name</code> and <code>RsslElementEntry.dataType</code> must be properly populated.</p>
rsslEncodeElementEntryInit	<p>Encodes an <code>RsslElementEntry</code> from a complex type, such as a container type or an array. This function expects the same <code>RsslEncodeIterator</code> that was used with <code>rsslEncodeElementListInit</code>. After this call, the encode functions from the housed type can be used to encode the contained type. You must properly populate <code>RsslElementEntry.name</code> and <code>RsslElementEntry.dataType</code>.</p> <p>To reserve an appropriate amount of space while encoding, you can pass a max-length hint value (associated with the expected maximum-encoded length of this element) into this function. If the approximate encoded length is not known, you can pass in a value of <code>0</code>.</p>
rsslEncodeElementEntryComplete	<p>Completes the encoding of an <code>RsslElementEntry</code>. This function expects the same <code>RsslEncodeIterator</code> that was used with <code>rsslEncodeElementListInit</code>, <code>rsslEncodeElementEntryInit</code>, and all other entry encoding.</p> <ul style="list-style-type: none"> If this specific entry was encoded successfully, an <code>RsslBool success</code> parameter setting of <code>true</code> finishes entry encoding. If this specific entry was not encoded successfully, an <code>RsslBool success</code> parameter setting of <code>false</code> rolls back the encoding of only this <code>RsslElementEntry</code>.

Table 111: Rssl ElementList Encoding Interfaces (Continued)

11.3.2.5 RsslElementEntry Encoding Example

The following example demonstrates how to encode an `RsslElementList` and encodes four `RsslElementEntry` values:

- The first encodes an entry from a primitive `RsslTime` type
- The second encodes from a pre-encoded buffer containing an encoded `RsslUInt`
- The third encodes as a blank `RsslReal` value
- The fourth encodes as an `RsslFieldList` container type

The pattern used to encode the fourth entry can be used to encode any container type into an `RsslElementEntry`. This example demonstrates error handling for the initial encode function. However, additional error handling is omitted to simplify the example. This example shows the encoding of standard `name`, `dataType`, and `value` data.

```

/* populate element list structure prior to call to rsslEncodeElementListInit */
/* NOTE: the element names and elementListNum values used for this example may not correspond
   to actual
/* name values */

/* indicate that standard data will be encoded and that elementListNum is included */
elemList.flags = RSSL_ELF_HAS_STANDARD_DATA | RSSL_ELF_HAS_ELEMENT_LIST_INFO;
/* populate elementListNum with info needed to cache */
elemList.elementListNum = 5;

/* begin encoding of element list - assumes that encIter is already populated with buffer and
   version
/* information, store return value to determine success or failure */
if (( retVal = rsslEncodeElementListInit(&encIter, &elemList, 0, 0) < RSSL_RET_SUCCESS)
{
    /* error condition - switch our success value to false so we can roll back */
    success = RSSL_FALSE;
    /* print out message with return value string, value, and text */
    printf("Error %s (%d) encountered with rsslEncodeElementListInit. Error Text: %s\n",
        rsslRetCodeToString(retVal), retVal, rsslRetCodeInfo(retVal));
}
else
{
    /* elementListInit encoding was successful */
    /* create a single RsslElementEntry and reuse for each entry */
    RsslElementEntry elemEntry = RSSL_INIT_ELEMENT_ENTRY;
    /* stack allocate a time and populate {hour, minute, second, millisecond} */
    RsslTime rsslTime = {10, 21, 16, 777};
    RsslFieldList fieldList = RSSL_INIT_FIELD_LIST;

    /* FIRST Element Entry: encode entry from the RsslTime primitive type */
    /* populate and encode element entry with name and dataType information for this element */
    elemEntry.name.data = "Element1 - Primitive";
    elemEntry.name.length = 20;
    elemEntry.dataType = RSSL_DT_TIME;
}

```

```

 retVal = rsslEncodeElementEntry(&encIter, &elemEntry, &rsslTime);

/* SECOND Element Entry: encode entry from preencoded buffer containing an encoded RsslUInt
type */
/* populate and encode element entry with name and dataType information for this element */
/* because we are re-populating all values on RsslElementEntry, there is no need to clear
it */

elemEntry.name.data = "Element2 - Pre-Encoded";
elemEntry.name.length = 22;
elemEntry.dataType = RSSL_DT_UINT;
/* assuming pEncUInt is an RsslBuffer with length and data properly populated */
elemEntry.encData.length = pEncUInt->length;
elemEntry.encData.data = pEncUInt->data;
/* void* parameter is passed in as NULL because pre-encoded data is set on RsslElementEntry
itself */
retVal = rsslEncodeElementEntry(&encIter, &elemEntry, NULL);

/* THIRD Element Entry: encode entry as a blank RsslReal primitive type */
/* populate and encode element entry with name and dataType information for this element
need to */
/* ensure that RsslElementEntry is appropriately cleared - clearing will ensure that
encData is */
/* properly emptied */
rsslClearElementEntry(&elemEntry);

elemEntry.name.data = "Element3 - Blank";
elemEntry.name.length = 16;
elemEntry.dataType = RSSL_DT_REAL;
/* void* parameter is passed in as NULL and encData is empty due to clearing */
retVal = rsslEncodeElementEntry(&encIter, &elemEntry, NULL);

/* FOURTH Element Entry: encode entry as a container type, RsslFieldList */
/* populate and encode element entry with name and dataType information for this element
need to */
/* ensure that RsslElementEntry is appropriately cleared - clearing will ensure that
encData is */
/* properly emptied */
rsslClearElementEntry(&elemEntry);
elemEntry.name.data = "Element4 - Container";
elemEntry.name.length = 20;
fieldEntry.dataType = RSSL_DT_FIELD_LIST;
/* begin complex element entry encoding, we are not sure of the approximate max encoding
length */
retVal = rsslEncodeElementEntryInit(&encIter, &elemEntry, 0);
{
    /* now encode nested container using its own specific encode functions */
    /* begin encoding of field list - using same encIterator as element list */
    fieldList.flags = RSSL_FLF_HAS_STANDARD_DATA;
}

```

```

if ((retval = rsslEncodeFieldListInit(&encIter, &fieldList, 0, 0)) < RSSL_RET_SUCCESS)

/*----- Continue encoding field entries. See example in Section 11.3.1.6 ----- */

/* Complete nested container encoding */
retval = rsslEncodeFieldListComplete(&encIter, success);
}

/* complete encoding of complex element entry. If any field list encoding failed, success
is false */
retval = rsslEncodeElementEntryComplete(&encIter, success);
}

/* complete elementList encoding. If success parameter is true, this will finalize encoding.
/* If success parameter is false, this will roll back encoding prior to
rsslEncodeElementListInit */
retval = rsslEncodeElementListComplete(&encIter, success);

```

Code Example 25: Rssl ElementList Encoding Example

11.3.2.6 RsslElementList Decoding Interfaces

A decoded **RsslElementEntry** structure provides access to the encoded content of the element entry. The entry's contents can be further decoded by invoking the specific contained type's decode function or can be skipped by invoking the entry decoder to move to the next **RsslElementEntry**.

DECODE INTERFACE	DESCRIPTION
rsslDecodeElementList	Begins decoding an RsslElementList . This function will decode from the RsslBuffer referred to by the passed-in RsslDecodeIterator . This function allows for the user to pass local set definitions. If the RsslElementList structure contains set-defined data (e.g., RSSL_ELF_HAS_SET_DATA is present), the Transport API will decode set-defined entries when their definitions are present. Otherwise, the Transport API skips set-defined entries when decoding entries.
rsslDecodeElementEntry	Decodes an RsslElementEntry . This function expects the same RsslDecodeIterator used with rsslDecodeElementList and populates encData with encoded entry contents. After this function returns, you can use the RsslElementEntry.dataType to invoke the correct contained type's decode functions. Calling rsslDecodeElementEntry again will begin decoding the next entry in the RsslElementList until no more entries are available.

Table 112: Rssl ElementList Decode Functions

11.3.2.7 RsslElementList Decoding Examples

The following sample demonstrates how to decode an `RsslElementList` and is structured to decode each entry to its contained value. This example uses a switch statement to invoke the specific decoder for the contained type, however for sample clarity, unnecessary cases have been omitted. This example uses the same `RsslDecodeIterator` when calling the primitive decoder function. An application could optionally use a new `RsslDecodeIterator` by setting the `encData` on a new iterator. For simplification, the example omits some error handling.

```

/* decode into the element list structure */
if ((RetVal = rsslDecodeElementList(&decIter, &elemList, &localSetDefs)) >=
RSSL_RET_SUCCESS)
{
/* decode each element entry */
while ((RetVal = rsslDecodeElementEntry(&decIter, &elemEntry)) !=
RSSL_RET_END_OF_CONTAINER)
{
    if (RetVal < RSSL_RET_SUCCESS)
    {
        /* decoding failure tends to be unrecoverable */
        printf("Error %s (%d) encountered with rsslDecodeElementEntry. Error Text: %s\n",
               rsslRetCodeToString(RetVal), RetVal, rsslRetCodeInfo(RetVal));
    }
    else
    {
        /* use elemEntry.dataType to call correct primitive decode function */
        switch (elemEntry.dataType)
        {
            case RSSL_DT_REAL:
                RetVal = rsslDecodeReal(&decIter, &rsslReal);
                break;
            case RSSL_DT_TIME:
                RetVal = rsslDecodeTime(&decIter, &rsslTime);
                break;
            /* full switch statement omitted to shorten sample code */
        }
    }
}
else
{
    /* decoding failure tends to be unrecoverable */
    printf("Error %s (%d) encountered with rsslDecodeElementList. Error Text: %s\n",
           rsslRetCodeToString(RetVal), RetVal, rsslRetCodeInfo(RetVal));
}

```

Code Example 26: Rssl ElementList Decoding Example

11.3.2.8 Rss1ElementList Utility Functions

The Transport API provides the following utility functions for use with the **Rss1ElementList** type:

FUNCTION NAME	DESCRIPTION
rss1ClearElementList	Clears members from an Rss1ElementList structure. Useful for structure reuse.
rss1ClearElementEntry	Clears members from an Rss1ElementEntry structure. Useful for structure reuse.

Table 113: **Rss1 ElementList** Utility Functions

11.3.3 Rss1Map

The **Rss1Map** is a uniform container type of associated key-value pair entries. Each entry, known as an **Rss1MapEntry**, contains an entry key, which is a base primitive type (Section 11.2) and value. An **Rss1Map** can contain zero to N^9 entries, where zero entries indicate an empty **Rss1Map**.

11.3.3.1 Rss1Map Structure Members

An **Rss1Map** structure contains the following Structure Members:

STRUCTURE MEMBER	DESCRIPTION
flags	Combination of bit values to indicate the presence of optional Rss1Map content. For more information about flag values, refer to Section 11.3.3.2.
keyPrimitiveType	The Rss1DataType enumeration value that describes the base primitive type of each Rss1MapEntry 's key. keyPrimitiveType accepts values between 1 and 63, cannot be specified as blank, and cannot be the RSSL_DT_ARRAY or RSSL_DT_UNKNOWN primitive types. For more information about base primitive types, refer to Section 11.2.
keyFieldId	(Optional) Specifies a fieldId associated with the entry key information. This is mainly used as an optimization to avoid inclusion of redundant data. In situations where key information is also a member of the entry payload (e.g., Order Id for Market By Order domain type), this allows removal of data from each entry's payload prior to encoding as it is already present via the key and keyFieldId . keyFieldId has an allowable range of -32,768 to 32,767 where positive values are Thomson Reuters-defined and negative values are user-defined.
containerType	The Rss1DataType enumeration value that describes the container type of each Rss1MapEntry 's payload.

Table 114: **Rss1 Map** Structure Members

9. An **Rss1 Map** currently has a maximum entry count of 65,535. This type has an approximate maximum encoded length of 5 gigabytes but may be limited to 65,535 bytes if housed inside of a container entry. The content of an **Rss1 MapEntry** has a maximum encoded length of 65,535 bytes. These limitations could be changed in subsequent releases.

Structure Member	DESCRIPTION
totalCountHint	A four-byte unsigned integer that indicates an approximate total number of entries associated with this stream. This is typically used when multiple <code>Rss1Map</code> containers are spread across multiple parts of a refresh message (for more information about message fragmentation and multi-part message handling, refer to Section 13.1). <code>totalCountHint</code> provides an approximation of the total number of entries sent across all maps on all parts of the refresh message. This information is useful when determining the amount of resources to allocate for caching or displaying all expected entries. <code>totalCountHint</code> values have a range of 0 to 1,073,741,824.
encSummaryData	Length and pointer to the encoded summary data, if any, contained in the message. If populated, summary data contains information that applies to every entry encoded in the <code>Rss1Map</code> (e.g., currency type). The container type of summary data should match the <code>containerType</code> specified on the <code>Rss1Map</code> . If <code>encSummaryData</code> is populated while encoding, contents are used as pre-encoded summary data. Encoded summary data has maximum allowed length of 32,767 bytes. For more information, refer to Section 11.5.
encSetDefs	Length and pointer to the encoded local set definitions, if any, contained in the message. If populated, these definitions correspond to data contained within the <code>Rss1Map</code> 's entries and are used for encoding or decoding their contents. Encoded local set definitions have a maximum allowed length of 32,767 bytes. For more information, refer to Section 11.6.
encEntries	Length and pointer to the all encoded key-value pair data, if any, contained in the message. This would refer to encoded <code>Rss1Map</code> payload and length information.

Table 114: `Rss1 Map` Structure Members (Continued)

11.3.3.2 RsslMap Flag Enumeration Values

FLAG ENUMERATION	MEANING
RSSL_MPF_HAS_KEY_FIELD_ID	Indicates the presence of the <code>keyFieldId</code> member. <code>keyFieldId</code> should be provided if the key information is also a field that would be contained in the entry payload. This optimization allows <code>keyFieldId</code> to be included once instead of in every entry's payload.
RSSL_MPF_HAS_TOTAL_COUNT_HINT	Indicates the presence of the <code>totalCountHint</code> member. This member can provide an approximation of the total number of entries sent across all maps on all parts of the refresh message. This information is useful when determining the amount of resources to allocate for caching or displaying all expected entries.
RSSL_MPF_HAS_PER_ENTRY_PERM_DATA	Indicates that permission information is included with some map entries. The <code>RsslMap</code> encoding functionality sets this flag value on the user's behalf if any entry is encoded with its own <code>permData</code> . A decoding application can check this flag to determine if any contained entry has <code>permData</code> , often useful for fan out devices (if an entry does not have <code>permData</code> , the fan out device can likely pass on data and not worry about special permissioning for the entry). Each entry will also indicate the presence of permission data via the use of <code>RSSL_MPEF_HAS_PERM_DATA</code> .
RSSL_MPF_HAS_SUMMARY_DATA	Indicates that the <code>RsslMap</code> contains summary data. If this flag is set while encoding, summary data must be provided by encoding or populating <code>encSummaryData</code> with pre-encoded information. If this flag is set while decoding, summary data is contained as part of the <code>RsslMap</code> and the user can choose whether to decode it.
RSSL_MPF_HAS_SET_DEFS	Indicates that the <code>RsslMap</code> contains local set definition information. Local set definitions correspond to data contained within this <code>RsslMap</code> 's entries and are used for encoding or decoding their contents. For more information, refer to Section 11.6.

Table 115: Rssl Map Flags

11.3.3.3 RsslMapEntry Structure Members

`RsslMapEntry`s can house only other container types. `RsslMap` is a uniform type, where the `RsslMap.containerType` indicates the single type housed in each entry. Each entry has an associated action which informs the user of how to apply the information contained in the entry.

Structure Member	DESCRIPTION
flags	Combination of bit values to indicate the presence of any optional <code>RsslMapEntry</code> content. For more information about flag values, refer to Table 11.3.3.4.
action	The entry <code>action</code> helps to manage change processing rules and tells the consumer how to apply the information contained in the entry. For specific information about possible <code>action</code> 's associated with an <code>RsslMapEntry</code> , refer to Table 11.3.3.5.

Table 116: Rssl MapEntry Structure Members

Structure Member	DESCRIPTION
encKey	Length and pointer to the encoded map entry key information. The encoded type of the key corresponds to the <code>Rss1Map</code> 's <code>keyPrimitiveType</code> . The key value must be a base primitive type and cannot be blank, <code>RSSL_DT_ARRAY</code> , or <code>RSSL_DT_UNKNOWN</code> primitive types. If populated on encode functions, this indicates that the key is pre-encoded and <code>encKey</code> will be copied while encoding. While decoding, this would contain only this encoded <code>Rss1MapEntry</code> key's payload and length information.
permData	(Optional) Specifies authorization information for this specific entry. If present, <code>RSSL_MPEF_HAS_PERM_DATA</code> should be set. <code>permData</code> has a maximum allowed length of 32,767 bytes. <ul style="list-style-type: none"> For more information on permissioning, refer to Section 11.4. For more information about <code>Rss1MapEntry</code> flag values, refer to Table 11.3.3.4.
encData	Length and pointer to the encoded content of this <code>Rss1MapEntry</code> . If populated on encode functions, this indicates that data is pre-encoded, and <code>encData</code> will be copied while encoding. While decoding, this would refer to this encoded <code>Rss1MapEntry</code> 's payload and length information.

Table 116: `Rss1MapEntry` Structure Members (Continued)

11.3.3.4 `Rss1MapEntry` Flag Enumeration Value

FLAG ENUMERATION	MEANING
<code>RSSL_MPEF_HAS_PERM_DATA</code>	Indicates that the container entry includes a <code>permData</code> member and also specifies any authorization information for this entry. For more information, refer to Section 11.4.

Table 117: `Rss1MapEntry` Flags

11.3.3.5 `Rss1MapEntry` Action Enumeration Values

ACTION ENUMERATION	MEANING
<code>RSSL_MPEA_ADD_ENTRY</code>	Indicates that the consumer should add the entry. An add action typically occurs when an entry is initially provided. It is possible for multiple add actions to occur for the same entry. If this occurs, any previously received data associated with the entry should be replaced with the newly added information.
<code>RSSL_MPEA_UPDATE_ENTRY</code>	Indicates that the consumer should update any previously stored or displayed information with the contents of this entry. An update action typically occurs when an entry has already been added and changes to the contents need to be conveyed. If an update action occurs prior to the add action for the same entry, the update action should be ignored.
<code>RSSL_MPEA_DELETE_ENTRY</code>	Indicates that the consumer should remove any stored or displayed information associated with the entry. No map entry payload is included when the action is delete.

Table 118: `Rss1MapEntry` Actions

11.3.3.6 RsslMapEntry Encoding Interfaces

An **RsslMapEntry** can be encoded from pre-encoded data or by encoding individual pieces of information as they are provided.

ENCODE INTERFACE	DESCRIPTION
rsslEncodeMapInit	<p>Begins encoding of an RsslMap which can include summary data (Section 11.5) and local set definitions (Section 11.2).</p> <ul style="list-style-type: none"> If summary data and set definitions are pre-encoded, they can be populated on the encSummaryData and encSetDefs prior to calling rsslEncodeMapInit. Additional work is not needed to complete encoding this content. If summary data and set definitions are not pre-encoded, rsslEncodeMapInit performs the Init for these values. You must call the corresponding Complete functions after this content is encoded. Summary data and set definition encoded length hint values can be passed into this function to reserve the appropriate amount of space while encoding. If either is not being encoded or the approximate encoded length is unknown, a value of 0 can be passed in. This is required only when content is not pre-encoded.
rsslEncodeMapComplete	<p>Completes the encoding of an RsslMap. This function expects the same RsslEncodeIterator that was used with rsslEncodeMapInit, any summary data, set data, and all entries.</p> <ul style="list-style-type: none"> If encoding was successful, the RsslBool success parameter should be set to true to finish encoding. If encoding of any component failed, the RsslBool success parameter should be set to false to roll back to the last previously successful encoded point in the contents. <p>All map content should be encoded prior to this call.</p>
rsslEncodeMapSummaryDataComplete	<p>Completes encoding of any non-pre-encoded RsslMap summary data. If RSSL_MPFI_HAS_SUMMARY_DATA is set and encSummaryData is not populated, summary data is expected after rsslEncodeMapInit or rsslEncodeMapSetDefsComplete returns. This function expects the same RsslEncodeIterator that was used with previous map encoding functions.</p> <ul style="list-style-type: none"> If encoding of summary data was successful, the RsslBool success parameter should be true to finish encoding. If encoding of summary data failed, the RsslBool success parameter should be set to false to roll back to the last previously successful encoded point in the contents. <p>If both RSSL_MPFI_HAS_SUMMARY_DATA and RSSL_MPFI_HAS_SET_DEFS are present, then set definitions are expected first, and summary data is encoded after the call to rsslEncodeMapSetDefsComplete.</p>

Table 119: **Rssl MapEntry** Encode Functions

Encode Interface	Description
rsslEncodeMapSetDefsComplete	<p>Completes encoding of any non pre-encoded local set definition data. If RSSL_MPFI_HAS_SET_DEFS is set and encSetDefs is not populated, local set definition data is expected after rsslEncodeMapInit returns. This function expects the same RsslEncodeIterator that was used with rsslEncodeMapInit.</p> <ul style="list-style-type: none"> • If set definition data is encoded successfully, the RsslBool success parameter should be true to finish encoding. • If set definition data failed to encode, the RsslBool success parameter should be set to false to roll back to the last previously successful encoded point in the contents. <p>If both RSSL_MPFI_HAS_SUMMARY_DATA and RSSL_MPFI_HAS_SET_DEFS are present, set definitions are expected first, while any summary data is encoded after the call to rsslEncodeMapSetDefsComplete.</p>
rsslEncodeMapEntry	<p>Encodes an RsslMapEntry from pre-encoded data. This function expects the same RsslEncodeIterator that was used with rsslEncodeMapInit. The pre-encoded map entry payload can be passed in via the RsslMapEntry.encData parameter.</p> <ul style="list-style-type: none"> • If the entry key is pre-encoded, this can be passed in via RsslMapEntry.encKey void* set to NULL. • If the entry key is not pre-encoded, a pointer to the primitive key representation can be passed in via void* without populating encKey. <p>This function is called after rsslEncodeMapInit and after completing any summary data and local set definition data encoding.</p>
rsslEncodeMapEntryInit	<p>Encodes an RsslMapEntry from a container type. This function expects the same RsslEncodeIterator used with rsslEncodeMapInit. After this call, housed-type encode functions can be used to encode contained types.</p> <ul style="list-style-type: none"> • If the entry key is pre-encoded, it can be passed in via RsslMapEntry.encKey with void* set to NULL. • If the entry key is not pre-encoded, a pointer to the primitive key representation can be passed in via void* without populating encKey. <p>This function would be called after rsslEncodeMapInit and any summary data and local set definition data encoding has been completed. A max length hint value, associated with the expected maximum encoded length of this entry, can be passed into this function to allow for appropriate space to be reserved while encoding. If the approximate encoded length is unknown, a value of 0 can be passed in.</p>
rsslEncodeMapEntryComplete	<p>Completes the encoding of an RsslMapEntry. This function expects the same RsslEncodeIterator used with rsslEncodeMapInit, rsslEncodeMapEntryInit, and all other encoding for this container.</p> <ul style="list-style-type: none"> • If encoding of this specific map entry was successful, the RsslBool success parameter should be set to true to finish entry encoding. • If encoding of this specific entry failed, the RsslBool success parameter should be set to false to roll back the encoding of only this RsslMapEntry.

Table 119: **Rssl MapEntry** Encode Functions (Continued)

11.3.3.7 RsslMapEntry Encoding Example

The following sample illustrates the encoding of an `RsslMap` containing `RsslFieldList` values. The example encodes three `RsslMapEntry` values as well as summary data:

- The first entry is encoded with an update action type and a passed in key value.
- The second entry is encoded with an add action type, pre-encoded data, and pre-encoded key.
- The third entry is encoded with a delete action type.

This example also demonstrates error handling for the initial encode function. To simplify the example, additional error handling is omitted, though it should be performed.

```

/* populate map structure prior to call to rsslEncodeMapInit */
/* NOTE: the key names used for this example may not correspond to actual name values */

/* indicate that summary data and a total count hint will be encoded */
rsslMap.flags = RSSL_MPFF_HAS_SUMMARY_DATA | RSSL_MPFF_HAS_TOTAL_COUNT_HINT;
/* populate maps keyPrimitiveType and containerType */
rsslMap.containerType = RSSL_DT_FIELD_LIST;
rsslMap.keyPrimitiveType = RSSL_DT_UINT;
/* populate total count hint with approximate expected entry count */
rsslMap.totalCountHint = 3;

/* begin encoding of map - assumes that encIter is already populated with buffer and version */
/* information, store return value to determine success or failure */
/* expect summary data of approx. 100 bytes, no set definition data */
if (( retVal = rsslEncodeMapInit(&encIter, &rsslMap, 100, 0 ) ) < RSSL_RET_SUCCESS)
{
    /* error condition - switch our success value to false so we can roll back */
    success = RSSL_FALSE;
    /* print out message with return value string, value, and text */
    printf("Error %s (%d) encountered with rsslEncodeMapInit. Error Text: %s\n",
        rsslRetCodeToString(retVal), retVal, rsslRetCodeInfo(retVal));
}
else
{
    /* mapInit encoding was successful */
    /* create a single RsslMapEntry and RsslFieldList and reuse for each entry */
    RsslMapEntry mapEntry = RSSL_INIT_MAP_ENTRY;
    RsslFieldList fieldList = RSSL_INIT_FIELD_LIST;
    RsslUInt entryKeyUInt = 0;

    /* encode expected summary data, init for this was done by rsslEncodeMapInit - this type
       should */
    /* match rsslMap.containerType */
    {
        /* now encode nested container using its own specific encode functions */
        /* begin encoding of field list - using same encIterator as map list */

```

```

fieldList.flags = RSSL_FLF_HAS_STANDARD_DATA;

if (( retVal = rsslEncodeFieldListInit(&encIter, &fieldList, 0, 0)) < RSSL_RET_SUCCESS)

    /*----- Continue encoding field entries. Refer to the example in Section 11.3.1.6 -----*/
}

/* Complete nested container encoding */
retVal = rsslEncodeFieldListComplete(&encIter, success);
}

/* complete encoding of summary data. If any field list encoding failed, success is false */
/*
retVal = rsslEncodeMapSummaryDataComplete(&encIter, success);

/* FIRST Map Entry: encode entry from non pre-encoded data and key. Approx. encoded length
unknown */
mapEntry.action = RSSL_MPEA_UPDATE_ENTRY;
entryKeyUInt = 1;
retVal = rsslEncodeMapEntryInit(&encIter, &mapEntry, &entryKeyUInt, 0);
/* encode contained field list - this type should match rsslMap.containerType */
{
    /* now encode nested container using its own specific encode functions */
    /* clear, then begin encoding of field list - using same encIterator as map */
    rsslClearFieldList(&fieldList);
    fieldList.flags = RSSL_FLF_HAS_STANDARD_DATA;

    if (( retVal = rsslEncodeFieldListInit(&encIter, &fieldList, 0, 0)) < RSSL_RET_SUCCESS)

        /*----- Continue encoding field entries. Refer to the example in Section 11.3.1.6 -----*/
    }

    /* Complete nested container encoding */
    retVal = rsslEncodeFieldListComplete(&encIter, success);
}
retVal = rsslEncodeMapEntryComplete(&encIter, success);

/* SECOND Map Entry: encode entry from pre-encoded buffer containing an encoded
RsslFieldList */
/* because we are re-populating all values on RsslMapEntry, there is no need to clear it */
mapEntry.action = RSSL_MPEA_ADD_ENTRY;
/* assuming pEncUInt RsslBuffer contains the pre-encoded key with length and data properly
populated
*/
mapEntry.encKey.length = pEncUInt->length;
mapEntry.encKey.data = pEncUInt->data;
/* assuming pEncFieldList RsslBuffer contains the pre-encoded payload with data and length
populated
*/

```

```

mapEntry.encData.length = pEncFieldList->length;
mapEntry.encData.data = pEncFieldList->data;

/* void* parameter is passed in as NULL because pre-encoded key is set on RsslMapEntry
itself */
retVal = rsslEncodeMapEntry(&encIter, &mapEntry, NULL);

/* THIRD Map Entry: encode entry with delete action. Delete actions have no payload */
/* need to ensure that RsslMapEntry is appropriately cleared
/* - clearing will ensure that encData and encKey are properly emptied */
rsslClearMapEntry(&mapEntry);

mapEntry.action = RSSL_MPEA_DELETE_ENTRY;
entryKeyUInt = 3;
/* void* parameter is passed in as pointer to key primitive value. encData is empty for
delete */
retVal = rsslEncodeMapEntry(&encIter, &mapEntry, &entryKeyUInt);
}

/* complete map encoding. If success parameter is true, this will finalize encoding.
/* If success parameter is false, this will roll back encoding prior to rsslEncodeMapInit */
retVal = rsslEncodeMapComplete(&encIter, success);

```

Code Example 27: Rssl Map Encoding Example

11.3.3.8 RsslMapEntry Decoding Interfaces

A decoded `RsslMapEntry` structure provides access to the encoded content of the map entry. You can skip further decoding of the entry's content by invoking the entry decoder to move to the next `RsslMapEntry` or you can further decode the contents by invoking the specific contained-type's decode function.

DECODE INTERFACE	DESCRIPTION
<code>rsslDecodeMap</code>	Begins decoding an <code>RsslMap</code> . This function will decode from the <code>RsslBuffer</code> referred to by the passed-in <code>RsslDecodeIterator</code> .
<code>rsslDecodeMapEntry</code>	Decodes an <code>RsslMapEntry</code> and can optionally decode the <code>RsslMapEntry.encKey</code> . This function expects the same <code>RsslDecodeIterator</code> that was used with <code>rsslDecodeMap</code> . This populates <code>encData</code> with encoded entry contents and <code>encKey</code> with the encoded entry key. After this function returns, you can use the <code>RsslMap.containerType</code> to invoke the correct contained-type's decode functions. Calling <code>rsslDecodeMapEntry</code> again continues the decoding of the next entry in the <code>RsslMap</code> until no more entries are available. <ul style="list-style-type: none"> If <code>void*</code> parameter is <code>NULL</code>, decoding of entry key is not performed. If <code>void*</code> is passed in as a pointer to the type defined in <code>RsslMap.keyPrimitiveType</code>, the entry key will also be decoded into the passed-in primitive. As entries are received, the action indicates how to apply contents.

Table 120: Rssl MapEntry Decode Functions

11.3.3.9 RsslMapEntry Decode Example

The following sample demonstrates the decoding of an `RsslMap` and is structured to decode each entry to the contained value. This sample assumes that the housed container type is an `RsslFieldList` and that the `keyPrimitiveType` is `RSSL_DT_INT`. This sample also uses the `rsslDecodeMapEntry` function to perform key decoding. Typically an application would invoke the specific container-type decoder for the housed type or use a switch statement to allow for a more generic map entry decoder. This example uses the same `RsslDecodeIterator` when calling the content's decoder function. An application could optionally use a new `RsslDecodeIterator` by setting the `encData` on a new iterator. To simplify the sample, some error handling is omitted.

```

/* decode contents into the map structure */
if ((RetVal = rsslDecodeMap(&decIter, &rsslMap)) >= RSSL_RET_SUCCESS)
{
    /* create primitive value to have key decoded into and a single map entry to reuse */
    RsslInt rsslInt = 0;
    RsslMapEntry mapEntry = RSSL_INIT_MAP_ENTRY;

    /* if summary data is present, invoking decoder for that type (instead of DecodeEntry)
     * indicates to UPA that user wants to decode summary data */
    if (rsslMap.flags & RSSL_MPF_HAS_SUMMARY_DATA)
    {
        /* summary data is present. Its type should be an rsslMap.containerType */
        RsslFieldList fieldList;
        retVal = rsslDecodeFieldList(&decIter, &fieldList, 0);
        /* Continue decoding field entries. Refer to the example in Section 11.3.1.8 */
    }

    /* decode each map entry, passing in pointer to keyPrimitiveType decodes mapEntry key as
     * well */
    while ((RetVal = rsslDecodeMapEntry(&decIter, &mapEntry, &rsslInt)) != RSSL_RET_END_OF_CONTAINER)
    {
        if (RetVal < RSSL_RET_SUCCESS)
        {
            /* decoding failure tends to be unrecoverable */
            printf("Error %s (%d) encountered with rsslDecodeMapEntry. Error Text: %s\n",
                   rsslRetCodeToString(RetVal), RetVal, rsslRetCodeInfo(RetVal));
        }
        else
        {
            RsslFieldList fieldList;
            retVal = rsslDecodeFieldList(&decIter, &fieldList, 0);
            /* Continue decoding field entries. Refer to the example in Section 11.3.1.8 */
        }
    }
}
else
{
}

```

```

/* decoding failure tends to be unrecoverable */
printf("Error %s (%d) encountered with rsslDecodeMap. Error Text: %s\n",
       rsslRetCodeToString(retval), retval, rsslRetCodeInfo(retval));
}

```

Code Example 28: Rssl Map Decoding Example

11.3.3.10 RsslMap Utility Functions

The Transport API provides the following utility functions to aid with the use of the [RsslMap](#) type:

FUNCTION NAME	DESCRIPTION
rsslClearMap	Clears members from an RsslMsg structure. Useful for structure reuse.
rsslClearMapEntry	Clears members from an RsslMapEntry structure. Useful for structure reuse.

Table 121: Rssl Map Utility Functions

11.3.4 Rss1Series

The **Rss1Series** is a uniform container type. Each entry, known as an **Rss1SeriesEntry**, contains only encoded data. This container is often used to represent table-based information, where no explicit indexing is present or required. An **Rss1Series** can contain zero to N^{10} entries, where zero entries indicates an empty **Rss1Series**.

11.3.4.1 Rss1Series Structure Members

Structure Member	DESCRIPTION
flags	A combination of bit values (flags) that indicates the presence of optional Rss1Series content. For more information about flag values, refer to Section 11.3.4.2.
containerType	The Rss1DataType enumeration value that describes the container type of each Rss1SeriesEntry 's payload.
totalCountHint	A four-byte unsigned integer that indicates an approximate total number of entries associated with this stream. This is typically used when multiple Rss1Series containers are spread across multiple parts of a refresh message (For more information about message fragmentation and multi-part message handling, refer to Section 13.1). The totalCountHint provides an approximation of the total number of entries sent across all series on all parts of the refresh message. This information is useful when determining the amount of resources to allocate for caching or displaying all expected entries. totalCountHint values have a range of 0 to 1,073,741,824.
encSummaryData	Sets the length and pointer to encoded summary data, if any, contained in the message. If populated, summary data contains information that applies to every entry encoded in the Rss1Series (e.g., currency type). The container type of summary data should match the containerType specified on the Rss1Series . If encSummaryData is populated while encoding, the contents will be used as pre-encoded summary data. For more information, refer to Section 11.5. Encoded summary data a maximum allowed length of 32,767 bytes.
encSetDefs	Sets the length and pointer to the encoded local set definitions, if any, contained in the message. If populated, these definitions correspond to data contained within this Rss1Series 's entries and are used to encode or decode their contents. For more information, refer to Section 11.6. Encoded local set definitions have a maximum allowed length of 32,767 bytes.
encEntries	Sets the length and pointer to the all encoded key-value pair encoded data, if any, contained in the message. This refers to encoded Rss1Series payload and length data.

Table 122: **Rss1Series** Structure Members

10. An **Rss1Series** currently has a maximum entry count of 65,535. This type has an approximate maximum encoded length of 4 gigabytes but may be limited to 65,535 bytes if housed inside of a container entry. The content of an **Rss1SeriesEntry** has a maximum encoded length of 65,535 bytes. These limitations can change in subsequent releases.

11.3.4.2 RsslSeries Flag Enumeration Values

FLAG ENUMERATION	MEANING
RSSL_SRF_HAS_TOTAL_COUNT_HINT	Indicates the presence of the <code>totalCountHint</code> member, which can provide an approximation of the total number of entries sent across maps on all parts of the refresh message. Such information is useful when determining resource allocation for caching or displaying all expected entries.
RSSL_SRF_HAS_SUMMARY_DATA	Indicates that the <code>RsslSeries</code> contains summary data. <ul style="list-style-type: none"> If set while encoding, summary data must be provided by encoding or populating <code>encSummaryData</code> with pre-encoded information. If set while decoding, summary data is contained as part of <code>RsslSeries</code> and the user can choose to decode it.
RSSL_SRF_HAS_SET_DEFS	Indicates that the <code>RsslSeries</code> contains local set definition information. Local set definitions correspond to data contained in this <code>RsslSeries</code> 's entries and encode or decode their contents. For more information, refer to Section 11.6.

Table 123: `RsslSeries` Flags

11.3.4.3 RsslSeriesEntry Structure Members

Each `RsslSeriesEntry` can house other Container Types only. `RsslSeries` is a uniform type, where `RsslSeries.containerType` indicates the single type housed in each entry. As entries are received, they are appended to any previously received entries.

Structure Member	DESCRIPTION
encData	Length and pointer to the encoded content of this <code>RsslSeriesEntry</code> . <ul style="list-style-type: none"> If populated on encode functions, this indicates that data is pre-encoded and <code>encData</code> will be copied while encoding. If populated while decoding, this refers to this encoded <code>RsslSeriesEntry</code>'s payload and length data.

Table 124: `RsslSeriesEntry` Structure Members

11.3.4.4 RsslSeriesEntry Encoding Interfaces

An `RsslSeriesEntry` can be encoded from pre-encoded data or by encoding individual pieces of information as they are provided.

Encode Interface	Description
<code>rsslEncodeSeriesInit</code>	<p>Begins encoding an <code>RsslSeries</code> and allows for the encoding of summary data (Section 11.5) and local set definitions (Section 11.6). Further summary data, set definitions, or entries can be encoded after this function returns.</p> <ul style="list-style-type: none"> If summary data or set definitions are pre-encoded they can be populated on the <code>encSummaryData</code> and <code>encSetDefs</code> prior to calling <code>rsslEncodeSeriesInit</code>. No additional work is needed to complete the encoding of this content. If summary data or set definitions are not pre-encoded, <code>rsslEncodeSeriesInit</code> will perform the <code>Init</code> for these components. After this content is encoded, the corresponding <code>Complete</code> functions must be called. Summary data and set definition encoded length hint values can be passed into this function to reserve space while encoding. If either is not being encoded or the approximate encoded length is unknown, a value of <code>0</code> can be passed in. This is only needed when the content is not pre-encoded.
<code>rsslEncodeSeriesComplete</code>	<p>Completes the encoding of an <code>RsslSeries</code>. This function expects the same <code>RsslEncodeIterator</code> used with <code>rsslEncodeSeriesInit</code>, any summary data, set data, and all entries.</p> <ul style="list-style-type: none"> If encoding was successful, the <code>RsslBool success</code> parameter should be set to true to finish encoding. If the encoding of any component failed, the <code>RsslBool success</code> parameter should be false to roll back to the last previously successful encoded point in the contents. <p>All series content should be encoded prior to this call.</p>
<code>rsslEncodeSeriesSummaryDataComplete</code>	<p>Completes the encoding of any non-pre-encoded <code>RsslSeries</code> summary data. If the <code>RSSL_SRF_HAS_SUMMARY_DATA</code> flag is set and <code>encSummaryData</code> is not populated, summary data is expected after <code>rsslEncodeSeriesInit</code> or <code>rsslEncodeSeriesSetDefsComplete</code> returns. This function expects the same <code>RsslEncodeIterator</code> used with previous series encoding functions.</p> <ul style="list-style-type: none"> If encoding of summary data was successful, the <code>RsslBool success</code> parameter should be true to finish encoding. If encoding of summary data failed, the <code>RsslBool success</code> parameter should be false to roll back to the encoding prior to summary data. If both <code>RSSL_SRF_HAS_SUMMARY_DATA</code> and <code>RSSL_SRF_HAS_SET_DEFS</code> are present, set definitions are expected first, while any summary data is encoded after the call to <code>rsslEncodeSeriesSetDefsComplete</code>.

Table 125: `Rssl Series` Encode Functions

ENCODE INTERFACE	DESCRIPTION
rsslEncodeSeriesSetDfsComplete	<p>Completes encoding of any non pre-encoded local set definition data. If the RSSL_SRF_HAS_SET_DEFS flag is set and <code>encSetDfs</code> is not populated, local set definition data is expected after <code>rsslEncodeSeriesInit</code> returns. This function expects the same <code>RsslEncodeIterator</code> used with <code>rsslEncodeSeriesInit</code>.</p> <ul style="list-style-type: none"> • If encoding of set definition data was successful, the <code>RsslBool success</code> parameter should be true to finish encoding. • If encoding of set definition data failed, the <code>RsslBool success</code> parameter should be false to roll back to the encoding prior to set definition data. <p>If both RSSL_SRF_HAS_SUMMARY_DATA and RSSL_SRF_HAS_SET_DEFS are present, set definitions are expected first, while any summary data is encoded after the call to <code>rsslEncodeSeriesSetDfsComplete</code>.</p>
rsslEncodeSeriesEntry	<p>Encodes an <code>RsslSeriesEntry</code> from pre-encoded data. This function expects the same <code>RsslEncodeIterator</code> used with <code>rsslEncodeSeriesInit</code>. The pre-encoded series entry payload can be passed in via <code>RsslSeriesEntry.encData</code>. <code>rsslEncodeSeriesEntry</code> is called after <code>rsslEncodeSeriesInit</code> and any summary data and local set definition data encoding has been completed.</p>
rsslEncodeSeriesEntryInit	<p>Encodes an <code>RsslSeriesEntry</code> from a container type. <code>rsslEncodeSeriesEntryInit</code> expects the same <code>RsslEncodeIterator</code> used with <code>rsslEncodeSeriesInit</code>. After this call, you can use housed-type encode functions to encode the contained type. The contained type's encode function would be called after <code>rsslEncodeSeriesInit</code> and any summary data and local set definition data encoding has been completed.</p> <p>A max length hint value, associated with the expected maximum encoded length of this entry, can be passed into this function reserve space while encoding. If the approximate encoded length is not known, a value of 0 can be passed in.</p>
rsslEncodeSeriesEntryComplete	<p>Completes the encoding of an <code>RsslSeriesEntry</code>. This function expects the same <code>RsslEncodeIterator</code> used with <code>rsslEncodeSeriesInit</code>, <code>rsslEncodeSeriesEntryInit</code>, and all other encoding for this container.</p> <ul style="list-style-type: none"> • If encoding was successful, the <code>RsslBool success</code> parameter should be true to finish entry encoding. • If encoding of this specific entry fails, <code>RsslBool success</code> parameter should be false to roll back the encoding of only this <code>RsslSeriesEntry</code>.

Table 125: Rssl Series Encode Functions (Continued)

11.3.4.5 RsslSeries Encoding Example

The following sample illustrates how to encode an **RsslSeries** containing **RsslElementList** values. The example encodes two **RsslSeriesEntry** values as well as summary data.

- The first entry is encoded from an unencoded element list.
- The second entry is encoded from a buffer containing a pre-encoded element list.

The example demonstrates error handling for the initial encode function. To simplify the example, additional error handling is omitted, though it should be performed.

```
/* populate series structure prior to call to rsslEncodeSeriesInit */

/* indicate that summary data and a total count hint will be encoded */
rsslSeries.flags = RSSL_SRF_HAS_SUMMARY_DATA | RSSL_SRF_HAS_TOTAL_COUNT_HINT;
/* populate containerType and total count hint */
rsslSeries.containerType = RSSL_DT_ELEMENT_LIST;
rsslSeries.totalCountHint = 2;

/* begin encoding of series - assumes that encIter is already populated with buffer and
version
/* information, store return value to determine success or failure */
/* summary data approximate encoded length is unknown, pass in 0 */
if ((retVal = rsslEncodeSeriesInit(&encIter, &rsslSeries, 0, 0 )) < RSSL_RET_SUCCESS)
{
    /* error condition - switch our success value to false so we can roll back */
    success = RSSL_FALSE;
    /* print out message with return value string, value, and text */
    printf("Error %s (%d) encountered with rsslEncodeSeriesInit. Error Text: %s\n",
        rsslRetCodeToString(retVal), retVal, rsslRetCodeInfo(retVal));
}
else
{
    /* series init encoding was successful */
    /* create a single RsslSeriesEntry and RsslElementList and reuse for each entry */
    RsslSeriesEntry seriesEntry = RSSL_INIT_SERIES_ENTRY;
    RsslElementList elementList = RSSL_INIT_ELEMENT_LIST;

    /* encode expected summary data, init for this was done by rsslEncodeSeriesInit - this type
should
    /* match rsslSeries.containerType */
    {
        /* now encode nested container using its own specific encode functions */
        /* begin encoding of element list - using same encIterator as series */
        elementList.flags = RSSL_ELF_HAS_STANDARD_DATA;

        if ((retVal = rsslEncodeElementListInit(&encIter, &elementList, 0, 0)) <
RSSL_RET_SUCCESS)
```

```

/*----- Continue encoding element entries. See example in Section 11.3.2 ----- */

/* Complete nested container encoding */
 retVal = rsslEncodeElementListComplete(&encIter, success);
}

/* complete encoding of summary data. If any element list encoding failed, success is
false */
retVal = rsslEncodeSeriesSummaryDataComplete(&encIter, success);

/* FIRST Series Entry: encode entry from unencoded data. Approx. encoded length unknown */
retVal = rsslEncodeSeriesEntryInit(&encIter, &seriesEntry, 0);
/* encode contained element list - this type should match rsslSeries.containerType */
{
    /* now encode nested container using its own specific encode functions */
    /* clear, then begin encoding of element list - using same encIterator as series */
    rsslClearElementList(&elementList);
    elementList.flags = RSSL_ELF_HAS_STANDARD_DATA;

    if ((retVal = rsslEncodeElementListInit(&encIter, &elementList, 0, 0)) <
RSSL_RET_SUCCESS)

        /*----- Continue encoding element entries. See example in Section 11.3.2 ----- */

        /* Complete nested container encoding */
        retVal = rsslEncodeElementListComplete(&encIter, success);

        /* SECOND Series Entry: encode entry from pre-encoded buffer containing an encoded
RsslElementList */
        /* assuming pEncElementList RsslBuffer contains the pre-encoded payload with data and
length */
        /* populated */
        seriesEntry.encData.length = pEncElementList->length;
        seriesEntry.encData.data = pEncElementList->data;

        retVal = rsslEncodeSeriesEntry(&encIter, &seriesEntry);
    }

    /* complete series encoding. If success parameter is true, this will finalize encoding.
    /* If success parameter is false, this will roll back encoding prior to rsslEncodeSeriesInit
     */
    retVal = rsslEncodeSeriesComplete(&encIter, success);
}

```

Code Example 29: Rssl Series Encoding Example

11.3.4.6 RsslSeriesEntry Decoding Interfaces

A decoded `RsslSeriesEntry` structure provides access to the encoded content of the series entry. Further decoding of the entry's content can be skipped (by invoking the entry decoder to move to the next `RsslSeriesEntry`) or the contents can be further decoded (by invoking the specific contained type's decode function).

Decode Interface	Description
<code>rsslDecodeSeries</code>	Begins decoding an <code>RsslSeries</code> . This function decodes from the <code>RsslBuffer</code> specified by <code>RsslDecodeIterator</code> .
<code>rsslDecodeSeriesEntry</code>	Decodes an <code>RsslSeriesEntry</code> . This function expects the same <code>RsslDecodeIterator</code> used with <code>rsslDecodeSeries</code> and populates <code>encData</code> with encoded entry. After <code>rsslDecodeSeriesEntry</code> returns, you can use <code>RsslSeries.containerType</code> to invoke the correct contained type's decode functions. Calling <code>rsslDecodeSeriesEntry</code> again decodes the next entry in the <code>RsslSeries</code> until no more entries are available. As entries are received, they are appended to previously received entries.

Table 126: Rssl Series Decode Functions

11.3.4.7 RsslSeries Decoding Example

The following sample illustrates how to decode an `RsslSeries` and is structured to decode each entry to the contained value. The sample code assumes the housed container type is an `RsslElementList`. Typically an application invokes the specific container type decoder for the housed type or uses a switch statement to allow for a more generic series entry decoder. This example uses the same `RsslDecodeIterator` when calling the content's decoder function. An application could optionally use a new `RsslDecodeIterator` by setting `encData` on a new iterator. To simplify the sample, some error handling is omitted.

```

/* decode contents into the series structure */
if (( retVal = rsslDecodeSeries(&decIter, &rsslSeries) ) >= RSSL_RET_SUCCESS)
{
    /* create single series entry and reuse while decoding each entry */
    RsslSeriesEntry seriesEntry = RSSL_INIT_SERIES_ENTRY;
    /* if summary data is present, invoking decoder for that type (instead of DecodeEntry)
     * indicates to UPA that user wants to decode summary data */
    if (rsslSeries.flags & RSSL_SRF_HAS_SUMMARY_DATA)
    {
        /* summary data is present. Its type should be that of rsslSeries.containerType */
        RsslElementList elementList;
        retVal = rsslDecodeElementList(&decIter, &elementList, 0);
        /* Continue decoding element entries. See example in Section 11.3.2 */
    }

    /* decode each series entry until there are no more left */
    while (( retVal = rsslDecodeSeriesEntry(&decIter, &seriesEntry) ) != RSSL_RET_END_OF_CONTAINER)
    {
        if (retVal < RSSL_RET_SUCCESS)
        {
            /* decoding failure tends to be unrecoverable */
            printf("Error %s (%d) encountered with rsslDecodeSeriesEntry. Error Text: %s\n",

```

```

        rsslRetCodeToString(retval), retval, rsslRetCodeInfo(retval));
    }
    else
    {
        RsslElementList elementList;
        retval = rsslDecodeElementList(&decIter, &elementList, 0);
        /* Continue decoding element entries. See example in Section 11.3.2 */
    }
}
else
{
    /* decoding failure tends to be unrecoverable */
    printf("Error %s (%d) encountered with rsslDecodeSeries. Error Text: %s\n",
           rsslRetCodeToString(retval), retval, rsslRetCodeInfo(retval));
}

```

Code Example 30: **Rssl Series** Decoding Example

11.3.4.8 **RsslSeries** Utility Functions

The Transport API provides the following utility functions for use with the **RsslSeries** type.

FUNCTION NAME	DESCRIPTION
<code>rsslClearSeries</code>	Clears members from an RsslSeries structure. Useful for structure reuse.
<code>rsslClearSeriesEntry</code>	Clears members from an RsslSeriesEntry structure. Useful for structure reuse.

Table 127: **Rssl Series** Utility Functions

11.3.5 RsslVector

The **RsslVector** is a uniform container type of **index**-value pair entries. Each entry, known as an **RsslVectorEntry**, contains an index that correlates to the entry's position in the information stream and value. An **RsslVector** can contain zero to N^{11} entries (zero entries indicates an empty **RsslVector**).

11.3.5.1 RsslVector Structure Members

Structure Member	Description
flags	A combination of bit values that indicate special behaviors and whether optional RsslVector content is present. For more information about flag values, refer to Section 11.3.5.2.
containerType	An RsslDataType enumeration value that describes the container type of each RsslVectorEntry 's payload.
totalCountHint	A four-byte, unsigned integer that indicates the approximate total number of entries sent across all vectors on all parts of the refresh message. totalCountHint is typically used when multiple RsslVector containers are spread across multiple parts of a refresh message (for more information about message fragmentation and multi-part message handling, refer to Section 13.1). Such information helps in determining the amount of resources to allocate for caching or displaying all expected entries. totalCountHint values have a range of 0 to 1,073,741,824.
encSummaryData	The length and pointer to any encoded summary data contained in the message. If populated, summary data contains information that applies to every entry encoded in the RsslVector (e.g. currency type). The container type of summary data must match the containerType specified on the RsslVector . If encSummaryData is populated while encoding, contents are used as pre-encoded summary data. Encoded summary data a maximum allowed length of 32,767 bytes. For more information, refer to Section 11.5.
encSetDefs	Length and pointer to any encoded local set definitions contained in the message. If populated, these definitions correspond to data contained within this RsslVector 's entries and are used to encode or decode their contents. Encoded local set definitions have a maximum allowed length of 32,767 bytes. For more information, refer to Section 11.6.
encEntries	Length and pointer to any encoded index -value pair encoded data contained in the message. This would refer to encoded RsslVector payload and length information.

Table 128: **Rssl Vector** Structure Members

11. An **Rssl Vector** currently has a maximum entry count of 65,535. This type has an approximate maximum encoded length of 4 gigabytes but may be limited to 65,535 bytes if housed inside of a container entry. The content of an **Rssl VectorEntry** has a maximum encoded length of 65,535 bytes. These limitations can change in future releases.

11.3.5.2 RsslVector Flag Enumeration Values

FLAG ENUMERATION	MEANING
RSSL_VTF_HAS_TOTAL_COUNT_HINT	Indicates that the <code>totalCountHint</code> member is present. <code>totalCountHint</code> can provide an approximation of the total number of entries sent across all vectors on all parts of the refresh message. Such information is useful in determining the amount of resources to allocate for caching or displaying all expected entries.
RSSL_VTF_HAS_PER_ENTRY_PERM_DATA	Indicates that permission information is included with some vector entries. The <code>RsslVector</code> encoding functionality sets this flag value on the user's behalf if an entry is encoded with its own <code>permData</code> . A decoding application can check this flag to determine whether a contained entry has <code>permData</code> and is often useful for fan out devices (if an entry does not have <code>permData</code> , the fan out device can likely pass on data and not worry about special permissioning for the entry). Each entry also indicates the presence of permission data via the use of <code>RSSL_VTEF_HAS_PERM_DATA</code> . Refer to Section 11.3.5.4.
RSSL_VTF_HAS_SUMMARY_DATA	Indicates that the <code>RsslVector</code> contains summary data. <ul style="list-style-type: none"> If this flag is set while encoding, summary data must be provided by encoding or populating <code>encSummaryData</code> with pre-encoded data. If this flag is set while decoding, summary data is contained as part of <code>RsslVector</code> and the user can choose whether to decode it.
RSSL_VTF_HAS_SET_DEFS	Indicates that the <code>RsslVector</code> contains local set definition information. Local set definitions correspond to data contained in this <code>RsslVector</code> 's entries and are used for encoding or decoding their contents. For more information, refer to Section 11.6.
RSSL_VTF_SUPPORTS_SORTING	Indicates that the <code>RsslVector</code> may leverage sortable action types. If an <code>RsslVector</code> is sortable, all components must properly handle changing index values based on insert and delete actions. If a component does not properly handle these action types, it can result in the corruption of the <code>RsslVector</code> 's contents. For more information on proper handling, refer to Section 11.3.5.5.

Table 129: `RsslVector` Flags

11.3.5.3 RsslVectorEntry Structure Members

Each `RsslVectorEntry` can house other Container Types only. `RsslVector` is a uniform type, whereas `RsslVector.containerType` indicates the single-type housed in each entry. Each entry has an associated action which informs the user of how to apply the data contained in the entry.

Structure Member	DESCRIPTION
flags	A combination of bit values that indicate whether optional <code>RsslVectorEntry</code> content is present. For more information about flag values, refer to Section 11.3.5.4.
action	<code>action</code> helps to manage change processing rules and informs the consumer of how to apply the entry's data. For specific information about possible <code>action</code> 's associated with an <code>RsslVectorEntry</code> , refer to Section 11.3.5.5.
index	Indicates the entry's position in the <code>RsslVector</code> . This value can change over time based on other <code>RsslVectorEntry</code> actions. <code>index</code> has an allowable range of 0 to 1,073,741,823.
permData	(Optional) Specifies authorization information for this specific entry. If present, the <code>RSSL_VTEF_HAS_PERM_DATA</code> flag should be set. <ul style="list-style-type: none"> For more information, refer to Section 11.4. For more information about <code>RsslVectorEntry</code> flag values, refer to Section 11.3.5.4. <code>permData</code> has a maximum allowed length of 32,767 bytes.
encData	Length and pointer to this <code>RsslVectorEntry</code> 's encoded content. <ul style="list-style-type: none"> If populated using encode functions, this indicates that data is pre-encoded and <code>encData</code> is copied while encoding. If populated while decoding, this refers to this encoded <code>RsslVectorEntry</code>'s payload and length information.

Table 130: `Rssl VectorEntry` Structure Members

11.3.5.4 RsslVectorEntry Flag Enumeration Value

FLAG ENUMERATION	MEANING
<code>RSSL_VTEF_HAS_PERM_DATA</code>	Indicates the presence of the <code>permData</code> member in this container entry and indicates authorization information for this entry. For more information, refer to Section 11.4.

Table 131: `Rssl VectorEntry` Flag

11.3.5.5 RsslVectorEntry Action Enumeration Values

ACTION ENUMERATION	MEANING
RSSL_VTEA_SET_ENTRY	Indicates that the consumer should set the entry at this index position. A set action typically occurs when an entry is initially provided. It is possible for multiple set actions to target the same entry. If this occurs, any previously received data associated with the entry should be replaced with the newly-added information. RSSL_VTEA_SET_ENTRY can apply to both sortable and non-sortable vectors.
RSSL_VTEA_UPDATE_ENTRY	Indicates that the consumer should update any previously stored or displayed information with the contents of this entry. An update action typically occurs when an entry is already set or inserted and changes to the contents are required. If an update action occurs prior to the set or insert action for the same entry, the update action should be ignored. RSSL_VTEA_UPDATE_ENTRY can apply to both sortable and non-sortable vectors.
RSSL_VTEA_CLEAR_ENTRY	Indicates that the consumer should remove any stored or displayed information associated with this entry's index position. RSSL_VTEA_CLEAR_ENTRY can apply to both sortable and non-sortable vectors. No entry payload is included when the action is a 'clear.'
RSSL_VTEA_INSERT_ENTRY	Applies only to a sortable vector. The consumer should insert this entry at the index position. Any higher order index positions are incremented by one (e.g., if inserting at index position 5 the existing position 5 becomes 6, existing position 6 becomes 7, and so forth).
RSSL_VTEA_DELETE_ENTRY	Applies only to a sortable vector. The consumer should remove any stored or displayed data associated with this entry's index position. Any higher order index positions are decremented by one (e.g., if deleting at index position 5 the existing position 5 is removed, position 6 becomes 5, position 7 becomes 6, and so forth). No entry payload is included when the action is a 'delete.'

Table 132: **Rssl VectorEntry Actions**

11.3.5.6 RsslVectorEntry Encoding Interfaces

An `RsslVectorEntry` can be encoded from pre-encoded data or by encoding data as it arrives.

ENCODE INTERFACE	DESCRIPTION
<code>rsslEncodeVectorInit</code>	<p>Begins encoding an <code>RsslVector</code>. This function allows for the encoding of summary data (Section 11.5) and local set definitions (Section 11.6). Further summary data, set definitions, and/or entries can be encoded after this function returns.</p> <ul style="list-style-type: none"> If summary data and set definitions are pre-encoded, they can be populated on the <code>encSummaryData</code> and <code>encSetDefs</code> prior to calling <code>rsslEncodeVectorInit</code>. No additional work is needed to complete the encoding of this content. If summary data and set definitions are not pre-encoded, <code>rsslEncodeVectorInit</code> will perform the <code>Init</code> for these components. After encoding this content, the corresponding <code>Complete</code> functions must be called. Summary data and set definition encoded length hint values can be passed into this function to allow reserve space while encoding. If either is not being encoded or the approximate encoded length is unknown, a value of <code>0</code> can be passed in. This is only needed when the content is not pre-encoded.
<code>rsslEncodeVectorComplete</code>	<p>Completes the encoding of an <code>RsslVector</code>. This function expects the same <code>RsslEncodeIterator</code> used with <code>rsslEncodeVectorInit</code>, any summary data, set data, and all entries.</p> <ul style="list-style-type: none"> If all components encoded successfully, the <code>RsslBool success</code> parameter should be true to finish encoding. If any component failed to encode, the <code>RsslBool success</code> parameter should be false to roll back encoding to the last successfully-encoded point in the contents. <p>Vector content should be encoded prior to this call.</p>
<code>rsslEncodeVectorSummaryDataComplete</code>	<p>Completes the encoding of any non-pre-encoded <code>RsslVector</code> summary data. If <code>RSSL_VTF_HAS_SUMMARY_DATA</code> is set and <code>encSummaryData</code> is not populated, summary data is expected after <code>rsslEncodeVectorInit</code> or <code>rsslEncodeVectorSetDefsComplete</code> returns. This function expects the same <code>RsslEncodeIterator</code> used with previous vector encoding functions.</p> <ul style="list-style-type: none"> If summary data was encoded successfully, the <code>RsslBool success</code> parameter should be true to finish encoding. If summary data failed to encode, the <code>RsslBool success</code> parameter should be false to roll back to the last successfully-encoded point prior to summary data. If both <code>RSSL_VTF_HAS_SUMMARY_DATA</code> and <code>RSSL_VTF_HAS_SET_DEFS</code> are present, set definitions are expected first, while summary data is encoded after the call to <code>rsslEncodeVectorSetDefsComplete</code>.

Table 133: `Rssl Vector` Encode Functions

ENCODE INTERFACE	DESCRIPTION
rsslEncodeVectorSetDfsComplete	<p>Completes encoding of any non-pre-encoded local set definition data. If RSSL_VTF_HAS_SET_DEFS is set and encSetDfs is not populated, local set definition data is expected after rsslEncodeVectorInit returns. This function expects the same RsslEncodeIterator used with rsslEncodeVectorInit.</p> <ul style="list-style-type: none"> • If set definition data encoded successfully, the RsslBool success parameter should be true to finish encoding. • If set definition data failed to encode, the RsslBool success parameter should be false to roll back to the last successfully-encoded point prior to set definition data. • If both RSSL_VTF_HAS_SUMMARY_DATA and RSSL_VTF_HAS_SET_DEFS are present, set definitions are expected first, and then any summary data is encoded after the call to rsslEncodeVectorSetDfsComplete.
rsslEncodeVectorEntry	<p>Encodes an RsslVectorEntry from pre-encoded data. This function expects the same RsslEncodeIterator used with rsslEncodeVectorInit. The pre-encoded vector entry payload can be passed in via RsslVectorEntry.encData. This function is called after rsslEncodeVectorInit and after the completion of any summary data and local set definition data encoding.</p>
rsslEncodeVectorEntryInit	<p>Encodes an RsslVectorEntry from a container type. This function expects the same RsslEncodeIterator used with rsslEncodeVectorInit. After this call, housed-type encode functions can encode the contained type. This function is called after rsslEncodeVectorInit and after the completion of any summary data and local set definition data encoding.</p> <p>A max length hint value, corresponding to the expected maximum encoded length of this entry, can be passed into this function to reserve space while encoding. If you do not know the approximate encoded length, you can pass in a value of 0.</p>
rsslEncodeVectorEntryComplete	<p>Completes the encoding of an RsslVectorEntry. This function expects the same RsslEncodeIterator used with rsslEncodeVectorInit, rsslEncodeVectorEntryInit, and all other encoding for this container.</p> <ul style="list-style-type: none"> • If encoding of this specific vector entry was successful: the RsslBool success parameter should be true to finish entry encoding. • If encoding of this specific entry failed, RsslBool success parameter should be false to roll back encoding of only this RsslVectorEntry.

Table 133: **Rssi Vector** Encode Functions (Continued)

11.3.5.7 RsslVector Encoding Example

The following sample demonstrates how to encode an `RsslVector` containing `RsslSeries` values. The example encodes three `RsslVectorEntry` values as well as summary data:

- The first entry is encoded from an unencoded series
- The second entry is encoded from a buffer containing a pre-encoded series and has perm data
- The third is a clear action type with no payload.

This example demonstrates error handling for the initial encode function. To simplify the example, additional error handling is omitted (though it should be performed).

```

/* populate vector structure prior to call to rsslEncodeSeriesInit */

/* indicate that summary data and a total count hint will be encoded */
rsslVector.flags = RSSL_VTF_HAS_SUMMARY_DATA | RSSL_VTF_HAS_TOTAL_COUNT_HINT |
    RSSL_VTF_HAS_PER_ENTRY_PERM_DATA;
/* populate containerType and total count hint */
rsslVector.containerType = RSSL_DT_SERIES;
rsslVector.totalCountHint = 3;

/* begin encoding of vector - assumes that encIter is already populated with
/* buffer and version information, store return value to determine success or failure */
/* summary data approximate encoded length is 50 bytes */
if ((retVal = rsslEncodeVectorInit(&encIter, &rsslVector, 50, 0 )) < RSSL_RET_SUCCESS)
{
    /* error condition - switch our success value to false so we can roll back */
    success = RSSL_FALSE;
    /* print out message with return value string, value, and text */
    printf("Error %s (%d) encountered with rsslEncodeVectorInit. Error Text: %s\n",
        rsslRetCodeToString(retVal), retVal, rsslRetCodeInfo(retVal));
}
else
{
    /* vector init encoding was successful */
    /* create a single RsslVectorEntry and RsslSeries and reuse for each entry */
    RsslVectorEntry vectorEntry = RSSL_INIT_VECTOR_ENTRY;
    RsslSeries rsslSeries = RSSL_INIT_SERIES;

    /* encode expected summary data, init for this was done by rsslEncodeVectorInit
     * - this type should match rsslVector.containerType */
    {
        /* now encode nested container using its own specific encode functions */
        /* begin encoding of series - using same encIterator as vector */
        if ((retVal = rsslEncodeSeriesInit(&encIter, &rsslSeries, 0, 0)) < RSSL_RET_SUCCESS)

            /*----- Continue encoding series entries. Refer to the example in Section 11.3.4.5*/
    }
}

```

```

/* Complete nested container encoding */
 retVal = rsslEncodeSeriesComplete(&encIter, success);
}

/* complete encoding of summary data. If any series entry encoding failed, success is
false */
retVal = rsslEncodeVectorSummaryDataComplete(&encIter, success);

/* FIRST Vector Entry: encode entry from unencoded data. Approx. encoded length 90 bytes
*/
/* populate index and action, no perm data on this entry */
vectorEntry.index = 1;
vectorEntry.flags = RSSL_VTEF_NONE;
vectorEntry.action = RSSL_VTEF_UPDATE_ENTRY;
retVal = rsslEncodeVectorEntryInit(&encIter, &vectorEntry, 90);
/* encode contained series - this type should match rsslVector.containerType */
{
    /* now encode nested container using its own specific encode functions */
    /* clear, then begin encoding of series - using same encIterator as vector */
    rsslClearSeries(&rsslSeries);
    if ((retVal = rsslEncodeSeriesInit(&encIter, &rsslSeries, 0, 0)) < RSSL_RET_SUCCESS)

        /*----- Continue encoding series entries. See example in Section 11.3.4 ----- */

    /* Complete nested container encoding */
    retVal = rsslEncodeSeriesComplete(&encIter, success);
}
retVal = rsslEncodeVectorEntryComplete(&encIter, success);

/* SECOND Vector Entry: encode entry from pre-encoded buffer containing an encoded
RsslSeries */
/* assuming pEncSeries RsslBuffer contains the pre-encoded payload with data and length
populated
/* and pPermData contains permission data information */
vectorEntry.index = 2;
/* by passing permData on an entry, the map encoding functionality will implicitly set the
/* RSSL_VTF_HAS_PER_ENTRY_PERM flag */
vectorEntry.flags = RSSL_VTEF_HAS_PERM_DATA;
vectorEntry.action = RSSL_VTEF_SET_ENTRY;

vectorEntry.permData.length = pPermData->length;
vectorEntry.permData.data = pPermData->data;

vectorEntry.encData.length = pEncSeries->length;
vectorEntry.encData.data = pEncSeries->data;

retVal = rsslEncodeVectorEntry(&encIter, &vectorEntry);

```

```

/* THIRD Vector Entry: encode entry with clear action, no payload on clear */
/* Should clear entry for safety, this will set flags to NONE */
rsslClearVectorEntry(&vectorEntry);
vectorEntry.index = 3;
vectorEntry.action = RSSL_VTEF_CLEAR_ENTRY;

RetVal = rsslEncodeVectorEntry(&encIter, &vectorEntry);
}
/* complete vector encoding. If success parameter is true, this will finalize encoding.
/* If success parameter is false, this will roll back encoding prior to rsslEncodeVectorInit */
*/
RetVal = rsslEncodeVectorComplete(&encIter, success);

```

Code Example 31: Rssl Vector Encoding Example

11.3.5.8 RsslVectorEntry Decoding Interfaces

A decoded **RsslVectorEntry** structure provides access to the encoded content of the vector entry. Further decoding of the entry's content can be skipped by invoking the entry decoder to move to the next **RsslVectorEntry** or the contents can be further decoded by invoking the specific contained type's decode function.

Decode Interface	Description
<code>rsslDecodeVector</code>	Begins decoding an RsslVector . This function decodes from the RsslBuffer referred to by the passed-in RsslDecodeIterator .
<code>rsslDecodeVectorEntry</code>	Decodes an RsslVectorEntry . This function expects the same RsslDecodeIterator used with <code>rsslDecodeVector</code> and populates <code>encData</code> with an encoded entry. After this function returns, you can use the <code>RsslVector.containerType</code> to invoke the correct contained type's decode functions. Calling <code>rsslDecodeVectorEntry</code> again will continue to decode subsequent entries in RsslVector until no more entries are available. As entries are received, the action will indicate how to apply their contents.

Table 134: Rssl Vector Decode Functions

11.3.5.9 RsslVector Decoding Example

The following sample illustrates how to decode an **RsslVector** and is structured to decode each entry to the contained value. This sample code assumes the housed container type is an **RsslSeries**. Typically an application would invoke the specific container type decoder for the housed type or use a switch statement to allow a more generic series entry decoder. This example uses the same **RsslDecodeIterator** when calling the content's decoder function. Optionally, an application could use a new **RsslDecodeIterator** by setting the `encData` on a new iterator. To simplify the sample, some error handling is omitted.

```

/* decode contents into the vector structure */
if ((RetVal = rsslDecodeVector(&decIter, &rsslVector)) >= RSSL_RET_SUCCESS)
{
    /* create single vector entry and reuse while decoding each entry */
    RsslVectorEntry vectorEntry = RSSL_INIT_VECTOR_ENTRY;
    /* if summary data is present, invoking decoder for that type (instead of DecodeEntry) */

```

```

/* indicates to UPA that user wants to decode summary data */
if (rsslVector.flags & RSSL_VTF_HAS_SUMMARY_DATA)
{
    /* summary data is present. Its type should be that of rsslVector.containerType */
    RsslSeries rsslSeries;
    retVal = rsslDecodeSeries(&decIter, &rsslSeries);
    /* Continue decoding series entries. See example in Section 11.3.4 */
}

/* decode each vector entry until there are no more left */
while ((retVal = rsslDecodeVectorEntry(&decIter, &vectorEntry)) != RSSL_RET_END_OF_CONTAINER)
{
    if (retVal < RSSL_RET_SUCCESS)
    {
        /* decoding failure tends to be unrecoverable */
        printf("Error %s (%d) encountered with rsslDecodeVectorEntry. Error Text: %s\n",
               rsslRetCodeToString(retVal), retVal, rsslRetCodeInfo(retVal));
    }
    else
    {
        RsslSeries rsslSeries;
        retVal = rsslDecodeSeries(&decIter, &rsslSeries);
        /* Continue decoding series entries. Refer to the example in Section 11.3.4*/
    }
}
else
{
    /* decoding failure tends to be unrecoverable */
    printf("Error %s (%d) encountered with rsslDecodeVector. Error Text: %s\n",
           rsslRetCodeToString(retVal), retVal, rsslRetCodeInfo(retVal));
}

```

Code Example 32: **RsslVector** Decoding Example

11.3.5.10 RsslVector Utility Functions

The Transport API provides the following utility functions for use with **RsslVector**.

FUNCTION NAME	DESCRIPTION
rsslClearVector	Clears members from an RsslVector structure. Useful for structure reuse.
rsslClearVectorEntry	Clears members from an RsslVectorEntry structure. Useful for structure reuse.

Table 135: **Rssl Vector** Utility Functions

11.3.6 RssIFilterList

The **RssIFilterList** is a non-uniform container type of **filterId**-value pair entries. Each entry, known as an **RssIFilterEntry**, contains an **id** corresponding to one of 32 possible bit-value identifiers. These identifiers are typically defined by a domain model specification and can indicate interest in or the presence of specific entries through the inclusion of the **filterId** in the message key's **filter** member. An **RssIFilterList** can contain zero to N^{12} entries, where zero indicates an empty **RssIFilterList**, though this type is typically limited by the number of available **filterId** values.

11.3.6.1 RssIFilterList Structure Members

Structure Member	DESCRIPTION
flags	A combination of bit values to indicate presence of optional RssIFilterList content. For more information about flag values, refer to Section 11.3.6.2.
containerType	An RssIDataType value that, for most efficient bandwidth use, should describe the most common container type across all housed filter entries. All housed entries may match this type, though one or more entries may differ. If an entry differs, the entry specifies its own type via the RssIFilterEntry.containerType member.
totalCountHint	A four-byte unsigned integer that indicates an approximate total number of entries associated with this stream. totalCountHint is used typically when multiple RssIFilterList containers are spread across multiple parts of a refresh message (for more information about message fragmentation and multi-part message handling, refer to Section 13.1). totalCountHint is useful in determining the amount of resources to allocate for caching or displaying all expected entries. totalCountHint values have a range of 0 to 1,073,741,824, though the RssIFilterList is typically limited by available filterId values.
encEntries	Length and pointer to the filterId -value pair encoded data, if any, contained in the message. This would refer to the encoded RssIFilterList payload and length information.

Table 136: **RssIFilterList** Structure Members

12. An **RssIFilterList** currently has a maximum entry count of 65,535, though due to the allowable range of id values, this typically does not exceed 32. If all entry count values are allowed, this type has an approximate maximum encoded length of 4 GB but may be limited to 65,535 bytes if housed inside a container entry. The content of an **RssIFilterEntry** has a maximum encoded length of 65,535 bytes. These limitations can change in future releases.

11.3.6.2 RsslFilterList Flag Enumeration Values

FLAG	MEANING
HAS_TOTAL_COUNT_HINT	Indicates the presence of the <code>totalCountHint</code> member. <code>totalCountHint</code> provides an approximation of the total number of entries sent across all filter lists on all parts of the refresh message. This information is useful in determining the amount of resources to allocate for caching or displaying all expected entries.
HAS_PER_ENTRY_PERM_DATA	Indicates some filter entries include permission information. The <code>RsslFilterList</code> encoding functionality sets this flag value on the user's behalf if any entry is encoded with its own <code>permData</code> . A decoding application can check this flag to determine whether any contained entry has <code>permData</code> , often useful for fan out devices (if entries do not have <code>permData</code> , the fan out device can pass along the data and not worry about special permissioning for an entry). Each entry will also indicate permission data presence via the use of the <code>RSSL_FTEF_HAS_PERM_DATA</code> flag. Refer to Section 11.3.6.4.

Table 137: Rssl FilterList Flags

11.3.6.3 RsslFilterEntry Structure Members

Each `RsslFilterEntry` can house only other container types. `RsslFilterList` is a non-uniform type, where the `RsslFilterList.containerType` should indicate the most common type housed in each entry. Entries that differ from this type must specify their own type via `RsslFilterEntry.containerType`.

Structure Member	DESCRIPTION
flags	A combination of bit values that indicate the presence of optional <code>RsslFilterEntry</code> content. For more information about flag values, refer to Section 11.3.6.4.
action	Helps manage change processing rules and informs the consumer how to apply the information contained in the entry. For specific information about possible <code>action</code> 's associated with an <code>RsslFilterEntry</code> , refer to Section 11.3.6.5.
id	An ID associated with the entry. Each possible <code>id</code> corresponds to a bit-value that can be used with the message key's <code>filter</code> member. This bit-value can be specified on the <code>filter</code> to indicate interest in the <code>id</code> when present in an <code>RsslRequestMsg</code> or to indicate presence of the <code>id</code> when present in other messages. For additional information about the filter, refer to Section 12.1.2.
containerType	An <code>RsslDataType</code> enumeration value describing the type of this <code>RsslFilterEntry</code> . If present, the <code>RsslFilterEntry</code> flag (<code>RSSL_FTEF_HAS_CONTAINER_TYPE</code>) should be set by the user. For more information about <code>RsslFilterEntry</code> flag values, refer to Section 11.3.6.4.

Table 138: Rssl FilterEntry Structure Members

Structure Member	DESCRIPTION
permData	<p>(Optional) Specifies authorization information for this entry. If <code>permData</code> is present, the user should set the <code>Rss1FilterEntry</code> flag (<code>RSSL_FTEF_HAS_PERM_DATA</code>).</p> <p><code>permData</code> has a maximum allowed length of 32,767 bytes.</p> <ul style="list-style-type: none"> For more information about <code>Rss1FilterEntry</code> flag values, refer to Section 11.3.6.4. For more information, refer to Section 11.4.
encData	<p>Length and pointer to the <code>Rss1FilterEntry</code>'s encoded content.</p> <ul style="list-style-type: none"> If populated on encode functions, <code>encData</code> indicates that data is pre-encoded, and <code>encData</code> will be copied while encoding. If populated while decoding, this refers to this encoded <code>Rss1FilterEntry</code>'s payload and length information.

Table 138: `Rss1FilterEntry` Structure Members (Continued)

11.3.6.4 `Rss1FilterEntry` Flag Enumeration Values

FLAG ENUMERATION	MEANING
<code>RSSL_FTEF_HAS_PERM_DATA</code>	<p>Indicates the presence of <code>permData</code> in this container entry and indicates authorization information for this entry.</p> <p>For more information, refer to Section 11.4.</p>
<code>RSSL_FTEF_HAS_CONTAINER_TYPE</code>	<p>Indicates the presence of <code>containerType</code> in this entry. This flag is used when the entry's <code>containerType</code> differs from the specified <code>Rss1FilterList.containerType</code>.</p>

Table 139: `Rss1FilterEntry` Flags

11.3.6.5 `Rss1FilterEntry` Action Flag Values

Each entry has an associated `action` which informs the user of how to apply the entry's contents.

ACTION ENUMERATION	MEANING
<code>RSSL_FTEA_SET_ENTRY</code>	<p>Indicates that the consumer should set the entry corresponding to this <code>id</code>. A set action typically occurs when an entry is initially provided. Multiple set actions can occur for the same entry <code>id</code>, in which case, any previously received data associated with the entry <code>id</code> should be replaced with the newly-added information.</p>
<code>RSSL_FTEA_UPDATE_ENTRY</code>	<p>Indicates that the consumer should update any previously stored or displayed information with the contents of this entry. An update action typically occurs when an entry is set and changes to the contents need to be conveyed. An update action can occur prior to the set action for the same entry <code>id</code>, in which case, the update action should be ignored.</p>
<code>RSSL_FTEA_CLEAR_ENTRY</code>	<p>Indicates that the consumer should remove any stored or displayed information associated with this entry's <code>id</code>. No entry payload is included when the action is a clear.</p>

Table 140: `Rss1FilterEntry` Actions

11.3.6.6 RsslFilterEntry Encoding Interfaces

An `RsslFilterEntry` can be encoded from pre-encoded data or by encoding individual pieces of information as they are provided.

Encode Interface	Description
<code>rsslEncodeFilterListInit</code>	Begins encoding an <code>RsslFilterList</code> . <code>containerType</code> should define the most common entry type.
<code>rsslEncodeFilterListComplete</code>	Completes the encoding of an <code>RsslFilterList</code> . This function expects the same <code>RsslEncodeIterator</code> used with <code>rsslEncodeFilterListInit</code> . <ul style="list-style-type: none"> If all entries encoded successfully, the <code>RsslBool success</code> parameter should be set to <code>true</code> to finish encoding. If any entry fails to encode, the <code>RsslBool success</code> parameter should be set to <code>false</code> to roll back to the last successfully encoded point in the contents. All entries should be encoded prior to this call.
<code>rsslEncodeFilterEntry</code>	Encodes an <code>RsslFilterEntry</code> from pre-encoded data. This function expects the same <code>RsslEncodeIterator</code> used with <code>rsslEncodeFilterInit</code> . The pre-encoded filter entry payload can be passed in via <code>RsslFilterEntry.encData</code> . This function can be called after <code>rsslEncodeFilterListInit</code> completes. If this filter entry houses a type other than what is specified in <code>RsslFilterList.containerType</code> , the entry's <code>containerType</code> should be populated to indicate the difference.
<code>rsslEncodeFilterEntryInit</code>	Encodes an <code>RsslFilterEntry</code> from a container type. This function expects the same <code>RsslEncodeIterator</code> used with <code>rsslEncodeFilterListInit</code> . After this call, the housed type encode function can begin to encode the contained type. This function can be called after <code>rsslEncodeFilterListInit</code> has been completed. <ul style="list-style-type: none"> A max length hint value, associated with the expected maximum encoded length of the entry, can be passed into <code>rsslEncodeFilterEntryInit</code> to reserve space while encoding. If the approximate encoded length is not known, a value of <code>0</code> can be passed in. If this filter entry houses a type other than that specified in <code>RsslFilterList.containerType</code>, the entry's <code>containerType</code> value must indicate the difference.
<code>rsslEncodeFilterEntryComplete</code>	Completes the encoding of an <code>RsslFilterEntry</code> . This function expects the same <code>RsslEncodeIterator</code> used with <code>rsslEncodeFilterListInit</code> , <code>rsslEncodeFilterEntryInit</code> , and all other encoding for this container. <ul style="list-style-type: none"> If the filter entry encoded successfully, the <code>RsslBool success</code> parameter should be set to true to finish entry encoding. If the entry failed to encode, the <code>RsslBool success</code> parameter should be set to false to roll back the encoding of this <code>RsslFilterEntry</code>.

Table 141: Rssl FilterList Encode Functions

11.3.6.7 RsslFilterList Encoding Example

The following sample illustrates how to encode an `RsslFilterList` containing a mixture of housed types. The example encodes three `RsslFilterEntry` values:

- The first is encoded from an unencoded element list.
- The second is encoded from a buffer containing a pre-encoded element list.
- The third is encoded from an unencoded map value.

This example demonstrates error handling only for the initial encode function, and to simplify the example, omits additional error handling (though it should be performed).

```

/* populate filterList structure prior to call to rsslEncodeFilterListInit */

filterList.flags = RSSL_FTF_NONE;
/* populate containerType. Because there are two element lists, this is most common so specify
   that type */
filterList.containerType = RSSL_DT_ELEMENT_LIST;

/* begin encoding of filterList - assumes that encIter is already populated with buffer and
   version
/* information, store return value to determine success or failure */
if ((RetVal = rsslEncodeFilterListInit(&encIter, &filterList)) < RSSL_RET_SUCCESS)
{
    /* error condition - switch our success value to false so we can roll back */
    success = RSSL_FALSE;
    /* print out message with return value string, value, and text */
    printf("Error %s (%d) encountered with rsslEncodeFilterListInit. Error Text: %s\n",
        rsslRetCodeToString(RetVal),RetVal, rsslRetCodeInfo(RetVal));
}
else
{
    /* filterList init encoding was successful */
    /* create a single RsslFilterEntry and reuse for each entry */
    RsslFilterEntry filterEntry = RSSL_INIT_FILTER_ENTRY;

    /* FIRST Filter Entry: encode entry from unencoded data. Approx. encoded length 350 bytes
     */
    /* populate id and action */
    filterEntry.id = 1;
    filterEntry.action = RSSL_FTEF_SET_ENTRY;
    RetVal = rsslEncodeFilterEntryInit(&encIter, &filterEntry, 350);
    /* encode contained element list */
    {
        RsslelementList elementList = RSSL_INIT_ELEMENT_LIST;
        elementList.flags = RSSL_ELF_HAS_STANDARD_DATA;
        /* now encode nested container using its own specific encode functions */
    }
}

```

```

if ((RetVal = rsslEncodeElementListInit(&encIter, &elementList, 0, 0)) <
RSSL_RET_SUCCESS)
    /*----- Continue encoding element entries. See example in Section 11.3.2 ----- */
    /* Complete nested container encoding */
    retVal = rsslEncodeElementListComplete(&encIter, success);
}
retVal = rsslEncodeFilterEntryComplete(&encIter, success);

/* SECOND Filter Entry: encode entry from pre-encoded buffer containing an encoded element
list */
/* assuming pEncElemList RsslBuffer contains the pre-encoded payload with data and length
populated */
filterEntry.id = 2;
filterEntry.action = RSSL_FTEF_UPDATE_ENTRY;

filterEntry.encData.length = pEncElemList->length;
filterEntry.encData.data = pEncElemList->data;

RetVal = rsslEncodeFilterEntry(&encIter, &filterEntry);

/* THIRD Filter Entry: encode entry from an unencoded map */
filterEntry.id = 3;;
filterEntry.action = RSSL_FTEF_UPDATE_ENTRY;
/* because type is different from filterList.containerType, we need to specify on entry */
filterEntry.flags = RSSL_FTEF_HAS_CONTAINER_TYPE;
filterEntry.containerType = RSSL_DT_MAP;

RetVal = rsslEncodeFilterEntryInit(&encIter, &filterEntry, 0);
/* encode contained map */
{
    RsslMap rsslMap = RSSL_INIT_MAP;
    rsslMap.keyPrimitiveType = RSSL_DT_ASCII_STRING;
    rsslMap.containerType = RSSL_DT_FIELD_LIST;
    /* now encode nested container using its own specific encode functions */
    if ((RetVal = rsslEncodeMapInit(&encIter, &rsslMap, 0, 0)) < RSSL_RET_SUCCESS)
        /*----- Continue encoding map entries. Refer to the example in Section 11.3.3 ----- */
        /* Complete nested container encoding */
        retVal = rsslEncodeMapComplete(&encIter, success);
    }
    retVal = rsslEncodeFilterEntryComplete(&encIter, success);
}
/* complete filterList encoding. If success parameter is true, this will finalize encoding.
If success parameter is false, this will roll back encoding prior to rsslEncodeFilterListInit
*/
retVal = rsslEncodeFilterListComplete(&encIter, success);

```

Code Example 33: Rssl FilterList Encoding Example

11.3.6.8 RsslFilterEntry Decoding Interfaces

A decoded `RsslFilterEntry` structure provides access to the encoded content of the filter entry. You can skip further decoding of an entry's content by invoking the entry decoder to move to the next `RsslFilterEntry` or the contents can be further decoded by invoking the specific contained type's decode function.

DECODE INTERFACE	DESCRIPTION
<code>rsslDecodeFilterList</code>	Begins decoding of an <code>RsslFilterList</code> . This function decodes from the <code>RsslBuffer</code> specified in <code>RsslDecodeIterator</code> .
<code>rsslDecodeFilterEntry</code>	Decodes an <code>RsslFilterEntry</code> . This function expects the same <code>RsslDecodeIterator</code> that was used with <code>rsslDecodeFilterList</code> . This populates <code>encData</code> with an encoded entry. As an entry is received, its action indicates how to apply contents. After this function returns, the <code>RsslFilterList.containerType</code> (or <code>RsslFilterEntry.containerType</code> if present) can invoke the correct contained type's decode functions. Calling <code>rsslDecodeFilterEntry</code> again decodes the remaining entries in the <code>RsslFilterList</code> .

Table 142: `RsslFilterList` Decode Functions

11.3.6.9 RsslFilterEntry Decoding Example

The following sample illustrates how to decode an `RsslFilterList` and is structured to decode each entry to its contained value. The sample code uses a switch statement to decode the contents of each filter entry. Typically an application invokes the specific container type decoder for the housed type or uses a switch statement to use a more generic series entry decoder. This example uses the same `RsslDecodeIterator` when calling the content's decoder function. Optionally, an application could use a new `RsslDecodeIterator` by setting the `encData` on a new iterator. To simplify the example, some error handling is omitted.

```

/* 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;

    /* decode each filter entry until there are no more left */
    while (( retVal = rsslDecodeFilterEntry(&decIter, &filterEntry)) != RSSL_RET_END_OF_CONTAINER)
    {
        if (retVal < RSSL_RET_SUCCESS)
        {
            /* decoding failure tends to be unrecoverable */
            printf("Error %s (%d) encountered with rsslDecodeFilterEntry. Error Text: %s\n",
                   rsslRetCodeToString(retVal), retVal, rsslRetCodeInfo(retVal));
        }
        else
        {
            /* if filterEntry.containerType is present, switch on that,
               Otherwise switch on filterList.containerType */
            RsslContainerType cType;

```

```

if (filterEntry.flags & RSSL_FTEF_HAS_CONTAINER_TYPE)
    cType = filterEntry.containerType;
else
    cType = filterList.containerType;

switch (cType)
{
    case RSSL_DT_MAP:
        retVal = rsslDecodeMap(&decIter, &rsslMap);
        /* Continue decoding map entries. See example in Section 11.3.3 */
        break;
    case RSSL_DT_ELEMENT_LIST:
        retVal = rsslDecodeElementList(&decIter, &elementList, 0);
        /* Continue decoding element entries. See example in Section 11.3.2 */
        break;
    /* full switch statement omitted to shorten sample code */
}
}
}
}
else
{
    /* decoding failure tends to be unrecoverable */
    printf("Error %s (%d) encountered with rsslDecodeFilterList. Error Text: %s\n",
        rsslRetCodeToString(retVal), retVal, rsslRetCodeInfo(retVal));
}
}

```

Code Example 34: **RsslFilterList** Decoding Example

11.3.6.10 RsslFilterEntry Utility Functions

The Transport API provides the following utility functions for use with **RsslFilterList**.

FUNCTION NAME	DESCRIPTION
rsslClearFilterList	Clears members from an RsslFilterList structure. Useful for structure reuse.
rsslClearFilterEntry	Clears members from an RsslFilterEntry structure. Useful for structure reuse.

Table 143: **RsslFilterList** Utility Functions

11.3.7 Non-RWF Container Types

Transport API messages and container entries allow non-RWF content. Non-RWF content can be:

- A specific type of formatted data such as ANSI Page or XML, where an `Rss1DataType` enumeration value aids in identifying the type.
- A type of customized, user-defined information. You can use `Rss1DataType`'s enumerated range of **225 - 255** to define custom types.

11.3.7.1 Non-RWF Encode Functions

The Transport API provides utility functions to help encode non-RWF types. These functions work in conjunction with `Rss1EncodeIterator` to provide appropriate encoding position and length data to the user, which can then be used with specific functions for the non-RWF type being encoded.

function NAME	DESCRIPTION
<code>rss1EncodeNonRWFDataTypelnit</code>	Uses the <code>Rss1EncodeIterator</code> to populate an <code>Rss1Buffer</code> with encoding information for the user. <code>Rss1Buffer.data</code> contains the position where encoding begins and <code>Rss1Buffer.length</code> contains the number of available bytes for encoding. You can populate this buffer using non-RWF encode functions.
<code>rss1EncodeNonRWFDataTypelComplete</code>	Integrates content encoded into <code>Rss1Buffer</code> with other pre-encoded information. <code>Rss1Buffer.length</code> should be set to the actual number of bytes encoded prior to this function being called.

Table 144: Non-RWF Type Encode Functions

11.3.7.2 Non-RWF Encoding Example

Note: Do not change the value of `Rss1Buffer.data` between calls to `rss1EncodeNonRWFDataTypelnit` and `rss1EncodeNonRWFDataTypelComplete`.

The following sample demonstrates how to encode an `Rss1Series` containing a non-RWF type of ANSI Page. This example demonstrates error handling for the initial encode function while omitting additional error handling (though it should be performed).

```

rss1Series.flags = RSSL_SRF_NONE;
/* populate containerType with the ANSI dataType enumerated value - this could be any non-RWF
   type enum */
rss1Series.containerType = RSSL_DT_ANSI_PAGE;

/* begin encoding of series - assumes that encIter is already populated with buffer and
   version
/* information, store return value to determine success or failure */
if ((RetVal = rss1EncodeSeriesInit(&encIter, &rss1Series, 0, 0 )) < RSSL_RET_SUCCESS)
{
    /* error condition - switch our success value to false so we can roll back */
    success = RSSL_FALSE;
    /* print out message with return value string, value, and text */
    printf("Error %s (%d) encountered with rss1EncodeSeriesInit. Error Text: %s\n",

```

```

    rsslRetCodeToString(retval), retval, rsslRetCodeInfo(retval));
}
else
{
    /* series init encoding was successful */
    /* begin our series entry and then nest ANSI Page inside of it using non-RWF encode
functions */
    RsslSeriesEntry seriesEntry = RSSL_INIT_SERIES_ENTRY;
    /* create an empty buffer for information to be populated into */
    RsslBuffer nonRWFBuffer = RSSL_INIT_BUFFER;

    retval = rsslEncodeSeriesEntryInit(&encIter, &seriesEntry, 0);
    /* encode contained non-RWF type using non-RWF encode functions */
    {
        retval = rsslEncodeNonRWFDataInit(&encIter, &nonRWFBuffer);
        /* now encode nested container using its own specific encode functions -
        Ensure that we do not exceed nonRWFBuffer.length */
        /* we could memcpy into the nonRWFBuffer.data or use it with other encode functions */
        /* The encAnsibuffer shown here is expected to be populated with data from an
        external ANSI encoder. The native ANSI encode functions could be called, instead
        of a memcpy with pre-encoded ANSI content, to directly encode into the nonRWFBuffer
    */
        memcpy(&nonRWFBuffer.data, &encAnsibuffer.data, encAnsibuffer.length);
        /* Set nonRWFBuffer.length to amount of data encoded into buffer and complete */
        nonRWFBuffer.length = encAnsibuffer.length;
        retval = rsslEncodeNonRWFDataComplete(&encIter, &nonRWFBuffer, success);
    }
    retval = rsslEncodeSeriesEntryComplete(&encIter, success);
}
/* complete series encoding. If success parameter is true, this will finalize encoding.
If success parameter is false, this will roll back encoding prior to rsslEncodeSeriesInit */
retval = rsslEncodeSeriesComplete(&encIter, success);

```

Code Example 35: Non-RWF Type Encoding Example

11.3.7.3 Decoding Non-RWF Types

When decoding, the user can obtain non-RWF data via the `encData` member and use this with functions specific to the non-RWF type being decoded.

11.4 Permission Data

Permission Data is optional authorization information. The DACS Lock API provides functionality for creating and manipulating permissioning information. For more information on DACS usage and permission data creation, refer to the *Transport API DACS LOCK Library Reference Manual*.

Permission data can be specified in some messages. When permission data is included in an `RsslRefreshMsg` or an `RsslStatusMsg`, this generally defines authorization information associated with all content on the stream. You can change permission data on an existing stream by sending a subsequent `RsslStatusMsg` or `RsslRefreshMsg` which contains the new permission data. When permission data is included in an `RsslUpdateMsg`, this generally defines authorization information that applies only to that specific `RsslUpdateMsg`.

Permission data can also be specified in some container entries. When a container entry includes permission data, it generally defines authorization information that applies only to that specific container entry. Specific usage and inclusion of permissioning information can be further defined within a domain model specification.

Permission data typically ensures that only entitled parties can access restricted content. On TREP, all content is restricted (or filtered) based on user permissions.

When content is contributed, permission data in an `RsslPostMsg` is used to permission the user who posts the information. If the payload of the `RsslPostMsg` is another message type with permission data (i.e., `RsslRefreshMsg`), the nested message's permissions can change the permission expression associated with the posted item. If permission data for the nested message is the same as permission data on the `RsslPostMsg`, the nested message does not need permission data.

11.5 Summary Data

Some container types allow summary data. **Summary data** conveys information that applies to every entry housed in the container. Using summary data ensures data is sent only once, instead of repetitively including data in each entry. An example of summary data is the currency type because it is likely that all entries in the container share the same currency. Summary data is optional and applications can determine when to employ it.

Specific domain model definitions typically indicate whether summary data should be present, along with information on its content. When included, the `containerType` of the summary data is expected to match the `containerType` of the payload information (e.g., if summary data is present on an `RsslVector`, the `RsslVector.containerType` defines the type of summary data and `RsslVectorEntry` payload).

11.6 Set Definitions and Set-Defined Data

A **Set-Defined Primitive Type** is similar to a primitive type (described in Section 11.2) with several key differences. While primitive types can be encoded as a variable number of bytes, most set-defined primitive types use a fixed-length encoding. Fixed-length encoding can help reduce the number of bytes required to contain the encoded primitive type. **RsslDataType** enumerated values between **64** and **127** are set-defined primitive types and set fixed-length encodings for many base primitive types (e.g., **RSSL_DT_INT_1** is a one-byte fixed-length encoding of **RSSL_DT_INT**). Whereas all primitive types can represent blank data, only several set-defined primitive types can do so. All encoding and decoding continues to use primitive type definitions and should continue to function in the same manner as described in the previous sections. The **RsslDataType** enumeration exposes values that define each set-defined primitive, though these values are only used inside of a set definition. When using set-defined primitive types, a set definition is required to encode or decode content.

A **Set Definition** can define the contents of an **RsslFieldList** or an **RsslElementList** and allow additional optimizations. Use of a set definition can reduce overall encoded content by eliminating repetitive type and length information.

- A set definition describing an **RsslFieldList** contains **fieldId** and type information specified in the same order as the contents are arranged in the encoded field list.
- A set definition describing an **RsslElementList** contains element **name** and type information specified in the same order as the contents are arranged in the encoded element list.

When encoding, in addition to providing set definition information, an application encodes the field list or element list content. Internally the RSSL encoder uses the provided set definition to perform type encoding specific to the definition and omit redundant information needed only in the definition.

When decoding, in addition to providing set definition information, an application decodes the field list or element list content. Internally, the RSSL decoder uses the provided set definition to decode any type-specific optimizations and to reintroduce redundant information omitted during the encoding.

Instead of including multiple instances of the same content, you can use a set definition (i.e., an **RsslMap** containing **RsslFieldList** content in each entry). In this case, a set definition can be provided once as part of the **RsslMap** to define the layout of repetitive field list information contained in the **RsslMapEntry** (i.e., **fieldId**). When encoding each **RsslFieldList**, this content will be omitted because it is included in the set definition.

A set definition can contain primitive type enumerations (Section 11.2), set-defined primitive type enumerations, and container type enumerations (Section 11.3). Encoding and decoding occurs exactly the same as primitive type and container type encoding or decoding.

11.6.1 Set-Defined Primitive Types

Set primitive types do not use separate interface functions for encoding or decoding. Decoding uses the same primitive type decoder used when decoding the primitive type. Because these types can only be contained in an `RsslFieldList` or `RsslElementList`, encoding occurs as usual by calling `rsslEncodeFieldEntry` or `rsslEncodeElementEntry`. When calling these functions, populate the field or element entry using the base primitive type. The table below provides a brief description of each set-defined primitive type, along with its corresponding base primitive type enumeration and its respective RSSL decode interface.

ENUMERATED TYPE NAME	BASE PRIMITIVE TYPE ENUMERATION	PRIMITIVE TYPE	DECODE INTERFACE	TYPE DESCRIPTION
RSSL_DT_INT_1	RSSL_DT_INT	Rssllnt	rsslDecodeIntInt.decode	A signed, one-byte integer type that represents a value up to 7 bits with a one-bit sign (positive or negative). Allowable range is (-2^7) to $(2^7 - 1)$. This type cannot be represented as blank.
RSSL_DT_INT_2	RSSL_DT_INT	Rssllnt	rsslDecodeIntInt.decode	A signed, two-byte integer type that represents a value up to 15 bits with a one-bit sign (positive or negative). Allowable range is (-2^{15}) to $(2^{15} - 1)$. This type cannot be represented as blank.
RSSL_DT_INT_4	RSSL_DT_INT	Rssllnt	rsslDecodeIntInt.decode	A signed, four-byte integer type that represents a value up to 31 bits with a one-bit sign (positive or negative). Allowable range is (-2^{31}) to $(2^{31} - 1)$. This type cannot be represented as blank.
RSSL_DT_INT_8	RSSL_DT_INT	Rssllnt	rsslDecodeIntInt.decode	A signed, eight-byte integer type that represents a value up to 63 bits with a one-bit sign (positive or negative). Allowable range is (-2^{63}) to $(2^{63} - 1)$. This type cannot be represented as blank.
RSSL_DT_UINT_1	RSSL_DT_UINT	RsslUInt	rsslDecodeUIntUInt.decode	An unsigned, one-byte integer type that represents an unsigned value with precision of up to 8 bits. Allowable range is 0 to $(2^8 - 1)$. This type cannot be represented as blank.
RSSL_DT_UINT_2	RSSL_DT_UINT	RsslUInt	rsslDecodeUIntUInt.decode	An unsigned, two-byte integer type that represents an unsigned value with precision of up to 16 bits. Allowable range is 0 to $(2^{16} - 1)$. This type cannot be represented as blank.

Table 145: Set-Defined Primitive Types

ENUMERATED TYPE NAME	BASE PRIMITIVE TYPE ENUMERATION	PRIMITIVE TYPE	DECODE INTERFACE	TYPE DESCRIPTION
RSSL_DT_UINT_4	RSSL_DT_UINT	RsslUInt	rsslDecodeUIntUInt.decode	An unsigned, four-byte integer type that represents an unsigned value with precision of up to 32 bits. Allowable range is 0 to ($2^{32} - 1$). This type cannot be represented as blank.
RSSL_DT_UINT_8	RSSL_DT_UINT	RsslUInt	rsslDecodeUIntUInt.decode	An unsigned, eight-byte integer type that represents an unsigned value with precision of up to 64 bits. Allowable range is 0 to ($2^{64} - 1$). This set-defined primitive type cannot be represented as blank.
RSSL_DT_FLOAT_4	RSSL_DT_FLOAT	RsslFloat	rsslDecodeFloatFloat.decode	A four-byte, floating point type that represents the same range of values allowed by the system float type. Follows the IEEE 754 specification. This type cannot be represented as blank.
RSSL_DT_DOUBLE_8	RSSL_DT_DOUBLE	RsslDouble	rsslDecodeDoubleDouble.decode	An eight-byte, floating point type that represents the same range of values allowed by the system double type. Follows the IEEE 754 specification. This type cannot be represented as blank.
RSSL_DT_REAL_4RB	RSSL_DT_REAL	RsslReal	rsslDecodeRealReal.decode	An optimized RWF representation of a decimal or fractional value which typically requires less bytes on the wire than float or double types. This type allows up to a four-byte value, with a hint value, which can add or remove up to seven trailing zeros, ten decimal places, or fractional denominators up to 256. Allowable range is (- 2^{31}) to ($2^{31} - 1$) This type can be represented as blank. For more details on this type, refer to Section 11.2.1.
RSSL_DT_REAL_8RB	RSSL_DT_REAL	RsslReal	rsslDecodeRealReal.decode	An optimized RWF representation of a decimal or fractional value which typically requires less bytes on the wire than float or double types. This type allows up to an eight byte value, with a hint value, which can add or remove up to seven trailing zeros, 14 decimal places, or fractional denominators up to 256. Allowable range is (- 2^{63}) to ($2^{63} - 1$) This type can be represented as blank. For more details on this type, refer to Section 11.2.1.

Table 145: Set-Defined Primitive Types (Continued)

ENUMERATED TYPE NAME	BASE PRIMITIVE TYPE ENUMERATION	PRIMITIVE TYPE	DECODE INTERFACE	TYPE DESCRIPTION
RSSL_DT_DATE_4	RSSL_DT_DATE	RsslDate	rsslDecodeDateDate.decode	Representation of a date containing month, day, and year values. This value can be represented as blank. For more details on this type, refer to Section 11.2.2.
RSSL_DT_TIME_3	RSSL_DT_TIME	RsslTime	rsslDecodeTimeTime.decode	Representation of a time containing hour, minute, and second values. This value can be represented as blank. For more details on this type, refer to Section 11.2.3.
RSSL_DT_TIME_5	RSSL_DT_TIME	RsslTime	rsslDecodeTimeTime.decode	Representation of a time containing hour, minute, second, and millisecond values. This value can be represented as blank. For more details on this type, refer to Section 11.2.3.
RSSL_DT_DATETIME_7	RSSL_DT_DATETIME	RsslDateTime	rsslDecodeDateTimeDateTime.decode	Combined representation of date and time. Contains all members of RSSL_DT_DATE and hour, minute, and second from RSSL_DT_TIME . This value can be represented as blank. For more details on this type, refer to Section 11.2.4.
RSSL_DT_DATETIME_9	RSSL_DT_DATETIME	RsslDateTime	rsslDecodeDateTimeDateTime.decode	Combined representation of date and time. Contains all members of RSSL_DT_DATE and all members of RSSL_DT_TIME . This value can be represented as blank. For more details on this type, refer to Section 11.2.4.

Table 145: Set-Defined Primitive Types (Continued)

11.6.2 Set Definition Use

In the Transport API, an application can leverage local set definitions. A ***local set definition*** is a set definition sent along with the content it defines. Local set definitions are valid only within the scope of the container of which they are a part and apply only to the information in the container on which they are specified (e.g., an **RsslMap**'s set definition content applies only to the payload within the map's entries). Set definitions are divided into two concrete types

- **Field set definition:** A set definition that defines **RsslFieldList** content
- **Element set definition:** A set definition that defines **RsslElementList** content

Set definitions can contain multiple entries, each defining a specific encoding type for an **RsslFieldEntry** or **RsslElementEntry**.

11.6.2.1 RsslFieldSetDef Structure Members

The following table defines **RsslFieldSetDef** Structure Members. **RsslFieldSetDef** represents a single field set definition and can define the contents of multiple entries in an **RsslFieldList**.

Structure Member	DESCRIPTION
setId	<p>The field set definition's identifier value. Any field list content that leverages this definition should have RsslFieldList.setId match this identifier.</p> <p>setId values have an allowed range of 0 to 32,767. However, only values 0 to 15 are valid for local set definition content. For more information, refer to Section 11.6.</p> <p>For more details on how RsslFieldList indicates the use of a set definition, refer to Section 11.3.1</p>
count	<p>The number of RsslFieldSetDefEntriess contained in this definition. Each entry defines how an RsslFieldEntry is encoded or decoded. A set definition is limited to 255 entries.</p> <p>For more information, refer to Section 11.6.2.2</p>
pEntries	<p>A pointer to the array of RsslFieldSetDefEntriess. Each entry defines how an RsslFieldEntry is encoded or decoded.</p> <p>For more information, refer to Section 11.6.2.2.</p>

Table 146: **Rssl FieldSetDef** Structure Member

11.6.2.2 RsslFieldSetDefEntry Structure Members

Structure Member	DESCRIPTION
fieldId	<p>The fieldId value that corresponds to this entry in the set-defined RsslFieldList content. fieldId is a signed, two-byte value that refers to specific name and type information defined by an external field dictionary, such as the RDMFieldDictionary. Negative fieldId values typically refer to user-defined values while positive fieldId values typically refer to Thomson Reuters-defined values. When encoding, the RsslFieldEntry.fieldId should match the value that the set definition expects. When decoding, the RsslFieldEntry.fieldId is populated with the fieldId value indicated in the set definition.</p> <p>fieldId has an allowable range of -32,768 to 32,767 where positive values are Thomson Reuters-defined and negative values are user-defined. The fieldId value of 0 is reserved to indicate dictionaryId changes, where the type of fieldId 0 is an RsslInt.</p>
dataType	<p>Defines the RsslDataType of the entry as it encodes or decodes when using this set definition. This can be a base primitive type, a set-defined primitive type, or a container type.</p> <ul style="list-style-type: none"> While encoding, populate the RsslFieldEntry.dataType with the base primitive type or container type value that corresponds to the type contained in this definition. While decoding, RsslFieldEntry.dataType is populated with the specific RsslDataType information as indicated by the Set Definition, where any set-defined primitive type is converted to the corresponding base primitive type. <p>For a map of set-defined primitive types and their corresponding base primitive types, refer to Section 11.6.1.</p>

Table 147: **RsslFieldSetDefEntry** Structure Members

11.6.2.3 RsslElementSetDef Structure members

The following table defines **RsslElementSetDef** Structure Members. **RsslElementSetDef** represents a single element set definition, and can define content for multiple entries in an **RsslElementList**.

Structure Member	DESCRIPTION
setId	<p>The field set definition's identifier value. Any element list content that leverages this definition should have the RsslElementList.setId matching this identifier.</p> <p>Though setId values have an allowed range of 0 to 32,767, the only values valid for local set definition content are 0 - 15. These indicate locally defined set definition use. For more information, refer to Section 11.6.</p> <p>For more information about how an RsslElementList indicates use of a set definition, refer to Section 11.3.2.</p>
count	<p>The number of RsslElementSetDefEntries contained in this definition. Each entry defines how to encode or decode an RsslElementEntry. A set definition is limited to 255 entries.</p> <p>For more information, refer to Section 11.6.2.4.</p>
pEntries	<p>A pointer to the array of RsslElementSetDefEntries. Each entry defines how to encode or decode an RsslElementEntry.</p> <p>For more information, refer to Section 11.6.2.4.</p>

Table 148: **RsslElementSetDef** Structure Members

11.6.2.4 RsslElementSetDefEntry Structure Members

Structure Member	Description
name	<p>The <code>name</code> that corresponds to this set-defined element; contained in the structure as an <code>RsslBuffer</code>. Element names are defined outside of the Transport API, typically as part of a domain model specification or dictionary. When encoding, you can optionally populate <code>RsslElementEntry.name</code> with the <code>name</code> expected in the set definition.</p> <p>If <code>name</code> is not used, validation checking is not provided and information might be encoded that does not properly correspond to the definition. When decoding, <code>RsslElementEntry.name</code> is populated with the information indicated in the set definition.</p> <p>The <code>name</code> buffer allows content length ranging from 0 bytes to 32,767 bytes.</p>
dataType	<p>When encoding or decoding an entry using this set definition, <code>dataType</code> defines the entry's <code>RsslDataType</code>. This can be a base primitive type, a set-defined primitive type, or a container type.</p> <ul style="list-style-type: none"> While encoding, populate <code>RsslElementEntry.dataType</code> with the base primitive type or container type value that corresponds to the type contained in this definition. While decoding, populate <code>RsslElementEntry.dataType</code> with the specific <code>RsslDataType</code> information as indicated by set definition, where any set-defined primitive type is converted to the corresponding base primitive type. <p>For a map of set-defined primitive types and their corresponding base primitive types, refer to Section 11.6.1.</p>

Table 149: `RsslElementSetDefEntry` Structure Members

11.6.3 Set Definition Database

A **set definition database** can group definitions together. Using a database can be helpful when the content leverages multiple definitions; the database provides an easy way to pass around all set definitions necessary to encode or decode information. For instance, an **RsslVector** can contain multiple set definitions via a set definition database with the contents of each **RsslVectorEntry** requiring a different definition from the database.

11.6.3.1 RsslLocalFieldSetDefDb Structure Members

RsslLocalFieldSetDefDb represents multiple local field set definitions and uses the following Structure Members.

Structure Member	Description
definitions	An array containing up to fifteen RsslFieldSetDefs . Each contained field set definition defines a unique setId for use in the container.
entries ^a	<p>An RsslBuffer that helps manage memory associated with set definition entries for each RsslFieldSetDef. Optionally, a decoding application can populate RsslBuffer.data and length with its own memory, causing RSSL to decode the definitions into user-provided storage. If RsslBuffer.data and length are not populated, RSSL uses internal memory which will no longer be valid after decoding completes. Though an encoding application does not need to use this, it can be helpful to populate the RsslBuffer.data and length with its own memory, referring to its entry array content.</p> <p>Note: If an application decodes content over multiple threads and data might contain set definitions, to ensure thread safety, the application should populate the RsslLocalFieldSetDefDb.entries or RsslLocalElementSetDefDb.entries buffer with its own memory.</p>

Table 150: **Rssl Local FieldSetDefDb** Structure Members

- a. If an application uses multiple **Rssl Decoder Iterator** structures in the same thread, where each decode iterator requires the use of a local set definition database, the application must provide the memory into which entries decode.

11.6.3.2 RsslLocalElementSetDefDb Structure Members

[RsslLocalElementSetDefDb](#) (which represents multiple local element set definitions) has the following members:

Structure Member	DESCRIPTION
definitions	An array containing up to fifteen RsslElementSetDef structures. Each contained element set definition defines a unique setId for use within the container on which this is present.
entries ^a	<p>An RsslBuffer that helps manage memory associated with set definition entries for each RsslElementSetDef. Optionally, a decoding application can populate RsslBuffer.data and length with its own memory, causing RSSL to decode definitions in user-provided storage. If RsslBuffer.data and length are not populated, RSSL uses internal memory, which is no longer valid after the container decoding completes. Though an encoding application does not need to use this, it might be helpful to populate the RsslBuffer.data and length with its own memory referring to its entry array content.</p> <p>Note: If an application decodes content over multiple threads and the data may contain set definitions, to ensure thread safety, the application should populate the RsslLocalFieldSetDefDb.entries or RsslLocalElementSetDefDb.entries buffer with its own memory.</p>

Table 151: Rssl Local ElementSetDefDb Structure Members

a. Within the same thread, if an application is using multiple [RsslDecodeIterator](#) structures, where each decode iterator requires the use of a local set definition database, the application must provide entries memory for decoding into.

11.6.3.3 Local Set Definition Database Encoding Interfaces

Applications can send or receive local set definitions while using the [RsslMap](#), [RsslVector](#), or [RsslSeries](#) container types. To provide local set definition information, an application can populate the [encSetDefs](#) member with a pre-encoded set definition database, or encode this using the Transport API-provided functionality described in this section.

The following table describes all available encoding functions required to provide set definition database content on an [RsslMap](#), [RsslVector](#), or [RsslSeries](#). When present, this information should apply to any [RsslFieldList](#) or [RsslElementList](#) content within the types' entries. When encoding set-defined field or element list content, the application must pass [RsslLocalFieldSetDefDb](#) or [RsslLocalElementSetDefDb](#) into the [rsslEncodeFieldListInit](#) and [rsslEncodeElementListInit](#) functions.

ENCODE INTERFACE	DESCRIPTION
rsslEncodeLocalFieldSetDefDb	Encodes a non-pre-encoded local field set definition database into its own buffer for use with encSetDefs or directly into an RsslMap , RsslVector , or RsslSeries . After the container's EncodeInit function, local set definition encoding is expected prior to any summary data or container entries.
rsslEncodeLocalElementSetDefDb	Encodes a non-pre-encoded local element set definition database into its own buffer for use with encSetDefs or directly into an RsslMap , RsslVector , or RsslSeries . After the containers EncodeInit function, local set definition encoding is expected prior to any summary data or container entries.
rsslEncodeMapSetDefsComplete	Completes encoding non-pre-encoded element or field set definition database content. This applies to local set definition database content on an RsslMap , refer to Section 11.3.3.

Table 152: Local Set Definition Database Encode Functions

ENCODE INTERFACE	DESCRIPTION
rsslEncodeSeriesSetDefsComplete	Completes encoding non-pre-encoded element or field set definition database content. This applies to local set definition database content on an RsslSeries , refer to Section 11.3.4.
rsslEncodeVectorSetDefsComplete	Completes encoding non-pre-encoded element or field set definition database content. This applies to local set definition database content on an RsslVector , refer to Section 11.3.5.

Table 152: Local Set Definition Database Encode Functions (Continued)

11.6.3.4 Local Set Definition Database Decoding Interfaces

The following table describes decoding functions for use with a local set definition database. When decoding set-defined content, the application can pass the **RsslLocalFieldSetDefDb** or **RsslLocalElementSetDefDb** into the **rsslDecodeFieldList** and **rsslDecodeElementList** functions. If this information is not provided, RSSL skips decoding set-defined content.

DECODE INTERFACE	DESCRIPTION
rsslDecodeLocalFieldSetDefDb	Decodes encSetDefs into a local field set definition database for use when decoding contained RsslFieldList information.
rsslDecodeLocalElementSetDefDb	Decodes encSetDefs into a local field set definition database for use when decoding contained RsslElementList information.

Table 153: Local Set Definition Database Decode Functions

11.6.3.5 Local Set Definition Database Utility Functions

The Transport API provides the following utility functions for use with `RsslLocalFieldSetDefDb` and `RsslLocalElementSetDefDb` types:

FUNCTION NAME	DESCRIPTION
<code>rsslClearFieldSetDefEntry</code>	Clears members from an <code>RsslFieldSetDefEntry</code> structure. Useful for structure reuse.
<code>rsslClearFieldSetDef</code>	Clears members from an <code>RsslFieldSetDef</code> structure. This will not free memory pointed to by <code>pEntries</code> . Useful for structure reuse.
<code>rsslClearLocalFieldSetDefDb</code>	Clears members from an <code>RsslLocalFieldSetDefDb</code> structure. If the user populated the <code>entries</code> buffer with their own memory, this function will not free up memory: it sets <code>length</code> and <code>data</code> to <code>0</code> . This structure must be cleared prior to use.
<code>rsslClearElementSetDefEntry</code>	Clears members from an <code>RsslElementSetDefEntry</code> structure. This will not free memory associated with <code>name.data</code> . Useful for structure reuse.
<code>rsslClearElementSetDef</code>	Clears members from an <code>RsslElementSetDef</code> structure. This will not free memory pointed to by <code>pEntries</code> . Useful for structure reuse.
<code>rsslClearLocalElementSetDefDb</code>	Clears members from an <code>RsslLocalElementSetDefDb</code> structure. If the user populated the <code>entries</code> buffer with their own memory, this function does not free up memory: it sets <code>length</code> and <code>data</code> to <code>0</code> . This structure must be cleared prior to use.

Table 154: Local Set Definition Database Utility Functions

11.6.3.6 Field Set Definition Database Encoding Example

The following example demonstrates encoding of a field set definition database into an [RsslMap](#). The field set definition database contains one definition, made up of three field set definition entries. After set-defined content encoding is completed, an additional standard data field entry is encoded.

```
RsslMap rsslMap = RSSL_INIT_MAP;
/* Create the fieldSetDefDb and field set definition */
RsslLocalFieldSetDefDb fieldSetDefDb;
RsslFieldSetDef fieldSetDef;
/* create entries arrays */
RsslFieldSetDefEntry fieldSetDefEntries[3] =
{
    { 22, RSSL_DT_REAL },      /* Contains BID as an RsslReal */
    { 25, RSSL_DT_REAL_8RB }, /* Contains ASK as an optimized RsslReal */
    { 18, RSSL_DT_TIME_3 }    /* Contains TRADE TIME as an optimized RsslTime */
};

/* Populate the entries into our set definition */
fieldSetDef.setId = 5; /* This definition has an ID of 5 */
fieldSetDef.count = 3; /* There are three entries in this definition */
fieldSetDef.pEntries = fieldSetDefEntries; /* Set this to the array containing the
                                           definitions */

/* Now populate the definition into the set definition Db. If there were more than one
definition,
/* all required defs would be populated into the same Db */
/* Structure must be cleared first */
rsslClearLocalFieldSetDefDb(&fieldSetDefDb);
/* set the definition into the slot that corresponds to its ID */
/* since this definition is ID 5, it goes into definitions array position 5 */
fieldSetDefDbdefinitions[5] = fieldSetDef;

/* begin encoding of map that will contain set def DB - assumes that encIter is already
populated with
/* buffer and version information, store return value to determine success or failure */
rsslMap.flags = RSSL_MPFF_HAS_SET_DEFS;
rsslMap.containerType = RSSL_DT_FIELD_LIST;
rsslMap.keyPrimitiveType = RSSL_DT_UINT;
if ((RetVal = rsslEncodeMapInit(&encIter, &rsslMap, 0, 0)) < RSSL_RET_SUCCESS)
{
    /* error condition - switch our success value to false so we can roll back */
    success = RSSL_FALSE;
    /* print out message with return value string, value, and text */
    printf("Error %s (%d) encountered with rsslEncodeMapInit. Error Text: %s\n",
        rsslRetCodeToString(RetVal), RetVal, rsslRetCodeInfo(RetVal));
}
else
```

```

{
    /* map init encoding was successful */
    RsslMapEntry mapEntry = RSSL_INIT_MAP_ENTRY;
    RsslFieldList fieldList = RSSL_INIT_FIELD_LIST;
    RsslFieldEntry fieldEntry = RSSL_INIT_FIELD_ENTRY;
    RsslReal rsslReal = RSSL_INIT_REAL;
    RsslTime rsslTime = RSSL_INIT_TIME;
    RsslInt rsslUInt = 0;

    /* It expects the local set definition database to be encoded next */
    /* because we are encoding a local field set definition database, we have to call the
    correct function */
    retVal = rsslEncodeLocalFieldSetDefDb(&encIter, &fieldSetDefDb);
    /* Our set definition db is now encoded into the map, we must complete the map portion of
    this
    /* encoding and then begin encoding entries */
    retVal = rsslEncodeMapSetDefsComplete(&encIter, RSSL_TRUE);
    /* begin encoding of map entry - this contains a field list using the set definition encoded
    above */
    mapEntry.action = RSSL_MPEA_ADD_ENTRY;
    mapEntry.flags = RSSL_MPEF_NONE;
    rsslUInt = 100212; /* populate map entry key */
    retVal = rsslEncodeMapEntryInit(&encIter, &mapEntry, &rsslUInt, 0);
    /* set field list flags - this has a setId and set defined data - we can also have standard
    data after
    /* set defined data is encoded */
    fieldList.flags = RSSL_FLF_HAS_SET_ID | RSSL_FLF_HAS_SET_DATA |
    RSSL_FLF_HAS_STANDARD_DATA;
    fieldList.setId = 5; /* this field list will use the set definition from above */
    /* when encoding set defined data, the database containing the necessary definitions must
    be passed
    in */
    retVal = rsslEncodeFieldListInit(&encIter, &fieldList, &fieldSetDefDb, 0);
    /* for each field entry we encode that is set defined, the Rssl encoder verifies that the
    correct
    /* fieldId and content type are passed in. Order must match definition */

    /* Encode FIRST field in set definition */
    fieldEntry.fieldId = 22; /* fieldId of the first set definition entry */
    fieldEntry.dataType = RSSL_DT_REAL; /* base primitive type of the first set definition
    entry */
    rsslReal.hint = RSSL_RH_EXPONENT_2;
    rsslReal.value = 227;
    /* encode the first entry - this matches the fieldId and type specified in the first
    definition entry
    */
    retVal = rsslEncodeFieldEntry(&encIter, &fieldEntry, &rsslReal);
}

```

```

/* Encode SECOND field in set definition */
fieldEntry.fieldId = 25; /* fieldId of the second set definition entry */
fieldEntry.dataType = RSSL_DT_REAL; /* base primitive type of the second set definition
entry */
rsslReal_hint = RSSL_RH_EXPONENT_4;
rsslReal.value = 22801;
/* encode the second entry - this matches the fieldId and type specified in the first
definition entry
*/
retVal = rsslEncodeFieldEntry(&encIter, &fieldEntry, &rsslReal);

/* Encode THIRD field in set definition */
fieldEntry.fieldId = 18; /* fieldId of the third set definition entry */
fieldEntry.dataType = RSSL_DT_TIME; /* base primitive type of the third set definition
entry */
rsslTime.hour = 8;
rsslTime.minute = 39;
rsslTime.second = 24;
/* encode the third entry - this matches the fieldId and type specified in the first
definition entry
*/
retVal = rsslEncodeFieldEntry(&encIter, &fieldEntry, &rsslTime);

/* Encode standard data after field set definition is complete */
fieldEntry.fieldId = 2; /* fieldId of the first standard data entry after set definition is
complete*/
fieldEntry.dataType = RSSL_DT_UINT; /* base primitive type of the first set definition
entry */
/* encode the standard data in the message after set data is complete */
retVal = rsslEncodeFieldEntry(&encIter, &fieldEntry, &rsslUInt);

/* complete encoding of the content */
retVal = rsslEncodeFieldListComplete(&encIter, RSSL_TRUE);
retVal = rsslEncodeMapEntryComplete(&encIter, RSSL_TRUE);
retVal = rsslEncodeMapComplete(&encIter, RSSL_TRUE);

```

Code Example 36: Field Set Definition Database Encoding Example

11.6.3.7 Field Set Definition Database Decoding Example

The following example illustrates how to decode a field set definition database from an `RsslMap`. After decoding the database, it can be passed in while decoding `RsslFieldList` content.

```
RsslMap rsslMap = RSSL_INIT_MAP;
RsslMapEntry mapEntry = RSSL_INIT_MAP_ENTRY;
RsslUInt rsslUInt = 0; /* for decoding map entry keys */
/* Create the fieldSetDefDb to decode into */
RsslLocalFieldSetDefDb fieldSetDefDb;

/* Decode the map */
retVal = rsslDecodeMap(&decIter, &rsslMap);
/* If the map flags indicate that set definition content is present, decode the set def db */
if (rsslMap.flags & RSSL_MPFF_HAS_SET_DEFS)
{
    /* must ensure it is the correct type - if map contents are field list, this is a field set
     * definition
    /* db */
    if (rsslMap.containerType == RSSL_DT_FIELD_LIST)
    {
        rsslClearLocalFieldSetDefDb(&fieldSetDefDb);
        retVal = rsslDecodeLocalFieldSetDefDb(&decIter, &fieldSetDefDb);
    }
    /* If map contents are an element list, this is an element set definition db */
    if (rsslMap.containerType == RSSL_DT_ELEMENT_LIST)
        /* this is an element list set definition db */
}
/* decode map entries */
while ((retVal = rsslDecodeMapEntry(&decIter, &mapEntry, &rsslUInt)) != RSSL_RET_END_OF_CONTAINER)
{
    if (retVal < RSSL_RET_SUCCESS)
    {
        /* decoding failure tends to be unrecoverable */
        printf("Error %s (%d) encountered with rsslDecodeMapEntry. Error Text: %s\n",
               rsslRetCodeToString(retVal), retVal, rsslRetCodeInfo(retVal));
    }
    else
    {
        RsslFieldList fieldList;
        /* entries contain field lists - since there were definitions provided they should be
         * passed
        /* in for field list decoding. Any set defined content will use the definition when
         * decoding.
        /* If set definition db is not passed in, any set content will not be decoded */
    }
}
```

```

    retVal = rsslDecodeFieldList(&decIter, &fieldList, &fieldSetDefDb);
    /* Continue decoding field entries. See example in Section 11.3.1.8 */
}
}

```

Code Example 37: Field Set Definition Database Decoding Example

11.6.3.8 Element Set Definition Database Encoding Example

The following example illustrates how to encode an element set definition database into an `RsslSeries`. The database contains one element set definition with three element set definition entries. After encoding is completed, the sample encodes an additional standard data element entry.

```

RsslSeries rsslSeries = RSSL_INIT_SERIES;
/* Create the elementSetDefDb and element set definition */
RsslLocalElementSetDefDb elementSetDefDb;
RsslElementSetDef elementSetDef;
/* create entries arrays */
RsslElementSetDefEntry elementSetDefEntries[3] =
{
    { { 3, "BID" }, RSSL_DT_REAL }, /* Contains BID as an RsslReal */
    { { 3, "ASK" }, RSSL_DT_REAL_8RB }, /* Contains ASK as an optimized RsslReal */
    { { 10, "TRADE TIME" }, RSSL_DT_TIME_3 } /* Contains TRADE TIME as an optimized RsslTime */
};
/* Populate the entries into our set definition */
elementSetDef.setId = 10; /* This definition has an ID of 10 */
elementSetDef.count = 3; /* There are three entries in this definition */
elementSetDef.pEntries = elementSetDefEntries; /* Set this to the array containing the
definitions */

/* Now populate the definition into the set definition Db. If there were more than one
definition,
/* all required defs would be populated into the same Db */
/* Structure must be cleared first */
rsslClearLocalElementSetDefDb(&elementSetDefDb);
/* set the definition into the slot that corresponds to its ID */
/* since this definition is ID 10, it goes into definitions array position 10 */
elementSetDefDbdefinitions[10] = elementSetDef;

/* begin encoding of series that will contain set def DB - assumes that encIter is already
populated with
/* buffer and version information, store return value to determine success or failure */
rsslSeries.flags = RSSL_SRF_HAS_SET_DEFS;
rsslSeries.containerType = RSSL_DT_ELEMENT_LIST;
if ((retVal = rsslEncodeSeriesInit(&encIter, &rsslSeries, 0, 0 )) < RSSL_RET_SUCCESS)
{
    /* error condition - switch our success value to false so we can roll back */
}

```

```

success = RSSL_FALSE;
/* print out message with return value string, value, and text */
printf("Error %s (%d) encountered with rsslEncodeSeriesInit. Error Text: %s\n",
       rsslRetCodeToString(retval), retval, rsslRetCodeInfo(retval));
}
else
{
    /* series init encoding was successful */
RsslSeriesEntry seriesEntry = RSSL_INIT_SERIES_ENTRY;
RsslElementList elementList = RSSL_INIT_ELEMENT_LIST;
RsslElementEntry elementEntry = RSSL_INIT_ELEMENT_ENTRY;
RsslReal rsslReal = RSSL_INIT_REAL;
RsslTime rsslTime = RSSL_INIT_TIME;
RsslInt rsslUInt = 2112;

/* It expects the local set definition database to be encoded next */
/* because we are encoding a local element set definition database, we have to call the
correct
/* function */
retval = rsslEncodeLocalElementSetDefDb(&encIter, &elementSetDefDb);
/* Our set definition db is now encoded into the series, we must complete the series portion
of this
/* encoding and then begin encoding entries */
retval = rsslEncodeSeriesSetDefsComplete(&encIter, RSSL_TRUE);
/* begin encoding of series entry - this contains an element list using the set definition
encoded
/* above */
retval = rsslEncodeSeriesEntryInit(&encIter, &seriesEntry, 0);
/* set element list flags - this has a setId and set defined data - we can also have
standard data
/* after set defined data is encoded */
elementList.flags = RSSL_ELF_HAS_SET_ID | RSSL_ELF_HAS_SET_DATA |
RSSL_ELF_HAS_STANDARD_DATA;
elementList.setId = 10; /* this element list will use the set definition from above */
/* when encoding set defined data, the database containing the necessary definitions must
be passed
/* in */
retval = rsslEncodeElementListInit(&encIter, &elementList, &elementSetDefDb, 0);
/* for each element entry we encode that is set defined, the Rssl encoder verifies that the
correct
/* element name and content type are passed in. Order must match definition */

/* Encode FIRST element in set definition */
elementEntry.name.length = 3; /* name of the first set definition entry */
elementEntry.name.data = "BID";
elementEntry.dataType = RSSL_DT_REAL; /* base primitive type of the first set definition
entry */
rsslReal_hint = RSSL_RH_EXPONENT_2;

```

```

rsslReal.value = 227;
/* encode the first entry - this matches the name and type specified in the first definition
entry */
retVal = rsslEncodeElementEntry(&encIter, &elementEntry, &rsslReal);

/* Encode SECOND element in set definition */
elementEntry.name.length = 3; /* name of the second set definition entry */
elementEntry.name.data = "ASK";
elementEntry.dataType = RSSL_DT_REAL; /* base primitive type of the second set definition
entry */
rsslReal_hint = RSSL_RH_EXPONENT_4;
rsslReal.value = 22801;
/* encode the second entry - this matches the name and type specified in the first
definition entry */
retVal = rsslEncodeElementEntry(&encIter, &elementEntry, &rsslReal);

/* Encode THIRD field in set definition */
elementEntry.name.length = 10; /* name of the third set definition entry */
elementEntry.name.data = "TRADE TIME";
elementEntry.dataType = RSSL_DT_TIME; /* base primitive type of the third set definition
entry */
rsslTime.hour = 8;
rsslTime.minute = 39;
rsslTime.second = 24;
/* encode the third entry - this matches the name and type specified in the first definition
entry */
retVal = rsslEncodeElementEntry(&encIter, &elementEntry, &rsslTime);

/* Encode standard data after element set definition is complete */
elementEntry.name.length = 15; /* name of the first standard data entry after set
definition is
/* complete*/
elementEntry.name.data = "DISPLAYTEMPLATE";
elementEntry.dataType = RSSL_DT_UINT; /* base primitive type of the first set definition
entry */
/* encode the standard data in the message after set data is complete */
retVal = rsslEncodeElementEntry(&encIter, &elementEntry, &rsslUInt);

/* complete encoding of the content */
retVal = rsslEncodeElementListComplete(&encIter, RSSL_TRUE);
retVal = rsslEncodeSeriesEntryComplete(&encIter, RSSL_TRUE);
retVal = rsslEncodeSeriesComplete(&encIter, RSSL_TRUE);
}

```

Code Example 38: Element Set Definition Database Encoding Example

11.6.3.9 Element Set Definition Database Decoding Example

The following example illustrates how to decode an element set definition database from an `RsslSeries`. After decoding the database, it can be passed in while decoding `RsslElementList` content.

```
RsslSeries rsslSeries = RSSL_INIT_SERIES;
RsslSeriesEntry seriesEntry = RSSL_INIT_SERIES_ENTRY;
/* Create the elementSetDefDb to decode into */
RsslLocalElementSetDefDb elementSetDefDb;

/* Decode the series */
retVal = rsslDecodeSeries(&decIter, &rsslSeries);
/* If the series flags indicate that set definition content is present, decode the set def db */
if (rsslSeries.flags & RSSL_SRF_HAS_SET_DEFS)
{
    /* must ensure it is the correct type - if series contents are element list, this is an
     * element set
    /* definition db */
    if (rsslSeries.containerType == RSSL_DT_ELEMENT_LIST)
    {
        rsslClearLocalElementSetDefDb(&elementSetDefDb);
        retVal = rsslDecodeLocalElementSetDefDb(&decIter, &elementSetDefDb);
    }
    /* If map contents are an field list, this is a field set definition db */
    if (rsslSeries.containerType == RSSL_DT_FIELD_LIST)
        /* this is a field list set definition db */
}

/* decode series entries */
while ((retVal = rsslDecodeSeriesEntry(&decIter, &seriesEntry)) != RSSL_RET_END_OF_CONTAINER)
{
    if (retVal < RSSL_RET_SUCCESS)
    {
        /* decoding failure tends to be unrecoverable */
        printf("Error %s (%d) encountered with rsslDecodeSeriesEntry. Error Text: %s\n",
               rsslRetCodeToString(retVal), retVal, rsslRetCodeInfo(retVal));
    }
    else
    {
        RsslElementList elementList;
        /* entries contain element lists - since there were definitions provided they should be
         * passed
        /* in for element list decoding. Any set defined content will use the definition when
         * decoding.
        /* If set definition db is not passed in, any set content will not be decoded */
        retVal = rsslDecodeElementList(&decIter, &elementList, &elementSetDefDb);
    }
}
```

```
    /* Continue decoding element entries. See example in Section 11.3.2 */  
}
```

Code Example 39: Element Set Definition Database Decoding Example

Chapter 12 Message Package Detailed View

12.1 Concepts

Messages communicate data between system components: to exchange information, indicate status, permission users and access, and for a variety of other purposes. Many messages have associated semantics for efficient use in market data systems to request information, respond to information, or provide updated information. Other messages have relatively loose semantics, allowing for a more dynamic use either inside or outside market data systems.

An individual flow of related messages within a connection is typically referred to as a ***stream***, and the message package allows multiple simultaneous streams to coexist in a connection. An information stream is instantiated between a consuming application and a providing application when the consumer issues an **RsslRequestMsg** followed by the provider responding with an **RsslRefreshMsg** or **RsslStatusMsg**. At this point the stream is established and allows other messages to flow within the stream. The remainder of this chapter discusses streams, stream identification, and stream uniqueness.

The Message Package offers a suite of message header definitions; each optimized to communicate a specific set of information. There are constructs to allow for communication stream identification and to determine uniqueness of streams within a connection. The following sections describe the various constructs, concepts, and processes involved with use of the Message Package.

12.1.1 Common Message Base

Each Transport API message consists of both unique members and common message members. The common members form the **msgBase** portion of the message structure.

12.1.1.1 Message Base Structure Members

STRUCTURE MEMBER	DESCRIPTION
msgClass	<p>Required on all messages.</p> <p>Identifies the specific type of a message (e.g. RsslUpdateMsg, RsslRequestMsg). msgClass allows a range from 0 to 31, with all values reserved for use by Thomson Reuters.</p> <p>For more details about the various message classes, refer to Section 12.1.1.2.</p>
domainType	<p>Required on all messages.</p> <p>Identifies the specific domain message model type. domainType allows a range from 0 to 255, where Thomson Reuters-defined values are between 0 and 127 and user-defined values are between 128 and 255.</p> <p>The domain model definition is decoupled from the API and domain models are typically defined in a specification document. Domain models defined by Thomson Reuters are specified in the <i>Transport API RDM Usage Guide</i>.</p>
containerType	<p>Required on all messages.</p> <p>Identifies the type of message payload content and indicates the presence of a Transport API container type (value 129 - 224), some type of customer-defined, or non-RWF container type (225 - 255), or no message payload (128).</p> <p>For more details about container type definitions and use, refer to Section 11.3.</p>

Table 155: Message Base Structure Members

STRUCTURE MEMBER	DESCRIPTION
msgKey	<p>Required on an RsslRequestMsg and optional on RsslRefreshMsg, RsslStatusMsg, RsslUpdateMsg, RsslGenericMsg, RsslPostMsg, and RsslAckMsg.</p> <p>Houses various attributes that help identify contents flowing within a stream. The msgKey on the initial RsslRefreshMsg, in conjunction with QoS and domainType, uniquely identifies the stream. The key typically includes naming and service-related information.</p> <p>For more information about the message key and stream identification, refer to Section 12.1.2 and Section 12.1.3.</p>
streamId	<p>Required on all messages.</p> <p>Specifies a unique, signed-integer identifier associated with all messages flowing within a stream. streamId allows a range from -2,147,483,648 to 2,147,483,647, where:</p> <ul style="list-style-type: none"> Positive values indicate a consumer-instantiated stream (typically via RsslRequestMsg). Negative values indicate a provider-instantiated stream (often associated with NIPs). <p>For more information about stream identification and streamId use, refer to Section 12.1.3.</p>
encDataBody	<p>Length and pointer to any encoded data contained in the message. If populated, the content type is described by containerType. encDataBody would contain only encoded message payload and length information.</p> <p>encDataBody can represent up to 4,294,967,295 bytes of payload. This payload length is typically limited by the contained type's specification.</p> <ul style="list-style-type: none"> When encoding, encDataBody refers to any pre-encoded message payload. When decoding, encDataBody refers to any encoded message payload.
encMsgBuffer	<p>Length and pointer to the entire encoding of the message. encMsgBuffer would contain both encoded message header and encoded message payload and length information.</p> <p>encMsgBuffer is typically populated only while decoding, and refers to the entire encoded message header and payload.</p>

Table 155: Message Base Structure Members (Continued)

12.1.1.2 Message Class Information

ENUMERATED MESSAGE CLASS	MESSAGE STRUCTURE NAME	DESCRIPTION
RSSL_MC_REQUEST	RsslRequestMsg	<p>Consumers use RsslRequestMsg to express interest in a new stream or modify some parameters on an existing stream; typically results in the delivery of an RsslRefreshMsg or RsslStatusMsg.</p> <p>For more information, refer to Section 12.2.1.</p>
RSSL_MC_REFRESH	RsslRefreshMsg	<p>The Interactive Provider can use this class to respond to a consumer's request for information (solicited) or provide a data resynchronization point (unsolicited).</p> <p>The NIP can use this class to initiate a data flow on a new item stream. Conveys state information, QoS, stream permissioning information, and group information in addition to payload.</p> <p>For more information, refer to Section 12.2.2.</p>
RSSL_MC_UPDATE	RsslUpdateMsg	<p>Interactive or NIPs use the RsslUpdateMsg to convey changes to information on a stream. Update messages typically flow on a stream after delivery of a refresh.</p> <p>For more information, refer to Section 12.2.3.</p>
RSSL_MC_STATUS	RsslStatusMsg	<p>Indicates changes to the stream or data properties. A provider uses RsslStatusMsg to close streams and to indicate successful establishment of a stream when there is no data to convey. For more information, refer to Section 12.2.4.</p> <p>This message can indicate changes:</p> <ul style="list-style-type: none"> • In streamState or dataState • In a stream's permissioning information • To the item group to which the stream belongs
RSSL_MC_CLOSE	RsslCloseMsg	<p>A consumer uses RsslCloseMsg to indicate no further interest in a stream. As a result, the stream should be closed.</p> <p>For more information, refer to Section 12.2.5.</p>
RSSL_MC_GENERIC	RsslGenericMsg	<p>A bi-directional message that does not have any implicit interaction semantics associated with it, thus the name generic. For more information, refer to Section 12.2.6.</p> <p>After a stream is established via a request-refresh/status interaction:</p> <ul style="list-style-type: none"> • A consumer can send this message to a provider. • A provider can send this message to a consumer. • NIPs can send this message to the ADH.

Table 156: Message Class Information

ENUMERATED MESSAGE CLASS	MESSAGE STRUCTURE NAME	DESCRIPTION
RSSL_MC_POST	RsslPostMsg	A consumer uses RsslPostMsg to push content upstream. This information can be applied to an Enterprise Platform cache or routed further upstream to a data source. After receiving posted data, upstream components can republish it to downstream consumers. For more information, refer to Section 12.2.7.
RSSL_MC_ACK	RsslAckMsg	A provider uses RsslAckMsg to inform a consumer of success or failure for a specific RsslPostMsg or RsslCloseMsg . For more information, refer to Section 12.2.8.

Table 156: Message Class Information (Continued)

12.1.2 Message Key

The **Message Key** ([msgKey](#)) houses a variety of attributes that help identify content that flows in a particular stream. A data stream is uniquely identified by the [domainType](#), QoS data, and message key.

12.1.2.1 Message Key Structure Members

Structure Member	DESCRIPTION
flags	Combination of bit values to indicate the presence of optional msgKey members. For more information about flag values, refer to Section 12.1.2.2.
serviceId	The two-byte, unsigned integer identifier associated with a service (a logical mechanism that provides or enables access to a set of capabilities). serviceId allows a range from 0 to 65,535 , with 0 being reserved. This value should correspond to the service content being requested or provided. In the Transport API, a service corresponds to a subset of content provided by a component, where the Source Directory domain defines specific attributes associated with each service. These attributes include information such as QoS, the specific domain types available, and any dictionaries required to consume information from the service. The Source Directory domain model can obtain this and other types of information. For details, refer to the <i>Transport API RDM Usage Guide</i> .
nameType	Numeric value, typically enumerated, that indicates the type of the name member. Examples are User Name or RIC (i.e., the Reuters Instrument Code). nameTypes are defined on a per-domain model basis. nameType allows a range from 0 to 255 . Name type values and rules are defined within domain message model specifications. Values associated with Thomson Reuters domain models can be found in the rssIRDM.h header file.
name	The name associated with the contents of the stream. Specific name type and contents should comply with the rules associated with the nameType member. name is an RsslBuffer type that allows for a name of up to 255 bytes.

Table 157: [msgKey](#) Structure Members

Structure Member	DESCRIPTION
filter	<p>Combination of up to 32 unique filterID bit-values (where each filterID corresponds to a filter bit-value) that describe content for domain model types with an RsslFilterList payload. Filter identifier values are defined by the corresponding domain model specification.</p> <ul style="list-style-type: none"> When specified in an RsslRequestMsg, filter conveys information which entries to include in responses. When specified on a message housing an RsslFilterList payload, filter conveys information about which filter entries are present. <p>For more information, refer to Section 11.3.6.</p>
identifier	<p>User-specified numeric identifier defined on a per-domain model basis. identifier allows a range from -2,147,483,648 to 2,147,483,647.</p> <p>Note: More information should be present as part of the specific domain model definition.</p>
attribContainerType	<p>Identifies the content type of the msgKey.encAttrib information. Can indicate the presence of a Transport API container type (value 129 - 224) or some type of customer-defined container type (225 - 255).</p> <p>For more details about container type definitions and use, refer to Section 11.3.</p>
encAttrib	<p>Length and pointer to additional, encoded, message key attribute information. If populated, contents are described by the attribContainerType member. Additional attribute information typically allows for further uniqueness in the identification of a stream. encAttrib is an RsslBuffer that can represent up to 32,767 bytes of information.</p>

Table 157: **msgKey** Structure Members (Continued)

12.1.2.2 Message Key Flag Enumeration Values

FLAG ENUMERATION	MEANING
RSSL_MKF_HAS_SERVICE_ID	Indicates the presence of the serviceId member.
RSSL_MKF_HAS_NAME	Indicates the presence of the name member.
RSSL_MKF_HAS_NAME_TYPE	Indicates the presence of the nameType member.
RSSL_MKF_HAS_FILTER	Indicates the presence of the filter member.
RSSL_MKF_HAS_IDENTIFIER	Indicates the presence of the identifier member.
RSSL_MKF_HAS_ATTRIB	Indicates the presence of the attribContainerType and encAttrib members.

Table 158: Message Key Flags

12.1.2.3 Message Key Utility Functions

FUNCTION NAME	DESCRIPTION
rsslClearMsgKey	Clears a <code>msgKey</code> structure on an <code>RsslMsg</code> . Useful in clearing only the key members in a message for reuse.
rsslCompareMsgKeys	Compares two <code>msgKey</code> structures to determine whether they are the same. Returns success if the keys match; failure otherwise.
rsslCopyMsgKey	Performs a deep copy of a <code>msgKey</code> and expects the destination <code>msgKey</code> to have sufficient memory to receive the copied data.
rsslAddFilterIdToFilter	Converts a <code>filterId</code> value into the bit-value representation and adds bit-value to the <code>msgKey.filter</code> member. Used with <code>RsslFilterList</code> container types. For more information, refer to Section 11.3.6.
rsslCheckFilterForFilterId	Converts a <code>filterId</code> value into the bit-value representation and checks for the bit-value presence in the <code>msgKey.filter</code> member. Used with <code>RsslFilterList</code> container types. For more information, refer to Section 11.3.6.

Table 159: `MsgKey` Utility Functions

12.1.3 Stream Identification

The Transport API allows users to simultaneously interact across multiple, independent data streams within a single network connection. Each data stream can be uniquely identified by the specified `domainType`¹, QoS, and `msgKey` contents. The `msgKey` contains a variety of attributes used in defining a stream. To avoid repeatedly sending `msgKey` and QoS on all messages in a stream², a signed integer (referred to as a `streamId` or stream identifier) is used. This `streamId` can convey all of the same stream identification information, but consumes only a small, fixed-size (four bytes). A positive value `streamId` indicates a consumer-instantiated stream while a negative value `streamId` indicates a provider-instantiated stream, usually, but not always, associated with a NIP application.

For a consumer application, a positive value `streamId` should be specified on any `RsslRequestMsg`, along with the `domainType`, `msgKey` and additional key attributes, and desired QoS information. An interactive provider application should provide a response, typically an `RsslRefreshMsg`, which contains the same `streamId`, `domainType`, and message key information. If the request specified a QoS range, this response will also contain the concrete or actual QoS being provided for the stream. For more information about QoS, refer to Section 11.2.5.

For an NIP, the initial `RsslRefreshMsg` published for each item should contain `domainType`, message key information, and the QoS being provided for the stream. In addition, the NIP should specify a negative value `streamId` to be associated with the stream for the remainder of the run-time.

12.1.3.1 Stream Comparison

To most efficiently use a connection's bandwidth, Thomson Reuters recommends that you combine like streams when possible. Two streams are identical when all identifying aspects match - that is the two streams have the same `domainType`, provided QoS, and all `msgKey` members. When these message members match, a new stream should not be established, rather the existing stream and `streamId` should be leveraged to consume or provide this content.

A consumer application can issue a subsequent `RsslRequestMsg` using the existing `streamId`, referred to as a *reissue*. This allows the consumer application to obtain an additional refresh, if desired, and to indicate a change in the priority of the stream. The additional solicited `RsslRefreshMsg` can satisfy the additional request, and any `RsslStatusMsg`, `RsslUpdateMsg`, and `RsslGenericMsg` content can be provided to both requestors, if different. This behavior is called fan-out and is the responsibility of the consumer application when combining multiple like-streams into a single stream.

A provider application can choose to allow multiple like-streams to be simultaneously established or, more commonly, it can reject any subsequent requests on a different `streamId` using an `RsslStatusMsg`. In this case, the `RsslStatusMsg` would contain a `streamState` of `RSSL_STREAM_CLOSED_RECOVER`, a `dataState` of `RSSL_DATA_SUSPECT`, and a state `code` of `RSSL_SC_ALREADY_OPEN`. This status message informs the consumer that they already have a stream open for this information and that they should use the existing `streamId` when re-requesting this content. For more details about the state information, refer to Section 11.2.6.

1. When off-stream posting, it is possible for the post messages sent on the Login stream to contain a different `domainType`. This is a specialized use case and more information is available in Section 13.9.

2. `domainType` is present on all messages and cannot be optimized out like quality of service and `msgKey` information.

12.1.3.2 Private Streams

The Transport API provides **private stream** functionality, an easy way to ensure delivery of content only between a stream's two endpoints. Private streams behave in a manner similar to standard streams, with the following exceptions:

- All data on a private stream flow between the end provider and the end consumer of the stream.
- Intermediate components do not fan out content (i.e., do not distribute it to other consumers).
- Intermediate components should not cache content.
- In the event of connection or data loss, intermediate components do not recover content. All private stream recovery is the responsibility of the consumer application.

These behaviors ensure that only the two endpoints of the private stream send or receive content associated with the stream. As a result, a private stream can exchange identifying information so the provider can validate the consumer, even through multiple intermediate components (such as might exist in a TREP deployment). After a private stream is established, content can flow freely within the stream, following either existing market data semantics (i.e., private Market Price domain) or any other user-defined semantics (i.e., bidirectional exchange of **RsslGenericMsgs**).

For more information about private stream instantiation, refer to Section 13.12.

12.1.3.3 Changeable Stream Attributes

A select number of attributes may change during the life of a stream. A consumer can change attributes via a subsequent **RsslRequestMsg** that uses the same **streamId** as previous requests. An Interactive or NIP can change attributes via a subsequent solicited or unsolicited **RsslRefreshMsg**.

The message key's **filter** member, though not typical, can change between the consumer request and provider response. A change is likely due to a difference between the filter entries for which the consumer asks and the filter entries that the provider can provide. If this behavior is allowed within a domain, it is defined on a per-domain model basis. More information should be present as part of the specific domain model definition.

Contents of the message key's **encAttrib** may change. If this behavior is allowed within a domain, it is defined on a per-domain model basis. More information should be present as part of the specific domain model definition.

A consumer can change the **priorityClass** or **priorityCount** via a subsequent **RsslRequestMsg** to indicate more or less interest in a stream. For more information, refer to Section 13.2.

If a QoS range is requested, the provided **RsslRefreshMsg** includes only the concrete QoS, which may be different from the best and worst specified. If a **dynamic** QoS is supported, QoS may occasionally change over the life of the stream, however this should stay within the range requested in **RsslRequestMsg**.

An item's identification might also change, which can result in changes to multiple **msgKey** members. Such a case can occur via a **redirect**, an **RsslRefreshMsg** or **RsslStatusMsg** with a **streamState** of **RSSL_STREAM_REDIRECTED** (for more information on the redirected state value, refer to see Section 11.2.6.2). The user can determine the original item identification from the **msgKey** information previously associated with the **streamId** contained in the redirect message. The new item identification that should be requested is provided via the redirect's **msgKey** member. When a redirect occurs, the stream closes. At this point, the user can open a new stream and continue the flow of data by issuing a new **RsslRequestMsg**, containing the redirected **msgKey**.

Some **RsslRequestMsg.flag** values are allowed to change over the life of a stream. These values include the **RSSL_RQMF_PAUSE** and **RSSL_RQMF_STREAMING** flags, used when pausing or resuming content flow on a stream. For more details, refer to Section 13.6. Additionally, the **RSSL_RQMF_NO_REFRESH** flag can be changed. This allows subsequent reissue requests to be performed where the user does not require a response - this can be useful for a reissue to change the priority of a stream.

12.2 RSSL Messages

12.2.1 RSSL Request Message Class

An OMM consumer uses an `RsslRequestMsg` to express interest in a particular information stream. The request's `msgKey` members help identify the stream and priority information can be used to indicate the stream's importance to the consumer. QoS information can be used to express either a specific desired QoS or a range of acceptable qualities of service that can satisfy the request (refer to Section 13.3).

When an `RsslRequestMsg` is issued with a new `streamId`, this is considered a request to open the stream. If requested information is available and the consumer is entitled to receive the information, this typically results in an `RsslRefreshMsg` being delivered to the consumer, though an `RsslStatusMsg` is also possible - either message can be used to indicate a stream is open. If information is not available or the user is not entitled, an `RsslStatusMsg` is typically delivered to provide more detailed information to the consumer.

Issuing an `RsslRequestMsg` on an existing stream allows a consumer to modify some parameters associated with the stream (also refer to Section 12.1.3.2). Also known as a *reissue*, this can be used to pause or resume a stream (also refer to Section 13.6), change a Dynamic View (also refer to Section 13.8), increase or decrease the stream's priority (also refer to Section 13.2) or request a new refresh.

12.2.1.1 RSSL Request Message Structure Members

Structure Member	DESCRIPTION
msgBase	Members common to all messages. An <code>RsslRequestMsg</code> requires <code>msgKey</code> information to be populated. For details, refer to Section 12.1.1.
extendedHeader	Available for domain-specific user-specified header information. Contents and formatting are defined by the domain model specification. This data is not used in determining stream uniqueness and may not pass through all components. To determine support, refer to the relevant component documentation.
flags	Combination of bit values to indicate special behaviors and the presence of optional <code>RsslRequestMsg</code> content. For more information about flag values, refer to Section 12.2.1.2.
priorityClass	Indicates the class level to associate with the stream. <code>priorityClass</code> can contain values ranging from 0 to 255. For more information about priority and its use, refer to Section 13.2.
priorityCount	Indicates the count at the specified <code>priorityClass</code> level. <code>priorityCount</code> can contain values ranging from 0 to 65,535. For more information about priority and its use, refer to Section 13.2.

Table 160: `Rssl RequestMsg` Structure Members

Structure Member	DESCRIPTION
qos	<p>Sets the allowable QoS for the requested stream.</p> <ul style="list-style-type: none"> When specified without a <code>worstQos</code> member, this is the only allowable QoS for the requested stream. If this QoS is unavailable, the stream is not opened. When specified with a <code>worstQos</code>, this is the best in the range of allowable QoSs. When a QoS range is specified, any QoS within the range is acceptable for servicing the stream. If neither <code>qos</code> nor <code>worstQos</code> are present on the request, this indicates that any available QoS will satisfy the request. <p>Some components may require <code>qos</code> on initial request and reissue messages. See specific component documentation for details.</p> <ul style="list-style-type: none"> For more information, refer to Section 11.2.5. For specific handling information, refer to Section 13.3.
worstQos	<p>The least acceptable QoS for the requested stream. When specified with a <code>qos</code> value, this is the worst in the range of allowable QoSs. When a QoS range is specified, any QoS within the range is acceptable for servicing the stream.</p> <ul style="list-style-type: none"> For more information, refer to Section 11.2.5. For specific handling information, refer to Section 13.3.

Table 160: Rssi RequestMsg Structure Members (Continued)

12.2.1.2 RSSL Request Message Flag Enumeration Values

FLAG ENUMERATION	MEANING
RSSL_RQMF_STREAMING	<p>Indicates whether the request is for streaming data.</p> <ul style="list-style-type: none"> If present, the OMM consumer wants to continue to receive changes to information after the initial refresh is complete. If absent, the OMM consumer wants to receive only the refresh, after which the OMM Provider should close the stream. Such a request is typically referred to as a <i>non-streaming</i> or <i>snapshot</i> data request. <p>Because a refresh can be split into multiple parts, it is possible for updates to occur between the first and last part of the refresh, even as part of a non-streaming request.</p> <p>For more information about multi-part message handling, refer to Section 13.1.</p>
RSSL_RQMF_NO_REFRESH	<p>Indicates that the consumer application does not require a refresh for this request.</p> <p>This typically occurs after an initial request handshake is completed, usually to change stream attributes (e.g., priority). In some instances, a provider might still deliver a refresh message (but if the consumer does not explicitly ask for it, the message is unsolicited).</p>
RSSL_RQMF_PAUSE	<p>Indicates that the consumer would like to pause the stream, though this does not guarantee that the stream will pause.</p> <p>To resume data flow, the consumer must send a subsequent request message with the RSSL_RQMF_STREAMING flag set.</p> <p>For more information, refer to Section 13.6.</p>
RSSL_RQMF_HAS_PRIORITY	<p>Indicates the presence of priority information via the priorityClass and priorityCount members.</p> <p>For more information about using priority, refer to Section 13.2.</p>
RSSL_RQMF_HAS_QOS	<p>Indicates the presence of the qos member.</p> <ul style="list-style-type: none"> For more information, refer to Section 12.2.1.1 and Section 11.2.5. For specific handling information, refer to Section 13.3.
RSSL_RQMF_HAS_WORST_QOS	<p>Indicates the presence of the worstQos member.</p> <ul style="list-style-type: none"> For more information, refer to Section 12.2.1.1 and Section 11.2.5. For specific handling information, refer to Section 13.3.
RSSL_RQMF_HAS_VIEW	<p>Indicates that the request message payload might contain a dynamic view, specifying information the application wishes to receive (or that the application wishes to continue receiving a previously specified view). If this flag is not present, any previously specified view is discarded and a full view is provided.</p> <p>For more information about using dynamic views, refer to Section 13.8.</p>
RSSL_RQMF_HAS_BATCH	<p>Indicates that the request message payload contains a list of items of interest, all with matching msgKey information.</p> <p>For more information on using batch requests, refer to Section 13.7.</p>

Table 161: Rssl RequestMsg Flags

FLAG ENUMERATION	MEANING
RSSL_RQMF_HAS_EXTENDED_HEADER	Indicates that the <code>extendedHeader</code> member is present. Information in the <code>extendedHeader</code> is defined outside of the scope of the Transport API.
RSSL_RQMF_MSG_KEY_IN_UPDATES	Indicates that the consumer wants to receive the full <code>msgKey</code> in update messages. This flag does not guarantee that the <code>msgKey</code> is present in an update message. Instead, the provider application determines whether this information is present (the consumer should be written to handle either the presence or absence of <code>msgKey</code> in any <code>RsslUpdateMsg</code>). When specified on a request to ADS, the ADS fulfills the request.
RSSL_RQMF_CONF_INFO_IN_UPDATES	Indicates that the consumer wants to receive conflation information in update messages delivered on this stream. This flag does not guarantee that conflation information is present in update messages. Instead, the provider application determines whether this information is present (the consumer should be capable of handling conflation information in any <code>RsslUpdateMsg</code>). For details about conflation information on update messages, refer to Section 12.2.3.
RSSL_RQMF_PRIVATE_STREAM	Requests that the stream be opened as private. For details, refer to Section 13.12.

Table 161: `Rssl RequestMsg` Flags (Continued)

12.2.1.3 RSSL Request Message Utility Functions

The Transport API provides the following utility functions for use with the `RsslRequestMsg`:

FUNCTION NAME	DESCRIPTION
<code>rsslClearRequestMsg</code>	Clears an <code>RsslRequestMsg</code> structure. Useful for structure reuse.
<code>rsslSetStreamingFlag</code>	Sets the <code>RSSL_RQMF_STREAMING</code> flag on an already encoded <code>RsslRequestMsg</code> .
<code>rsslUnsetStreamingFlag</code>	Removes the <code>RSSL_RQMF_STREAMING</code> flag on an already encoded <code>RsslRequestMsg</code> .
<code>rsslSetNoRefreshFlag</code>	Sets the <code>RSSL_RQMF_NO_REFRESH</code> flag on an already encoded <code>RsslRequestMsg</code> .
<code>rsslUnsetNoRefreshFlag</code>	Removes the <code>RSSL_RQMF_NO_REFRESH</code> flag on an already encoded <code>RsslRequestMsg</code> .
<code>rsslSetMsgKeyInUpdatesFlag</code>	Sets the <code>RSSL_RQMF_MSG_KEY_IN_UPDATES</code> flag on an already encoded <code>RsslRequestMsg</code> .
<code>rsslUnsetMsgKeyInUpdatesFlag</code>	Removes the <code>RSSL_RQMF_MSG_KEY_IN_UPDATES</code> flag on an already encoded <code>RsslRequestMsg</code> .
<code>rsslSetConflInfoInUpdatesFlag</code>	Sets the <code>RSSL_RQMF_CONF_INFO_IN_UPDATES</code> flag on an already encoded <code>RsslRequestMsg</code> .
<code>rsslUnsetConflInfoInUpdatesFlag</code>	Removes the <code>RSSL_RQMF_CONF_INFO_IN_UPDATES</code> flag on an already encoded <code>RsslRequestMsg</code> .

Table 162: `Rssl RequestMsg` Utility Functions

12.2.2 RSSL Refresh Message Class

RsslRefreshMsg is often provided as an initial response or when an upstream source requires a data resynchronization point. An **RsslRefreshMsg** contains payload information along with state, QoS, permissioning, and group information.

- If provided as a response to an **RsslRequestMsg**, the refresh is a **solicited refresh**. Typically, solicited refresh messages are delivered only to the requesting consumer application
- If some kind of information change occurs (e.g., some kind of error is detected on a stream), an upstream provider can push out an **RsslRefreshMsg** to downstream consumers. This type of refresh is an **unsolicited refresh**. Typically, unsolicited refresh messages are delivered to all consumers using each consumer's respective stream.

When an OMM Interactive Provider sends an **RsslRefreshMsg**, the **streamId** should match the **streamId** on the corresponding **RsslRequestMsg**. The **msgKey** should be populated with the appropriate stream identifying information, and often matches the **msgKey** of the request. When an OMM NIP sends an **RsslRefreshMsg**, the provider should assign a negative **streamId** (when establishing a new stream, the **streamId** should be unique). In this scenario, the **msgKey** should define the information that the stream provides.

Using **RsslRefreshMsg**, an application can fragment the contents of a message payload and deliver the content across multiple messages, with the final message indicating that the refresh is complete. This is useful when providing large sets of content that may require multiple cache look-ups or be too large for an underlying transport layer. Additionally, an application receiving multiple parts of a response can potentially begin processing received portions of data before all content has been received. For more details on multi-part message handling, refer to Section 13.1.

12.2.2.1 RSSL Refresh Message Structure Members

Structure Member	DESCRIPTION
msgBase	Specifies the members common to all messages. An RsslRefreshMsg can optionally contain msgKey information. For details, refer to Section 12.1.1.
flags	A combination of bit values that indicate special behaviors and the presence of optional RsslRefreshMsg content. For more information about flag values, refer to Section 12.2.2.2.
partNum	Sets the part number of this refresh. partNum can contain values ranging from 0 to 32,767 where a value of 0 indicates the initial part of a refresh. <ul style="list-style-type: none"> • On multi-part refresh messages, partNum should start at 0 (to indicate the initial part) and increment by 1 for each subsequent message in the multi-part message. • If sent on a single-part refresh, a partNum of 0 should be used.
seqNum	A user-defined sequence number, which allows for values ranging from 0 to 4,294,967,295. seqNum should typically increase to help with temporal ordering, but may have gaps depending on the sequencing algorithm in use. Details about sequence number use should be defined within the domain model specification or any documentation for products which require the use of seqNum .
state	Conveys stream and data state information, which can change over time via subsequent refresh, status messages, or group status notifications. <ul style="list-style-type: none"> • For details about state information, refer to Section 11.2.6. • For a decision table that provides example behavior for various state combinations, refer to Appendix A.

Table 163: **Rssl RefreshMsg** Structure Members

Structure Member	DESCRIPTION
qos	<p>The concrete QoS of the stream. If a range was requested by the RsslRequestMsg, the qos should fall somewhere in this range, otherwise qos should exactly match what was requested.</p> <ul style="list-style-type: none"> • For more details on QoS, refer to Section 11.2.5. • For specific handling information, refer to Section 13.3.
permData	<p>Optional.</p> <p>Specifies authorization information for this stream. permData has a maximum allowed length of 32,767 bytes.</p> <p>When permData is specified on an RsslRefreshMsg, this indicates authorization information for all content on the stream, unless additional permission information is provided with specific content (e.g., RsslMapEntry.permData).</p> <p>For more information, refer to Section 11.4.</p>
groupId	<p>An RsslBuffer containing information about the item group to which this stream belongs. The groupId RsslBuffer has a maximum allowed length of 255 bytes.</p> <p>You can change the associated groupId via a subsequent RsslStatusMsg or RsslRefreshMsg. Group status notifications can change the state of an entire group of items.</p> <p>For more information about item groups, refer to Section 13.4.</p>
postUserInfo	<p>Optional.</p> <p>Contains information that identifies the user posting this information. If present on an RsslRefreshMsg, this implies that the refresh was posted to the system by the user described in postUserInfo.</p> <ul style="list-style-type: none"> • For more information about posting, refer to Section 13.9. • For more information about the Visible Publisher Identifier (VPI), refer to Section 13.10.
extendedHeader	<p>Available for domain-specific user-specified header information. The domain model specification defines contents and formatting. This information is not used in determining stream uniqueness, and may not pass through all components.</p> <p>To determine support, see appropriate component documentation.</p>
reqMsgKey	<p>Houses various attributes about requested item data. Used to identify data on client multicast networks.</p> <p>This does not need to be set by Interactive or Non-Interactive Provider applications.</p>

Table 163: [Rssl RefreshMsg](#) Structure Members (Continued)

12.2.2.2 RSSL Refresh Message Flag Enumeration Values

FLAG ENUMERATION	MEANING
RSSL_RFMF_REFRESH_COMPLETE	<p>Indicates that the message is the final part of the <code>Rss1RefreshMsg</code>. This flag value should be set when:</p> <ul style="list-style-type: none"> The message is a single-part refresh (i.e., atomic refresh). The message is the final part of a multi-part refresh. <p>For more information about multi-part message handling, refer to Section 13.1.</p>
RSSL_RFMF_SOLICITED	<p>Indicates that the refresh is sent as a response to a request, referred to as a solicited refresh.</p> <p>A refresh sent to inform a consumer of an upstream change in information (i.e., an unsolicited refresh) must not include this flag.</p>
RSSL_RFMF_DO_NOT_CACHE	Indicates that the message's payload information should not be cached. This flag value applies only to the message on which it is present.
RSSL_RFMF_CLEAR_CACHE	<p>Indicates that the stream's stored payload information should be cleared. This is typically set by providers when:</p> <ul style="list-style-type: none"> Sending the initial solicited <code>Rss1RefreshMsg</code>. Sending the first part of a multi-part <code>Rss1RefreshMsg</code>. Some portion of data is known to be invalid.
RSSL_RFMF_HAS_MSG_KEY	<p>Indicates that the <code>Rss1RefreshMsg</code> contains a populated <code>msgKey</code>. This can aid in associating a request with its corresponding refresh or identify an item sent from an NIP application.</p>
RSSL_RFMF_HAS_REQ_MSG_KEY	Indicates the presence of the <code>reqMsgKey</code> member.
RSSL_RFMF_HAS_QOS	<p>Indicates the presence of the <code>qos</code> member.</p> <p>For specific handling information, refer to Section 13.3.</p>
RSSL_RFMF_HAS_SEQ_NUM	Indicates the presence of the <code>seqNum</code> member.
RSSL_RFMF_HAS_PART_NUM	Indicates the presence of the <code>partNum</code> member.
RSSL_RFMF_HAS_PERM_DATA	Indicates the presence of the <code>permData</code> member.
RSSL_RFMF_HAS_POST_USER_INFO	Indicates that this message includes <code>postUserInfo</code> , implying that this <code>Rss1RefreshMsg</code> was posted by the user described in <code>postUserInfo</code> .
RSSL_RFMF_HAS_EXTENDED_HEADER	Indicates the presence of the <code>extendedHeader</code> member.
RSSL_RFMF_PRIVATE_STREAM	<p>Acknowledges the initial establishment of a private stream or, when combined with a <code>streamState</code> value of <code>RSSL_STREAM_REDIRECTED</code>, indicates that a stream can only be opened as private.</p> <p>For details, refer to Section 13.12.</p>

Table 164: `Rss1 RefreshMsg` Flags

12.2.2.3 RSSL Refresh Message Utility Functions

The Transport API provides the following utility functions for use with `RsslRefreshMsg`:

FUNCTION NAME	DESCRIPTION
<code>rsslClearRefreshMsg</code>	Clears an <code>RsslRefreshMsg</code> structure. Useful for structure reuse.
<code>rsslSetSolicitedFlag</code>	Sets the RSSL_RFMF_SOLICITED flag on an already encoded <code>RsslRefreshMsg</code> .
<code>rsslUnsetSolicitedFlag</code>	Removes the RSSL_RFMF_SOLICITED flag on an already encoded <code>RsslRefreshMsg</code> .
<code>rsslSetRefreshCompleteFlag</code>	Sets the RSSL_RFMF_REFRESH_COMPLETE flag on an already encoded <code>RsslRefreshMsg</code> .
<code>rsslUnsetRefreshCompleteFlag</code>	Removes the RSSL_RFMF_REFRESH_COMPLETE flag on an already encoded <code>RsslRefreshMsg</code> .

Table 165: `Rssl RefreshMsg` Utility Functions

12.2.3 RSSL Update Message Class

Providers (both interactive and non-interactive) use `RsslUpdateMsg` to convey changes to data associated with an item stream. When streaming, update messages typically flow after the delivery of an initial refresh. Update messages can be delivered between parts of a multi-part refresh message, even in response to a non-streaming request. For more information on multi-part message handling, refer to Section 13.1.

Some providers can aggregate the information from multiple update messages into a single update message using a technique called conflation. Conflation typically occurs if a conflated QoS is requested (refer to Section 11.2.5), a stream is paused (refer to Section 13.6), or if a consuming application is unable to keep up with a stream's data rates. If conflation is used, specific information can be provided with `RsslUpdateMsg` via optional conflation information.

12.2.3.1 RSSL Update Message Structure Members

Structure Member	DESCRIPTION
msgBase	Specifies the members common to all messages. An <code>RsslUpdateMsg</code> can optionally contain <code>msgKey</code> information. For details, refer to Section 12.1.1.
flags	A combination of bit values that indicate special behaviors and the presence of optional content. For more information about flag values, refer to Section 12.2.3.2.
updateType	Specifies the type of data in the <code>RsslUpdateMsg</code> , where values are typically defined in an enumeration (valid values range from 0 to 255). Examples of possible update types include: Trade , Quote , or Closing Run . <ul style="list-style-type: none"> Domain message model specifications define available update types. For Thomson Reuters's provided domain models, the <code>rssIRDM.h</code> header file defines available update types.
seqNum	Specifies a user-defined sequence number, which can range in value from 0 to 4,294,967,295. To help with temporal ordering, <code>seqNum</code> should increase across messages, but can have gaps depending on the sequencing algorithm in use. Details about sequence number use should be defined within the domain model specification or any documentation for products which require the use of <code>seqNum</code> .
conflationCount	When conflating data, this value indicates the number of updates conflated or aggregated into this <code>RsslUpdateMsg</code> . <code>conflationCount</code> allows for values ranging from 1 to 32,767.
conflationTime	When conflating data, this value indicates the period of time over which individual updates were conflated or aggregated into this <code>RsslUpdateMsg</code> (typically in milliseconds; for further details, refer to specific component documentation). <code>conflationTime</code> allows for values ranging from 1 to 65,535.
permData	Optional. Specifies authorization information for this stream. When specified, <code>permData</code> indicates authorization information for only the content within this message, though this can be overridden for specific content within the message (e.g., <code>RsslMapEntry.permData</code>). <code>permData</code> has a maximum allowed length of 32,767 bytes. For more information, refer to Section 11.4.

Table 166: `RsslUpdateMsg` Structure Members

Structure Member	DESCRIPTION
postUserInfo	<p>Optional. Identifies the user that posted this information.</p> <ul style="list-style-type: none"> For more information about posting, refer to Section 13.9. For more information about the Visible Publisher Identifier, refer to Section 13.10.
extendedHeader	<p>Available for domain-specific user-specified header information. The domain model specification defines the contents and formatting. extendedHeader information is not used to determine stream uniqueness, and might not pass through all components. To determine support, refer to the appropriate component documentation.</p>

Table 166: **Rssi UpdateMsg** Structure Members (Continued)

12.2.3.2 RSSL Update Message Flag Enumeration Values

FLAG ENUMERATION	MEANING
RSSL_UPMF_DO_NOT_CACHE	Indicates that payload information associated with this message should not be cached. RSSL_UPMF_DO_NOT_CACHE applies only to the message on which it is present.
RSSL_UPMF_DO_NOT_CONFLATE	Indicates that this message should not be conflated. This flag value only applies to the message on which it is present.
RSSL_UPMF_DO_NOT_RIPPLE	Indicates that the contents of this message should not be rippled. Rippling is typically associated with an Rss1FieldList . For additional information, refer to Section 11.3.1.5.
RSSL_UPMF_HAS_MSG_KEY	Indicates that the RsslUpdateMsg contains a populated msgKey . The additional key information can help associate a request with updates or identify an item being sent from an NIP application. This information is typically not necessary in an RsslUpdateMsg as the streamId can be used to determine the same information with less bandwidth cost.
RSSL_UPMF_HAS_SEQ_NUM	Indicates the presence of the seqNum member.
RSSL_UPMF_HAS_CONF_INFO	Indicates the presence of conflationTime and conflationCount information.
RSSL_UPMF_HAS_PERM_DATA	Indicates the presence of the permData member.
RSSL_UPMF_HAS_POST_USER_INFO	Indicates that this message includes postUserInfo , implying that this RsslUpdateMsg was posted by the user described in the postUserInfo .
RSSL_UPMF_HAS_EXTENDED_HEADER	Indicates the presence of the extendedHeader member.

Table 167: **RsslUpdateMsg** Flags

12.2.3.3 RSSL Update Message Utility Function

The Transport API provides the following utility function for use with the **RsslUpdateMsg**:

FUNCTION NAME	DESCRIPTION
<code>rsslClearUpdateMsg</code>	Clears an RsslUpdateMsg structure. Useful for structure reuse.

Table 168: **RsslUpdateMsg** Utility Functions

12.2.4 RSSL Status Message Class

An **RsslStatusMsg** can convey changes in **streamstate** or **datastate** (refer to Section 11.2.6), changes in a stream's permissioning information (refer to Section 9.4), or changes to the item group of which the stream is a part (refer to Section 13.4). A Provider application uses **RsslStatusMsg** to close streams to a consumer, in conjunction with an initial request or later after the stream has been established. An **RsslStatusMsg** can also indicate the successful establishment of a stream, though the message might not contain data (useful in establishing a stream solely to exchange bi-directional **RsslGenericMsgs**).

12.2.4.1 RSSL Status Message Structure Members

Structure Member	DESCRIPTION
msgBase	Specifies the members common to all messages. Optionally, an RsslStatusMsg can contain msgKey information. For details, refer to Section 12.1.1.
flags	Specifies a combination of bit values indicating special behaviors and the presence of optional content. For more information about flag values, refer to Section 12.2.4.2.
state	Conveys stream and data state information, which can change over time via subsequent refresh or status messages or group status notifications. <ul style="list-style-type: none"> For details about state information, refer to Section 11.2.6. For a decision table that provides example behavior for various state combinations, refer to Appendix A.
permData	Optional. When specified on an RsslStatusMsg , permData indicates authorization information for this stream, unless additional permission information is provided with specific content (e.g., RsslMapEntry.permData). permData allows a maximum length of 32,767 bytes. For more information, refer to Section 11.4.
groupId	An RsslBuffer with a maximum allowed length of 255 bytes that contains information about the item group to which this stream belongs. A subsequent RsslStatusMsg or RsslRefreshMsg can change the item group's associated groupId , while group status notifications can change the state of an entire group of items. For more information about item groups, refer to Section 13.4.
postUserInfo	Optional. Identifies the user who posted this information. <ul style="list-style-type: none"> For more information about posting, refer to Section 13.9. For more information about Visible Publisher Identifier, refer to Section 13.10.
extendedHeader	Available for domain-specific user-specified header information. Contents and formatting are determined by the domain model specification. This header information is not used to determine stream uniqueness, and might not pass through all components. To determine support, see appropriate component documentation.
reqMsgKey	Houses various attributes about requested item data. Used to identify data on client multicast networks. This does not need to be set by Interactive or Non-Interactive Provider applications.

Table 169: **Rssl StatusMsg** Structure Members

12.2.4.2 RSSL Status Message Flag Enumeration Values

FLAG ENUMERATION	MEANING
RSSL_STMF_CLEAR_CACHE	Indicates that the application should clear stored header or payload information associated with the stream. This can happen if some portion of data is invalid.
RSSL_STMF_HAS_MSG_KEY	Indicates that the <code>RsslStatusMsg</code> contains a populated <code>msgKey</code> . The <code>msgKey</code> can be used to aid in associating a request to a status message or identify an item sent from an NIP application.
RSSL_STMF_HAS_REQ_MSG_KEY	Indicates presence of the <code>reqMsgKey</code> member.
RSSL_STMF_HAS_STATE	Indicates the presence of <code>state</code> information. If <code>state</code> information is not present, the message might be changing the stream's permission information or <code>groupId</code> .
RSSL_STMF_HAS_PERM_DATA	Indicates the presence of <code>permData</code> . When present, the message might be changing the stream's permission information.
RSSL_STMF_HAS_GROUP_ID	Indicates the presence of <code>groupId</code> . When present, the message might be changing the stream's <code>groupId</code> .
RSSL_STMF_HAS_POST_USER_INFO	Indicates the presence of <code>postUserInfo</code> , which identifies the user who posted the <code>RsslStatusMsg</code> .
RSSL_STMF_HAS_EXTENDED_HEADER	Indicates the presence of <code>extendedHeader</code> .
RSSL_STMF_PRIVATE_STREAM	Acknowledges the establishment of a private stream, or when combined with a <code>streamState</code> value of <code>RSSL_STREAM_REDIRECTED</code> , indicates that a stream can be opened only as private. For details, refer to Section 13.12.

Table 170: `Rssl StatusMsg` Flags

12.2.4.3 RSSL Status Message Utility Function

The Transport API provides the following utility function to aid in using `RsslStatusMsg`.

FUNCTION NAME	DESCRIPTION
<code>rsslClearStatusMsg</code>	Clears an <code>RsslStatusMsg</code> structure. Useful for structure reuse.

Table 171: `Rssl StatusMsg` Utility Functions

12.2.5 RSSL Close Message Class

A consumer uses `RsslCloseMsg` to indicate no further interest in an item stream and to close the stream. The `streamId` indicates the item stream to which `RsslCloseMsg` applies.

12.2.5.1 RSSL Close Message Structure Members

STRUCTURE MEMBER	DESCRIPTION
<code>msgBase</code>	Specifies the members common to all messages. An <code>RsslCloseMsg</code> does not contain any <code>msgKey</code> information. For details, refer to Section 12.1.1.
<code>flags</code>	Specifies a combination of bit values indicating special behaviors and the presence of optional content. For available flag values, refer to Table 173.
<code>extendedHeader</code>	Available for domain-specific user-specified header information. Contents and formatting are specified by a domain model specification. <code>extendedHeader</code> information does not determine stream uniqueness, and might not pass through all components. To determine support, see the appropriate component documentation.

Table 172: `RsslCloseMsg` Structure Members

12.2.5.2 RSSL Close Message Flag Enumeration Values

FLAG ENUMERATION	MEANING
<code>RSSL_CLMF_ACK</code>	If present, the consumer wants the provider to send an <code>RsslAckMsg</code> to indicate that the <code>RsslCloseMsg</code> has been processed properly and the stream is properly closed. This functionality might not be available with some components; for details, refer to the component's documentation.
<code>RSSL_CLMF_HAS_EXTENDED_HEADER</code>	Indicates the presence of <code>extendedHeader</code> .

Table 173: `RsslCloseMsg` Flags

12.2.5.3 RSSL Close Message Utility Functions

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

FUNCTION NAME	DESCRIPTION
<code>rsslClearCloseMsg</code>	Clears an <code>RsslCloseMsg</code> structure. Useful for structure reuse.

Table 174: `RsslCloseMsg` Utility Functions

12.2.6 RSSL Generic Message Class

RsslGenericMsg is a bi-directional message without any implicit interaction semantics associated with it, hence the name generic. After a stream is established via a request-refresh/status interaction, both consumers and providers can send **RsslGenericMsgs** to one another, and NIP applications can leverage them. Generic messages are transient and typically not cached by Enterprise Platform components.

The **msgKey** of an **RsslGenericMsg** does not need to match the **msgKey** information of the stream over which the generic message flows. Thus, key information can be used independently within the stream. A domain message model specification typically defines any specific message usage, **msgKey** usage, expected interactions, and handling instructions.

12.2.6.1 RSSL Generic Message Structure Members

Structure Member	DESCRIPTION
msgBase	<p>Specifies the members common to all messages. An RsslGenericMsg can optionally contain msgKey information.</p> <p>For details, refer to Section 12.1.1.</p>
flags	<p>Specifies a combination of bit values that indicate special behaviors and the presence of optional content.</p> <p>For more information about flag values, refer to Section 12.2.6.2.</p>
partNum	<p>Specifies the part number of this generic message, typically used with multi-part generic messages. partNum can contain values ranging from 0 to 32,767.</p> <ul style="list-style-type: none"> If sent on a single-part post message, use a partNum of 0. On multi-part post messages, use a partNum of 0 on the initial part and increment partNum in each subsequent part by 1.
seqNum	<p>Specifies a user-defined sequence number ranging in value from 0 to 4,294,967,295. A seqNum typically corresponds to the sequencing of this message.</p> <p>To help with temporal ordering, seqNum should increase across messages, but can have gaps depending on the sequencing algorithm in use. Details about using seqNum should be defined in the domain model specification or the documentation for products that must use seqNum.</p>
secondarySeqNum	<p>Specifies an additional user-defined sequence number ranging in value from 0 to 4,294,967,295. When using RsslGenericMsg on a stream in a bi-directional manner, secondarySeqNum is often used as an acknowledgment sequence number.</p> <p>For example, a consumer sends a generic message with seqNum populated to indicate the sequence of this message in the stream and secondarySeqNum set to the seqNum last received from the provider. This effectively acknowledges all messages received up to that point while still sending additional information.</p> <p>Sequence number use should be defined within the domain model specification or any documentation for products that use secondarySeqNum.</p>
permData	<p>Optional. Indicates authorization information for content within this message only, though this can be overridden for specific content within the message (e.g. RsslMapEntry.permData).</p> <p>permData allows a maximum length of 32,767 bytes.</p> <p>For more information, refer to Section 11.4.</p>

Table 175: **RsslGenericMsg** Structure Members

Structure Member	DESCRIPTION
extendedHeader	Available for domain-specific user-specified header information. The domain model specification defines content and formatting. extendedHeader does not determine stream uniqueness, and might not pass through all components. To determine support, refer to the appropriate component documentation.
reqMsgKey	Houses various attributes about requested item data. Used to identify data on client multicast networks. This does not need to be set by Interactive or Non-Interactive Provider applications.

Table 175: **Rssl Generic cMsg** Structure Members (Continued)

12.2.6.2 RSSL Generic Message Flag Enumeration Values

FLAG ENUMERATION	MEANING
RSSL_GNMF_MESSAGE_COMPLETE	When set, this flag indicates that the message is the final part of an RsslGenericMsg . This flag should be set on: <ul style="list-style-type: none"> Single-part generic messages (i.e., an atomic generic message). The last message (final part) in a multi-part generic message. For more information on handling multi-part messages, refer to Section 13.1.
RSSL_GNMF_HAS_MSG_KEY	Indicates the presence of a populated msgKey . Use of a msgKey differentiates a generic message from the msgKey information specified for other messages within the stream. Contents and semantics associated with an RsslGenericMsg.msgKey should be defined by the domain model specification that employs them.
RSSL_GNMF_HAS_REQ_MSG_KEY	Indicates the presence of the reqMsgKey member.
RSSL_GNMF_HAS_SEQ_NUM	Indicates the presence of the seqNum member.
RSSL_GNMF_HAS_SECONDARY_SEQ_NUM	Indicates the presence of the secondarySeqNum member.
RSSL_GNMF_HAS_PART_NUM	Indicates the presence of the partNum member.
RSSL_GNMF_HAS_PERM_DATA	Indicates the presence of the permData member.
RSSL_GNMF_HAS_EXTENDED_HEADER	Indicates presence of the extendedHeader member.

Table 176: **Rssl Generic cMsg** Flags

12.2.6.3 RSSL Generic Message Utility Function

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

FUNCTION NAME	DESCRIPTION
rsslClearGenericMsg	Clears an RsslGenericMsg structure. Useful for structure reuse.
rsslSetGenericCompleteFlag	Sets the RSSL_GNMF_MESSAGE_COMPLETE flag on an already encoded RsslGenericMsg .

Table 177: **Rssl Generic cMsg** Utility Functions

FUNCTION NAME	DESCRIPTION
rsslUnsetGenericCompleteFlag	Removes the RSSL_GNMF_MESSAGE_COMPLETE flag on an already encoded RsslGenericMsg .

Table 177: **Rssl Generic cMsg** Utility Functions (Continued)

12.2.7 RSSL Post Message Class

A consumer application uses **RsslPostMsg** to push content to upstream components. Such content can be applied to a TREP cache or routed further upstream to the source of data. After upstream components receive the content, the components can republish the data to their downstream consumers.

Post messages can be routed along a specific item stream, referred to as **on-stream** posting, or along a user's Login stream, referred to as **off-stream** posting. **RsslPostMsg** can contain any Transport API container type, including other messages. User identification information can be associated with a post message and be provided along with posted content. For more details, refer to Section 13.9.

12.2.7.1 RSSL Post Message Structure Members

Structure Member	DESCRIPTION
msgBase	Specifies the members common to all messages. An RsslPostMsg can optionally contain msgKey information. For details, refer to Section 12.1.1.
flags	Specifies a combination of bit values that indicate special behaviors and the presence of optional content. For more information about flag values, refer to Section 12.2.7.2.
partNum	Specifies the part number for this post message, typically used with multi-part post messages. partNum can contain values ranging from 0 to 32,767. <ul style="list-style-type: none"> • If sent on a single-part post message, use a partNum of 0. • On multi-part post messages, use a partNum of 0 on the initial part and in each subsequent part, increment partNum part by 1.
postId	Specifies a consumer-assigned identifier, which can range in value from 0 to 4,294,967,295. postId distinguishes different post messages. In multi-part post messages, each part must use the same postId value.
seqNum	Specifies a user-defined sequence number, typically corresponding to the sequencing of the message. seqNum allows for values ranging from 0 to 4,294,967,295. To help with temporal ordering, seqNum should increase, though gaps might exist depending on the sequencing algorithm in use. Details about seqNum use should be defined in the domain model specification or any documentation for products that use seqNum . When acknowledgments are requested, the seqNum will be provided back in the RsslAckMsg to help identify the RsslPostMsg being acknowledged.

Table 178: **Rssl PostMsg** Structure Members

Structure Member	DESCRIPTION
permData	<p>Optional. When present, <code>permData</code> indicates authorization information for content in this message only. <code>permData</code> can be overridden for specific content within the message (e.g. <code>RsslMapEntry.permData</code>).</p> <p><code>permData</code> allows a maximum length of 32,767 bytes.</p> <p>For more information, refer to Section 11.4.</p>
postUserInfo	<p>Identifies the posting user. <code>postUserInfo</code> can optionally be provided along with posted content via a <code>RsslRefreshMsg</code>, <code>RsslUpdateMsg</code>, and <code>RsslStatusMsg</code>.</p> <ul style="list-style-type: none"> • For more information about posting, refer to Section 13.9. • For more information about Visible Publisher Identifier, refer to Section 13.10.
postUserRights	<p>Conveys the rights or abilities of the user posting this content, which can indicate whether the user is permissioned to:</p> <ul style="list-style-type: none"> • Create items in the cache of record, • Delete items from the cache of record, or • Modify the <code>permData</code> on items already present in the cache of record. <p>For details about different rights, refer to Section 12.2.7.3.</p>
extendedHeader	<p>Available for domain-specific, user-specified header information. The domain model specification defines the contents and formatting. The <code>extendedHeader</code> does not determine stream uniqueness and might not pass through all components.</p> <p>To determine support, see appropriate component documentation.</p>

Table 178: `Rssi PostMsg` Structure Members (Continued)

12.2.7.2 RSSL Post Message Flag Enumeration Values

FLAG ENUMERATION	MEANING
RSSL_PSMF_POST_COMPLETE	<p>Indicates that this is the final part of the <code>RssiPostMsg</code>. This flag should be set on:</p> <ul style="list-style-type: none"> • Single-part post messages (i.e., an atomic post message). • The final part of a multi-part post message. <p>For more information about multi-part message handling, refer to Section 13.1.</p>
RSSL_PSMF_ACK	<p>Specifies that the consumer wants the provider to send an <code>RsslAckMsg</code> to indicate that the <code>RssiPostMsg</code> was processed properly. When acknowledging an <code>RssiPostMsg</code>, the provider must include the <code>postId</code> in the <code>ackId</code> and communicate any associated <code>seqNum</code>.</p>
RSSL_PSMF_HAS_MSG_KEY	<p>Indicates that the <code>RssiPostMsg</code> contains a populated <code>msgKey</code> that identifies the stream on which the information is posted. A <code>msgKey</code> is typically required for off-stream posting and is not necessary when on-stream posting.</p> <p>For more detailed information about posting, refer to Section 13.9.</p>
RSSL_PSMF_HAS_SEQ_NUM	Indicates the presence of the <code>seqNum</code> member.
RSSL_PSMF_HAS_POST_ID	Indicates the presence of the <code>postId</code> member.

Table 179: `Rssi PostMsg` Flags

FLAG ENUMERATION	MEANING
RSSL_PSMF_HAS_POST_USER_RIGHTS	Indicates the presence of the <code>postUserRights</code> member.
RSSL_PSMF_HAS_PART_NUM	Indicates the presence of the <code>partNum</code> member.
RSSL_PSMF_HAS_PERM_DATA	Indicates the presence of the <code>permData</code> member.
RSSL_PSMF_HAS_EXTENDED_HEADER	Indicates the presence of the <code>extendedHeader</code> member.

Table 179: `Rssl PostMsg` Flags (Continued)

12.2.7.3 RSSL Post User Rights Flag Enumeration Values

FLAG ENUMERATION	MEANING
RSSL_PSUR_NONE	The user has no additional posting abilities.
RSSL_PSUR_CREATE	The user is allowed to create items in the cache of record.
RSSL_PSUR_DELETE	The user is allowed to remove items from the cache of record.
RSSL_PSUR MODIFY_PERM	The user is allowed to modify the <code>permData</code> associated with items already in the cache of record.

Table 180: `Rssl PostRights` Flags

12.2.7.4 RSSL Post Message Utility Function

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

FUNCTION NAME	DESCRIPTION
<code>rsslClearPostMsg</code>	Clears an <code>RsslPostMsg</code> structure. Useful for structure reuse.

Table 181: `Rssl PostMsg` Utility Functions

12.2.8 RSSL Acknowledgment Message Class

A provider can send an `RsslAckMsg` to a consumer to indicate receipt of a specific message. The acknowledgment carries success or failure (i.e., a negative acknowledgment or 'NAK') information to the consumer. Currently, a consumer can request acknowledgment for an `RsslPostMsg` or `RsslCloseMsg`.

12.2.8.1 RSSL Acknowledgment Message Structure Members

Structure Member	DESCRIPTION
msgBase	The common message base members. An <code>RsslAckMsg</code> can optionally contain <code>msgKey</code> information. For details, refer to Section 12.1.1.
flags	Specifies a combination of bit values indicating special behaviors and the presence of optional content. For more information about flag values, refer to Section 12.2.8.2.
ackId	Associates the <code>RsslAckMsg</code> with the message it acknowledges. <code>ackId</code> allows for values ranging from 0 to 4,294,967,295. When acknowledging an <code>RsslPostMsg</code> , <code>ackId</code> typically matches the post message's <code>postID</code> .
seqNum	Specifies a user-defined sequence number, ranging in value from 0 to 4,294,967,295. To help with temporal ordering, <code>seqNum</code> should increase, though gaps might exist depending on the sequencing algorithm in use. The acknowledgment message may populate this with the <code>seqNum</code> of the <code>RsslPostMsg</code> being acknowledged. This helps correlate the message being acknowledged when the <code>postID</code> alone is not sufficient (e.g., multi-part post messages).
nakCode	If present, this message indicates a NAK. The <code>nakCode</code> is an enumerated code value (ranging in value from 1 to 255) that provides additional information about the reason for the NAK. <code>nakCode</code> values are defined in Section 12.2.8.3
text	Optional. Provides additional information about the acceptance or rejection of the message being acknowledged. <code>text</code> has a maximum allowed length of 65,535 bytes.
extendedHeader	Available for domain-specific user-specified header information. The domain model specification defines contents and formatting. <code>extendedHeader</code> does not determine stream uniqueness and might not pass through all components. To determine support, refer to the appropriate component documentation.

Table 182: `RsslAckMsg` Structure Members

12.2.8.2 RSSL Acknowledgment Message Flag Enumeration Values

FLAG ENUMERATION	MEANING
RSSL_AKMF_HAS_MSG_KEY	Indicates the presence of a populated <code>msgKey</code> . When present, this is typically populated to match the information being acknowledged.
RSSL_AKMF_HAS_SEQ_NUM	Indicates the presence of the <code>seqNum</code> member.
RSSL_AKMF_HAS_NAK_CODE	Indicates the presence of the <code>nakCode</code> member.
RSSL_AKMF_HAS_TEXT	Indicates the presence of the <code>text</code> member.
RSSL_AKMF_HAS_EXTENDED_HEADER	Indicates presence of the <code>extendedHeader</code> member.
RSSL_AKMF_PRIVATE_STREAM	Acknowledges the initial establishment of a private stream. For details, refer to Section 13.12.

Table 183: Rssl AckMsg Flags

12.2.8.3 RSSL Acknowledgment Message Enumerated Names

ENUMERATED NAME	DESCRIPTION
RSSL_NAKC_ACCESS_DENIED	The user is not permissioned to post on the item or service.
RSSL_NAKC_DENIED_BY_SRC	The source being posted to has denied accepting this post message.
RSSL_NAKC_SOURCE_DOWN	The source being posted to is down or unavailable.
RSSL_NAKC_SOURCE_UNKNOWN	The source being posted to is unknown and unreachable.
RSSL_NAKC_NO_RESOURCES	Some component along the path of the post message does not have appropriate resources available to continue processing the post.
RSSL_NAKC_NO_RESPONSE	There is no response from the source being posted to. This may mean that the source is unavailable or that there is a delay in processing the posted information.
RSSL_NAKC_GATEWAY_DOWN	A gateway device for handling posted or contributed information is down or unavailable.
RSSL_NAKC_SYMBOL_UNKNOWN	The system does not recognize the item information provided with the post message. This may be an invalid item.
RSSL_NAKC_NOT_OPEN	The item being posted to does not have an available stream.
RSSL_NAKC_INVALID_CONTENT	The content of the post message is invalid (it does not match the expected formatting) and cannot be posted.

Table 184: Rssl AckMsg NakCode Values

12.2.8.4 RSSL Acknowledgement Message Utility Function

The Transport API provides the following utility function for use with the [RsslAckMsg](#).

FUNCTION NAME	DESCRIPTION
rsslClearAckMsg	Clears an RsslAckMsg structure. Useful for structure reuse.

Table 185: [Rssl AckMsg](#) Utility Functions

12.2.9 The RSSL Message Union

The [RsslMsg](#) structure is a union of the various classes of Messages. For example:

```
typedef union {
    RsslMsgBase      msgBase; /* Common Message Base; refer to Section 12.1.1 */
    RsslRequestMsg  requestMsg; /* RSSL Request Message; refer to Section 12.2.1 */
    RsslAckMsg       ackMsg; /* RSSL Acknowledgement Message; refer to Section 12.2.8 */
    RsslRefreshMsg   refreshMsg; /* RSSL Refresh Message; refer to Section 12.2.2 */
    RsslStatusMsg    statusMsg; /* RSSL Status Message; refer to Section 12.2.4 */
    RsslUpdateMsg    updateMsg; /* RSSL Update Message; refer to Section 12.2.3 */
    RsslCloseMsg     closeMsg; /* RSSL Close Message; refer to Section 12.2.5 */
    RsslGenericMsg   genericMsg; /* RSSL Generic (Bidirectional) Message; refer to
                                  Section 12.2.6 */
    RsslPostMsg      postMsg; /* RSSL Post Message; refer to Section 12.2.7 */
} RsslMsg;
```

Code Example 40: [Rssl Msg](#) Union

12.2.9.1 RsslMsg Encoding Interfaces

All message encoding and decoding functions expect the [RsslMsg](#) type. Any specific message class can be cast to the [RsslMsg](#), and an [RsslMsg](#) can be cast to any specific message class. An [RsslMsg](#) can be encoded from pre-encoded data or by encoding individual pieces of data as they are provided.

ENCODE INTERFACE	DESCRIPTION
rsslEncodeMsg	<p>Encodes a message where all message content is pre-encoded.</p> <ul style="list-style-type: none"> • msgKey attribute information should be encoded and populated on msgKey.encAttrib prior to this call. • extendedHeader information should be encoded and populated on the message's extendedHeader member prior to this call. • Message payload information should be encoded and populated on the encDataBody member prior to this call.

Table 186: [Rssl Msg](#) Encode Functions

ENCODE INTERFACE	DESCRIPTION
rsslEncodeMsgInit	<p>Begins encoding of an <code>RsslMsg</code>.</p> <p>All message header elements should be properly populated. The <code>containerType</code> member should be populated with the specific type of message payload.</p> <ul style="list-style-type: none"> If encoding <code>msgKey</code> attribute information: pre-encoded <code>msgKey</code> attribute information should be populated in <code>msgKey.encAttrib</code>. Unencoded <code>msgKey</code> attribute information should be encoded after <code>rsslEncodeMsgInit</code> returns, followed by <code>rsslEncodeMsgKeyAttribComplete</code>. If encoding <code>extendedHeader</code> information: pre-encoded <code>extendedHeader</code> information should be populated in the <code>extendedHeader</code> member of the message. Unencoded <code>extendedHeader</code> information should be encoded after the call to <code>rsslEncodeMsgInit</code> and after <code>msgKey</code> attribute information is encoded. When <code>extendedHeader</code> encoding is completed, call <code>rsslEncodeExtendedHeaderComplete</code>.
rsslEncodeMsgComplete	<p>Completes encoding of an <code>RsslMsg</code>.</p> <p>All message content should be encoded prior to this call. This function expects the same <code>RsslEncodeIterator</code> that was used with <code>rsslEncodeMsgInit</code>.</p> <ul style="list-style-type: none"> If the content (i.e., payload, <code>msgKey</code> attrib, and <code>extendedHeader</code>) encodes successfully, the <code>RsslBool success</code> parameter should be set to <code>true</code> to finish encoding. If any of the content fails to encode, the <code>RsslBool success</code> parameter should be set to <code>false</code> to roll back the encoding of the message.
rsslEncodeMsgKeyAttribComplete	<p>Completes encoding of any non-pre-encoded <code>msgKey</code> attribute information.</p> <p>Can be used only when message encoding leverages <code>rsslEncodeMsgInit</code>. If the <code>RSSL_MKF_HAS_ATTRIB</code> flag is set and <code>msgKey.encAttrib</code> is not populated, <code>msgKey</code> attribute information is expected after <code>rsslEncodeMsgInit</code> returns, with the specific <code>attribContainerType</code> functions being used to encode it. This function expects the same <code>RsslEncodeIterator</code> used with <code>rsslEncodeMsgInit</code>.</p> <ul style="list-style-type: none"> If encoding of the <code>msgKey</code> attribute information succeeds, the <code>RsslBool success</code> parameter should be set to <code>true</code> to finish attribute encoding. If encoding of attributes fails, the <code>RsslBool success</code> parameter should be set to <code>false</code> to roll back encoding prior to <code>msgKey</code> attributes. <p>If both <code>msgKey</code> attributes and <code>extendedHeader</code> information are being encoded, <code>msgKey</code> attributes are expected first with <code>extendedHeader</code> being encoded after the call to <code>rsslEncodeMsgKeyAttribComplete</code>.</p>

Table 186: `RsslMsg` Encode Functions (Continued)

Encode Interface	Description
rsslEncodeExtendedHeaderComplete	<p>Completes encoding of any non-pre-encoded <code>extendedHeader</code> information. Can be used only when the message encoding leverages <code>rsslEncodeMsgInit</code>. If the specific message's <code>HAS_EXTENDED_HEADER</code> flag is set and <code>extendedHeader</code> is not populated, this information is expected after <code>rsslEncodeMsgInit</code> (and <code>rsslEncodeMsgKeyAttribComplete</code> if encoding <code>msgKey</code> attributes) returns. This function expects the same <code>RsslEncodeIterator</code> used with previous message encoding functions.</p> <ul style="list-style-type: none"> If encoding of <code>extendedHeader</code> succeeds, the <code>RsslBool success</code> parameter should be set to <code>true</code> to finish encoding. If encoding of <code>extendedHeader</code> fails, the <code>RsslBool success</code> parameter should be set to <code>false</code> to roll back to encoding prior to <code>extendedHeader</code>. <p>If both <code>msgKey</code> attributes and <code>extendedHeader</code> information are being encoded, <code>msgKey</code> attributes are expected first, while <code>extendedHeader</code> should be encoded after the call to <code>rsslEncodeMsgKeyAttribComplete</code>.</p>

Table 186: Rssl Msg Encode Functions (Continued)

12.2.9.2 RsslMsg Encoding Example 1

The following code sample demonstrates `RsslMsg` encoding, showing the use of `rsslEncodeMsgInit` with `rsslEncodeMsgComplete` and includes unencoded `msgKey` attribute information, unencoded payload, and unencoded `extendedHeader` information. While this example demonstrates error handling for the initial encode function, it omits additional error handling to simplify the example (though it should still be performed).

```

/* EXAMPLE 1 - EncodeMsgInit/Complete with unencoded msgKey attribute, payload, and
/* extendedHeader */

/* Populate and encode a requestMsg */
RsslRequestMsg reqMsg = RSSL_INIT_REQUEST_MSG;
reqMsg.msgBase.msgClass = RSSL_MC_REQUEST; /* message is a request */
reqMsg.msgBase.domainType = RSSL_DMT_MARKET_PRICE;
reqMsg.msgBase.containerType = RSSL_DT_ELEMENT_LIST;
/* Choose a stream Id that is not in use if this is a new request, otherwise reuse associated
/* id */
reqMsg.msgBase.streamId = 6;
/* Populate flags for request message members and behavior - our message is for a streaming
/* request, will specify a quality of service range, priority, contains an extended header and
/* payload is a dynamic view request */
reqMsg.flags = RSSL_RQMF_STREAMING | RSSL_RQMF_HAS_PRIORITY | RSSL_RQMF_HAS_QOS |
    RSSL_RQMF_HAS_WORST_QOS | RSSL_RQMF_HAS_EXTENDED_HEADER | RSSL_RQMF_HAS_VIEW;

/* Populate qos range and priority */
reqMsg.priorityClass = 2;
reqMsg.priorityCount = 1;
/* Populate best qos allowed */
reqMsg.qos.rate = RSSL_QOS_RATE_TICK_BY_TICK;
reqMsg.qos.timeliness = RSSL_QOS_TIME_REALTIME;

```

```

/* Populate worst qos allowed, rate and timeliness values allow for rateInfo and timeInfo to
/* be sent */
reqMsg.worstQos.rate = RSSL_QOS_RATE_TIME_CONFLATED;
reqMsg.worstQos.rateInfo = 1500;
reqMsg.worstQos.timeliness = RSSL_QOS_TIME_DELAYED;
reqMsg.worstQos.timeInfo = 20;

/* Populate msgKey to specify a serviceId, a name with type of RIC (which is default nameType)
/* and attrib */
reqMsg.msgBase.msgKey.flags = RSSL_MKF_HAS_SERVICE_ID | RSSL_MKF_HAS_NAME |
    RSSL_MKF_HAS_ATTRIB;
reqMsg.msgBase.msgKey.serviceId = 1;
/* Specify name and length of name. Because this is a RIC, no nameType is required. */
reqMsg.msgBase.msgKey.name.data = "TRI";
reqMsg.msgBase.msgKey.name.length = 3;
/* Msg Key attribute info will be encoded after rsslEncodeMsgInit returns */
reqMsg.msgBase.msgKey.attribContainerType = RSSL_DT_ELEMENT_LIST;

/* begin encoding of message - assumes that encIter is already populated with buffer and
/* version information, store return value to determine success or failure data max */
/* encoded size is unknown so 0 is used */
if ((RetVal = rsslEncodeMsgInit(&encIter, (RsslMsg*)&reqMsg, 0)) < RSSL_RET_SUCCESS)
{
    /* error condition - switch our success value to false so we can roll back */
    success = RSSL_FALSE;
    /* print out message with return value string, value, and text */
    printf("Error %s (%d) encountered with rsslEncodeMsgInit. Error Text: %s\n",
        rsslRetCodeToString(RetVal), RetVal, rsslRetCodeInfo(RetVal));
}
else
{
    RsslElementList elementList = RSSL_INIT_ELEMENT_LIST;
    RsslBuffer nonRWFBuffer = RSSL_INIT_BUFFER;
    /* retVal should be RSSL_RET_ENCODE_MSG_KEY_OPAQUE */
    /* encode msgKey attrib as element list to match setting of attribContainerType */
    {
        RsslElementList elementList = RSSL_INIT_ELEMENT_LIST;
        elementList.flags = RSSL_ELF_HAS_STANDARD_DATA;
        /* now encode nested container using its own specific encode functions */
        if ((RetVal = rsslEncodeElementListInit(&encIter, &elementList, 0, 0)) <
            RSSL_RET_SUCCESS)
            /*----- Continue encoding element entries. See example in Section 11.3.2 ----- */
            /* Complete nested container encoding */
            retVal = rsslEncodeElementListComplete(&encIter, success);
    }
    /* now that it is done, complete msgKey attrib encoding. */
    retVal = rsslEncodeMsgKeyAttribComplete(&encIter, success);
}

```

```

/* retVal should be RSSL_RET_ENCODE_EXTENDED_HEADER */
/* encode extended header as non-RWF type using non-RWF encode functions */
{
    retVal = rsslEncodeNonRWFDataInit(&encIter, &nonRWFBuffer);
    /* now encode extended header using its own specific encode functions -
    Ensure that we do not exceed nonRWFBuffer.length */
    /* we could memcpy into the nonRWFBuffer.data or use it with other encode functions */
    memcpy(&nonRWFBuffer.data, &data, length);
    /* Set nonRWFBuffer.length to amount of data encoded into buffer and complete */
    nonRWFBuffer.length = encAnsiBuffer.length;
    retVal = rsslEncodeNonRWFDataComplete(&encIter, &nonRWFBuffer, success);
}
retVal = rsslEncodeExtendedHeaderComplete(&encIter, success);

/* retVal should be RSSL_RET_ENCODE_CONTAINER */
/* encode message payload to match msgBase.containerType */
{
    RsslElementList elementList = RSSL_INIT_ELEMENT_LIST;
    elementList.flags = RSSL_ELF_HAS_STANDARD_DATA;
    /* now encode nested container using its own specific encode functions */
    if ((retVal = rsslEncodeElementListInit(&encIter, &elementList, 0, 0)) <
RSSL_RET_SUCCESS)
        /*----- Continue encoding element entries. See example in Section 11.3.2 ----- */
        /* Complete nested container encoding */
        retVal = rsslEncodeElementListComplete(&encIter, success);
    }
    /* now that specified msgKey attrib, extendedHeader and payload are done, complete message
    /* encoding. */
    retVal = rsslEncodeMsgComplete(&encIter, success);
}

```

Code Example 41: **RsslMsg Encoding Example #1, rsslEncodeMsgInit / rsslEncodeMsgComplete Use**

12.2.9.3 RsslMsg Encoding Example 2

The following code sample demonstrates **RsslMsg** encoding and shows the use of **rsslEncodeMsg** with pre-encoded **msgKey** attribute information and payload. While this example demonstrates error handling for the initial encode function, it omits additional error handling to simplify the example (though it should still be performed).

```

/* EXAMPLE 2 - EncodeMsg with pre-encoded msgKey.attrib and pre-encoded payload, no
/* extendedHeader */

/* Populate and encode a refreshMsg */
RsslRefreshMsg refreshMsg = RSSL_INIT_REFRESH_MSG;
refreshMsg.msgBase.msgClass = RSSL_MC_REFRESH; /* message is a refresh */

```

```

refreshMsg.msgBase.domainType = RSSL_DMT_MARKET_PRICE;
refreshMsg.msgBase.containerType = RSSL_DT_FIELD_LIST;
/* Use the stream Id corresponding to the request, because it is in reply to a request, it's
/* solicited */
refreshMsg.msgBase.streamId = 6;
/* Populate stream and data state information. This is required on an RsslRefreshMsg */
refreshMsg.state.streamState = RSSL_STREAM_OPEN;
refreshMsg.state.dataState = RSSL_DATA_OK;
/* Populate flags for refresh message members and behavior - because this in response to a
/* request this should be solicited, msgKey should be present, single part refresh so it is
/* complete, and also want the concrete qos of the stream */
refreshMsg.flags = RSSL_RFMF_SOLICITED | RSSL_RFMF_HAS_MSG_KEY | RSSL_RFMF_REFRESH_COMPLETE
    | RSSL_RFMF_HAS_QOS | RSSL_RFMF_CLEAR_CACHE;
/* Populate msgKey to specifie a serviceId, a name with type of RIC (which is default nameType)
/* and attrib */
refreshMsg.msgBase.msgKey.flags = RSSL_MKF_HAS_SERVICE_ID | RSSL_MKF_HAS_NAME |
    RSSL_MKF_HAS_ATTRIB;
refreshMsg.msgBase.msgKey.serviceId = 1;
/* Specify name and length of name. Because this is a RIC, no nameType is required. */
refreshMsg.msgBase.msgKey.name.data = "TRI";
refreshMsg.msgBase.msgKey.name.length = 3;
/* Msg Key attribute info is pre-encoded, should be set in encAttrib */
refreshMsg.msgBase.attribContainerType = RSSL_DT_ELEMENT_LIST;
/* assuming pEncodedAttrib RsslBuffer contains the pre-encoded msgKey attribute info with
/* data and length populated */
refreshMsg.msgBase.msgKey.encAttrib.data = pEncodedAttrib->data;
refreshMsg.msgBase.msgKey.encAttrib.length = pEncodedAttrib->length;
/* assuming pEncodedPayload RsslBuffer contains the pre-encoded payload information with
/* data and length populated */
refreshMsg.msgBase.encDataBody.data = pEncodedPayload->data;
refreshMsg.msgBase.encDataBody.length = pEncodedPayload->length;

/* encode message - assumes that encIter is already populated with buffer and version
/* information, store return value to determine success or failure */
/* Because this function expects all portions to be populated and pre-encoded, all message
/* encoding is complete after this returns. */
if ((RetVal = rsslEncodeMsg(&encIter, (RsslMsg*)&refreshMsg )) < RSSL_RET_SUCCESS)
{
    /* error condition - switch our success value to false so we can roll back */
    success = RSSL_FALSE;
    /* print out message with return value string, value, and text */
    printf("Error %s (%d) encountered with rsslEncodeMsg. Error Text: %s\n",
        rsslRetCodeToString(RetVal), RetVal, rsslRetCodeInfo(RetVal));
}

```

Code Example 42: Rssl Msg Encoding Example #2, rssl EncodeMsg Use

12.2.9.4 RsslMsg Decoding Interfaces

When decoding, `RsslMsg.msgBase` contains common members that can identify the specific message class or domain type. Because `msgKey` is optional and specified on a per-message class basis, do not use `msgBase.msgKey` until the specific message class flags are consulted to determine whether the `msgKey` is present.

A decoded `RsslMsg` structure provides access to the encoded content of the message. You can further decode the message's content by invoking the specific contained type's decode function.

All message encoding and decoding functions expect the `RsslMsg` type. Any specific message class can be cast to the `RsslMsg`, and an `RsslMsg` can be cast to any specific message class.

Decode Interface	Description
rsslDecodeMsg	<p>Decodes <code>RsslMsg</code> header members. Any <code>msgKey</code> attribute information remains encoded unless the user chooses to decode it. This can be accomplished by setting the <code>encAttrib</code> buffer on a separate <code>RsslDecodeIterator</code> or by calling <code>rsslDecodeMsgKeyAttrib</code> followed by decode functions for the specified <code>attribContainerType</code>. Any message payload content will be described by the message's <code>containerType</code> member and will be present in the <code>encDataBody</code>. This can be decoded by calling the <code>containerType</code>'s specific decode functions using the same <code>RsslDecodeIterator</code> or by setting the <code>encDataBody</code> on a new decode iterator. Any <code>extendedHeader</code> information is expected to be decoded by using a separate <code>RsslDecodeIterator</code>. This function will decode from the <code>RsslBuffer</code> to which the passed in <code>RsslDecodeIterator</code> refers.</p>
rsslDecodeMsgKeyAttrib	<p>Prepares the <code>RsslDecodeIterator</code> to decode <code>RsslMsg.msgKey.encAttrib</code> information. This function expects the same <code>RsslDecodeIterator</code> as was used with <code>rsslDecodeMsg</code> and the <code>RsslMsg.msgKey</code> member that was populated by calling <code>rsslDecodeMsg</code>. This populates <code>encData</code> with an encoded entry. After this function returns, you can call the <code>msgKey.attribContainerType</code> decode functions to decode attribute information. If you do not want to decode <code>msgKey</code> attribute information, you can decode the payload by using the <code>containerType</code>'s decode functions after <code>rsslDecodeMsg</code> returns.</p>

Table 187: Rssl Msg Decode Functions

12.2.9.5 RsslMsg Decoding Example

The following code sample demonstrates how to decode an `RsslMsg`. This sample code uses a switch statement to decode the message's content. Typically an application would invoke the specific container type decoder for the housed type or use a switch statement to allow for a more generic message decoding. The example uses the same `RsslDecodeIterator` when decoding the `msgKey.encAttrib` and the message payload. An application could optionally use a new `RsslDecodeIterator` by setting the `encAttrib` or `encDataBody` on a new iterator. To simplify the sample code, some error handling is omitted.

```

/* decode contents into the RsslMsg structure */
if ((RetVal = rsslDecodeMsg(&decIter, &rsslMsg)) >= RSSL_RET_SUCCESS)
{
    /* we can cast to the appropriate message class for convenience or use the accessor */
    /* methods */
    const RsslMsgKey *pKey;
    /* use the ease of use accessor to get the msgKey if it exists on this msgClass */
    pKey = rsslGetMsgKey(&rsslMsg);
    /* if we have a key and it has attribute information, decode it */
    if (pKey && (pKey->flags & RSSL_MKF_HAS_ATTRIB))
    {
        /* need to set up the decodeIterator to expect decoding of attribute information,
         * Otherwise it will assume we are decoding the payload */
        retVal = rsslDecodeMsgKeyAttrib(&decIter, pKey));
    }

    switch (pKey->attribContainerType)
    {

        case RSSL_DT_FIELD_LIST:
            retVal = rsslDecodeFieldList(&decIter, &fieldList, 0);
            /* Continue decoding field entries. Refer to the example in Section 11.3.1 */
            break;
        case RSSL_DT_ELEMENT_LIST:
            retVal = rsslDecodeElementList(&decIter, &elementList, 0);
            /* Continue decoding element entries. Refer to the example in Section 11.3.2*/
            break;
        /* full switch statement omitted to shorten sample code */
    }
}

/* Decode any contained payload information */
switch (rsslMsg.msgBase.containerType)
{
    case RSSL_DT_NO_DATA:
        printf("No payload contained in message.\n");
        break;
    case RSSL_DT_FIELD_LIST:
        retVal = rsslDecodeFieldList(&decIter, &fieldList, 0);
        /* Continue decoding field entries. Refer to the example in Section 11.3.1 */
}

```

```

break;
case RSSL_DT_ELEMENT_LIST:
    retVal = rsslDecodeElementList(&decIter, &elementList, 0);
    /* Continue decoding element entries. Refer to the example in Section 11.3.2 */
break;
/* full switch statement omitted to shorten sample code */
}

}
else
{
    /* decoding failure tends to be unrecoverable */
printf("Error %s (%d) encountered with rsslDecodeMsg. Error Text: %s\n",
       rsslRetCodeToString(retVal), retVal, rsslRetCodeInfo(retVal));
}

```

Code Example 43: RsslMsg Decoding Example

12.2.9.6 RsslMsg Utility Functions

The Transport API provides the following utility functions for use with the [RsslMsg](#).

FUNCTION NAME	DESCRIPTION
rsslClearMsg	Clears members from an RsslMsg structure. Useful for structure reuse.
rsslValidateMsg	Performs a basic validation on the populated RsslMsg structure (useful when encoding) ensuring that optional members indicated as present are correctly populated (e.g., that length and data are both populated).
rsslIsFinalMsg	Returns true if the message is the last message received on a stream, such as: <ul style="list-style-type: none"> Final response to non-streaming requests Messages with a streamState indicating a closed stream (refer to Section 11.2.6) Explicitly closed streams (e.g. closed with an RsslCloseMsg). Returns false if data is to continue streaming.
rsslSizeOfMsg	Performs a deep sizeof function on an RsslMsg structure. rsslSizeOfMsg is not the same as the encoded size of the message, though it can be useful for approximating the encoded size (it is typically smaller than the structural representation).
rsslCopyMsg	Performs a deep copy of an RsslMsg structure. rsslCopyMsg can internally create the memory needed for copying or the user can pass in the needed memory. <ul style="list-style-type: none"> If memory is passed in by the user, the user is responsible for managing the memory. If rsslCopyMsg creates memory for copying, you must call rsslReleaseCopiedMsg to ensure proper cleanup.

Table 188: RsslMsg Utility Functions

FUNCTION NAME	DESCRIPTION
rsslReleaseCopiedMsg	Performs proper cleanup of memory allocated by <code>rsslCopyMsg</code> . Only memory internally created by <code>rsslCopyMsg</code> should be passed into this function. This function cleans up on a per-message basis (e.g., each message created by <code>rsslCopyMsg</code> requires individual calls to <code>rsslReleaseCopiedMsg</code>).
rsslGetFlags	Takes a populated <code>RsslMsg</code> structure and returns the specific <code>msgClass</code> 's flags.
rsslGetMsgKey	Takes a populated <code>RsslMsg</code> structure, determines whether <code>msgKey</code> is present and returns it if available, NULL otherwise. For more details about the <code>msgKey</code> , refer to Section 12.1.2.
rsslGetSeqNum	Takes a populated <code>RsslMsg</code> structure, determines whether <code>seqNum</code> is present and returns it if available, NULL otherwise.
rsslGetState	Takes a populated <code>RsslMsg</code> structure, determines whether <code>state</code> information is present and returns it if available, NULL otherwise. For more information about state values, refer to Section 11.2.6.
rsslGetPermData	Takes a populated <code>RsslMsg</code> structure, determines whether <code>permData</code> information is present and returns it when available, NULL otherwise. For more information about permission data, refer to Section 11.4.
rsslGetGroupId	Takes a populated <code>RsslMsg</code> structure, determines whether <code>groupId</code> information is present and returns it when available, NULL otherwise. For more information about group use, refer to Section 13.4.
rsslGetExtendedHeader	Takes a populated <code>RsslMsg</code> structure, determines whether <code>extendedHeader</code> information is present, and returns it if available, NULL otherwise.
rsslExtractMsgClass	Takes an encoded message and returns the <code>msgClass</code> information without fully decoding the message header. Note: Multiple <code>rsslExtract*</code> calls on the same encoded message will likely be less efficient than a single call to <code>rsslDecodeMsg</code> .
rsslExtractDomainType	Takes an encoded message and returns the <code>domainType</code> information without fully decoding the message header. Note: Multiple <code>rsslExtract*</code> calls on the same encoded message will likely be less efficient than a single call to <code>rsslDecodeMsg</code> .
rsslExtractStreamId	Takes an encoded message and returns the <code>streamId</code> information without fully decoding the message header. For more details on the <code>streamId</code> , refer to Section 12.1.3. Note: Multiple <code>rsslExtract*</code> calls on the same encoded message will likely be less efficient than a single call to <code>rsslDecodeMsg</code> .
rsslExtractSeqNum	Takes an encoded message and returns the <code>seqNum</code> information without fully decoding the message header. Note: Multiple <code>rsslExtract*</code> calls on the same encoded message will likely be less efficient than a single call to <code>rsslDecodeMsg</code> .

Table 188: `Rssl Msg` Utility Functions (Continued)

FUNCTION NAME	DESCRIPTION
rsslExtractGroupId	Takes an encoded message and returns the <code>groupId</code> information without fully decoding the message header. For more information about group use, refer to Section 13.4. Note: Multiple <code>rsslExtract*</code> calls on the same encoded message will likely be less efficient than a single call to <code>rsslDecodeMsg</code> .
rsslExtractPostId	Takes an encoded message and returns the <code>postId</code> information without fully decoding the message header. For more information, refer to Section 13.9. Note: Multiple <code>rsslExtract*</code> calls on the same encoded message will likely be less efficient than a single call to <code>rsslDecodeMsg</code> .
rsslReplaceDomainType	Takes an encoded message and replaces the <code>domainType</code> without re-encoding the message.
rsslReplaceStreamId	Takes an encoded message and replaces the <code>streamId</code> without re-encoding the message. For more details on the <code>streamId</code> , refer to Section 12.1.3.
rsslReplaceSeqNum	Takes an encoded message and replaces the <code>seqNum</code> without re-encoding the message.
rsslReplaceGroupId	Takes an encoded message and replaces the <code>groupId</code> without re-encoding the message. For more information about group use, refer to Section 13.4.
rsslReplacePostId	Takes an encoded message and replaces the <code>postId</code> without re-encoding the message. For more information, refer to Section 13.9.
rsslReplaceStreamState	Takes an encoded message and replaces the <code>streamState</code> without re-encoding the message. For more information about state values, refer to Section 11.2.6.
rsslReplaceDataState	Takes an encoded message and replaces the <code>dataState</code> without re-encoding the message. For more information about state values, refer to Section 11.2.6.
rsslReplaceStateCode	Takes an encoded message and replaces the <code>state.code</code> without re-encoding the message. For more information about state values, refer to Section 11.2.6.

Table 188: Rssl Msg Utility Functions (Continued)

Chapter 13 Advanced Messaging Concepts

13.1 Multi-Part Message Handling

`RsslRefreshMsg`, `RsslPostMsg`, and `RsslGenericMsg` all support splitting payload content across multiple message parts, commonly referred to as **message fragmentation**. Each message part includes relevant message header information along with the part's payload, where payload can be combined by following the modification semantics associated with the specific `containerType` (for specific container details, refer to Section 11.3). Message fragmentation is typically used to split large payload information into smaller, more manageable pieces. The size of each message part can vary, and is controlled by the application that performs the fragmentation. Often, sizes are chosen based on a specific transport layer frame or packet size.

When sending a multi-part message, several message members can convey additional part information. Each message class that supports fragmentation has an optional `partNum` member that can order and ensure receipt of every part of the message. For consistency and compatibility with TREP components, `partNum` should begin with **0** and increment by one for each subsequent part. Several container types have an optional `totalCountHint` value. This can convey information about the expected entry count across all message parts, and often helps size needed storage or display for the message contents.

These message classes have an associated **COMPLETE** flag value (specifically `RSSL_RFMF_REFRESH_COMPLETE`, `RSSL_PSMF_POST_COMPLETE`, and `RSSL_GNMF_MESSAGE_COMPLETE`). A flag value of **COMPLETE** indicates the final part of a multi-part message (or that the message is a single-part and no subsequent parts will be delivered).

For both streaming and non-streaming information, other messages might arrive between parts of a fragmented message. For example, it is expected that update messages be received between individual parts of a multi-part refresh message. Such updates indicate changes to data being received on the stream and should be applied according to the modification semantics associated with the `containerType` of the payload. If non-streaming, no additional messages should be delivered after the final part.

If a transport layer is used, messages can fan out in the order in which they are received. On a transport where reliability is not guaranteed and the order can be determined by a sequence number, special rules should be used by consumers when processing a multi-part message. The following description explains how a multi-part refresh message can be handled. After the request is issued, any messages received on the stream should be stored and properly ordered based on sequence number. When an application encounters the first part of the `RsslRefreshMsg`, the application should process the part and note its sequence number. The application can drop (i.e., not process) stored messages with earlier sequence numbers. When the application encounters the next part of the `RsslRefreshMsg`, the application should first process any stored message with a sequence number intermediate between this refresh part and the previous part then the application should process the refresh part. This process should continue until the final part of the `RsslRefreshMsg` is encountered, at which time any remaining stored messages with a later sequence number should be processed and the stream's data flow can continue as normal.

13.2 Stream Priority

Consumers use `RsslRequestMsg` to indicate the stream's level of importance, conveyed by the priority information. When a consumer is aggregating streams on behalf of multiple users, the priority typically corresponds to the number of users interested in the particular stream. A consumer can increase or decrease a stream's associated priority information by issuing a subsequent request message on an already open stream.

A Provider application tracks the priority of each of its open streams. If the consumer reaches some kind of item count limitation (i.e., the maximum allowable number of streams), the provider can employ a preemption algorithm. Specific details must be defined by the provider application. The ADH uses the combination of `priorityCount` and `priorityClass` to preempt items when the user's allowable cache list size is exceeded. ADH always preempts the item with the lowest `priorityCount` within the `priorityClass` and then provides an `RsslStatusMsg` with a `streamState` of `RSSL_STREAM_CLOSED_RECOVER` for the item.

Priority is represented by a `priorityClass` value and a `priorityCount` value.

- The **priority class** indicates the general importance of the stream to the consumer.
- The **priority count** indicates the stream's specific importance within the priority class.

The **priorityClass** value takes precedence over any **priorityCount** value. For example, a stream with a **priorityClass** of 5 and **priorityCount** of 1 has a higher overall priority than a stream with a **priorityClass** of 3 and a **priorityCount** of 10,000.

Because priority information is optional on an **RsslRequestMsg**:

- If priority information is not present on an initial request to open a stream, it is assumed that the stream has a **priorityClass** and a **priorityCount** of 1.
- If priority information is not present on a subsequent request message on an open stream, this means that the priority has not changed and previously stored priority information continues to apply.

If a consumer aggregates identical streams, the consumer should use the highest **priorityClass** value. Individual **priorityCount** values are always combined on a per-**priorityClass** basis.

For example, if a consumer application combines three identical streams:

- One with **priorityClass** 3 and **priorityCount** 5
- One with **priorityClass** 2 and **priorityCount** 10
- One with **priorityClass** 3 and **priorityCount** of 1

In this case, the aggregate priority information would be **priorityClass** 3 (i.e., the highest **priorityClass**) and **priorityCount** of 6 (the combined **priorityCount** values for that class level).

13.3 Stream Quality of Service

A consumer can use **RsslRequestMsg** to indicate the desired QoS for its streams. This can be a request for a specific QoS or a range of qualities of service, where any value within the range will satisfy the request. The **RsslRefreshMsg** includes the QoS used to indicate the QoS being provided for a stream. When issuing a request, the QoS specified on the request typically matches the advertised QoS of the service, as conveyed via the Source Directory domain model. For more information, refer to the *Transport API C Edition RDM Usage Guide*.

- An initial request containing only **RsslRequestMsg.qos** indicates a request for the specified QoS. If a provider cannot satisfy this QoS, the request should be rejected.
- An initial request containing both **RsslRequestMsg.qos** and **RsslRequestMsg.worstQos** sets the range of acceptable QoSs. Any QoS within the range, inclusive of the specified **qos** and **worstQos**, will satisfy the request. If a provider cannot provide a QoS within the range, the provider should reject the request.

When a provider responds to an initial request, the **RsslRefreshMsg.qos** should contain the actual QoS being provided for the stream. Subsequent requests issued on the stream should not specify a range as the QoS has been established for the stream.

Because QoS information is optional on an **RsslRequestMsg** some special handling is required when it is absent.

- If neither **qos** nor **worstQos** are specified on an initial request to open a stream, it is assumed that any QoS will satisfy the request.
- If QoS information is absent on a subsequent reissue request, it is assumed that QoS, timeliness, and rate conform to the stream's currently established settings.
- If QoS information is absent in an initial **RsslRefreshMsg**, this should be assumed to have a **timeliness** of **RSSL_QOS_TIME_REALTIME** and a **rate** of **RSSL_QOS_RATE_TICK_BY_TICK**. On any subsequent solicited or unsolicited refresh, this should be assumed to match any QoS already established by the initial **RsslRefreshMsg**.

To determine whether components require QoS information on initial and reissue requests, refer to the documentation for the specific component.

13.4 Item Group Use

You can use item groups to efficiently update the state for multiple item streams via a single group status message (instead of using multiple, individual item status messages). Each open data stream is assigned an item group. This information is associated with the stream through the `RsslRefreshMsg.groupId` (refer to Section 12.2.2) or `RsslStatusMsg.groupId` (refer to Section 12.2.4) members. Once established, item group information can be modified via a subsequent `RsslStatusMsg` or `RsslRefreshMsg` containing a different `groupId` affiliation.

Item groups are defined on a per-service basis. While two item groups can have the same `groupId`, each group's `serviceId` will be unique. A consumer application should track `serviceId-groupId` pairings to ensure the correct sets of items are modified whenever group status messages are received. A provider can establish item group assignments according to the application's needs, but must maintain the uniqueness of each item group within a service. For example, a provider that aggregates multiple upstream services into a single downstream service might establish a different item group for each aggregated service. Thus, should an upstream service become unavailable, the provider can mark all items as being suspect while items from other upstream services remain in their prior state.

13.4.1 Item Group Buffer Contents

The consuming application should treat data (which may be of varying length) contained in the `groupId` buffer as opaque. A simple memory comparison operation can determine whether two groups are equivalent. The actual data contained in the `groupId` buffer is a collection of one or more unsigned two-byte, unsigned integer values, where each two-byte value is appended to the end of the current `groupId RsslBuffer`. Providers that combine multiple data sources must ensure that the item groups in the resulting service are unique, which can be accomplished by appending an additional two-byte value to each on-passed `groupId`.

For example, the following figure depicts two NIP applications, each publishing item streams belonging to specific services and item groups.

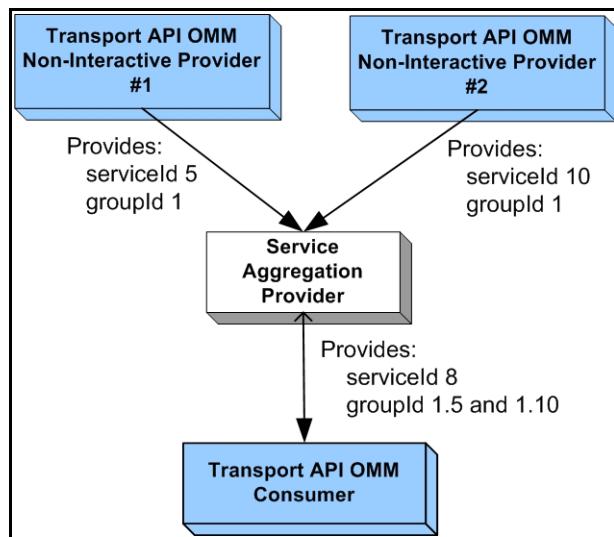


Figure 38. Item Group Example

Though the providers in this diagram use the same `groupId` for an item, using different `serviceIds` makes items unique. Both providers communicate with an application that consumes data from both services, aggregates the data into a single service, and then distributes the data to consumer applications. To ensure uniqueness to downstream components, the service aggregation provider appends additional identifiers to the group information it receives from the provider applications. In this

example, the aggregation device modifies `serviceId 5, groupId 1` into a `groupId` of **1.5** and `serviceId 10, groupId 1` into a `groupId` of **1.10**. If for any reason NIP #1's service becomes unavailable, the aggregation device can send a single group status message to inform the consumer that all items belonging to `groupId 1.5` are suspect. This would have no impact to any items belonging to `groupId 1.10`.

13.4.2 Item Group Utility Functions

The Transport API provides the following utility functions for use with and modification of the `groupId RsslBuffer`.

function NAME	DESCRIPTION
<code>rsslAddGroupId</code>	Appends a two-byte, unsigned integer to existing <code>groupId</code> content. Useful when modifying <code>groupId</code> buffers to ensure uniqueness.
<code>rsslGetGroupId</code> (from <code>RsslRefreshMsg</code> and <code>RsslStatusMsg</code>)	Takes a populated <code>RsslMsg</code> structure, determines if <code>groupId</code> information is present and if available, returns it; NULL otherwise.
<code>rsslExtractGroupId</code>	Takes an encoded message and returns the <code>groupId</code> without fully decoding the message header.
	Note: Multiple <code>rsslExtract*</code> calls on the same encoded message will likely be less efficient than a single call to <code>rsslDecodeMsg</code> .
<code>rsslReplaceGroupId</code>	Takes an encoded message and replaces the <code>groupId</code> without re-encoding the message.

Table 189: Item Group Utility Functions

13.4.3 Group Status Message Information

Information regarding state changes and the merging of item groups occurs via group status messages. A group status message is communicated via the Source Directory domain message model. Specific group information is contained in the Directory's Group `RsslFilterEntry` which corresponds to the specific service associated with the group.

- For more specific information, refer to the Source Directory Domain section in the *Transport API C Edition RDM Usage Guide*.
- For a decision table providing example behavior for various state combinations, refer to Appendix A.

Note: If an application does not subscribe to the Source Directory's group filter, the application will not receive group status messages. This can result in potentially incorrect item state information, as relevant status information might be missed.

13.4.4 Group Status Responsibilities by Application Type

Dissemination and handling of group status information is distributed across providers and consumers. This section discusses responsibilities by application type.

An OMM interactive provider or NIP application is responsible for:

- Assigning and providing item group id values. This is accomplished by specifying the `RsslRefreshMsg.groupId` or `RsslStatusMsg.groupId` for all provided content¹.
- If a group of items becomes unavailable (i.e., an upstream service or provider goes down), group status messages should be sent out for all affected item groups. These are sent via the Source Directory domain.

1. This does not include administrative domains such as Login, Source Directory, and Dictionary.

For more information about group status messages (including specific message content and formatting), refer to the *Transport API C Edition RDM Usage Guide*.

- If items become available again, recovery should occur and items' states should be updated via a subsequent `RsslRefreshMsg` or `RsslStatusMsg` provided to any downstream components interested in the item.

An OMM consumer application is responsible for:

- Subscribing to the item group filter when requesting Source Directory information.

For more information about the item group filter and group status messages (including specific message content and formatting), refer to the *Transport API C Edition RDM Usage Guide*.

- If group status changes are received, the state change should be propagated to all items associated with the indicated group, as noted by the `RsslRefreshMsg.groupId` or `RsslStatusMsg.groupId` provided with the item stream.
- Any recovery should follow `SingleOpen` and `AllowSuspectData` rules, as described in the *Transport API C Edition RDM Usage Guide*.

13.5 Single Open and Allow Suspect Data Behavior

A consumer application can specify desired item recovery and state transition information on its Login domain `RsslRequestMsg` using the `SingleOpen` and `AllowSuspectData msgKey` attributes. A providing application can acknowledge support for the behavior in the Login domain `RsslRefreshMsg`, in which case the provider performs certain state transitions. This section offers a high-level description of item recovery and state transition behavior modifications.

- **Single open** behavior allows a consumer application to open an item stream once and have an upstream component handle stream recovery (if needed). With single open enabled, a consumer should not receive a `streamState` of `CLOSED_RECOVER`, as the providing application should convert to `SUSPECT` and attempt to recover on the consumer's behalf. If a stream is `CLOSED`, this will be propagated to the consumer application.
- **Allow suspect data** behavior indicates whether an application can tolerate an open stream with a `dataState` of `SUSPECT`, or if it is preferable to have the stream closed. If an application indicates that it does not wish to allow `SUSPECT` streams to remain open, the providing application should transition the `streamState` to `CLOSED_RECOVER`.

If the providing application does not support either behavior, the application should indicate such a restriction in the Login domain's `RsslRefreshMsg`. For additional information, including on the `RSSL_DMT_LOGIN` domain definition, refer to the *Transport API C Edition RDM Usage Guide*.

The following table shows how a provider can convert messages to correspond with the consumer's `SingleOpen` and `AllowSuspectData` settings. The first column in the table shows the actual `streamState` and `dataState`. Each subsequent column shows how this state information can be modified to follow the column's specific `SingleOpen` and `AllowSuspectData` settings. If a `SingleOpen` and `AllowSuspectData` configuration causes a behavioral contradiction (e.g., `SingleOpen` indicates that the provider should handle recovery, but `AllowSuspectData` indicates that the consumer does not want to receive suspect status), the `SingleOpen` configuration takes precedence.

Note: The Transport API does not perform special processing based on the `SingleOpen` and `AllowSuspectData` settings. The provider application must perform any necessary conversion.

ACTUAL STATE INFORMATION	CONVERSION WHEN: SINGLEOPEN = 1 ALLOWSSUSPECTDATA = 1	CONVERSION WHEN: SINGLEOPEN = 1 ALLOWSSUSPECTDATA = 0	CONVERSION WHEN: SINGLEOPEN = 0 ALLOWSSUSPECTDATA = 1	CONVERSION WHEN: SINGLEOPEN = 0 ALLOWSSUSPECTDATA = 0
streamState = OPEN dataState = SUSPECT	No conversion required	No conversion required	No conversion required	streamState = CLOSED_RECOVER dataState = SUSPECT
streamState = CLOSED_RECOVER dataState = SUSPECT	streamState = OPEN dataState = SUSPECT	streamState = OPEN dataState = SUSPECT	No conversion required	No conversion required

Table 190: SingleOpen and AllowSuspectData Effects

13.6 Pause and Resume

The Transport API allows applications to send or receive requests to pause or resume content flow on a stream.

- Issuing a **pause** on a stream can result in the temporary stop of **RsslUpdateMsg** flow.
- Issuing a **resume** on a paused stream restarts the **RsslUpdateMsg** flow.

Pause and resume can help optimize bandwidth by pausing streams that are only temporarily not of interest, instead of closing and re-requesting a stream. Though a pause request may be issued on a stream, it does not guarantee that the contents of the stream will actually be paused. Additionally, if the contents of the stream are paused, state-conveying messages can still be delivered (i.e., status messages and unsolicited refresh messages). Pause and resume is only valid for data streams instantiated as streaming (**RSSL_RQMF_STREAMING**). The consumer application is responsible for continuing to handle all delivered messages, even after the issuance of a pause request.

A consumer application can request to pause an individual item stream by issuing **RsslRequestMsg** with the **RSSL_RQMF_PAUSE** flag set. This can occur on the initial **RsslRequestMsg** or via a subsequent **RsslRequestMsg** on an established stream (i.e., a reissue). If a pause is issued on the initial request, it should always result in the delivery of the initial **RsslRefreshMsg** (this conveys initial state, permissioning, QoS, and group association information necessary for the stream). A paused stream remains paused until a resume request is issued. To resume data flow on a stream a consumer application can issue a subsequent **RsslRequestMsg** with the **RSSL_RQMF_STREAMING** flag set.

If a provider application receives a pause request from a consumer, it can choose to pause the content flow or continue delivering information. When pausing a stream, where possible, the provider should aggregate information updates until the consumer application resumes the stream. When resuming, an aggregate update message should be delivered to synchronize the consumer's information to the current content. However, if data cannot be aggregated, resuming the stream should result in a full, unsolicited **RsslRefreshMsg** to synchronize the consumer application's information to a current state.

A pause request issued on the **streamId** associated with a user's login is interpreted as a request to **pause all** streams associated with the user. A pause all request is only valid for use on an already established login stream and cannot be issued on the initial login request. A 'pause all' request affects open streams only. Any newly requested streams should follow behaviors specified on the request message itself (e.g. streaming, non-streaming, paused, etc). After a pause all request, the application can choose to either resume individual item streams or resume all streams. A **resume all** will result in all paused streams being transitioned to a resumed state. This is performed by issuing a subsequent **RsslRequestMsg** with the **RSSL_RQMF_STREAMING** flag set using the **streamId** associated with the applications login.

For more information about the **RsslRequestMsg** and the **RSSL_RQMF_PAUSE** or **RSSL_RQMF_STREAMING** flag values, refer to Section 12.2.1.

A provider application can indicate support for pause and resume behavior by sending the **msgKey** attribute **supportOptimizedPauseResume** in the Login domain **RsslRefreshMsg**. For more details on the Login **domainType** (**RSSL_DMT_LOGIN**), refer to the *Transport API C Edition RDM Usage Guide*.

13.7 Batch Messages

Applications can use the Transport API to send and / or receive batch messages as a more efficient way to handle requests, reissues, or closes of multiple items. When a consumer application wishes to open multiple similar items at once, or close multiple streams, it may perform the operation using a single message instead of sending a message for each individual stream.

Note: Batch messages use the `RsslElementEntry` names `:ItemList` and `:StreamIdList` in message payloads. These names follow a namespace scheme in which a name's content prior to the character `:` indicates a namespace. Thomson Reuters reserves the empty namespace (e.g., `:Element`), while other namespaces are left for custom element names (e.g., `Customer:Element`)

This section defines the following types of operations that can be performed using a batch message:

- **Batch Requests**, to open streams for items that have different names but for which other key content (if any) is identical.
- **Batch Reissues**, to change attributes of multiple open streams such as priority, or to pause or resume streams.
- **Batch Closes**, to close multiple open streams.
- A provider application can indicate support for each form of batch messaging by sending a bitmask in the `msgKey` attribute `supportBatchRequests` in the Login domain `RsslRefreshMsg`. For more details on the Login domainType (`RSSL_DMT_LOGIN`) and the general use of batch messages, refer to the *Transport API RDM Usage Guide*. The `rsslRDM.h` header file included with the Transport API defines batch request-related enumerations and element name string constants.

13.7.1 Batch Request

Consumers use a batch request to indicate interest in multiple like-item streams with a single `RsslRequestMsg`. In this message, the consumer specifies a list of names in the message payload representing the items that the consumer wishes to open. Batch requesting can be leveraged across all non-administrative² domain model types.

A consumer application can issue a batch request by using an `RsslRequestMsg` with the `RSSL_RQMF_HAS_BATCH` flag set and including a specifically formatted payload. The payload should contain an `RsslElementList` along with an `RsslElementEntry` named `:ItemList`.

The `:ItemList` contains an `RsslArray`, where the `RsslArray.primitiveType` is `RSSL_DT_ASCII_STRING`. Each contained string (populated in an `RsslBuffer`) corresponds to a requested name. The `msgKey` contents, `domainType`, and any specified `qos` will be applied to all names in the list, and a `msgKey.name` (or `RSSL_MKF_HAS_NAME`) should not be present.

When a provider application receives a batch request, it should respond on the same stream with an `RsslStatusMsg` that acknowledges receipt of the batch by indicating the `dataState` is `OK` and `streamState` is `CLOSED`. The stream on which the batch request was sent (i.e., the ‘batch stream’) then closes, because all additional responses are provided on individual streams. The `:ItemList` should be traversed to obtain each requested name and the batch `RsslRequestMsg.msgKey` content should be associated with each item. If any request cannot be fulfilled, the provider should send an `RsslStatusMsg` to close the stream and indicate the reason (for further details, refer to Section 12.2.4). If the provider is unable to process the batch request itself, it should use a **SUSPECT** `dataState` in its response to the batch message.

Assignment of `streamId` values for all requested items is sequential according to the order of the entries in the `RsslArray`, beginning with $(1 + \text{streamId})$ of the batch `RsslRequestMsg`. Because an OMM consumer requests the batch, positive `streamId` values should be assigned. By setting the initial `streamId`, the consumer application can control the resultant

2. Administrative domain types are considered to be the Login, Directory, and Dictionary domain models. All other domains are considered non-administrative.

`streamId` range, ensuring enough available `streamId` values exist to allocate identifiers for all requested items. Consider the following example:

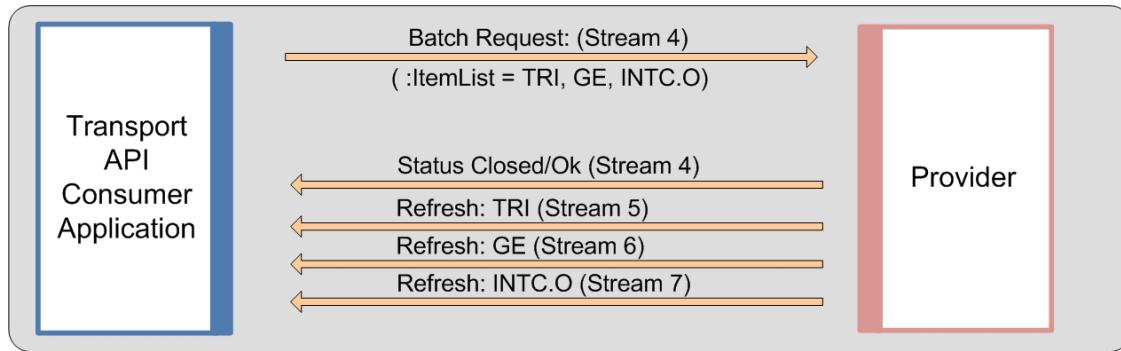


Figure 39. Batch Request Interaction Example“

In the above figure, the consumer uses `streamId` 4 to send a batch request for three items, having ensured that `streamIds` 5, 6, and 7 are available. The provider acknowledges the batch request by responding with an `RsslStatusMsg` on `streamId` 4, and provides the response for each of the three items on `streamIds` 5, 6, and 7, respectively.

Any view information (described in Section 13.8) included in a batch request should be applied for each item in the request. If a consumer application wants to reissue any item that was requested as part of a batch, the application can either issue a subsequent `RsslRequestMsg` on that item's `streamId`, or use a batch reissue to apply the reissue to multiple streams (described in Section 13.7.2).

- For an example of encoding a batch request, refer to Section 13.7.1.
- For more information about `RsslRequestMsg` and `RSSL_RQMF_HAS_BATCH` flag values, refer to Section 12.2.1.
- For more information about `RsslElementList`, refer to Section 11.3.2.

13.7.2 Batch Reissue

Consumers may use a batch reissue message to change attributes of multiple open streams (such as changing priority, or to pause or resume them) using a single `RsslRequestMsg`. In a batch reissue message, the consumer specifies a list of `streamIds` in the message payload representing the streams it wishes to reissue. Batch reissues can be leveraged across all non-Login domain model types.

A consumer application can issue a batch reissue by using an `RsslRequestMsg` with the `RSSL_RQMF_HAS_BATCH` flag set and including a specifically formatted payload. The payload should contain an `RsslElementList` along with an `RsslElementEntry` named `:StreamIdList`.

The `:StreamIdList` contains an `RsslArray`, where the `RsslArray.primitiveType` is `RSSL_DT_INT`. Each contained `streamId` (populated in an `RsslInt`) corresponds to the `streamId` of an open stream. The stream attributes specified (e.g., specifying the `RSSL_RQMF_PAUSE` flag, or changes to `priorityClass` and `priorityCount`) will be applied to each `streamId` in the list.

The consumer application may specify `streamIds` from from any non-Login domain in the `:streamIdList` of a batch reissue message; only the `streamId` is needed to identify the stream. The `qos`, `worstQos`, `msgKey`, `domainType`, and `extendedHeader` of the `RsslRequestMsg` are not used (do not set the `RSSL_RQMF_HAS_QOS`, `RSSL_RQMF_HAS_WORST_QOS`, or `RSSL_RQMF_HAS_EXTENDED_HEADER` flags. Set `msgKey.flags` to `RSSL_MKF_NONE`). Thomson Reuters recommends setting the `domainType` to `RSSL_DMT_MARKET_PRICE`). As with a batch request, a provider should respond on the same stream with an `RsslStatusMsg` that acknowledges receipt of the batch by indicating the `dataState` is `OK` and `streamState` is `CLOSED`, and the provider sends any additional responses on the individual streams. If any stream's reissue cannot be fulfilled, the provider should send an `RsslStatusMsg` on that stream to indicate the reason (for further details, refer to Section 12.2.4). If the provider is unable to process the batch message itself, it should use a `SUSPECT` `dataState` in the response to the batch message.

Consider the following interaction example:

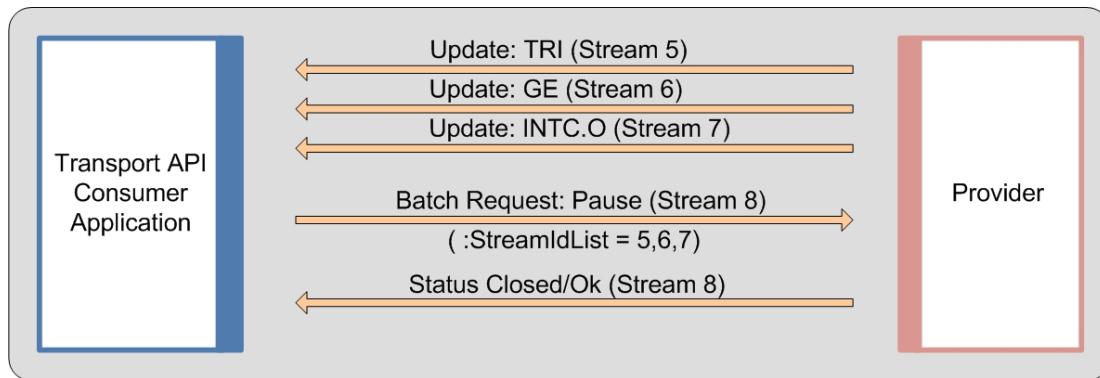


Figure 40. Batch Reissue (Pause) Interaction Example

In the above figure, the consumer currently has three items open on **streamIds** 5, 6, and 7 to a provider that supports pausing those streams. The consumer wishes to pause these three streams, so it sends an **RsslRequestMsg** using an unused **streamId**, 8. This message includes the **RSSL_RQMF_PAUSE** flag, and encodes the **streamIds** 5, 6, and 7 in the **:StreamIdList** element. The provider then responds with an **RsslStatusMsg** on **streamId** 8 to acknowledge the reissue message, and considers streams 5, 6, and 7 to be paused.

- For an example of encoding a batch reissue, refer to Section 13.7.5.
- For more information about **RsslRequestMsg** and **RSSL_RQMF_HAS_BATCH** flag values, refer to Section 12.2.1.
- For more information about the **RsslElementList**, refer to Section 11.3.2.

13.7.3 Batch Close

Consumers may use a batch close to close multiple open streams using a single **RsslCloseMsg**. In a batch close message, the consumer specifies a list of **streamIds** in the message payload representing the streams it wishes to close. Batch closes can be leveraged across all non-Login domain model types.

A consumer application can issue a batch close by using an **RsslCloseMsg** with the **RSSL_CLMF_HAS_BATCH** flag set and including a specifically formatted payload. The payload should contain an **RsslElementList** along with an **RsslElementEntry** named **:StreamIdList**.

The **:StreamIdList** contains an **RsslArray**, where the **RsslArray.primitiveType** is **RSSL_DT_INT**. Each contained **streamId** (populated in an **RsslInt**) corresponds to the **streamId** of an open stream which the consumer wishes to close.

The consumer application may specify **streamIds** from any non-Login domain in the **:streamIdList** of a batch close message; only the **streamId** is needed to identify the stream. The **domainType** is not used (Thomson Reuters recommends setting the **domainType** to **RSSL_DMT_MARKET_PRICE**). As with a batch request, a provider should respond on the same stream with an **RsslStatusMsg** that acknowledges receipt of the batch by indicating the **dataState** is **OK** and **streamState** is **CLOSED**. If the provider is unable to process the batch message itself, it should use a **SUSPECT** **dataState** in the response to the batch message.

Consider the following interaction example:

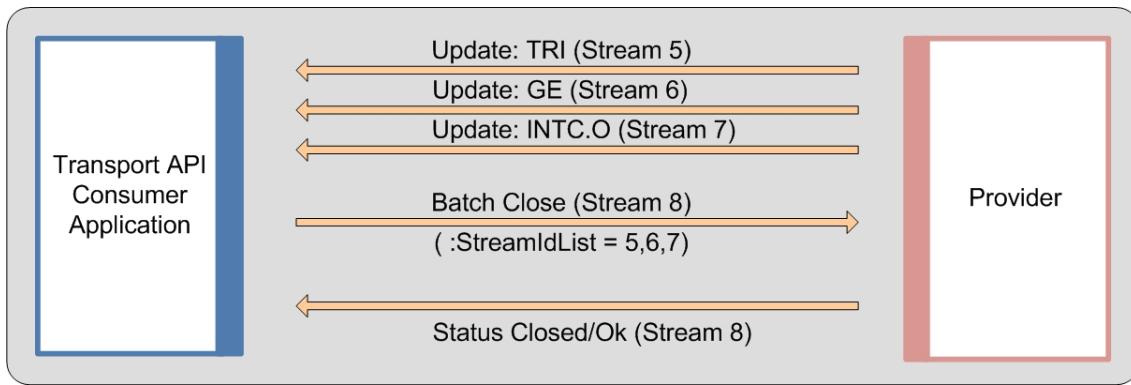


Figure 41. Batch Close Interaction Example

In the above figure, the consumer currently has streams open for three items with **streamIds** 5, 6, and 7. The consumer wishes to close these three streams, so it encodes **streamIds** 5, 6, 7, in the **:streamIdList** element of an **RsslCloseMsg** using an unused **streamId**, 8. This message encoded the **streamIds** 5, 6, and 7 in the **:streamIdList** element. The provider then responds with an **RsslStatusMsg** on **streamId** 8 to acknowledge the reissue message. The provider then responds with an **RsslStatusMsg** to acknowledge the close message and considers streams 5, 6, and 7 to be closed.

- For an example of encoding a batch close, refer to Section 13.7.6.
- For more information about **RsslCloseMsg** and **RSSL_CLMF_HAS_BATCH** flag values, refer to Section 12.2.5.
- For more information about the **RsslElementList**, refer to Section 11.3.2.

13.7.4 Batch Request Encoding Example

The following example demonstrates how to encode a batch request using an **RsslRequestMsg**. The request is sent using a **streamId** of **10** and contains an **:ItemList** of three items. Such a message should result in four responses:

- An **RsslStatusMsg** delivered on **streamId 10** which indicates that the batch is being processed and closes the stream.
- Three **RsslRefreshMsgs** are delivered, where the first item returns on **streamId 11**, the second on **streamId 12**, and the third on **streamId 13**.

To simplify the example, some error handling has been omitted; though applications should perform all appropriate error handling.

```

/* Example assumes encode iterator is properly initialized */
/* Create and populate request message with information pertaining to all items in batch, set
/* batch flag */

RsslRequestMsg reqMsg = RSSL_INIT_REQUEST_MSG;
reqMsg.msgBase.msgClass = RSSL_MC_REQUEST;
reqMsg.msgBase.domainType = RSSL_DMT_MARKET_PRICE;
/* Set RSSL_RQMF_HAS_BATCH so provider application is alerted to batch payload */
reqMsg.flags = RSSL_RQMF_HAS_QOS | RSSL_RQMF_STREAMING | RSSL_RQMF_HAS_BATCH;
reqMsg.qos.timeliness = RSSL_QOS_TIME_REALTIME;
/* Populate msgKey - no name should be provided as all names should be in payload */
reqMsg.msgBase.msgKey.flags = RSSL_MKF_HAS_NAME_TYPE | RSSL_MKF_HAS_SERVICE_ID;
reqMsg.msgBase.msgKey.nameType = RDM_INSTRUMENT_NAME_TYPE_RIC;
reqMsg.msgBase.msgKey.serviceId = 5;
  
```

```

/* Payload type is an element list */
reqMsg.msgBase.containerType = RSSL_DT_ELEMENT_LIST;
/* Populate streamId with value to start streamId assignment */
reqMsg.msgBase.streamId = 10; /* Batch status response should be delivered using streamId 10*/
/* Begin message encoding */
 retVal = rsslEncodeMsgInit(&encIter, (RsslMsg*)&reqMsg, 0);
{
    RsslElementList elementList = RSSL_INIT_ELEMENT_LIST;
    RsslElementEntry element = RSSL_INIT_ELEMENT;
    RsslArray nameList = RSSL_INIT_ARRAY;
    elementList.flags = RSSL_ELF_HAS_STANDARD_DATA;
    /* now encode nested container using its own specific encode functions */
    retVal = rsslEncodeElementListInit(&encIter, &elementList, 0, 0);
    /* Batch requests require an element with the name of :ItemList */
    element.name.data = ":ItemList";
    element.name.length = 9;
    element.dataType = RSSL_DT_ARRAY;
    /* encode array of item names in the element entry */
    retVal = rsslEncodeElementEntryInit(&encIter, &element, 0);
{
    RsslBuffer nameBuf = RSSL_INIT_BUFFER;
    /* Encode the array and the names */
    nameList.primitiveType = RSSL_DT_ASCII_STRING;
    nameList.itemLength = 0; /* Array will have variable length entries */
    retVal = rsslEncodeArrayInit(&encIter, &nameList);
    /* Populate first name in the list. This should use streamId 11 when the response */
    /* comes */
    nameBuf.data = "TRI";
    nameBuf.length = 3;
    /* Passed in as third parameter as data is not pre-encoded */
    rsslEncodeArrayEntry(&encIter, 0, &nameBuf);
    /* Populate the second name in the list. This should use streamId 12 when the response*/
    /* comes */
    nameBuf.data = "GOOG.O";
    nameBuf.length = 6;
    rsslEncodeArrayEntry(&encIter, 0, &nameBuf);
    /* Populate the third name in the list. This should use streamId 13 when the response */
    /* comes */
    nameBuf.data = "AAPL.O";
    nameBuf.length = 6;
    rsslEncodeArrayEntry(&encIter, 0, &nameBuf);
    /* List is complete, finish encoding array */
    retVal = rsslEncodeArrayComplete(&encIter, RSSL_TRUE);
}
/* Complete the element encoding and then the element list */
retVal = rsslEncodeElementEntryComplete(&encIter, RSSL_TRUE);
retVal = rsslEncodeElementListComplete(&encIter, RSSL_TRUE);

```

```

}

/* now that :ItemList is encoded in the payload, complete the message encoding */
 retVal = rsslEncodeMsgComplete(&encIter, RSSL_TRUE);

```

Code Example 44: Batch Request Encoding Example

13.7.5 Batch Reissue Encoding Example

The following example demonstrates how to encode a batch reissue `RsslRequestMsg` to pause three streams. The request is sent using a `streamId` of **10** and contains a `:StreamIdList` of three streams, **11**, **12**, and **13**. Such a message should result in an `RsslStatusMsg` delivered on `streamId` **10** which indicates that the batch is being processed and closes the stream.

To simplify the example, some error handling has been omitted; though applications should perform all appropriate error handling.

```

/* Example assumes encode iterator is properly initialized */
/* Create and populate request message. Set batch flag */
RsslRequestMsg reqMsg = RSSL_INIT_REQUEST_MSG;
reqMsg.msgBase.msgClass = RSSL_MC_REQUEST;
reqMsg.msgBase.domainType = RSSL_DMT_MARKET_PRICE;
/* Set RSSL_RQMF_HAS_BATCH so provider application is alerted to batch payload. Set pause flag
/* to indicate that we are requesting that these items be paused, and don't request a refresh.
/* Do not request a QoS. */
reqMsg.flags = RSSL_RQMF_STREAMING | RSSL_RQMF_HAS_BATCH | RSSL_RQMF_NO_REFRESH |
    RSSL_RQMF_PAUSE;
/* MsgKey is not used. */
reqMsg.msgBase.msgKey.flags = RSSL_MKF_NONE;
/* Payload type is an element list */
reqMsg.msgBase.containerType = RSSL_DT_ELEMENT_LIST;
/* Use a currently-unused streamId. */
reqMsg.msgBase.streamId = 10; /* Batch status response should be delivered using streamId 10*/
/* Begin message encoding */
retVal = rsslEncodeMsgInit(&encIter, (RsslMsg*)&reqMsg, 0);
{
    RsslElementList elementList = RSSL_INIT_ELEMENT_LIST;
    RsslElementEntry element = RSSL_INIT_ELEMENT_ENTRY;
    RsslArray streamIdList = RSSL_INIT_ARRAY;
    elementList.flags = RSSL_ELF_HAS_STANDARD_DATA;
    /* now encode nested container using its own specific encode functions */
    retVal = rsslEncodeElementListInit(&encIter, &elementList, 0, 0);
    /* Batch reissues require an element with the name of :StreamIdList */
    element.name.data = ":StreamIdList";
    element.name.length = 13;
    element.dataType = RSSL_DT_ARRAY;
    /* encode array of streamIds in the element entry */
    retVal = rsslEncodeElementEntryInit(&encIter, &element, 0);
}

```

```

RsslInt streamId;
/* Encode the array and the streamIds */
streamIdList.primitiveType = RSSL_DT_INT;
streamIdList.itemLength = 0; /* Use the default variable-length encoding. */
retVal = rsslEncodeArrayInit(&encIter, &streamIdList);
/* Encode an entry with a streamId of 11. */
streamId = 11;
rsslEncodeArrayEntry(&encIter, 0, &streamId);
/* Encode an entry with a streamId of 12. */
streamId = 12;
rsslEncodeArrayEntry(&encIter, 0, &streamId);
/* Encode an entry with a streamId of 13. */
streamId = 13;
rsslEncodeArrayEntry(&encIter, 0, &streamId);
/* List is complete, finish encoding array */
retVal = rsslEncodeArrayComplete(&encIter, RSSL_TRUE);
}
/* Complete the element encoding and then the element list */
retVal = rsslEncodeElementEntryComplete(&encIter, RSSL_TRUE);
retVal = rsslEncodeElementListComplete(&encIter, RSSL_TRUE);
}
/* now that :StreamIdList is encoded in the payload, complete the message encoding */
retVal = rsslEncodeMsgComplete(&encIter, RSSL_TRUE);

```

Code Example 45: Batch Reissue Encoding Example

13.7.6 Batch Close Encoding Example

The following example demonstrates how to encode a batch reissue `RsslCloseMsg` to close three streams. The close message is sent using a `streamId` of **10** and contains a `:StreamIdList` of three streams, **11**, **12**, and **13**. Such a message should result in an `RsslStatusMsg` delivered on `streamId` **10** which indicates that the batch is being processed and closes the stream.

For simplicity, the following example omits some error handling; though applications should perform error handling as appropriate.

```

/* Example assumes encode iterator is properly initialized */
/* Create and populate close message, and set batch flag. */
RsslCloseMsg closeMsg = RSSL_INIT_CLOSE_MSG;
closeMsg.msgBase.domainType = RSSL_DMT_MARKET_PRICE;
/* Set RSSL_CLMF_HAS_BATCH so provider application is alerted to batch payload */
closeMsg.flags = RSSL_CLMF_HAS_BATCH;
/* Payload type is an element list */
closeMsg.msgBase.containerType = RSSL_DT_ELEMENT_LIST;
/* Use a currently-unused streamId. */
closeMsg.msgBase.streamId = 10; /* Batch status response should be delivered using streamId
    /* 10 */

```

```

/* Begin message encoding */
retVal = rsslEncodeMsgInit(&encIter, (RsslMsg*)&closeMsg, 0);
{
    RsslElementList elementList = RSSL_INIT_ELEMENT_LIST;
    RsslElementEntry element = RSSL_INIT_ELEMENT_ENTRY;
    RsslArray streamIdList = RSSL_INIT_ARRAY;
    elementList.flags = RSSL_ELF_HAS_STANDARD_DATA;
    /* now encode nested container using its own specific encode functions */
    retVal = rsslEncodeElementListInit(&encIter, &elementList, 0, 0);
    /* Batch closes require an element with the name of :StreamIdList */
    element.name.data = ":StreamIdList";
    element.name.length = 13;
    element.dataType = RSSL_DT_ARRAY;
    /* encode array of streamIds in the element entry */
    retVal = rsslEncodeElementEntryInit(&encIter, &element, 0);
    {
        RsslInt streamId;
        /* Encode the array and the streamIds */
        streamIdList.primitiveType = RSSL_DT_INT;
        streamIdList.itemLength = 0; /* Use the default variable-length encoding. */
        retVal = rsslEncodeArrayInit(&encIter, &streamIdList);
        /* Encode an entry with a streamId of 11. */
        streamId = 11;
        rsslEncodeArrayEntry(&encIter, 0, &streamId);
        /* Encode an entry with a streamId of 12. */
        streamId = 12;
        rsslEncodeArrayEntry(&encIter, 0, &streamId);
        /* Encode an entry with a streamId of 13. */
        streamId = 13;
        rsslEncodeArrayEntry(&encIter, 0, &streamId);
        /* List is complete, finish encoding array */
        retVal = rsslEncodeArrayComplete(&encIter, RSSL_TRUE);
    }
    /* Complete the element encoding and then the element list */
    retVal = rsslEncodeElementEntryComplete(&encIter, RSSL_TRUE);
    retVal = rsslEncodeElementListComplete(&encIter, RSSL_TRUE);
}
/* now that :StreamIdList is encoded in the payload, complete the message encoding */
retVal = rsslEncodeMsgComplete(&encIter, RSSL_TRUE);

```

Code Example 46: Batch Close Encoding Example

13.8 Dynamic View Use

Applications can use the Transport API to send or receive requests for a dynamic view of a stream's content. A consumer application uses a **dynamic view** to specify a subset of data in which the application has interest. A provider can choose to

supply only this requested subset of content across all response messages. Filtering content in this manner can reduce the volume of data that flows across the connection. View use can be leveraged across all non-administrative³ domain model types, where the model definition should specify associated usage and support. Though a consumer might request a specific view, the provider might still send additional content and/or content might be unavailable (and not provided).

A consumer application can request a view through an `RsslRequestMsg` with the `RSSL_RQMF_HAS_VIEW` flag set and by including a specially-formatted payload. The payload should contain an `RsslElementList` along with:

- An `RsslElementEntry` for `:ViewType` which contains an `RSSL_DT_UINT` value indicating the specific type of view requested. Section 13.8.1 describes the currently defined `:ViewType` values.
- An `RsslElementEntry` for `:ViewData` which contains an `RsslArray` populated with the content being requested. For instance, when specifying a `fieldId` list, the array would contain two-byte fixed length `RSSL_DT_INT` entries. The specific contents of the `:ViewData` are indicated in the definition of the `:ViewType`.

Because payload content can include customer-defined portions and Thomson Reuters-defined portions, the Transport API uses a name-spacing scheme. Any content in the `name` member prior to the colon (`:`) is used as name space information (e.g., `Customer:Element`). Thomson Reuters reserves the empty name space (e.g., `:Element`). View-related enumerations and element name string constants are defined in the `rssIRDIM.h` header file.

If a consumer application wishes to change a previously-specified view, the same process can be followed by issuing a subsequent `RsslRequestMsg` using the same `streamId` (a reissue). In this case, `:ViewData` would contain the newly desired view. If a reissue is required and the consumer wants to continue using the same view, the `RsslRequestMsg` should continue to include the `RSSL_RQMF_HAS_VIEW` flag; `:ViewType` or `:ViewData` are not required. Sending an `RsslRequestMsg` without the `RSSL_RQMF_HAS_VIEW` flag removes any view associated with a stream.

A provider application can receive a view request and determine an appropriate way to respond. Response content can be filtered to abide by the view specification, or the provider can send full/additional content. Several `RsslState.code` values are available to convey view-related status. If a view's possible content changes (e.g., a previously requested field becomes available), an `RsslRefreshMsg` should be provided to convey such a change to the data. This refresh should follow the rules associated with solicited or unsolicited refresh messages.

A provider application can indicate support for dynamic view handling by sending the `msgKey` attribute `supportViewRequests` in the Login domain `RsslRefreshMsg`.

- For details on `RsslState.code` values, refer to Section 11.2.6.4.
- For details on the `RsslRequestMsg` and `RSSL_RQMF_HAS_VIEW` flag values, refer to Section 12.2.1.
- For details on the `RsslElementList`, refer to Section 11.3.2.
- For rules associated with refresh messages, refer to Section 12.2.2.
- For details on the Login `domainType` (`RSSL_DMT_LOGIN`) and general view use, refer to the *Transport API RDM Usage Guide*.

³. Administrative domain types are considered to be the Login, Directory, and Dictionary domain models. Other domains are considered non-administrative.

13.8.1 RDMViewTypes Enumerated Names

ENUMERATED NAME	DESCRIPTION
RDM_VIEW_TYPE_FIELD_ID_LIST	Indicates that <code>:ViewData</code> contains an array populated with <code>fieldId</code> values. The array should specify a <code>primitiveType</code> of <code>RSSL_DT_INT</code> and a fixed two-byte <code>itemLength</code> . For specific details about the <code>Rss1Array</code> , refer to Section 11.2.7.
RDM_VIEW_TYPE_ELEMENT_NAME_LIST	Indicates that <code>:ViewData</code> contains an array populated with element <code>name</code> values. The array should specify a <code>primitiveType</code> corresponding to the type used for the domain model's element names (e.g. <code>RSSL_DT_ASCII_STRING</code>). For specific details about the <code>Rss1Array</code> , refer to Section 11.2.7.

Table 191: RDMViewTypes Values

13.8.2 Dynamic View RsslRequestMsg Encoding Example

The following example demonstrates how to encode an `RsslRequestMsg` which specifies a `fieldId`-based view. The request asks for two fields, though it is possible that more will be delivered. For the sake of simplicity, some error handling is omitted from the example; though applications should perform all appropriate error handling.

```

/* Example assumes encode iterator is properly initialized */
/* Create and populate request message, set view flag */
RsslRequestMsg reqMsg = RSSL_INIT_REQUEST_MSG;
reqMsg.msgBase.msgClass = RSSL_MC_REQUEST;
reqMsg.msgBase.domainType = RSSL_DMT_MARKET_PRICE;
/* Set RSSL_RQMF_HAS_VIEW so provider is alerted to view payload */
reqMsg.flags = RSSL_RQMF_HAS_QOS | RSSL_RQMF_STREAMING | RSSL_RQMF_HAS_VIEW;
reqMsg.msgBase.streamId = 15;
reqMsg.qos.timeliness = RSSL_QOS_TIME_REALTIME;
reqMsg.qos.rate = RSSL_QOS_RATE_TICK_BY_TICK;
/* Populate msgKey */
reqMsg.msgBase.msgKey.flags = RSSL_MKF_HAS_NAME | RSSL_MKF_HAS_NAME_TYPE |
    RSSL_MKF_HAS_SERVICE_ID;
reqMsg.msgBase.msgKey.nameType = RDM_INSTRUMENT_NAME_TYPE_RIC;
reqMsg.msgBase.msgKey.name.data = "TRI";
reqMsg.msgBase.msgKey.name.length = 3;
reqMsg.msgBase.msgKey.serviceId = 5;
/* Payload type is an element list */
reqMsg.msgBase.containerType = RSSL_DT_ELEMENT_LIST;
/* Begin message encoding */
retVal = rsslEncodeMsgInit(&encIter, (RsslMsg*)&reqMsg, 0);
{
    RsslElementList elementList = RSSL_INIT_ELEMENT_LIST;
    RsslElementEntry element = RSSL_INIT_ELEMENT;
    RsslUInt viewTypeUInt;

```

```

RsslArray fidList = RSSL_INIT_ARRAY;
elementList.flags = RSSL_ELF_HAS_STANDARD_DATA;
/* now encode nested container using its own specific encode functions */
retVal = rsslEncodeElementListInit(&encIter, &elementList, 0, 0);
/* Initial view requests require two elements, one with the name of :ViewType and the other
/* :ViewData */
element.name.data = ":ViewType";
element.name.length = 9;
element.dataType = RSSL_DT_UINT;
viewTypeUInt = RDM_VIEW_TYPE_FIELD_ID_LIST;
retVal = rsslEncodeElementEntry(&encIter, &element, &viewTypeUInt);
/* encode array of fieldIds in the element entry */
element.name.data = ":ViewData";
element.name.length = 9;
element.dataType = RSSL_DT_ARRAY;
retVal = rsslEncodeElementEntryInit(&encIter, &element, 0);
{
    RsslInt fieldIdInt = 0;
    /* Encode the array and the fieldIds. FieldId list should be fixed two byte integers */
    nameList.primitiveType = RSSL_DT_INT;
    nameList.itemLength = 2; /* Array will have fixed 2 byte length entries */
    retVal = rsslEncodeArrayInit(&encIter, &nameList);
    /* Populate first fieldId in the list. */
    /* Passed in as third parameter as data is not pre-encoded */
    fieldIdInt = 22; /* fieldId for BID */
    rsslEncodeArrayEntry(&encIter, 0, &fieldIdInt);
    /* Populate the second fieldId in the list */
    fieldIdInt = 25; /* fieldId for ASK */
    rsslEncodeArrayEntry(&encIter, 0, &fieldIdInt);
    /* List is complete, finish encoding array */
    retVal = rsslEncodeArrayComplete(&encIter, RSSL_TRUE);
}
/* Complete the element encoding and then the element list */
retVal = rsslEncodeElementEntryComplete(&encIter, RSSL_TRUE);
retVal = rsslEncodeElementListComplete(&encIter, success);
}
/* now that :ViewType and :ViewData are encoded in the payload, complete the message
/* encoding */
retVal = rsslEncodeMsgComplete(&encIter, success);

```

Code Example 47: View Request Encoding Example

13.9 Posting

The Transport API provides **posting** functionality: an easy way for OMM consumer applications to publish content to upstream components for further distribution. Posting is similar in concept to unmanaged publications or SSL Inserts, where content originates from a consuming application and flows upstream to some destination component. After arriving at the destination component, content can be incorporated into cache and republished to downstream applications with an acknowledgment issued to the posting application. Via posting, the Transport API can push content to all non-administrative⁴ domain model types, where specific usage and support should be indicated in the model definition. **RsslPostMsg** payloads can include any Transport API container type; often this is an **RsslMsg (RSSL_DT_MSG)**. When payload is an **RsslMsg**, the contained message should be populated with any contributed header and payload information. For additional information on how to encode and decode container types, refer to Section 11.3.

The Transport API offers two types of posting:

- **On-stream posting**, where you send an **RsslPostMsg** on an existing data stream, in which case posted content corresponds to the stream on which it is posted. The upstream route of an on-stream post is determined by the route of the data stream over which it is sent. On-stream posting should be directed towards the provider that sources the item. Because on-stream post messages are flowing on the stream related to the item, a **msgKey** is not required. If the content is republished by the upstream provider, the consumer should receive it on the same stream over which they posted it.
- **Off-stream posting**, where you send an **RsslPostMsg** on the **streamId** associated with the users Login. Thus a consumer application can post data, regardless of whether they have an open stream associated with the post-related item. Post messages issued on this stream must indicate the specific **domainType** and **msgKey** corresponding to the content being posted. Off-stream posting is typically routed by configuration values on the upstream components.

An **RsslPostMsg** contains **Visible Publisher Identifier (VPI)** information (contained in **RsslPostMsg.postUserInfo**), which identifies the user who posted it. **RsslPostMsg.postUserInfo** must be populated and consists of:

- **postUserId**: which should be an ID associated with the user. For example, a DACS user ID or if unavailable, a process id)
- **postUserAddr**: which should contain the IP address⁵ of the application posting the content.

Optionally, such information can be carried along with republished **RsslRefreshMsgs**, **RsslUpdateMsgs**, or **RsslStatusMsgs** so that receiving consumers can identify the posting user. For more information about VPI, refer to Section 13.10.

RsslPostMsg.permData permissions the user who posts data. If the payload of the **RsslPostMsg** is another nested message type (i.e., **RsslRefreshMsg**) with permission data, such permission data can change the permission expression of the item being posted. However, if the permission data for the nested message is the same as the permission data on the **RsslPostMsg**, the nested message does not need to include permission data. The permission data is used in conjunction with the **RsslPostMsg.postUserRights**, which indicate:

- Whether the posting user can create or destroy items in the cache of record.
- Whether the user has the ability to change the **permData** associated with an item in the cache of record.

Each independent post message flowing in a stream should use a unique **postId** to distinguish between individual post messages and those used for acknowledgment purposes. The consumer can request an acknowledgment upon the successful receipt and processing of content. When the provider responds, the **RsslAckMsg.ackId** should be populated using the **RsslPostMsg.postId** to match the two messages. **seqNum** information can also be used during acknowledgment.

Note: Provider applications that support posting must have the ability to properly acknowledge posted content.

4. Administrative domain types are considered to be the Login, Directory, and Dictionary domain models. Other domains are considered non-administrative.

5. The **rssl HostByName** function can be used to help obtain the IP address of the application. Refer to Section 10.14.

You can split content across multiple `RsslPostMsg` messages. When sending a multi-part `RsslPostMsg`, the `postId` should match all parts of the post. If the consumer requests an acknowledgment, the `seqNum` is also required. Each part should be acknowledged by the receiving component, where each `RsslAckMsg.ackId` is populated using the `RsslPostMsg.postId`, and each `RsslAckMsg.seqNum` is populated using the `RsslPostMsg.seqNum`. Each part of the `RsslPostMsg` should specify a `partNum`, where the first part begins with 0. The final part of a multi-part `RsslPostMsg` should have the `RSSL_PSMF_POST_COMPLETE` flag set to indicate that it is the final part.

A provider application can indicate support for posting and acknowledgment use by sending the `msgKey` attribute `supportOmmPost` in the Login domain `RsslRefreshMsg`.

- For more information on the `RsslPostMsg`, refer to Section 12.2.7.
- For more information on the `RsslAckMsg`, refer to Section 12.2.8.
- For more information on managing multi-part `RsslPostMsgs`, refer to Section 13.1.
- For more details on the Login `domainType` (`RSSL_DMT_LOGIN`), see the *Transport API RDM Usage Guide*.

13.9.1 Post Message Encoding Example

The following example demonstrates how to encode an off-stream `RsslPostMsg` with a nested `RsslMsg`.

```
/* Example assumes encode iterator is properly initialized */
/* Create and populate post message - since it's off stream, msgKey is required */
RsslPostMsg postMsg = RSSL_INIT_POST_MSG;
postMsg.msgBase.msgClass = RSSL_MC_POST;
postMsg.msgBase.streamId = 1; /* Use streamId of the Login stream for off-stream posting */
postMsg.msgBase.domainType = RSSL_DMT_MARKET_PRICE; /* domainType of data being posted */
/* off stream requires key. Post asking for ACK and including postId and seqNum for ack
/* purposes. Since it's a single part post, the POST_COMPLETE flag must be set as well */
postMsg.flags = RSSL_PSMF_HAS_MSG_KEY | RSSL_PSMF_ACK | RSSL_PSMF_HAS_POST_ID |
    RSSL_PSMF_HAS_SEQ_NUM | RSSL_PSMF_POST_COMPLETE;
/* Populate msgKey with information about the item being posted to */
postMsg.msgBase.msgKey.flags = RSSL_MKF_HAS_NAME | RSSL_MKF_HAS_NAME_TYPE | |
    RSSL_MKF_HAS_SERVICE_ID;
postMsg.msgBase.msgKey.nameType = RDM_INSTRUMENT_NAME_TYPE_RIC;
postMsg.msgBase.msgKey.name.data = "TRI";
postMsg.msgBase.msgKey.name.length = 3;
postMsg.msgBase.msgKey.serviceId = 5;
/* populate postId with a unique ID for this posting, this and seqNum are used on ack */
postMsg.postId = 42;
postMsg.seqNum = 124;
/* postUserInfo must be populated, with processId and IP address */
postMsg.userInfo.postUserId = getpid();
/* Use RSSL Transport Helper function - refer to Section 10.14 */
/* example assumes hostNameBuf.data contains hostname and hostNameBuf.length indicates length
/* of hostname */
rsslHostByName(&hostNameBuf, &ipAddrUInt32);
postMsg.userInfo.postUserAddr = ipAddrUInt32;
/* put a message in the postMsg */
postMsg.containerType = RSSL_DT_MSG;
```

```

/* Begin message encoding */
 retVal = rsslEncodeMsgInit(&encIter, (RsslMsg*)&postMsg, 0);
{
    /* populate the message that is in the payload of the post message */
    RsslUpdateMsg updMsg = RSSL_INIT_UPDATE_MSG;
    updMsg.msgBase.msgClass = RSSL_MC_UPDATE;
    updMsg.msgBase.streamId = 1;
    updMsg.msgBase.domainType = RSSL_DMT_MARKET_PRICE;
    updMsg.flags = RSSL_UPMF_NONE;
    updMsg.updateType = RDM_UPD_EVENT_TYPE_QUOTE;
    updMsg.containerType = RSSL_DT_FIELD_LIST;
    /* begin encoding of the payload message */
    retVal = rsslEncodeMsgInit(&encIter, (RsslMsg*)&updMsg, 0);
    /* Continue encoding field list contents of the message - see example in Section 11.3.1.6
     */
    /* Complete the postMsg payload messages encoding */
    retVal = rsslEncodeMsgComplete(&encIter, RSSL_TRUE);
}
/* now complete encoding of postMsg */
retVal = rsslEncodeMsgComplete(&encIter, success);

```

Code Example 48: Off-Stream Posting Encoding Example

13.9.2 Post Acknowledgement Encoding Example

The following example demonstrates how to encode an [RsslAckMsg](#).

```

/* Example assumes encode iterator is properly initialized */
/* Create and populate ack message with information used to acknowledge the post */
RsslAckMsg ackMsg = RSSL_INIT_ACK_MSG;
ackMsg.msgBase.msgClass = RSSL_MC_ACK;
ackMsg.msgBase.domainType = RSSL_DMT_MARKET_PRICE;
ackMsg.streamId = 1; /* Ack should be sent back on same stream that post came on */
ackMsg.flags = RSSL_AKMF_HAS_SEQ_NUM;
/* Acknowledge the post from above, use its postId and seqNum */
ackMsg.ackId = postMsg.postId;
ackMsg.seqNum = postMsg.seqNum;
/* No payload associated with this acknowledgment */
ackMsg.containerType = RSSL_DT_NO_DATA;
/* Since there is no payload, no need for Init/Complete as everything is in the msg header */
retVal = rsslEncodeMsg(&encIter, (RsslMsg*)&ackMsg);

```

Code Example 49: Post Acknowledgment Encoding Example

13.10 Visible Publisher Identifier (VPI)

The Transport API offers the ***Visible Publisher Identifier (VPI)*** feature, which inserts originating publisher information into both RSSL and SSL message payloads. You can use VPI to identify the user ID and user address for users who post, insert, or publish to an interactive service or to a non-interactive service cache on the ADH.

VPI is present on Post, Refresh, Update, and Status Messages and is carried in `RsslPostMsg.postUserInfo`, which consists of:

- Post user ID (i.e., publisher ID)
- Post user address (i.e., publisher address)

They can both contain values assigned by and specific to the application.

An `RsslPostMsg` contains data (in `RsslPostMsg.userInfo`) that identifies the user who posts content. For this reason, `RsslPostMsg.userInfo` must be populated with a:

- `postUserId`: An ID associated with the posting user. The application should determine what information to put into this field (e.g., a DACS user ID).
- `postUserAddr` The address of the posting user's application that posted the contents. The application should decide what information to put into this field (e.g., an IP address⁶).

Optionally, this data can be republished by the provider in a `RsslRefreshMsgs`, `RsslUpdateMsgs`, or `RsslStatusMsgs` so that receiving consumers can identify the posting user.

The Transport API allows the VPI to be populated on Post messages submitted by an OMM Consumer application before the post is sent over the network.

Provider applications receive VPI in Post Messages. Additionally, OMM providers can optionally set VPI in their response messages. If the upstream provider is an intermediary device getting data from an upstream source, then the intermediary device will route the VPI as set in the `RsslPostMsg` to the upstream source. The final publisher in the upward chain decides whether to set the VPI in its published responses.

VPI information can also be communicated using FIDs defined in the publisher component. For further details refer to the publishing component's documentation.

6. You can use the `rssl HostByName` function to help obtain the IP address of the application. Refer to Section 10.14.

13.10.1 VPI Example: Using RsslPostUserInfo to Obtain VPI Data

The following example shows the **rsslConsumer** application using **RsslPostUserInfo** to obtain VPI data in the **processMarketPriceResponse()** function:

```
/* The Visible Publisher Identifier (VPI) can be found within the RsslPostUserInfo.
/* This will provide both the publisher ID and publisher address. Consumer can obtain the
/* information from the msg - The partially decoded message. */

if (msg->refreshMsg.flags & RSSL_RFMF_HAS_POST_USER_INFO)
{
    rsslIPAddrUIString(msg->refreshMsg.postUserInfo.postUserAddr, postUserAddrString);
    printf("\nReceived RefreshMsg for stream %i ", msg->refreshMsg.msgBase.streamId);
    printf("from publisher with user ID: \"%u\" at user address: \"%s\"\n",
        msg->refreshMsg.postUserInfo.postUserId, postUserAddrString);
}
```

Code Example 50: Consumer Using **RsslPostUserInfo to Obtain VPI Information**

13.10.2 VPI Example: Populating VPI in Post Messages from Consumer Applications

The following example populates VPI on the Post messages submitted by a Transport API OMM consumer application in the **encodePostWithMsg()** function internally used by the **sendPostMsg()** function. It encodes a PostMsg and populates the **PsslPostUserInfo** with the IP address and process ID of the machine running the application.

```
// Note: post message key not required for on-stream post
postMsg.flags = RSSL_PSMF_POST_COMPLETE
| RSSL_PSMF_ACK // request ACK
| RSSL_PSMF_HAS_POST_ID
| RSSL_PSMF_HAS_SEQ_NUM
| RSSL_PSMF_HAS_POST_USER_RIGHTS
| RSSL_PSMF_HAS_MSG_KEY;

postMsg.postId = nextPostId++;
postMsg.seqNum = nextSeqNum++;
postMsg.postUserRights = RSSL_PSUR_CREATE | RSSL_PSUR_DELETE;

/* populate post user info */
hostName.data = hostNameBuf;
hostName.length = 256;
gethostname(hostName.data, hostName.length);
hostName.length = (RsslUIInt32)strlen(hostName.data);
if ((ret = rsslHostByName(&hostName, &postMsg.postUserInfo.postUserAddr)) <
    RSSL_RET_SUCCESS)
{
    printf("Populating postUserInfo failed. Error %s (%d) with rsslHostByName: %s\n",
        hostName.data, ret, hostNameBuf);
```

```

    rsslRetCodeToString(ret), ret, rsslRetCodeInfo(ret));
return ret;
}
postMsg.postUserInfo.postUserId = getpid();

```

Code Example 51: Populating VPI on Post Messages Submitted by Transport API OMM Consumer Application

13.10.3 VPI Example: Getting VPI from Post Messages

```

/* The Visible Publisher Identifier (VPI) can be found within the RsslPostUserInfo.
/* This will provide both the publisher ID and publisher address. Providers may define this
/* when publishing from the postMsg. */
rsslIPAddrUIntToString(postMsg->postUserInfo.postUserAddr, postUserAddrString);
printf(" from client with publisher user ID: \"%u\" at user address: \"%s\"\n\n",
    postMsg->postUserInfo.postUserId, postUserAddrString);

// if the post message contains another message, then use the "contained" message as the
// update/refresh/status
if (postMsg->msgBase.containerType == RSSL_DT_MSG)
{
    rsslClearMsg(&nestedMsg);
    rsslDecodeMsg(dIter, &nestedMsg);
    switch(nestedMsg.msgBase.msgClass)
    {
        case RSSL_MC_REFRESH:
            nestedMsg.refreshMsg.postUserInfo = postMsg->postUserInfo;
            nestedMsg.refreshMsg.flags |= RSSL_RFMF_HAS_POST_USER_INFO;
            if (updateItemInfoFromPost(itemInfo, &nestedMsg, dIter, &error) != RSSL_RET_SUCCESS)
            {
                if (sendAck(chnl, postMsg, RSSL_NAKC_INVALID_CONTENT, error.text) !=
                    RSSL_RET_SUCCESS) return RSSL_RET_FAILURE;
                return RSSL_RET_SUCCESS;
            }
            break;

        case RSSL_MC_UPDATE:
            nestedMsg.updateMsg.postUserInfo = postMsg->postUserInfo;
            nestedMsg.updateMsg.flags |= RSSL_UPMF_HAS_POST_USER_INFO;
            if (updateItemInfoFromPost(itemInfo, &nestedMsg, dIter, &error) != RSSL_RET_SUCCESS)
            {
                if (sendAck(chnl, postMsg, RSSL_NAKC_INVALID_CONTENT, error.text) !=
                    RSSL_RET_SUCCESS) return RSSL_RET_FAILURE;
                return RSSL_RET_SUCCESS;
            }
            break;
    }
}

```

```

case RSSL_MC_STATUS:
    nestedMsg.statusMsg.postUserInfo = postMsg->postUserInfo;
    nestedMsg.statusMsg.flags |= RSSL_STMF_HAS_POST_USER_INFO;
    if ((nestedMsg.statusMsg.flags & RSSL_STMF_HAS_STATE) != 0 &&
        nestedMsg.statusMsg.state.streamState == RSSL_STREAM_CLOSED)
    {
        // check if the user has the rights to send a post that closes an item
        if ((postMsg->flags & RSSL_PSMF_HAS_POST_USER_RIGHTS) == 0 ||
            (postMsg->postUserRights & RSSL_PSUR_DELETE) == 0)
        {
            errText = (char *)"client has insufficient rights to close/delete an item";
            if (sendAck(chnl, postMsg, RSSL_NAKC_INVALID_CONTENT, errText) != RSSL_RET_SUCCESS)
                return RSSL_RET_FAILURE;
            printf(errText);
            return RSSL_RET_SUCCESS;
        }
    }
    break;
}
}

```

Code Example 52: Getting VPI from Post Messages and Setting VPI on Response Messages

13.11 TREP Authentication

The Transport API can use the TREP Authentication feature, which provides enhanced authentication functionality when used with TREP and DACS. This feature requires TREP 3.1 or later.

A consumer or non-interactive provider application can pass a token generated from a token generator based on the user's credentials to TREP. TREP passes this token to a local token authenticator for verification.

The token must be encoded in the initial login `Rss1RequestMsg` with:

- `msgKey.Name` set to one byte of `0x00`, and
- `msgKey.NameType` set to `RDM_LOGIN_USER_AUTHN_TOKEN`.

The token will be in the `msgKey.attrib`'s `Rss1ElementList`, with an `Rss1ElementEntry` named `authenticationToken`.

For additional information, refer to the *Transport API RDM Usage Guide* for encoding and decoding Login messages, and the *TREP Authentication User Manual*⁷ for details on setting up TREP and the token generator.

7. For further details on TREP Authentication, refer to the *TREP Authentication User Manual*, accessible on [Thomson Reuters MyAccount](#) in the DACS product documentation set.

13.12 Private Streams

The Transport API provides ***private stream*** functionality, an easy way to ensure delivery of content only between a stream's two endpoints. Private streams behave in a manner similar to standard streams, with the following exceptions:

- All data on a private stream flow between the end provider and the end consumer of the stream.
- Intermediate components do not fan out content (i.e., do not distribute it to other consumers).
- Intermediate components should not cache content.
- In the event of connection or data loss, intermediate components do not recover content. All private stream recovery is the responsibility of the consumer application.

These behaviors ensure that only the two endpoints of the private stream send or receive content associated with the stream. As a result, a private stream can exchange identifying information so the provider can validate the consumer, even through multiple intermediate components (such as might exist in a TREP deployment). After a private stream is established, content can flow freely within the stream, following either existing market data semantics (i.e., private Market Price domain) or any other user-defined semantics (i.e., bidirectional exchange of **Rss1GenericMsgS**).

In standard streams, if an application attempts to open the same stream using multiple, unique **streamId** values, provider applications reject subsequent requests. With private streams, even if the streams' identifying information (**msgKey**, domain type, etc.) matches, multiple private stream instances can be opened, allowing for the possibility of different user data contained in each private stream.

To establish a private stream, an OMM consumer observes the following general process:

- The OMM consumer application issues a request for the item data it wants on a private stream. This **Rss1RequestMsg** should include the **RSSL_RQMF_PRIVATE_STREAM** flag. If user-identifying information is required, it should be described in the respective domain message model definition.
- When a capable OMM provider application receives a request for a private stream, if it can honor the request, the provider application should acknowledge that the stream is established and is private by sending:
 - **Rss1RefreshMsg** with the **RSSL_RFMF_PRIVATE_STREAM** flag; typically sent when there is immediate content to provide in the response.
 - **Rss1StatusMsg** with the **RSSL_STMF_PRIVATE_STREAM** flag; typically sent when there is no immediate content to provide in the response but the provider wants to acknowledge the establishment of the private stream.
 - **Rss1AckMsg** with the **RSSL_AKMF_PRIVATE_STREAM** flag; can be used as an alternative to the **Rss1StatusMsg**.
- When the consumer application receives the above acknowledgment, the private stream is established and content can be exchanged. The **PRIVATE_STREAM** flag is no longer required on any messages exchanged within the stream.
- If the consumer application receives any other message, or the above messages without their respective **PRIVATE_STREAM** flag, the private stream is not established and the consumer should close the stream if it does not want to consume a standard stream.

Some content might be available as both standard stream and private stream delivery mechanisms. In the standard stream case, all users see the same stream content. Because private streams can support user identification, each private stream instance can contain modified or additional content tailored for the specific user.

Some content might be available only as standard streams, in which case the private stream request is ignored or rejected by sending an **Rss1StatusMsg** with a **streamState** of **RSSL_STREAM_CLOSED** or **RSSL_STREAM_CLOSED_RECOVER**, or by responding to the request with a standard stream (e.g., no **PRIVATE_STREAM** flag).

Some content might be available only as a private stream (e.g., some kind of restricted data set where users must be validated). If an OMM provider has private-only content, the provider can indicate to downstream applications that its content is private by redirecting standard stream requests.

If a standard stream `RsslRequestMsg` is received for private-only content, a provider can:

- Inform downstream applications that its content is private by sending a message (including the `msgKey`), with a `streamState` of `RSSL_STREAM_REDIRECTED` in an:
 - `RsslStatusMsg` including the `RSSL_STMF_PRIVATE_STREAM` flag; typically sent when there is not any content to provide as part of the redirect.
 - `RsslRefreshMsg` including the `RSSL_RFMF_PRIVATE_STREAM` flag; typically sent when there is some kind of content to provide as part of the redirect.
- If the consumer application sees a `streamState` of `RSSL_STREAM_REDIRECTED` and a `PRIVATE_STREAM` flag, it can issue a new `RsslRequestMsg` and use the `RSSL_RQMF_PRIVATE_STREAM` flag. This process follows standard stream redirect logic and the private stream establishment protocol described above.

Appendix A Item and Group State Decision Table

The following table describes various item and group status combinations and the common results in terms of application behavior. Though applications are not required to follow this behavior, the information is provided as an example of one possible behavior.

- For general information about [Rss1State](#), refer to Section 11.2.6.
- For general information about Item Groups, refer to Section 13.4.
- For information about group status delivery and formatting, refer to the *Transport API RDM Usage Guide*.
- For information about how item state is conveyed, refer to Section 12.2.2 and Section 12.2.4.

STATUS TYPE	STREAM STATE	DATA STATE	DESCRIPTION	APPLICATION ACTION
Item	RSSL_STREAM_OPEN	RSSL_DATA_OK	Stream is open and streaming. Data is ok.	No action.
Item	RSSL_STREAM_OPEN	RSSL_DATA_SUSPECT	Stream is open and streaming. Data is suspect.	No action. Upstream device should recover data and onpass.
Item	RSSL_STREAM_NON_STREAMING	RSSL_DATA_OK	Stream was opened as non-streaming. Data was provided for item and was OK.	No action.
Item	RSSL_STREAM_CLOSED	RSSL_DATA_SUSPECT	Stream is closed. Data is suspect.	Application can attempt to recover this or another service or provider.
Item	RSSL_STREAM_CLOSED_RECOVER	RSSL_DATA_SUSPECT	Stream is closed, but may become available on same service and provider later. Data is suspect.	Application can attempt to recover to this or another service or provider.
Item	RSSL_STREAM_CLOSED	RSSL_DATA_OK	Stream is closed. Data provided was OK.	Application can attempt to recover to this or another service or provider. This state combination is not common.

Table 192: Item and Group State Decision Table

Status Type	Stream State	Data State	Description	Application Action
Item	RSSL_STREAM_CLOSED_RECOVER	RSSL_DATA_OK	Stream is closed, but may become available on same service and provider later. Data provided was OK.	Application can attempt to recover to this or another service or provider. This state combination is not common.
Group	RSSL_STREAM_OPEN	RSSL_DATA_NO_CHANGE	All streams associated with the group remain open. Previous state communicated via item or group status continues to apply.	No action.
Group	RSSL_STREAM_OPEN	RSSL_DATA_SUSPECT	All streams associated with the group remain open. Data on all streams associated with the group is suspect.	Application should fan out dataState change to all items that are part of the group. Upstream device should recover data and onpass.
Group	RSSL_STREAM_OPEN	RSSL_DATA_OK	All streams associated with the group remain open. Data on all streams associated with the group is ok.	Application should fan out dataState change to all items that are part of the group. This state combination is not common. Typically individual item statuses are used to change items from suspect to ok.
Group	RSSL_STREAM_CLOSED_RECOVER	RSSL_DATA_SUSPECT	All streams associated with the group are closed, but may become available on same service and provider later. Data on all streams associated with the group is suspect.	Application should fan out streamState and dataState change to all items that are part of the group. Application can attempt to recover to this or another service or provider.

Table 192: Item and Group State Decision Table (Continued)

Appendix B Error Codes

The following table describes the various error codes returned through the use of an `RsslError` structure in the Transport Package, the meaning of the code, and the issue's recommended resolution.

For general information about `RsslError`, refer to Section 10.1.4.

Note: These return codes are provided as additional information for the purposes of debugging or logging. The user should rely on the return codes provided by the transport functions to respond programmatically to these errors.

ERROR CODE	DESCRIPTION	RECOMMENDED SOLUTION
0001	The Transport API has not been initialized.	Ensure that <code>rsslInitialize</code> has been called before any other function in the Transport Package.
0002	One or more arguments passed into this function were incorrectly passed in as NULL.	Verify that the argument indicated in the error string was properly allocated and set.
0003	Codec error in multicast transport.	This can indicate either: <ul style="list-style-type: none">An incorrectly formatted or corrupt message was passed into the transport, orData corruption (if reading).
0004	Cannot change mutex locking type	After <code>rsslInitialize</code> has been called, it is not possible to change the transport's locking strategy.
0005	Memory allocation error. The function is attempting to allocate new memory, but that attempt has failed.	Ensure that the application is not attempting to allocate more memory than required.
0006	The connection type is not supported.	Ensure that the proper configuration is used for <code>rsslBind</code> or <code>rsslConnect</code> .
0007	The channel state is incorrect for this operation.	The function that returns this error can only be called if the <code>RsslChannel</code> passed in is in a certain state, as noted by the error. Ensure that <code>rsslInitChannel</code> has been called on the channel, and that the channel's state matches the state listed in the error text.
0008	Buffer overflow. The buffer from the Transport Package is fully overwritten, and data might be corrupt.	Check encoding and decoding calls to ensure there is not a buffer overflow condition. If copying data into the transport buffer, ensure that the length of the copied data is less than the transport buffer's length.
0009	Buffer of length zero cannot be written by the transport.	Do not call <code>rsslWrite</code> with a buffer of length 0.
0010	Invalid buffer size specified.	Ensure that the requested buffer size is larger than zero.
0011	Buffer cannot be released due to integrity issues.	Data is corrupt. Ensure that the buffer received from <code>rsslGetBuffer</code> was not modified (except for its length).

Table 193: Error Codes

ERROR CODE	DESCRIPTION	RECOMMENDED SOLUTION
0012	Configuration error.	Follow the recommendation in the error message.
0013	Missing configuration.	Populate the missing configuration element before making the function call.
0014	Fragmentation error.	This may indicate network issues, or corrupt data.
0015	Cannot obtain a packed, fragmented buffer.	Buffers larger than the maximum fragment size on a connection cannot be packed. When requesting a packed buffer, ensure that the size of the requested buffer is smaller than the maximum fragment size.
0016	Indicates there is an issue with allocating or obtaining a buffer.	Ensure that the application is not using more memory than necessary.
0017	Invalid IOCTL code.	Ensure that the proper code or value is passed into the function.
0018	Channel does not own this buffer.	Ensure that the channel used for <code>rsslGetBuffer</code> and the buffer received from <code>rsslGetBuffer</code> are correctly passed into the function call.
0019	Corrupt or incomplete data.	This error indicates that the requested operation cannot be completed, and the data might have quality issues.
0020	Buffer too small.	The requested operation needs a larger buffer (or additional buffers) to complete. Ensure that enough data has been allocated to the buffer(s) to complete the operation.
1000	One or more arguments passed into this function were incorrectly passed in as NULL.	Check that the argument indicated in the error string has been properly allocated and set.
1001	Memory allocation error. The function is failing to allocate new memory.	Ensure that the application is not attempting to allocate more memory than is required.
1002	System error. The system call has returned an error.	The system errno in the error text corresponds to the errno value populated by the system after the call. This errno can be different depending on the underlying OS. Check the OS documentation for more information about the specific return code.
1003	The function failed because the channel is shutting down.	This might indicate a potential race condition in a multithreaded application. To avoid this error, do not make Transport Package calls after calling <code>rsslCloseChannel</code> .
1004	IOCtl configuration error.	Ensure that the configuration passing into <code>rsslIoctl</code> is correct according to the error text.
1005	Internal error	If this error code is received, contact support with the error text and any information regarding the circumstances behind it.

Table 193: Error Codes (Continued)

ERROR CODE	DESCRIPTION	RECOMMENDED SOLUTION
1006	Incoming connection has been rejected.	This may not be an error, depending on the text. The channel associated with this error should be closed in any case.
1007	Issue with the transport header has resulted in an error.	This might be a result of client or server misconfiguration of a TCP connection or misconfiguration of a UDP multicast peer.
1008	Internal error.	If you receive this error code, contact support with the error text and any information regarding the circumstances behind it.
1009	Out of output buffers.	Attempt to flush the channel before attempting to write again.

Table 193: Error Codes (Continued)

Appendix C Document Revision History

Document Version	Revision List
1.0	<ul style="list-style-type: none">• Rebranded product as the Elektron Transport API.• For document revisions prior to UPA being rebranded as the Transport API, refer to the UPA Developers Guide for release 7.6.1.• Added content for Sequenced Multicast connections.

Table 194: Transport API C Edition Document Revision History

© 2015 - 2017 Thomson Reuters. All rights reserved.

Republication or redistribution of Thomson Reuters content, including by framing or similar means, is prohibited without the prior written consent of Thomson Reuters. 'Thomson Reuters' and the Thomson Reuters logo are registered trademarks and trademarks of Thomson Reuters and its affiliated companies.

Any third party names or marks are the trademarks or registered trademarks of the relevant third party.

Document ID: ETAC310UM.170
Date of issue: 28 April 2017

